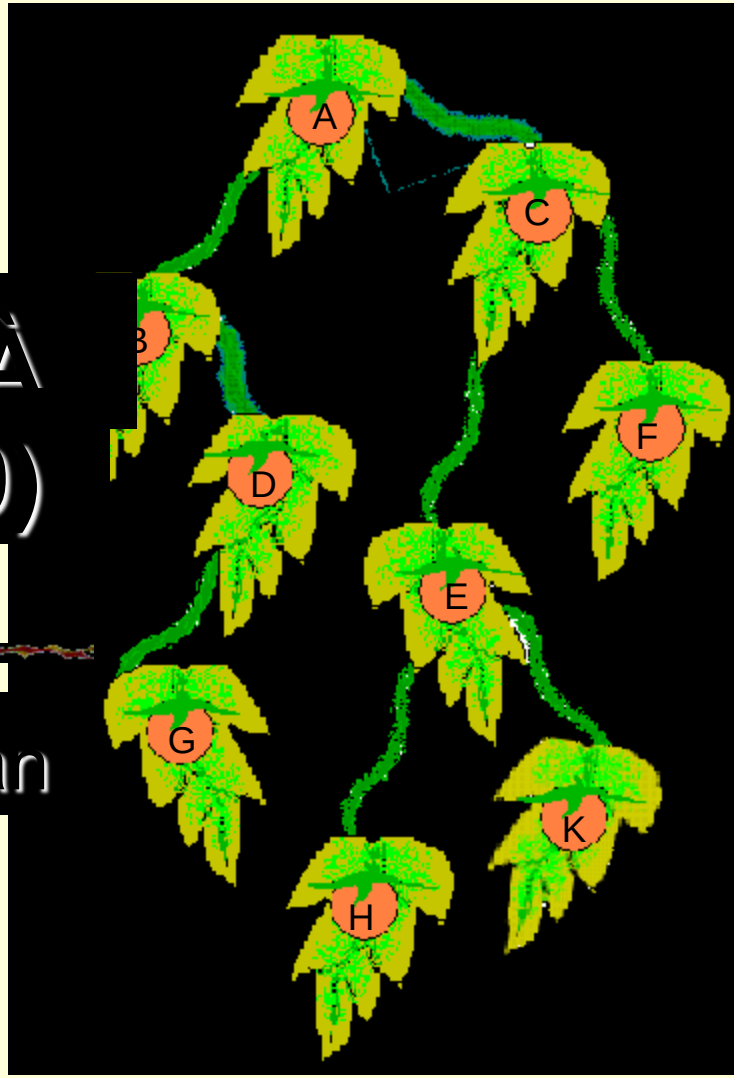
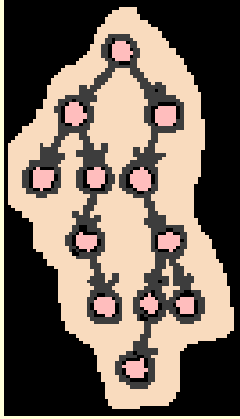
CẤU TRÚC DỮ LIỆU VÀ GIẢI THUẬT (501040)

Chương 1: Tổng quan

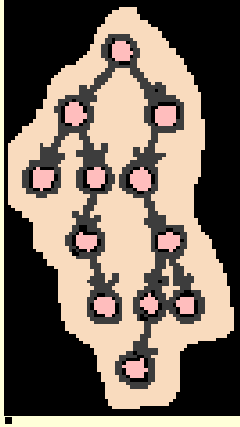


Giải bài toán bằng phần mềm

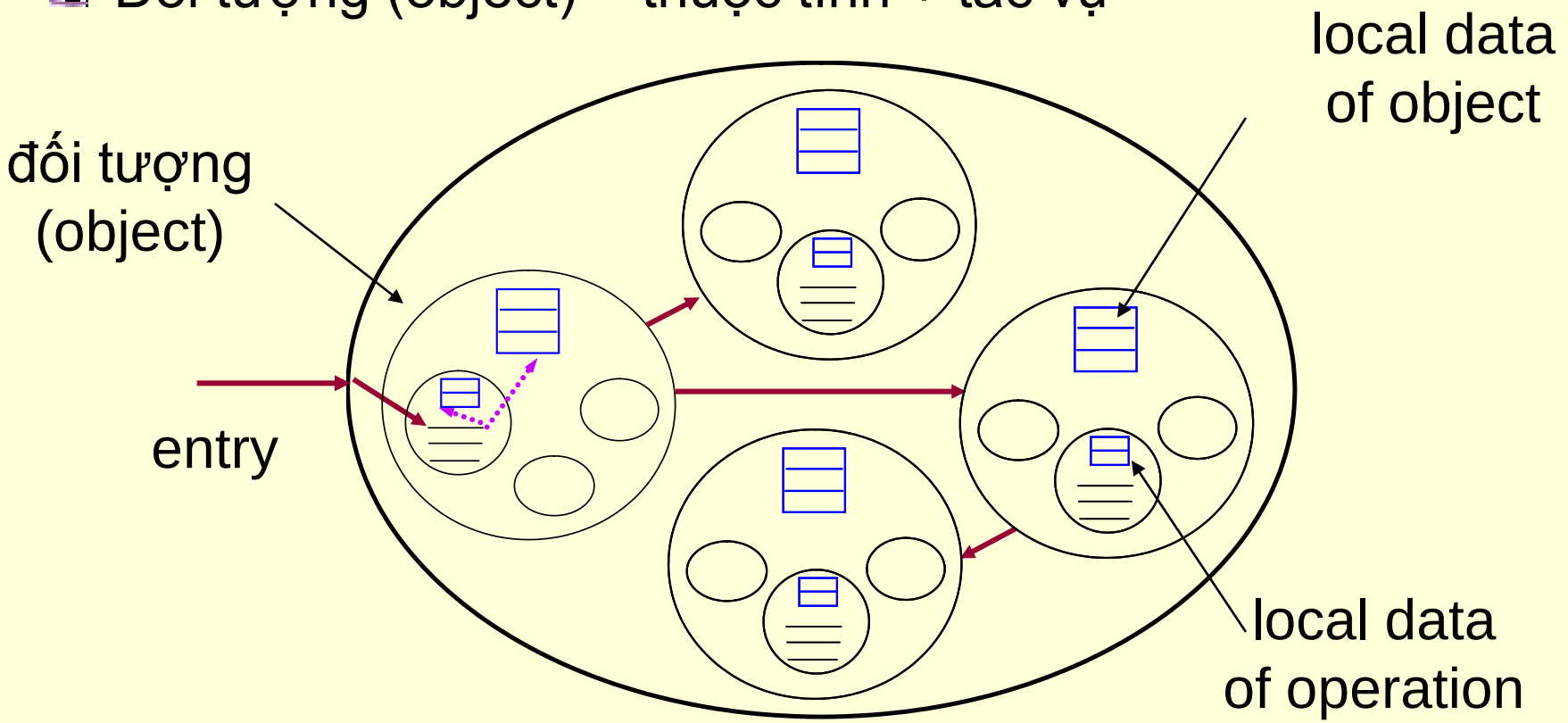


- 1. Xác định bài toán
- 2. Thiết kế phần mềm
- 3. Thiết kế dữ liệu
- 4. Thiết kế và phân tích giải thuật
- 5. Lập trình và gỡ rối
- 6. Kiểm tra phần mềm
- 7. Bảo trì

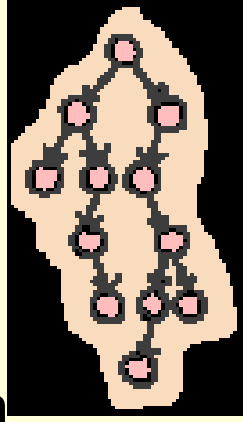
Lập trình hướng đối tượng (OOP)



- Chương trình = tập các đối tượng tương tác nhau.
- Đối tượng (object) = thuộc tính + tác vụ

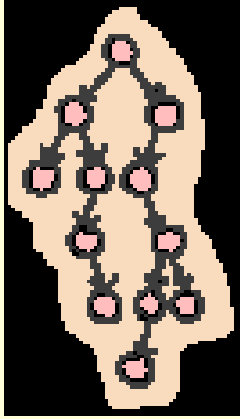


Kiểu trừu tượng



- Kiểu trừu tượng (abstract type): định nghĩa interface (tập các entry)
- Entry
  - Tên method
  - Danh sách tham số hình thức
  - Đặc tả chức năng
- Chưa có dữ liệu bên trong, chưa dùng được
- Chỉ dùng để thiết kế ý niệm

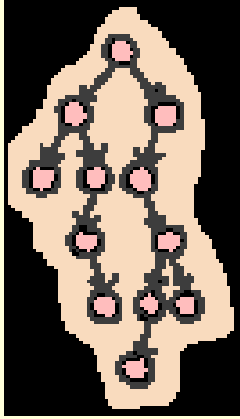
Hiện thực và sử dụng



- Class: hiện thực của abstract type
  - Định nghĩa các dữ liệu
  - Định nghĩa các phương thức + hàm phụ trợ (nội bộ)
  - Định nghĩa các phương phức ‘constructor’ và ‘destructor’ nếu cần
- Đối tượng = một instance của một class
- Thông điệp (message):
  - dùng tương tác lẫn nhau = lời gọi phương thức của các đối tượng

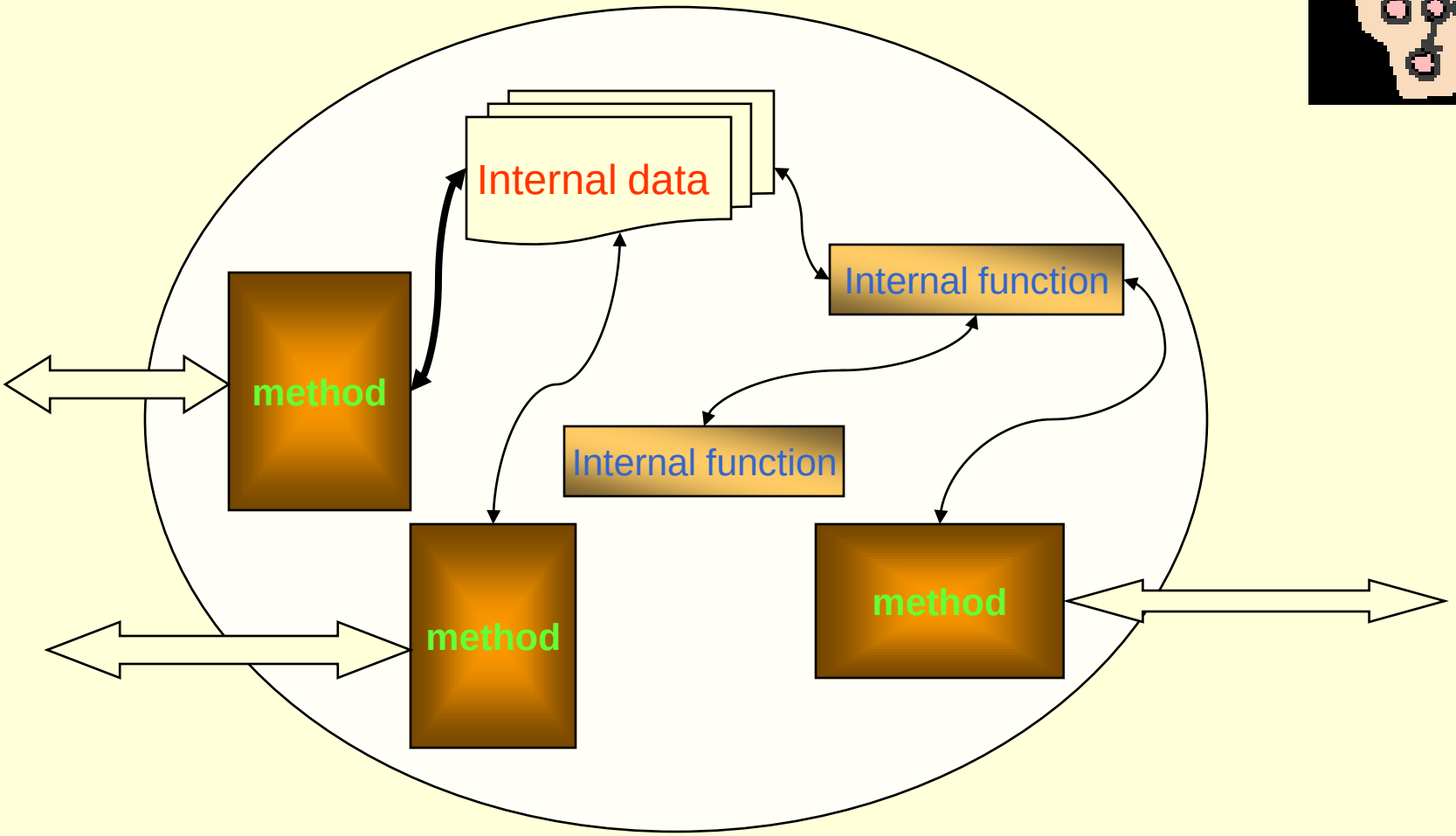
```
Student   aStudent;
aStudent.print();
```

Đặc điểm của OOP

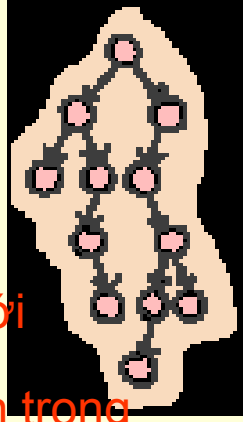


- Tính bao đóng:
  - Che dấu cấu trúc dữ liệu bên trong.
  - Che dấu cách thức hiện thực đối tượng.
- Kế thừa:
  - Định nghĩa thêm các dữ liệu và phương thức cần thiết từ một class có sẵn.
  - Cho phép overwrite/overload.
  - Cho phép dùng thay thế và khả năng dynamic biding.
- Bao gộp:
  - Một đối tượng chứa nhiều đối tượng khác.

Cấu trúc của đối tượng



Khai báo một class trên C++



```
class Student {
private:
    int StudentID;
    string StudentName;
public:
    Student();
    Student(const Student &)
    ~Student()
    void operator=(const Student &)
    void print();
};

void main() {
    Student aStudent;
    sStudent.print();
}
```

khai báo một lớp mới

khai báo dữ liệu bên trong

constructor

copy constructor

destructor

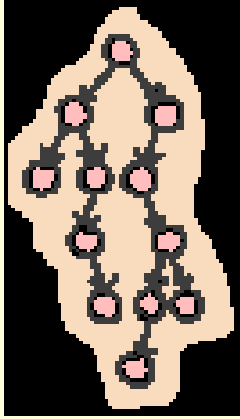
overload assignment operator

phương thức (hành vi)

khai báo một đối tượng

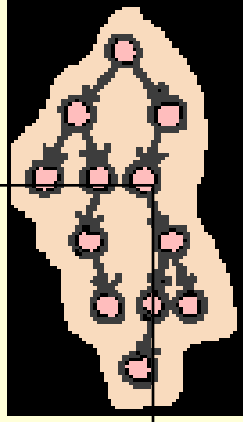
gọi phương thức

Dùng ghi chú làm rõ nghĩa



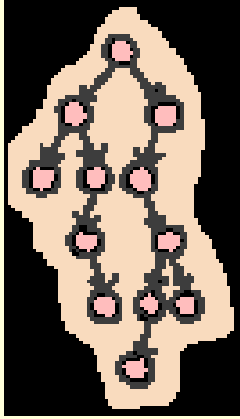
- 1. Ghi chú vào đầu mỗi hàm
  - (a) Người lập trình, ngày, bản sao
  - (b) Mục đích của hàm
  - (c) Input, output
  - (d) Các chỉ dẫn đến các tài liệu khác (nếu có)
  - Có thể dùng dạng: Precondition và Postcondition
- 2. Ghi chú vào mỗi biến, hằng, kiểu
- 3. Ghi chú vào mỗi phần của chương trình
- 4. Ghi chú mỗi khi dùng các kỹ thuật đặc biệt

Dùng ghi chú làm rõ nghĩa – Ví dụ



```
void Life::update()
/* Pre: grid đang chứa một trạng thái của thực thể sống
   Post: grid sẽ chứa trạng thái tiến hóa mới của thực thể sống này */
{
    int row, col;
    int new_grid[maxrow + 2][maxcol + 2];       //Chứa trạng thái mới vào đây
    for (row = 1; row <= maxrow; row++)
        for (col = 1; col <= maxcol; col++)
            switch (neighbor_count(row, col)) {
                case 2:                                 //Trạng thái của tế bào không đổi
                    new_grid[row][col] = grid[row][col]; break;
                case 3:                                 //Tế bào sẽ sống
                    new_grid[row][col] = 1; break;
                default:                                //Tế bào sẽ chết
                    new_grid[row][col] = 0;
             }
    for (row = 1; row <= maxrow; row++)
        for (col = 1; col <= maxcol; col++)
            grid[row][col] = new_grid[row][col];    //Cập nhật các tế bào cùng lúc
}
```

Stub và driver

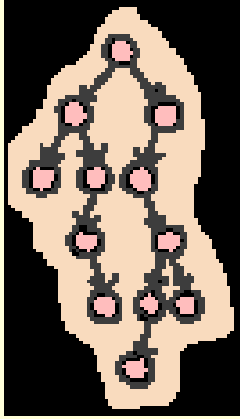


- Stub:
  - Viết các prototype trước
  - Viết dummy code để thử nghiệm
  - Ví dụ:

```
bool user says yes( ) {
    return(true);
}
```

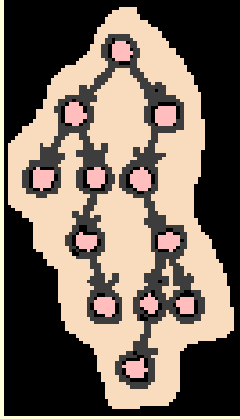
- Driver:
  - Viết một chương trình nhỏ để kiểm tra
- Thư viện cá nhân:
  - Gom các hàm dùng chung thành thư viện

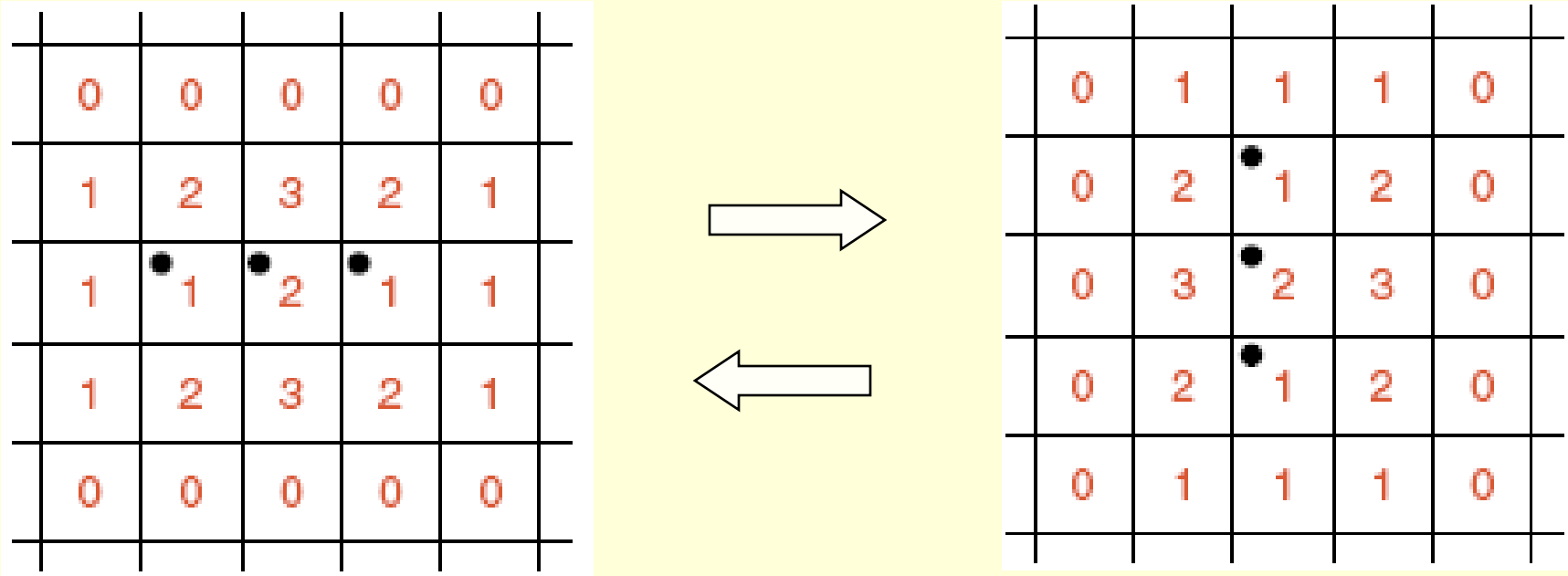
Trò chơi Life



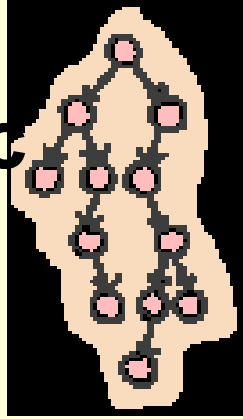
- Luật:
  - Một ma trận các tế bào là sống hay chết
  - Các tế bào lân cận được tính là tám ô xung quanh
  - Quá trình tiến hoá áp dụng cho một trạng thái hiện tại
  - Một tế bào sống là sống ở thế hệ kế nếu có 2 hoặc 3 tế bào sống lân cận và chết trong trường hợp khác
  - Một tế bào đang chết sẽ sống ở thế hệ kế nếu nó có chính xác 3 tế bào sống lân cận, nếu không nó vẫn chết tiếp.
  - Tất cả các tế bào được kiểm chứng cùng một lúc để quyết định trạng thái sống, chết ở thế hệ kế

Trò chơi Life – Ví dụ





Trò chơi Life – Thiết kế phương thức



int Life :: neighbor_count(int row, int col)

Pre: The Life object contains a configuration, and the coordinates row and col define a cell inside its hedge.

Post: The number of living neighbors of the specified cell is returned.

void Life :: update( )

Pre: The Life object contains a configuration.

Post: The Life object contains the next generation of configuration.

void Life :: initialize( )

Pre: None.

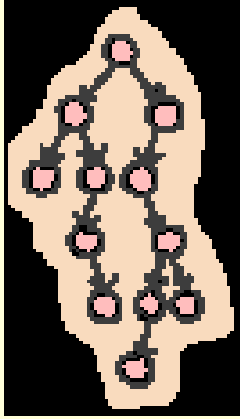
Post: The Life object contains a configuration specified by the user.

void Life :: print( )

Pre: The Life object contains a configuration.

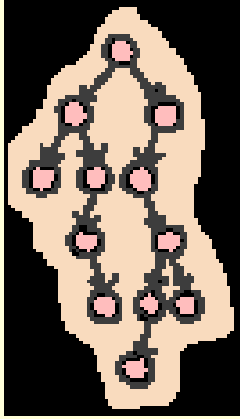
Post: The configuration is written for the user.

Trò chơi Life – Thiết kế class



```
const int maxrow = 20
const maxcol = 60;

class Life {
public:
  void initialize( );
  void print( );
  void update( );
private:
  int grid[maxrow][maxcol];
  int neighbor_count(int row, int col);
};
```

Trò chơi Life – Đếm số tế bào sống lân cận

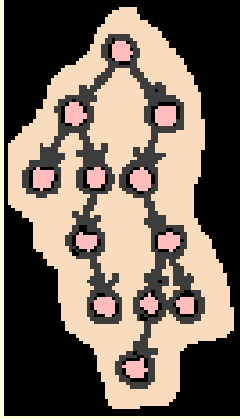


- Mã C++:

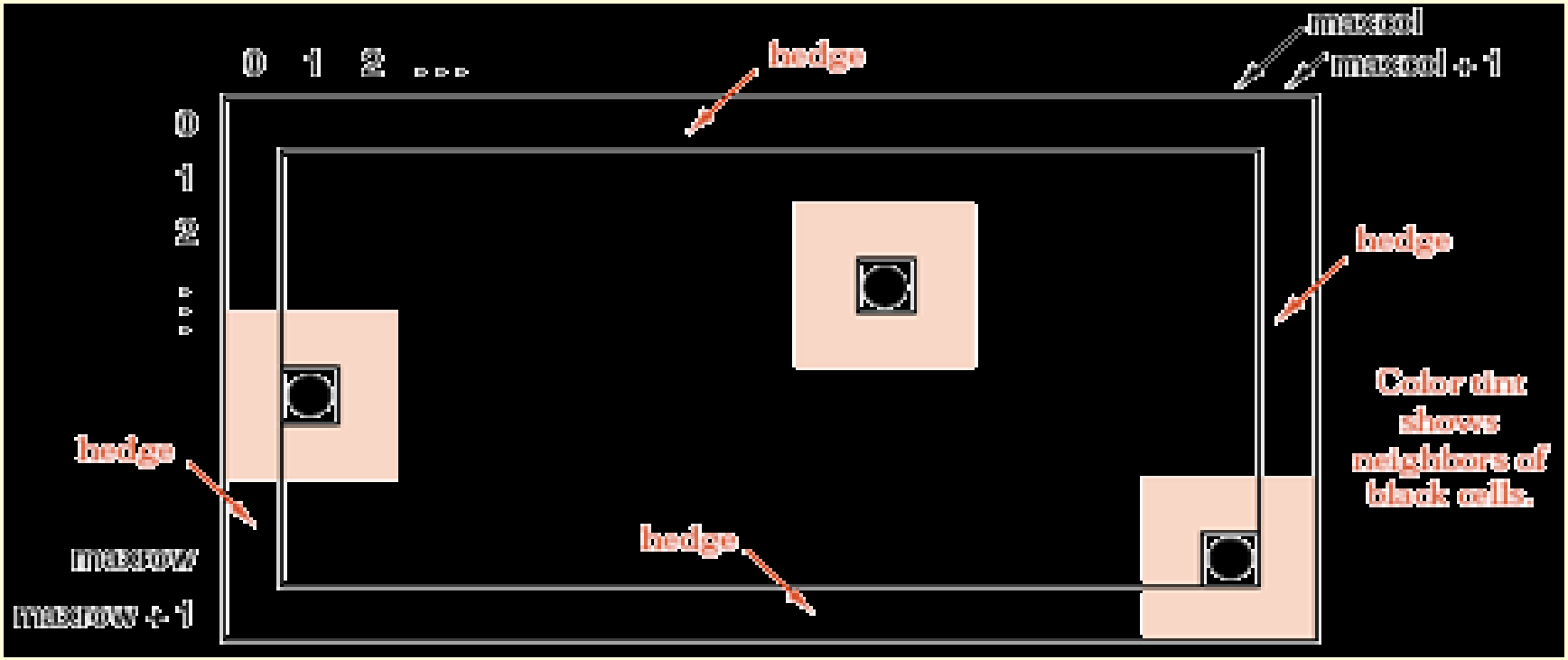
```
count = 0
for (i = row − 1; i <= row + 1; i++)
   for (j = col − 1; j <= col + 1; j++)
         count += grid[i][j];
count −= grid[row][col];
```

- Sai chỗ nào?
  - Nếu như row hoặc col là ngay các biên của array
  - Các giá trị của các tế bào không là 1 hoặc 0

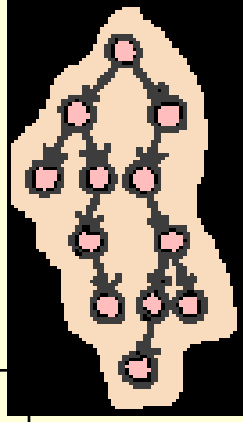
Trò chơi Life – Thay đổi thiết kế



- Giải pháp:
  - Thêm vào 2 cột và 2 hàng giả có giá trị luôn là 0
  - Khai báo dữ liệu: grid[maxrow + 2][maxcol + 2]



Trò chơi Life – Giải thuật cập nhật



Algorithm Update

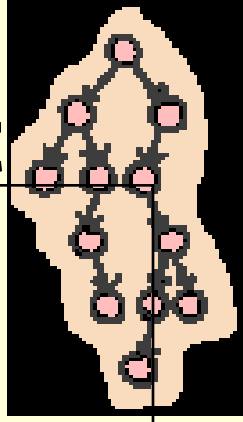
Input: một trạng thái sống

Output: trạng thái của thế hệ kế tiếp

1. Khai báo một grid mới
2. Duyệt qua toàn bộ tế bào của trạng thái hiện tại
    2.1. Đếm số tế bào sống xung quanh ô hiện tại
    2.2. Nếu là 2 thì trạng thái mới chính là trạng thái cũ
    2.3. Nếu là 3 thì trạng thái mới là sống
    2.4. Ngược lại là chết
3. Cập nhật grid mới vào trong grid cũ

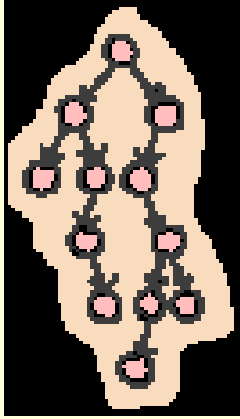
End Update

Trò chơi Life – Mã C++ cập nhật



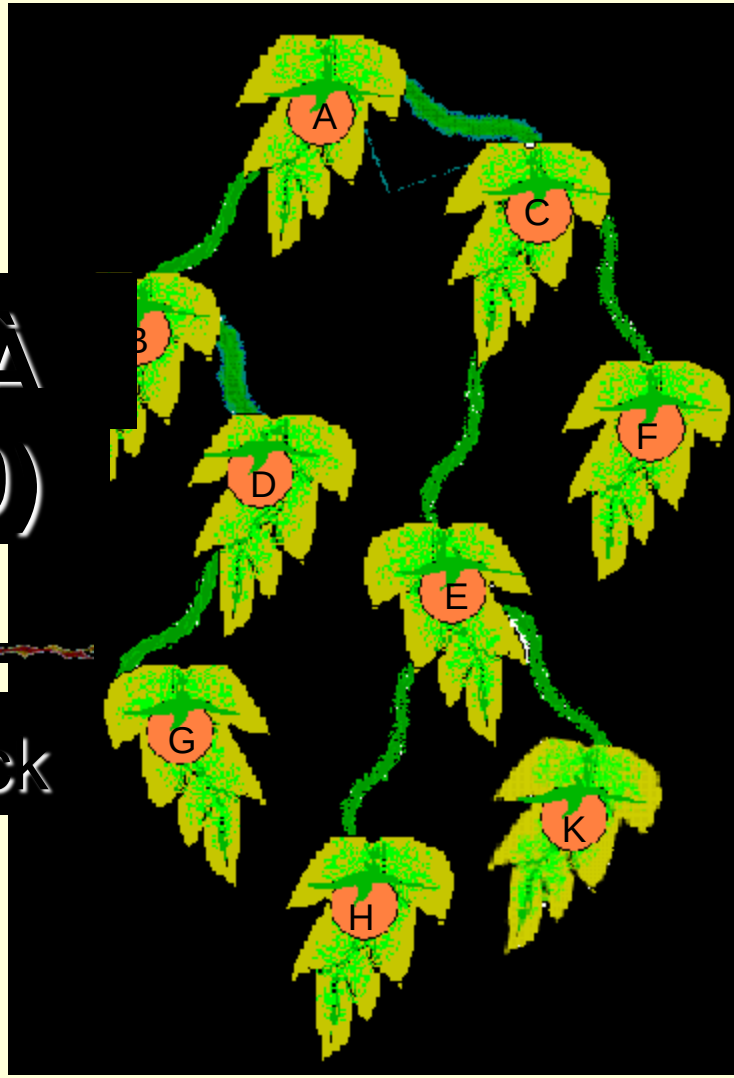
```
void Life::update()
/* Pre: grid đang chứa một trạng thái của thực thể sống
   Post: grid sẽ chứa trạng thái tiến hóa mới của thực thể sống này */
{
    int row, col;
    int new_grid[maxrow + 2][maxcol + 2];       //Chứa trạng thái mới vào đây
    for (row = 1; row <= maxrow; row++)
        for (col = 1; col <= maxcol; col++)
            switch (neighbor_count(row, col)) {
                case 2:                                 //Trạng thái của tế bào không đổi
                    new_grid[row][col] = grid[row][col]; break;
                case 3:                                 //Tế bào sẽ sống
                    new_grid[row][col] = 1; break;
                default:                                //Tế bào sẽ chết
                    new_grid[row][col] = 0;
             }
    for (row = 1; row <= maxrow; row++)
        for (col = 1; col <= maxcol; col++)
            grid[row][col] = new_grid[row][col];    //Cập nhật các tế bào cùng lúc
}
```

Kết luận



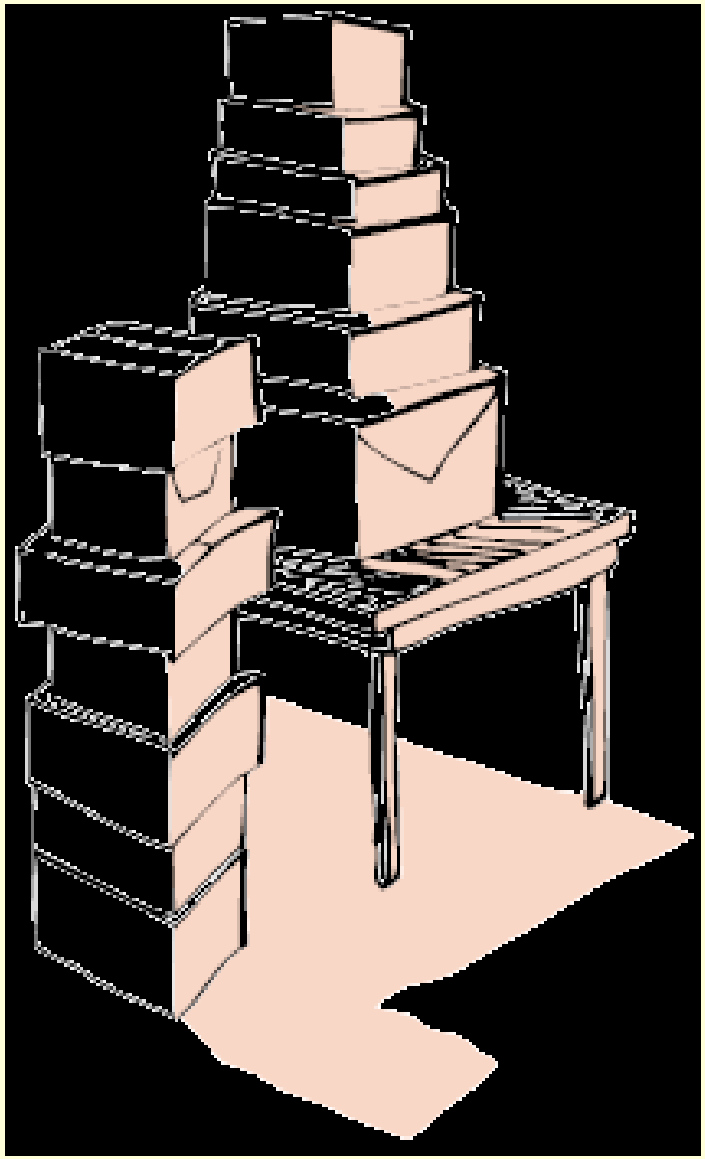
- Sự liên quan giữa CTDL và giải thuật:
  - Cấu trúc dữ liệu cụ thể: chọn giải thuật
  - Giải thuật cụ thể: chọn cấu trúc dữ liệu
- Cấu trúc dữ liệu trừu tượng:
  - Dữ liệu cụ thể bên trong
  - Các phương thức: interface ra bên ngoài
  - Thích hợp cho phương pháp hướng đối tượng

CẤU TRÚC DỮ LIỆU VÀ GIẢI THUẬT (501040)



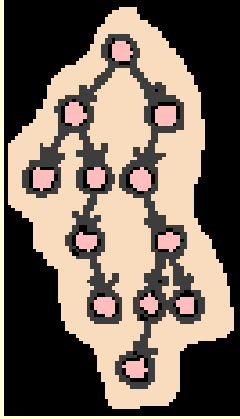
Chương 2: Stack

Mô tả stack

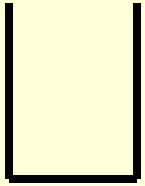


- Một stack là một cấu trúc dữ liệu mà việc thêm vào và loại bỏ được thực hiện tại một đầu (gọi là đỉnh – top của stack).
- Là một dạng vào sau ra trước – LIFO (Last In First Out)

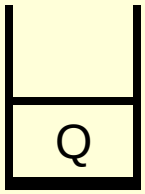
Ví dụ về stack



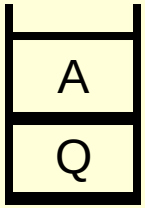
- Stack rỗng:



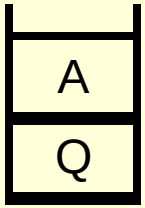
- Đẩy (push) Q vào:



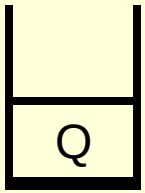
- Đẩy A vào:



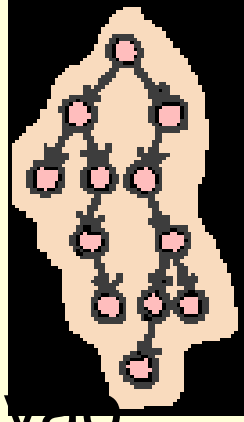
- Lấy (pop) ra một => được A:



- Lấy ra một => được Q và stack rỗng:



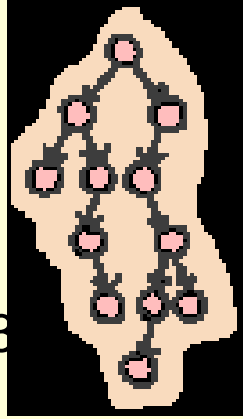
Ứng dụng: Đảo ngược danh sách



- Yêu cầu: Đảo ngược một danh sách nhập vào
- Giải thuật:

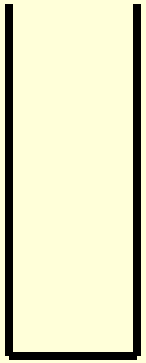
1. Lặp lại n lần
   1.1. Nhập vào một giá trị
   1.2. Đẩy nó vào stack
2. Lặp khi stack chưa rỗng
   2.1. Lấy một giá trị từ stack
   2.2. In ra

Đảo ngược danh sách – Ví dụ

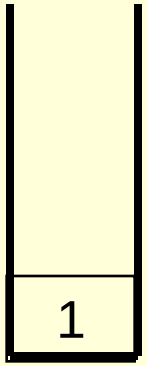


Cần nhập 4 số vào

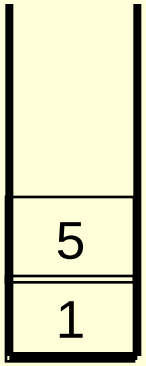
Ban đầu



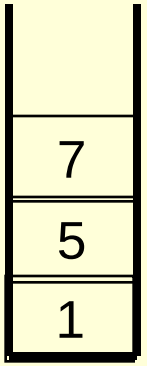
Nhập 1



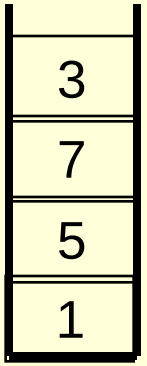
Nhập 5



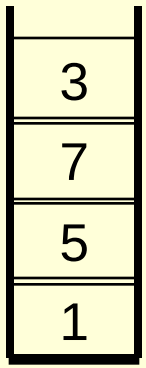
Nhập 7



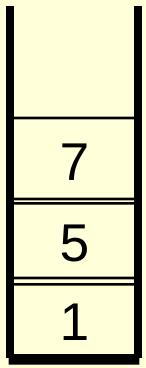
Nhập 3



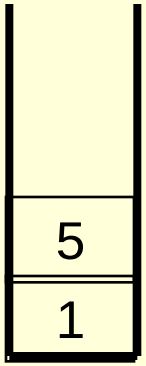
Lấy ra => 3



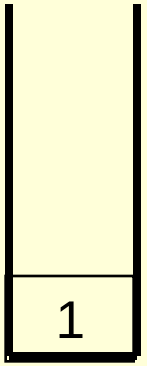
Lấy ra => 7



Lấy ra => 5

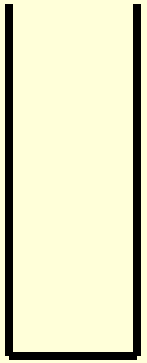


Lấy ra => 1

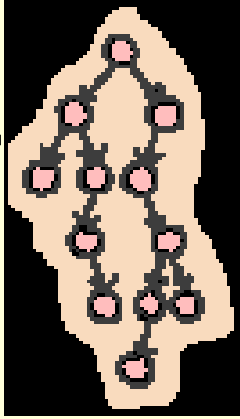


Stack đã rỗng

Ngừng



Đảo ngược danh sách – Mã C++



```
#include <stack>
using namespace std;

int main( ) {
    int n;
    double item;
    stack<double> numbers;
    cout << "Bao nhieu so nhap vao? "
    cin >> n;
    for (int i = 0; i < n; i++) {
        cin >> item;
        numbers.push(item);
    }
    while (!numbers.empty( )) {
        cout << numbers.top( ) << " ";
        numbers.pop( );
    } }
```

sử dụng STL
(Standard Template Library)

khai báo một stack có kiểu dữ liệu
của các phân tử bên trong là double

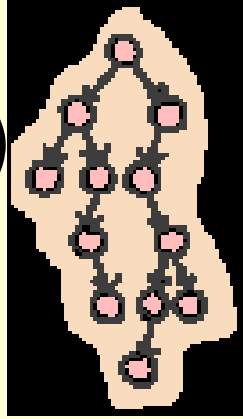
đẩy một số vào trong stack

kiểm tra xem stack có khác rỗng không

lấy giá trị trên đỉnh của stack ra,
stack không đổi

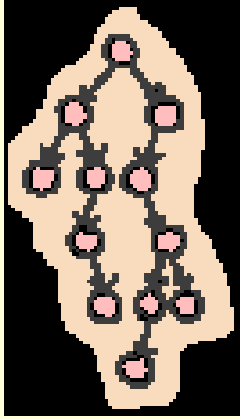
lấy giá trị trên đỉnh của stack ra khỏi stack,
đỉnh của stack bây giờ là giá trị kế tiếp

Kiểu trừu tượng (abstract data type)



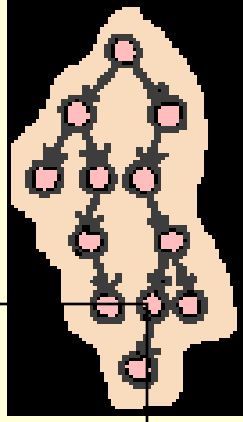
- ĐN1: Một kiểu (type)
  - một tập hợp
  - mỗi thành phần của tập hợp này là các giá trị (value)
  - Ví dụ: int, float, char là các kiểu cơ bản
- ĐN2: Một dãy của kiểu T
  - có chiều dài bằng 0 là rỗng
  - có chiều dài n (n>=1): bộ thứ tự (Sn-1, t)
    - Sn-1: dãy có chiều dài n-1 thuộc kiểu T
    - t là một giá trị thuộc kiểu T.

Stack trừu tượng



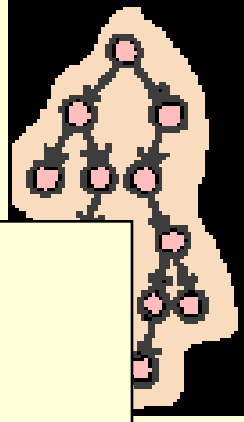
- Một stack kiểu T:
  - Một dãy hữu hạn kiểu T
  - Một số tác vụ:
    1. Khởi tạo stack rỗng (*create*)
    2. Kiểm tra rỗng (*empty*)
    3. Đẩy một giá trị vào trên đỉnh của stack (*push*)
    4. Bỏ giá trị đang có trên đỉnh của stack (*pop*)
    5. Lấy giá trị trên đỉnh của stack, stack không đổi (*top*)

Thiết kế stack



```
enum Error_code  {fail, success, overflow, underflow};

template <class Entry>
class Stack {
public:
        Stack();                                    //constructor
        bool empty() const;                         //kiểm tra rỗng
        Error_code push(const Entry &item);         //đẩy item vào
        Error_code pop();                           //bỏ phần tử trên đỉnh
        Error_code top(Entry &item);                //lấy giá trị trên đỉnh
        //khai báo một số phương thức cần thiết khác
private:
        //khai báo dữ liệu và hàm phụ trợ chỗ này
};
```

Thiết kế các phương thức



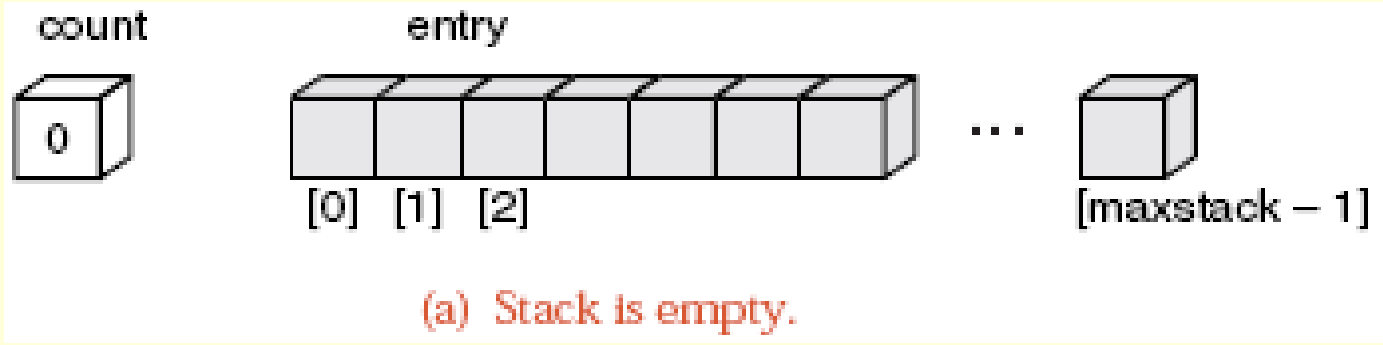
template <**class** Entry>
bool Stack<Entry>::empty() **const**;
Pre: Không có
Post: Trả về giá trị **true** nếu stack hiện tại là rỗng, ngược lại thì trả về **false**

template <**class** Entry>
Error_code Stack<Entry>::push(**const** Entry &item);
Pre: Không có
Post: Nếu stack hiện tại không đầy, item sẽ được thêm vào đỉnh của stack.
Ngược lại trả về giá trị **overflow** của kiểu Error_code và stack không đổi.

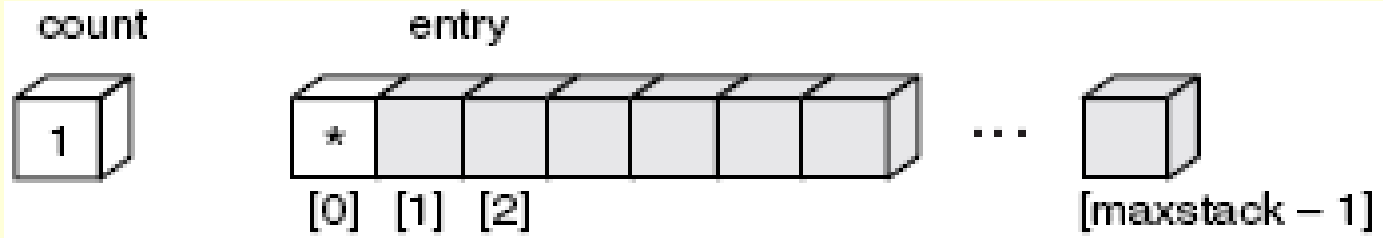
template <**class** Entry>
Error_code Stack<Entry>::pop() **const**;
Pre: Không có
Post: Nếu stack hiện tại không rỗng, đỉnh của stack hiện tại sẽ bị hủy bỏ.
Ngược lại trả về giá trị **underflow** của kiểu Error_code và stack không đổi.

template <**class** Entry>
Error_code Stack<Entry>::top(Entry &item) **const**;
Pre: Không có
Post: Nếu stack hiện tại không rỗng, đỉnh của stack hiện tại sẽ được chép vào tham biến item. Ngược lại trả về giá trị **fail** của kiểu Error_code.

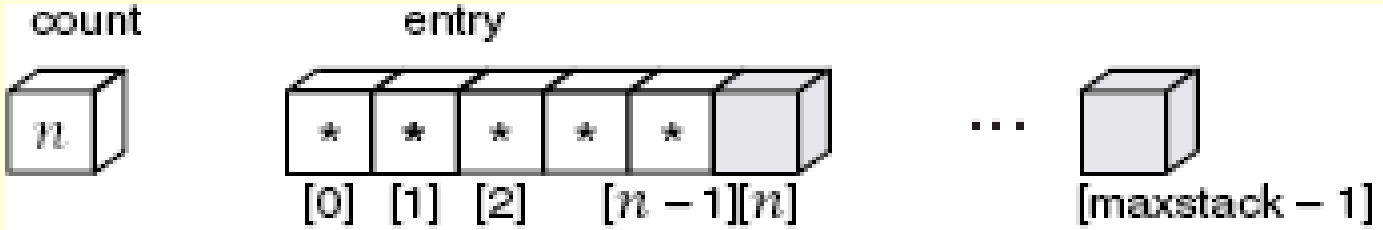
Hiện thực stack liên tục



(a) Stack is empty.

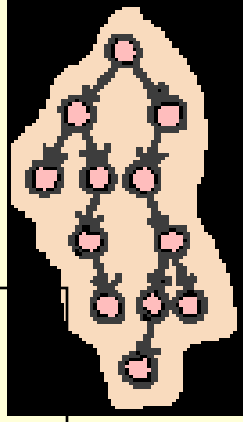


(b) Push the first entry.



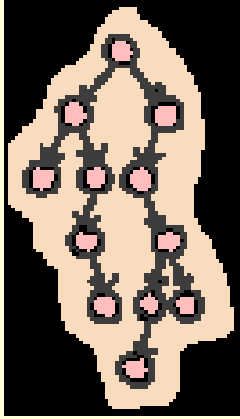
(c) n items on the stack

Khai báo stack liên tục

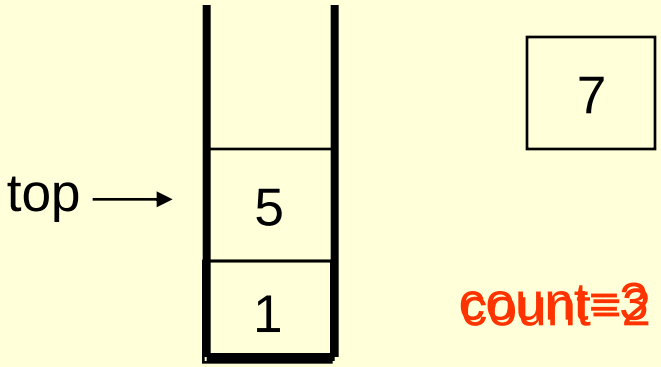


```
const  int  maxstack = 10;          //small number for testing

template <class Entry>
class Stack {
public:
          Stack( );
          bool empty( ) const;
          Error_code pop( );
          Error_code top(Entry &item) const;
          Error_code push(const Entry &item);
private:
          int count;
          Entry entry[maxstack];
};
```

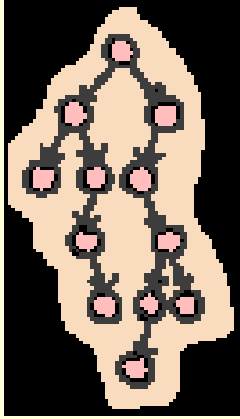
Đẩy một phần tử vào stack



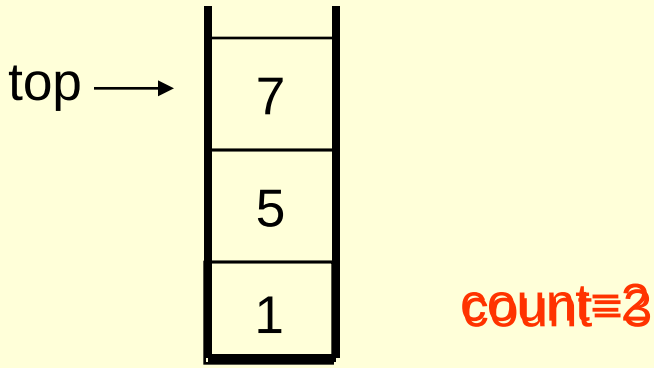
- Giải thuật:
  1. Nếu còn chỗ trống trong stack
     - 1.1. Tăng vị trí đỉnh lên 1
     - 1.2. Chứa giá trị vào vị trí đỉnh của stack
     - 1.3. Tăng số phần tử lên 1



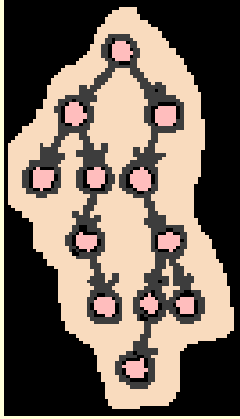
Bỏ phần tử trên đỉnh stack



- Giải thuật:
  1. Nếu còn phần tử trong stack
     - 1.1. Giảm vị trí đỉnh đi 1
     - 1.2. Giảm số phần tử đi 1



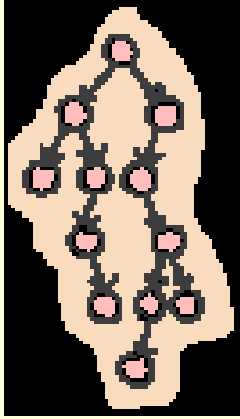
Thêm/Bỏ phần tử - Mã C++



```
template <class Entry>
Error_code Stack<Entry>:: push(const Entry &item) {
        if (count >= maxstack)
                return overflow;
        else
                entry[count++] = item;
        return success;
}
```

```
template <class Entry>
Error_code Stack<Entry>:: pop() {
        if (count == 0)
                return underflow;
        else
                count--;
        return success;
}
```

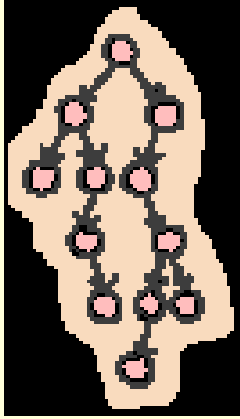
Lấy giá trị trên đỉnh stack



- Giải thuật:
  1. Nếu còn phần tử trong stack
     1.1. Trả về giá trị tại vị trí đỉnh
- Mã C++:

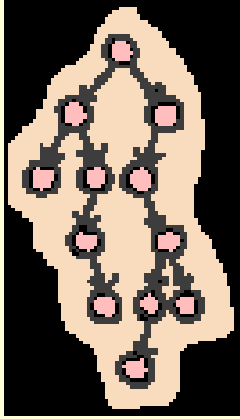
```
template <class Entry>
Error_code Stack<Entry>:: top(Entry &item) {
        if (count == 0)
                return underflow;
        else
                item = entry[count - 1];
        return success;
}
```

Reverse Polish Calculator



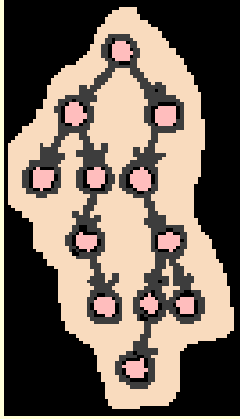
- Mô tả bài toán:
  - Các toán hạng được đọc vào trước và đẩy vào stack
  - Khi đọc vào toán tử, lấy hai toán hạng ra từ stack, tính toán với toán tử này, rồi đẩy kết quả vào stack
- Thiết kế phần mềm:
  - Cần một stack để chứa toán hạng
  - Cần hàm get_command để nhận lệnh từ người dùng
  - Cần hàm do_command để thực hiện lệnh

Reverse Polish Calculator – Thiết kế chức năng

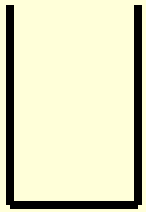


- Tập lệnh:
  - '?': đọc một giá trị rồi đẩy vào stack
  - Toán tử '+', '-', '*', '/': lấy 2 giá trị trong stack, tính toán và đẩy kết quả vào stack
  - Toán tử '=': in đỉnh của stack ra
  - 'q': kết thúc chương trình

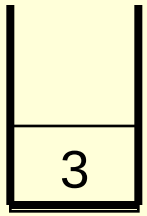
Reverse Polish Calculator – Ví dụ



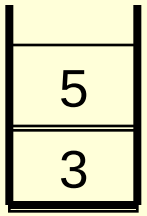
Tính toán biểu thức: 3 5 + 2 * =



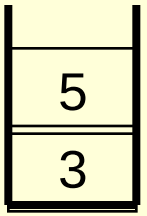
Ban đầu



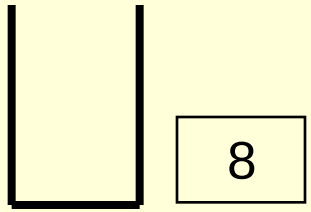
Toán tử ?
Nhập vào 3



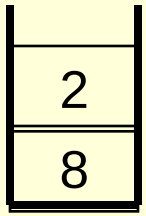
Toán tử ?
Nhập vào 5



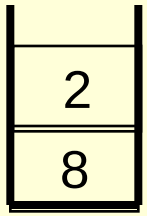
Toán tử +
Lấy ra 5 và 3
Tính 3 + 5 => 8



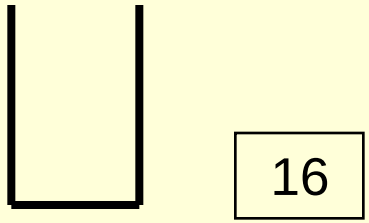
Đẩy 8 vào



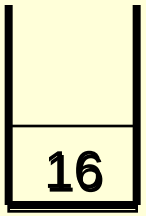
Toán tử ?
Nhập vào 2



Toán tử *
Lấy ra 2 và 8
Tính 8 * 2 => 16

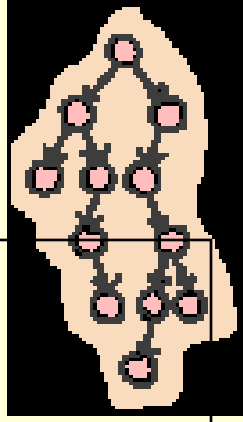


Đẩy vào 16



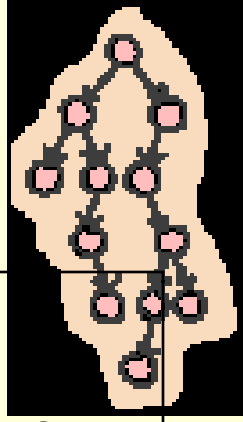
Toán tử =
In ra 16

Reverse Polish Calculator – Hàm get_command



```
char get command( ) {
  char command;
  bool waiting = true;
  cout << "Select command and press < Enter > :";
  while (waiting) {
    cin >> command;
    command = tolower(command);
    if (command == '?' || command == '=' || command == '+' ||
        command == '−'|| command == '*' || command == '/' ||
        command == 'q') waiting = false;
    else {
      cout << "Please enter a valid command:" << endl
           << "[?]push to stack [=]print top" <<endl
           << "[+] [−] [*] [/] are arithmetic operations" << endl
           << "[Q]uit." << endl;
    }
  }
  return command;
}
```

Reverse Polish Calculator – Giải thuật tính toán với toán tử



Algorithm Op_process

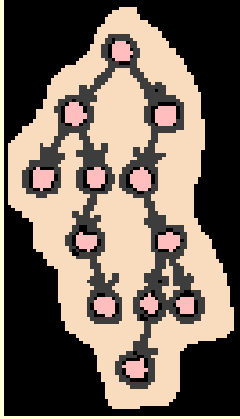
Input: toán tử *op*, stack chứa các toán hạng

Output: stack chứa các toán hạng sau khi tính xong toán tử *op*

1. Nếu stack không rỗng
   - 1.1. Lấy đỉnh stack ra thành p
   - 1.2. Bỏ phần tử trên đỉnh stack
   - 1.3. Nếu stack rỗng
     - 1.3.1. Đẩy p ngược lại
     - 1.3.2. Báo lỗi và thoát
   - 1.4. Lấy đỉnh stack ra thành q
   - 1.5. Bỏ phần tử trên đỉnh stack
   - 1.6. Tính toán (q *op* p)
   - 1.7. Đẩy kết quả vào stack

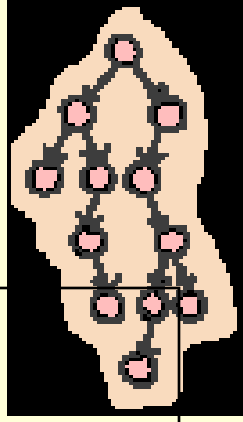
End Op_process

Reverse Polish Calculator – Mã C++ cho toán tử cộng



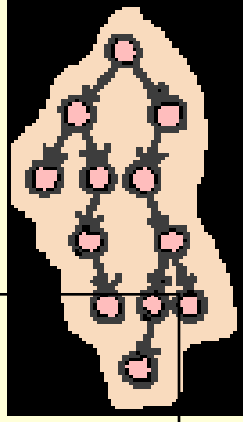
```
if (numbers.top(p) == underflow)
    cout << "Stack rỗng";
else {
    numbers.pop( );
    if (numbers.top(q) == underflow) {
        cout << "Stack chỉ có 1 trị";
        numbers.push(p);
    }
    else {
        numbers.pop( );
        if (numbers.push(q + p) == overflow)
            cout << "Stack đầy";
    }
}
```

Reverse Polish Calculator – Chương trình chính



```
#include "stack.cpp"

//prototype
void introduction( );
void instructions( );
char get_command( );
bool do_command(char command, Stack<double> &numbers);

int main( ) {
    Stack<double> stored_numbers;
    introduction( );
    instructions( );
    while (do_command(get_command( ), stored_numbers));
}
//implementation
…
```

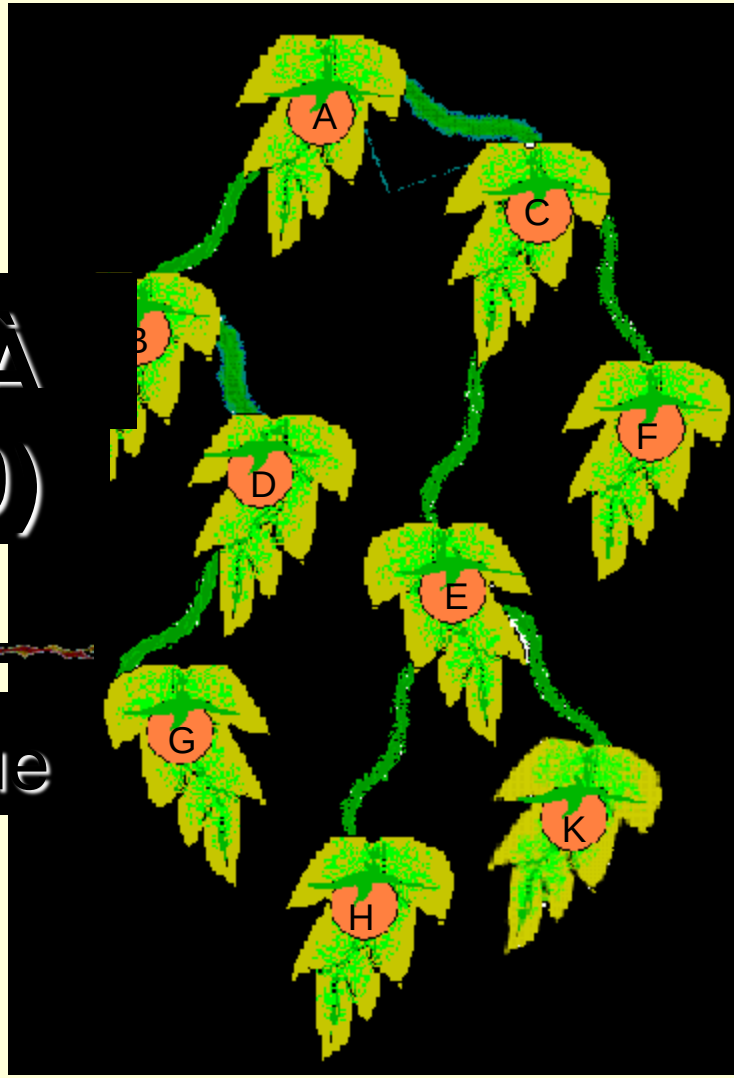
Reverse Polish Calculator – Hàm do_command



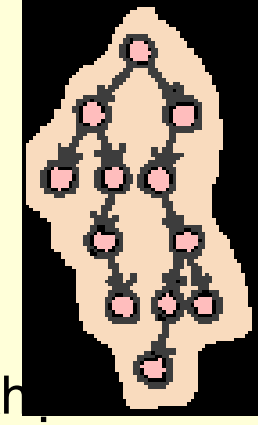
```
bool do_command(char command, Stack &numbers) {
    double p, q;
    switch (command) {
        case '?':
            cout << "Enter a real number: " << flush; cin >> p;
            if (numbers.push(p) == overflow)
                cout << "Warning: Stack full, lost number" << endl; break;
        case '=':
            if (numbers.top(p) == underflow) cout << "Stack empty" << endl;
            else cout << p << endl; break;

        // Add options for further user commands.
        case 'q': cout << "Calculation finished.\n"; return false;
    }
return true;
}
```

CẤU TRÚC DỮ LIỆU VÀ GIẢI THUẬT (501040)

Chương 3: Queue

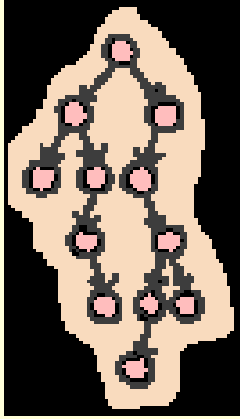


Mô tả queue



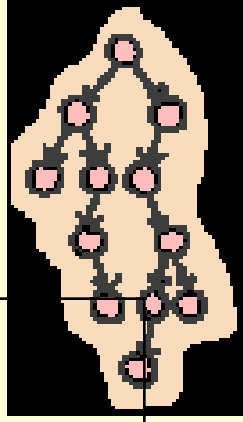
- Một queue là một cấu trúc dữ liệu mà việc thêm vào được thực hiện ở một đầu (rear) và việc lấy ra được thực hiện ở đầu còn lại (front)
- Phần tử vào trước sẽ ra trước – FIFO (First In First Out)

Queue trừu tượng



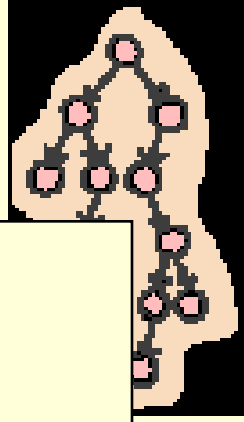
- Một queue kiểu T:
  - Một dãy hữu hạn kiểu T
  - Một số tác vụ:
    1. Khởi tạo queue rỗng (*create*)
    2. Kiểm tra rỗng (*empty*)
    3. Thêm một giá trị vào cuối của queue (*append*)
    4. Bỏ giá trị đang có ở đầu của queue (*serve*)
    5. Lấy giá trị ở đầu của queue, queue không đổi (*retrieve*)

Thiết kế queue



```
enum Error_code {fail, success, overflow, underflow};

template <class Entry>
class Queue {
public:
        Queue();                                    //constructor
        bool empty() const;                         //kiểm tra rỗng
        Error_code append(const Entry &item);       //đẩy item vào
        Error_code serve();                         //bỏ 1 phần tử ở đầu
        Error_code retrieve(Entry &item);           //lấy giá trị ở đầu
        //khai báo một số phương thức cần thiết khác
private:
        //khai báo dữ liệu và hàm phụ trợ chỗ này
};
```

Thiết kế các phương thức



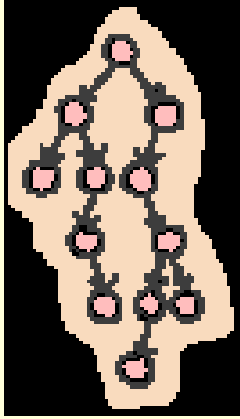
template <**class** Entry>
bool Queue<Entry>::empty() **const**;
Pre: Không có
Post: Trả về giá trị **true** nếu queue hiện tại là rỗng, ngược lại thì trả về **false**

template <**class** Entry>
Error_code Queue<Entry>::append(**const** Entry &item);
Pre: Không có
Post: Nếu queue hiện tại không đầy, item sẽ được thêm vào cuối của queue.
Ngược lại trả về giá trị **overflow** của kiểu Error_code và queue không đổi.

template <**class** Entry>
Error_code Queue<Entry>::serve() **const**;
Pre: Không có
Post: Nếu queue hiện tại không rỗng, đầu của queue hiện tại sẽ bị hủy bỏ.
Ngược lại trả về giá trị **underflow** của kiểu Error_code và queue không đổi.

template <**class** Entry>
Error_code Queue<Entry>::retrieve(Entry &item) **const**;
Pre: Không có
Post: Nếu queue hiện tại không rỗng, đầu của queue hiện tại sẽ được chép vào tham biến item. Ngược lại trả về giá trị **underflow** của kiểu Error_code.

Mở rộng queue

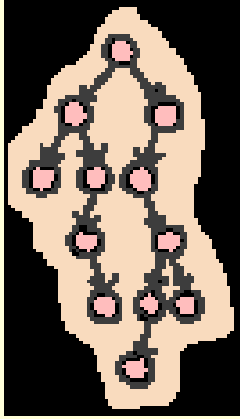


- Có thêm các tác vụ:
  - Kiểm tra đầy (*full*)
  - Tính kích thước (*size*)
  - Giải phóng queue (*clear*)
  - Lấy giá trị ở đầu và bỏ ra khỏi queue (*serve_and_retrieve*)
- Mã C++:

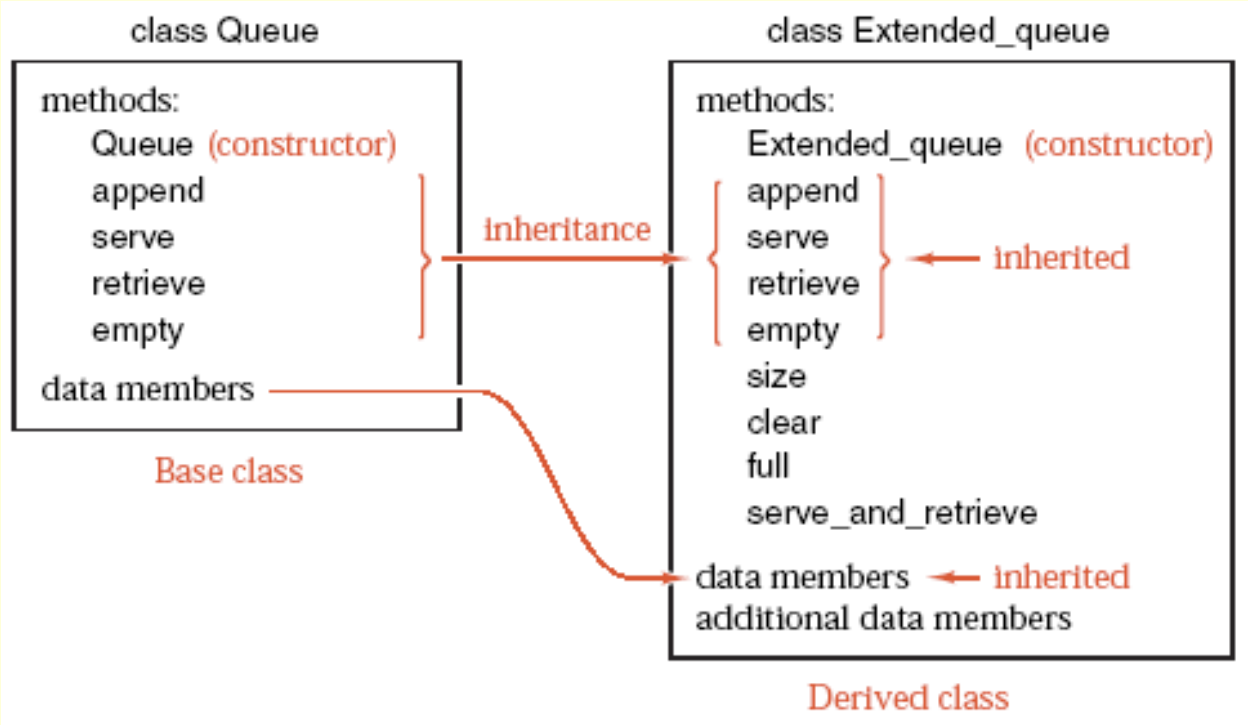
```
template <class Entry>
class Extended_queue: public Queue<Entry> {
public:
   bool full( ) const;
   int size( ) const;
   void clear( );
   Error_code serve_and_retrieve(Entry &item);
};
```

Có các khả năng **public**, **protected**, **private**

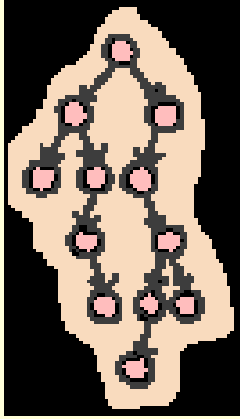
Tính thừa hưởng



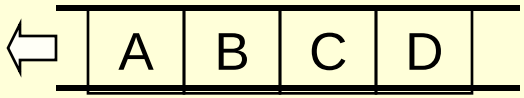
- Dùng tính thừa hưởng:
  - Extended_queue có đầy đủ các thành phần của Queue
  - Thêm vào đó các thành phần riêng của mình



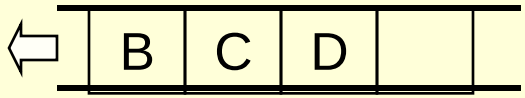
Queue liên tục



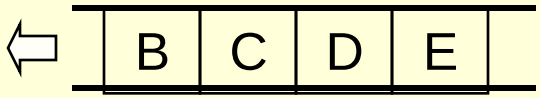
- Dùng một array: Có xu hướng dời về cuối array
- Hai cách hiện thực đầu tiên:
  - Khi lấy một phần tử ra thì đồng thời dời hàng lên một vị trí.



Ban đầu

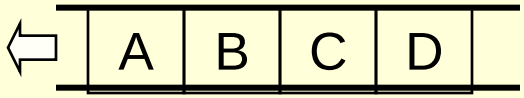


Lấy ra 1 phần tử:
dời tất cả về trước

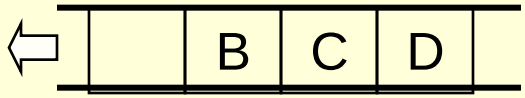


Thêm vào 1 phần tử

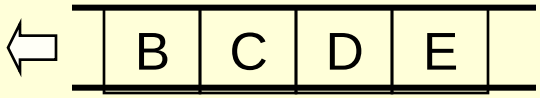
  - Chỉ dời hàng về đầu khi cuối hàng không còn chỗ



Ban đầu

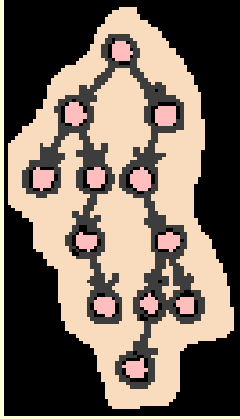


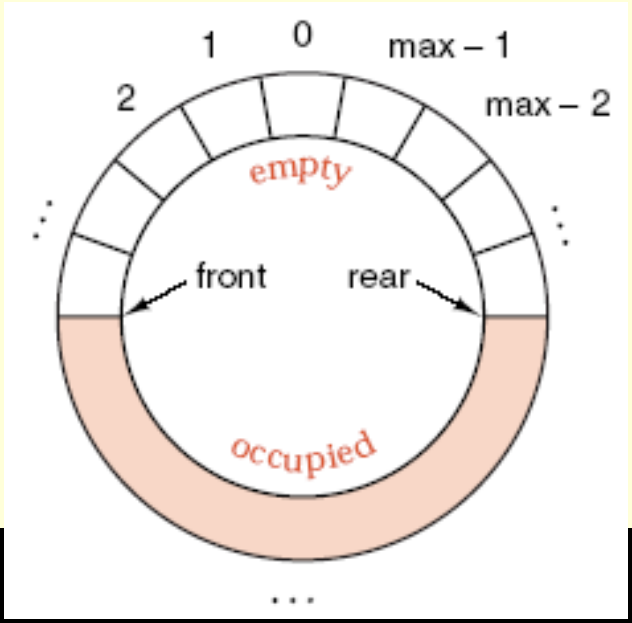
Lấy ra 1 phần tử

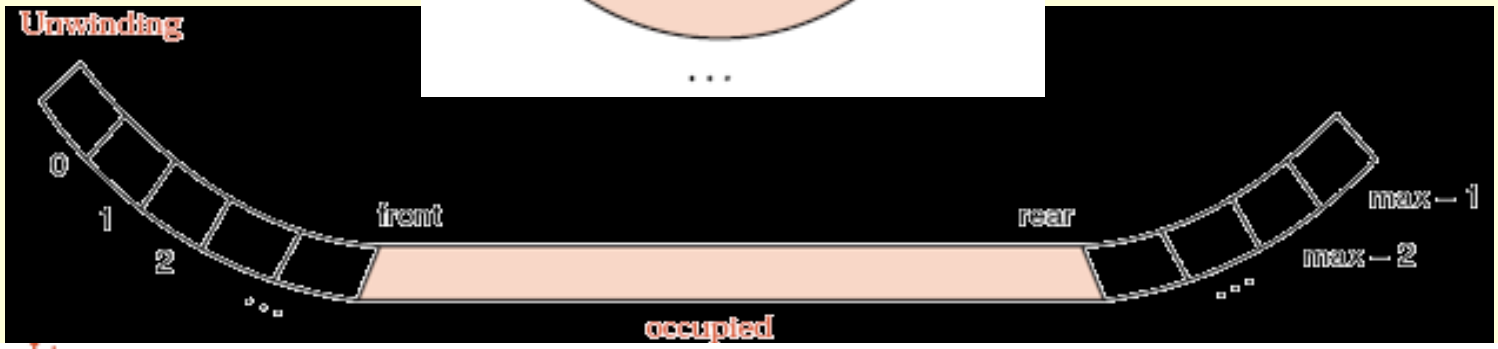


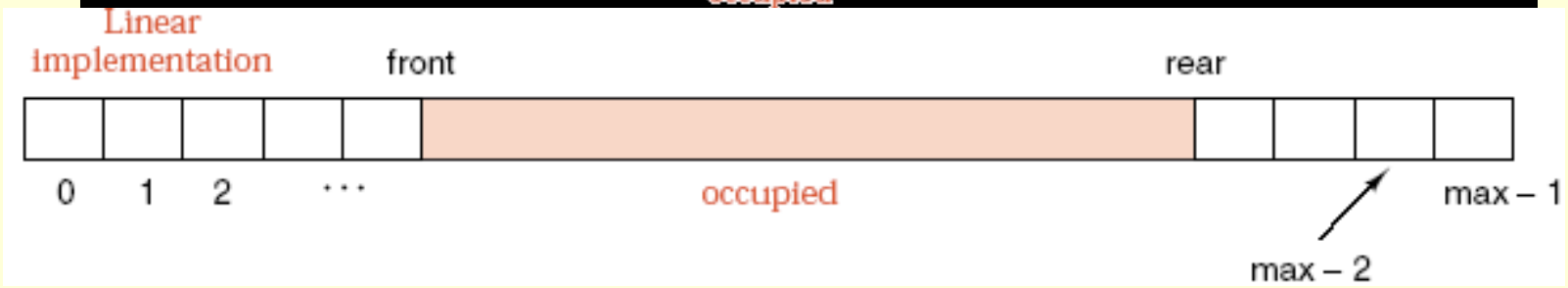
Thêm vào 1 phần tử:
dời tất cả về trước để
trống chỗ thêm vào

Queue là array vòng (circular array)

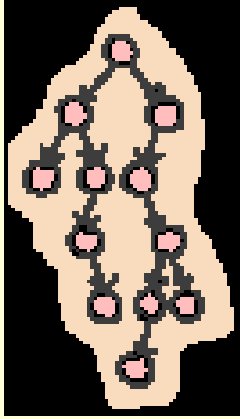






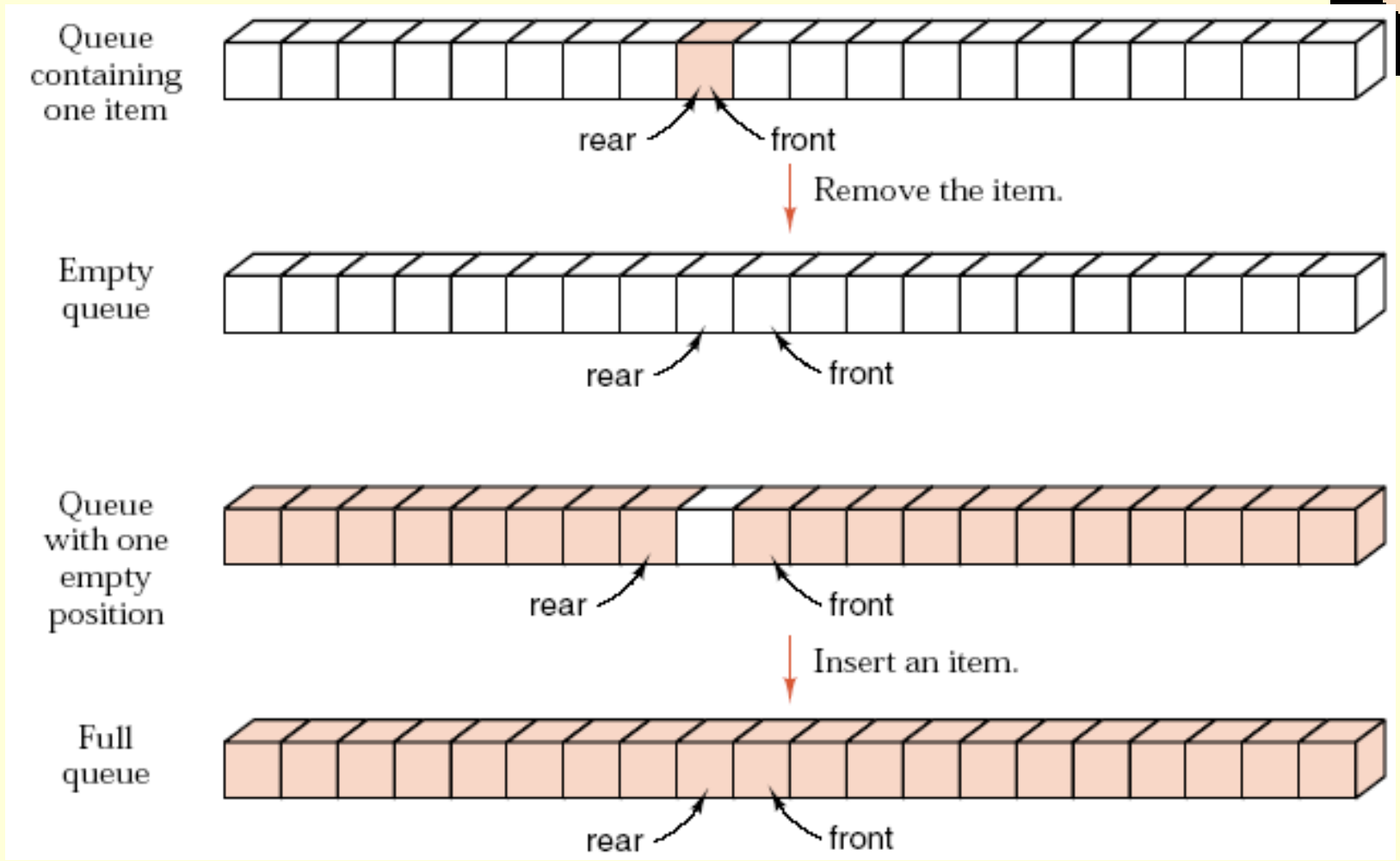


Array vòng với ngôn ngữ C++

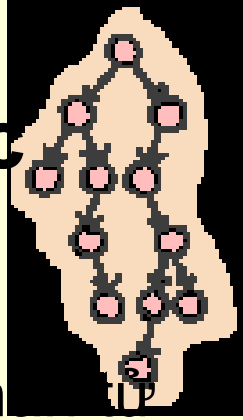


- Xem array như là một vòng:
  - phần tử cuối của array nối với phần tử đầu của array
- Tính toán vị trí kề:
  - i = ((i + 1) == max) ? 0 : (i + 1);
  - if ((i + 1) == max) i = 0; else i = i + 1;
  - i = (i + 1)%max;

Điều kiện biên của queue vòng

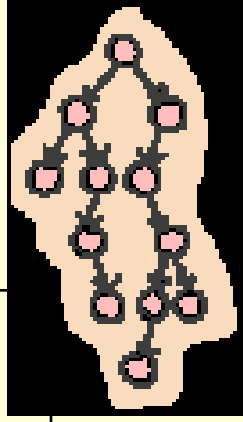


Một số cách hiện thực queue liên tục



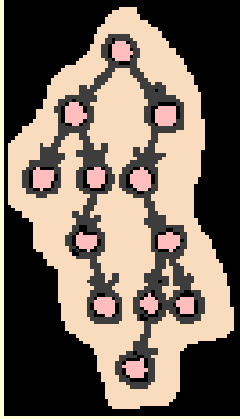
- Một array với front là phần tử đầu và tất cả các ph[illegible] sẽ được dời lên khi lấy ra một phần tử.
- Một array có hai chỉ mục luôn tăng chỉ đến phần tử đầu và cuối.
- Một array vòng có chỉ mục front và rear và một ô luôn trống.
- Một array vòng có chỉ mục front và rear và một cờ (flag) cho biết queue là đầy (rỗng) chưa.
- Một array vòng với chỉ mục front và rear có các giá trị đặc biệt cho biết queue đang rỗng.
- **Một array vòng với chỉ mục front và rear và một số chứa số phần tử của queue.**

Hiện thực queue liên tục



```
const  int  maxqueue = 10; // small value for testing

template <class Entry>
class Queue {
public:
        Queue( );
        bool empty( ) const;
        Error_code serve( );
        Error_code append(const Entry &item);
        Error_code retrieve(Entry &item) const;
protected:
        int count;
        int front, rear;
        Entry entry[maxqueue];
};
```

Khởi tạo và kiểm tra rỗng



- Khởi tạo:

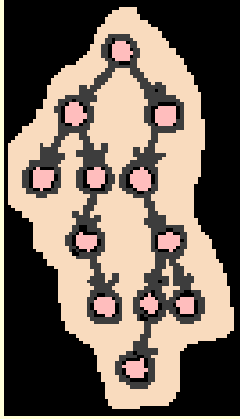
```
template <class Entry>
Queue<Entry>::Queue( ) {
    count = 0;
    rear = maxqueue − 1;
    front = 0;
}
```

- Kiểm tra rỗng:

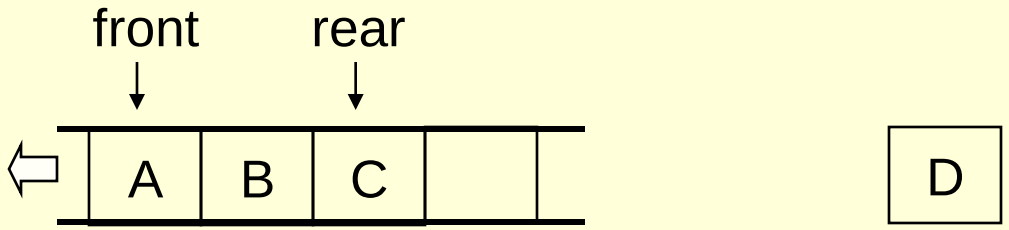
```
template <class Entry>
bool Queue<Entry>::empty( ) const {
    return count == 0;
}
```

Dùng biến count để biết số phần tử trong queue

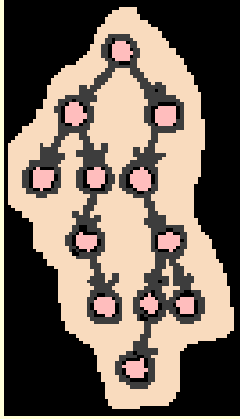
Thêm một giá trị vào queue



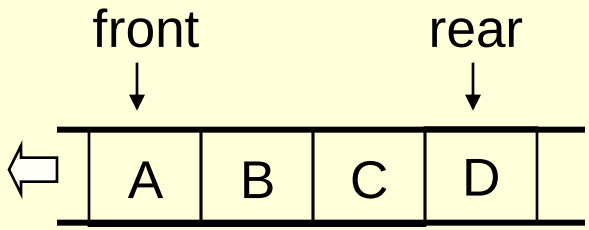
- Giải thuật:
  1. Nếu hàng đầy
     1.1. Báo lỗi overflow
  2. Tính toán vị trí cuối mới theo array vòng
  3. Gán giá trị vào vị trí cuối mới này
  4. Tăng số phần tử lên 1
  4. Báo success



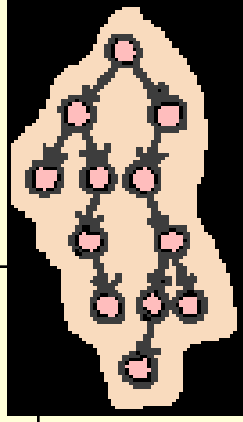
Loại một giá trị khỏi queue



- Giải thuật:
  1. Nếu hàng rỗng
     1.1. Báo lỗi underflow
  2. Tính toán vị trí đầu mới theo array vòng
  3. Giảm số phần tử đi 1
  3. Báo success



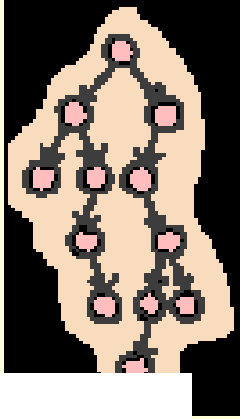
Thêm/loại một giá trị – Mã C++



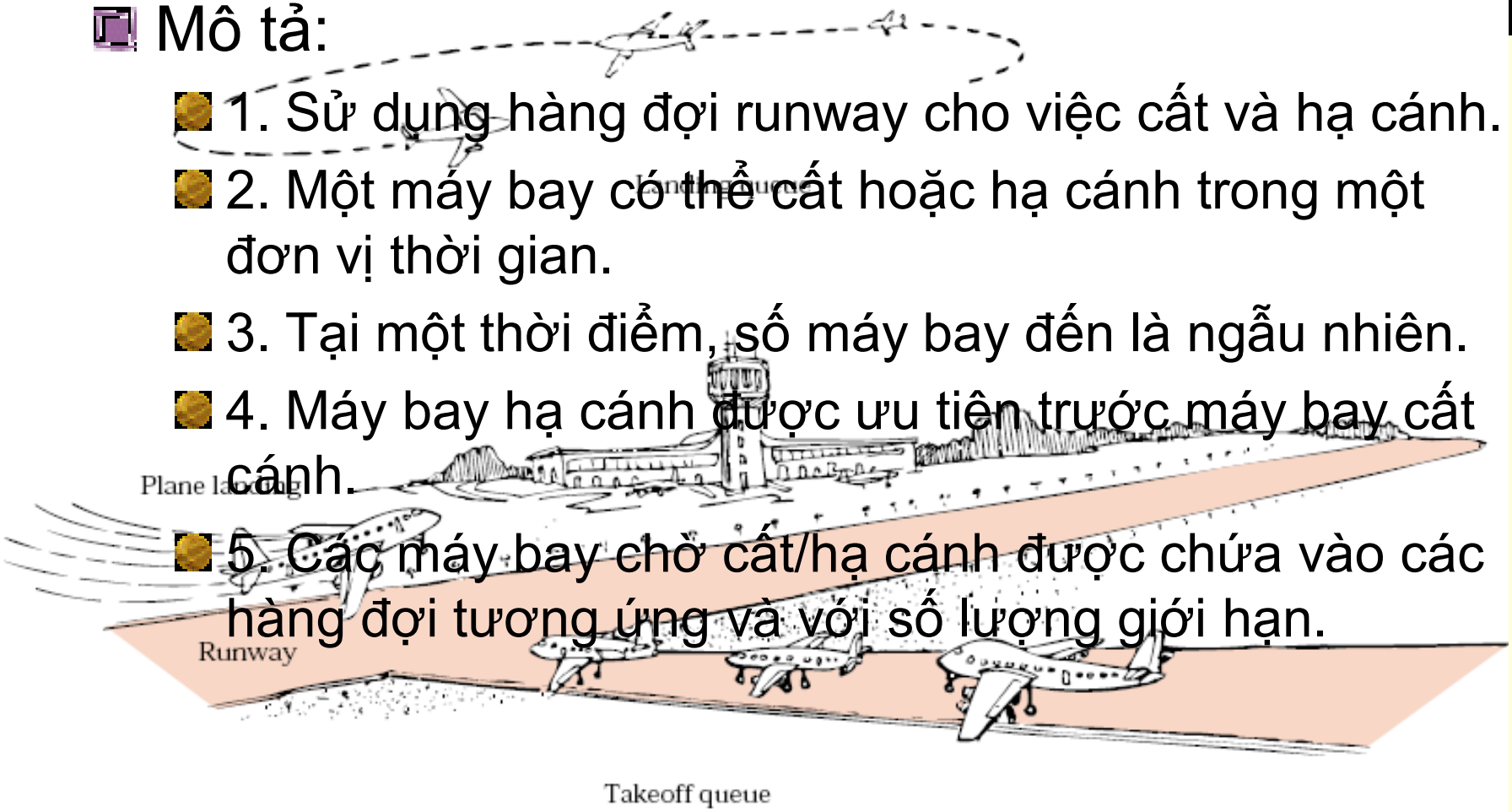
```
template <class Entry>
Error_code Queue<Entry>::append(const Entry &item) {
        if (count >= maxqueue) return overflow;
        count++;
        rear = ((rear + 1) == maxqueue) ? 0 : (rear + 1);
        entry[rear] = item;
        return success;
}
```

```
template <class Entry>
Error_code Queue<Entry>::serve() {
        if (count <= 0) return underflow;
        count--;
        front = ((front + 1) == maxqueue) ? 0 : (front + 1);
        return success;
}
```

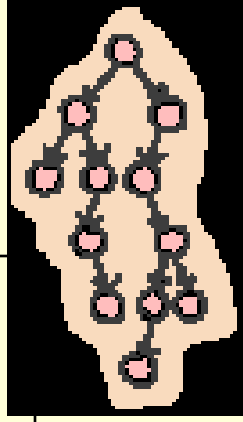
Ứng dụng: Giả lập phi trường



- Mô tả:
  - 1. Sử dụng hàng đợi runway cho việc cất và hạ cánh.
  - 2. Một máy bay có thể cất hoặc hạ cánh trong một đơn vị thời gian.
  - 3. Tại một thời điểm, số máy bay đến là ngẫu nhiên.
  - 4. Máy bay hạ cánh được ưu tiên trước máy bay cất cánh.
  - 5. Các máy bay chờ cất/hạ cánh được chứa vào các hàng đợi tương ứng và với số lượng giới hạn.

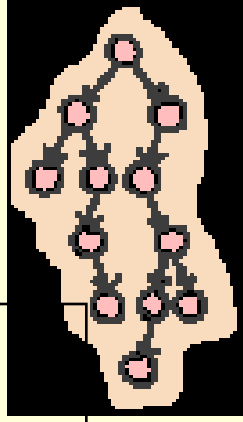


Giả lập phi trường – Hàng đợi



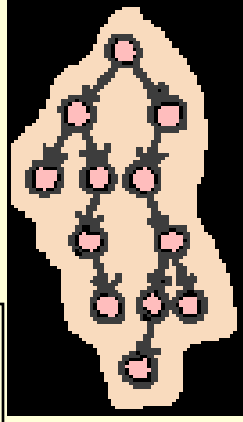
```
enum Runway_activity {idle, land, takeoff};

class Runway {
public:
        Runway(int limit);
        Error_code can_land(const Plane &current);
        Error_code can_depart(const Plane &current);
        Runway_activity activity(int time, Plane &moving);
        void shut_down(int time) const;
private:
        Extended queue landing;
        Extended queue takeoff;
        int queue_limit;
        …
};
```

Giả lập phi trường – Hạ cánh



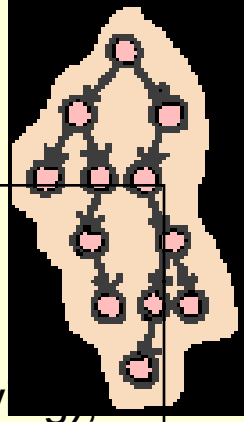
```
Error_code Runway :: can_land(const Plane &current) {
        Error_code result;
        if (landing.size( ) < queue_limit)
                result = landing.append(current);
        else
                result = fail;
        num_land_requests++;
        if (result != success)
                num_land_refused++;
        else
                num_land_accepted++;
        return result;
}
```

Giả lập phi trường – Xử lý



```
Runway_activity Runway::activity(int time, Plane &moving) {
        Runway_activity in_progress;
        if (!landing.empty( )) {
                landing.retrieve(moving);
                in_progress = land;
                landing.serve( );
        } else if (!takeoff.empty( )) {
                takeoff.retrieve(moving);
                in_progress = takeoff;
                takeoff.serve( );
        } else
                in_progress = idle;
        return in_progress;
}
```

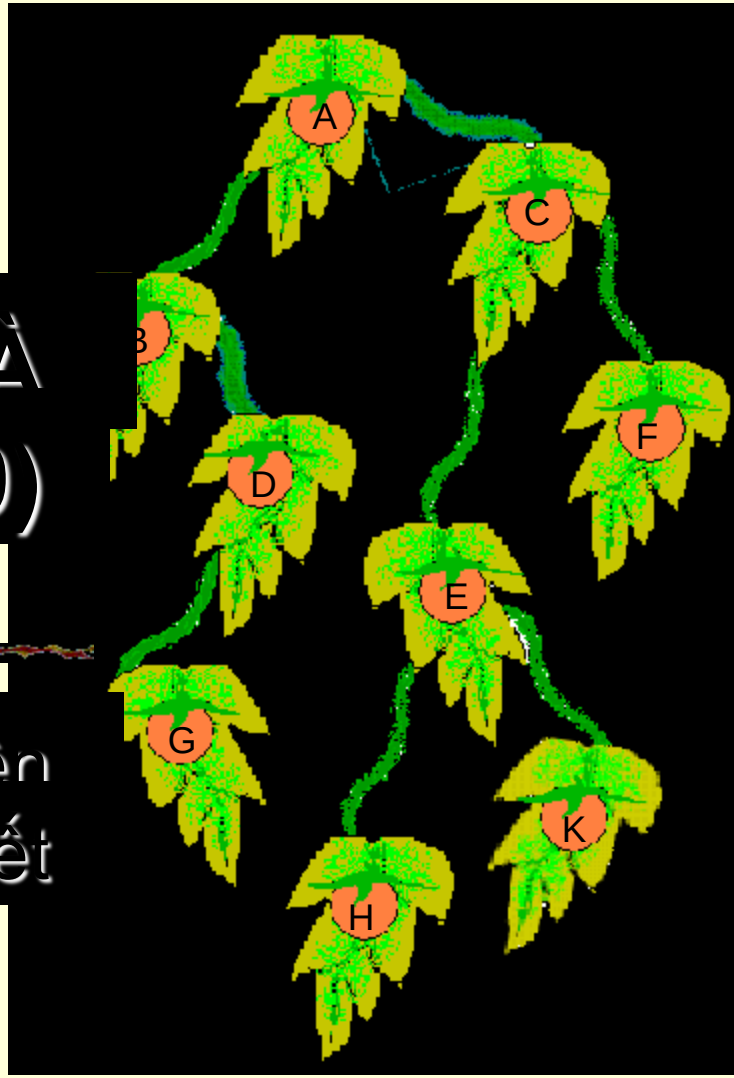
Giả lập phi trường – Giả lập



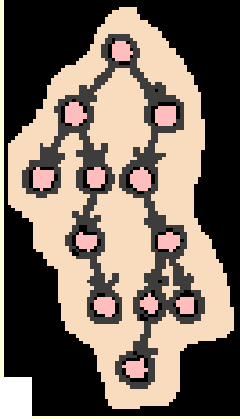
```
for (int current_time = 0; current_time < end_time; current_time++) {
        int number_arrivals = variable.poisson(arrival_rate);
        for (int i = 0; i < number_arrivals; i++) {
                Plane current_plane(flight_number++, current_time, arriving);
                if (small_airport.can_land(current_plane) != success)
                        current_plane.refuse( );
        }
        int number_departures = variable.poisson(departure_rate);
        for (int j = 0; j < number_departures; j++) {
                Plane current_plane(flight_number++, current_time, departing);
                if (small_airport.can_depart(current_plane) != success)
                        current_plane.refuse( );
        }
        Plane moving_plane;
        switch (small_airport.activity(current_time, moving_plane)) {
                case land: moving_plane.land(current_time); break;
                case takeoff: moving_plane.fly(current_time); break;
                case idle: run_idle(current_time);
        }
}
```

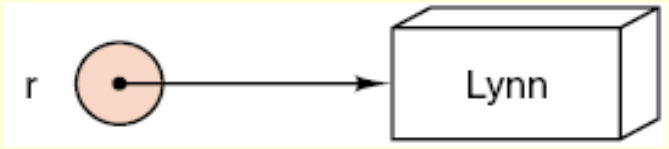
CẤU TRÚC DỮ LIỆU VÀ GIẢI THUẬT (501040)

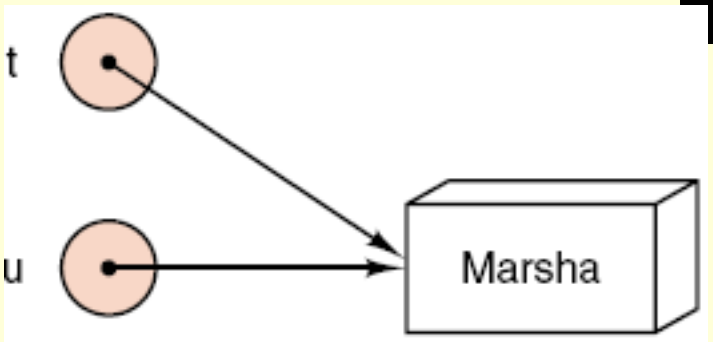
Chương 4: Stack và Queue liên kết

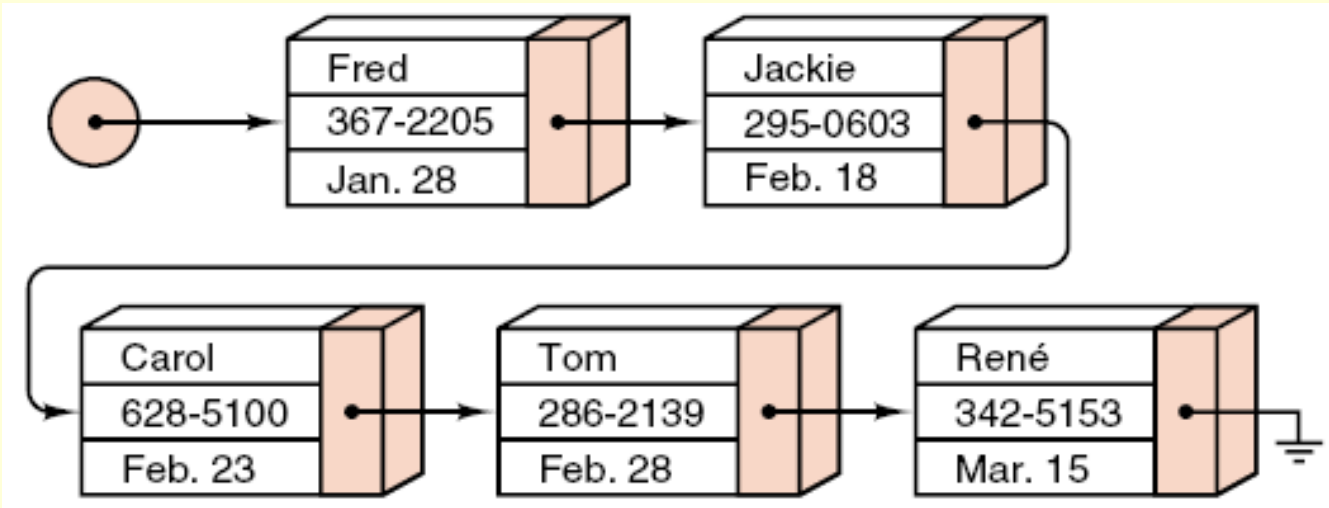


Con trỏ

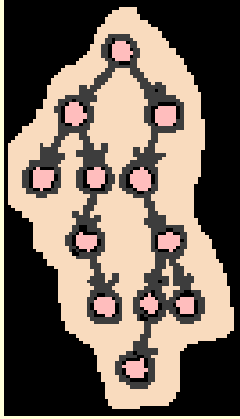



r
Lynn
s

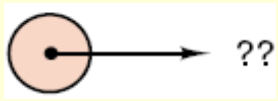

t
u
Marsha

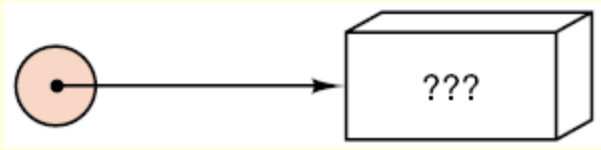

Fred
367-2205
Jan. 28
Jackie
295-0603
Feb. 18
Carol
628-5100
Feb. 23
Tom
286-2139
Feb. 28
René
342-5153
Mar. 15

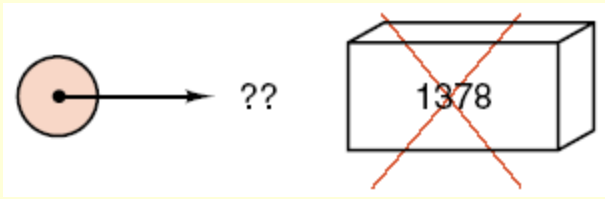
Biểu diễn con trỏ bằng C++

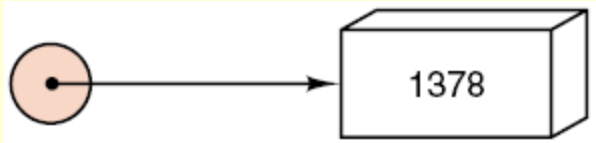


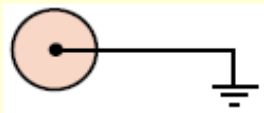
- Khai báo biến:
  - Item * item_ptr1, * item_ptr2;
- Tạo mới đối tượng:
  - item_ptr1 = **new** Item;
- Hủy bỏ đối tượng:
  - **delete** item_ptr1;
- Sử dụng:
  - *item_ptr1 = 1378;
  - cout << Student_ptr **->** StudentID;
- Con trỏ NULL:
  - item_ptr2 = NULL;



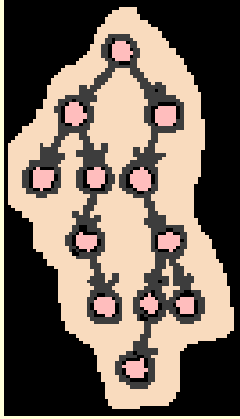








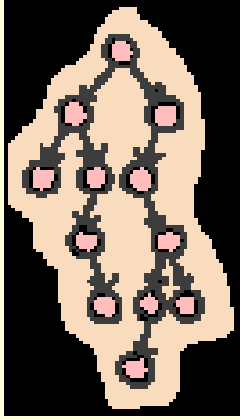
Sử dụng con trỏ trong C++

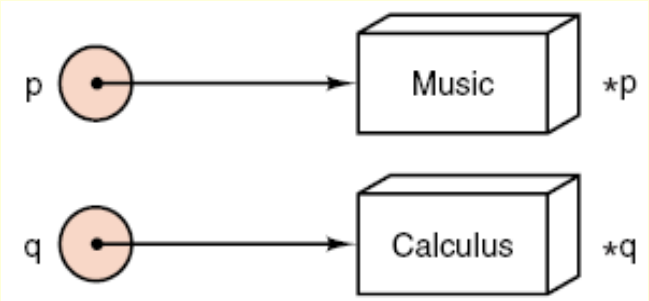


- Địa chỉ của biến:
  - Biến: int_ptr = &x;
  - Array: arr_ptr = an_array;
- Dynamic array:
  - Trong C++, array có thể được quản lý như một con trỏ và ngược lại
  - Ví dụ:

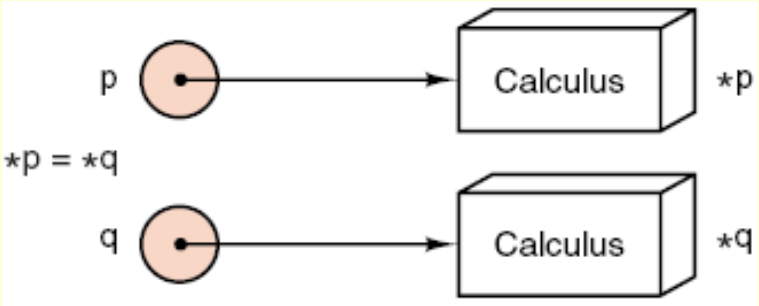
```
int arr[3] = {0, 1, 2, 3};
int *arr_ptr = arr;
//in ra 0 – 1 – 2
cout << *arr_ptr << " - " << *(arr_ptr + 1) << " - " << arr_ptr[2];
```

Gán con trỏ trong C++

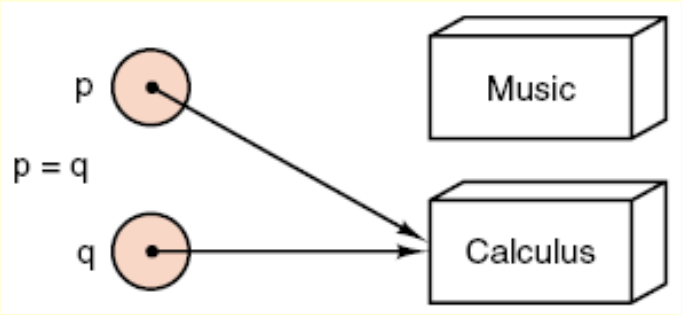




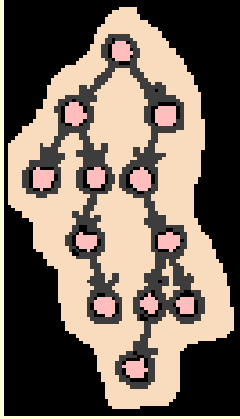
Gán nội dung: bình thường



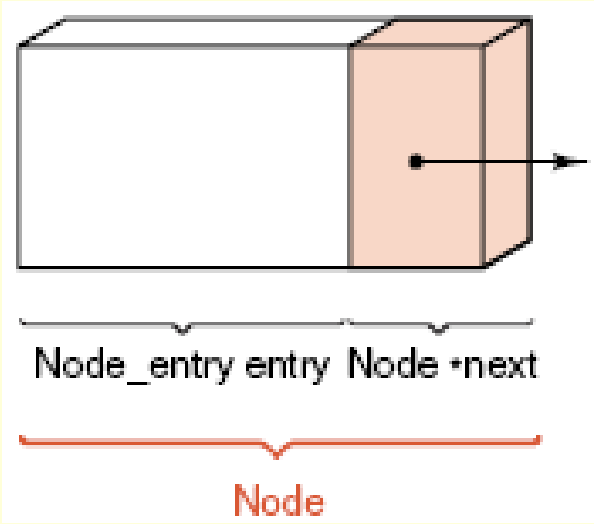
Gán con trỏ: nguy hiểm



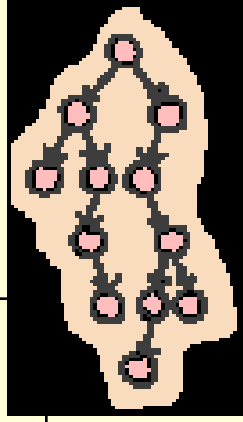
Thiết kế node liên kết



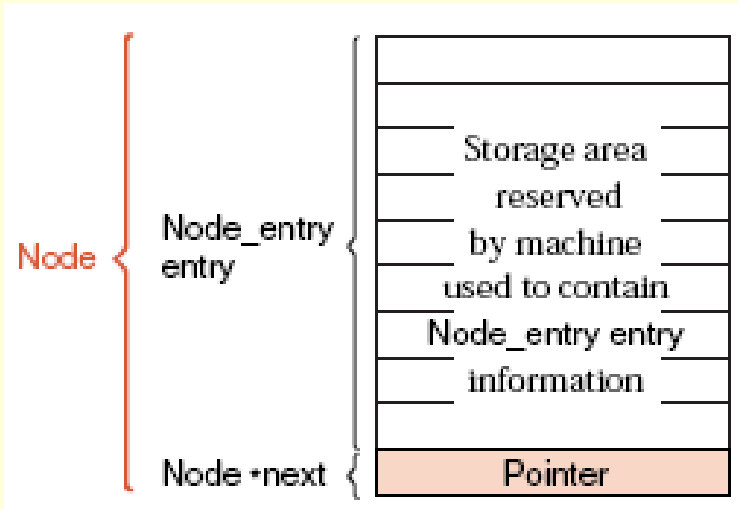
- Cần:
  - Dữ liệu
  - Con trỏ để trỏ đến node sau
  - Constructor



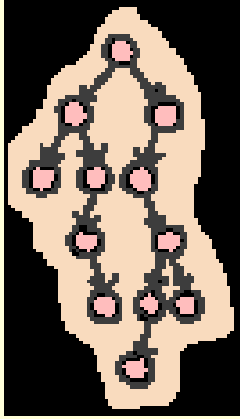
Thiết kế node liên kết bằng C++



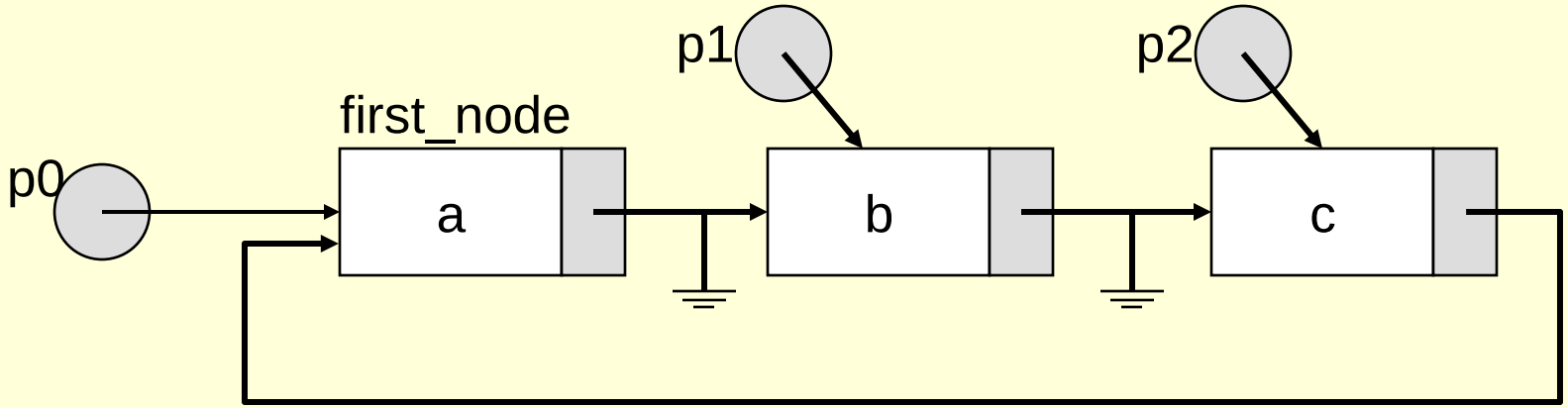
```
template <class Entry>
struct Node {
        Entry entry;                                    // data members
        Node<Entry> *next;
        Node( );                                        // constructors
        Node(Entry  item, Node<Entry> *add on = NULL);
};
```



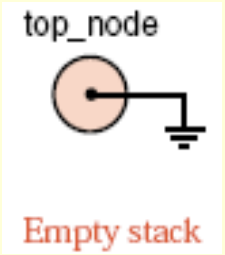
Ví dụ với node liên kết

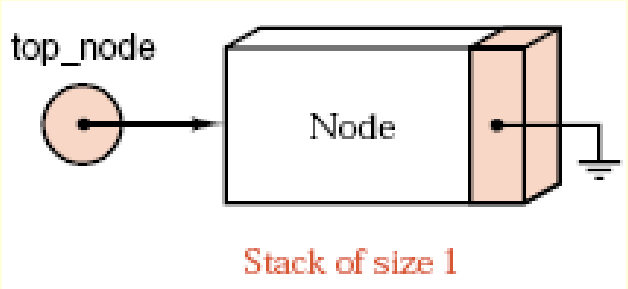


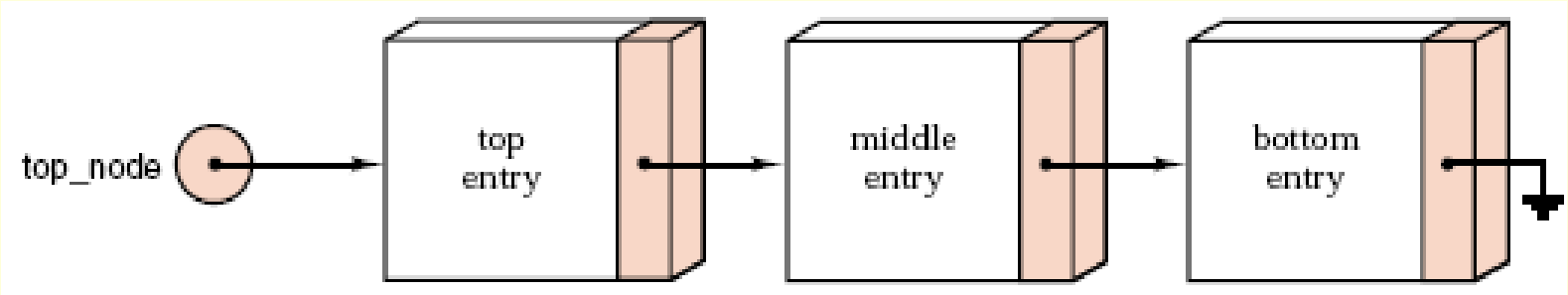
```
Node<char> first_node('a');
Node<char> *p0 = &first_node;
Node<char> *p1 = new Node<char>('b');
p0->next = p1;
Node<char> *p2 = new Node<char>('c', p0);
p1->next = p2;
```



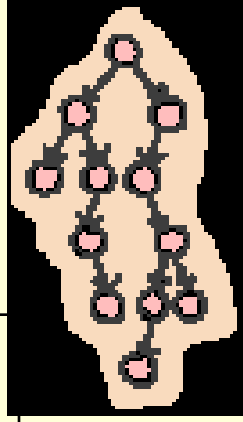
Stack liên kết





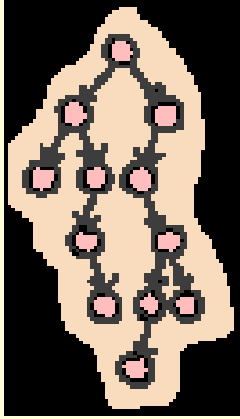


Khai báo stack liên kết

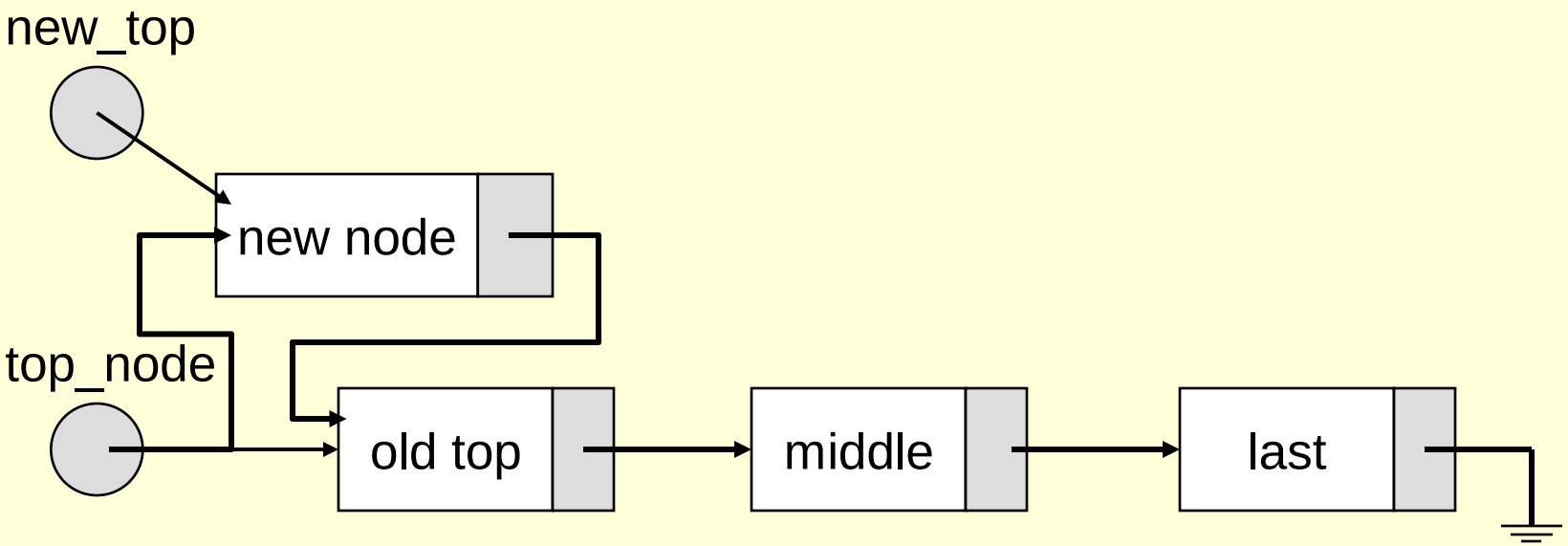


```
template <class Entry>
class Stack {
public:
        Stack( );
        bool empty( ) const;
        Error_code push(const Entry &item);
        Error_code pop( );
        Error_code top(Entry &item) const;
        Stack(const Stack<Entry> &copy);
        ~Stack();
        void operator=(const Stack<Entry> &copy);
protected:
        Node<Entry> *top_node;
};
```

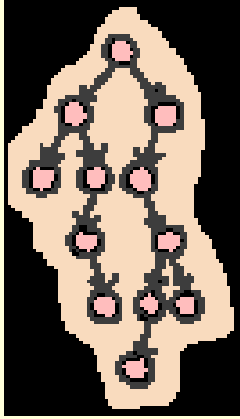
Thêm vào một stack liên kết



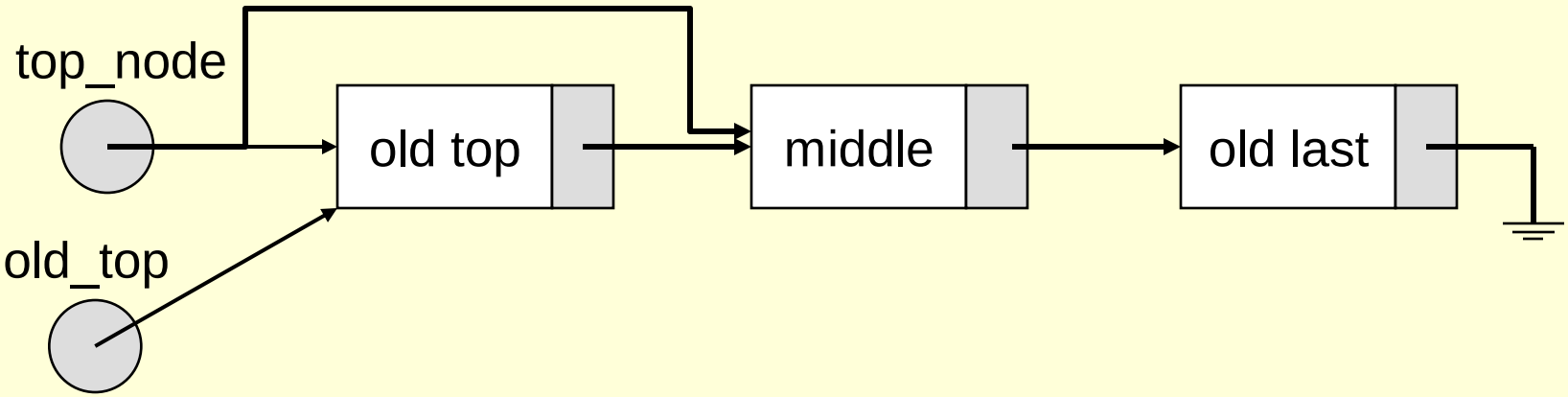
- Giải thuật
  1. Tạo ra một node mới với giá trị cần thêm vào
  2. Trỏ nó đến đỉnh hiện tại của stack
  3. Trỏ đỉnh của stack vào node mới



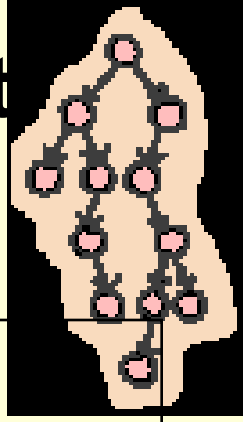
Bỏ đỉnh của một stack liên kết



- Giải thuật:
  1. Gán một con trỏ để giữ đỉnh của stack
  2. Trỏ đỉnh của stack vào node ngay sau đỉnh hiện tại
  3. Xóa node cũ đi



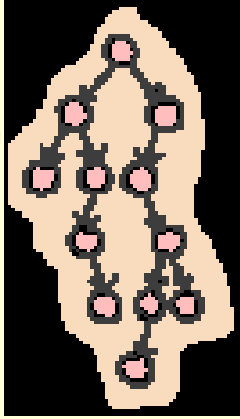
Thêm/Bỏ đỉnh của một stack liên kết – Mã C++



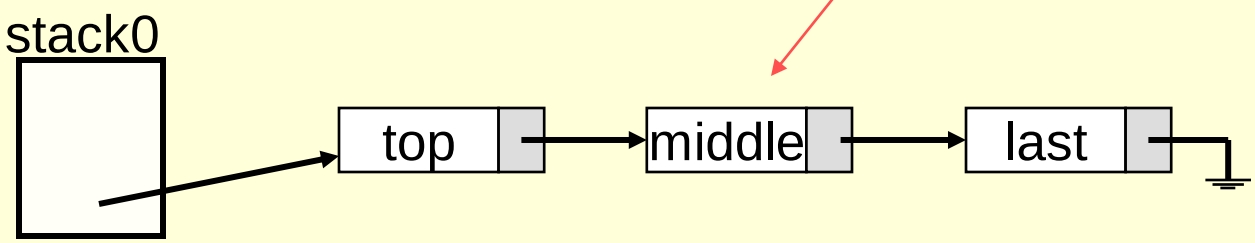
```
template <class Entry>
Error_code push(const Entry &item) {
        Node<Entry> *new_top = new Node<Entry>(item, top_node);
        if (new_top == NULL) return overflow;
        top_node = new_top;
}
```

```
template <class Entry>
Error_code pop( ) {
        Node<Entry> *old_top = top_node;
        if (top_node == NULL) return underflow;
        top_node = old_top->next;
        delete old_top;
}
```

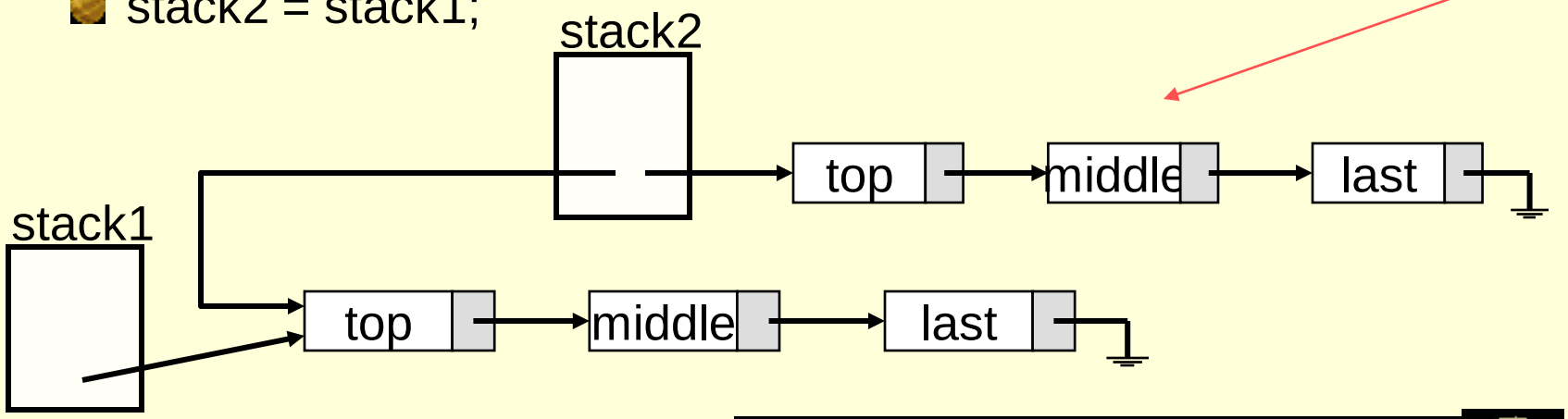
Sự không an toàn con trỏ trong C++



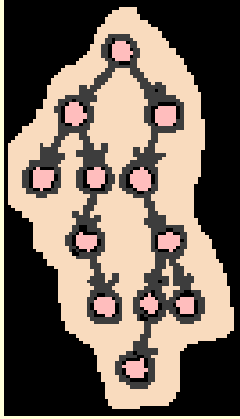
- Kết thúc biến stack nhưng bộ nhớ còn lại:
  - delete stack0;



- Gán hai stack: cả hai dùng chung một vùng dữ liệu
  - stack2 = stack1;

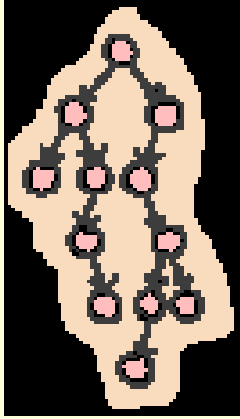


Đảm bảo an toàn con trỏ trong C++



- Destructor:
  - Sẽ được gọi ngay trước khi đối tượng kết thúc thời gian sống
  - Dùng xóa hết vùng dữ liệu
- Copy constructor:
  - Sẽ được gọi khi khởi tạo biến lúc khai báo, hoặc truyền dữ liệu bằng tham trị
  - Sao chép nguồn thành một vùng dữ liệu mới
- Assignment operator:
  - Sẽ được gọi khi gán đối tượng này vào đối tượng khác
  - Xóa vùng dữ liệu của đích và đồng thời sao chép nguồn thành một vùng dữ liệu mới

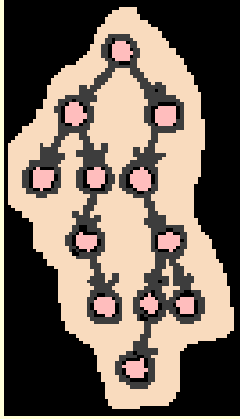
Xóa vùng dữ liệu đang có



- Giải thuật:
  1. Trong khi stack chưa rỗng
     1.1. Bỏ đỉnh của stack
- Mã C++:

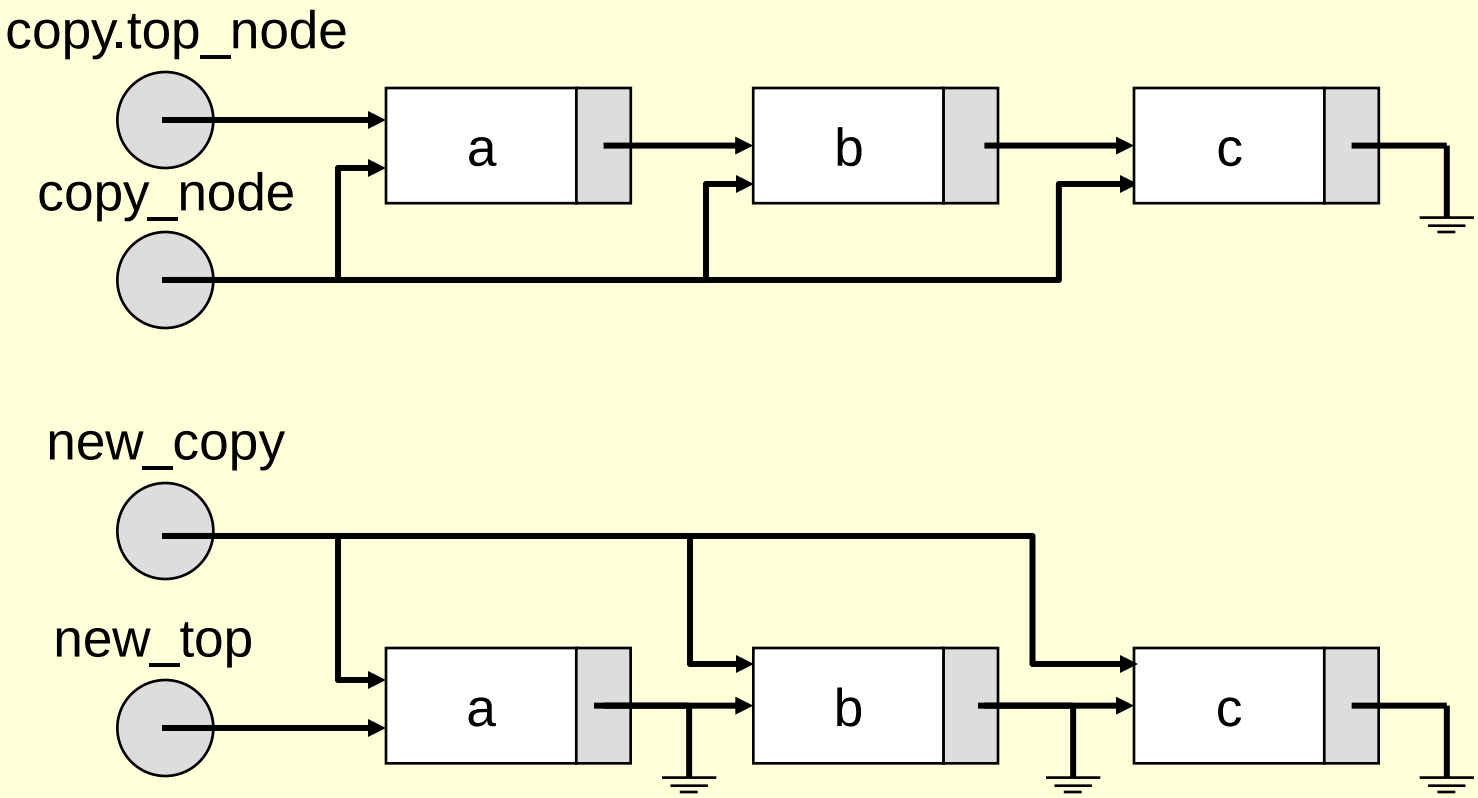
```
while (!empty())
  pop();
```

Sao chép vùng dữ liệu

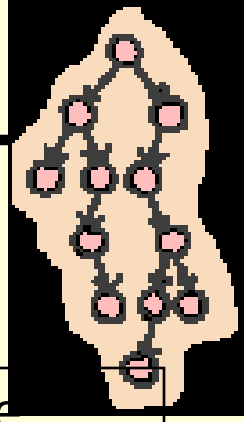


- Giải thuật:
  1. Tạo một đỉnh của danh sách mới với dữ liệu của đỉnh nguồn
  2. Giữ một con trỏ đuôi chỉ vào cuối danh sách mới
  2. Duyệt qua danh sách nguồn
     - 2.1. Tạo một node mới với dữ liệu từ node nguồn hiện tại
     - 2.2. Nối vào cuối danh sách mới
     - 2.3. Con trỏ đuôi là node mới

Sao chép vùng dữ liệu – Ví dụ

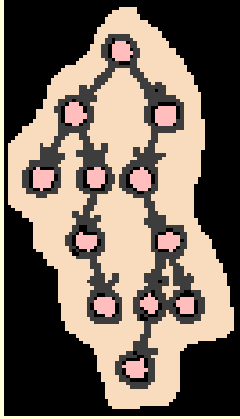


Sao chép vùng dữ liệu – Mã C++

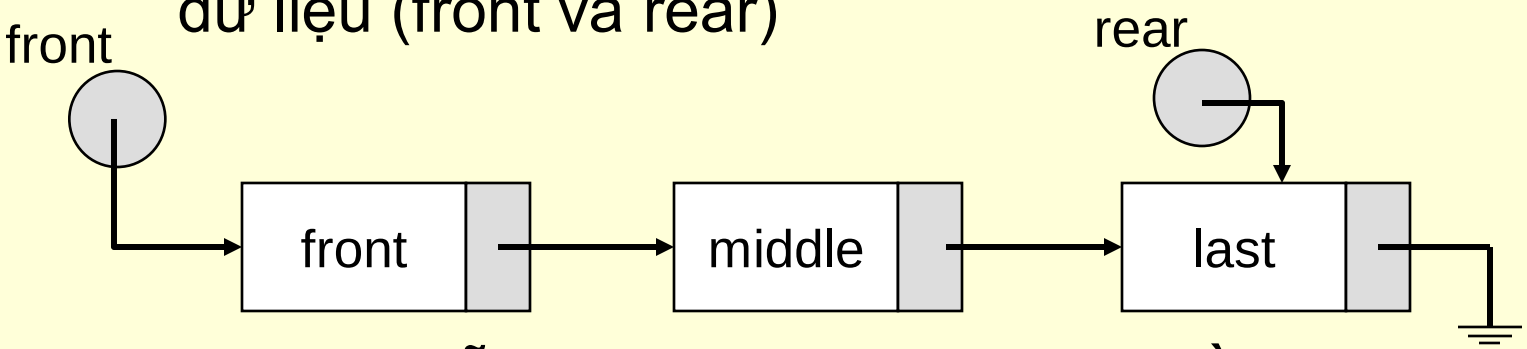


```
Node<Entry> *new_top, *new_copy, *copy_node = copy.top_node;
if (copy_node == NULL) new_top = NULL;
else {
    // Sao chép vùng dữ liệu thành danh sách mới
    new_copy = new_top = new Node<Entry>(copy_node->entry);
    while (copy_node->next != NULL) {
        copy_node = copy_node->next;
        new_copy->next = new Node<Entry>(copy_node->entry);
        new_copy = new_copy->next;
    }
}
clear();                    //xóa rỗng dữ liệu hiện tại trước
top_node = new_top;         // thay thế dữ liệu bằng danh sách mới.
```

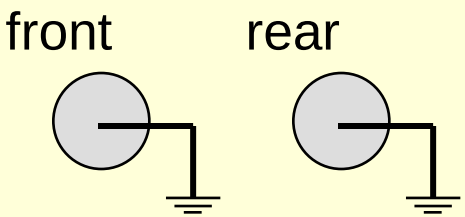
Queue liên kết



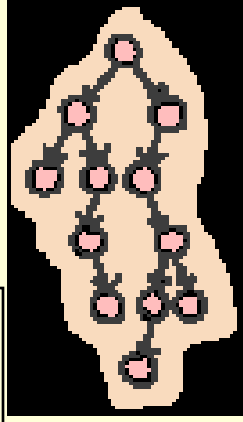
- Thiết kế:
  - Dùng hai con trỏ chỉ đến đầu và cuối của danh sách dữ liệu (front và rear)



  - Khởi tạo rỗng: gán cả front và rear về NULL



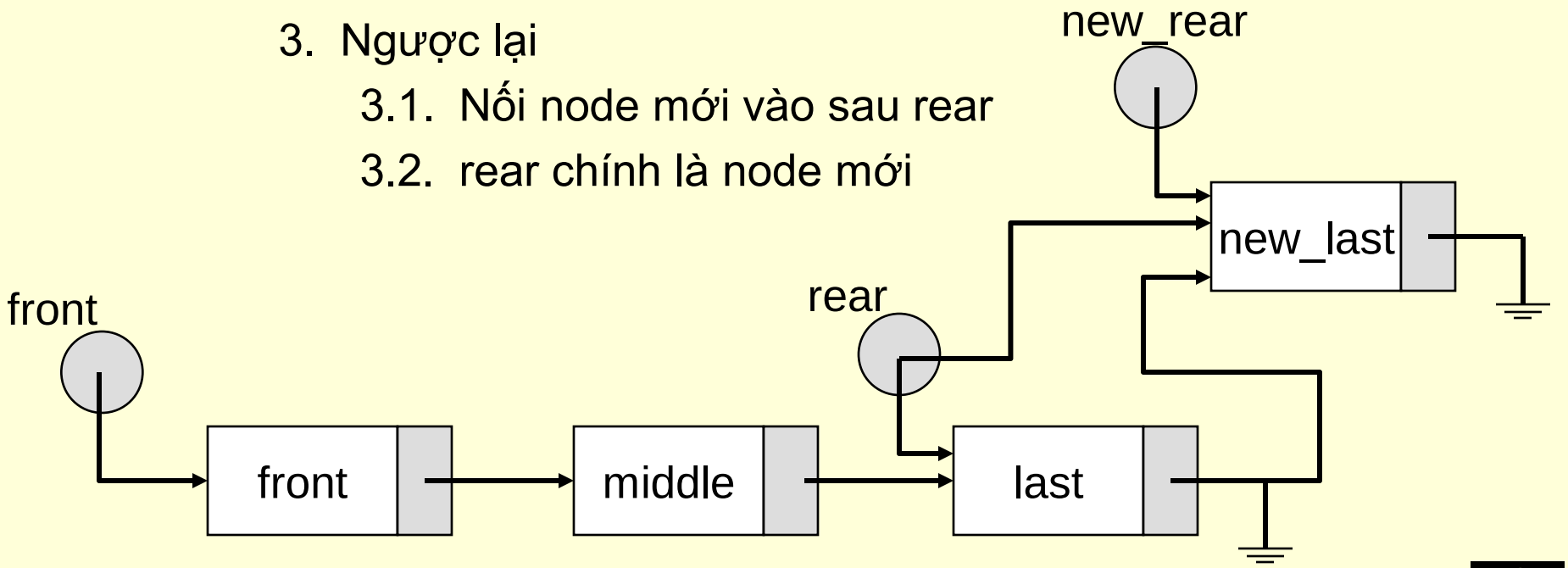
Khai báo Queue liên kết



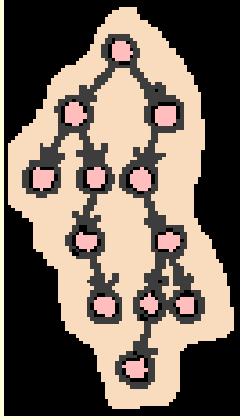
```
template <class Entry>
class Queue {
public:
        Queue( );
        bool empty( ) const;
        Error_code append(const Entry &item);
        Error_code serve( );
        Error_code retrieve(Entry &item) const;
        ~Queue( );
        Queue(const Queue<Entry> &original);
        void operator = (const Queue<Entry> &original);
protected:
        Node<Entry> *front, *rear;
};
```

Thêm phần tử vào một queue liên kết

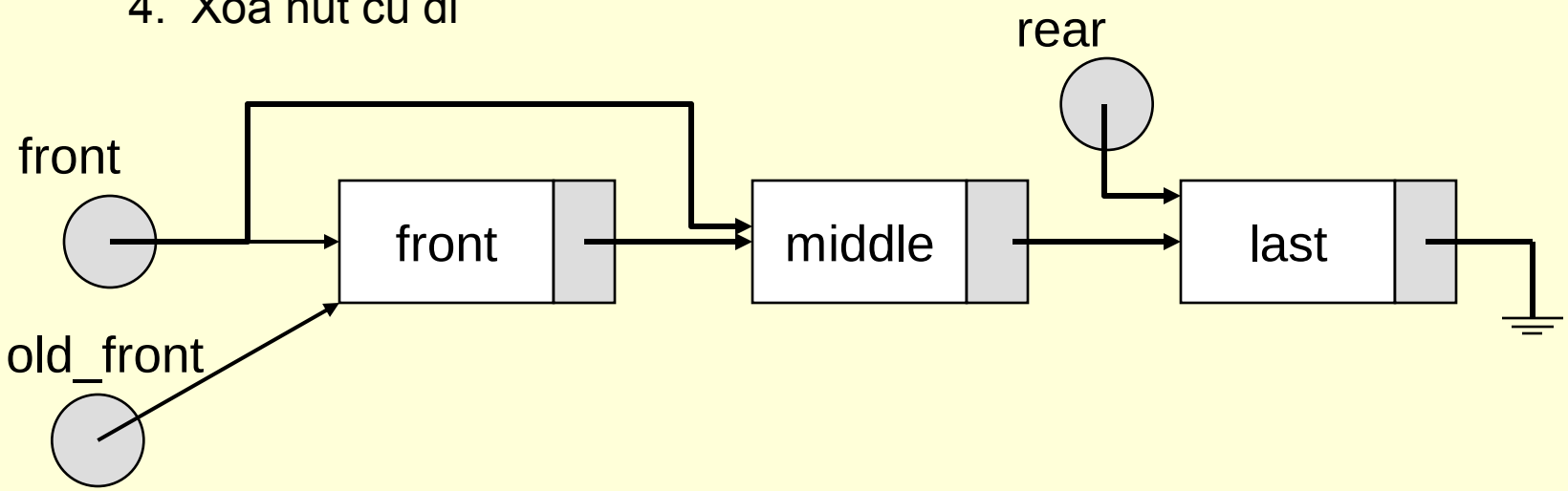
- Giải thuật:
  1. Tạo một node mới với dữ liệu cần thêm vào
  2. Nếu queue đang rỗng
     2.1. front và rear là node mới
  3. Ngược lại
     3.1. Nối node mới vào sau rear
     3.2. rear chính là node mới



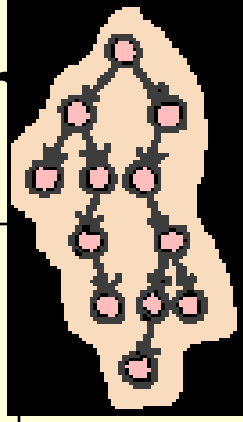
Bỏ phần tử khỏi một queue liên kết



- Giải thuật:
  1. Dùng một con trỏ để giữ lại front hiện tại
  2. Nếu queue có một phần tử
     2.1. Gán front và rear về NULL
  3. Ngược lại
     3.1. Trỏ front đến nút kế sau
  4. Xóa nút cũ đi



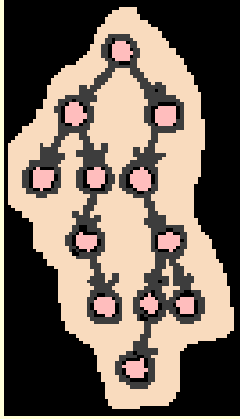
Thêm/Bỏ phần tử của một queue liên kết – Mã C++



```
template <class Entry>
Error_code append(const Entry &item) {
        Node<Entry> *new_rear = new Node<Entry>(item);
        if (new_rear == NULL) return overflow;
        if (rear == NULL) front = rear = new_rear;
        else {  rear->next = new_rear; rear = new_rear; }
        return success;
}
```

```
template <class Entry>
Error_code serve() {
        if (front == NULL) return underflow;
        Node<Entry> *old_front = front;
        front = old_front->next;
        if (front == NULL) rear = NULL;
        delete old_front;
        return success;
}
```

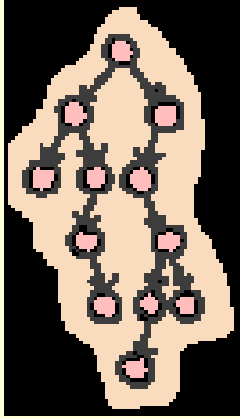
Kích thước của một queue liên kết



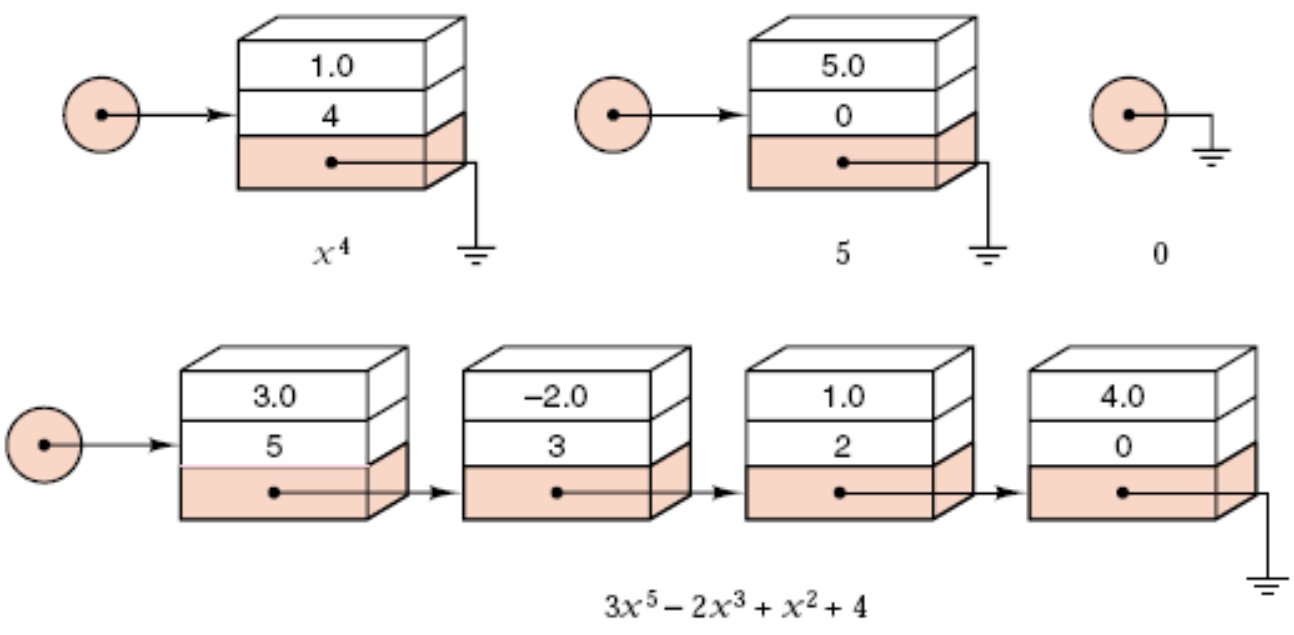
- Giải thuật:
  1. Khởi tạo biến đếm là 0
  2. Duyệt qua danh sách
     2.1. Đếm tăng số phần tử lên 1
- Mã C++:

```
Node<Entry> *window = front;
int count = 0;
while (window != NULL) {
   window = window->next;
   count++;
}
return count;
```

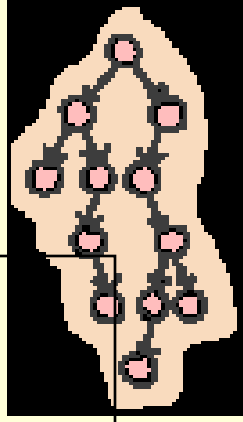
Ứng dụng: tính toán đa thức



- Dùng lại bài reverse Polish calculator
- Thiết kế cấu trúc dữ liệu cho đa thức:
  - Một bản ghi có thành phần mũ và hệ số
  - Một danh sách các bản ghi theo thứ tự giảm của số mũ
  - Có thể dùng queue



Giải thuật cộng hai đa thức 1



Algorithm Equals_sum1

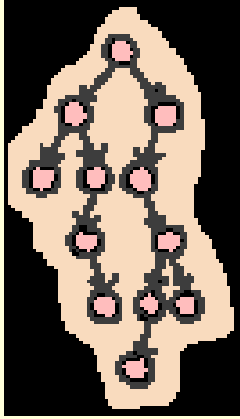
Input: p,q là hai đa thức

Output: đa thức tổng

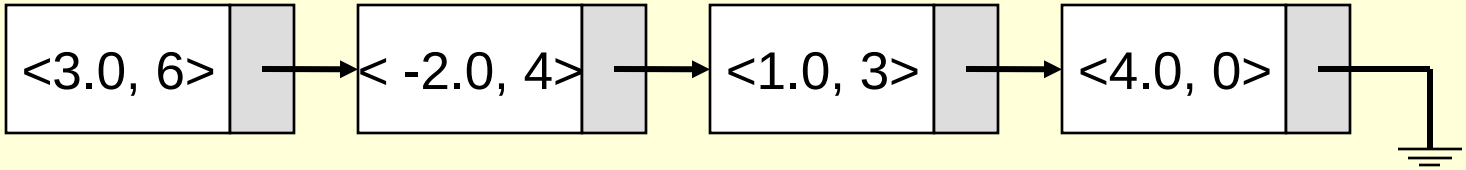
1. Trong khi p và q chưa rỗng
   1.1. Lấy phần tử front của p và q thành p_term, q_term
   1.2. Nếu bậc của p_term lớn (hoặc nhỏ) hơn bậc của q_term
      1.2.1. Đẩy p_term (hoặc q_term) vào kết quả
      1.2.2. Bỏ phần tử đầu trong p (hoăc trong q)
   1.3. Ngược lại
      1.3.1. Tính hệ số mới cho số hạng này
      1.3.2. Đẩy vào kết quả
2. Nếu p (hoặc q) chưa rỗng
   2.1. Đẩy toàn bộ p (hoặc q) vào kết quả

End Equals_sum1

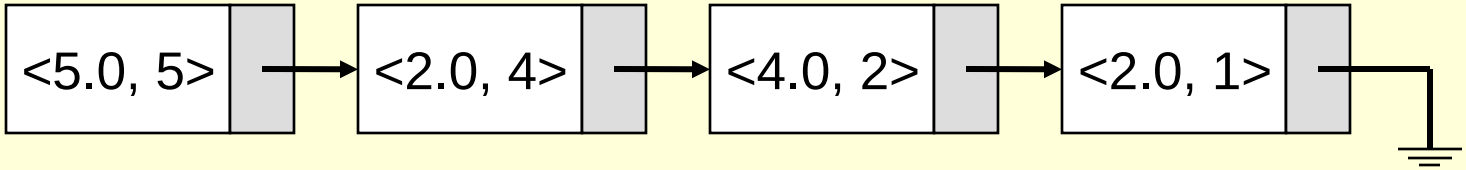
Ví dụ cộng hai đa thức bằng giải thuật 1



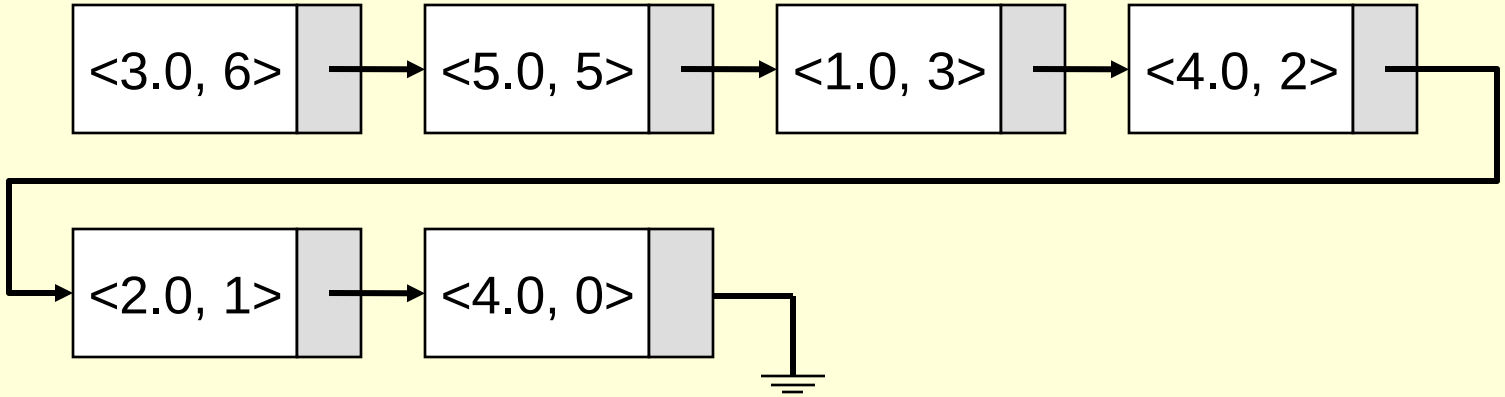
$p = 3x^6 – 2x^4 + x^3 + 4$



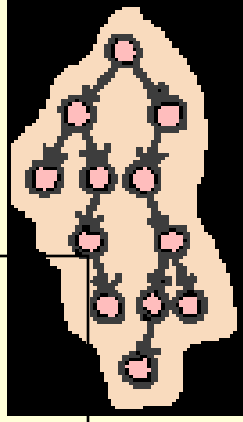
$q = 5x^5 + 2x^4 + 4x^2 + 2x$



$p + q = 3x^6 + 5x^5 + x^3 + 4x^2 + 2x + 4$

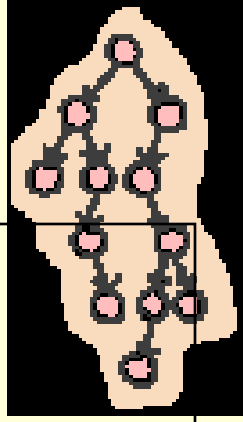


Mã C++ cộng hai đa thức 1



```
Term p_term, q_term;
while (!p.empty( ) && !q.empty( )) {
    p.retrieve(p_term); q.retrieve(q_term);
    if (p_tem.degree > q_term.degree) {
        p.serve(); append(p_term);
    } else if (q_term.degree > p_term.degree) {
        q.serve(); append(q_term);
    } else {
        p.serve(); q.serve();
        if (p_term.coefficient + q_term.coefficient != 0) {
            Term answer_term(p_term.degree,
                                    p_term.coefficient + q_term.coefficient);
            append(answer_term);
        } } }
while (!p.empty()) { p.serve_and_retrieve(p_term); append(p_term); }
while (!q.empty()) { q.serve_and_retrieve(q_term); append(q_term); }
```

Giải thuật cộng hai đa thức 2



Algorithm Bac_da_thuc
Input: đa thức
Output: bậc của đa thức

1. Nếu đa thức rỗng
   1.1. Trả về -1
2. Trả về bậc của phần tử đầu

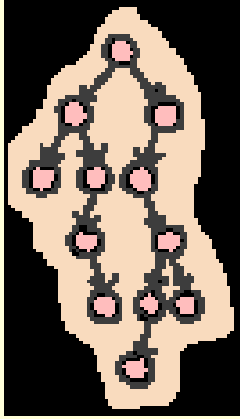
End Bac_da_thuc

Algorithm Equals_sum2
Input: p,q là hai đa thức
Output: đa thức tổng

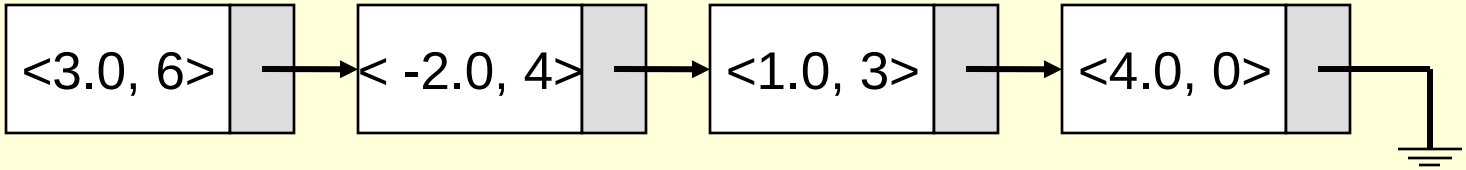
1. Trong khi p hoặc q chưa rỗng
   1.1. Nếu bậc của p lớn hơn bậc của q
      1.1.1. Lấy từ p thành term
      1.1.2. Đẩy term vào kết quả
   1.2. Nếu bậc của q lớn hơn bậc của p
      1.2.1. Lấy từ q thành term
      1.2.2. Đẩy term vào kết quả
   1.3. Ngược lại
      1.3.1. Lấy p_term, q_term từ p và q
      1.3.2. Tính tổng hai hệ số
      1.3.3. Nếu hệ số kết quả khác không
         1.3.3.1. Đẩy vào kết quả

End Equals_sum2

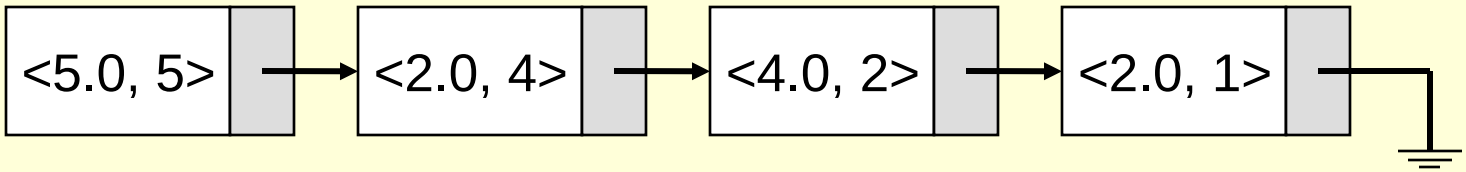
Ví dụ cộng hai đa thức bằng giải thuật 2



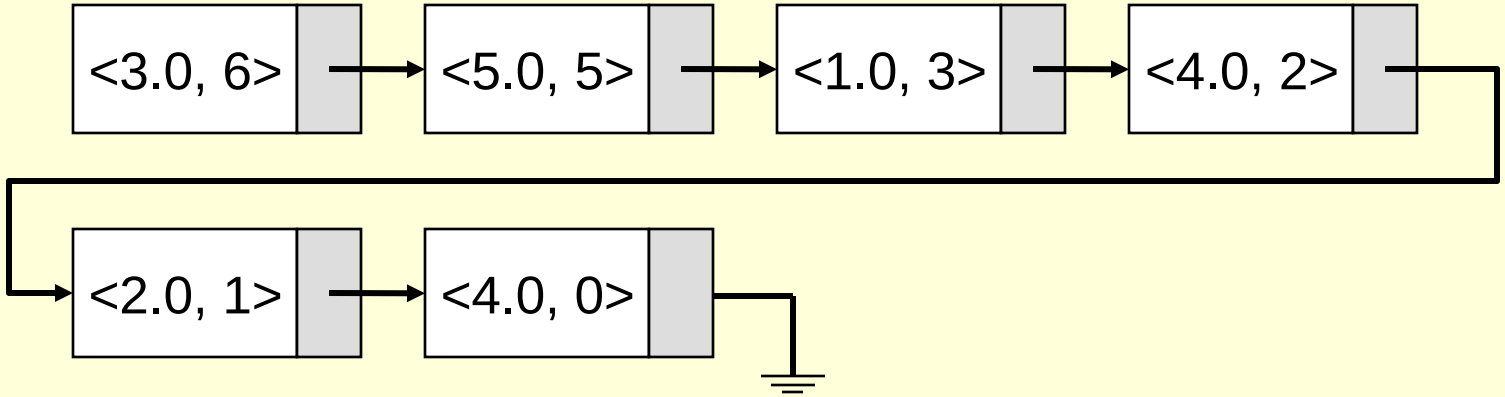
$p = 3x^6 – 2x^4 + x^3 + 4$ degree(p) = 6



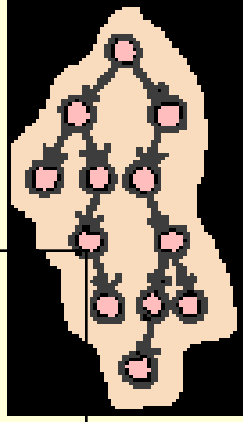
$q = 5x^5 + 2x^4 + 4x^2 + 2x$ degree(p) = 5



$p + q = 3x^6 + 5x^5 + x^3 + 4x^2 + 2x + 4$

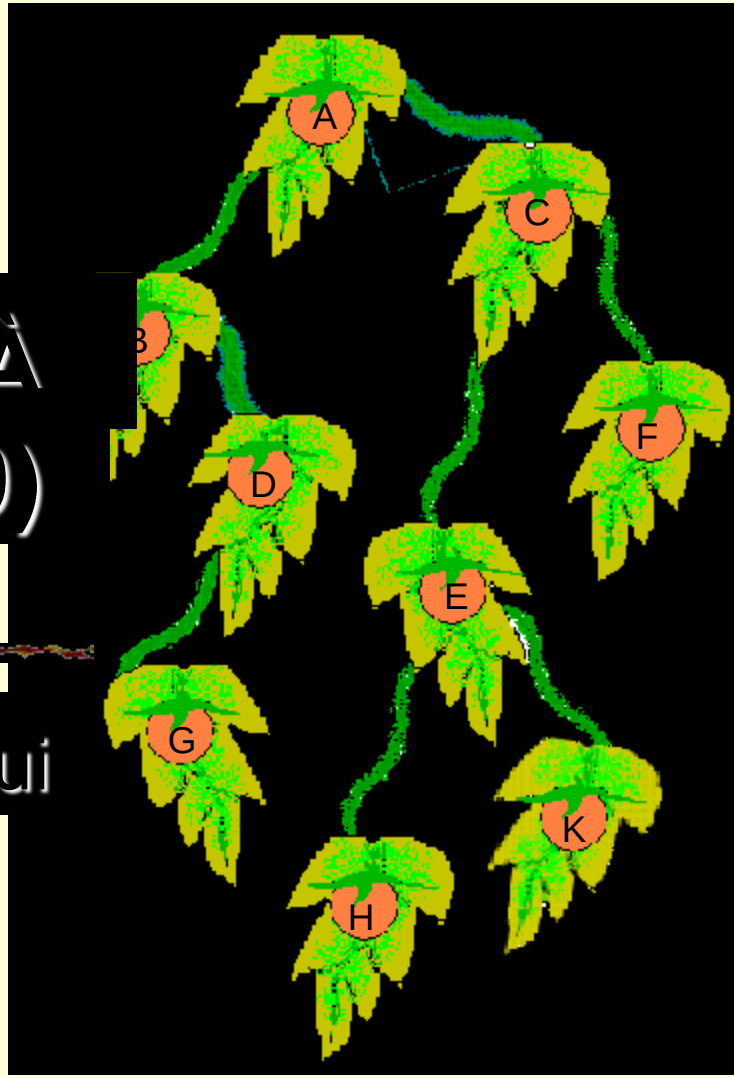


Mã C++ cộng hai đa thức 2



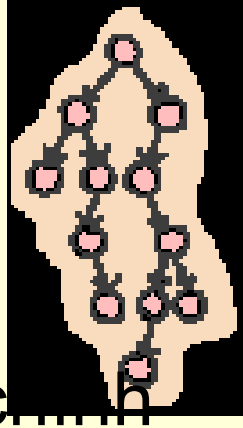
```
while (!p.empty( ) || !q.empty( )) {
    Term p_term, q_term;
    if (p.degree( ) > q.degree( )) {
        p.serve_and_retrieve(p_term);
        append(p_term);
    } else if (q.degree( ) > p.degree( )) {
        q.serve_and_retrieve(q_term);
        append(q_term);
    } else {
        p.serve_and_retrieve(p_term);
        q.serve_and_retrieve(q_term);
        if (p_term.coefficient + q_term.coefficient != 0) {
            Term answer_term(p_term.degree,
                                p_term.coefficient + q_term.coefficient);
            append(answer_term);
        }   }   }
```

CẤU TRÚC DỮ LIỆU VÀ GIẢI THUẬT (501040)



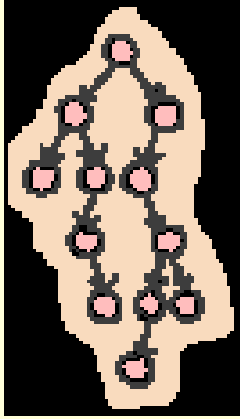
Chương 5: Đệ qui

Khái niệm đệ qui



- Khái niệm (định nghĩa) đệ qui có dùng lại chính nó.
  - Ví dụ: giai thừa của n là 1 nếu n là 0 hoặc là n nhân cho giai thừa của n-1 nếu n > 0
- Quá trình đệ qui gồm 2 phần:
  - Trường hợp cơ sở (base case)
  - Trường hợp đệ qui: cố gắng tiến về trường hợp cơ sở
- Ví dụ trên:
  - Giai thừa của n là 1 nếu n là 0
  - Giai thừa của n là n * (giai thừa của n-1) nếu n>0

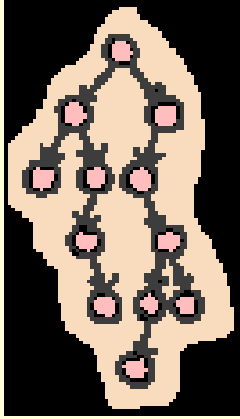
Tính giai thừa

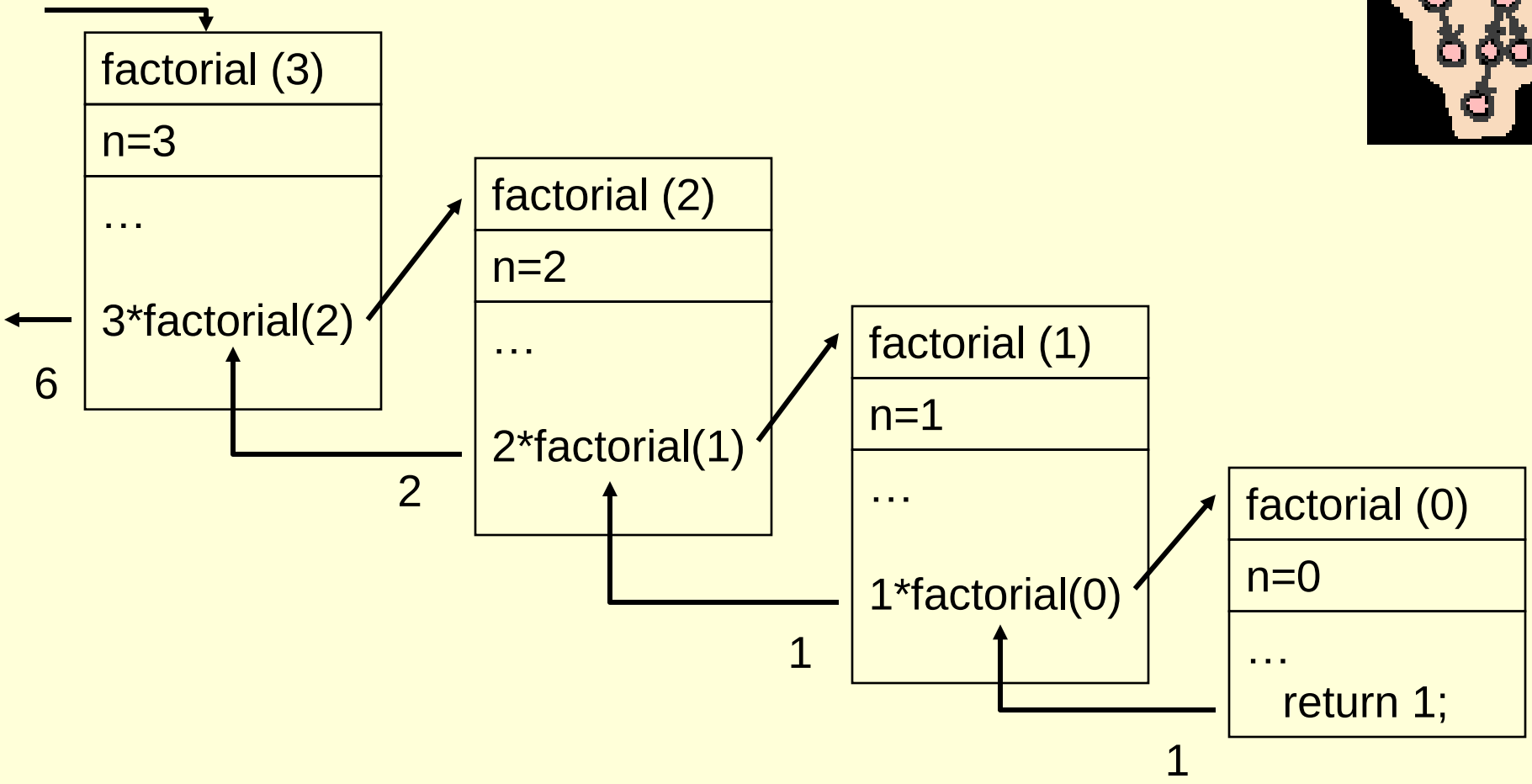


- Định nghĩa không đệ qui:
  - n! = n * (n-1) * … * 1
- Định nghĩa đệ qui:
  - $n! = \begin{cases} 1 & \text{nếu } n=0 \\ n * (n-1)! & \text{nếu } n>0 \end{cases}$
- Mã C++:

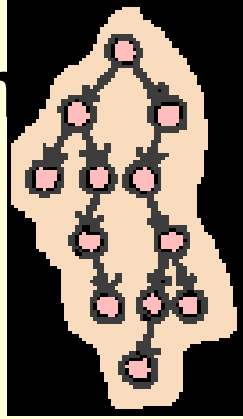
```
int factorial(int n) {
    if (n==0)  return 1;
    else       return (n * factorial(n - 1));
}
```

Thi hành hàm tính giai thừa

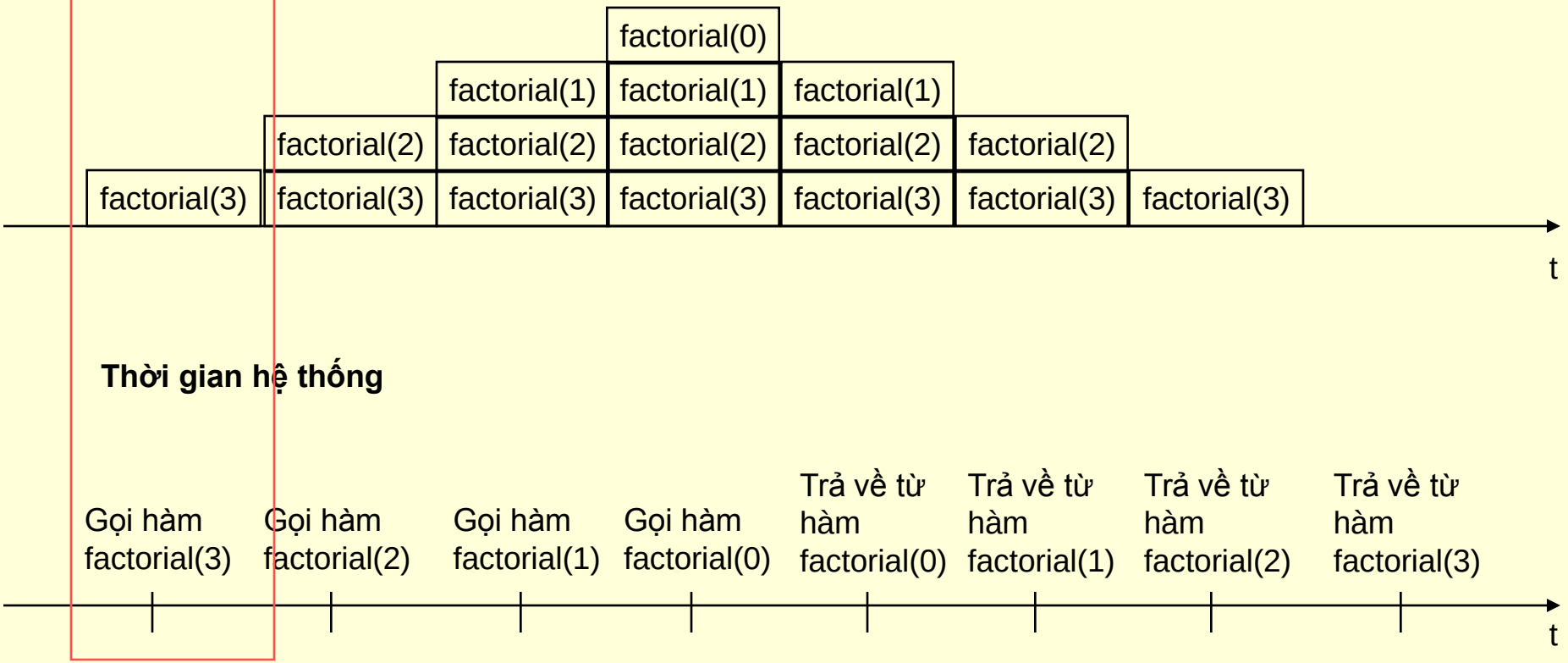




Trạng thái hệ thống khi thi hành hàm tính giai thừa

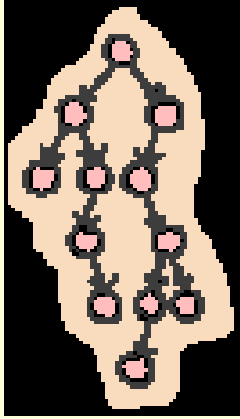


Stack hệ thống

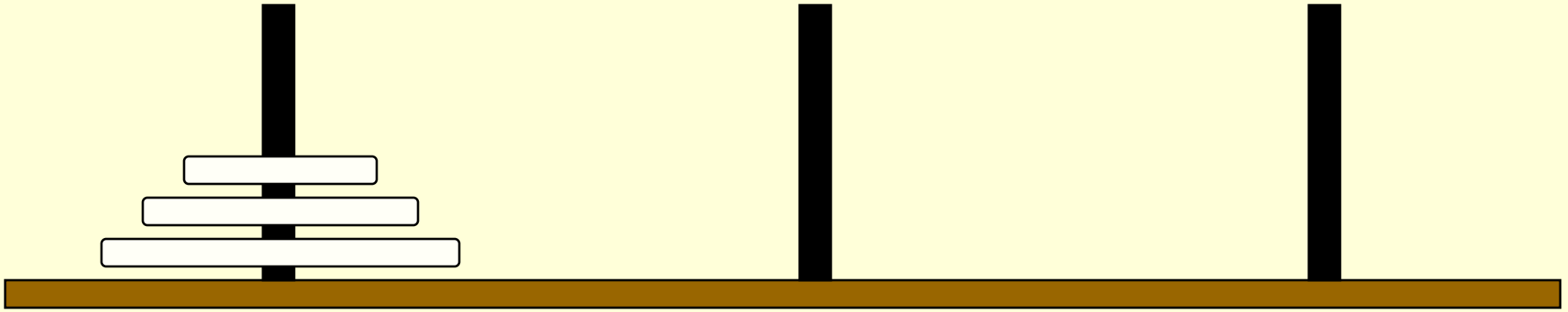


Thời gian hệ thống

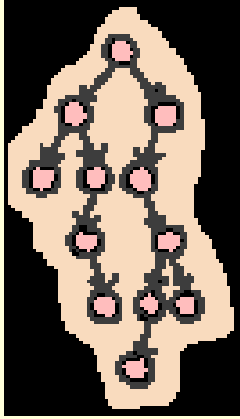
Bài toán Tháp Hà nội



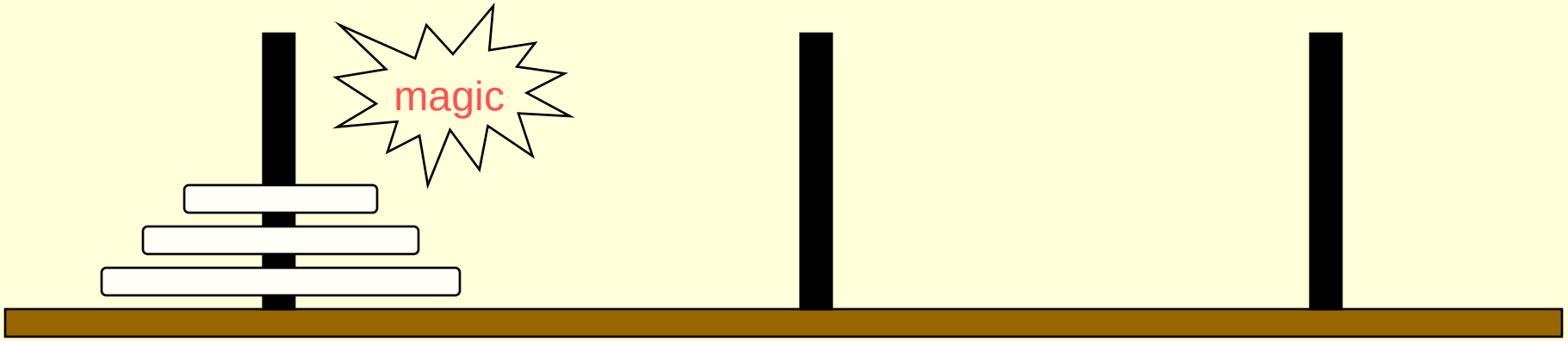
- Luật:
  - Di chuyển mỗi lần một đĩa
  - Không được đặt đĩa lớn lên trên đĩa nhỏ



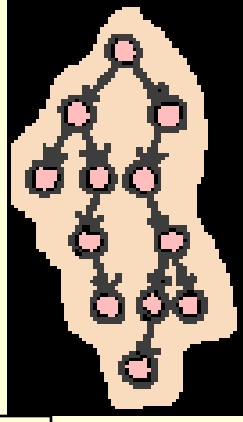
Bài toán Tháp Hà nội – Thiết kế hàm



- Hàm đệ qui:
  - Chuyển (count-1) đĩa trên đỉnh của cột start sang cột temp
  - Chuyển 1 đĩa (cuối cùng) của cột start sang cột finish
  - Chuyển count-1 đĩa từ cột temp sang cột finish

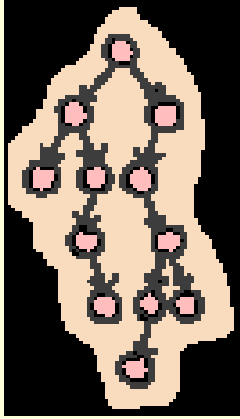


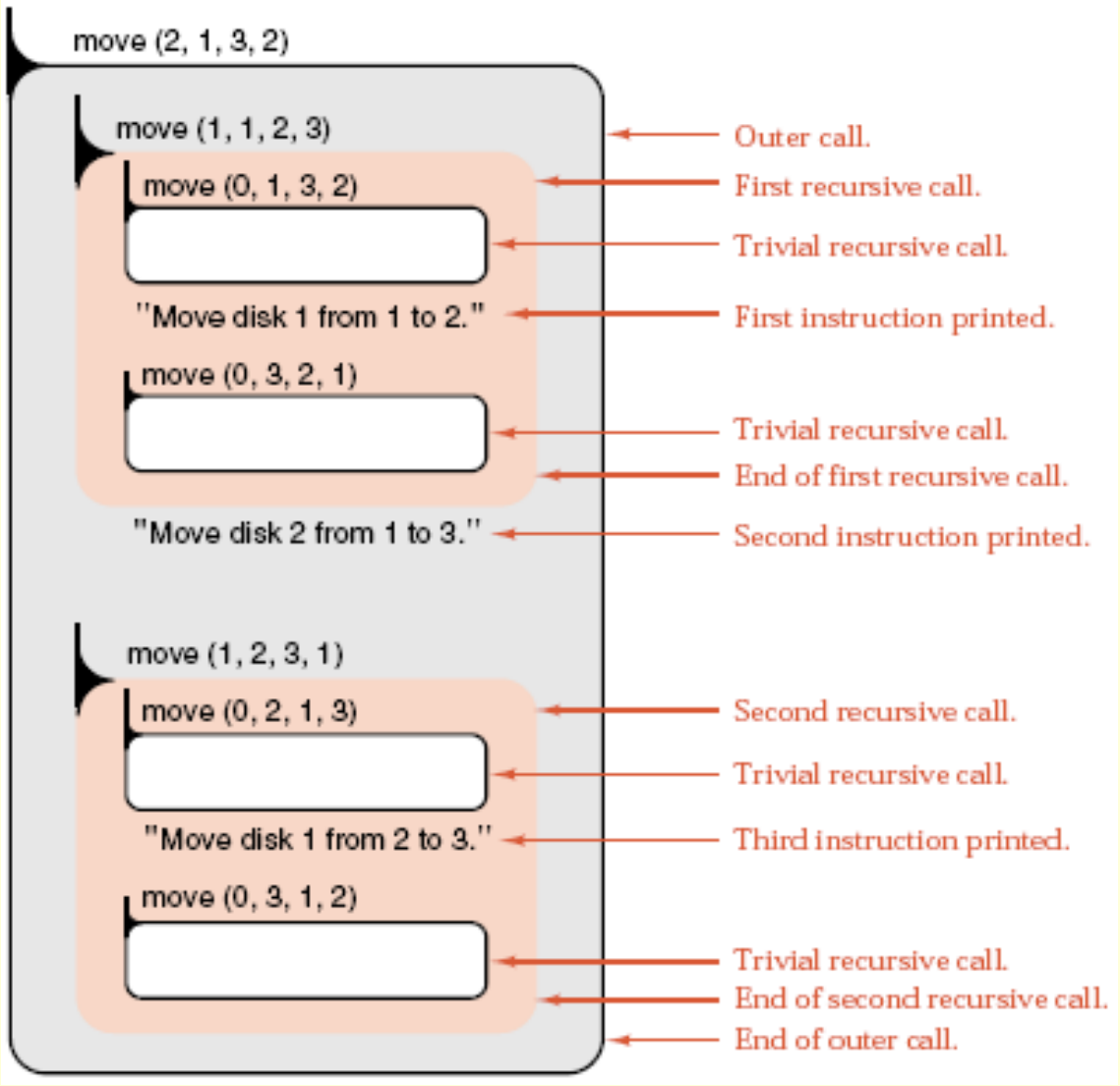
Bài toán Tháp Hà nội – Mã C++



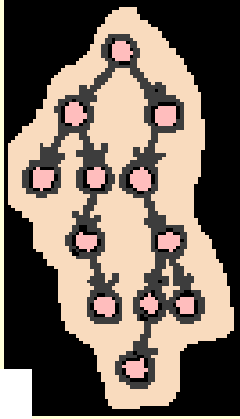
```
void move(int count, int start, int finish, int temp) {
        if (count > 0) {
                move(count − 1, start, temp, finish);
                cout << "Move disk " << count << " from " <<
          start
                        << " to " << finish << "." << endl;
                move(count − 1, temp, finish, start);
        }
}
```

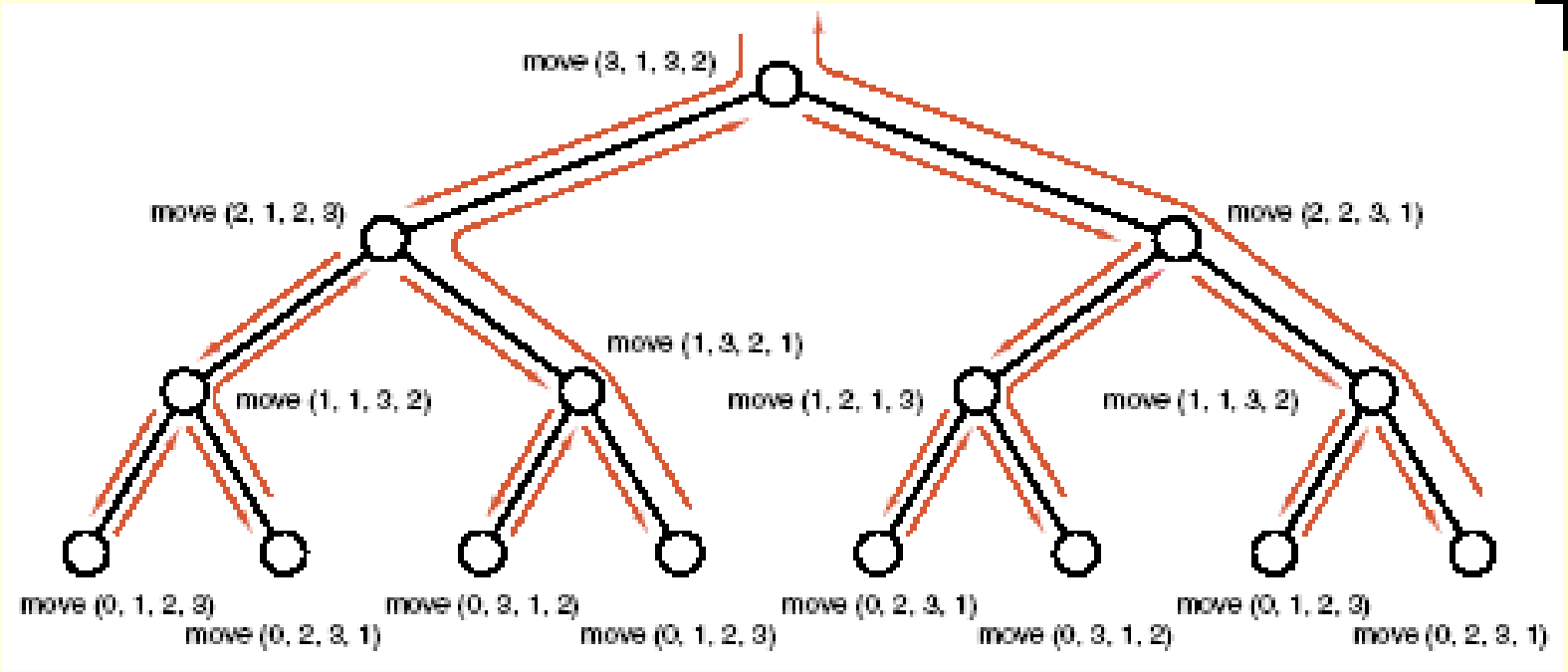
Bài toán Tháp Hà nội – Thi hành



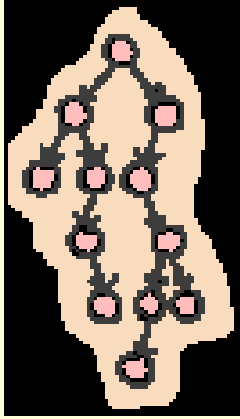


Bài toán Tháp Hà nội – Cây đệ qui



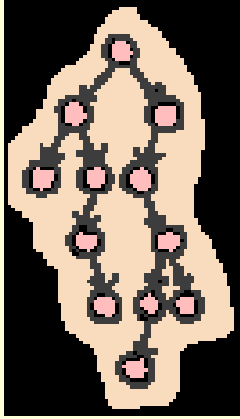


Thiết kế các giải thuật đệ qui

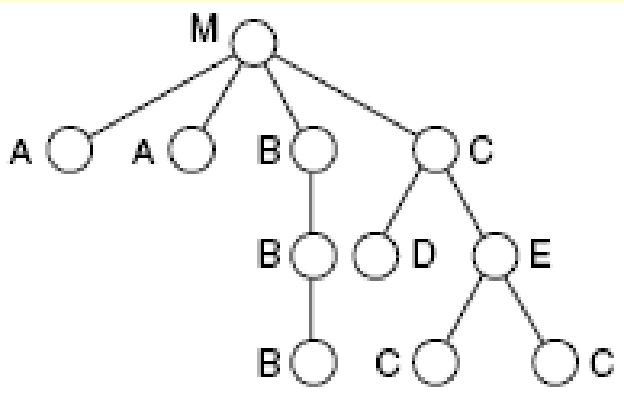


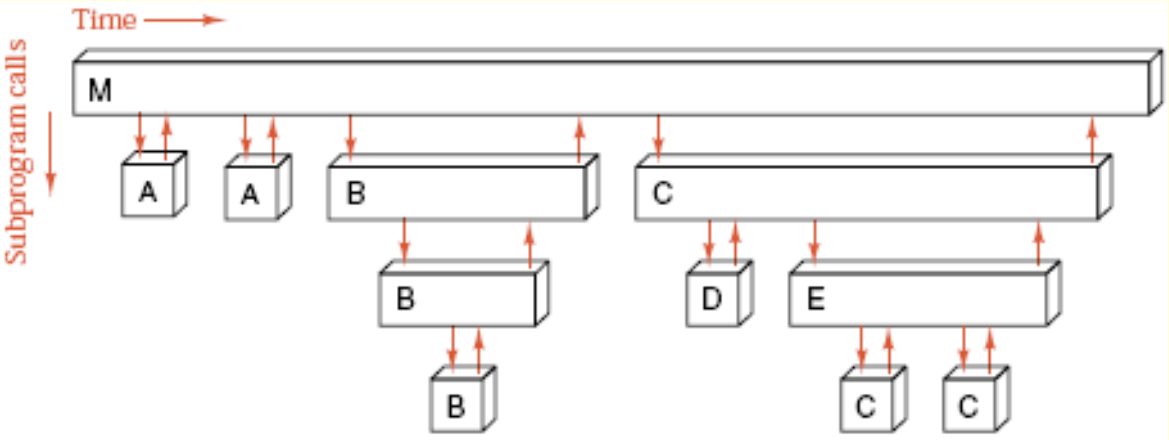
- Tìm bước chính yếu (bước đệ qui)
- Tìm qui tắc ngừng
- Phác thảo giải thuật
  - Dùng câu lệnh if để lựa chọn trường hợp.
- Kiểm tra điều kiện ngừng
  - Đảm bảo là giải thuật luôn dừng lại.
- Vẽ cây đệ qui
  - Chiều cao cây ảnh hưởng lượng bộ nhớ cần thiết.
  - Số nút là số lần bước chính yếu được thi hành.

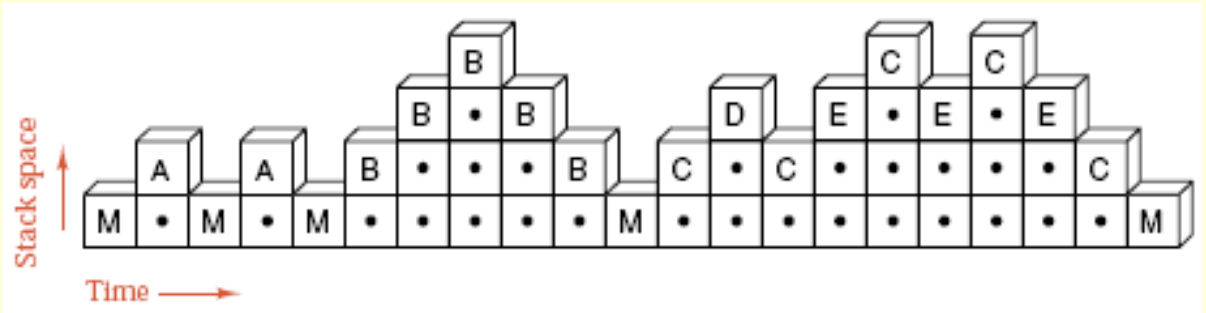
Cây thi hành và stack hệ thống



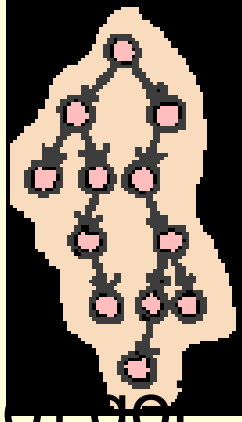
Cây thi hành



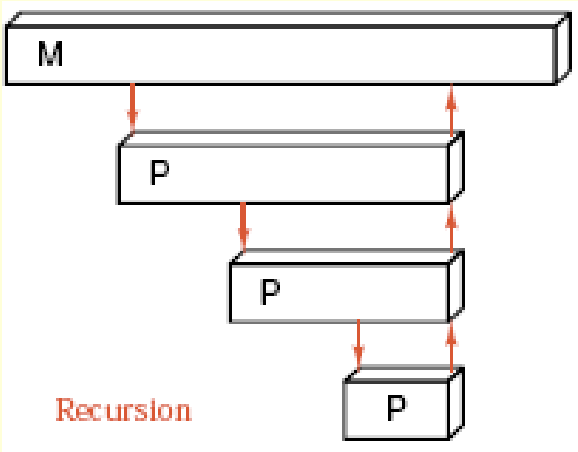


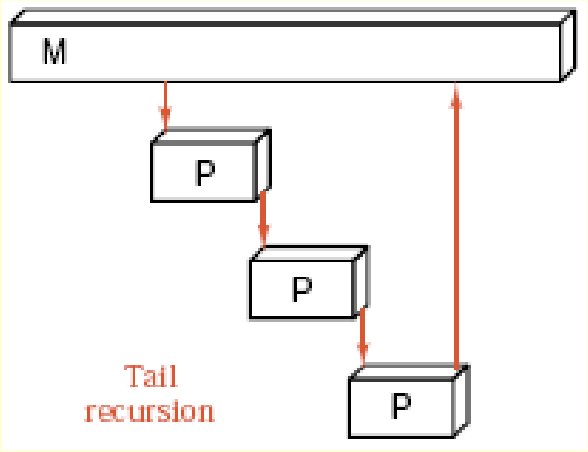


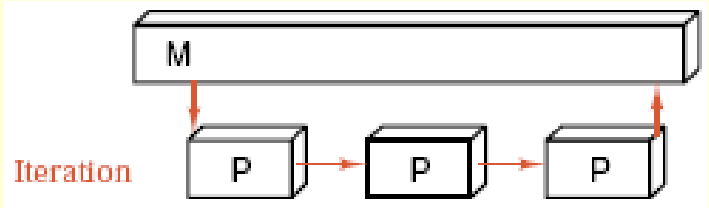
Đệ qui đuôi (tail recursion)



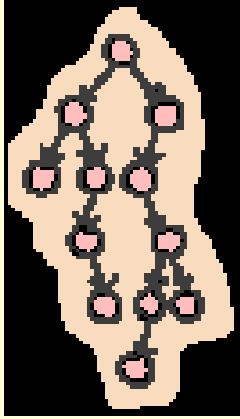
- Định nghĩa: câu lệnh thực thi cuối cùng là lời gọi đệ qui đến chính nó.
- Khử: chuyển thành vòng lặp.



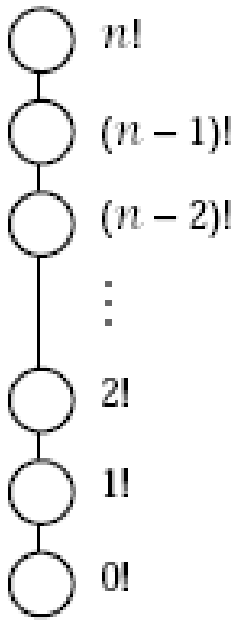




Khử đệ qui đuôi hàm giai thừa



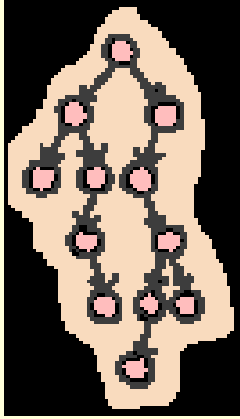
```
factorial(5)  = 5 * factorial(4)
              = 5 * (4 * factorial(3))
              = 5 * (4 * (3 * factorial(2)))
              = 5 * (4 * (3 * (2 * factorial(1))))
              = 5 * (4 * (3 * (2 * (1 * factorial(0)))))
              = 5 * (4 * (3 * (2 * (1 * 1))))
              = 5 * (4 * (3 * (2 * 1)))
              = 5 * (4 * (3 * 6))
              = 5 * (4 * 6)
              = 5 * 24
              = 120.
```



- Giải thuật:

```
product=1
for (int count=1; count < n; count++)
    product *= count;
```

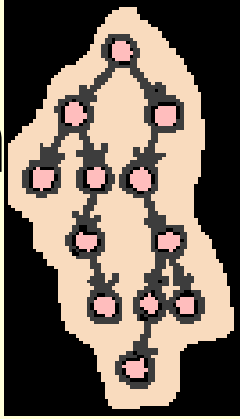
Dãy số Fibonacci

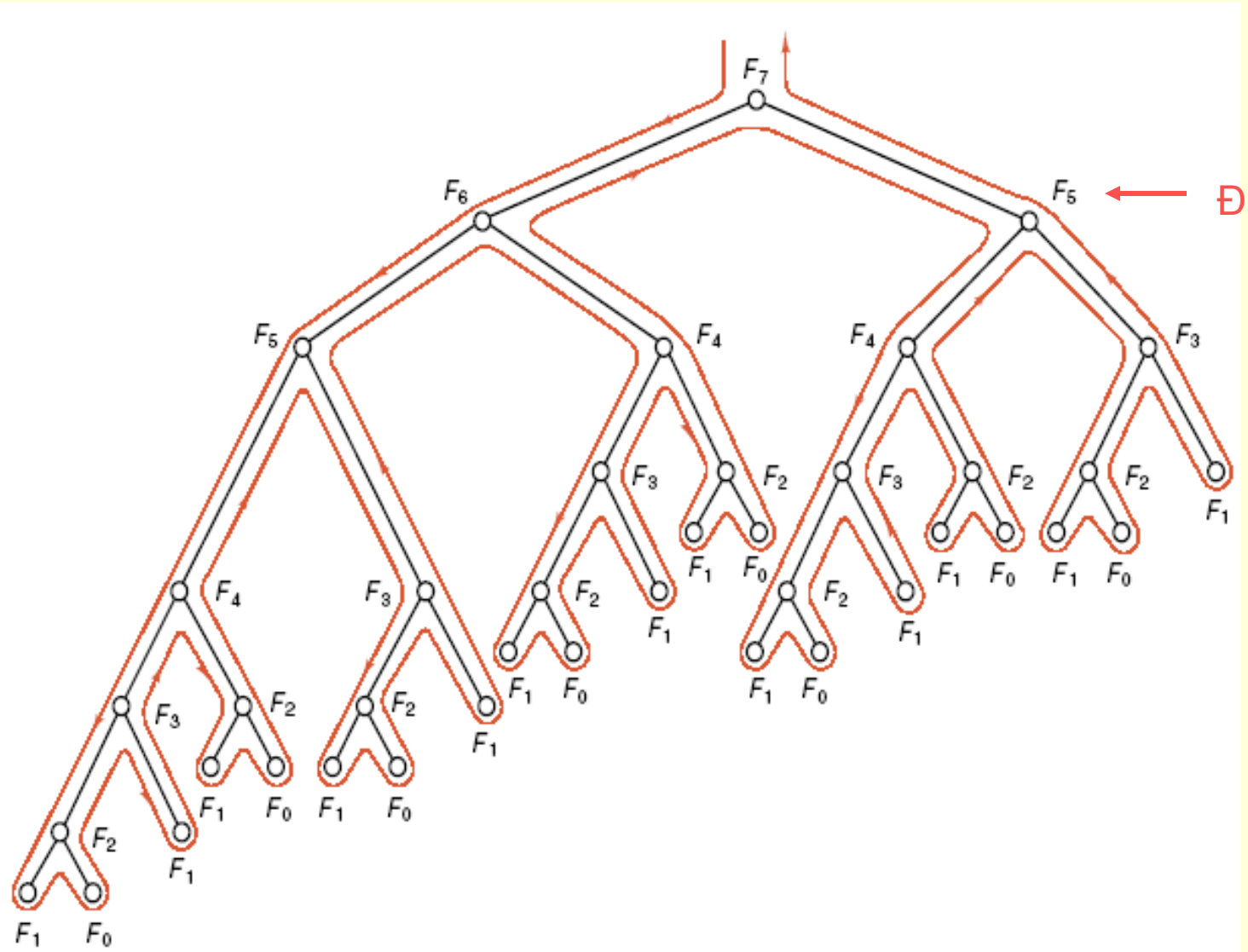


- Định nghĩa:
  - $F_0 = 0$
  - $F_1 = 1$
  - $F_n = F_{n-1} + F_{n-2}$ khi n>2
- Ví dụ: 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, …
- Hàm đệ qui:

```
int fibonacci (int n) {
    if (n<=0) return 0;
    if (n==1) return 1;
    else return (fibonacci(n-1) + fibonacci(n-2));
}
```

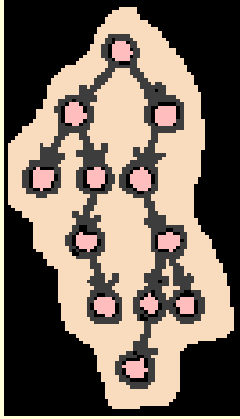
Dãy số Fibonacci – Cây thi hành





Đã tính rồi

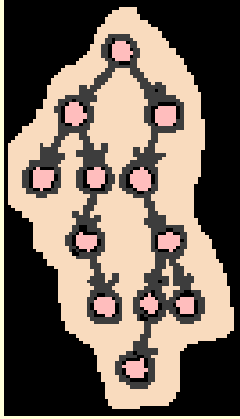
Dãy số Fibonacci – Khử đệ qui

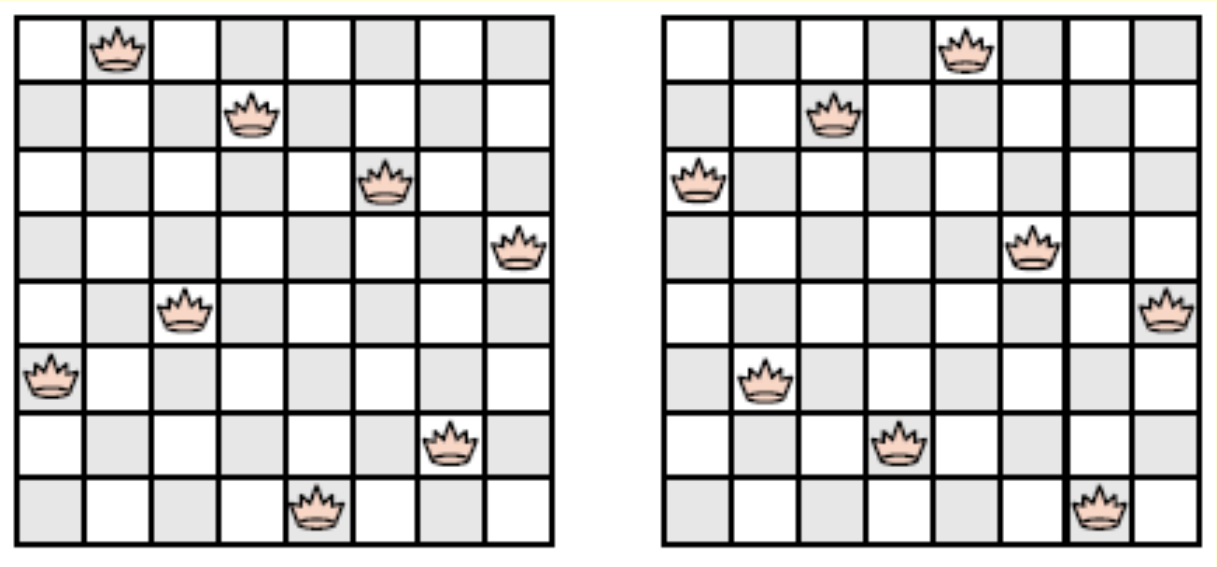


- Nguyên tắc:
  - Dùng biến lưu trữ giá trị đã tính của F_{n-2}
  - Dùng biến lưu trữ giá trị đã tính của F_{n-1}
  - Tính $F_n = F_{n-1} + F_{n-2}$ và lưu lại để dùng cho lần sau
- Giải thuật:

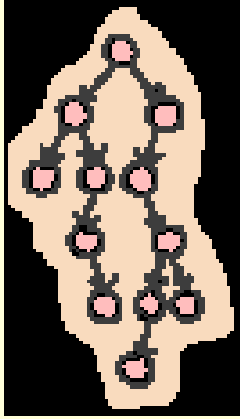
```
int Fn2=0, Fn1=1, Fn;
for (int i = 2; i <= n; i++) {
    Fn = Fn1 + Fn2;
    Fn2 = Fn1; Fn1 = Fn;
}
```

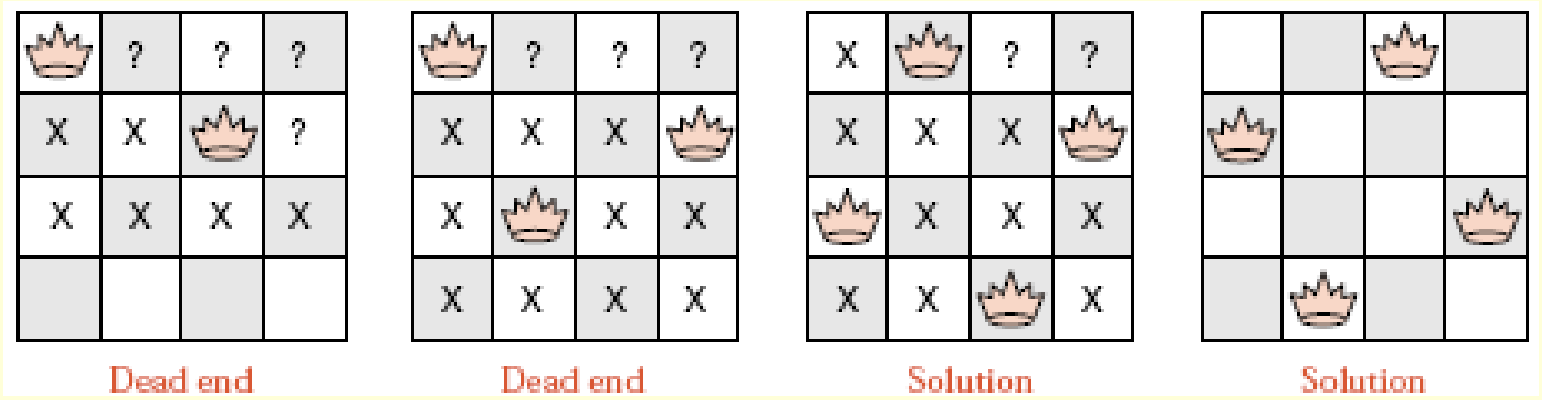
Bài toán 8 con Hậu



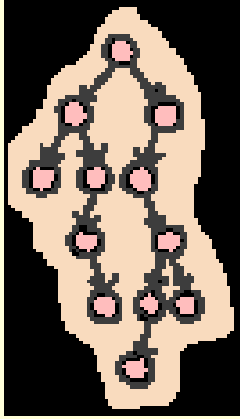


Bài toán 4 con Hậu





Bài toán 8 con Hậu – Giải thuật



Algorithm Solve

Input trạng thái bàn cờ

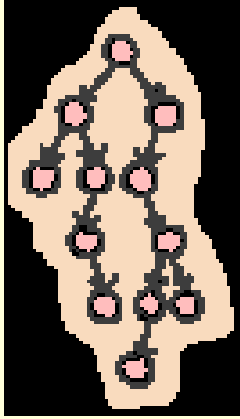
Output

1. **if** trạng thái bàn cờ chứa đủ 8 con hậu
   1.1. In trạng thái này ra màn hình
2. **else**
   2.1. **for** mỗi ô trên bàn cờ mà còn an toàn
      2.1.1. thêm một con hậu vào ô này
      2.1.2. dùng lại giải thuật Solve với trạng thái mới
      2.1.3. bỏ con hậu ra khỏi ô này

End Solve

Vét cạn

Bài toán 8 con Hậu – Thiết kế phương thức



bool Queens :: unguarded(**int** col) **const**;

Post: Returns **true** or **false** according as the square in the first unoccupied row (row count) and column col is not guarded by any queen.

void Queens :: insert(**int** col);

Pre: The square in the first unoccupied row (row count) and column col is not guarded by any queen.

Post: A queen has been inserted into the square at row count and column col; count has been incremented by 1.

void Queens :: remove(**int** col);

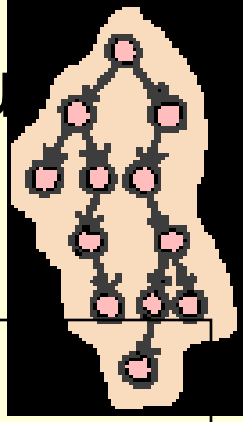
Pre: There is a queen in the square in row count − 1 and column col.

Post: The above queen has been removed; count has been decremented by 1.

bool Queens :: is_solved( ) **const**;

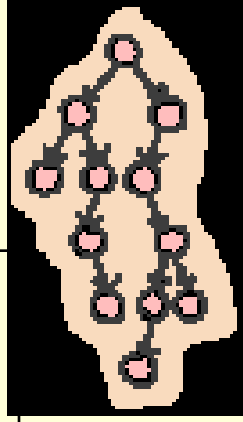
Post: The function returns **true** if the number of queens already placed equals board_size; otherwise, it returns **false**.

Bài toán 8 con Hậu – Thiết kế dữ liệu đơn giản



```
const int max_board = 30;

class Queens {
public:
        Queens(int size);
        bool is_solved( ) const;
        void print( ) const;
        bool unguarded(int col) const;
        void insert(int col);
        void remove(int col);
        int board_size; // dimension of board = maximum number of queens
private:
        int count; // current number of queens = first unoccupied row
        bool queen_square[max_board][max_board];
};
```

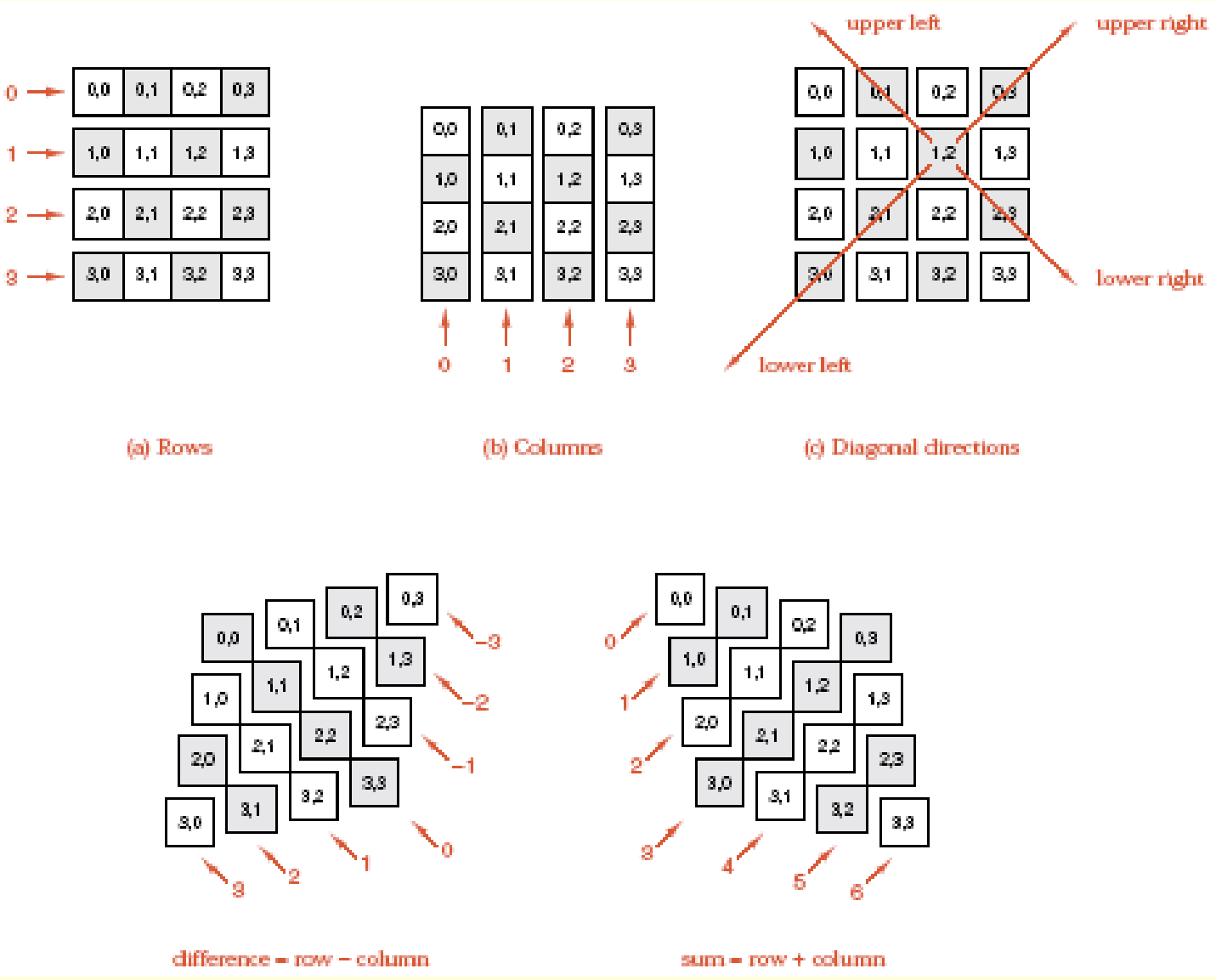
Bài toán 8 con Hậu – Mã C++



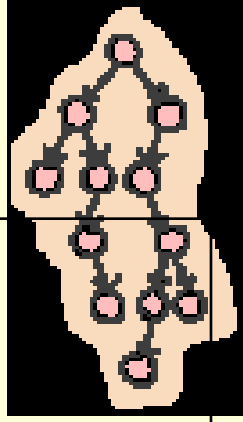
```
void Queens :: insert(int col) {
        queen_square[count++][col] = true;
}
```

```
bool Queens :: unguarded(int col) const {
        int i;
        bool ok = true;
        for (i = 0; ok && i < count; i++)                    //kiểm tra tại một cột
                ok = !queen_square[i][col];
        //kiểm tra trên đường chéo lên
        for (i = 1; ok && count − i >= 0 && col − i >= 0; i++)
                ok = !queen_square[count − i][col − i];
        //kiểm tra trên đường chéo xuống
        for (i = 1; ok && count − i >= 0 && col + i < board_size; i++)
                ok = !queen_square[count − i][col + i];
        return ok;
}
```

Bài toán 8 con Hậu – Góc nhìn khác

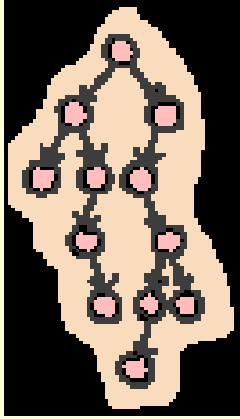


Bài toán 8 con Hậu – Thiết kế mới



```
const int max_board = 30;
class Queens {
public:
        Queens(int size);
        bool is_solved( ) const;
        void print( ) const;
        bool unguarded(int col) const;
        void insert(int col);
        void remove(int col);
        int board size;
private:
        int count;
        bool col_free[max board];
        bool upward_free[2 * max board − 1];
        bool downward_free[2 * max board − 1];
        int queen_in_row[max board]; //column number of queen in each row
};
```

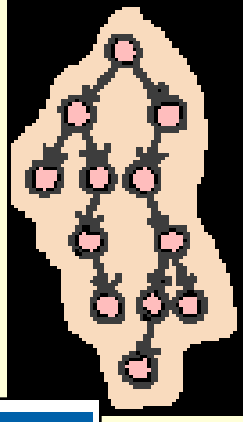
Bài toán 8 con Hậu – Mã C++ mới



```
Queens :: Queens(int size) {
        board size = size;
        count = 0;
        for (int i = 0; i < board_size; i++)
                col_free[i] = true;
        for (int j = 0; j < (2 * board_size − 1); j++)
                upward_free[j] = true;
        for (int k = 0; k < (2 * board_size − 1); k++)
                downward_free[k] = true;
}
```

```
void Queens :: insert(int col) {
        queen_in_row[count] = col;
        col_free[col] = false;
        upward_free[count + col] = false;
        downward_free[count − col + board size − 1] = false;
        count++;
}
```

Bài toán 8 con Hậu – Đánh giá



Thiết kế đầu

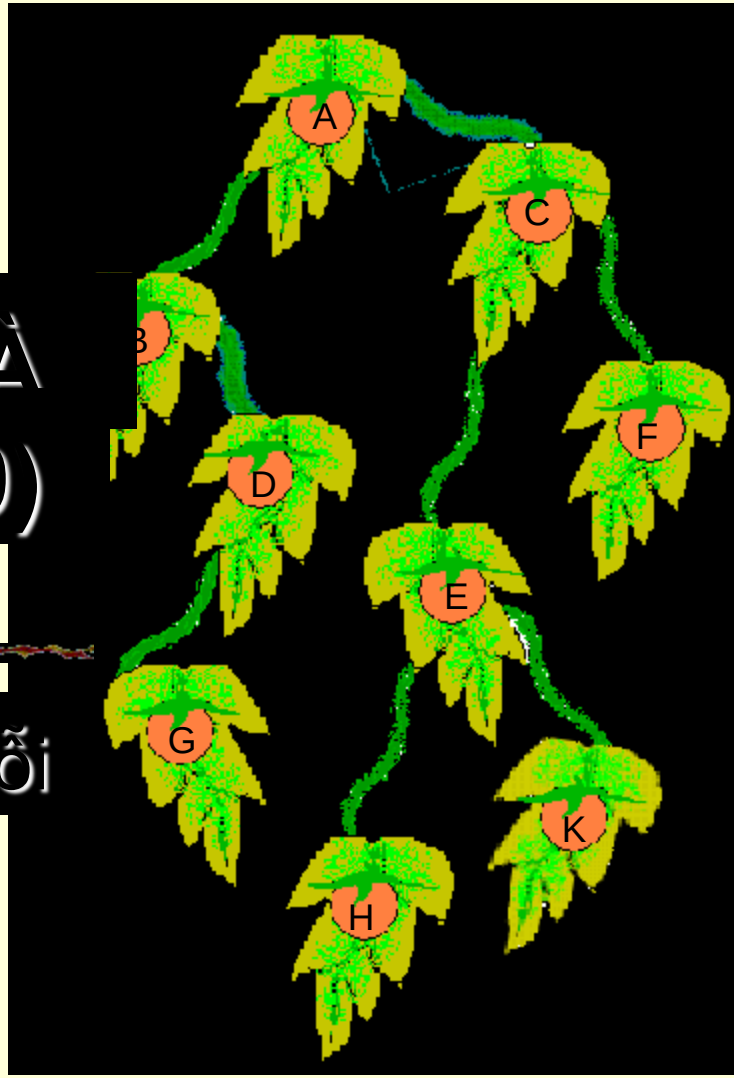
Size	8	9	10	11	12	13
Number of solutions	92	352	724	2680	14200	73712
Time (seconds)	0.05	0.21	1.17	6.62	39.11	243.05
Time per solution (ms.)	0.54	0.60	1.62	2.47	2.75	3.30

Thiết kế mới

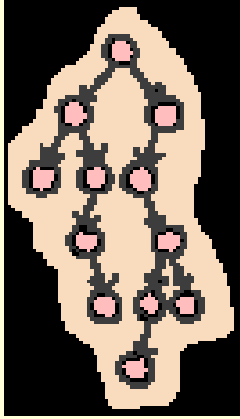
Size	8	9	10	11	12	13
Number of solutions	92	352	724	2680	14200	73712
Time (seconds)	0.01	0.05	0.22	1.06	5.94	34.44
Time per solution (ms.)	0.11	0.14	0.30	0.39	0.42	0.47

CẤU TRÚC DỮ LIỆU VÀ GIẢI THUẬT (501040)

Chương 6: Danh sách và chuỗi

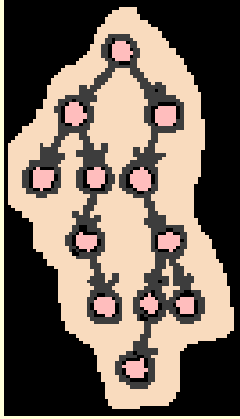


Danh sách trừu tượng



- Một danh sách (list) kiểu T
  - Một dãy hữu hạn kiểu T
  - Một số tác vụ:
    1. Khởi tạo danh sách rỗng (*create*)
    2. Kiểm tra rỗng (*empty*)
    3. Kiểm tra đầy (*full*)
    4. Tính kích thước (*size*)
    5. Xóa rỗng danh sách (*clear*)
    6. Thêm một giá trị vào danh sách tại một ví trí cụ thể (*insert*)
    7. Lấy một giá trị tại một vị trí cụ thể ra khỏi danh sách (*remove*)
    8. Nhận về giá trị tại một vị trí cụ thể (*retrieve*)
    9. Thay thế một giá trị tại một vị trí cụ thể (*replace*)
    10. Duyệt danh sách và thi hành một tác vụ tại mỗi vị trí (*traverse*)

Thiết kế các phương thức



```
List :: List( );
```
Post: The List has been created and is initialized to be empty.

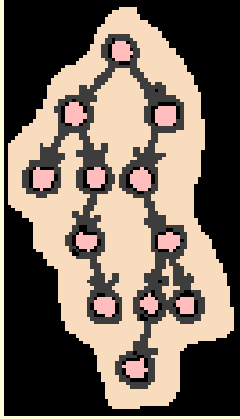
```
bool List :: empty( ) const;
```
Post: The function returns **true** or **false** according to whether the List is empty or not.

```
void List :: clear( );
```
Post: All List entries have been removed; the List is empty.

```
int List :: size( ) const;
```
Post: The function returns the number of entries in the List.

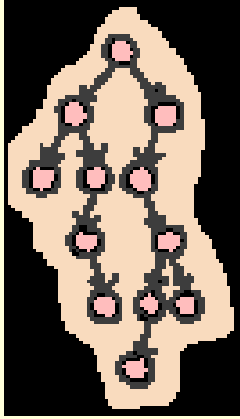
```
bool List :: full( ) const;
```
Post: The function returns **true** or **false** according to whether the List is full or not.

Chỉ số các phần tử



- Đánh chỉ số một danh sách có n phần tử:
  - Đánh chỉ số từ 0, 1, … các phần tử
  - Ví dụ: a_0, a_1, a_2, …, a_{n-1}
  - Phần tử a_{idx} đứng sau a_{idx-1} và trước a_{idx+1} (nếu có)
- Dùng chỉ số:
  - Tìm thấy một phần tử, trả về vị trí (chỉ số) của nó.
  - Thêm vào một phần tử tại vị trí *idx* thì chỉ số các phần tử cũ từ *idx* trở về sau đều tăng lên 1.
  - Chỉ số này được dùng bất kể danh sách được hiện thực thế nào ở cấp vật lý.

Phương thức insert và remove



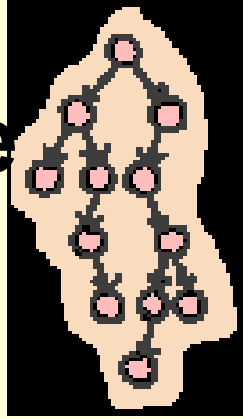
Error_code List :: insert(int position, const List_entry &x);

Post: If the List is not full and $0 \leq$ position $\leq n$, where n is the number of entries in the List, the function succeeds: Any entry formerly at position and all later entries have their position numbers increased by 1, and x is inserted at position in the List.
Else: The function fails with a diagnostic error code.

Error_code List :: remove(int position, List_entry &x);

Post: If $0 \leq$ position $< n$, where n is the number of entries in the List, the function succeeds: The entry at position is removed from the List, and all later entries have their position numbers decreased by 1. The parameter x records a copy of the entry formerly at position.
Else: The function fails with a diagnostic error code.

Phương thức retrieve và replace



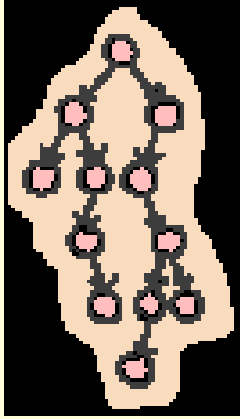
Error_code List::retrieve(int position, List_entry &x) const;

Post: If $0 \leq$ position $< n$, where n is the number of entries in the List, the function succeeds: The entry at position is copied to x; all List entries remain unchanged.
Else: The function fails with a diagnostic error code.

Error_code List::replace(int position, const List_entry &x);

Post: If $0 \leq$ position $< n$, where n is the number of entries in the List, the function succeeds: The entry at position is replaced by x; all other entries remain unchanged.
Else: The function fails with a diagnostic error code.

Phương thức traverse và tham số hàm



```
void List :: traverse(void (*visit)(List_entry &));
```

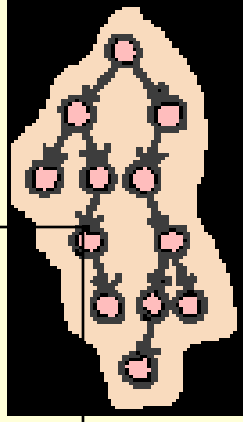
Post: The action specified by function *visit has been performed on every entry of the List, beginning at position 0 and doing each in turn.

```
void print_int(int &x) {   cout << x << " "; }
void increase_int(int &x) {   x++; }

void main() {
        List<int> alist;
        …
        alist.traverse(print_int);
        …
        alist.traverse(increase_int);
        …
}
```

Khi gọi tham số hàm, chương trình dịch phải nhìn thấy hàm được gọi.

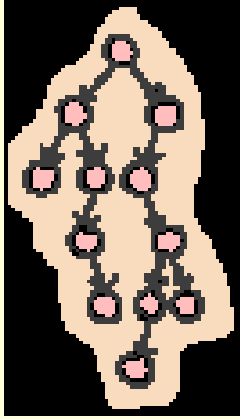
Tùy theo mục đích mà gọi các hàm khác nhau.

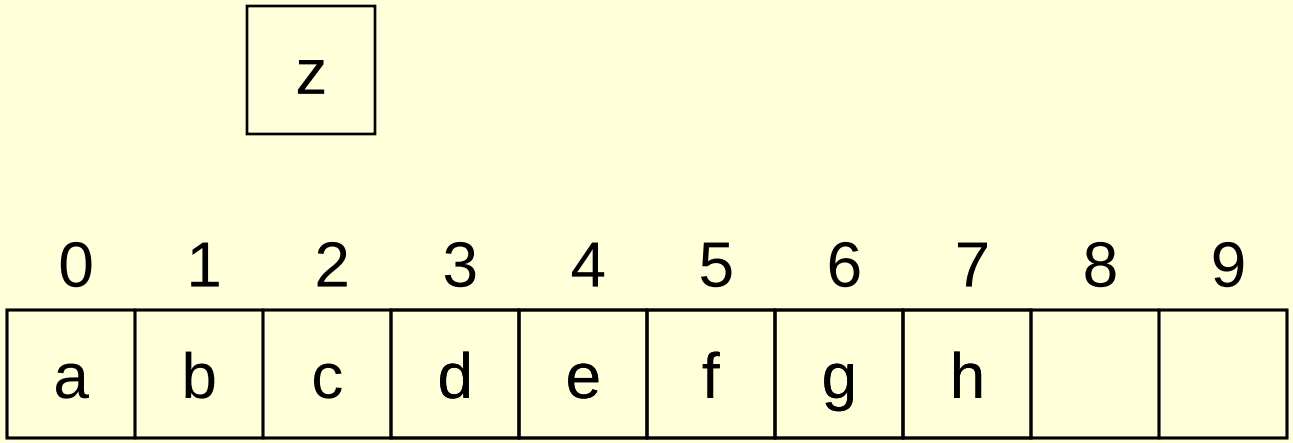
Hiện thực danh sách liên tục



```
template <class List_entry>
class List {
public:
        // methods of the List ADT
        List( );
        int size( ) const;
        bool full( ) const;
        bool empty( ) const;
        void clear( );
        void traverse(void (*visit)(List_entry &));
        Error_code retrieve(int position, List_entry &x) const;
        Error_code replace(int position, const List_entry &x);
        Error_code remove(int position, List_entry &x);
        Error_code insert(int position, const List_entry &x);
protected:
        // data members for a contiguous list implementation
        int count;
        List_entry entry[max_list];
};
```

Thêm vào một danh sách liên tục

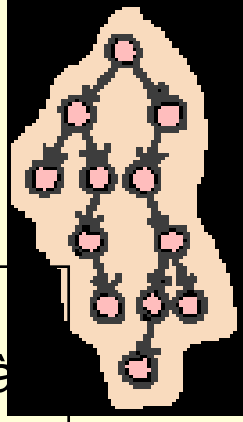




count=8 9

insert(3, ‘z’)

Giải thuật thêm vào một danh sách liên tục



Algorithm Insert

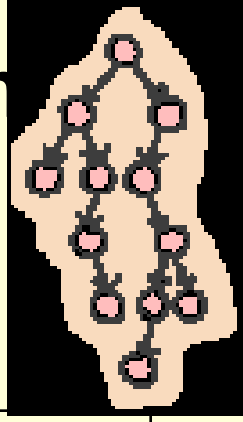
Input: position là vị trí cần thêm vào, x là giá trị cần thêm và

Output: danh sách đã thêm vào x

1. **if** list đầy
   1.1. **return** overflow
2. **if** position nằm ngoài khoảng [0..count]
   2.1. **return** range_error
   //Dời tất cả các phần tử từ position về sau 1 vị trí
3. **for** index = count-1 **down to** position
   3.1. entry[index+1] = entry[index]
4. entry[position] = x *//Gán x vào vị trí position*
5. count++ *//Tăng số phần tử lên 1*
6. **return** success;

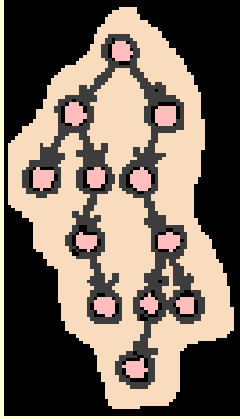
End Insert

Mã C++ thêm vào một danh sách liên tục

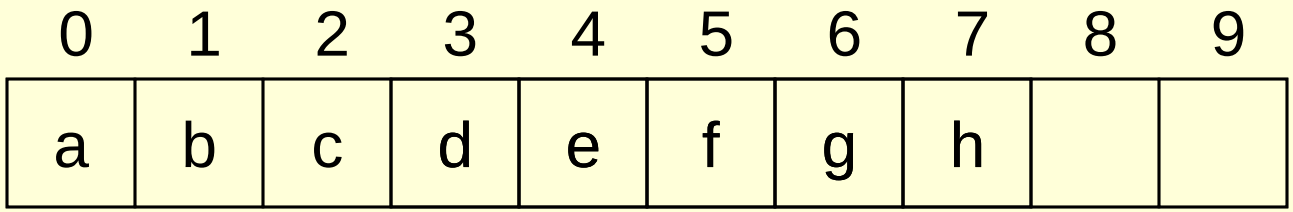


```
template <class List_entry>
Error_code List<List_entry> :: insert(int position, const List_entry &x) {
        if (full( ))
                return overflow;
        if (position < 0 || position > count)
                return range_error;
        for (int i = count − 1; i >= position; i−−)
                entry[i + 1] = entry[i];
        entry[position] = x;
        count++;
        return success;
}
```

Xóa từ một danh sách liên tục



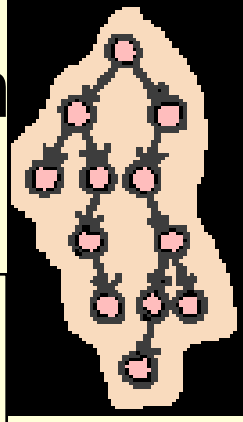
x



count=8

remove(3, x)

Giải thuật xóa từ một danh sách liên tục



Algorithm Remove
Input: position là vị trí cần xóa bỏ, x là giá trị lấy ra được
Output: danh sách đã xóa bỏ phần tử tại position

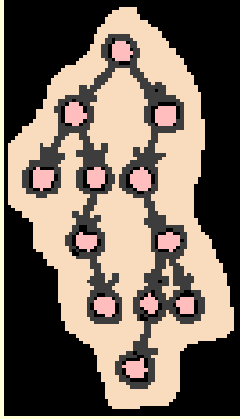
1. **if** list rỗng
   1.1. **return** underflow
2. **if** position nằm ngoài khoảng [0..count-1]
   2.1. **return** range_error
3. x = entry[position] *//Lấy x tại vị trí position ra*
4. count-- *//Giảm số phần tử đi 1*

//Dời tất cả các phần tử từ position về trước 1 vị trí

5. **for** index = position **to** count-1
   5.1. entry[index] = entry[index+1]
6. **return** success;

End Remove

Giải thuật duyệt một danh sách liên tục

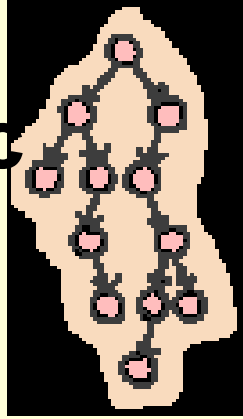


Algorithm Traverse
Input: hàm visit dùng để tác động vào từng phần tử
Output: danh sách được cập nhật bằng hàm visit

//Quét qua tất cả các phần tử trong list
1. **for** index = 0 **to** count-1
   1.1. Thi hành hàm visit để duyệt phần tử entry[index]

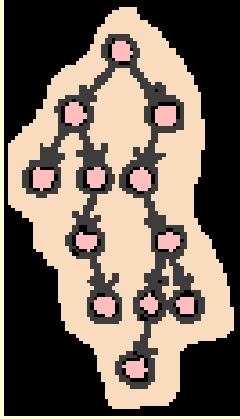
End Traverse

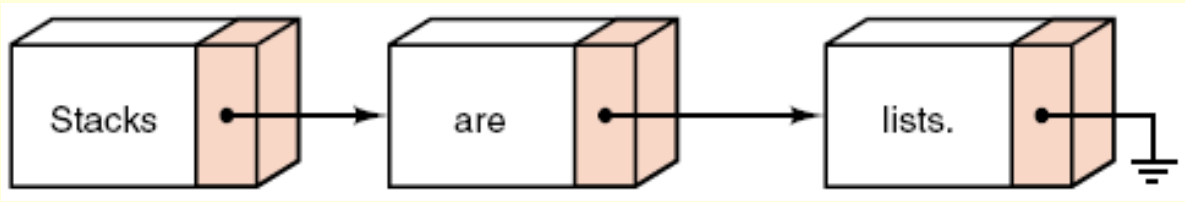
Mã C++ duyệt một danh sách liên tục

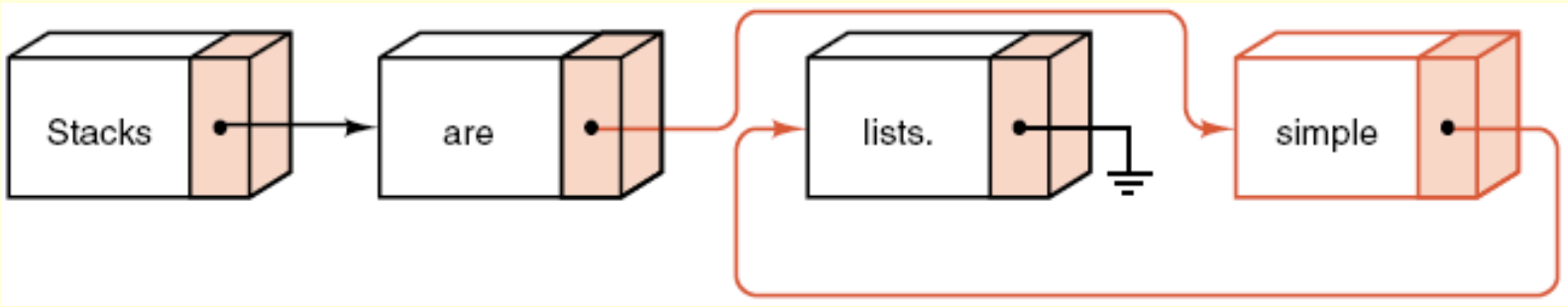


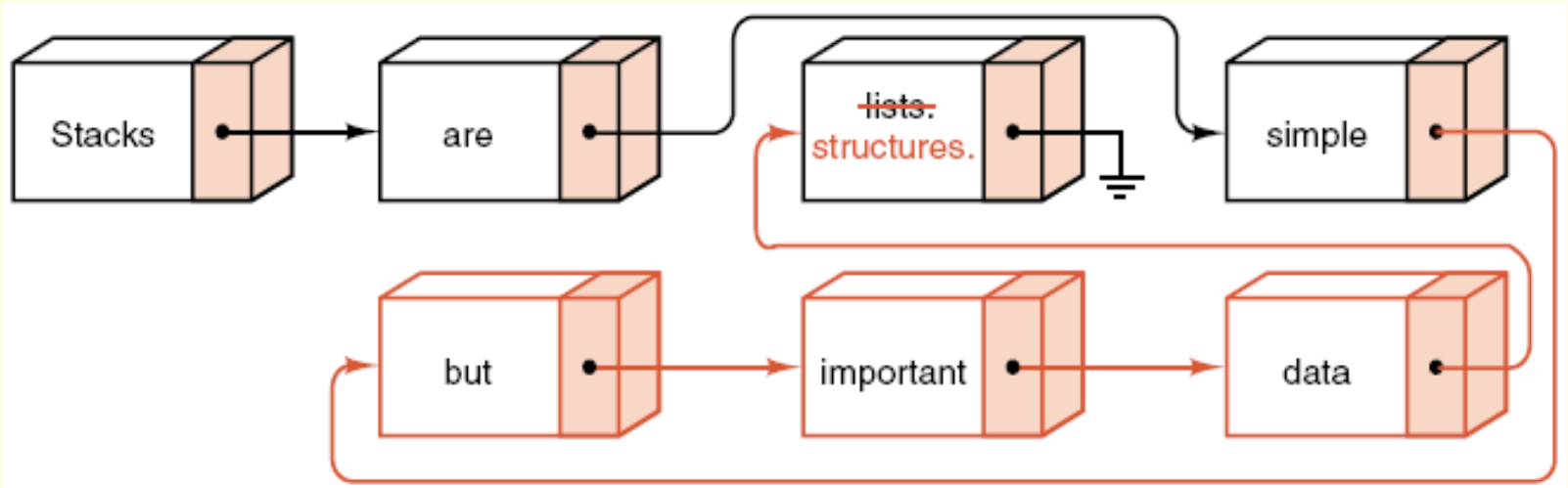
```
template <class List_entry>
void List<List_entry> :: traverse(void (*visit)(List_entry &))
/* Post: Tác vụ cho bởi hàm visit sẽ được thi hành tại mỗi
thành phần của list bắt đầu từ vị trí 0 trở đi. */
{
        for (int i = 0; i < count; i++)
                (*visit)(entry[i]);
}
```

Danh sách liên kết đơn (DSLK đơn)

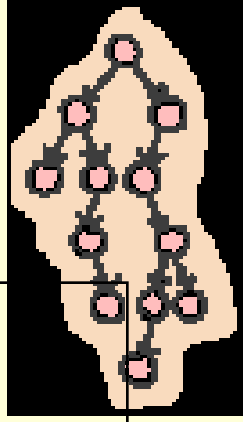






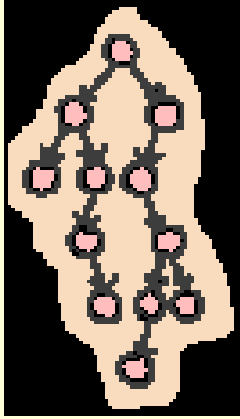


Hiện thực DSLK đơn



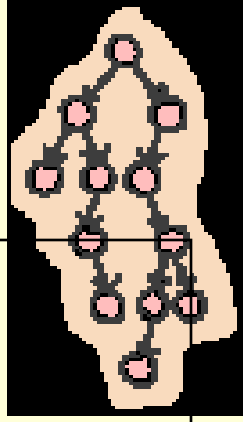
```
template <class List_entry>
class List {
public:
        // Specifications for the methods of the list ADT go here.
        // The following methods replace compiler-generated defaults.
        List( );
        ~List( );
        List(const List<List_entry> &copy);
        void operator = (const List<List_entry> &copy);
protected:
        // Data members for the linked list implementation now follow.
        int count;
        Node<List_entry> * head;
        // The following auxiliary function is used to locate list positions
        Node<List_entry> *set_position(int position) const;
};
```

Tìm vị trí trên DSLK đơn



- Nhu cầu:
  - Nhập vào chỉ số của một phần tử
  - Cho biết đó là phần tử nào (con trỏ chỉ đến phần tử)
- Ý tưởng:
  - Bắt đầu từ phần tử đầu tiên
  - Di chuyển đúng *position* bước thì đến được phần tử cần tìm
  - Phải đảm bảo là *position* nằm trong khoảng [0..count-1]

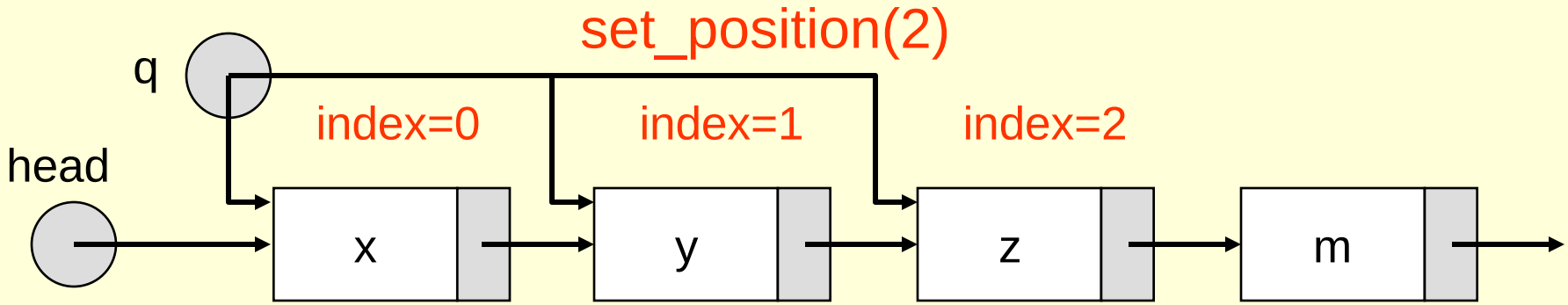
Giải thuật tìm vị trí trên DSLK đơn



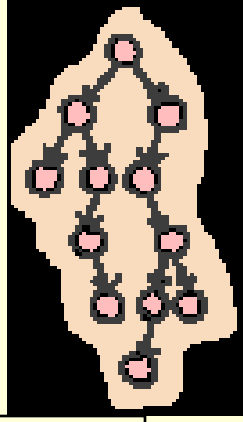
Algorithm Set position
Input: position là vị trí cần tìm
Output: con trỏ chỉ đến phần tử tại vị trí cần tìm

1. **set** q to head
2. **for** index =0 **to** position //Thi hành position bước
   2.1. **advance** q to the next element //Trỏ q đến phần tử kế tiếp
3. **return** q

End Set position

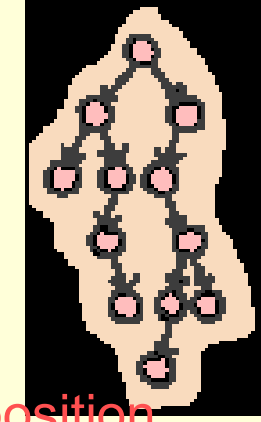


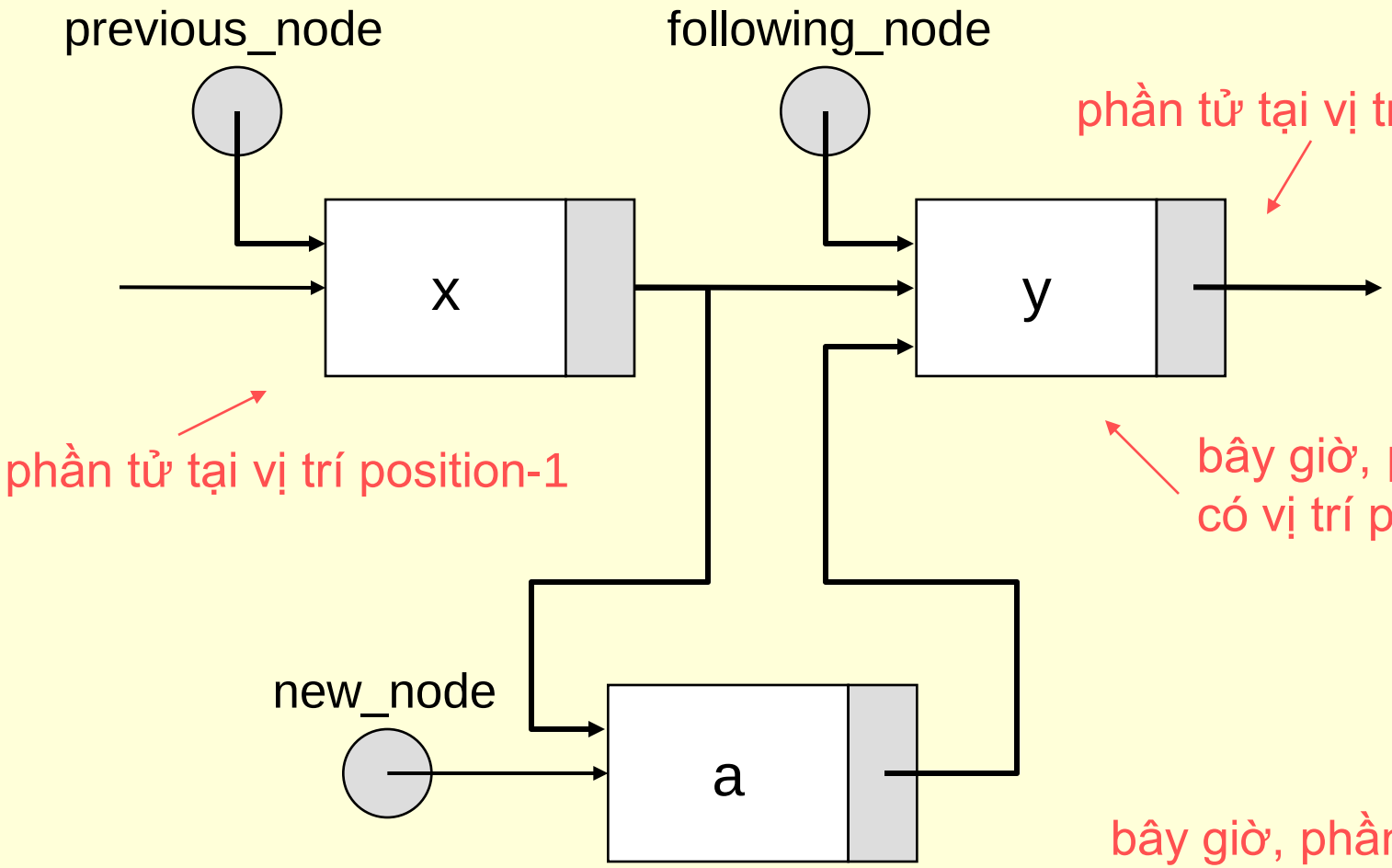
Mã C++ tìm vị trí trên DSLK đơn



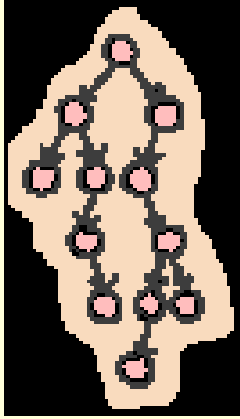
```
template <class List_entry>
Node<List_entry> *List<List_entry> :: set_position(int position) const
/* Pre: position là vị trí hợp lệ trong list, 0 < position < count.
   Post: Trả về một con trỏ chỉ đến Node đang ở vị trí position
*/
{
        Node<List_entry> *q = head;
        for (int i = 0; i < position; i++)
                q = q->next;
        return q;
}
```

Thêm vào một DSLK đơn





Giải thuật thêm vào một DSLK đơn



Algorithm Insert

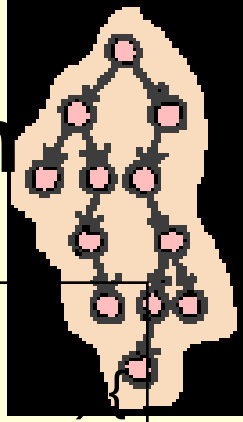
Input: position là vị trí thêm vào, x là giá trị thêm vào

Output: danh sách đã thêm vào x tại vị trí position

1. **Nếu** position > 0
   1.1. Trỏ previous đến phần tử tại vị trí position-1
   1.2. Trỏ following đến phần tử sau previous
2. **Ngược lại**
   2.1. Trỏ following đến head
3. **Tạo ra** node mới là new_node với giá trị x
4. Trỏ next của new_node đến following
5. **Nếu** position là 0
   5.1. Trỏ head đến new_node
6. **Ngược lại**
   6.1. Trỏ next của previous đến new_node
7. **Tăng** số lượng các phần tử lên 1

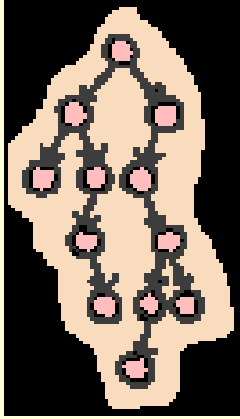
End Insert

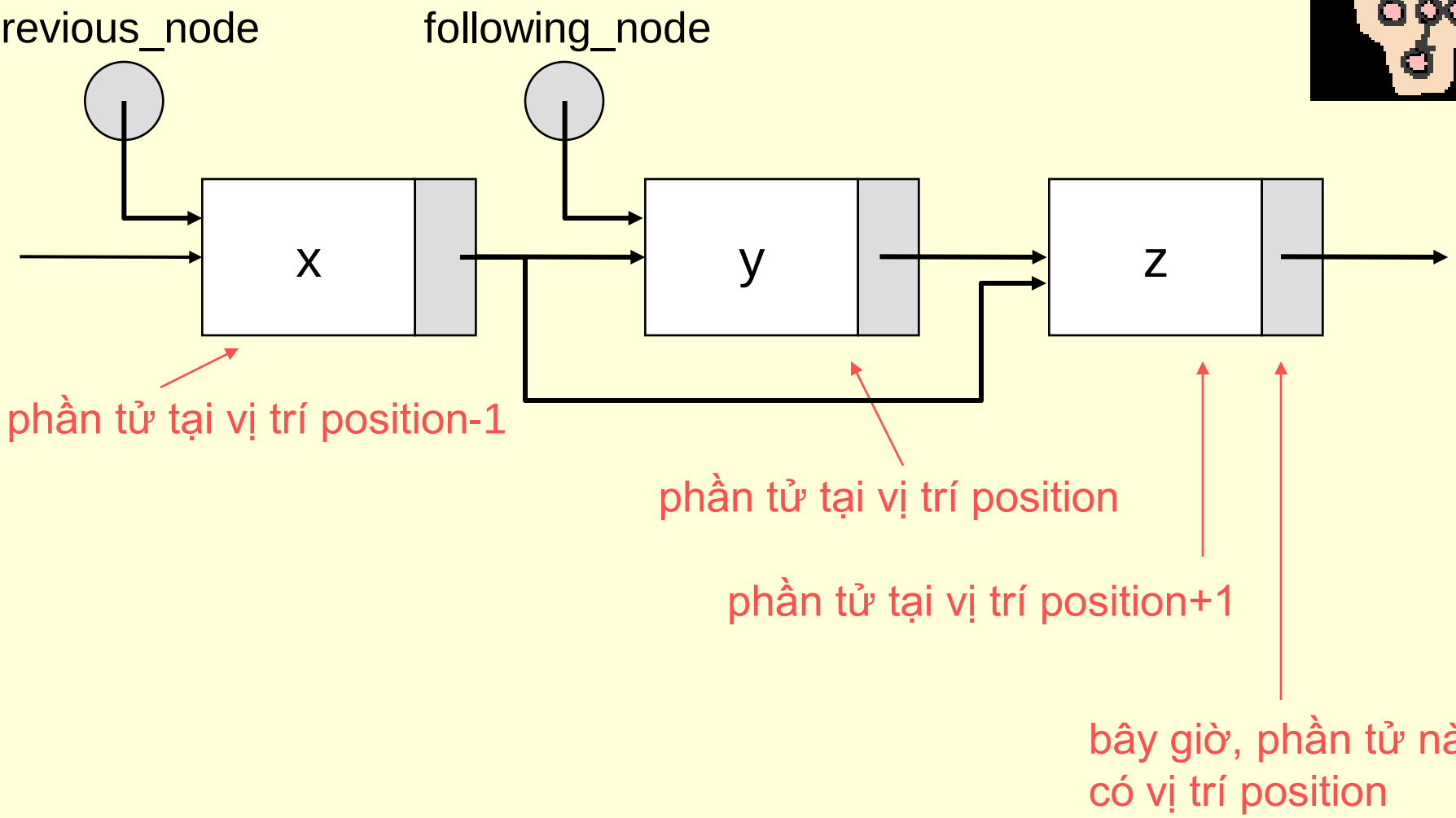
Mã C++ thêm vào một DSLK đơn



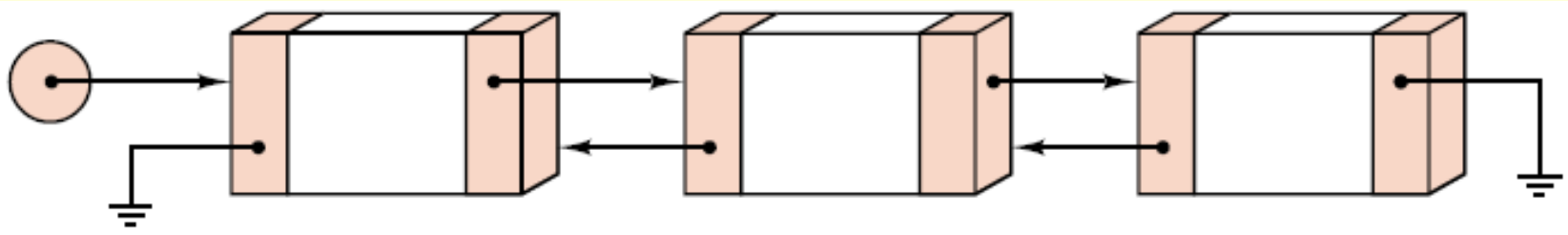
```
template <class List_entry>
Error_code List<List_entry> :: insert(int position, const List_entry &x)
{
        if (position < 0 || position > count)
                return range_error;
        Node<List_entry> *new_node, *previous, *following;
        if (position > 0) {
                previous = set_position(position − 1);
                following = previous->next;
        } else following = head;
        new_node = new Node<List_entry>(x, following);
        if (new_node == NULL) return overflow;
        if (position == 0) head = new_node;
        else previous->next = new_node;
        count++;
        return success;
}
```

Xóa bỏ từ một DSLK đơn

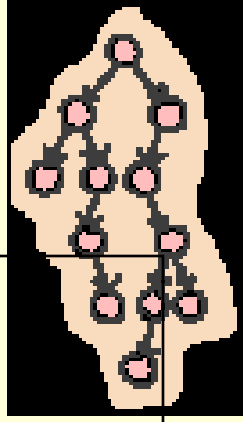




DSLK kép (Doubly linked list)



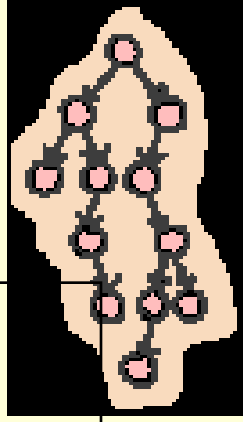
Định nghĩa DSLK kép



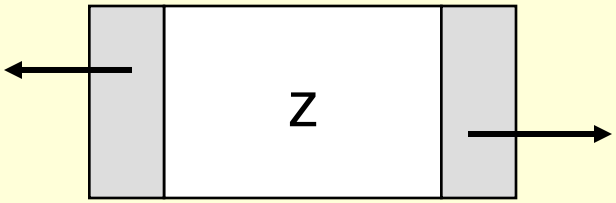
```
template <class List_entry>
class List {
public:
        // Add specications for methods of the list ADT.
        // Add methods to replace compiler generated defaults.
protected:
        // Data members for the doubly-linked list implementation follow:
        int count;
        mutable int current_position;
        mutable Node<List_entry> *current;
        // The auxiliary function to locate list positions follows:
        void set_position(int position) const;
};
```

Các hàm hằng (**const**) có thể thay đổi giá trị của các biến **mutable** này

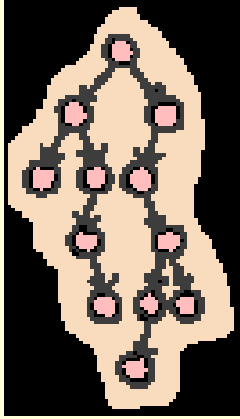
Định nghĩa Node cho DSLK kép



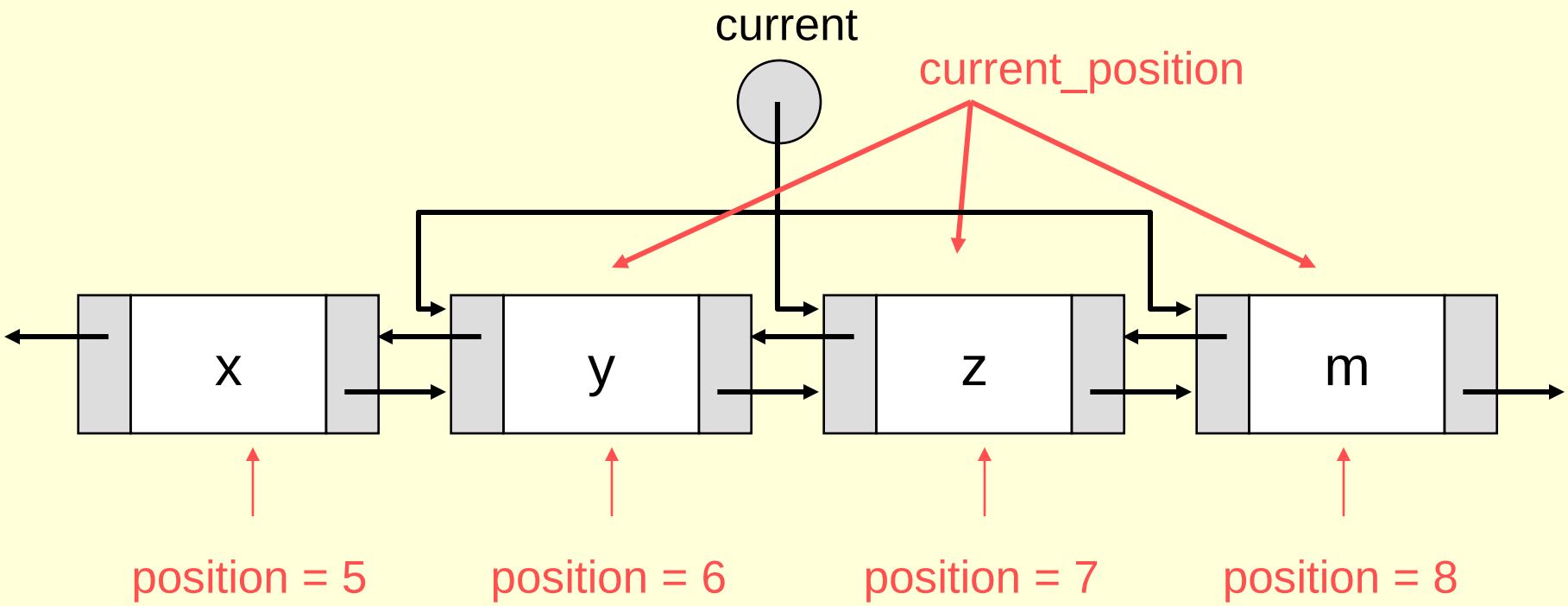
```
template <class Node_entry>
struct Node {
        // data members
        Node_entry entry;
        Node<Node_entry> *next;
        Node<Node_entry> *back;
        // constructors
        Node( );
        Node(Node_entry, Node<Node_entry> *link_back = NULL,
                Node<Node_entry> *link_next = NULL);
};
```



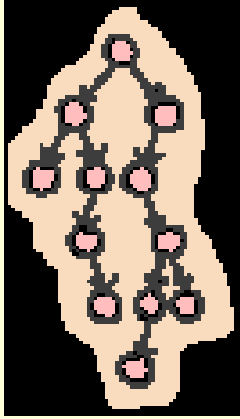
Tìm vị trí trong DSLK kép

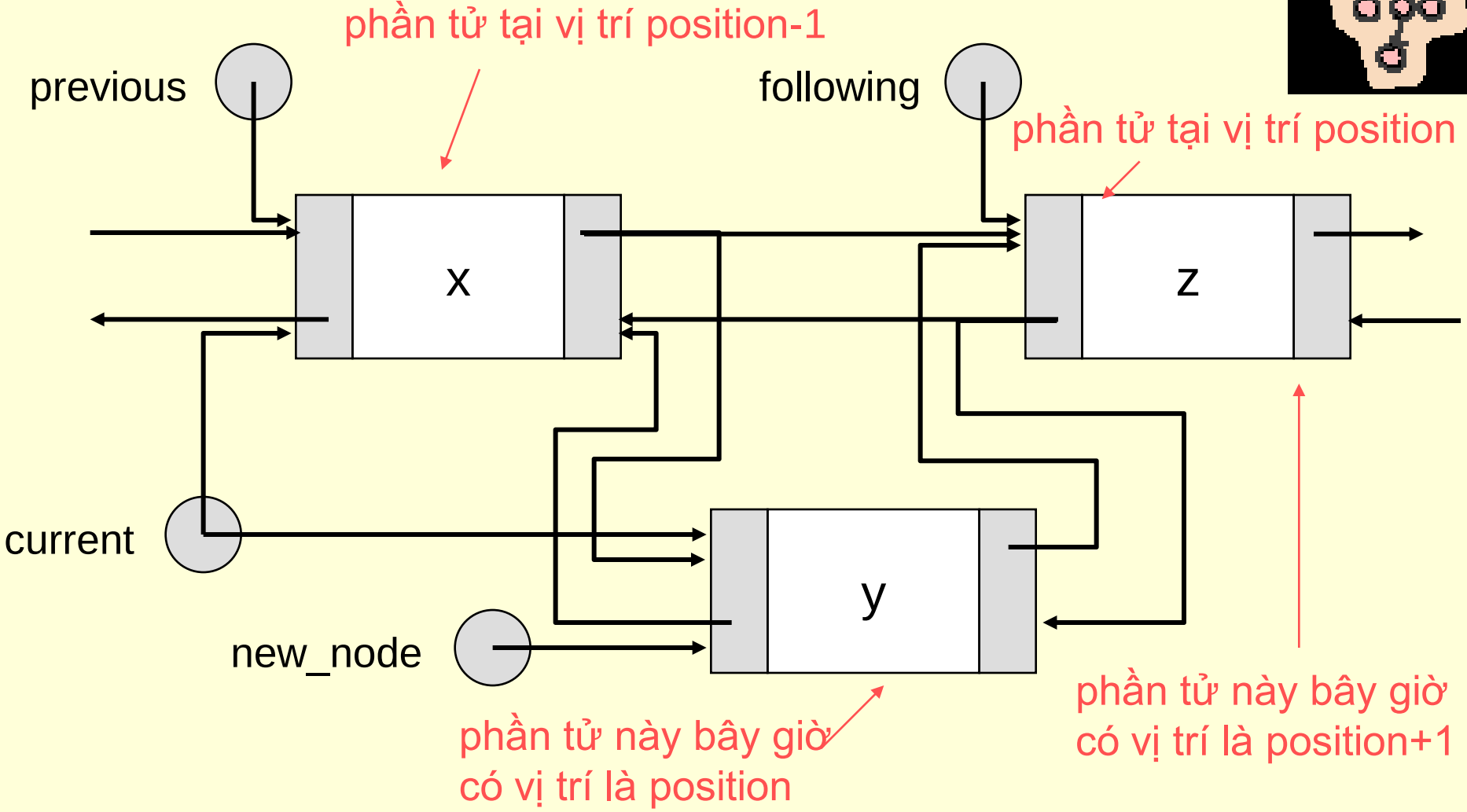


set_position(8)

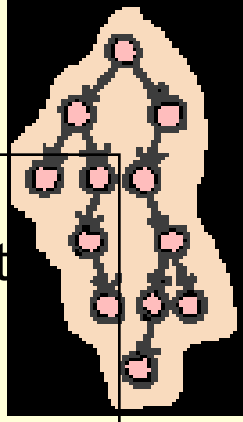


Thêm vào trong DSLK kép





Thêm vào trong DSLK kép

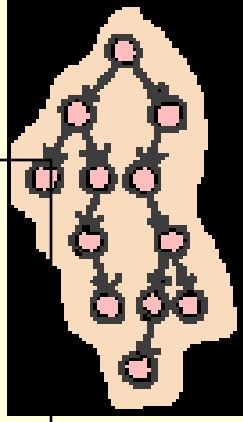


Algorithm Insert
Input: x là giá trị cần thêm vào tại position (0<=position<=count
Output: danh sách đã thêm giá trị x vào vị trí position

1. **if** position là 0
   1.1. **if** số phần tử là 0
      1.1.1. Trỏ following đến NULL
   1.2. Trỏ preceding đến NULL
2. **else**
   2.1. Trỏ preceding đến vị trí position -1, following đến vị trí position
3. **Tạo** ra phần tử mới new_node
4. Trỏ next và back của new_node đến following và preceding
5. **if** preceding khác rỗng
   5.1. Trỏ next của preceding đến new_node
6. **if** following khác rỗng
   6.1. Trỏ back của following đến new_node
7. **Tăng** số phần tử lên 1

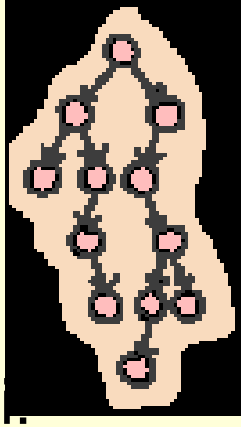
End Insert

Mã C++ thêm vào trong DSLK kép



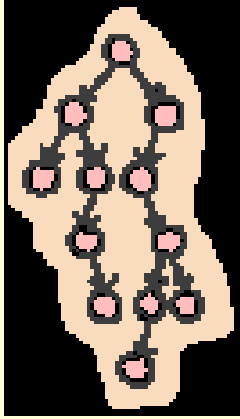
```
template <class List_entry>
Error_code List<List_entry> :: insert(int position, const List_entry &x)
        Node<List_entry> *new node, *following, *preceding;
        if (position < 0 || position > count) return range_error;
        if (position == 0) {
                if (count == 0) following = NULL;
                else { set_position(0); following = current; }
                preceding = NULL;
        }  else {
                set_position(position − 1);
                preceding = current; following = preceding->next;
        }
        new_node = new Node<List_entry>(x, preceding, following);
        if (new_node == NULL) return overflow;
        if (preceding != NULL) preceding->next = new_node;
        if (following != NULL) following->back = new_node;
        current = new_node;  current_position = position;
        count++;
        return success;
}
```

So sánh cách hiện thực liên tục và cách hiện thực liên kết



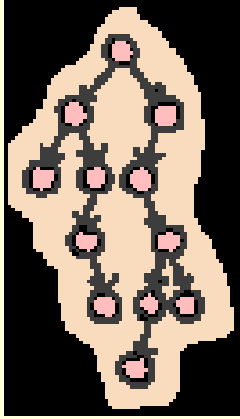
- DS liên tục thích hợp khi:
  - Kích thước từng phần tử là rất nhỏ
  - Kích thước của cả danh sách (số phần tử) đã biết khi lập trình
  - Có ít sự thêm vào hay loại bỏ ở giữa danh sách
  - Hình thức truy cập trực tiếp là quan trọng
- DSLK thích hợp khi:
  - Kích thước từng phần tử là lớn
  - Kích thước của danh sách không biết trước
  - Có nhiều sự thêm vào, loại bỏ, hay xắp xếp các phần tử trong danh sách

Chuỗi (string)



- Chuỗi là một dãy các ký tự
- Ví dụ:
  - “This is a string” là 1 chuỗi có 16 ký tự
  - “” là một chuỗi rỗng (có 0 ký tự)
- Chuỗi trừu tượng:
  - Có thể xem là danh sách
  - Có các tác vụ thường dùng:
    - Sao chép (*strcpy*)
    - Nối kết (*strcat*)
    - Tính chiều dài (*strlen*)
    - So sánh 2 chuỗi (*strcmp*)
    - Tìm một chuỗi trong chuỗi khác (*strstr*)

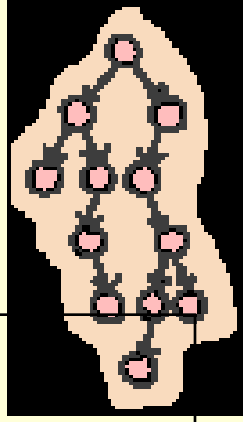
Chuỗi trên C



- Có kiểu là **char ***
- Kết thúc bằng ký tự '\0' (NULL)
  - Số phần tử trong bộ nhớ nhiều hơn chiều dài chuỗi là 1
- Cần chuẩn bị bộ nhớ cần thiết khi thao tác
  - Ví dụ:

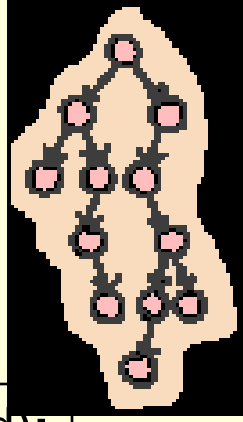
```
char *str1, *str2;
…
delete str2;
str2 = new char[strlen(str1) + 1];
strcpy(str2, str1);
```

Thiết kế lại kiểu dữ liệu chuỗi



```
class String {
public: // methods of the string ADT
        String( );
        ~String( );
        String (const String &copy);        // copy constructor
        String (const char * copy);         // conversion from C-string
        String (List<char> &copy);          // conversion from List
        void operator = (const String &copy);
        const char *c_str( ) const;         // conversion to C-style string
protected:
        char *entries;
        int length;
};
```

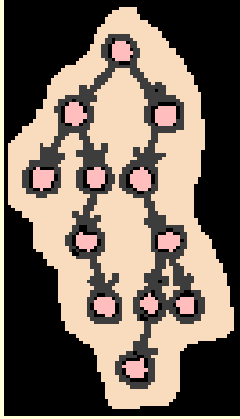
Thiết kế các toán tử cần thiết



```
bool operator == (const String &first, const String &second);
bool operator > (const String &first, const String &second);
bool operator < (const String &first, const String &second);
bool operator >= (const String &first, const String &second);
bool operator <= (const String &first, const String &second);
bool operator != (const String &first, const String &second);
```

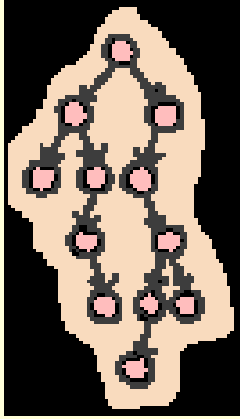
```
bool operator == (const String &first, const String &second) {
        return strcmp(first.c_str( ), second.c_str( )) == 0;
}
```

Khởi tạo với chuỗi C



```
String :: String (const char *in_string)
/* Pre: The pointer in_string references a C-string.
Post: The String is initialized by the C-string in_string. */
{
        length = strlen(in_string);
        entries = new char[length + 1];
        strcpy(entries, in_string);
}
```

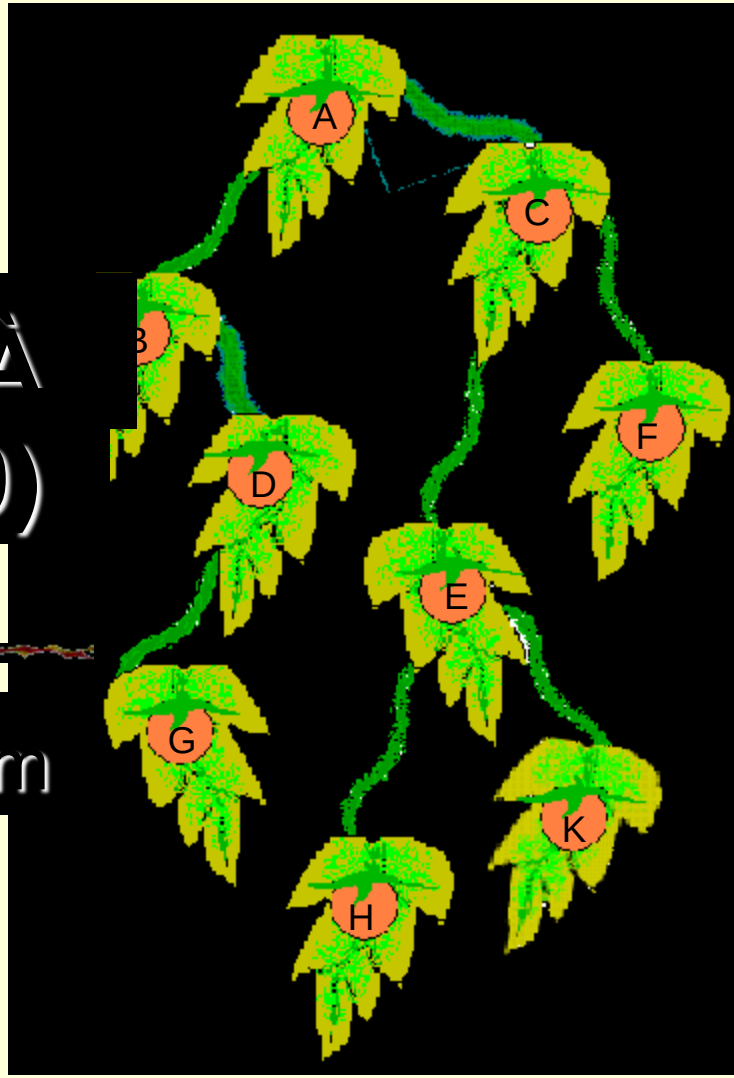
Khởi tạo với danh sách ký tự



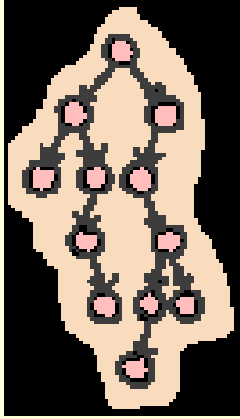
```
String :: String (List<char> &in_list)
/* Post: The String is initialized by the character List in_list. */
{
        length = in_list.size( );
        entries = new char[length + 1];
        for (int i = 0; i < length; i++)
                in_list.retrieve(i, entries[i]);

        //Gán '\0' để kết thúc chuỗi
        entries[length] = '\0';
}
```

CẤU TRÚC DỮ LIỆU VÀ GIẢI THUẬT (501040)

Chương 7: Tìm kiếm

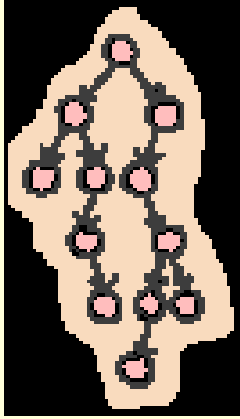


Khái niệm tìm kiếm

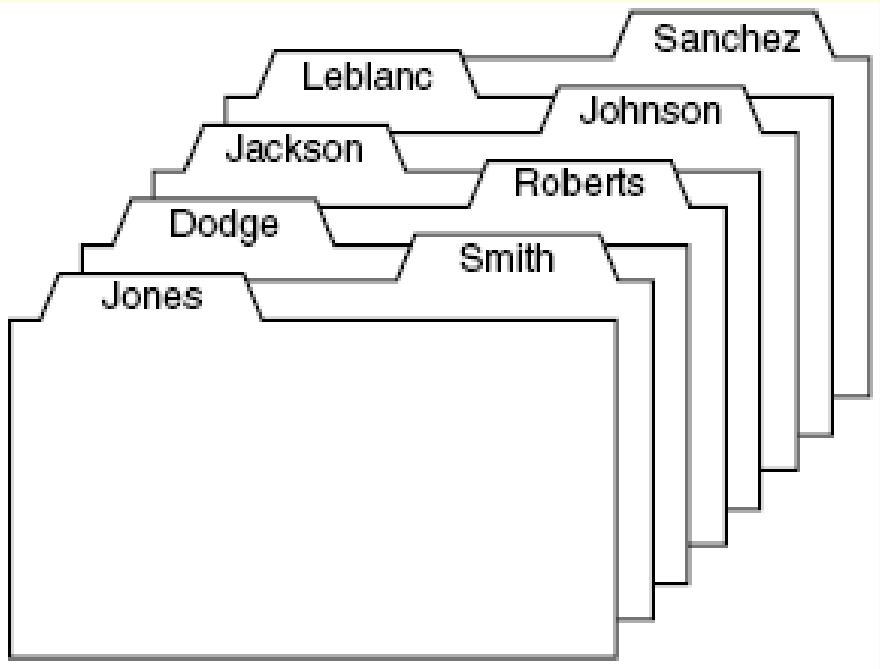


- Cho biết:
  - Một danh sách các bản ghi (record).
  - Một khóa cần tìm.
- Tìm bản ghi có khóa trùng với khóa cần tìm (nếu có).
- Đo độ hiệu quả:
  - Số lần so sánh khóa cần tìm và khóa của các bản ghi
- Phân loại:
  - Tìm kiếm nội (internal searching)
  - Tìm kiếm ngoại (external searching)

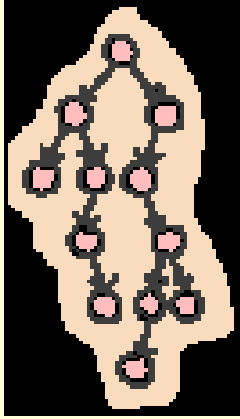
Bản ghi và khóa



- Bản ghi:
  - Khóa
  - Dữ liệu
- Khóa:
  - So sánh được
  - Thường là số
- Trích khóa từ bản ghi:
  - So sánh các bản ghi



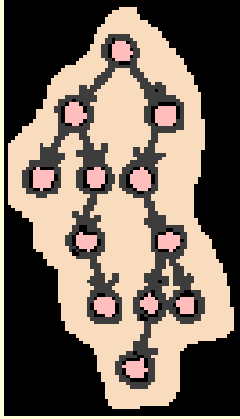
Bản ghi và khóa trên C++



```
class Key {
public:  // Add any constructors and methods for key data.
private: // Add declaration of key data members here.
};
bool operator == (const Key &x, const Key &y);
bool operator > (const Key &x, const Key &y);
bool operator < (const Key &x, const Key &y);
bool operator >= (const Key &x, const Key &y);
bool operator <= (const Key &x, const Key &y);
bool operator != (const Key &x, const Key &y);
```

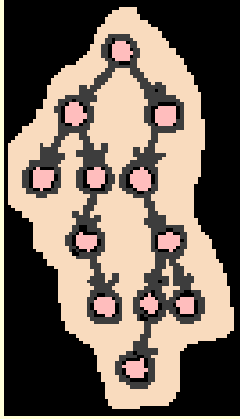
```
class Record{
public:
    operator Key( ); // implicit conversion from Record to Key .
    // Add any constructors and methods for Record objects.
private:
    // Add data components.
};
```

Hàm tìm kiếm



- Tham số vào:
  - Danh sách cần tìm
  - Khóa cần tìm
- Tham số ra:
  - Vị trí phần tử tìm thấy (nếu có)
- Kết quả hàm: kiểu Error_code
  - Tìm thấy: success
  - Không tìm thấy: not_present

Tìm tuần tự (sequential search)



5

Target key

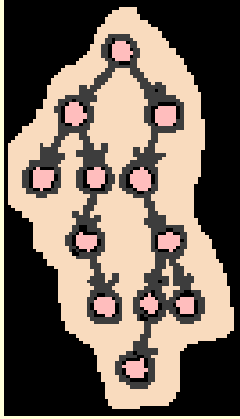
position = 2

0	1	2	3	4	5	6	7
7	13	5	21	6	2	8	15

return success

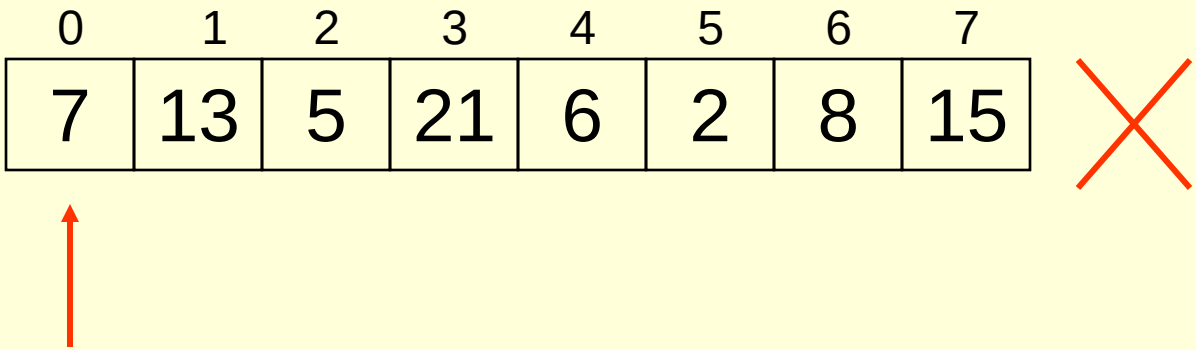
Số lần so sánh: 3

Tìm tuần tự - không tìm thấy



9

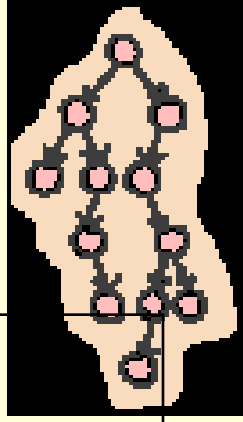
Target key



return not_present

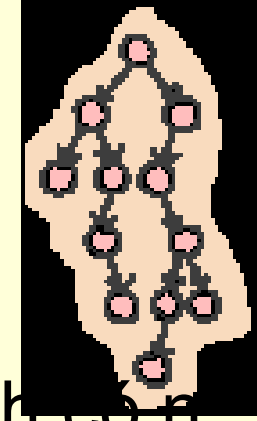
Số lần so sánh: 8

Tìm tuần tự - Mã C++



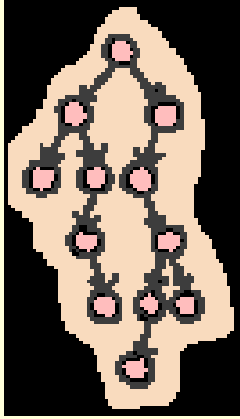
```
Error_code sequential_search(const List<Record> &the_list,
                             const Key &target, int &position)
/* Post: If an entry in the_list has key equal to target, then return success
and the output parameter position locates such an entry within the list.
Otherwise return not_present and position becomes invalid. */
{
    int s = the_list.size( );
    for (position = 0; position < s; position++) {
        Record data;
        the_list.retrieve(position, data);
        if (data == target) return success;
    }
    return not_present;
}
```

Tìm tuần tự - Đánh giá



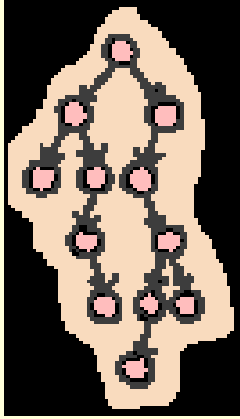
- Số lần so sánh trên khóa đối với danh sách có n phần tử:
  - Tìm không thành công: n.
  - Tìm thành công, trường hợp tốt nhất: 1.
  - Tìm thành công, trường hợp xấu nhất: n.
  - Tìm thành công, trung bình: (n + 1)/2.

Tìm trên danh sách có thứ tự



- Danh sách có thứ tự (ordered list):
  - Phần tử tại vị trí i có khóa nhỏ hơn hoặc bằng phần tử tại vị trí j (i<j).
- Tìm tuần tự có thể kết thúc sớm hơn:
  - Khi khóa cần tìm nhỏ hơn khóa của phần tử hiện tại.
- Trả giá:
  - Mỗi bước lặp cần kiểm tra xem ngừng được chưa.
  - Tốn 2 phép so sánh trên khóa cho mỗi lần lặp.
  - Số phép so sánh "có vẻ" gấp đôi so với phép tìm trên danh sách bất kỳ.

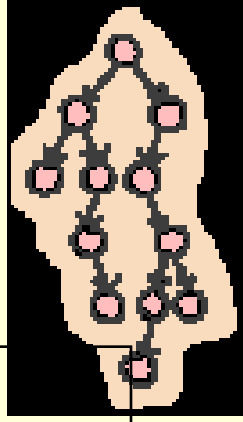
Quản lý danh sách có thứ tự



- Thừa hưởng từ List và
  - Hiệu chỉnh (override) lại các phương thức insert, replace: Đảm bảo là danh sách kết quả vẫn còn thứ tự.
  - Thiết kế thêm (overload) phương thức insert mới không cần tham số position.

```
class Ordered_list: public List<Record> {
public:
    …
    Error_code insert (const Record &data);
};
```

Thêm vào danh sách có thứ tự - Giải thuật



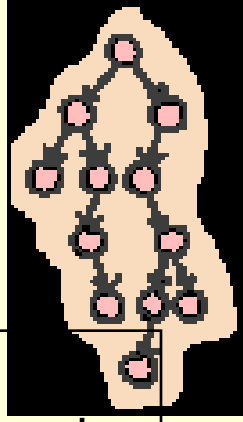
Algorithm Insert
Input: x là giá trị cần thêm vào
Output: danh sách đã thêm x vào và vẫn có thứ tự

// Đi tìm vị trí position mà khóa của x nằm giữa khóa của các phần từ
// tại vị trí position – 1 và position.

1. **for** position = 0 **to** size
   1.1. list_data = phần tử tại vị trí position
   1.2. **if** x nhỏ hơn hoặc bằng list_data
      1.2.1. **thêm** vào tại vị trí này
      1.2.2. ngừng lại

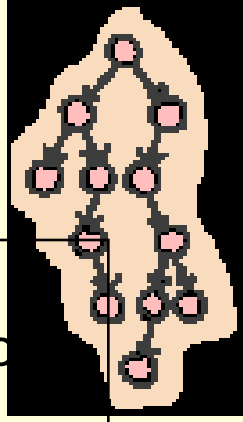
End Insert

Thêm vào danh sách có thứ tự - Mã C++



```
Error_code Ordered_list :: insert(const Record &data)
/* Post: If the Ordered_list is not full, the function succeeds: The Record
data is inserted into the list, following the last entry of the list with a strictly
lesser key (or in the rst list position if no list element has a lesser key).
Else: the function fails with the diagnostic Error_code overflow. */
{
    int s = size( );
    int position;
    for (position = 0; position < s; position++) {
        Record list_data;
        retrieve(position, list_data);
        if (data <= list_data) break;
    }
    return List<Record> :: insert(position, data);
}
```

Thêm vào danh sách (viết đè) - Giải thuật



Algorithm Insert_overridden

Input: position là vị trí cần thêm vào, x là giá trị cần thêm vào

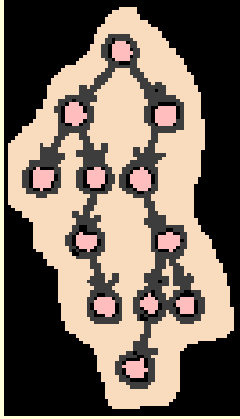
Output: danh sách đã thêm x vào và vẫn có thứ tự

// Kiểm tra xem có thỏa mãn mà khóa của x nằm giữa khóa của
// các phần từ tại vị trí position – 1 và position.

1. **if** position > 0
   1.1. list_data = phần tử tại vị trí position -1
   1.2. **if** x nhỏ hơn list_data
      1.2.1. có lỗi
2. **if** position < count
   2.1. list_data = phần tử tại vị trí position
   2.2. **if** x lớn hơn list_data
      2.2.1. có lỗi
3. **Thêm** vào tại vị trí này

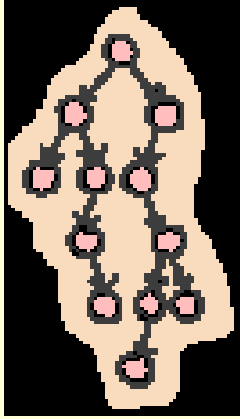
End Insert_overridden

Tìm nhị phân (binary search)



- Ý tưởng:
  - So sánh khóa cần tìm với phần tử giữa.
  - Nếu nó nhỏ hơn thì tìm bên trái danh sách.
  - Ngược lại tìm bên phải danh sách.
  - Lặp lại động tác này.
- Cần 2 chỉ mục top và bottom để giới hạn đoạn tìm kiếm trên danh sách.
- Khóa cần tìm nếu có chỉ nằm trong đoạn này.

Tìm nhị phân – Cách 2

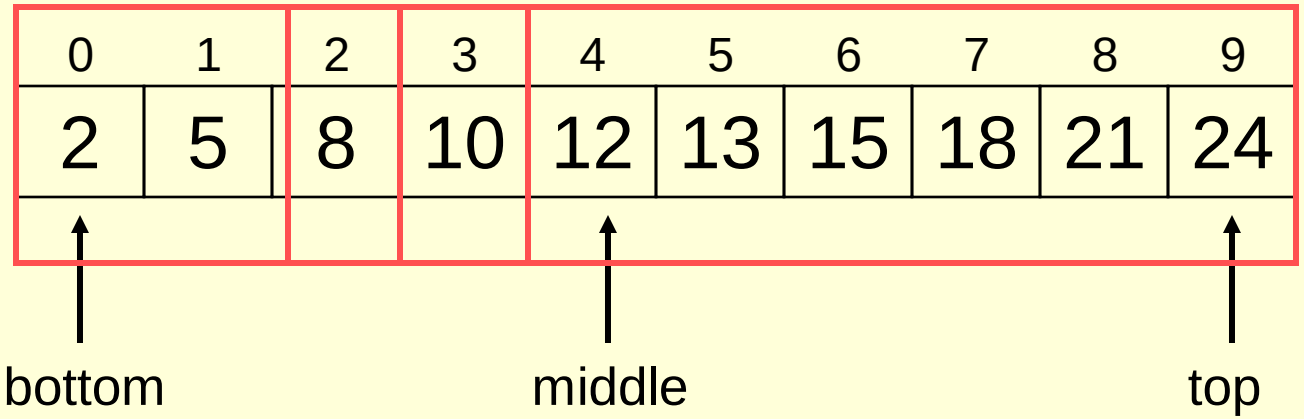


position = 3

10

Target key

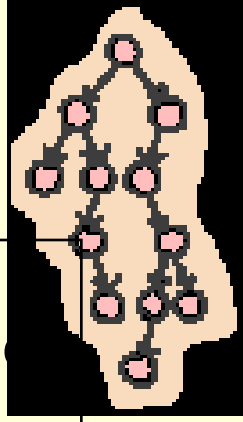
Khóa cần tìm bằng



return success

Số lần so sánh: 7

Tìm nhị phân – Giải thuật 2

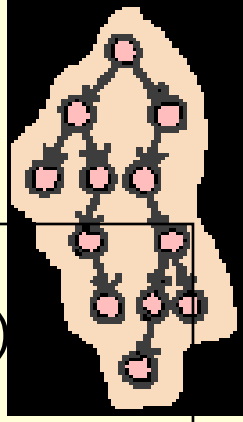


Algorithm Binary_search2
Input: target là khóa cần tìm, bottom và top là giới hạn danh sá
Output: position là vị trí nếu tìm thấy

1. **if** bottom > top
   1.1. **return** not_present
2. **if** bottom <= top
   2.1. list_data = phần tử tại vị trí mid = (bottom + top)/2
   2.2. **if** x == list_data
      2.2.1. position = mid
      2.2.2. **return** success
   2.3. **if** x < list_data
      2.3.1. **call** Binary_search2 với đoạn bên trái (bottom, mid-1)
   2.4. **else**
      2.4.1. **call** Binary_search2 với đoạn bên phải (mid+1, top)

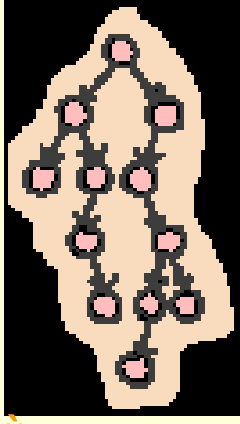
End Binary_search2

Tìm nhị phân 2 – Mã C++



```
Error_code recursive_binary_2(const Ordered_list &the list,
                              const Key &target, int bottom, int top, int &position)
   Record data;
   if (bottom <= top) {
      int mid = (bottom + top)/2;
      the_list.retrieve(mid, data);
      if (data == target) {
         position = mid;
         return success;
      }
      else if (data < target)
         return recursive_binary_2(the list, target, mid + 1, top, position);
      else
         return recursive_binary_2(the list, target, bottom, mid − 1, position);
   }
   else return not_present;
}
```

Tìm nhị phân – Cách 1

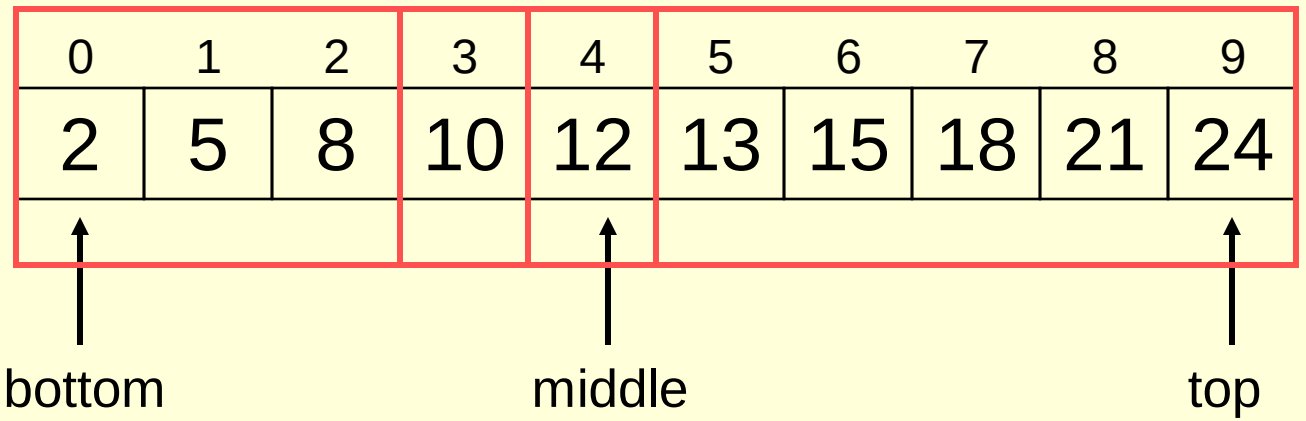


position = 3

10

Target key

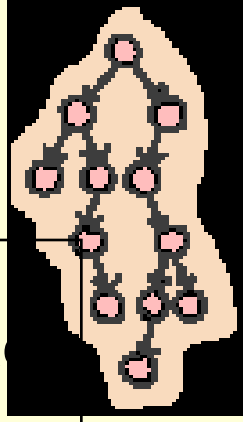
Khóa cần tìm bằng hoặc



return success

Số lần so sánh: 4

Tìm nhị phân – Giải thuật 1

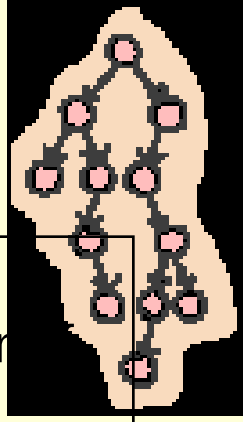


Algorithm Binary_search1
Input: target là khóa cần tìm, bottom và top là giới hạn danh sá
Output: position là vị trí nếu tìm thấy

1. **if** bottom == top
   1.1. **if** x == phần tử tại vị trí bottom
      1.1.1. position = bottom
      1.1.2. **return** success
2. **if** bottom > top
   2.1. return not_present
3. **if** bottom < top
   3.1. **if** x < phần tử tại vị trí mid = (bottom + top)/2
      3.1.1. **call** Binary_search1 với đoạn bên trái (bottom, mid-1)
   3.2. **else**
      3.2.1. **call** Binary_search1 với đoạn bên phải (mid, top)

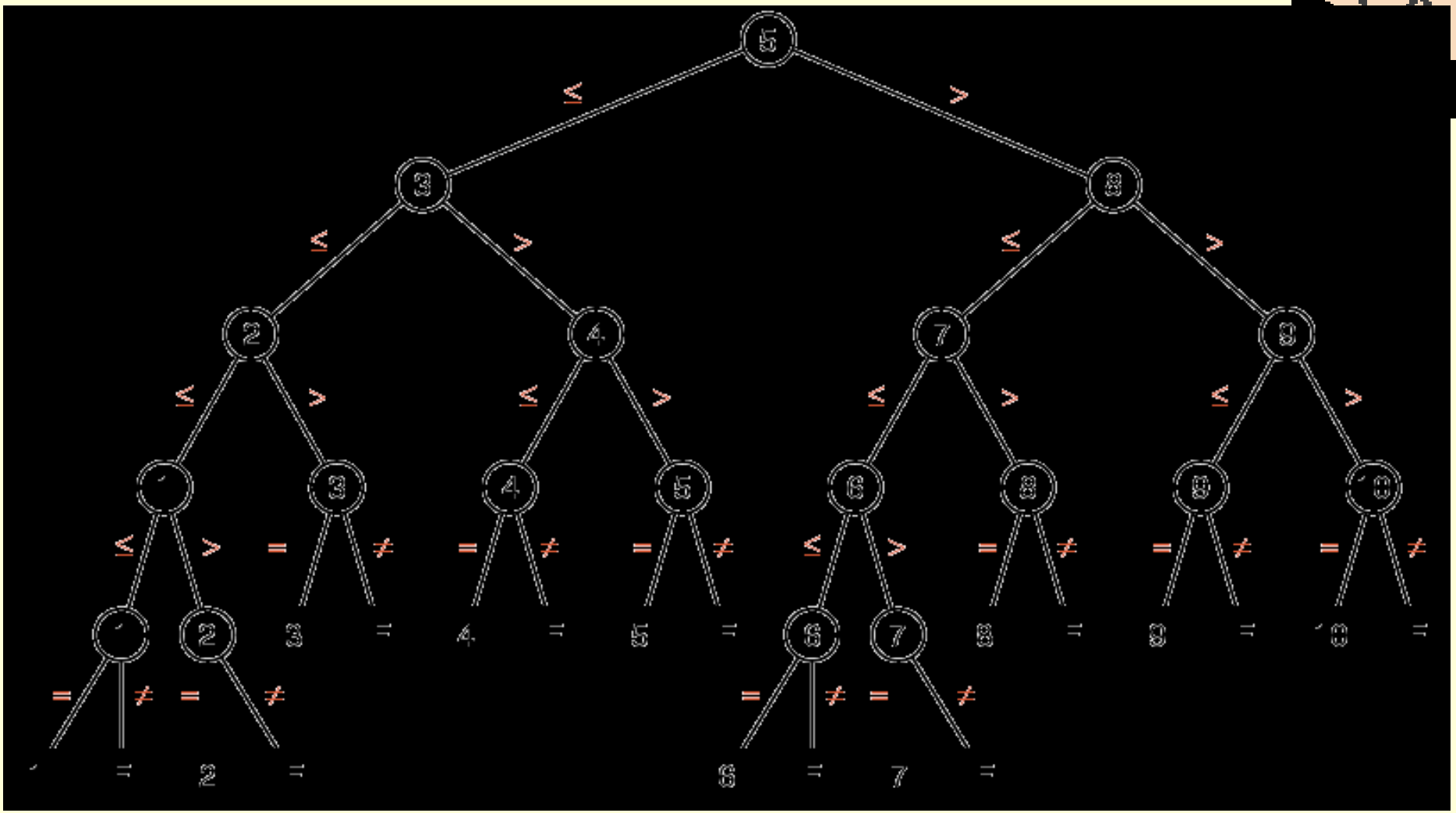
End Binary_search1

Tìm nhị phân 1 – Mã C++

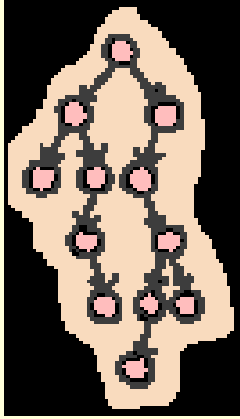


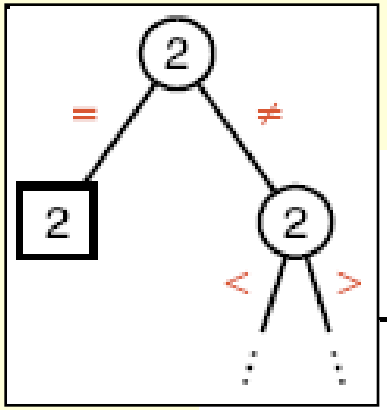
```
Error_code recursive_binary_1(const Ordered_list &the_list,
                              const Key &target, int bottom, int top, int &position
    Record data;
    if (bottom < top) { // List has more than one entry.
        the_list.retrieve((bottom + top)/2, data);
        if (data < target)
            return recursive_binary_1(the list, target, mid + 1, top, position);
        else // Reduce to bottom half of list.
            return recursive_binary_1(the list, target, bottom, mid, position);
    } else if (top < bottom)
        return not_present; // List is empty.
    else { position = bottom;
        the_list.retrieve(bottom, data);
        if (data == target) return success;
        else return not_present;
    }
}
```

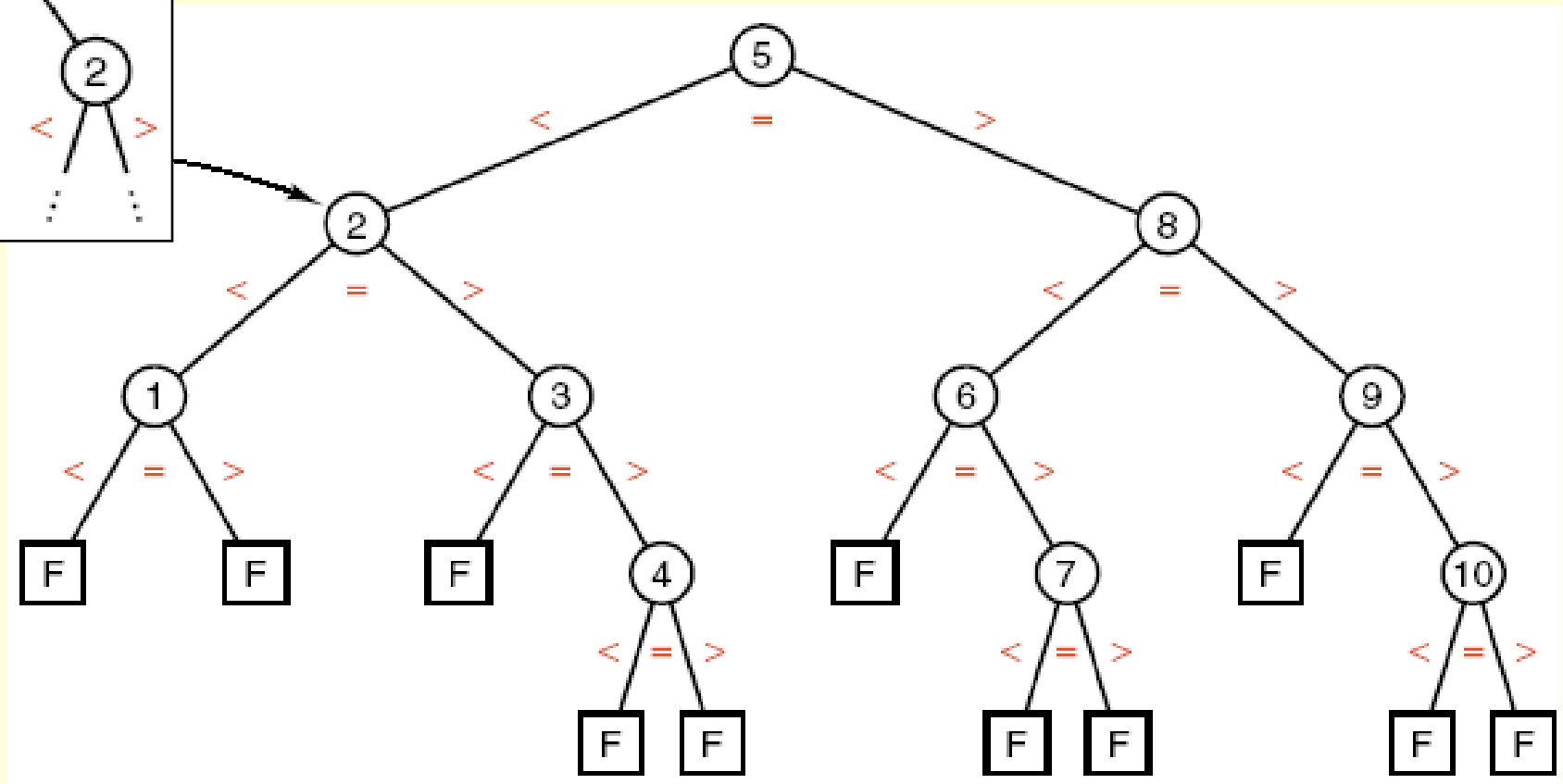
Cây so sánh của giải thuật 1



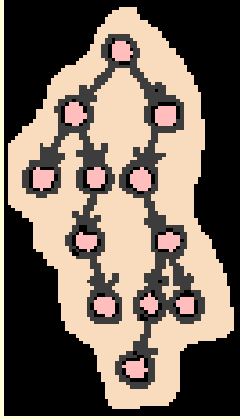
Cây so sánh của giải thuật 2





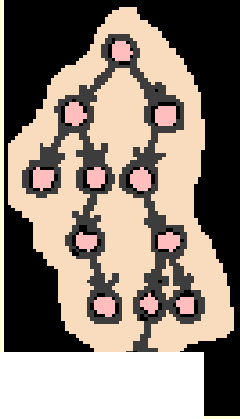


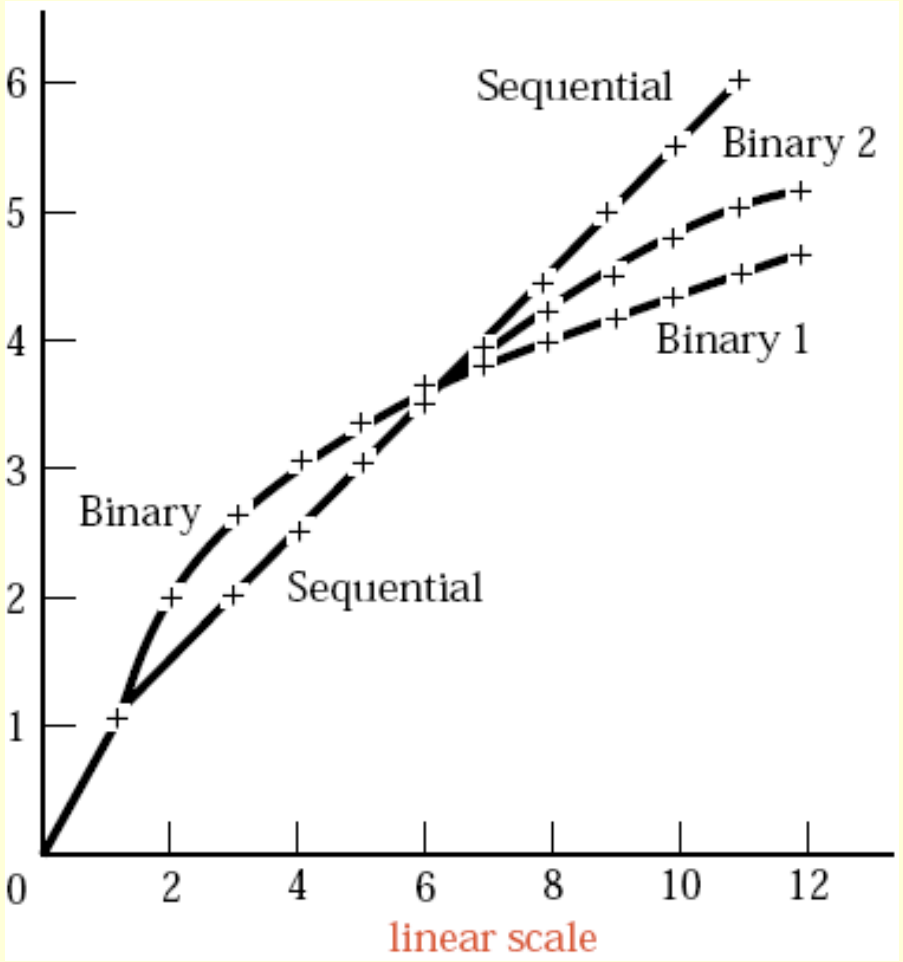
Tìm nhị phân – Đánh giá

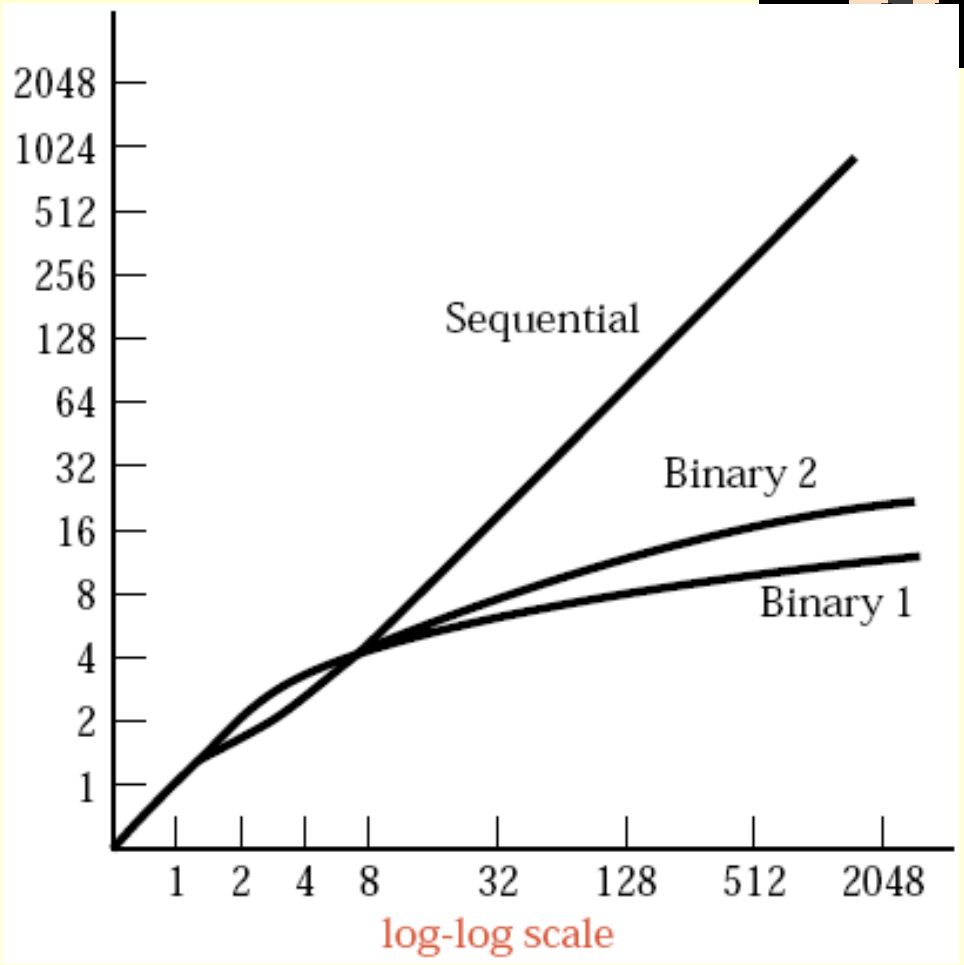


	Successful search	*Unsuccessful search*
binary_search_1	$\lg n + 1$	$\lg n + 1$
binary_search_2	$2 \lg n - 3$	$2 \lg n$

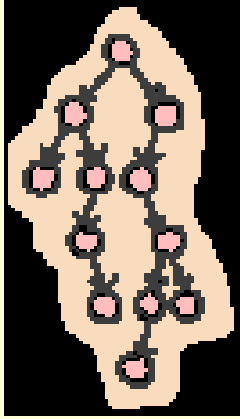
So sánh trong trường hợp trung bình các giải thuật





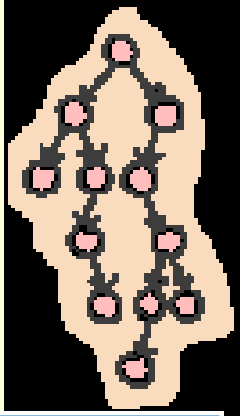


Đánh giá độ phức tạp của giải thuật



- So sánh với các hàm cơ bản:
  - $g(n) = 1$ Constant function
  - $g(n) = \log n$ Logarithmic function
  - $g(n) = n$ Linear function
  - $g(n) = n^2$ Quadratic function
  - $g(n) = n^3$ Cubic function
  - $g(n) = 2^n$ Exponential function

Độ phức tạp tính bằng tiệm cận



DEFINITION If $\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$ then:

$f(n)$ has ***strictly smaller order of magnitude*** than $g(n)$.

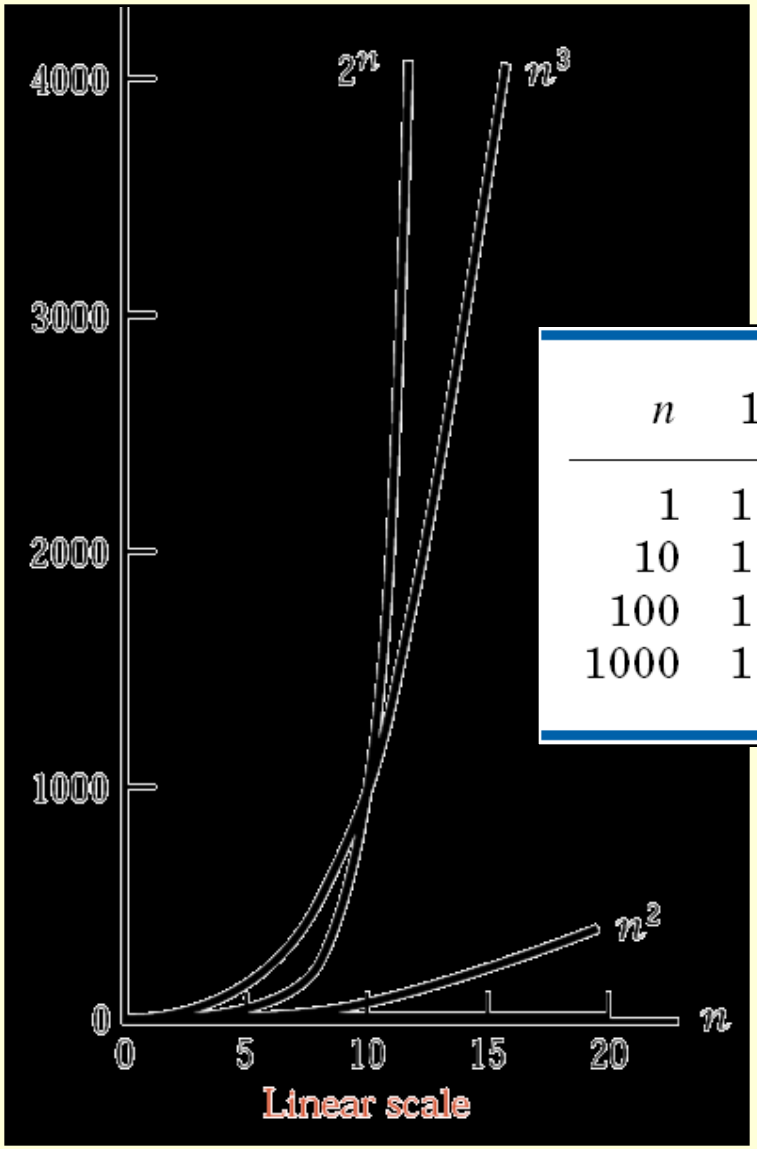
If $\lim_{n\to\infty} \frac{f(n)}{g(n)}$ is finite and nonzero then:

$f(n)$ has ***the same order of magnitude*** as $g(n)$.

If $\lim_{n\to\infty} \frac{f(n)}{g(n)} = \infty$ then:

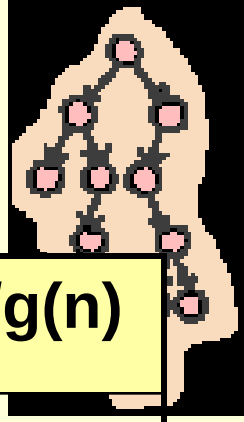
$f(n)$ has ***strictly greater order of magnitude*** than $g(n)$.

Độ tăng của các hàm chung



n	1	$\lg n$	n	$n \lg n$	n^2	n^3	2^n
1	1	0.00	1	0	1	1	2
10	1	3.32	10	33	100	1000	1024
100	1	6.64	100	664	10,000	1,000,000	1.27×10^{30}
1000	1	9.97	1000	9970	1,000,000	10^9	1.07×10^{301}

Ký hiệu Big-O



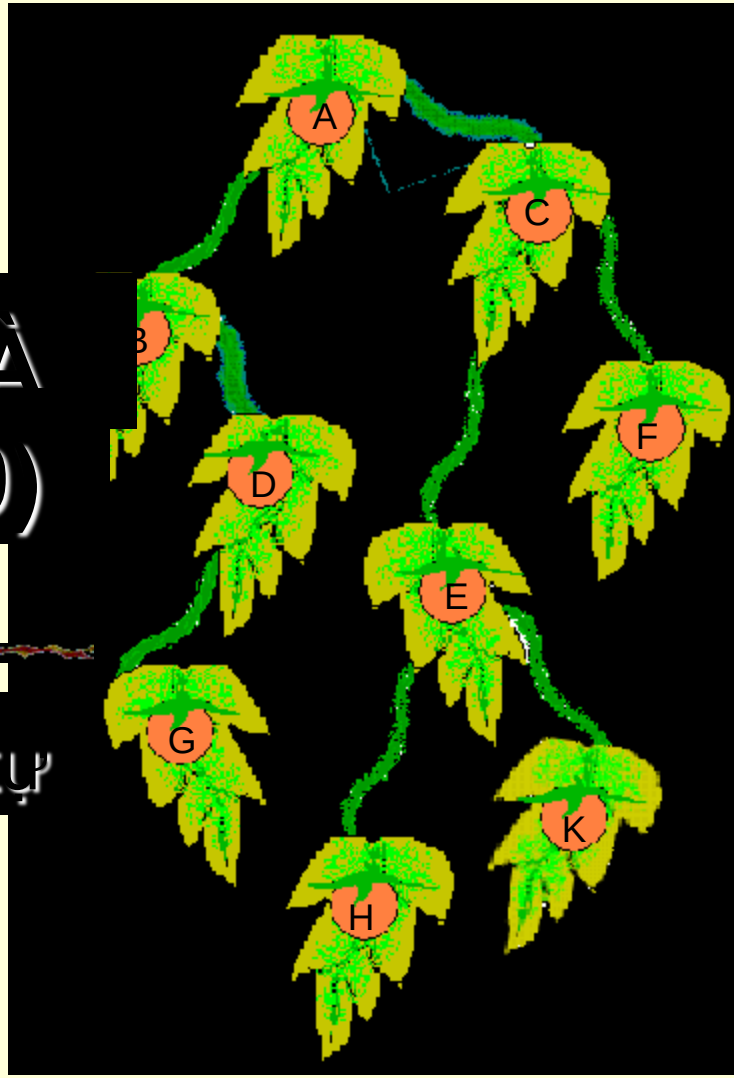
f(n) ∈	Bậc của f so với g	$\lim_{n\to\infty} (f(n)/g(n)$
$o(g(n))$	< : nhỏ hơn hẳn	0
$O(g(n))$	≤ : nhỏ hơn hoặc bằng	a
$\Theta(g(n))$	= : bằng	a ≠ 0
$\Omega(g(n))$	≥ : lớn hơn hoặc bằng	a ≠ 0 hoặc là ∞

Thứ tự tăng dần về độ lớn:

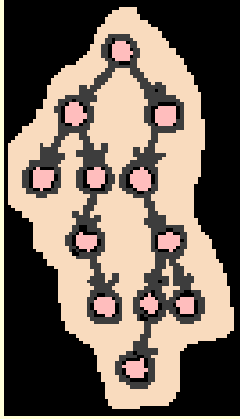
$O(1)$ $O(\lg n)$ $O(n)$ $O(n \lg n)$ $O(n^2)$ $O(n^3)$ $O(2^n)$

CẤU TRÚC DỮ LIỆU VÀ GIẢI THUẬT (501040)

Chương 8: Sắp thứ tự

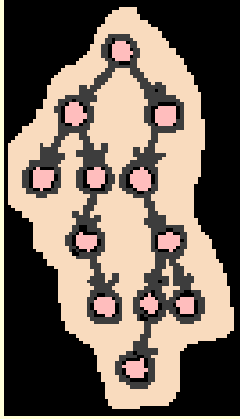


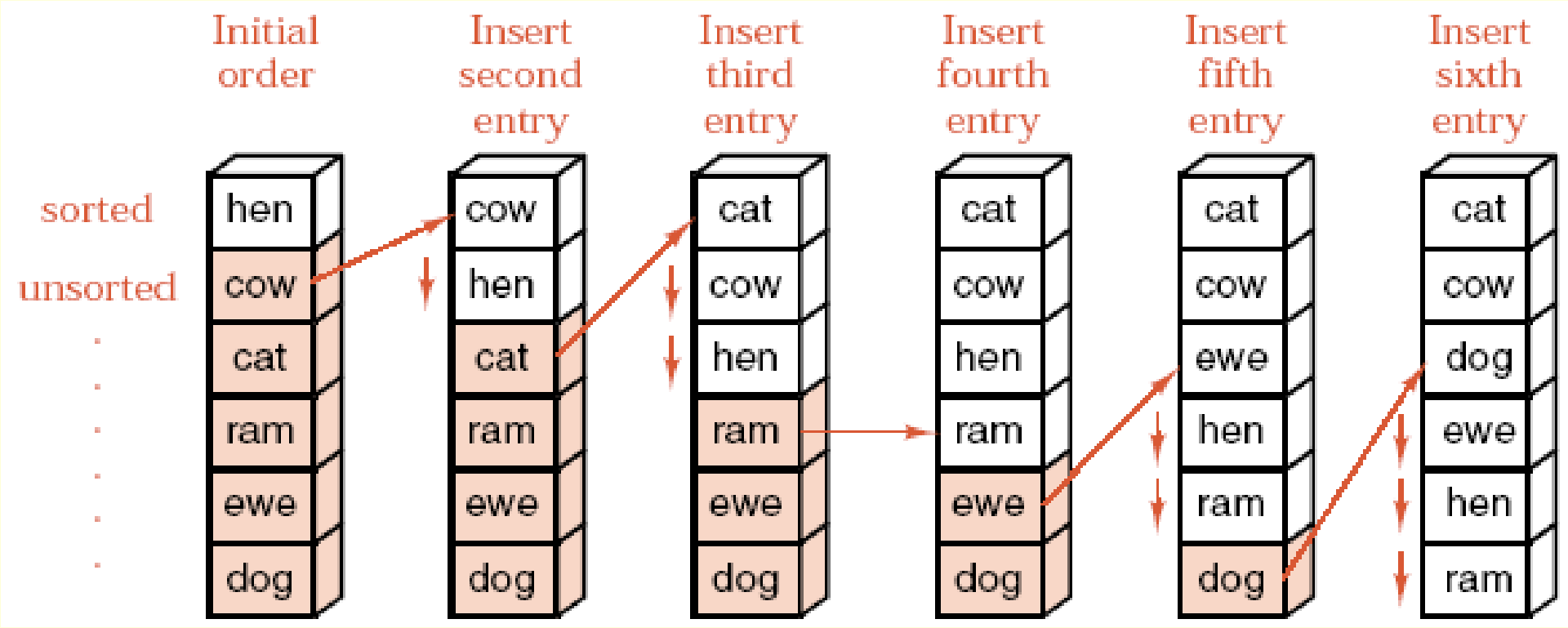
Khái niệm



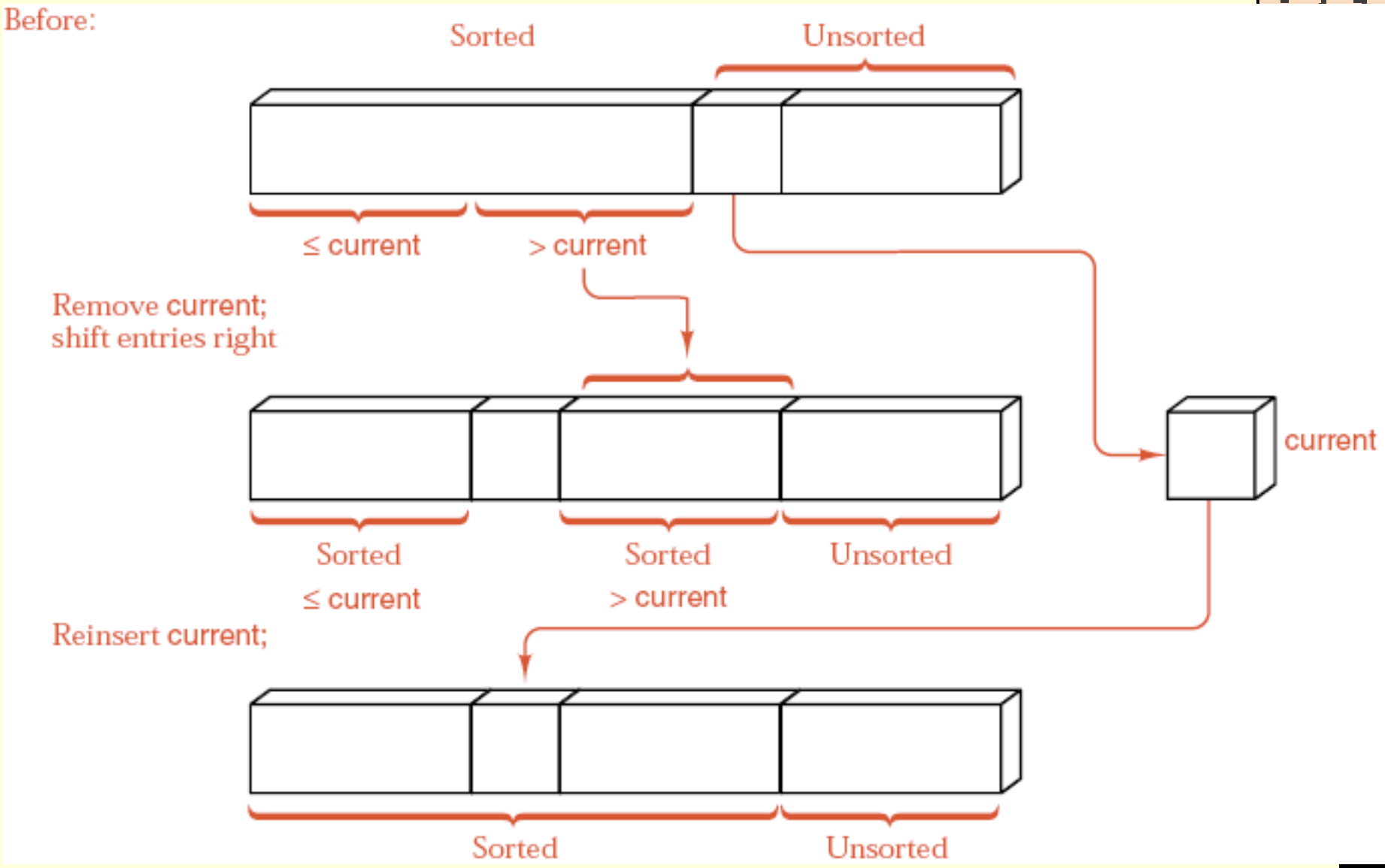
- Sắp thứ tự:
  - Đầu vào: một danh sách
  - Đầu ra: danh sách có thứ tự tăng (hoặc giảm) trên khóa
- Phân loại:
  - Sắp thứ tự ngoại (external sort): tập tin
  - Sắp thứ tự nội (internal sort): bộ nhớ
- Giả thiết:
  - Sắp thứ tự nội
  - Sắp tăng dần

Insertion sort

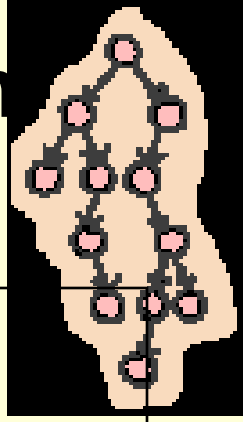




Insertion sort - Danh sách liên tục



Giải thuật insertion sort – Danh sách liên tục

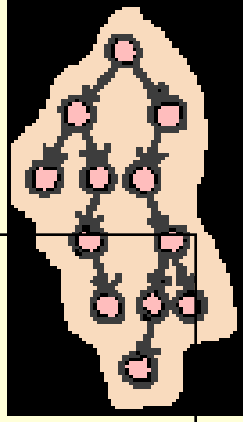


Algorithm Insertion_sort
Input: danh sách cần sắp thứ tự
Output: danh sách đã được sắp thứ tự

1. **for** first_unsorted = 1 to size
//Tìm vị trí hợp lý để chèn giá trị đang có vào
1.1. current = list[first_unsorted]
1.2. position = first_unsorted
1.3. **while** (position>0 **and** list[position - 1] > current)
//Dời chỗ các phần tử lớn về sau
1.3.1. list[position] = list[position - 1]
1.3.2. position = position - 1
//Chép phần tử trước đó vào đúng vị trí của nó
1.4. list[position - 1] = current

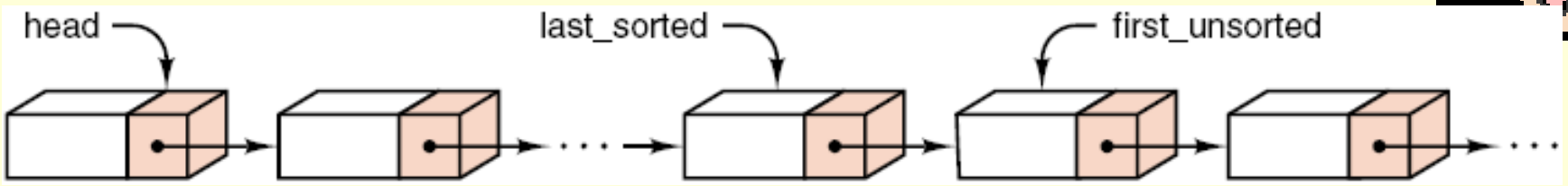
End Insertion_sort

Mã C++ Insertion sort – Danh sách liên tục

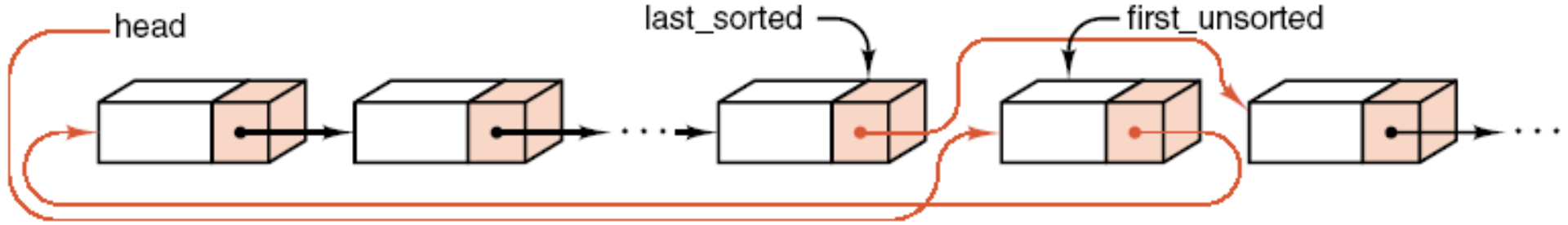


```
template <class Record>
void Sortable_list<Record> :: insertion_sort( ) {
    int first_unsorted;     // position of first unsorted entry
    int position;           // searches sorted part of list
    Record current;         // holds the entry temporarily removed from list
    for (first_unsorted = 1; first_unsorted < count; first_unsorted++)
        if (entry[first_unsorted] < entry[first_unsorted − 1]) {
            position = first_unsorted;
            current = entry[first_unsorted];    // Pull unsorted entry out of the list.
            do {
                // Shift all entries until the proper position is found.
                entry[position] = entry[position − 1];
                position−−;                     // position is empty.
            } while (position > 0 && entry[position − 1] > current);
            entry[position] = current;
    }
}
```

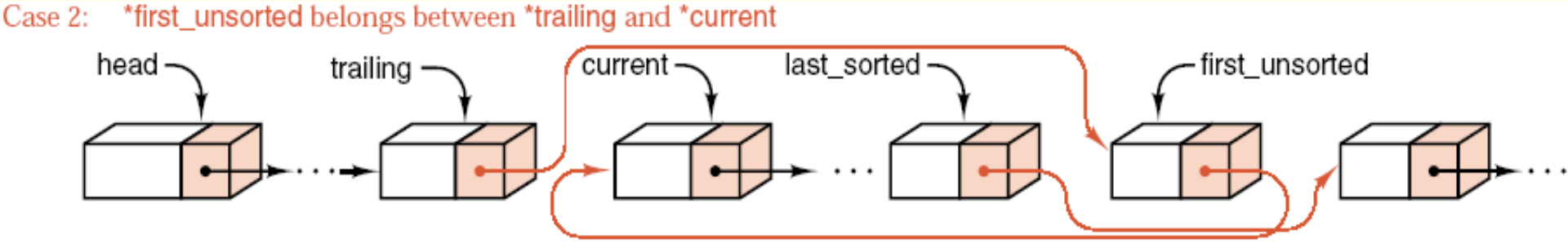
Insertion sort – DSLK



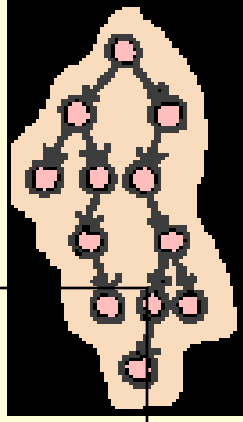
Case 1: *first_unsorted belongs at head of list



Case 2: *first_unsorted belongs between *trailing and *current



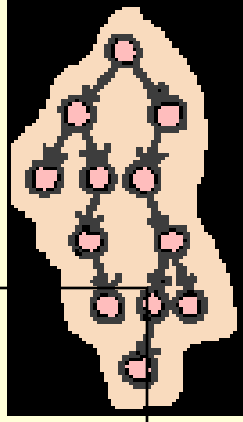
Giải thuật Insertion sort - DSLK



Algorithm Insertion_sort
Input: danh sách cần sắp thứ tự và có ít nhất 1 phần tử
Output: danh sách đã được sắp thứ tự

1. last_sorted = head
   //Đi dọc danh sách liên kết
2. **while** (last_sorted chưa là phần tử cuối)
   2.1. first_unsorted là phần tử kế của last_sorted
   //Chèn vào đầu?
   2.2. **if** (dữ liệu của first_unsorted < dữ liệu của head)
      //Chèn vào đầu
      2.2.1. Gỡ first_unsorted ra khỏi danh sách
      2.2.2. Nối first_unsorted vào đầu danh sách
      2.2.3. head = first_unsorted

Giải thuật Insertion sort – DSLK (tt.)



2.3. **else**

//Tìm vị trí hợp lý để chèn giá trị đang có vào

2.3.1. tailing = head

2.3.2. current là phần tử kế của tailing

2.3.3. **while** (dữ liệu của first_unsorted > dữ liệu của current)

2.3.3.1. Di chuyển tailing và current đến phần tử kế

2.3.4. **if** (first_unsorted chính là current)

2.3.4.1. last_sorted = current *//Đã đúng vị trí rồi*

2.3.5. **else**

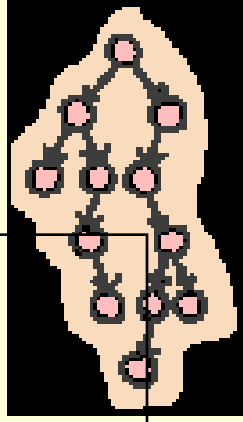
2.3.4.1. Gỡ first_unsorted ra khỏi danh sách

2.3.4.2. Nối first_unsorted vào giữa tailing và current

2.4. Di chuyển last_sorted đến phần tử kế

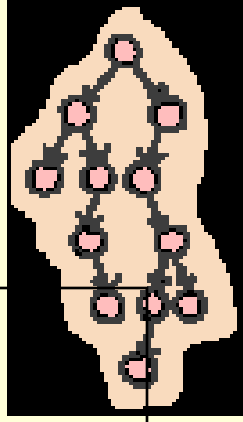
End Insertion_sort

Mã C++ Insertion sort - DSLK



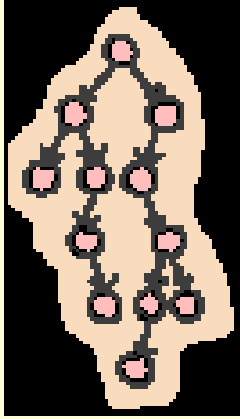
```
template <class Record>
void Sortable_list<Record> :: insertion_sort( ) {
    Node <Record> *first_unsorted, *last_sorted, *current, *trailing;
    if (head != NULL) {
        last_sorted = head;
        while (last_sorted->next != NULL) {
            first_unsorted = last sorted->next;
            if (first_unsorted->entry < head->entry) {
                last_sorted->next = first_unsorted->next;
                first_unsorted->next = head;
                head = first_unsorted; }
            else { trailing = head;
                current = trailing->next;
                while (first_unsorted->entry > current->entry) {
                    trailing = current;
                    current = trailing->next;
                }
```

Mã C++ Insertion sort – DSLK (tt.)



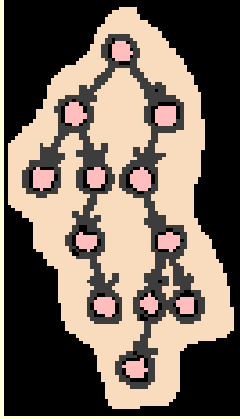
```
            if (first_unsorted == current)
                last_sorted = first_unsorted;
            else {
                last_sorted->next = first_unsorted->next;
                first_unsorted->next = current;
                trailing->next = first_unsorted;
            }
        }
    }
}
```

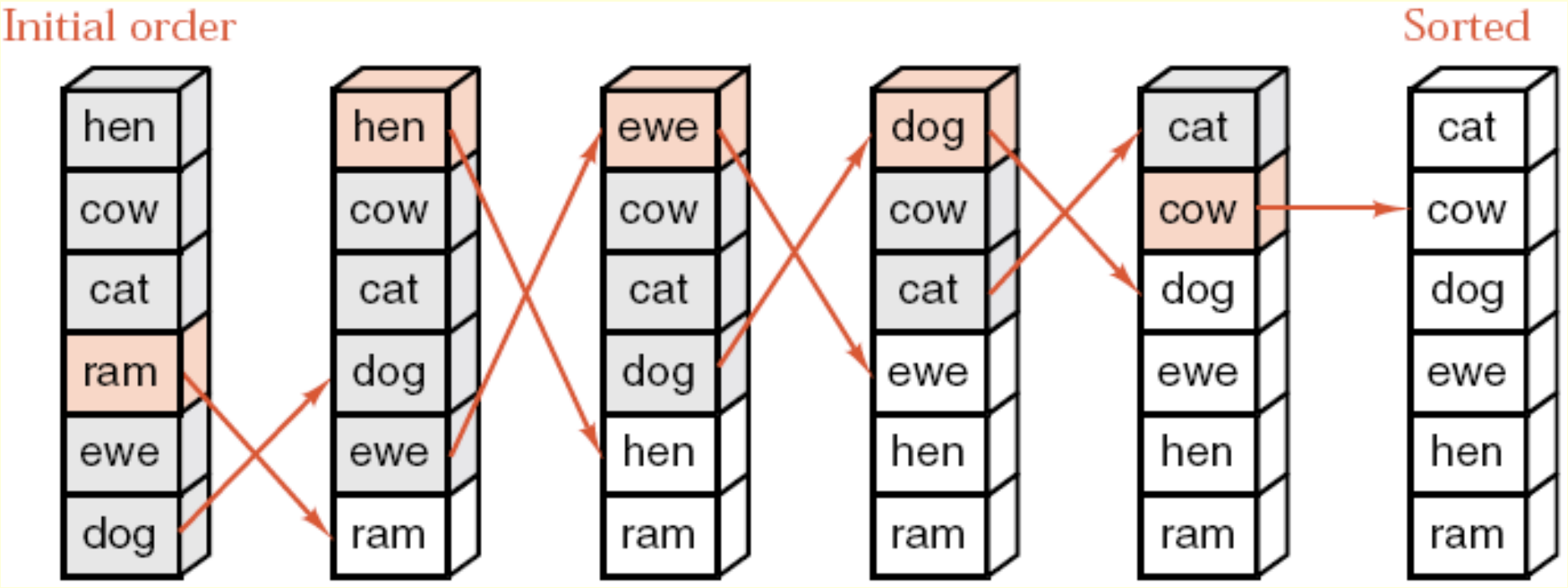
Đánh giá Insertion sort



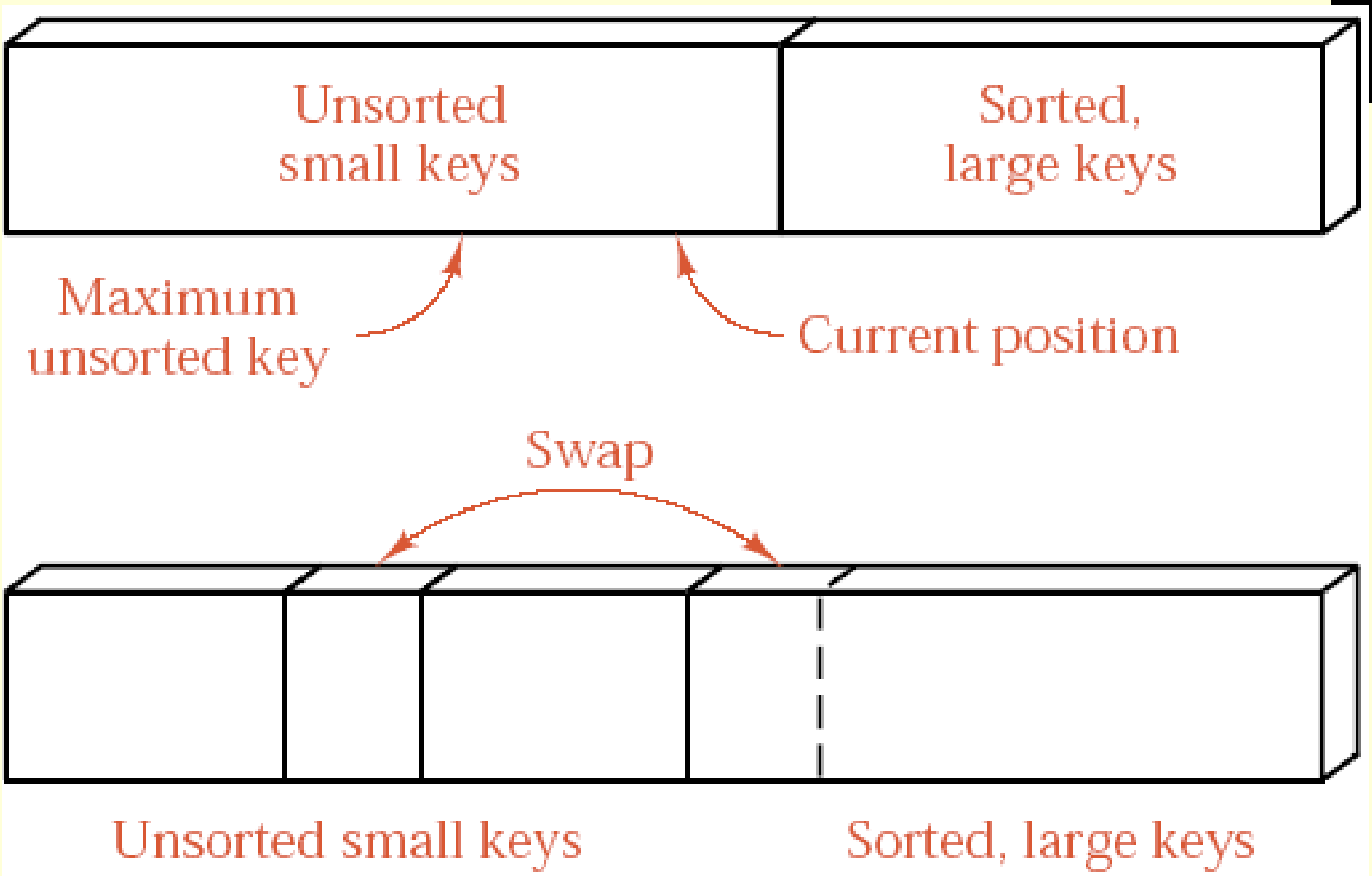
- Danh sách có thứ tự ngẫu nhiên:
  - So sánh trung bình là $n^2/4 + O(n)$
  - Dời chỗ trung bình là $n^2/4 + O(n)$
- Danh sách có thứ tự tăng dần: tốt nhất
  - So sánh n-1 lần
  - Dời chỗ 0 lần
- Danh sách có thứ tự giảm dần: tệ nhất
  - So sánh $n^2/2 + O(n)$ lần
  - Dời chỗ $n^2/2 + O(n)$ lần

Selection sort

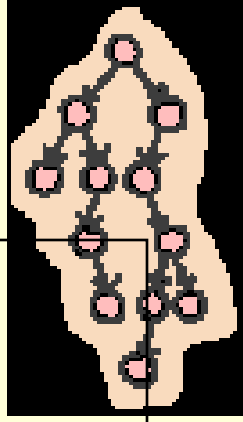




Selection sort – Danh sách liên tục



Giải thuật Selection sort



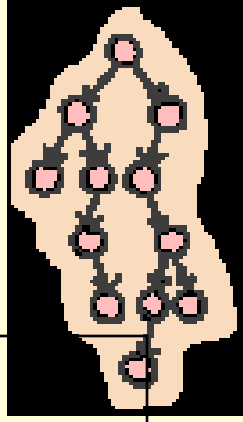
Algorithm Selection_sort
Input: danh sách cần sắp thứ tự
Output: danh sách đã được sắp thứ tự

1. **for** position = size − 1 **downto** 0
//Tìm vị trí phần tử có khóa lớn nhất trong phần chưa sắp thứ tự
1.1. max = 0 *//Giả sử phần tử đó ở tại 0*
1.2. **for** current = 1 **to** position *//Xét các phần tử còn lại*
1.2.1. **if** (list[current] > list[max]) *//Nếu có phần tử nào lớn hơn*
1.2.1.1. max = current *//thì giữ lại vị trí đó*

//Đổi chỗ phần tử này với phần tử đang xét
1.3. temp = list[max]
1.4. list[max] = list[position]
1.5. list[position] = temp

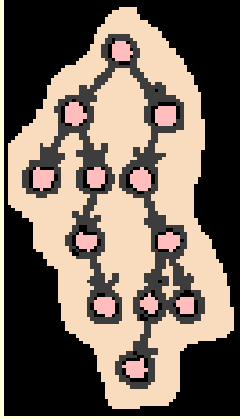
End Selection_sort

Mã C++ Selection sort



```
template <class Record>
void Sortable_list<Record> :: selection_sort( ) {
    Record temp;
    for (int position = count − 1; position > 0; position−−) {
        int largest = 0;
        for (int current = 1; current <= position; current++)
            if (entry[largest] < entry[current])
                largest = current;
        temp = entry[max];
        entry[max] = entry[position];
        entry[position] = temp;
    }
}
```

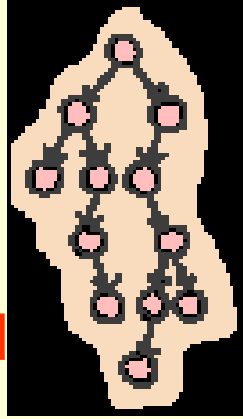
Đánh giá Selection sort

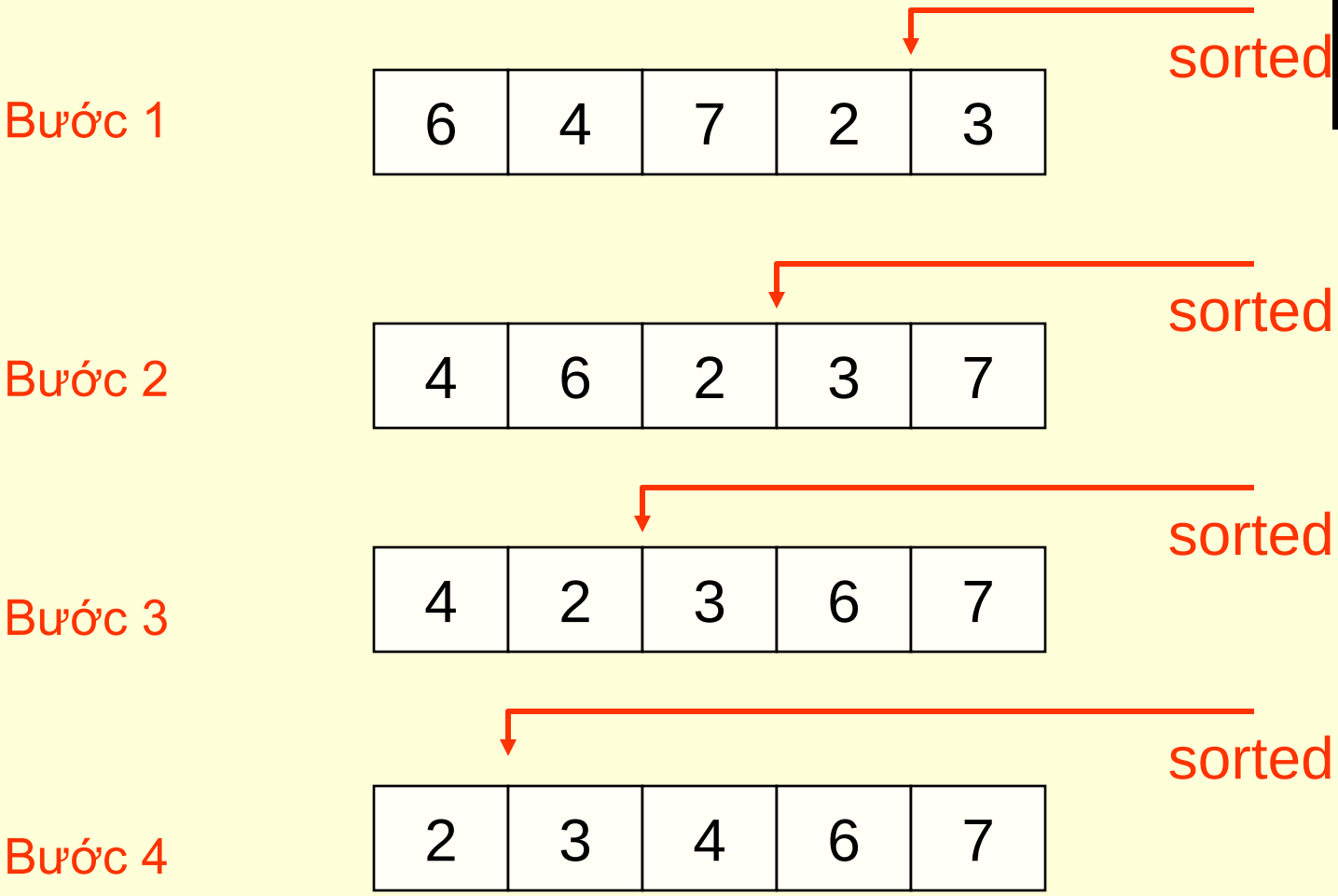


- Danh sách bất kỳ
  - Số lần so sánh: n(n-1)/2
  - Số lần dời chỗ: 3n
- So sánh với insertion sort:

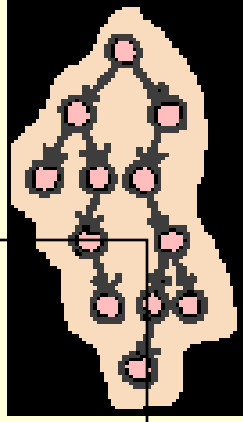
	Selection	*Insertion (average)*
Assignments of entries	$3.0n + O(1)$	$0.25n^2 + O(n)$
Comparisons of keys	$0.5n^2 + O(n)$	$0.25n^2 + O(n)$

Bubble sort




Bước 1
6 4 7 2 3
sorted
Bước 2
4 6 2 3 7
sorted
Bước 3
4 2 3 6 7
sorted
Bước 4
2 3 4 6 7
sorted

Giải thuật Bubble sort

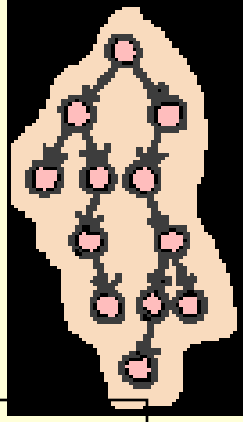


Algorithm Bubble_sort
Input: danh sách cần sắp thứ tự
Output: danh sách đã được sắp thứ tự

1. **for** step = 1 **to** size-1
//Với mỗi cặp phần tử kề bất kỳ, sắp thứ tự chúng.
//Sau mỗi bước phần tử cuối của danh sách hiện tại là lớn nhất,
//vì vậy được trừ ra cho bước kế tiếp
1.1. **for** current = 1 **to** (size - step)
//Nếu cặp phần tử kề hiện tại không đúng thứ tự
1.1.1. **if** (list[current] < list[current-1])
//Đổi chỗ chúng
1.1.1.1. temp = list[current]
1.1.1.2. list[current] = list[current-1]
1.1.1.3. list[current-1] = temp

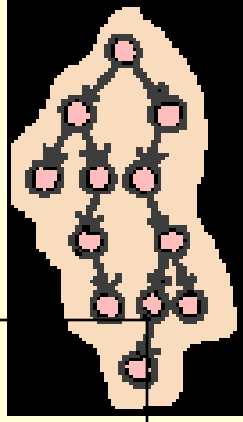
End Bubble_sort

Mã C++ Bubble sort



```
template <class Record>
void Sortable_list<Record> :: bubble_sort( ) {
    Record temp;
    for (int position = count − 1; position > 0; position−−)
        for (int current = 1; current < position; current++)
            if (entry[current] < entry[current-1]) {
                temp = entry[current];
                entry[current] = entry[current-1];
                entry[current-1] = temp;
            }
}
```

Bubble sort là exchange sort

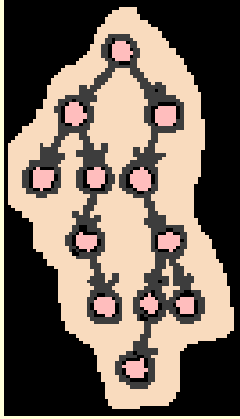


Algorithm Exchange_sort
Input: danh sách cần sắp thứ tự
Output: danh sách đã được sắp thứ tự

1. exchanged = true
2. **while** exchanged
   //Giả sử lần lặp này không có sự đổi chỗ thì nó đã có thứ tự
   2.1. exchanged = false
   2.2. **for** current = 1 **to** size – 1
      //Nếu cặp này không có thứ tự thì đổi chỗ và ghi nhận lại
      2.2.1. **if** (list[current] < list[current-1])
         2.2.1.1. exchange (current, current-1)
         2.2.1.2. exchanged = true

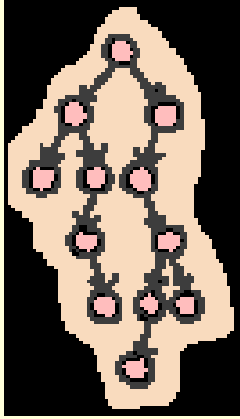
End Exchange_sort

Đánh giá Bubble sort



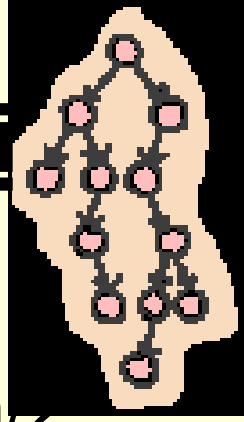
- Số lần so sánh: n(n-1)/2
- Số lần dời chỗ:
  - Danh sách có thứ tự tăng dần: tốt nhất là 0 lần
  - Danh sách có thứ tự giảm dần: tệ nhất là 3*n(n-1)/2

Chia để trị



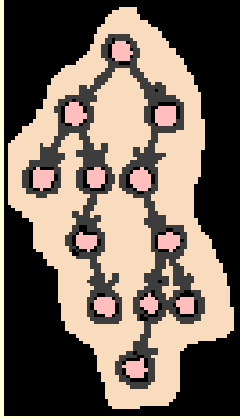
- Ý tưởng:
  - Chia danh sách ra làm 2 phần
  - Sắp thứ tự riêng cho từng phần
  - Trộn 2 danh sách riêng đó thành danh sách có thứ tự
- Hai giải thuật:
  - Merge sort:
    - Chia đều thành 2 danh sách
    - Sắp thứ tự riêng
    - Trộn lại
  - Quick sort:
    - Chia thành 3 phần: nhỏ, giữa (pivot), lớn
    - Sắp thứ tự riêng

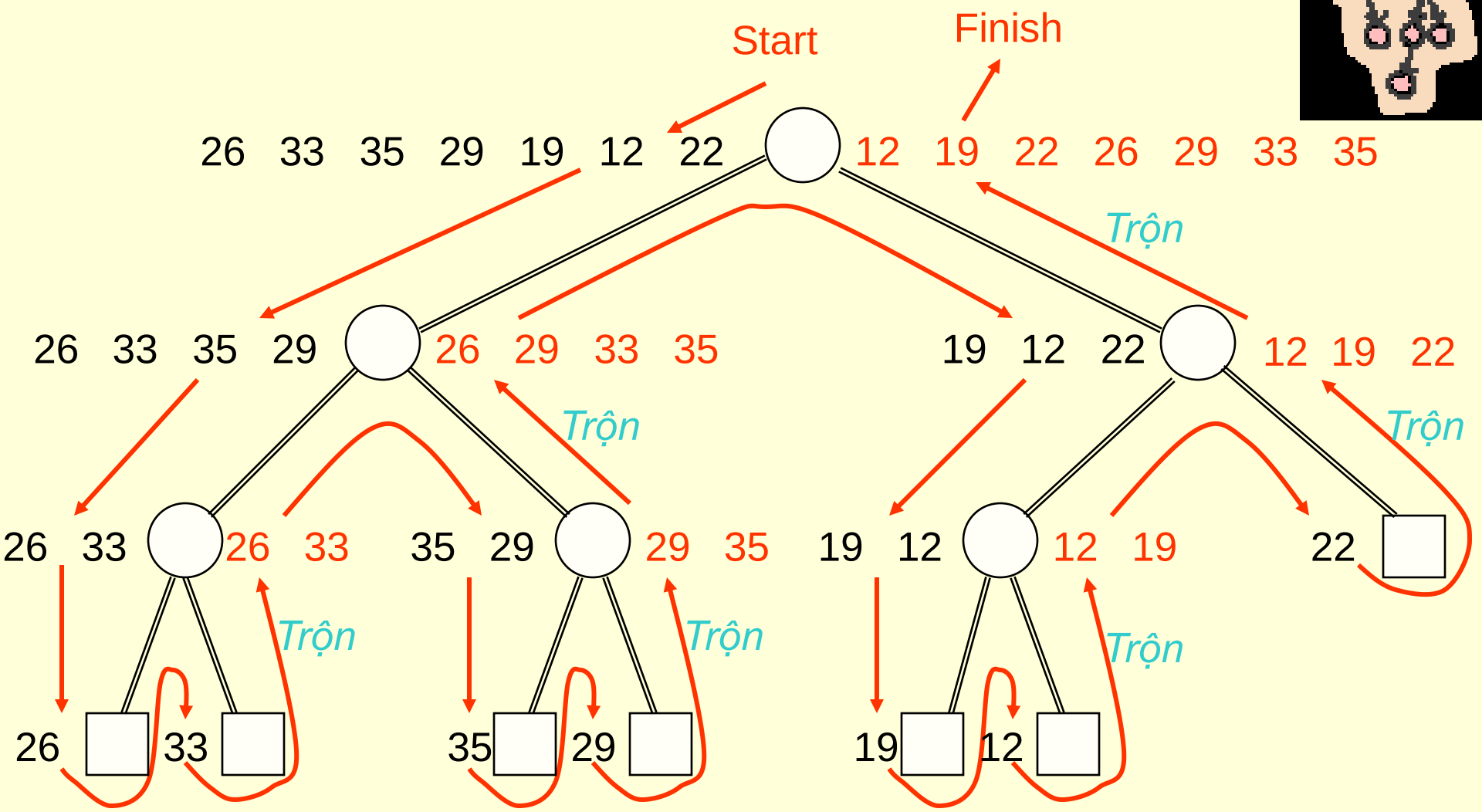
Đánh giá sơ giải thuật chia để trị



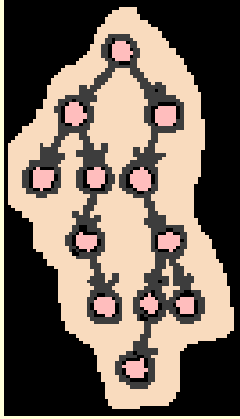
- Giả sử 2 danh sách có số phần tử là n' = n/2
- Dùng insertion sort riêng cho từng mảnh
- Trộn 2 danh sách tốn (n' + n') = n lần so sánh
- Số lần so sánh tổng cộng: $2*((n/2)^2/2 + O(n/2)) + n = n^2/4 + O(n)$
- So sánh với insertion sort là $n^2/2 + O(n)$
- Có vẻ nhanh hơn

Merge sort




Start
Finish
26 33 35 29 19 12 22
12 19 22 26 29 33 35
Trộn
26 33 35 29
26 29 33 35
19 12 22
12 19 22
Trộn
Trộn
26 33
26 33
35 29
29 35
19 12
12 19
22
Trộn
Trộn
Trộn
26
33
35
29
19
12

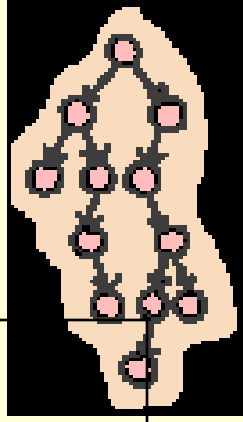
Đánh giá Merge sort



- Độ phức tạp:
  - Có tối đa *lgn* mức
  - Ở mỗi mức, cần trộn n phần tử
- Hạn chế:
  - Danh sách liên tục: di chuyển các phần tử nhiều
  - Nên dùng trong DSLK

Lower bound:	$\lg n! \approx n \lg n - 1.44n + O(\log n)$
Mergesort:	$n \lg n - 1.1583n + 1,$
Insertion sort:	$\frac{1}{4}n^2 + O(n)$

Giải thuật Merge sort - DSLK

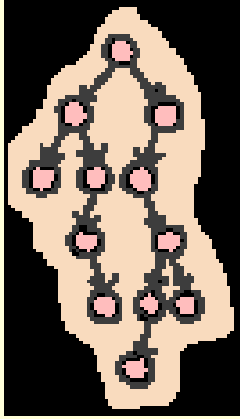


Algorithm Merge_sort
Input: danh sách cần sắp thứ tự
Output: danh sách đã được sắp thứ tự

1. **if** (có ít nhất 2 phần tử)
   //Chia danh sách ra 2 phần bằng nhau:
   1.1. second_half = divide_from (sub_list)
   //Sắp xếp riêng từng phần
   1.2. **Call** Merge_sort với sub_list
   1.3. **Call** Merge_sort với second_half
   //Trộn hai phần này với nhau
   1.4. **Call** Merge với sub_list và second_half

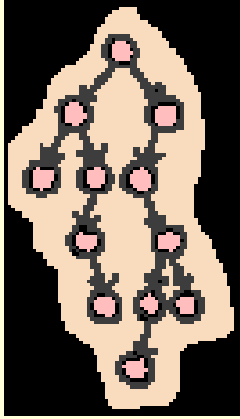
End Merge_sort

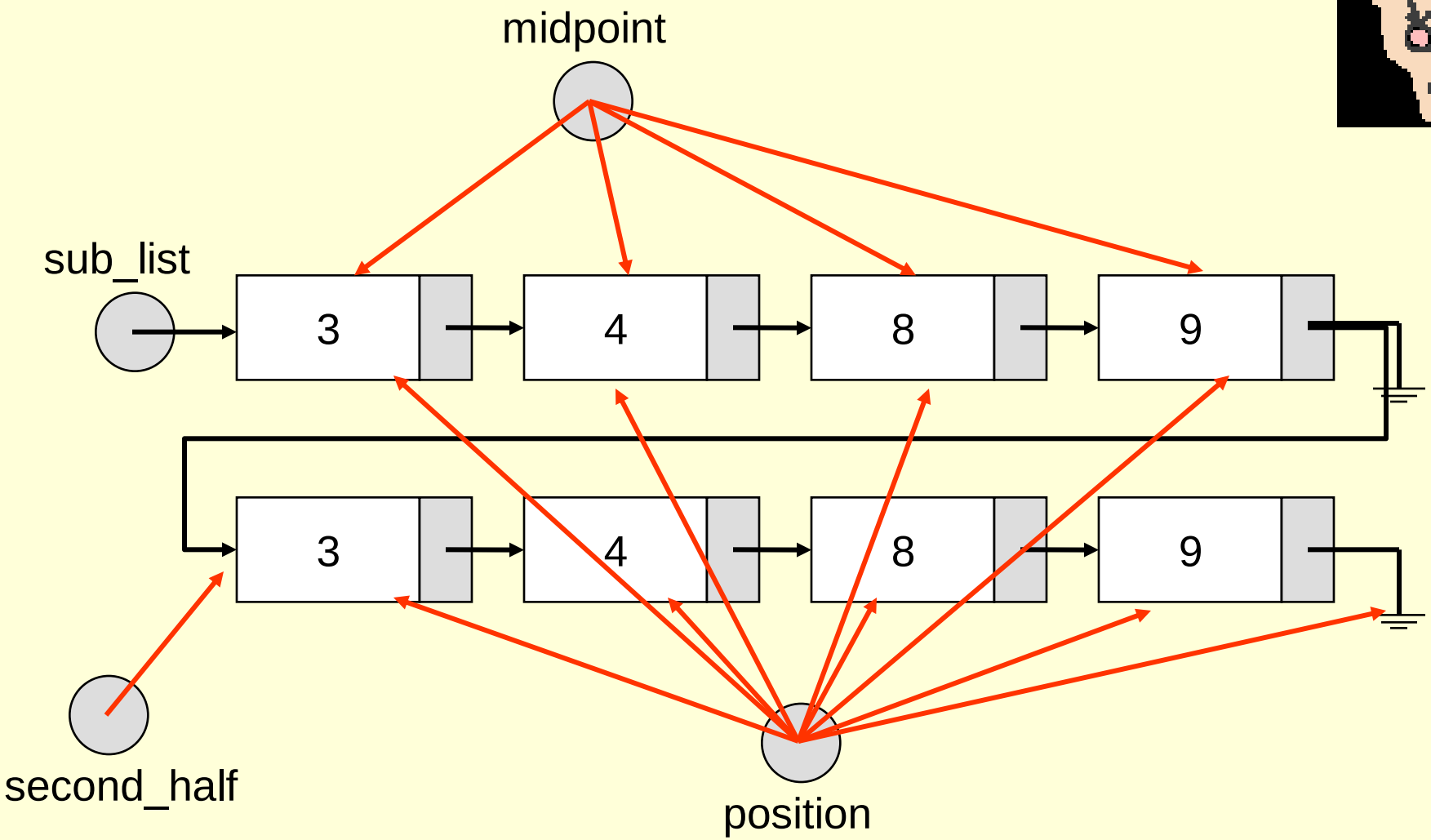
Mã C++ Merge sort



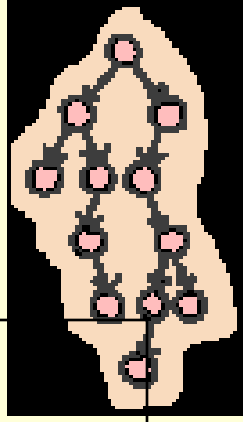
```
template <class Record>
void Sortable_list<Record> ::
                        recursive_merge_sort (Node<Record> * &sub_list) {
    if (sub_list != NULL && sub_list->next != NULL) {
        Node<Record> *second_half = divide_from(sub_list);
        recursive_merge_sort(sub_list);
        recursive_merge_sort(second_half);
        sub_list = merge(sub_list, second_half);
    }
}
```

Chia đôi DSLK





Giải thuật chia đôi DSLK

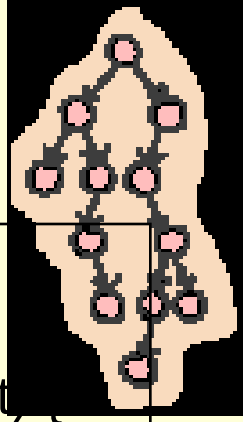


Algorithm divide_from
Input: danh sách cần chia đôi
Output: hai danh sách dài gần bằng nhau

1. **if** (có ít nhất 1 phần tử)
   //Dùng một con trỏ di chuyển nhanh gấp đôi con trỏ còn lại
   1.1. midpoint = sub_list
   1.2. position là phần tử kế của midpoint
   1.3. **while** (position is not NULL)
      1.3.1. Di chuyển position đến phần tử kế 2 lần
      1.3.2. Di chuyển midpoint đến phần tử kế
   1.4. Cắt danh sách ra 2 phần tại vị trí này

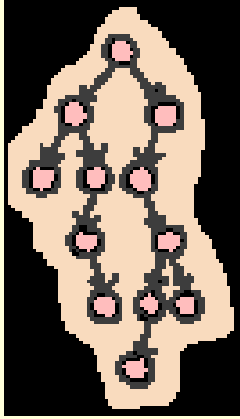
End divide_from

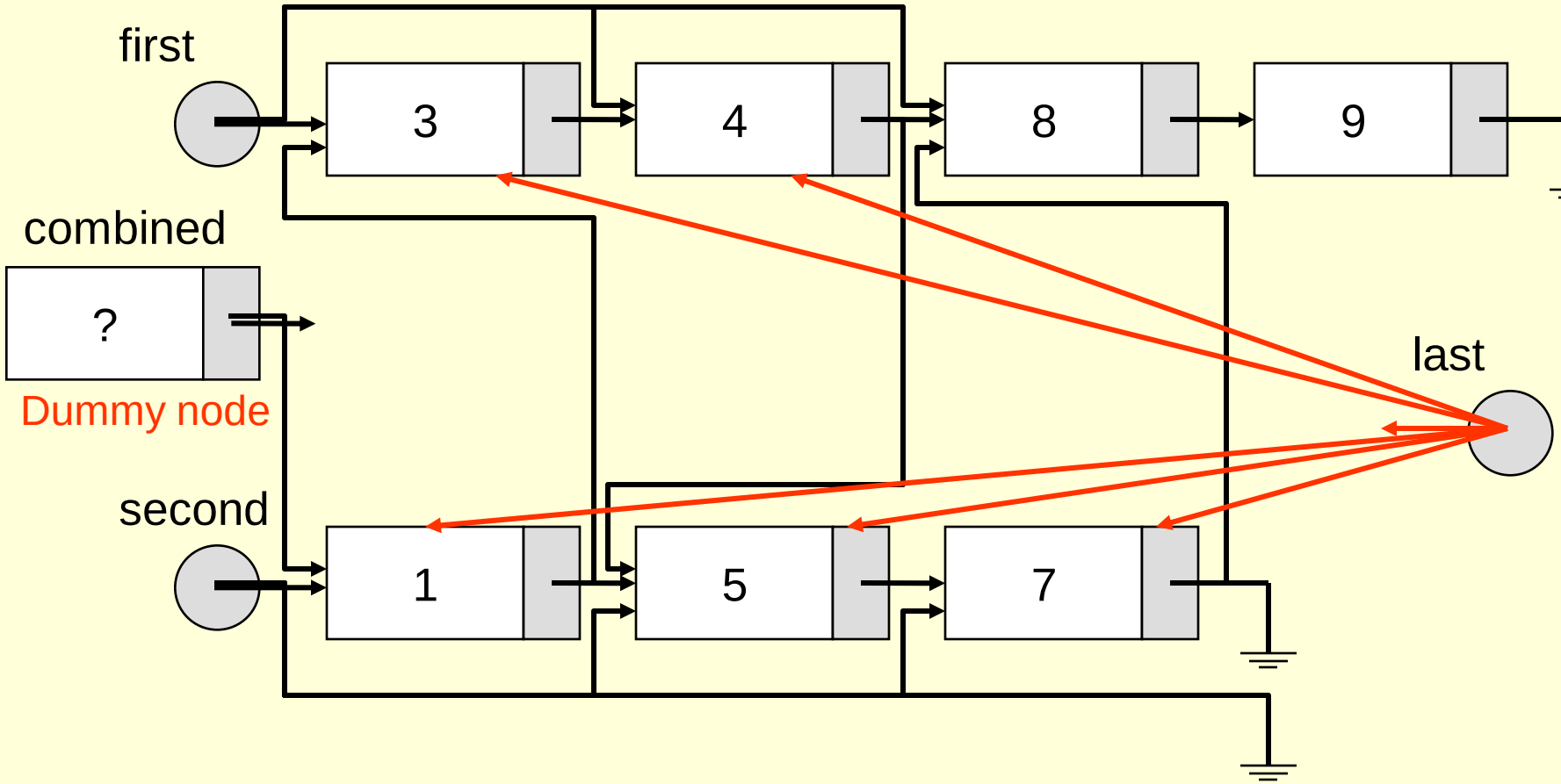
Mã C++ chia đôi DSLK



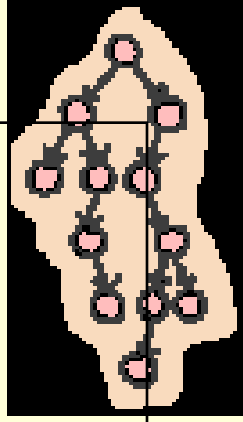
```
template <class Record>
Node<Record> *Sortable_list<Record> ::
                                    divide_from (Node<Record> *sub_list) {
    Node<Record> *position, *midpoint, *second_half;
    if ((midpoint = sub_list) == NULL) return NULL;
    position = midpoint->next;
    while (position != NULL) {
        position = position->next;          //Di chuyển một lần
        if (position != NULL) {             //Dừng ngay trước điểm giữa
            midpoint = midpoint->next;
            position = position->next;      //Di chuyển lần thứ hai
        }
    }
    second_half = midpoint->next;           //Phần sau là sau điểm dừng
    midpoint->next = NULL;                  //Tách đôi danh sách
    return second_half;
}
```

Trộn 2 DSLK có thứ tự





Giải thuật trộn hai DSLK có thứ tự

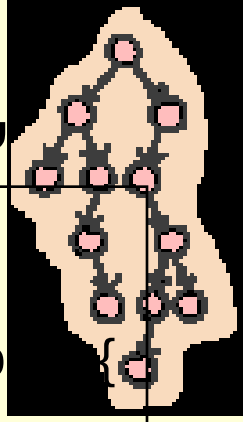


Algorithm Merge
Input: hai DSLK first và second có thứ tự
Output: một DSLK có thứ tự

1. last_sorted là một node giả
2. **while** (first và second khác NULL) *//Cả hai vẫn còn*
   2.1. **if** (dữ liệu của first nhỏ hơn dữ liệu của second)
      2.1.1. Nối first vào sau last_sorted *//Gỡ phần tử từ*
      2.1.2. last_sorted là first *//DSLK 1*
      2.1.3. Chuyển first đến phần tử kế *//gắn vào kết quả*
   2.2. **else**
      2.2.1. Nối second vào sau last_sorted *//Gỡ phần tử từ*
      2.2.2. last_sorted là second *//DSLK 2*
      2.2.3. Chuyển second đến phần tử kế *//gắn vào kết quả*
   2.3. **if** (danh sách first còn)
      2.3.1. Nối first vào sau last_sorted
   2.4. **else**
      2.4.1. Nối second vào sau last_sorted

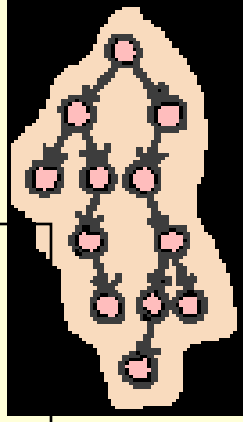
End Merge

Mã C++ trộn hai DSLK có thứ tự



```
template <class Record>
Node<Record> *Sortable_list<Record> ::
                    merge(Node<Record> *first, Node<Record> *seco
    Node<Record> combined, *last_sorted = &combined;
    while (first != NULL && second != NULL) {
        if (first->entry <= second->entry) {
            last_sorted->next = first;
            last_sorted = first;  first = first->next;
        } else {
            last_sorted->next = second;
            last_sorted = second;  second = second->next; }
    }
    if (first == NULL)
        last_sorted->next = second;
    else
        last_sorted->next = first;
    return combined.next;
}
```

Quick sort



Sort (26, 33, 35, 29, 19, 12, 22)

Phân thành (19, 12, 22) và (33,35,29) với pivot=26

Sort (19, 12, 22)

Phân thành (12) và (22) với pivot=19

Sort (12)

Sort (22)

Combine into (12, 19, 22)

Sort (33, 35, 29)

Phân thành (29) và (35) với pivot=33

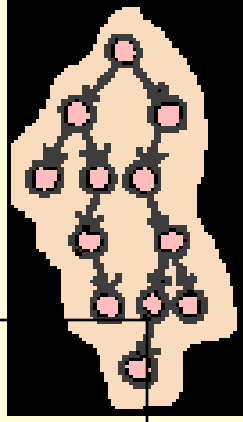
Sort (29)

Sort (35)

Combine into (29, 33, 35)

Combine into (12, 19, 22, 26, 29, 33, 35)

Giải thuật Quick sort

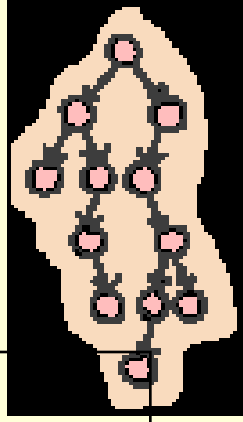


Algorithm quick_sort
Input: danh sách cần sắp xếp
Output: danh sách đã được sắp xếp

1. **if** (có ít nhất 2 phần tử)
   //Phân hoạch danh sách thành 3 phần:
   //- Phần nhỏ hơn phần tử giữa
   //- Phần tử giữa
   //- Phần lớn hơn phần tử giữa
   1.1. Phân hoạch danh sách ra 3 phần
   1.2. **Call** quick_sort cho phần bên trái
   1.3. **Call** quick_sort cho phần bên phải
   //Chỉ cần ghép lại là thành danh sách có thứ tự

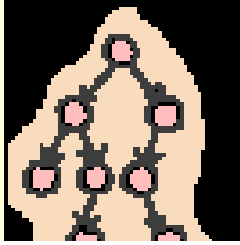
End quick_sort

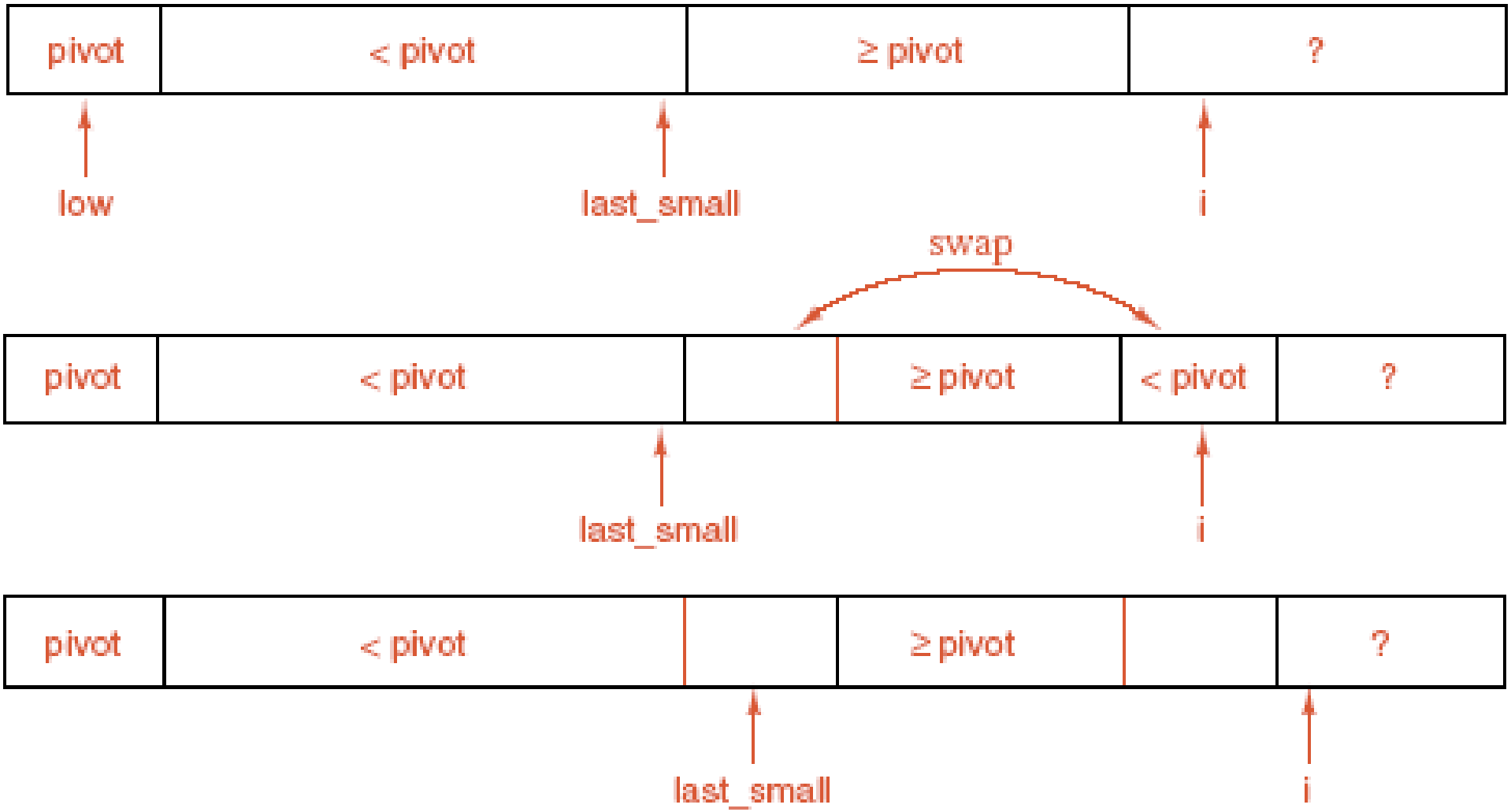
Mã C++ Quick sort trên danh sách liên tục



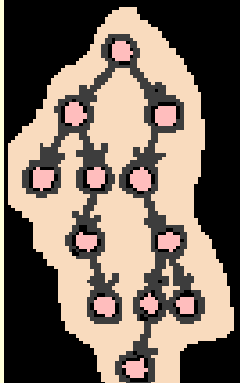
```
template <class Record>
void Sortable_list<Record> :: recursive_quick_sort(int low, int high) {
    //Phần được sắp xếp trong danh sách từ vị trí low đến vị trí high
    int pivot_position;
    if (low < high) {
       //pivot_postition là vị trí của phần tử giữa
        pivot_position = partition(low, high);

        recursive_quick_sort(low, pivot_position − 1);
        recursive_quick_sort(pivot_position + 1, high);

        //Danh sách kết quả đã có thứ tự trong khoảng từ low đến high
    }
}
```

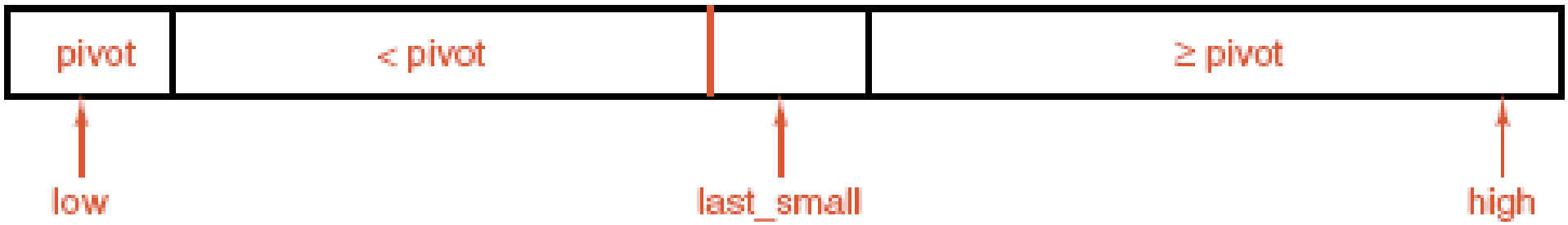
Phân hoạch cho quick sort

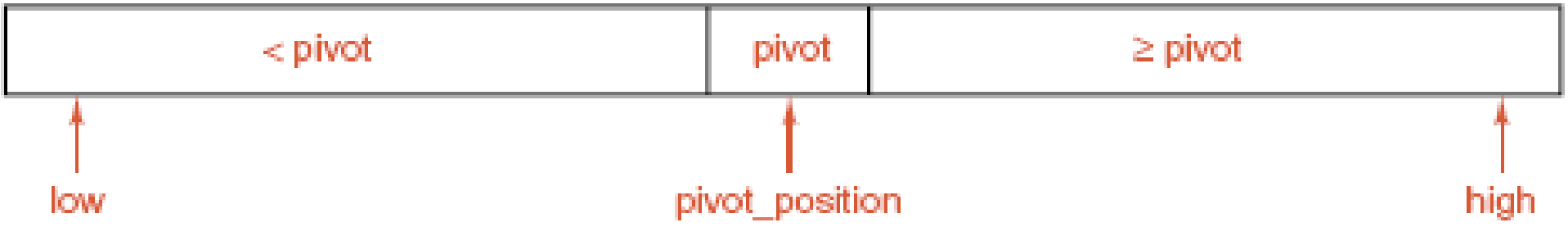




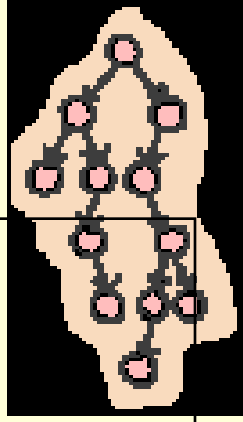
Phân hoạch cho quick sort (tt.)







Giải thuật phân hoạch



Algorithm partition

Input: danh sách cần phân hoạch từ low đến high

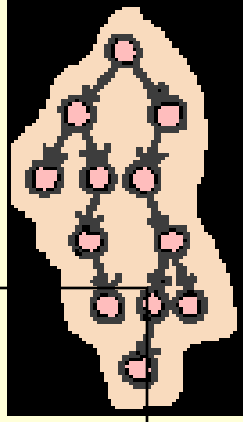
Output: đã phân hoạch làm 3 phần, vị trí pivot được ghi nhận

//Chọn phần tử tại vị trí giữa là phần tử pivot và chuyển về đầu

1. swap list[low], list[(low+high)/2]
2. pivot = list[low]
3. last_small = low
4. **for** index = low+1 **to** high *//Quét qua tất cả các phần tử còn lại*
   - 4.1. **if** list[index] < pivot
     - 4.1.1. last_small = last_small + 1
     - 4.1.2. swap list[last_small], list[index] *//Chuyển qua phần nhỏ hơn*
5. swap list[last_small], list[low] *//Trả phần tử pivot về lại chính giữa*
6. Vị trí pivot chính là last_small

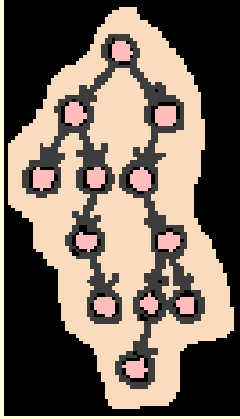
End partition

Mã C++ phân hoạch



```
template <class Record>
int Sortable_list<Record> :: partition(int low, int high) {
    //Giả sử hàm swap (ind1, ind2) sẽ đổi chỗ 2 phần tử tại 2 vị trí đó
    Record pivot;
    swap(low, (low + high)/2);
    pivot = entry[low];
    int last_small = low;
    for (int index = low + 1; index <= high; index++)
        if (entry[index] < pivot) {
            last_small = last_small + 1;
            swap(last_small, index);
        }
    swap(low, last_small);
    return last_small;
}
```

Ví dụ quick sort



recursive_quick_sort(0,6)

pivot

pivot_position = partition(0,6) = 3

26

0	1	2	3	4	5	6
19	35	33	26	29	12	22

low=0

high=6

last_small

pivot_position

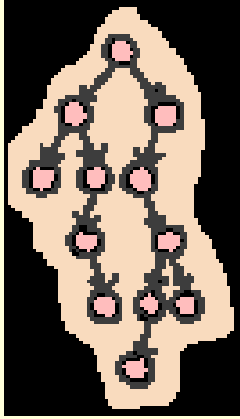
mid=(low+high)/2=3

index

recursive_quick_sort(0,2)

recursive_quick_sort(4,6)

Ví dụ quick sort (tt.)



recursive_quick_sort(0,2)

pivot_position = partition(0,2) = 1

pivot

19

0	1	2	3	4	5	6
22	19	12	26	29	33	35

low=0

high=2

last_small

mid=(low+high)/2=1

index

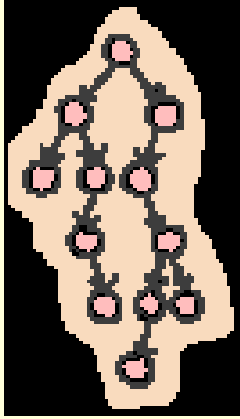
recursive_quick_sort(0,0)

(Không làm gì cả)

recursive_quick_sort(2,2)

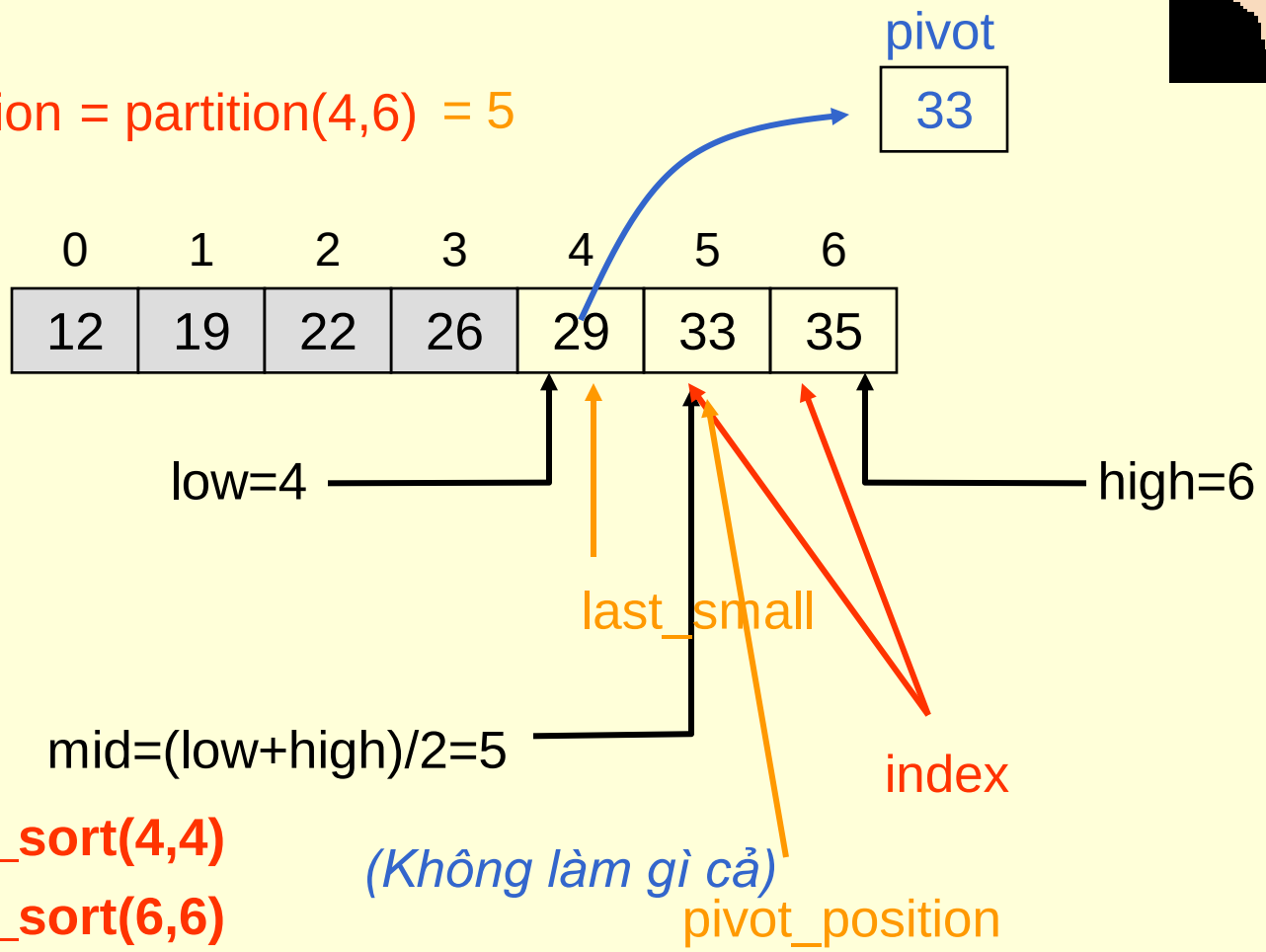
pivot_position

Ví dụ quick sort (tt.)



recursive_quick_sort(4,6)

pivot_position = partition(4,6) = 5

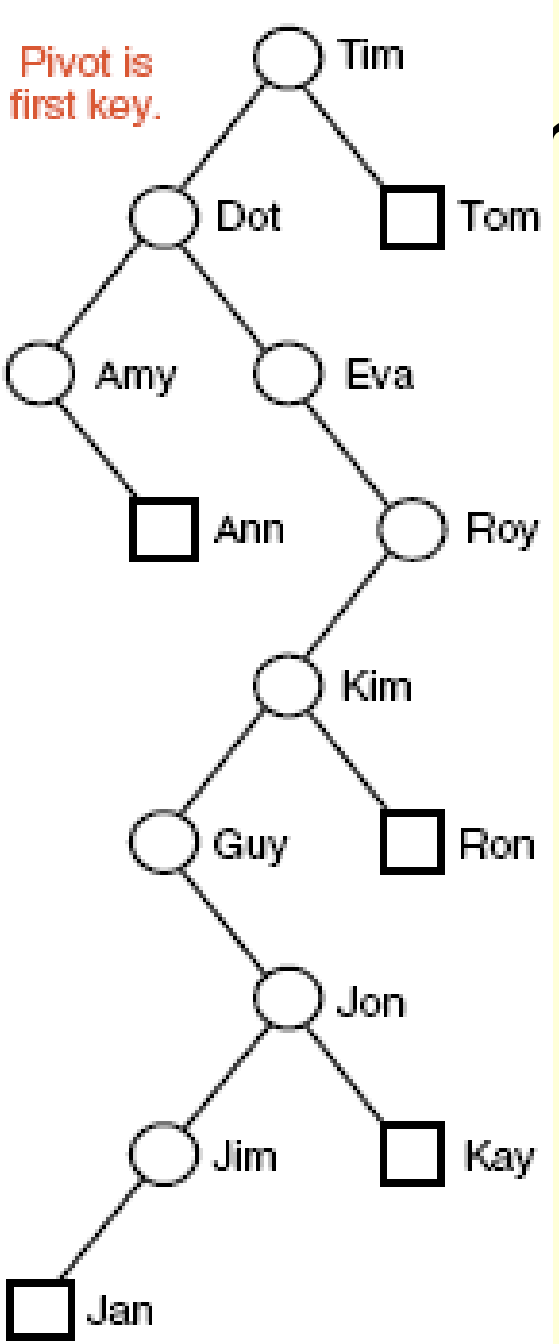


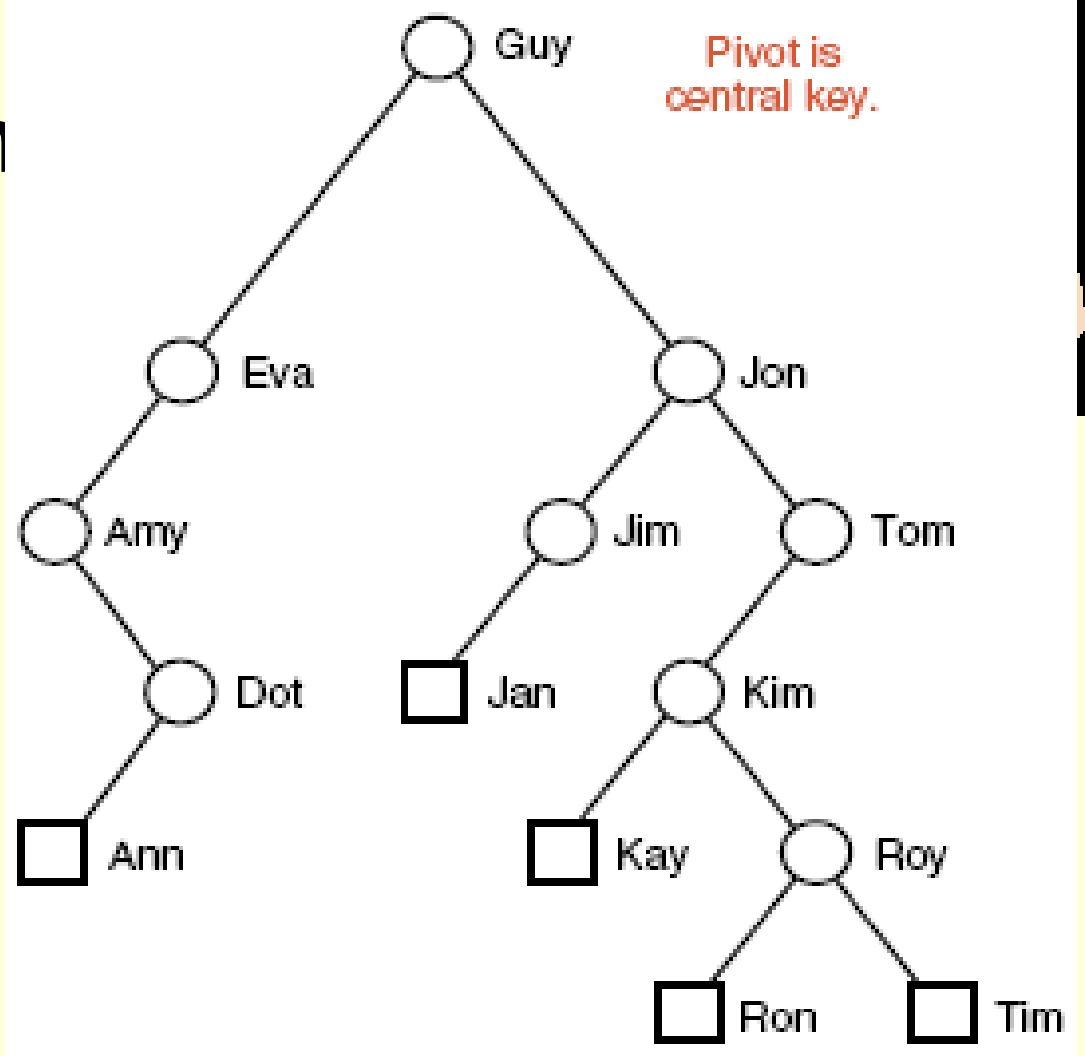
recursive_quick_sort(4,4)

recursive_quick_sort(6,6)

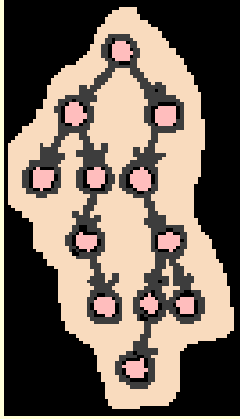
Unsorted

Tim
Dot
Eva
Roy
Tom
Kim
Guy
Amy
Jon
Ann
Jim
Kay
Ron
Jan



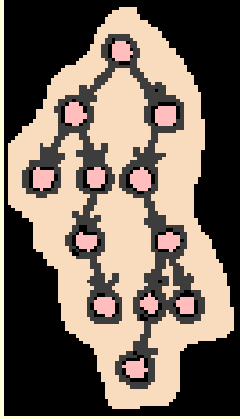


Đánh giá Quick sort



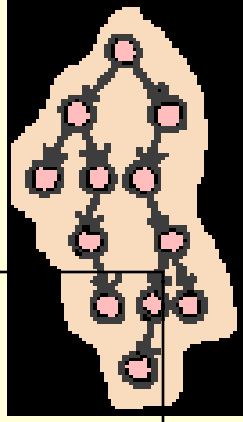
- Trường hợp xấu nhất:
  - Một bên rỗng và một bên là n-1 phần tử => n(n-1)/2
- Chọn phần tử pivot:
  - Đầu (hay cuối): trường hợp xấu xảy ra khi danh sách đang có thứ tự (hoặc thứ tự ngược)
  - Ở trung tâm, hoặc ngẫu nhiên: trường hợp xấu khó xảy ra
- Trường hợp trung bình:
  - $C(n) = 2n \ln n + O(n) \approx 1.39\, n \lg n + O(n)$

Heap và Heap sort



- Heap (định nghĩa lại):
  - Danh sách có n phần tử (từ 0 đến n-1)
  - $a_k \geq a_{2k+1}$ và $a_k \geq a_{2k+2}$ (a_k lớn nhất trong 3 phần tử)
- Đặc tính:
  - a_0 là phần tử lớn nhất
  - Danh sách chưa chắc có thứ tự
  - Nửa sau của danh sách bất kỳ thỏa định nghĩa heap
- Heap sort:
  - Lấy a_0 ra, tái tạo lại heap => Phần tử lớn nhất
  - Lấy a_0 mới ra, tái tạo lại heap => Phần tử lớn kề
  - …

Giải thuật Heap sort



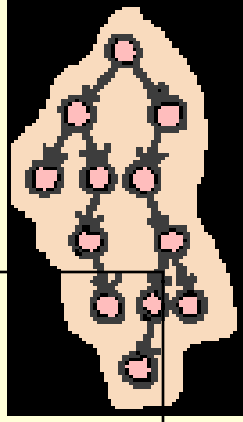
Algorithm heap_sort
Input: danh sách cần sắp xếp có n phần tử
Output: danh sách đã sắp thứ tự

//Xây dựng heap ban đầu
1. **Call** build_heap
//Lần lượt lấy phần tử đầu ra đem về cuối danh sách hiện tại
//rồi xây dựng heap trở lại
2. **for** index = size-1 **to** 0 *//index là vị trí cuối của phần còn lại*
   2.1. swap list[0], list[index] *//Đổi phần tử lớn nhất về cuối*

   //Xây dựng lại heap với số phần tử giảm đi 1
   2.2. **Call** rebuild_heap index-1

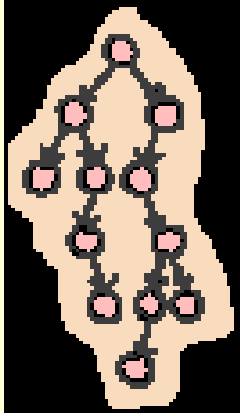
End heap_sort

Mã C++ Heap sort

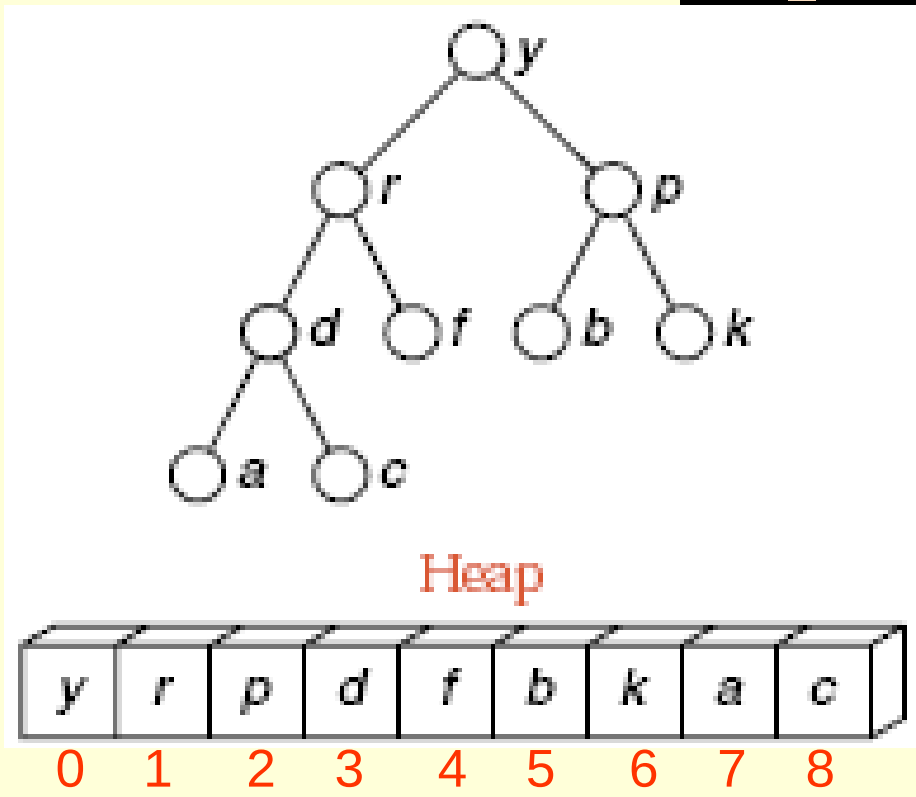


```
template <class Record>
void Sortable_list<Record> :: heap_sort( ) {
    Record current;
    //Xây dựng heap ban đầu
    build_heap( );
    for (int last_unsorted = count − 1; last_unsorted > 0; last_unsorted−−)
    {
        //Giữ lại phần tử cuối cũ
        current = entry[last_unsorted]; // Extract last entry from list.
        //Chép phần tử đầu (lớn nhất) về vị trí này
        entry[last_unsorted] = entry[0];

        //Xây dựng lại heap bằng cách chèn phần tử current vào đúng vị trí
        insert_heap(current, 0, last_unsorted − 1);
    }
}
```

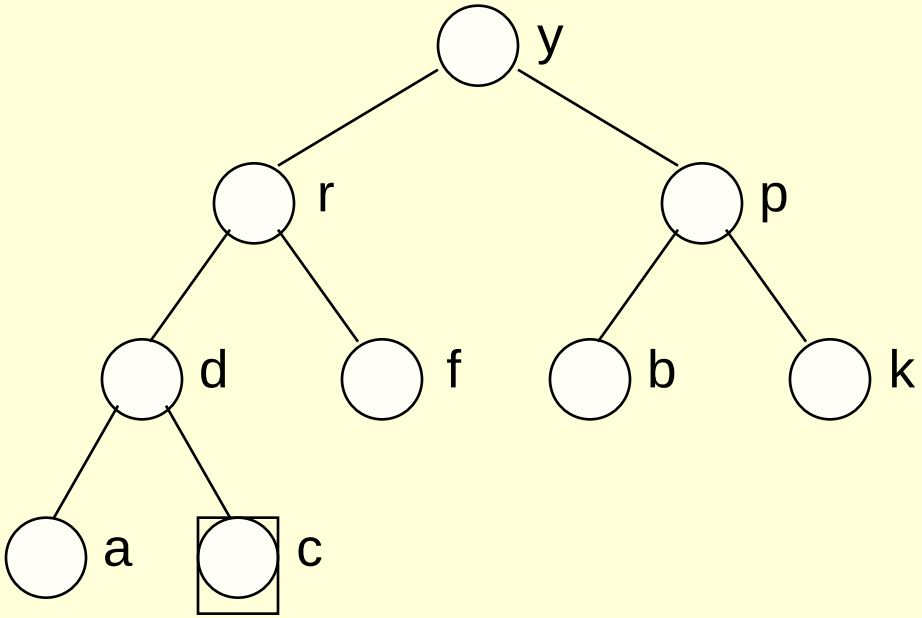
Biểu diễn Heap



- Dạng cây nhị phân:
  - Node gốc là a_0
  - 2 node con của phần tử a_k là 2 phần tử a_{2k+1} và a_{2k+2}
  - Ở mức cuối cùng, các node lấp đầy từ bên trái sang bên phải (cây nhị phân gần đầy đủ)



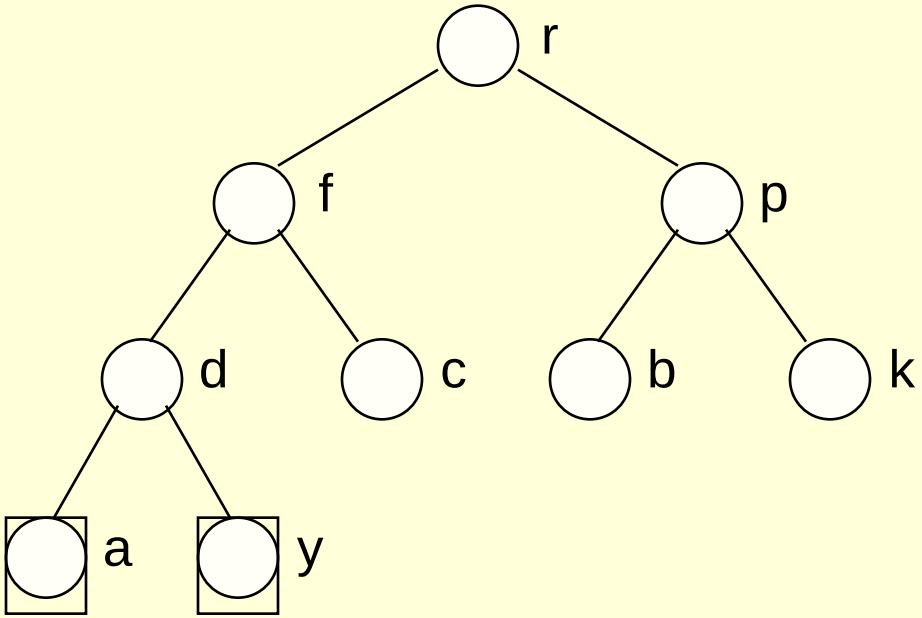
Ví dụ Heap sort



current

0	1	2	3	4	5	6	7	8
y	r	p	d	f	b	k	a	c

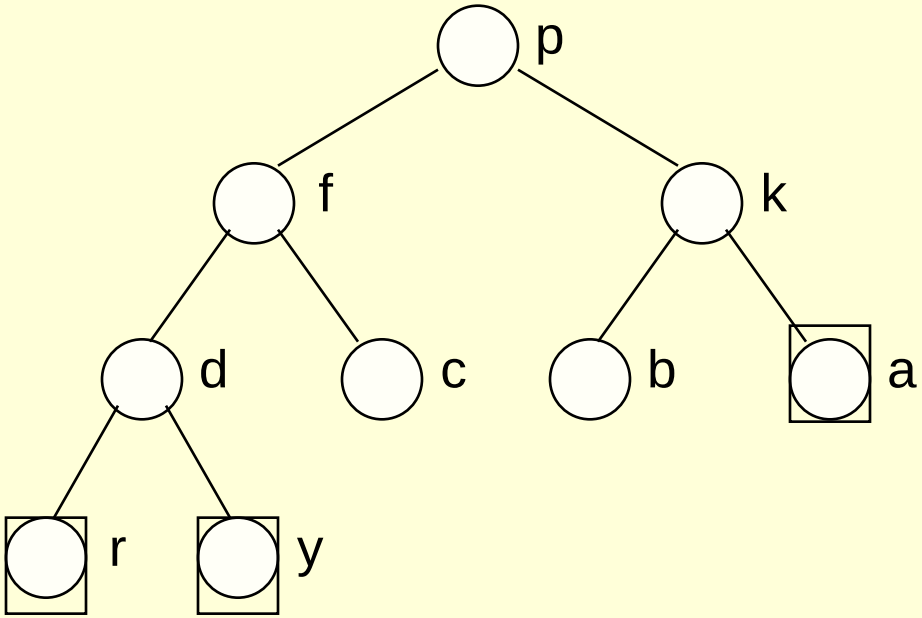
Ví dụ Heap sort (tt.)



current

0	1	2	3	4	5	6	7	8
r	f	p	d	c	b	k	a	y

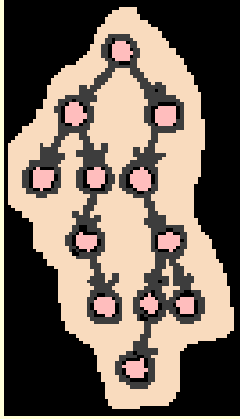
Ví dụ Heap sort (tt.)

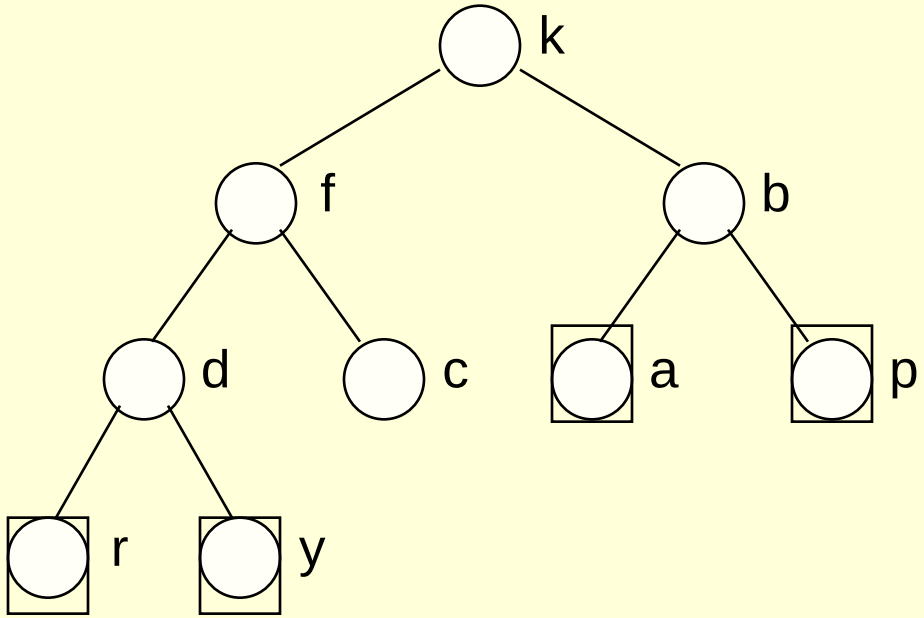


current

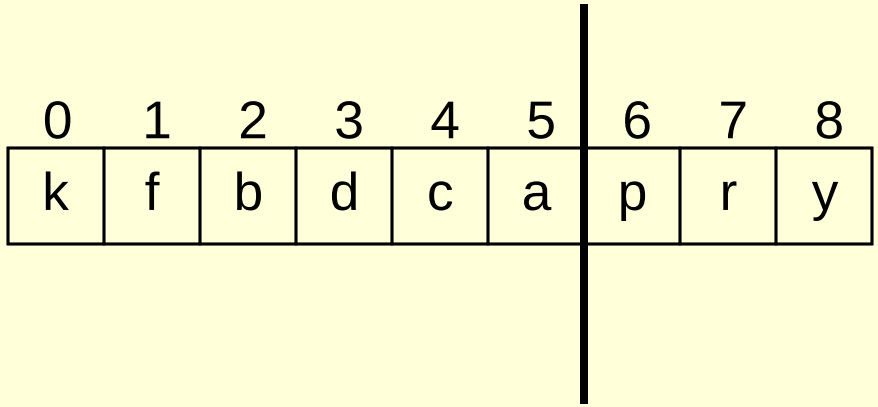
0	1	2	3	4	5	6	7	8
p	f	k	d	c	b	a	r	y

Ví dụ Heap sort (tt.)

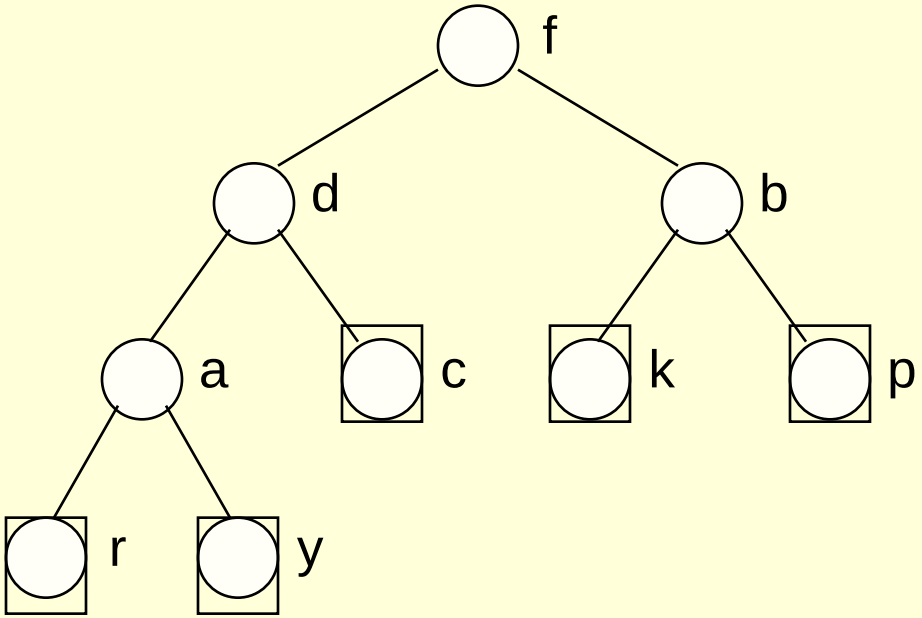




current



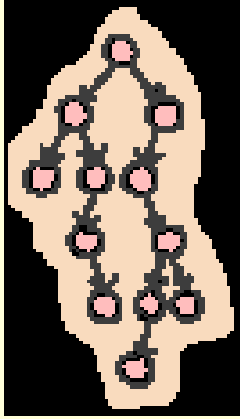
Ví dụ Heap sort (tt.)

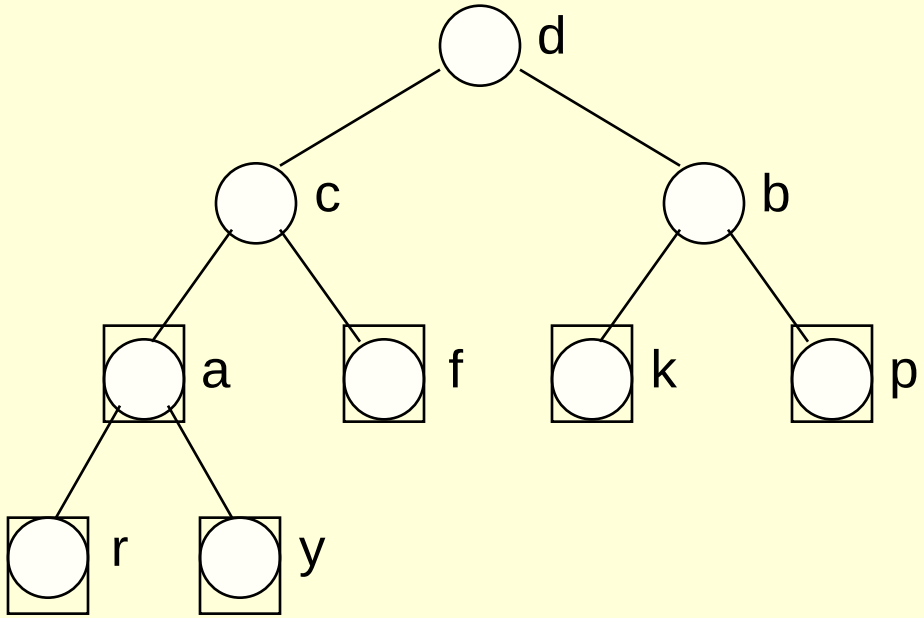


current

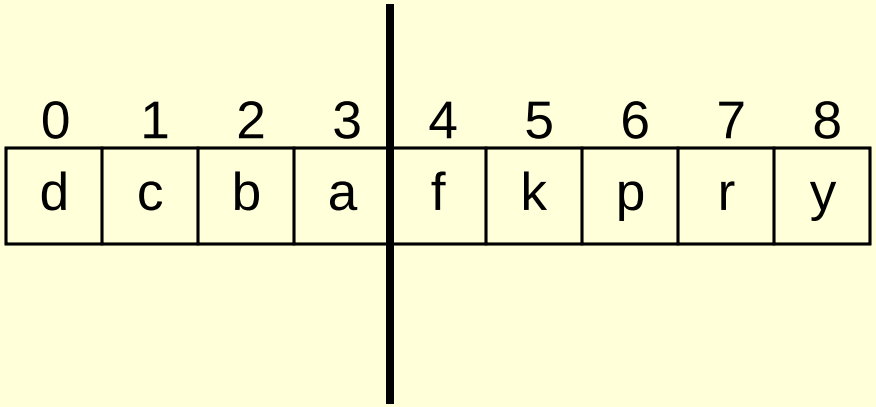
0	1	2	3	4	5	6	7	8
f	d	b	a	c	k	p	r	y

Ví dụ Heap sort (tt.)

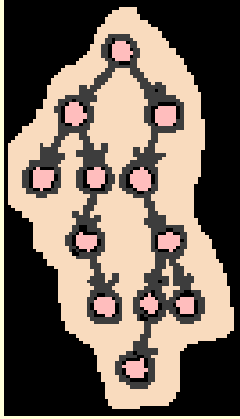


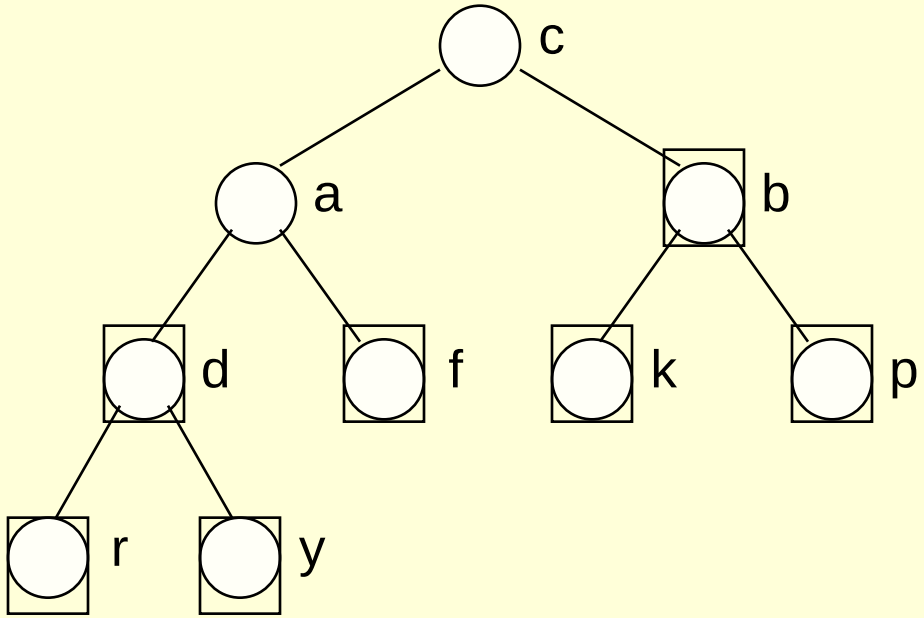


current

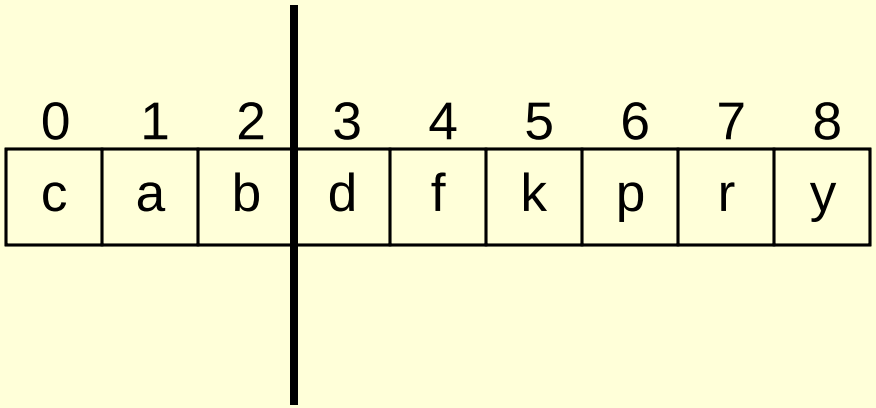


Ví dụ Heap sort (tt.)

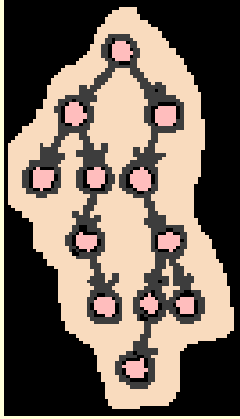


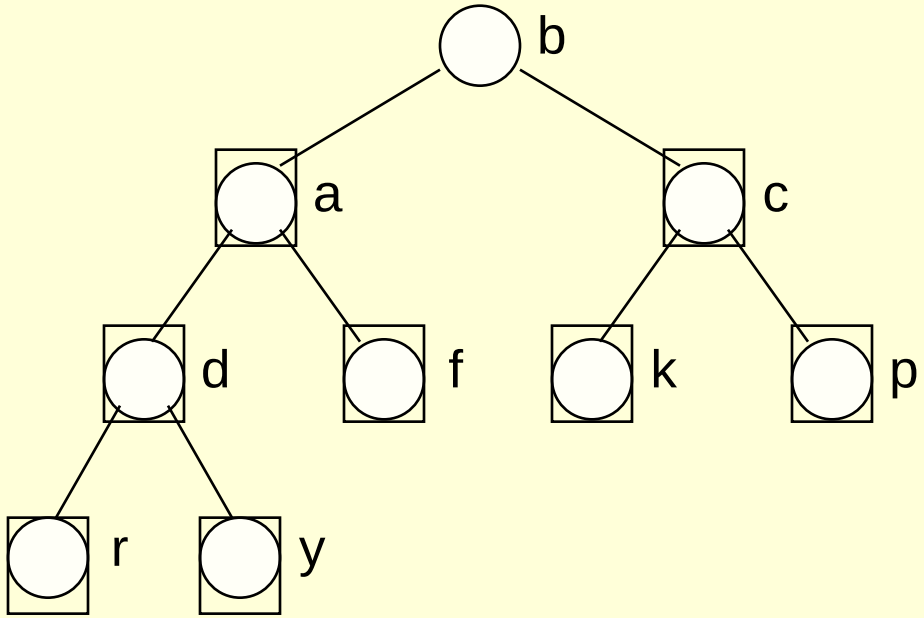


current

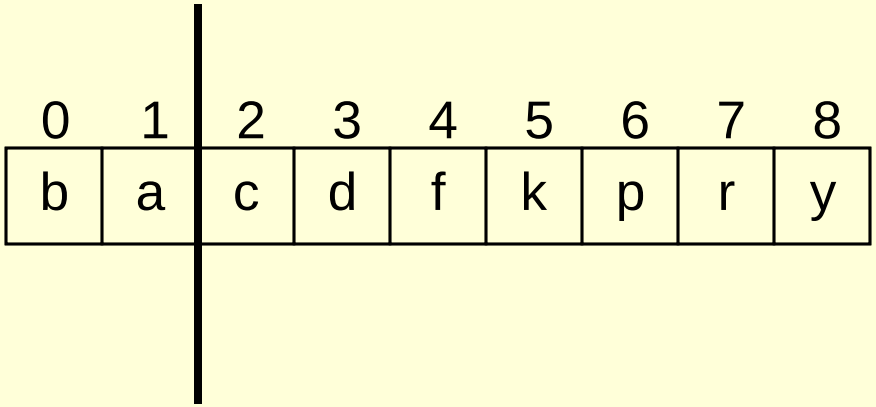


Ví dụ Heap sort (tt.)

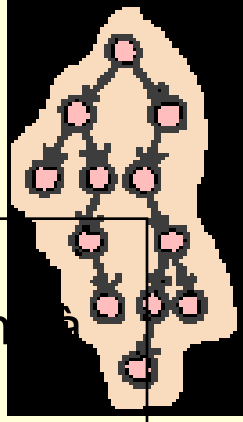




current



Giải thuật tái tạo lại heap



Algorithm insert_heap

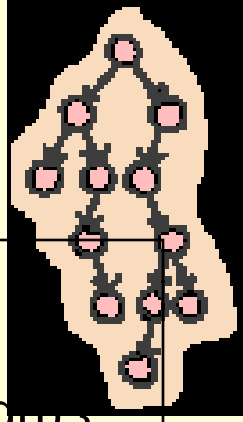
Input: danh sách là heap trong khoảng từ low+1 đến high, current là giá trị cần thêm vào

Output: danh sách là heap trong khoảng từ low đến high

1. Bắt đầu kiểm tra tại vị trí low
2. **while** (chưa kiểm tra xong đến high)
   - 2.1. Tìm lớn nhất trong bộ ba phần tử current, list[2k+1], list[2k+2]
   - 2.2. **if** (phần tử đó là current)
     - 2.2.1. Ngừng vòng lặp
   - 2.3. **else**
     - 2.3.1. Dời phần tử lớn nhất lên vị trí hiện tại
     - 2.3.2. Kiểm tra bắt đầu từ vị trí của phần tử lớn nhất này
3. Đưa current vào vị trí đang kiểm tra

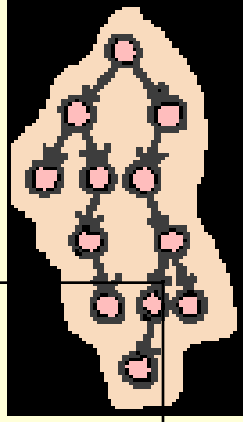
End insert_heap

Mã C++ tái tạo lại heap



```
template <class Record>
void Sortable_list<Record> ::
                    nsert_heap(const Record &current, int low, int high) {
    int large = 2 * low + 1;                    //P.tử lớn giả sử là tại 2k+1
    while (large <= high) {
        if (large < high && entry[large] < entry[large + 1])
            large++;                            //P.tử lớn tại 2k+2
        if (current >= entry[large])
            break;                              //Nếu current là lớn nhất thì thôi
        else {
            entry[low] = entry[large];          //Không thì đẩy p.tử lớn nhất lên
            low = large;                        //rồi tiếp tục kiểm tra về sau
            large = 2 * low + 1;
        }
    }
    entry[low] = current;                       //Đây là vị trí thích hợp cho current
}
```

Giải thuật xây dựng heap ban đầu



Algorithm build_heap
Input: danh sách bất kỳ cần biến thành heap
Output: danh sách đã là heap

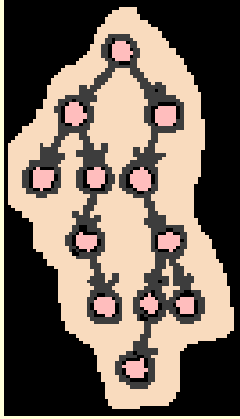
//Nửa sau của 1 danh sách bất kỳ thỏa tính chất của heap
//Ta tìm cách xây dựng heap ngược từ giữa về đầu
1. **for** low = size/2 **downto** 0

//Từ vị trí low+1 đến cuối danh sách đang là heap
1.1. current = list[low];

//Xây dựng lại heap trong khoảng [low, size-1]
1.2. **Call** insert_heap với current, low và size − 1

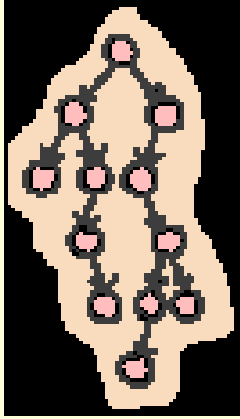
End build_heap

Mã C++ xây dựng heap ban đầu



```
template <class Record>
void Sortable_list<Record> :: build_heap( ) {
    //Nửa sau của 1 danh sách bất kỳ thỏa tính chất của heap
    //Ta tìm cách xây dựng heap ngược từ giữa về đầu
    for (int low = count/2 − 1; low >= 0; low−−) {
        Record current = entry[low];
        insert_heap(current, low, count − 1);
    }
}
```

Ví dụ xây dựng heap ban đầu



Bước 1

0	1	2	3	4	5	6	7	8
p	c	y	d	f	b	k	a	r

Bước 2

0	1	2	3	4	5	6	7	8
p	c	y	r	f	b	k	a	d

Bước 3

0	1	2	3	4	5	6	7	8
p	c	y	r	f	b	k	a	d

Bước 3'

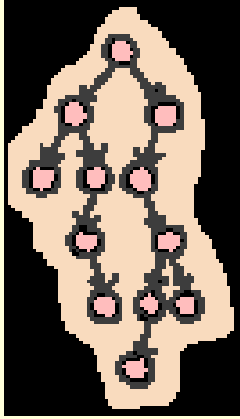
0	1	2	3	4	5	6	7	8
p	r	y	c	f	b	k	a	d

Bước 4

0	1	2	3	4	5	6	7	8
p	r	y	d	f	b	k	a	c

0	1	2	3	4	5	6	7	8
y	r	p	d	f	b	k	a	c

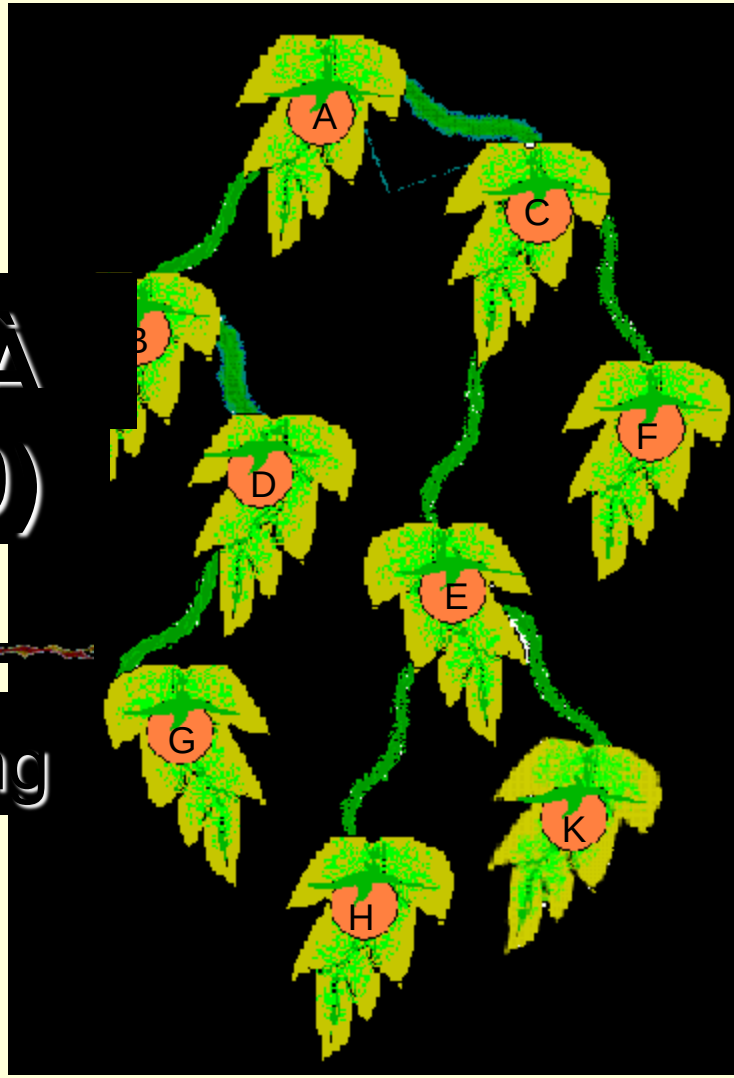
Đánh giá Heap sort



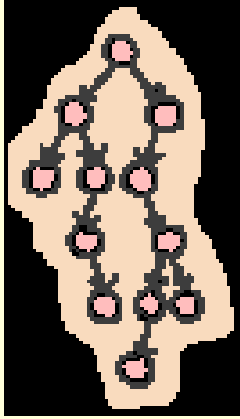
- Trường hợp xấu nhất:
  - C = $2n \lg n + O(n)$
  - M = $n \lg n + O(n)$
- So với Quick sort
  - Trung bình: chậm hơn quick sort
  - Xấu nhất: $O(n \lg n) < n(n-1)/2$

CẤU TRÚC DỮ LIỆU VÀ GIẢI THUẬT (501040)

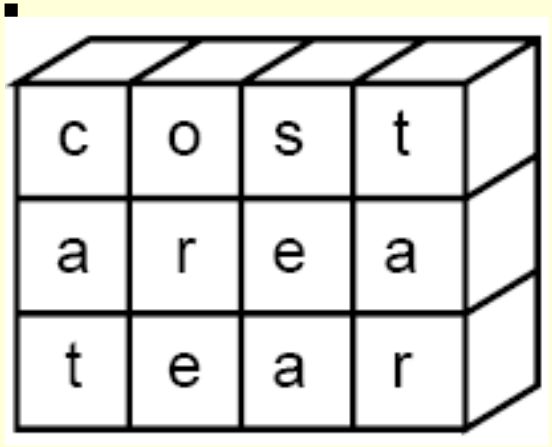
Chương 9: Bảng

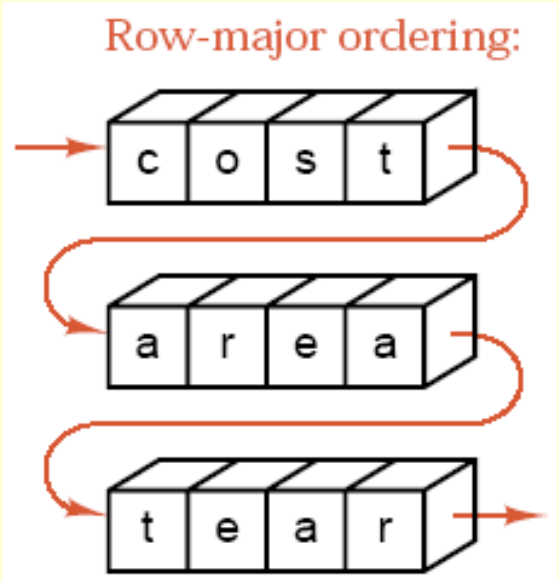


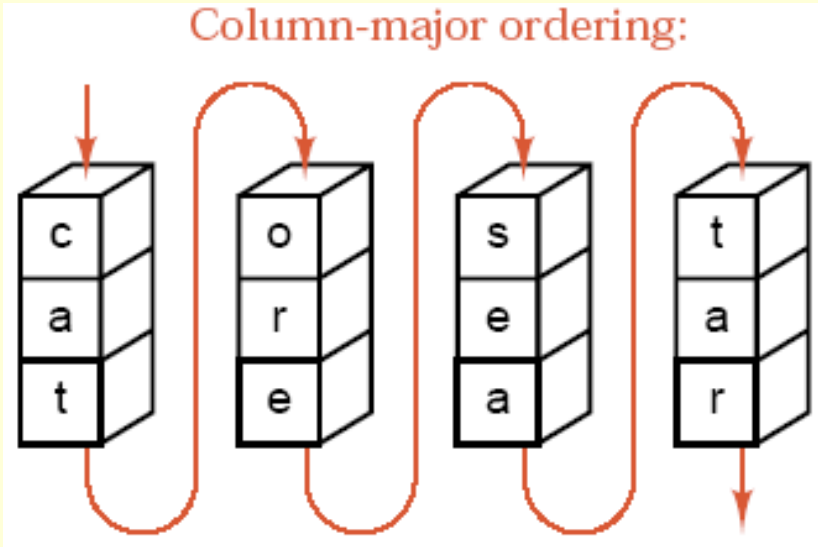
Ma trận 2 chiều vs. 1 chiều



A[i, j]

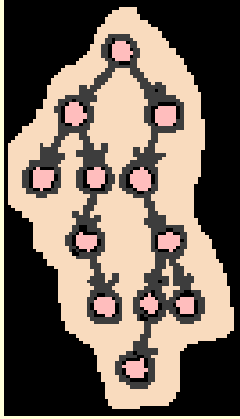






B[max_row*i + j]

C[i + max_col*j]

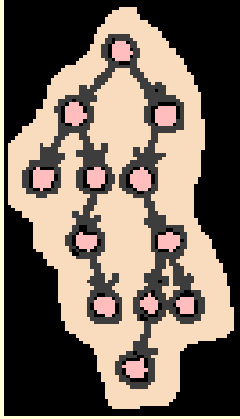


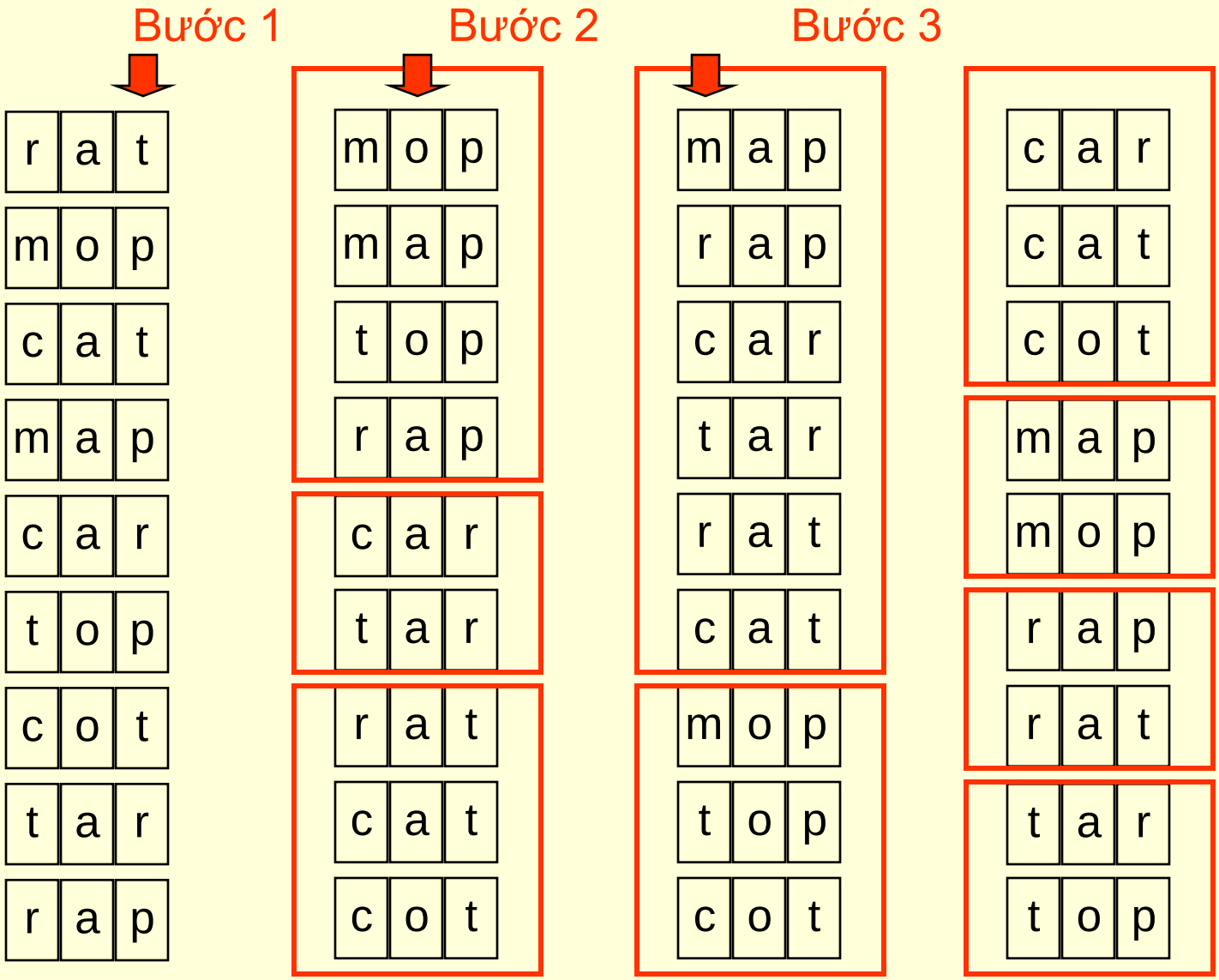
Index	*Name*	*Address*	*Phone*
1	Hill, Thomas M.	High Towers #317	2829478
2	Baker, John S.	17 King Street	2884285
3	Roberts, L. B.	53 Ash Street	4372296
4	King, Barbara	High Towers #802	2863386
5	Hill, Thomas M.	39 King Street	2495723
6	Byers, Carolyn	118 Maple Street	4394231
7	Moody, C. L.	High Towers #210	2822214

Access Tables

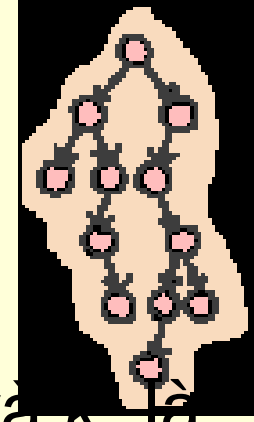
Name	*Address*	*Phone*
2	3	5
6	7	7
1	1	1
5	4	4
4	2	2
7	5	3
3	6	6

Radix sort

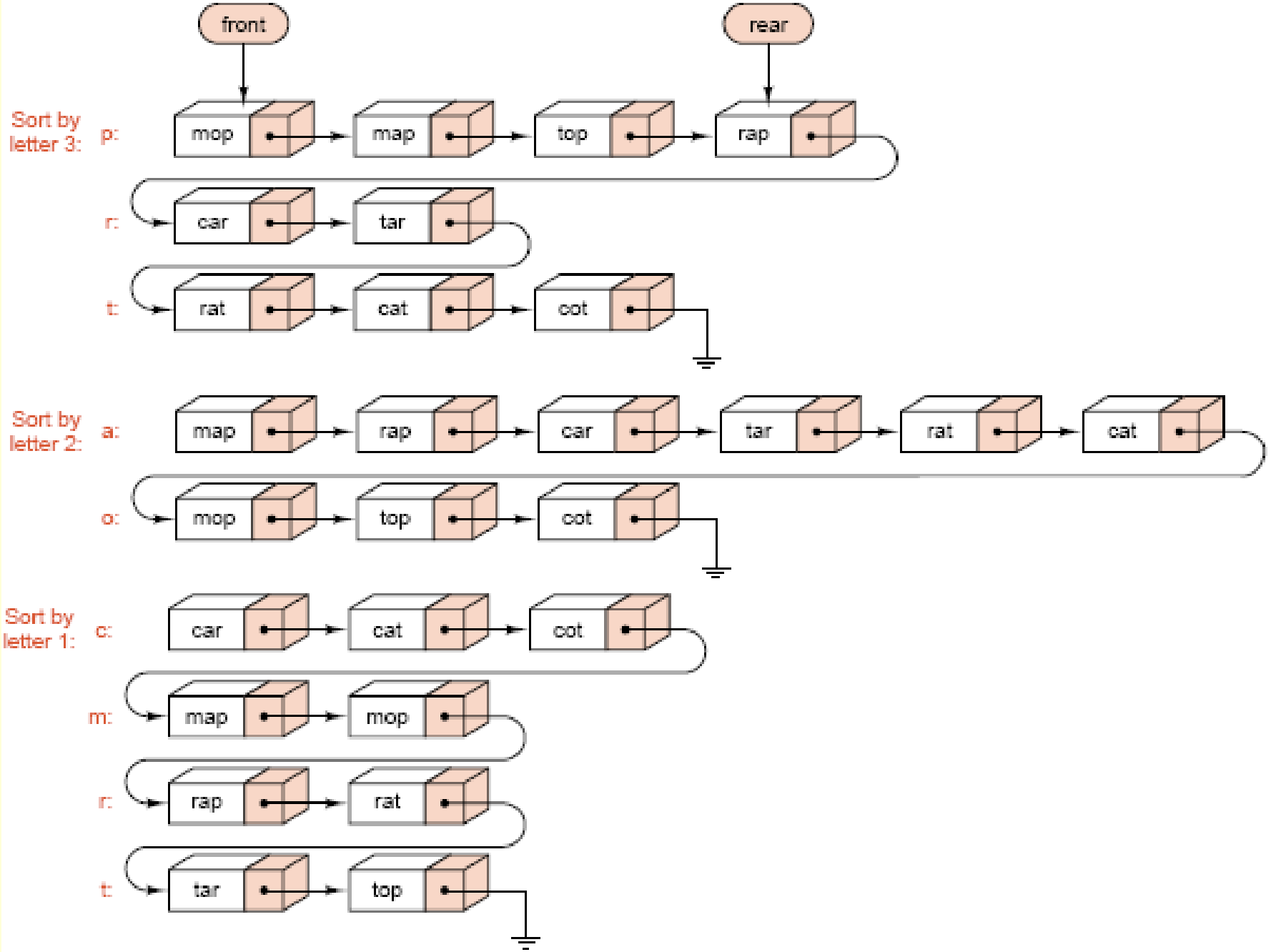



Bước 1
Bước 2
Bước 3
r a t
m o p
c a t
m a p
c a r
t o p
c o t
t a r
r a p
m o p
m a p
t o p
r a p
c a r
t a r
r a t
c a t
c o t
m a p
r a p
c a r
t a r
r a t
c a t
m o p
t o p
c o t
c a r
c a t
c o t
m a p
m o p
r a p
r a t
t a r
t o p

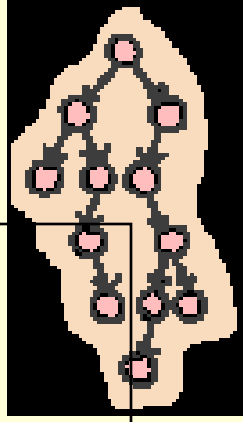
Đánh giá Radix sort



- Số lần so sánh là $\Theta(n\,k)$, n là số phần tử và k là số ký tự trên khóa
- So sánh với các phương pháp khác là $n \lg n$:
  - Nếu k lớn và n là nhỏ thì radix sort chậm
  - Nếu k nhỏ và n là lớn thì radix sort nhanh hơn
- Bất tiện:
  - Việc tách thành 27 danh sách con và ghép lại lúc sau trên DS liên tục gây ra việc di chuyển nhiều phần tử
  - Khóa so sánh là chuỗi nhị phân thì không tốt


front
rear
Sort by letter 3:
p:
mop
map
top
rap
r:
car
tar
t:
rat
cat
cot
Sort by letter 2:
a:
map
rap
car
tar
rat
cat
o:
mop
top
cot
Sort by letter 1:
c:
car
cat
cot
m:
map
mop
r:
rap
rat
t:
tar
top

Giải thuật Radix sort trên DSLK



Algorithm radix_sort
Input: danh sách cần sắp thứ tự
Output: danh sách đã sắp thứ tự

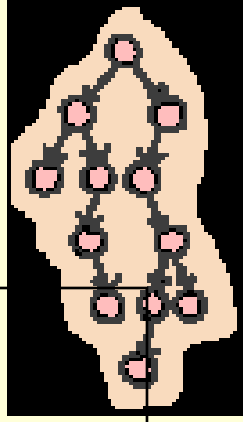
//Mỗi queue chứa các phần tử có ký tự tương ứng
1. queues là một dãy có max_character hàng

//Lặp k bước, kiểm tra các ký tự tại vị trí k
2. **for** position = size(khóa) **to** 0
    2.1. **while** (danh sách còn)
        2.1.1. Lấy phần tử đầu tiên
        2.1.2. Tính toán thứ tự của chữ cái ở vị trí k trong khóa
        2.1.3. Đẩy phần tử này vào queue tương ứng

    2.2. Nối tất cả các queue lại với nhau thành danh sách

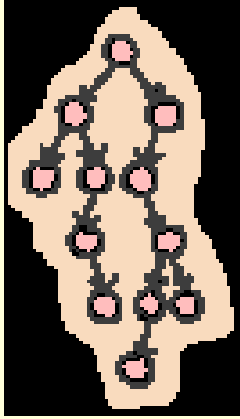
End radix_sort

Mã C++ Radix sort trên DSLK



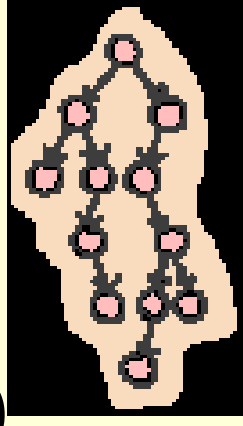
```
const int max_chars = 28;
template <class Record>
void Sortable_list<Record> :: radix_sort( ) {
    Record data;
    Queue queues[max_chars];
    for (int position = key_size − 1; position >= 0; position−−) {
        // Loop from the least to the most significant position.
        while (remove(0, data) == success) {
            int queue_number = alphabetic_order(data.key_letter(position));
            queues[queue_number].append(data); // Queue operation.
        }
        rethread(queues); // Reassemble the list.
    }
}
```

Nối các queue liên kết

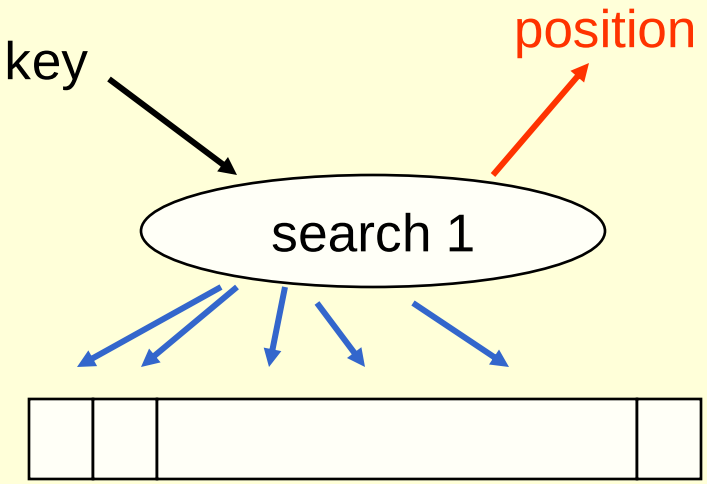


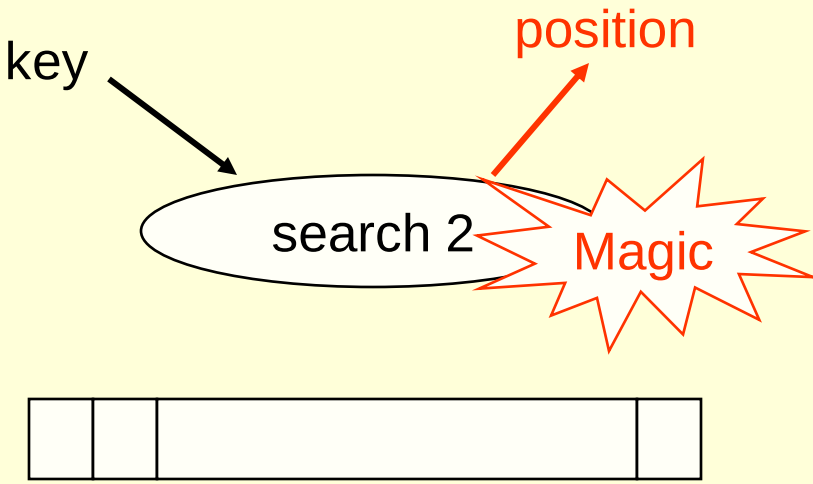
- Cách 1:
  - Dùng các CTDL queue
  - Phải dùng queue.retrieve và list.insert(list.size(),x)
- Cách 2:
  - Viết lại các CTDL kiểu queue trong chương trình
  - Chỉ cần tìm đến cuối mỗi queue và nối con trỏ vào đầu queue sau (hoặc đến NULL)

Tăng tốc tra cứu

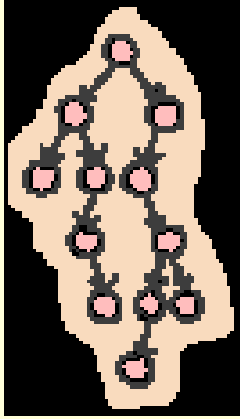


- Tìm kiếm: hàm f: key -> position =>O (lg n)
- Nếu có hàm f: key -> position với tốc độ $O(1)$
  - Ví dụ: Tra bảng với key chính là position
  - Hàm đổi một key thành position: hàm Hash



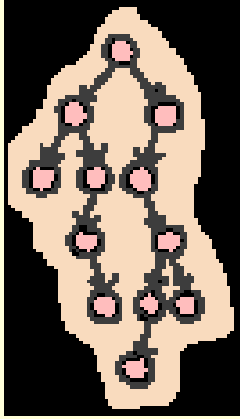


Bảng Hash



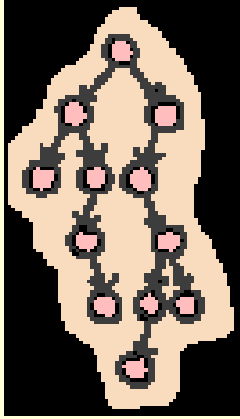
- Bảng Hash
  - Bảng
  - Vị trí của 1 phần tử được tính bằng hàm hash
- Hàm hash:
  - Nhận vào một khóa
  - Trả về một chỉ số vị trí
  - (Có thể chuyển vài khóa về cùng một vị trí)
- Đụng độ trên bảng hash:
  - Nếu vị trí tìm ra đúng là dữ liệu cần tìm: $O(1)$
  - Không đúng: giải quyết đụng độ (phải đảm bảo $O(1)$)

Hàm Hash



- Đảm bảo *O*(*1*)
- Ví dụ:
  - f(x) = x % m;
  - f(‘abc’) = (char_index(‘a’)*base_number^2 + char_index(‘b’)*base_number^1 + char_index(‘c’)*base_number^0) % hash_size

Ví dụ dùng bảng Hash



Các khóa: M, O, T, V, I, D, U

hash(x) = char_index(x) % 10

Tìm V

Tìm F

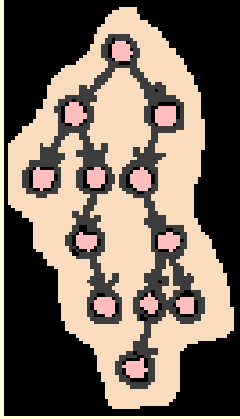
T	0
U	1
V	2
M	3
D	4
O	5
	6
	7
	8
I	9

Không có

char_index: Space=0, A=1, B=2, …. Z=27

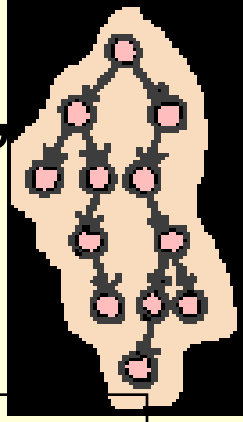
M	O	T	V	I	D	U	F
3	5	0	2	9	4	1	6

Phương pháp Địa chỉ mở (Open Addressing)



- Bảng hash là một array
- Các vị trí khi có đụng độ sẽ tìm vị trí mới bằng các phương pháp giải quyết:
  - Thử tuyến tính (linear probing):
    - Tăng chỉ số lên một: h = (h+i) % hash_size
  - Thử bậc hai (quadratic probing):
    - Tăng chỉ số lên theo bình phương: h = (h + i^2)% hash_size
  - Phương pháp khác
    - Ngẫu nhiên

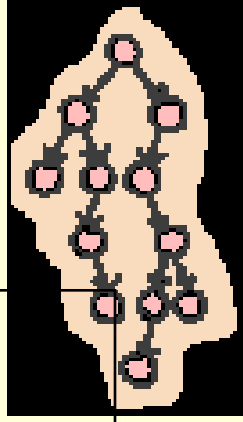
Thiết kế bảng Hash dùng địa chỉ mở



Đảm bảo phép thử tuyến tính không bị lặp vòng

```
const int hash_size = 997; // a prime number of appropriate size

class Hash_table {
public:
    Hash_table( );
    void clear( );
    Error_code insert(const Record &new entry);
    Error_code retrieve(const Key &target, Record &found) const;
private:
    Record table[hash_size];
};
```

Giải thuật thêm phần tử dùng bảng Hash địa chỉ mở



Algorithm Hash_Insert

Input: bảng Hash, mẫu tin cần thêm vào

Output: bảng Hash đã có mẫu tin thêm vào

1. probe = hash(input_key)
2. increment = 1 *//Dùng khi đụng độ*
3. **while** (table[probe] không rỗng) *//Có đụng độ*

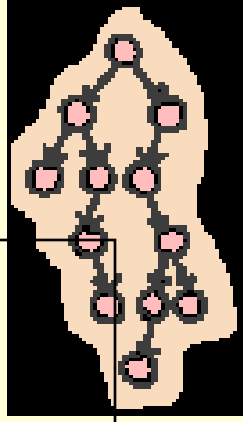
   //Dùng các phép thử (tuyến tính, bậc hai, …)

   3.1. probe = (probe + increment) % hash_size

   3.2. increment = increment + 2 *//Thử bậc hai*
4. table[probe] = new_data

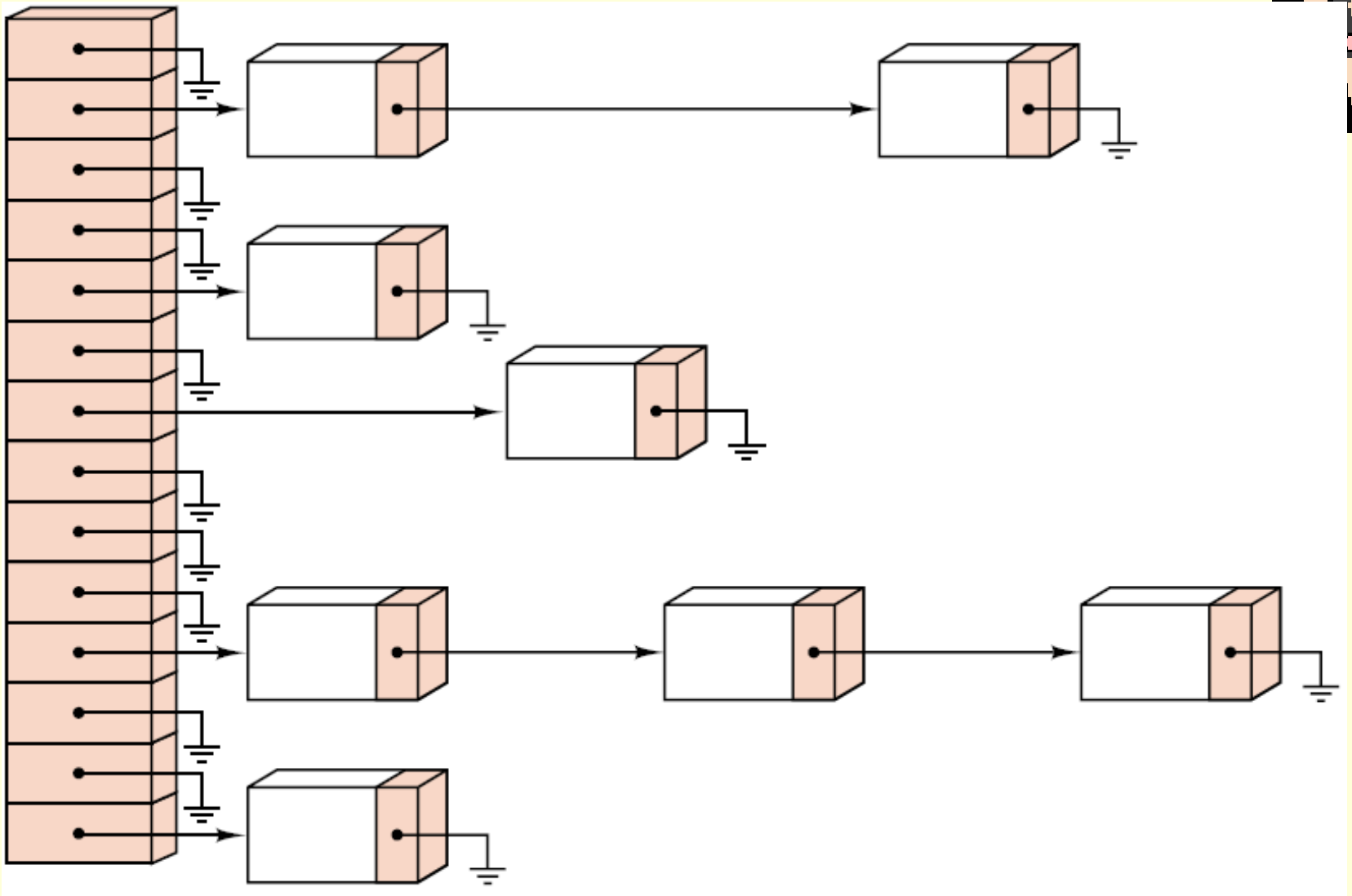
End Hash_insert

Mã C++ thêm phần tử dùng bảng Hash địa chỉ mở

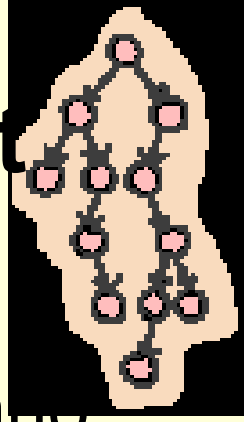


```
Error_code Hash_table :: insert(const Record &new entry) {
    Error_code result = success;
    int probe_count = 0, increment = 1, probe;
    Key null;
    probe = hash(new_entry);
    while (table[probe] != null && table[probe] != new_entry
                && probe_count < (hash size + 1)/2) {
        probe_count++;
        probe = (probe + increment)%hash_size;
        increment += 2;
    }
    if (table[probe] == null) table[probe] = new_entry;
    else if (table[probe] == new_entry) result = duplicate_error;
    else result = overflow;

    return result;
}
```

Phương pháp nối kết (chained hash table)

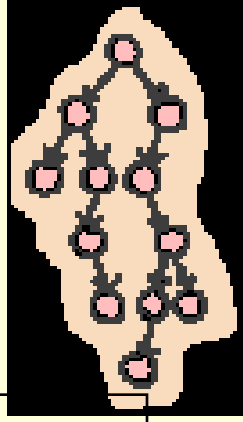


Lợi ích của phương pháp nối kết



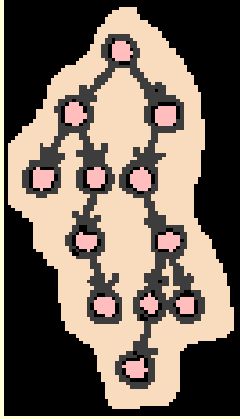
- Nếu số lượng mẫu tin lớn: tiết kiệm vùng nhớ.
- Giải quyết đụng độ: đơn giản là đẩy vào cùng một danh sách liên kết.
- Bảng hash nhỏ hơn nhiều so với số lượng mẫu tin.
- Xóa một phần tử là đơn giản và nhanh chóng.
- Độ phức tạp khi tìm kiếm:
  - Nếu có n mẫu tin, và bảng hash có kích thước m
  - Độ dài trung bình của DSLK là n/m

Thiết kế bảng Hash nối kết



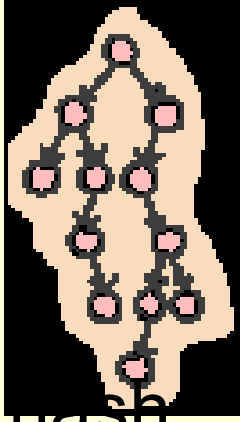
```
const int hash_size = 997; // a prime number of appropriate size

class Hash_table {
public:
   //Specify methods here
private:
   List<Record> table[hash_size];
};
```

Thiết kế các phương thức của bảng Hash nối kết



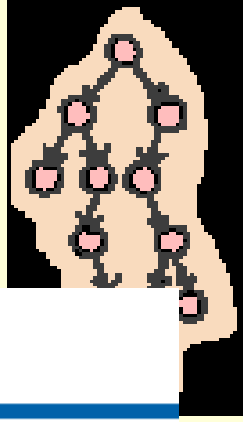
- Constructor:
  - Gọi constructor của mỗi danh sách trong array.
- Clear:
  - Gọi phương thức clear cho mỗi danh sách trong array.
- Retrieval:
  - sequential_search(table[hash(target)], target, position);
- Insertion:
  - table[hash(new_entry)].insert(0, new_entry);
- Deletion:
  - remove(const Key type &target, Record &x);
  - Nếu tìm thấy trong danh sách tương ứng thì xóa đi

Đánh giá phương pháp dùng bảng Hash



- load factor λ = số mẫu tin/kích thước bảng hash
- Tìm kiếm với bảng hash nối kết:
  - 1+(1/2)λ phép thử khi tìm thấy
  - λ phép thử khi không tìm thấy.
- Tìm với bảng hash địa chỉ mở (thử ngẫu nhiên):
  - (1/λ)ln (1/(1-λ)) phép thử khi tìm thấy
  - 1/(1-λ) phép thử khi không tìm thấy
- Tìm với bảng hash địa chỉ mở (thử tuyến tính):
  - (1/2)(1 + 1/(1-λ)) phép thử khi tìm thấy
  - (1/2)(1 + 1/(1-λ)^2) phép thử khi không tìm thấy

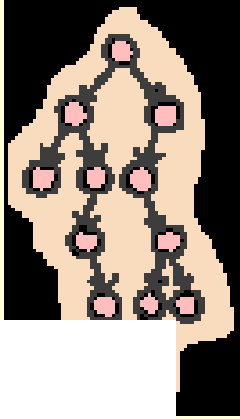
So sánh các phương pháp



Theoretical comparisons:

Load factor	0.10	0.50	0.80	0.90	0.99	2.00
Successful search, expected number of probes:						
Chaining	1.05	1.25	1.40	1.45	1.50	2.00
Open, Random probes	1.05	1.4	2.0	2.6	4.6	—
Open, Linear probes	1.06	1.5	3.0	5.5	50.5	—
Unsuccessful search, expected number of probes:						
Chaining	0.10	0.50	0.80	0.90	0.99	2.00
Open, Random probes	1.1	2.0	5.0	10.0	100.	—
Open, Linear probes	1.12	2.5	13.	50.	5000.	—

So sánh các phương pháp (tt.)

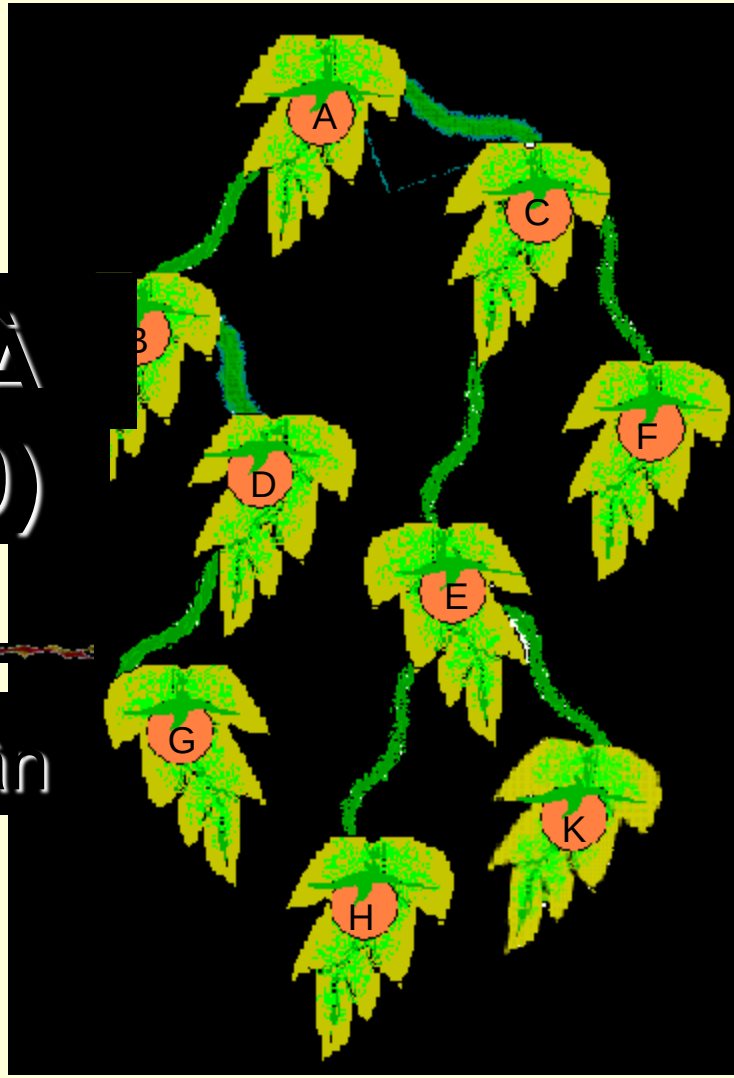


Empirical comparisons:

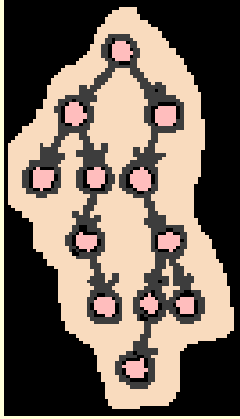
Load factor	0.1	0.5	0.8	0.9	0.99	2.0
Successful search, average number of probes:						
Chaining	1.04	1.2	1.4	1.4	1.5	2.0
Open, Quadratic probes	1.04	1.5	2.1	2.7	5.2	—
Open, Linear probes	1.05	1.6	3.4	6.2	21.3	—
Unsuccessful search, average number of probes:						
Chaining	0.10	0.50	0.80	0.90	0.99	2.00
Open, Quadratic probes	1.13	2.2	5.2	11.9	126.	—
Open, Linear probes	1.13	2.7	15.4	59.8	430.	—

CẤU TRÚC DỮ LIỆU VÀ GIẢI THUẬT (501040)

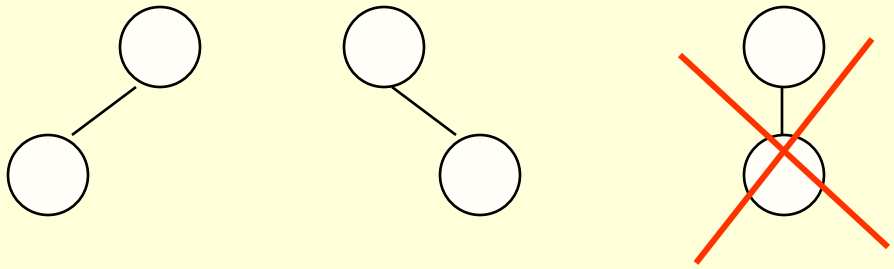
Chương 10: Cây nhị phân



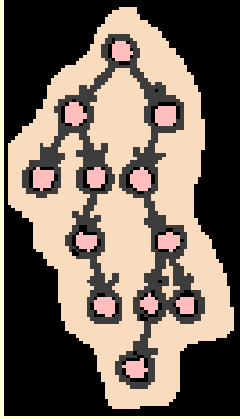
Định nghĩa



- Cây nhị phân
  - Cây rỗng
  - Hoặc có một node gọi là gốc (root) và 2 cây con gọi là cây con trái và cây con phải
- Ví dụ:
  - Cây rỗng:
  - Cây có 1 node: là node gốc
  - Cây có 2 node:

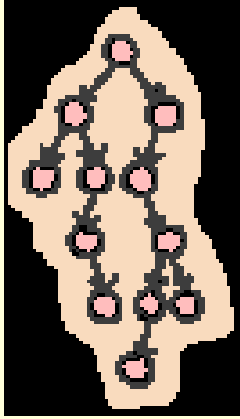


Các định nghĩa khác



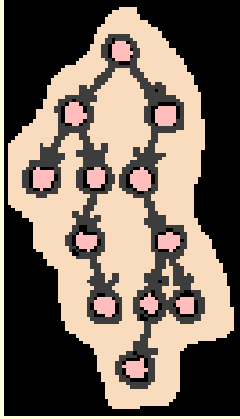
- Mức:
  - Node gốc ở mức 0.
  - Node gốc của các cây con của một node ở mức *m* là *m+1*.
- Chiều cao:
  - Cây rỗng là 0.
  - Chiều cao lớn nhất của 2 cây con cộng 1
  - (Hoặc: mức lớn nhất của các node cộng 1)
- Đường đi (path)
  - Tên các node của quá trình đi từ node gốc theo các cây con đến một node nào đó.

Các định nghĩa khác (tt.)



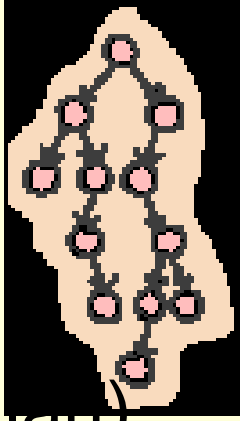
- Node trước, sau, cha, con:
  - Node x là trước node y (node y là sau node x), nếu trên đường đi đến y có x.
  - Node x là cha node y (node y là con node x), nếu trên đường đi đến y node x nằm ngay trước node y.
- Node lá, trung gian:
  - Node lá là node không có cây con nào.
  - Node trung gian không là node gốc hay node lá.

Các tính chất khác



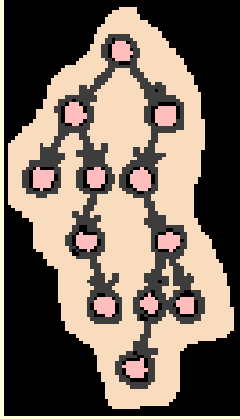
- Cây nhị phân đầy đủ, gần đầy đủ:
  - Đầy đủ: các node lá luôn nằm ở mức cao nhất và các nút không là nút lá có đầy đủ 2 nhánh con.
  - Gần đầy đủ: Giống như trên nhưng các node lá nằm ở mức cao nhất (hoặc trước đó một mức) và lấp đầy từ bên trái sang bên phải ở mức cao nhất.
- Chiều cao của cây có *n* node:
  - Trung bình $h = [\lg n] + 1$
  - Đầy đủ $h = \lg (n + 1)$
  - Suy biến $h = n$
- Số phần tử tại mức *i* nhiều nhất là 2^i

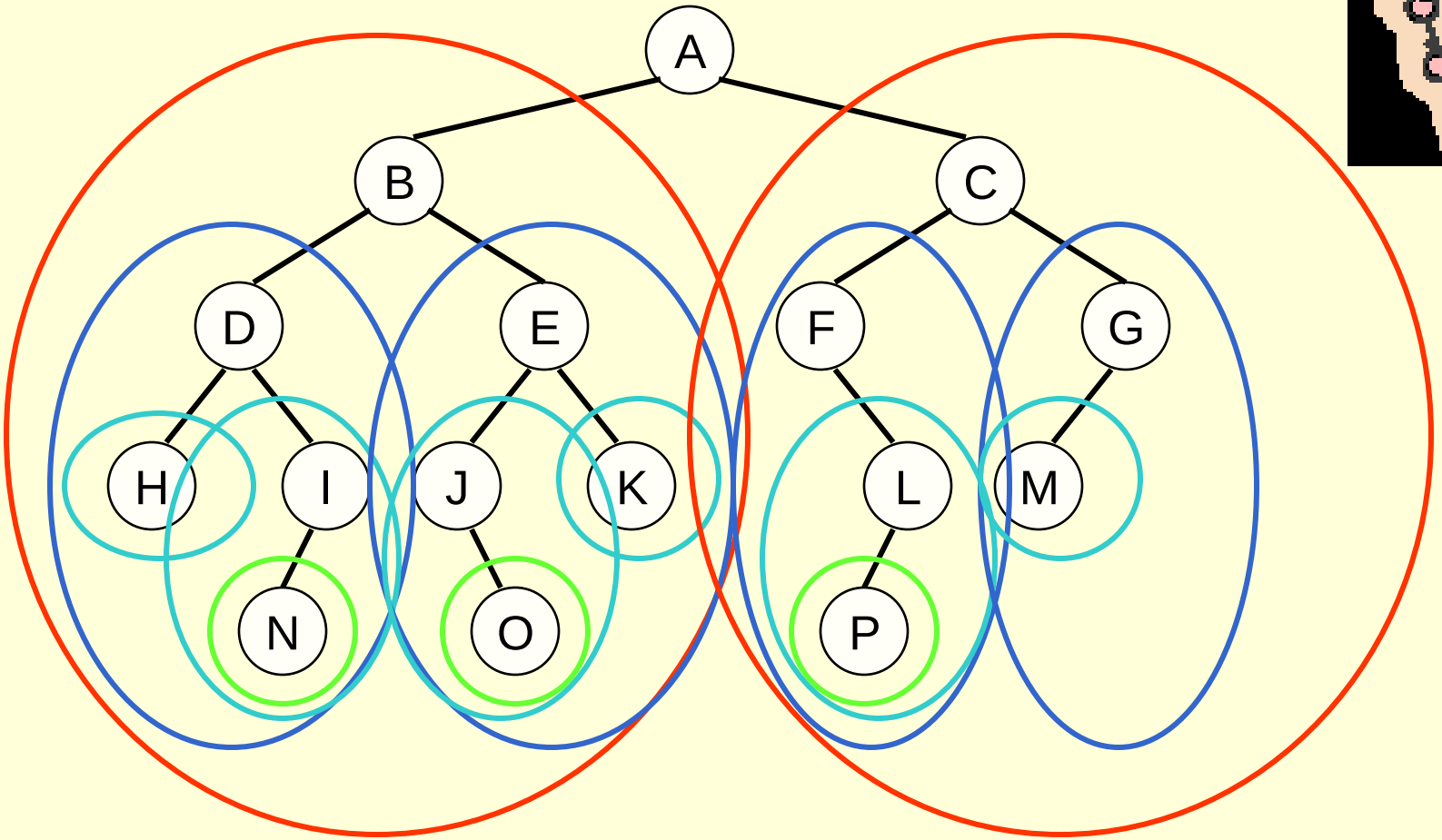
Phép duyệt cây



- Duyệt qua từng node của cây (mỗi node 1 lần)
- Cách duyệt:
  - Chính thức: NLR, LNR, LRN, NRL, RNL, RLN
  - Chuẩn: NLR (preorder), LNR (inorder), LRN (postorder)

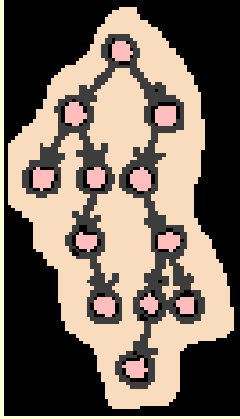
Ví dụ về phép duyệt cây NLR

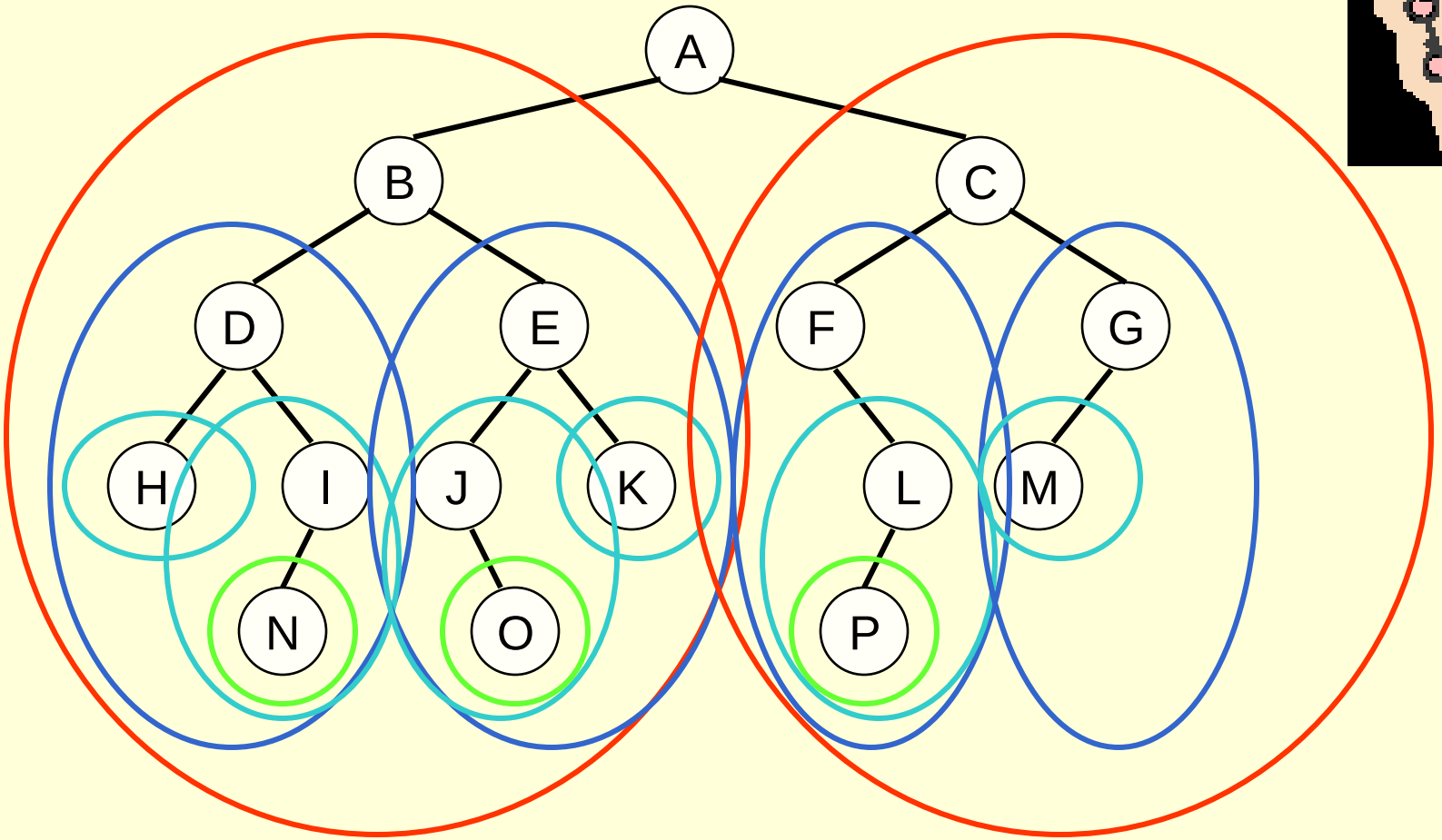




Kết quả: A B D H I N E J O K C F L P G M

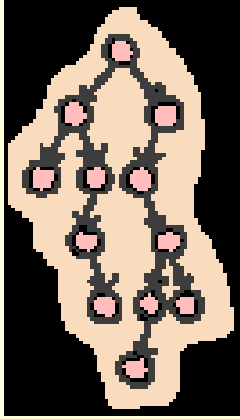
Ví dụ về phép duyệt cây LNR

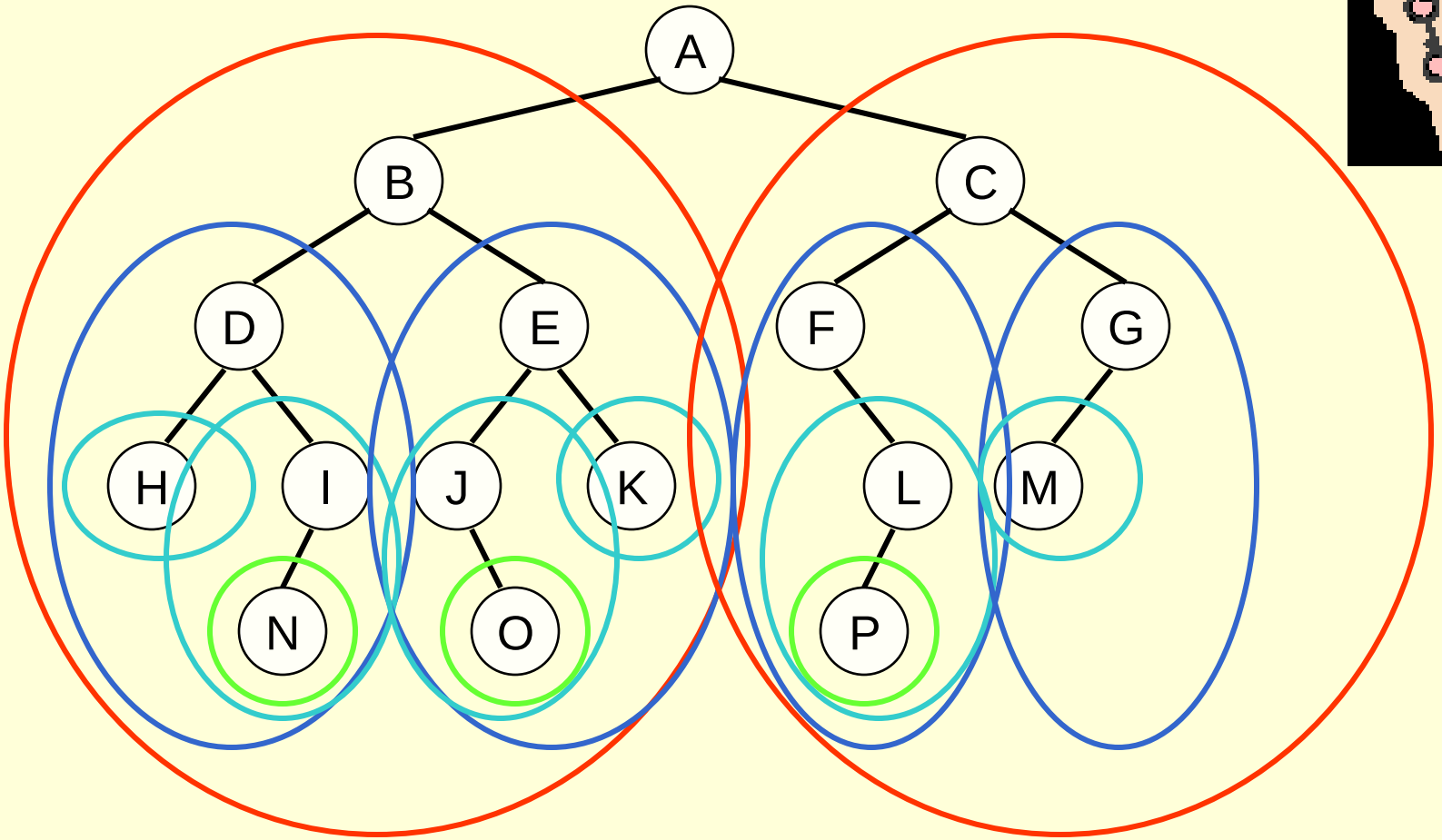




Kết quả: H D N I B J O E K A F P L C M G

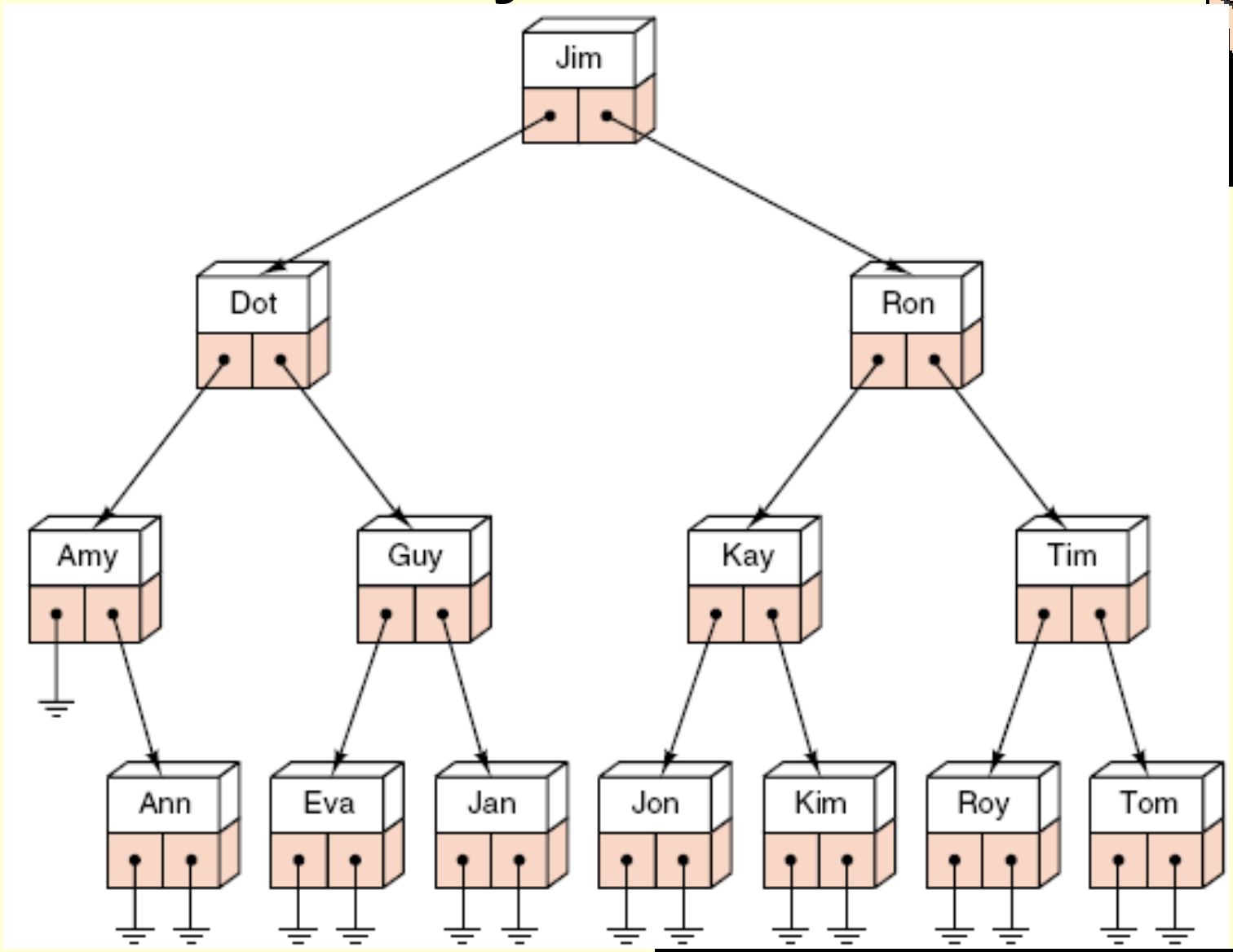
Ví dụ về phép duyệt cây LRN



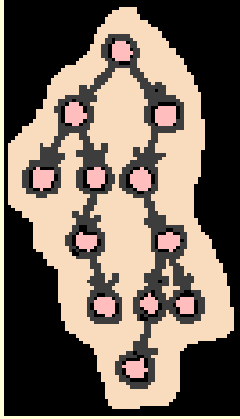


Kết quả: H N I D O J K E B P L F M G C A

Cây liên kết



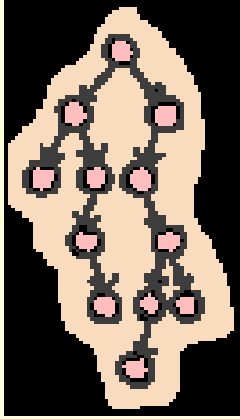
Thiết kế cây liên kết



```
template <class Entry>
struct Binary_node {
    // data members:
    Entry data;
    Binary_node<Entry> *left, *right;
    // constructors:
    Binary_node( );
    Binary_node(const Entry &x);
};
```

```
template <class Entry>
class Binary_tree {
public:
    // Add methods here.
protected:
    // Add auxiliary function prototypes here.
    Binary_node<Entry> *root;
};
```

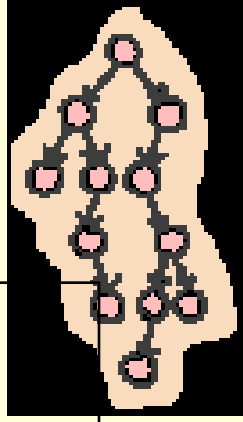
Khởi tạo và kiểm tra rỗng



```
template <class Entry>
Binary_tree<Entry>::Binary_tree() {
    root = NULL;
};
```

```
template <class Entry>
bool Binary_tree<Entry>::empty() {
    return root == NULL;
};
```

Thiết kế các phép duyệt cây

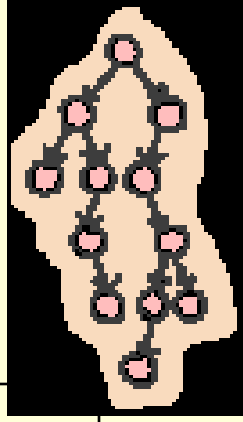


```
template <class Entry>
void Binary_tree<Entry> :: inorder(void (*visit)(Entry &)) {
    recursive_inorder(root, visit);
}
```

```
template <class Entry>
void Binary_tree<Entry> :: preorder(void (*visit)(Entry &)) {
    recursive_preorder(root, visit);
}
```

```
template <class Entry>
void Binary_tree<Entry> :: postorder(void (*visit)(Entry &)) {
    recursive_postorder(root, visit);
}
```

Giải thuật duyệt cây inorder

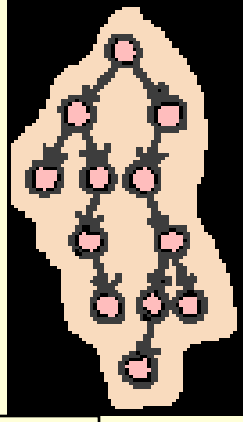


Algorithm recursive_inorder
Input: subroot là con trỏ node gốc và hàm *visit*
Output: kết quả phép duyệt

1. **if** (cây con không rỗng)
   1.1. **Call** recursive_inorder với nhánh trái của subroot
   1.2. Duyệt node subroot bằng hàm *visit*
   1.3. **Call** recursive_inorder với nhánh phải của subroot

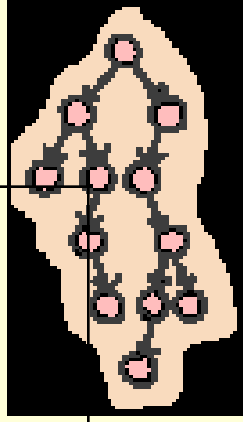
End recursive_inorder

Mã C++ duyệt cây inorder



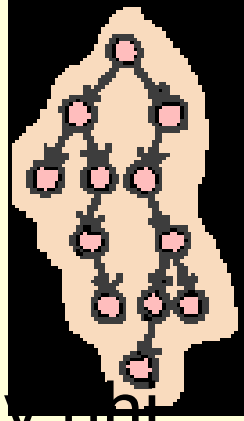
```
template <class Entry>
void Binary_tree<Entry> ::recursive_inorder
            (Binary_node<Entry> *sub_root, void (*visit)(Entry &)) {
    if (sub_root != NULL) {
        recursive_inorder(sub_root->left, visit);
        (*visit)(sub_root->data);
        recursive_inorder(sub_root->right, visit);
    }
}
```

Khai báo cây nhị phân



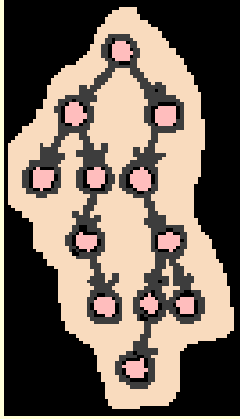
```
template <class Entry>
class Binary_tree {
public:
    Binary_tree( );
    bool empty( ) const;
    void preorder(void (*visit)(Entry &));
    void inorder(void (*visit)(Entry &));
    void postorder(void (*visit)(Entry &));
    int size( ) const;
    void clear( );
    int height( ) const;
    void insert(const Entry &);
    Binary_tree (const Binary_tree<Entry> &original);
    Binary_tree & operator = (const Binary_tree<Entry> &original);
    ~Binary_tree( );
protected:
    Binary_node<Entry> *root;
};
```

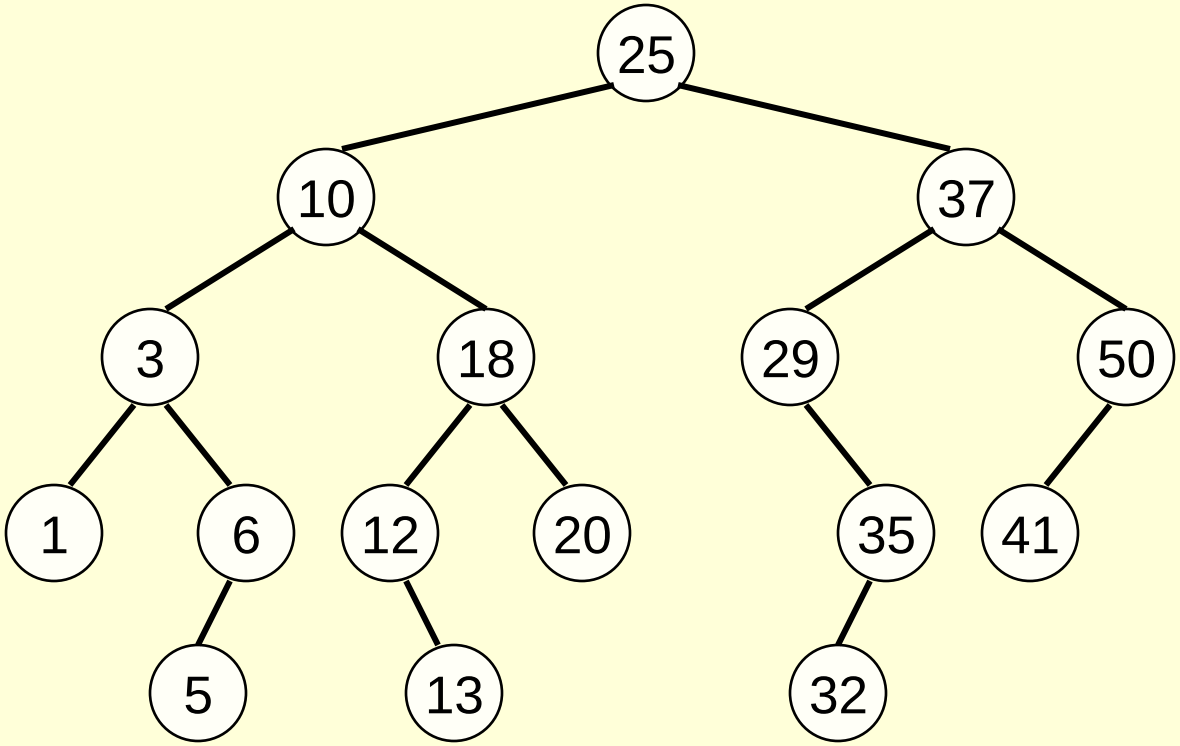
Cây nhị phân tìm kiếm – Binary search tree (BST)



- Một cây nhị phân tìm kiếm (BST) là một cây nhị phân rỗng hoặc mỗi node của cây này có các đặc tính sau:
  - 1. Khóa của node gốc lớn (hay nhỏ) hơn khóa của tất cả các node của cây con bên trái (hay bên phải)
  - 2. Các cây con (bên trái, phải) là BST
- Tính chất:
  - Chỉ cần đặc tính 1 là đủ
  - Duyệt inorder sẽ được danh sách có thứ tự

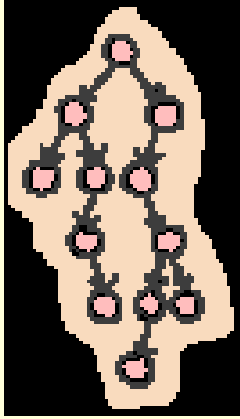
Ví dụ BST





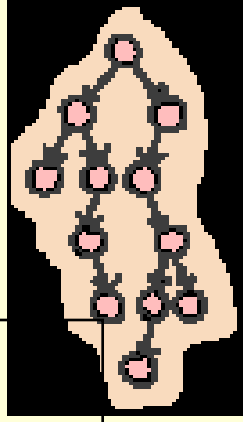
Duyệt inorder: 1 3 5 6 10 12 13 18 20 25 29 32 35 37 41 50

Các tính chất khác của BST



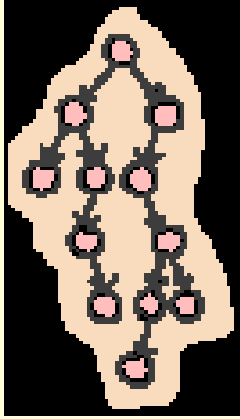
- Node cực trái (hay phải):
  - Xuất phát từ node gốc
  - Đi sang trái (hay phải) đến khi không đi được nữa
- Khóa của node cực trái (hay phải) là nhỏ nhất (hay lớn nhất) trong BST
- BST là cây nhị phân có tính chất:
  - Khóa của node gốc lớn (hay nhỏ) hơn khóa của node cực trái (hay cực phải)

Thiết kế BST



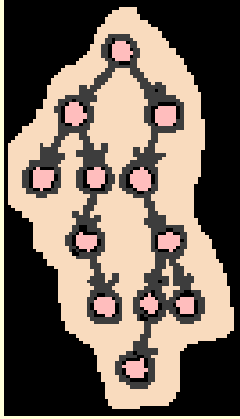
```
template <class Record>
class Search_tree: public Binary_tree<Record> {
public:
    //Viết lại phương thức chèn vào, loại bỏ để đảm bảo vẫn là BST
    Error_code insert(const Record &new_data);
    Error_code remove(const Record &old_data);

    //Thêm phương thức tìm kiếm dựa vào một khóa
    Error_code tree_search(Record &target) const;
private:
    // Add auxiliary function prototypes here.
};
```

Tìm kiếm trên BST



- Chọn hướng tìm theo tính chất của BST:
  - So sánh với node gốc, nếu đúng thì tìm thấy
  - Tìm bên nhánh trái (hay phải) nếu khóa cần tìm nhỏ hơn (hay lớn hơn) khóa của node gốc
- Giống phương pháp tìm kiếm nhị phân
- Thời gian tìm kiếm
  - Tốt nhất và trung bình: $O(\lg n)$
  - Tệ nhất: $O(n)$

Giải thuật tìm kiếm trên BST

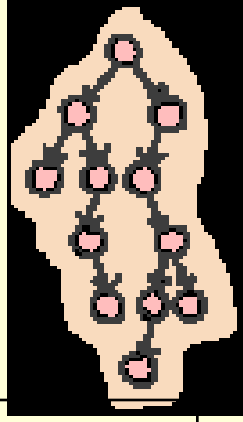


Algorithm BST_search
Input: subroot là node gốc và target là khóa cần tìm
Output: node tìm thấy

1. **if** (cây rỗng)
   1.1. **return** not_found
2. **if** (target trùng khóa với subroot)
   2.1. **return** subroot
3. **if** (target có khóa nhỏ hơn khóa của subroot)
   3.1. Tìm bên nhánh trái của subroot
4. **else**
   4.1. Tìm bên nhánh phải của subroot

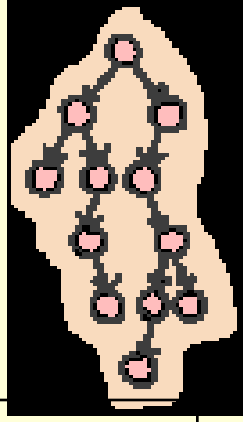
End BST_search

Mã C++ tìm kiếm trên BST



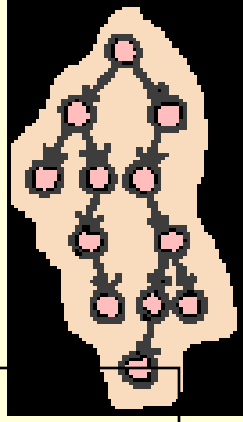
```
template <class Record>
Binary_node<Record> *Search_tree<Record> :: search_for_node
          (Binary_node<Record>* sub_root, const Record &target) const {

   if (sub_root == NULL || sub_root->data == target)
      return sub_root;
   else if (sub_root->data < target)
      return search_for_node(sub_root->right, target);
   else return search_for_node(sub_root->left, target);
}
```

Mã C++ tìm kiếm trên BST (không đệ qui)



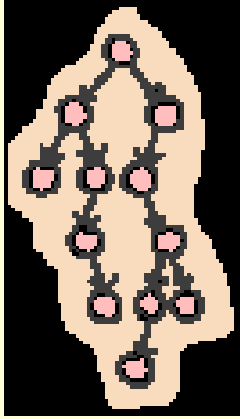
```
template <class Record>
Binary_node<Record> *Search_tree<Record> :: search_for_node
          (Binary_node<Record>* sub_root, const Record &target) const {

   while (sub_root != NULL && sub_root->data != target)
      if (sub_root->data < target) sub_root = sub_root->right;
      else sub_root = sub_root->left;
   return sub_root;
}
```

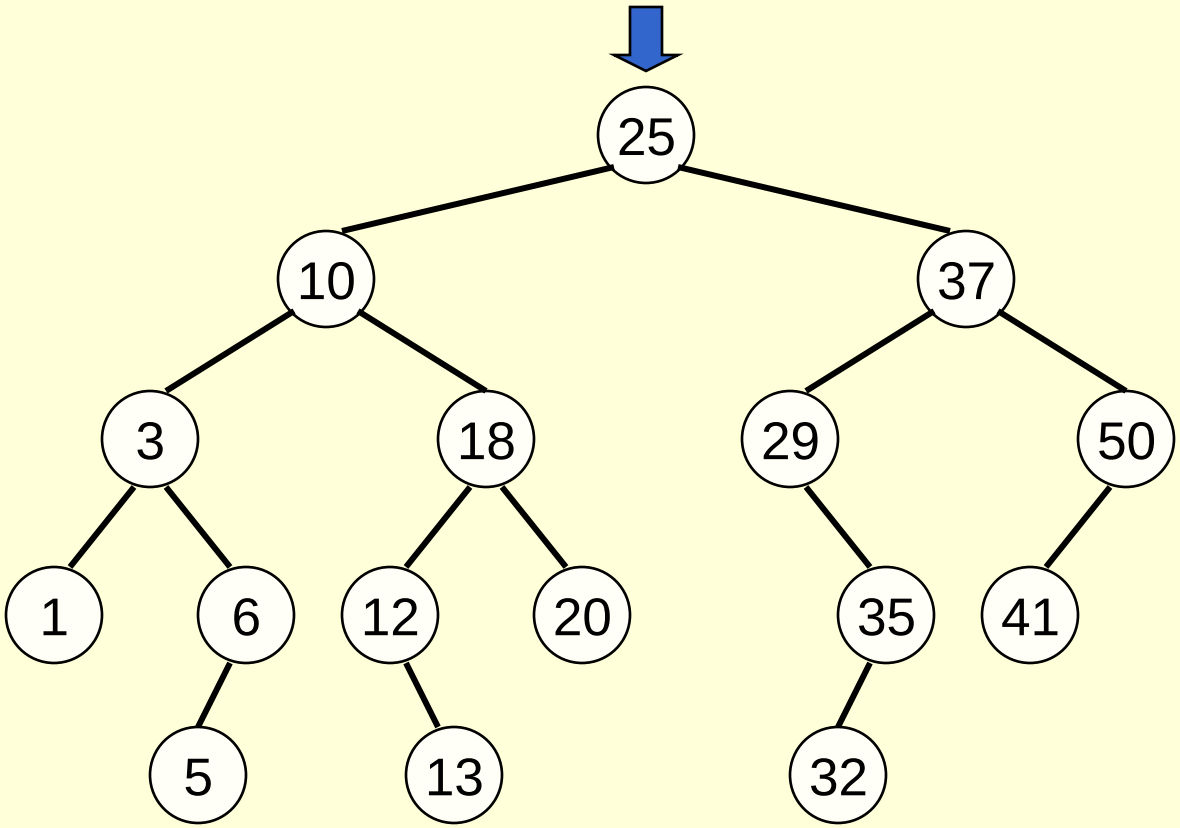
Phương thức tree_search



```
template <class Record>
Error_code Search_tree<Record> :: tree_search(Record &target) const {

    Error_code result = success;
    Binary_node<Record> *found = search_for_node(root, target);
    if (found == NULL)
        result = not_present;
    else
        target = found->data;
    return result;
}
```

Ví dụ tìm kiếm trên BST





Khóa nhỏ hơn

Khóa lớn hơn

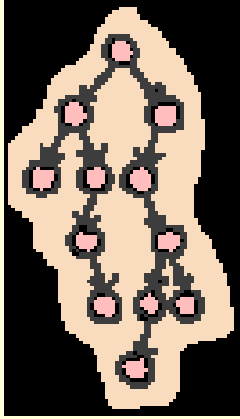
Tìm kiếm 13

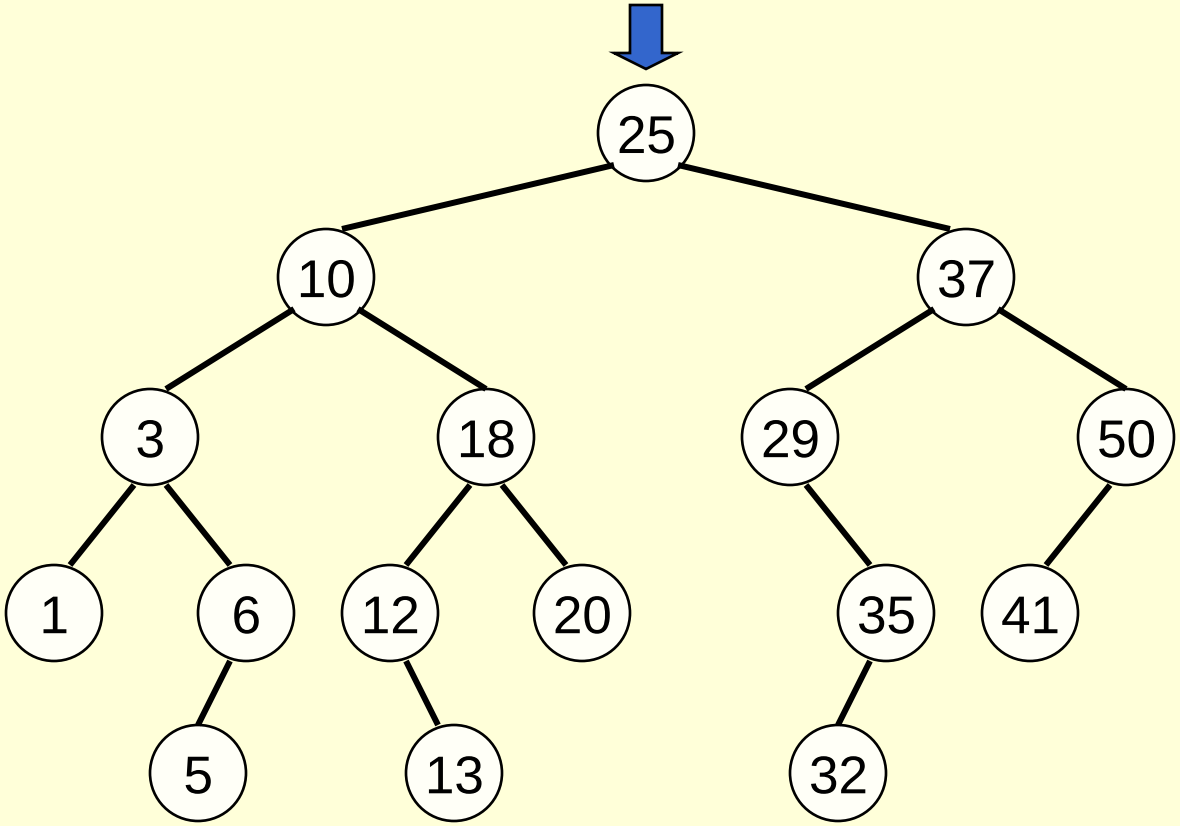
Tìm thấy

Số node duyệt: 5

Số lần so sánh: 9

Ví dụ tìm kiếm trên BST



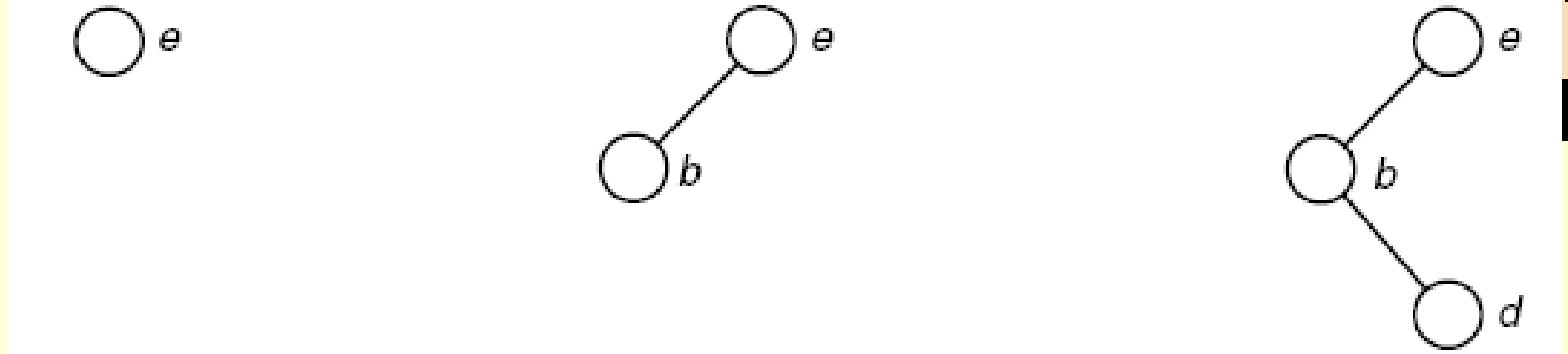


Khác gốc nhỏ hơn

Tìm kiếm 14 **Không tìm thấy**

Số node duyệt: 5
Số lần so sánh: 10

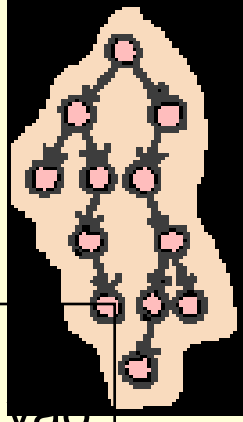
Thêm vào BST



(a) Insert *e* (b) Insert *b* (c) Insert *d*

(d) Insert *f* (e) Insert *a*

Giải thuật thêm vào BST

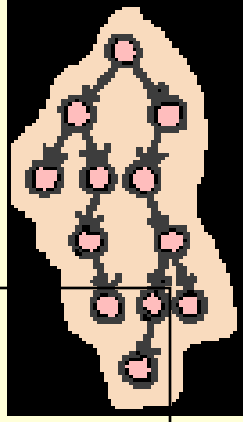


Algorithm BST_insert
Input: subroot là node gốc và new_data là dữ liệu cần thêm vào
Output: BST sau khi thêm vào

1. **if** (cây rỗng)
   1.1. Thêm vào tại vị trí này
2. **if** (target trùng khóa với subroot)
   2.1. **return** duplicate_error
3. **if** (new_data có khóa nhỏ hơn khóa của subroot)
   3.1. Thêm vào bên nhánh trái của subroot
4. **else**
   4.1. Thêm vào bên nhánh phải của subroot

End BST_insert

Mã C++ thêm vào BST



```
template <class Record>
Error_code Search_tree<Record> :: search_and_insert
          (Binary_node<Record> * &sub_root, const Record &new_data) {

   if (sub_root == NULL) {
      sub_root = new Binary_node<Record>(new_data);
      return success;
   }
   else if (new_data < sub_root->data)
      return search_and_insert(sub_root->left, new_data);
   else if (new_data > sub_root->data)
      return search_and_insert(sub_root->right, new_data);
   else return duplicate_error;
}
```

Algorithm BST_insert

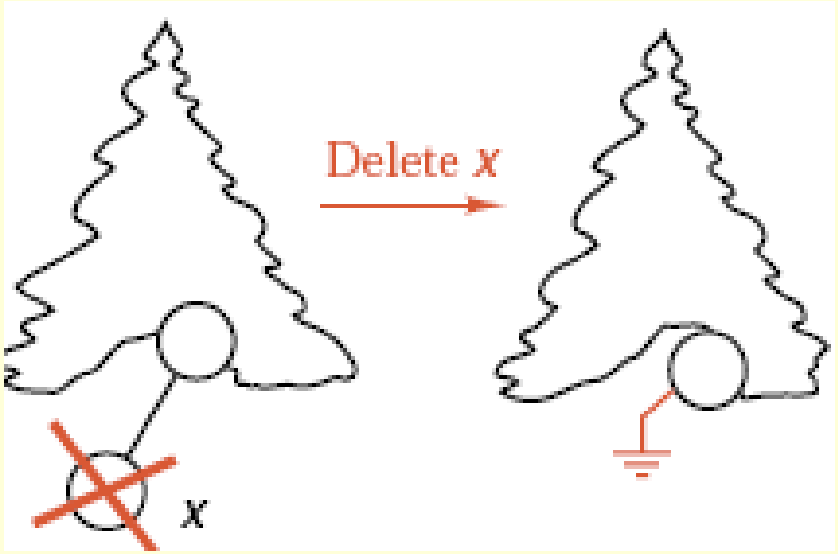
Input: subroot là node gốc và new_data là dữ liệu cần thêm vào

Output: BST sau khi thêm vào

1. parent là rỗng và left_or_right là “left”
2. **while** (subroot không rỗng)
   - 2.1. **if** (target trùng khóa với subroot)
     - 2.1.1. **return** duplicate_error
   - 2.2. **if** (new_data có khóa nhỏ hơn khóa của subroot)
     - 2.2.2. parent là subroot và left_or_right là “left”
     - 2.2.1. Chuyển subroot sang nhánh trái của subroot
   - 2.3. **else**
     - 2.3.2. parent là subroot và left_or_right là “right”
     - 2.3.1. Chuyển subroot sang nhánh phải của subroot
3. **if** (subroot là rỗng) *//Thêm vào tại vị trí này*
   - 3.1. **if** (parent là rỗng)
     - 3.1.1. Tạo node gốc của cây
   - 3.2. **else**
     - 3.2.1. Tạo node bên trái hay phải parent tùy theo left_or_right

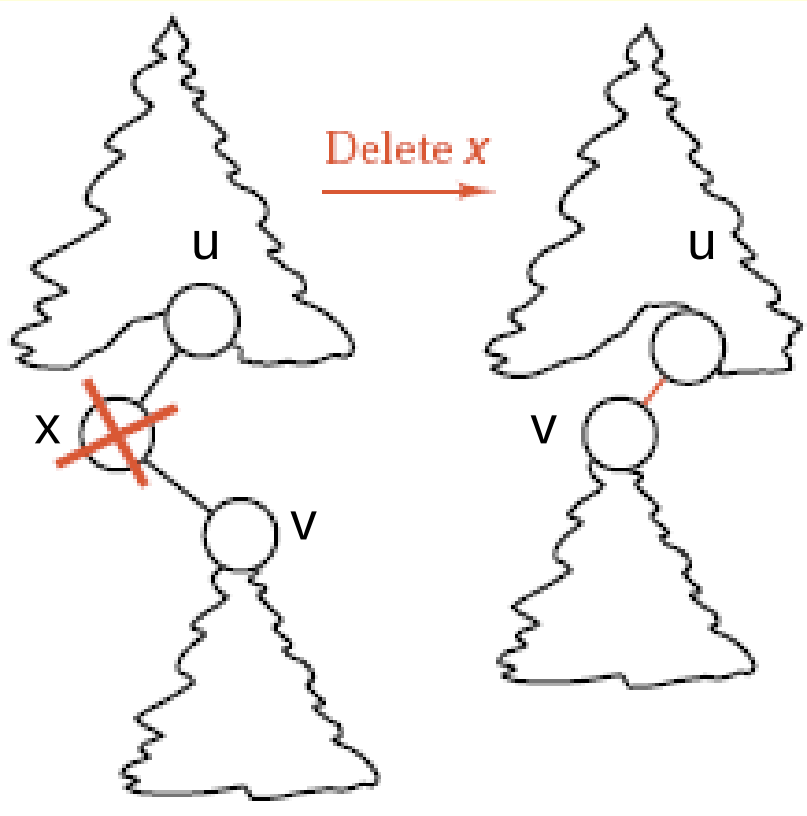
End BST_insert

Xóa một node lá khỏi BST



1. Xóa node này
2. Gán liên kết từ cha của nó thành rỗng

Xóa một node chỉ có một con



1. Gán liên kết từ cha của nó xuống con duy nhất của nó
2. Xóa node này

A. Đường dẫn đến các node của cây con v có dạng:

… u x v …

B. Không còn node nào trong cây có đường dẫn có dạng như vậy.

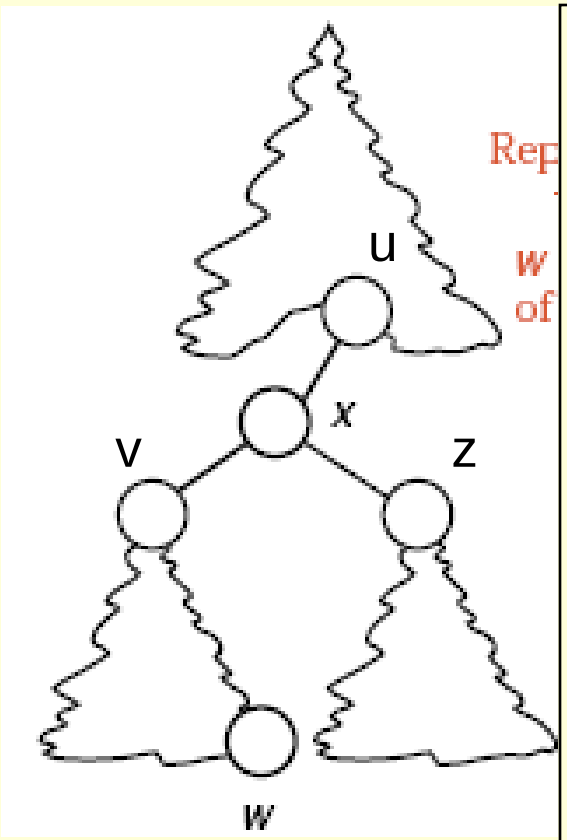
C. Sau khi xóa node x, đường dẫn đến các node của cây con v có dạng:

… u v …

D. Đường dẫn của các node khác trong cây không đổi.

E. Trước đó, các node của cây con v nằm trong nhánh con của x là bên trái (bên phải) của u và bây giờ cũng nằm bên trái (bên phải) của u nên vẫn thỏa mãn BST

Xóa một node có đủ 2 nhánh con



A. Đường dẫn đến các node của cây con v và z có dạng:

… u x v …

… u x z …

B. Nếu xóa node x thì đường dẫn đến các node của cây con v và z có dạng:

… u v …

… u z …

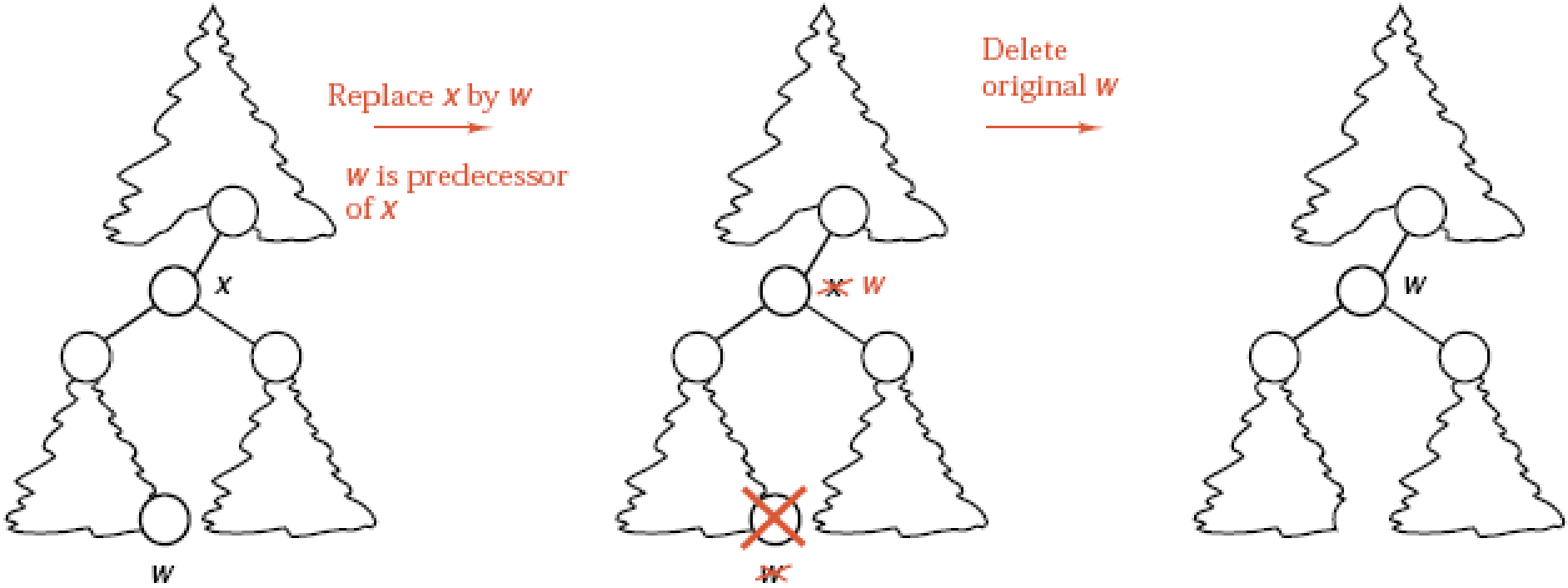
D. Điều này chỉ xảy ra khi cây con u và v nằm về 2 phía của u => không còn là BST.

E. Giải pháp là thay giá trị x bằng giá trị w thuộc cây này sao cho:

w lớn hơn tất cả khóa của các node của cây con v

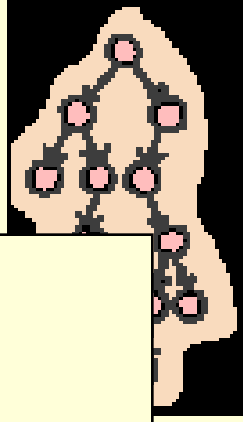
w nhỏ hơn tất cả khóa của các node của cây con z

Xóa một node có đủ 2 nhánh con (tt.)



1. Tìm w là node trước node x trên phép duyệt cây inorder (chính là node cực phải của cây con bên trái của x)
2. Thay x bằng w
3. Xóa node w cũ (giống trường hợp 1 hoặc 2 đã xét)

Giải thuật xóa một node

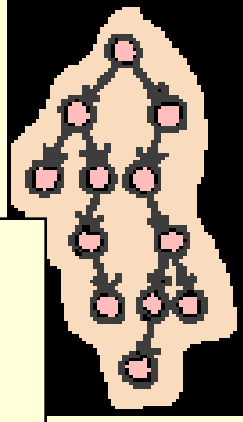


Algorithm BST_remove_root
Input: subroot là node gốc cần phải xóa
Output: BST sau khi xóa xong subroot

1. **if** (trường hợp 1 hoặc 2) *//subroot là node lá hoặc có một con*
   1.1. Gán liên kết cha đến rỗng hoặc nhánh con duy nhất của subroot
   1.2. Xóa subroot
2. **else** *//trường hợp 3: có 2 nhánh con*
   //Tìm node cực phải của cây con trái
   2.1. to_delete là node con trái của subroot
   2.2. **while** (nhánh phải của to_delete không rỗng)
      2.2.1. di chuyển to_delete sang node con phải
   2.2. Thay dữ liệu của subroot bằng dữ liệu của to_delete
   2.4. **Call** BST_remove_root với to_delete

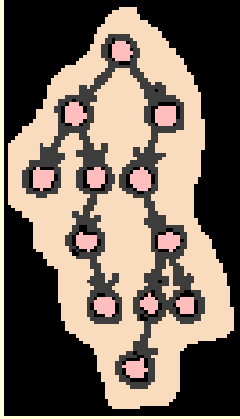
End BST_remove_root

Mã C++ xóa một node



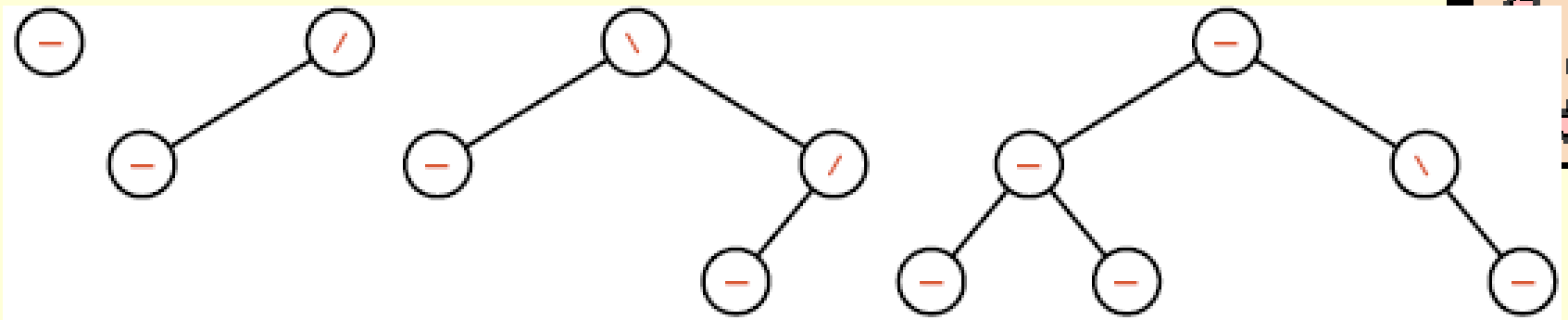
```
template <class Record>
Error_code Search_tree<Record> :: remove_root
            (Binary_node<Record> * &sub_root) {
    if (sub_root == NULL) return not_present;
    Binary_node<Record> *to_delete = sub_root;
    if (sub_root->right == NULL) sub_root = sub_root->left;
    else if (sub_root->left == NULL) sub_root = sub_root->right;
    else { to_delete = sub_root->left;
        Binary_node<Record> *parent = sub_root;
        while (to_delete->right != NULL) {
            parent = to_delete;
            to_delete = to_delete->right; }
        sub_root->data = to_delete->data;
        if (parent == sub_root) sub_root->left = to_delete->left;
        else parent->right = to_delete->left; }
    delete to_delete;
    return success;  }
```

Cây cân bằng chiều cao - AVL

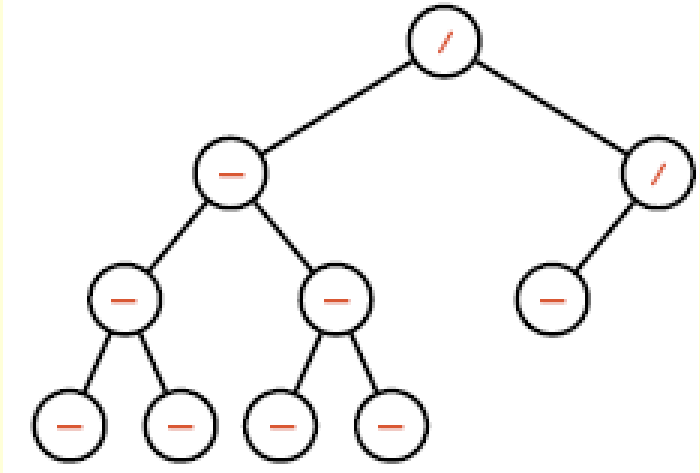


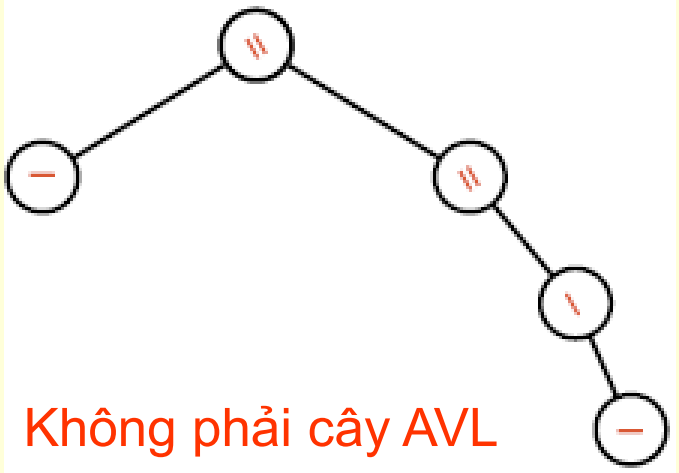
- Cây cân bằng hoàn toàn:
  - Số node của nhánh trái và nhánh phải chênh nhau không quá 1.
- ĐN cây AVL:
  - BST
  - Tại node bất kỳ, chiều cao nhánh trái và nhánh phải chênh nhau không quá 1.
- Ký hiệu cho mỗi node của cây AVL:
  - Node cân bằng: ‘-’
  - Nhánh trái cao hơn: ‘/’
  - Nhánh phải cao hơn: ‘\’

Ví dụ cây AVL



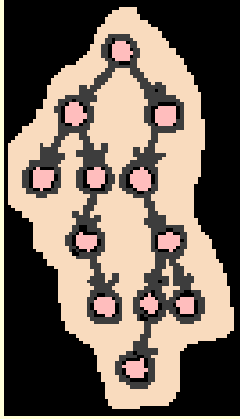
Cây AVL





Không phải cây AVL

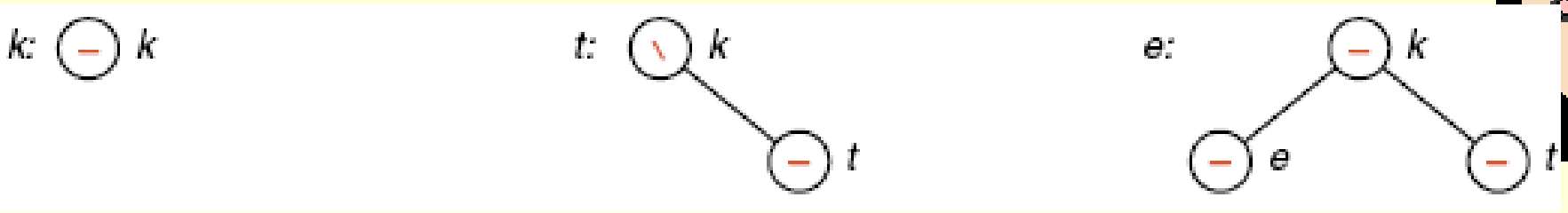
Khai báo cây AVL

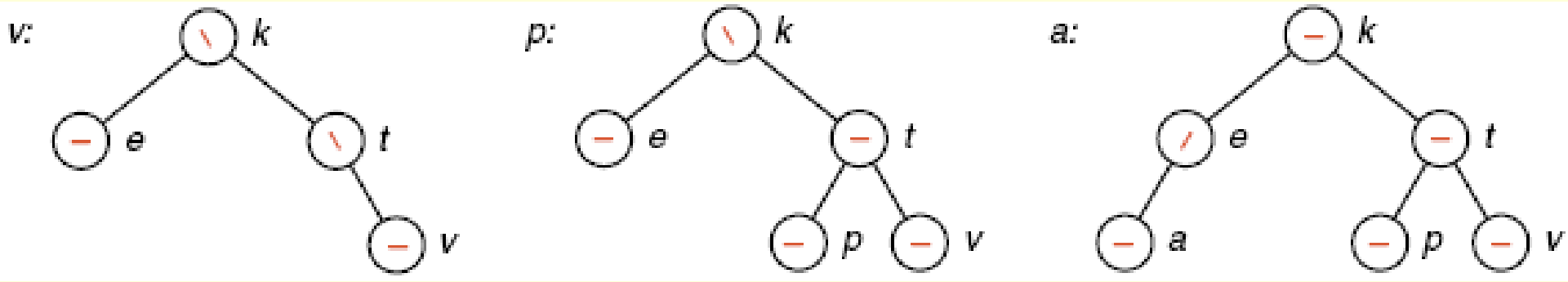


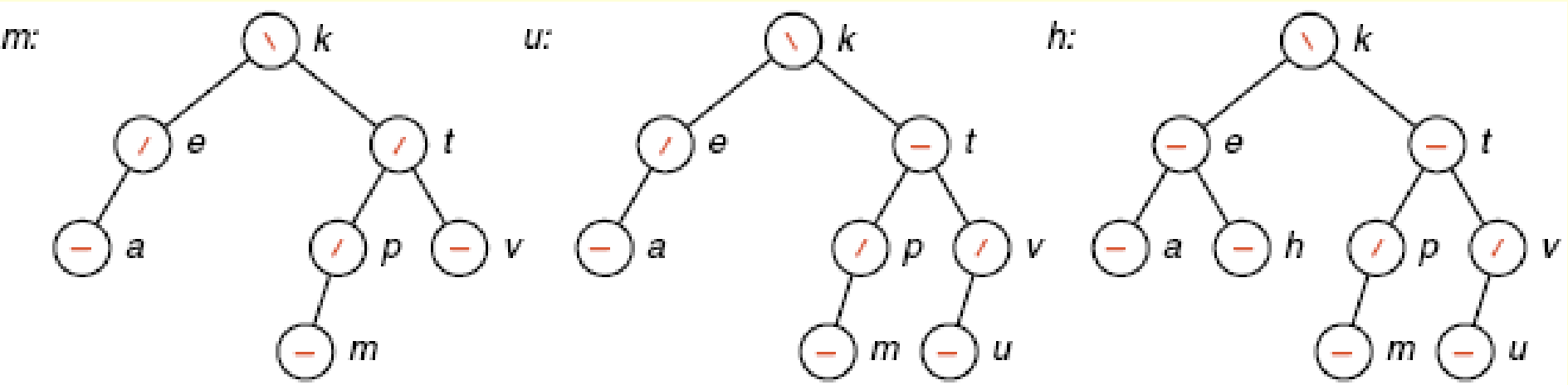
```
enum Balance_factor { left_higher, equal_height, right_higher };
template <class Record>
struct AVL_node: public Binary_node<Record> {
    // additional data member:
    Balance_factor balance;
    AVL_node( );
    AVL_node(const Record &x);
    void set_balance(Balance_factor b);
    Balance_factor get_balance( ) const;
};
```

```
template <class Record>
class AVL_tree: public Search_tree<Record> {
public:
    Error_code insert(const Record &new data);
    Error_code remove(const Record &old data);
private:
    // Add auxiliary function prototypes here.
};
```

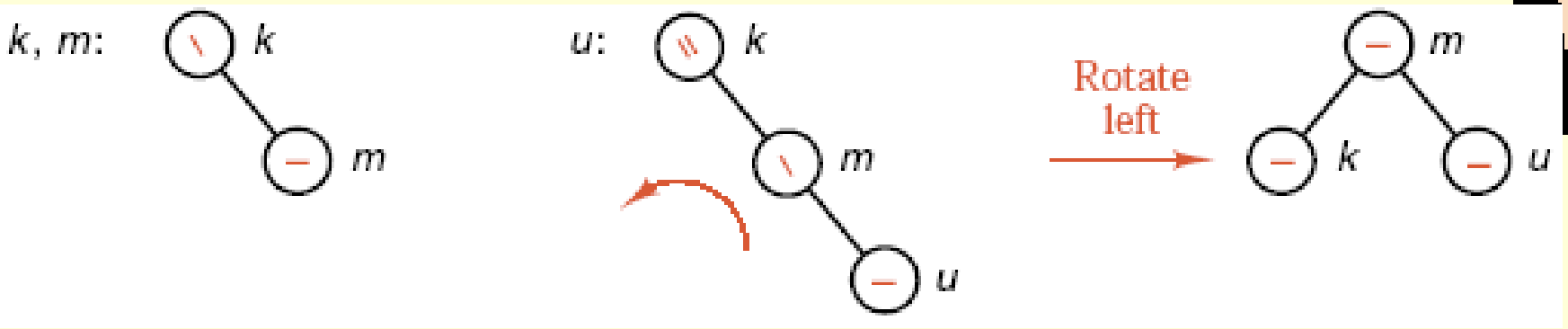
Ví dụ 1 thêm vào cây AVL

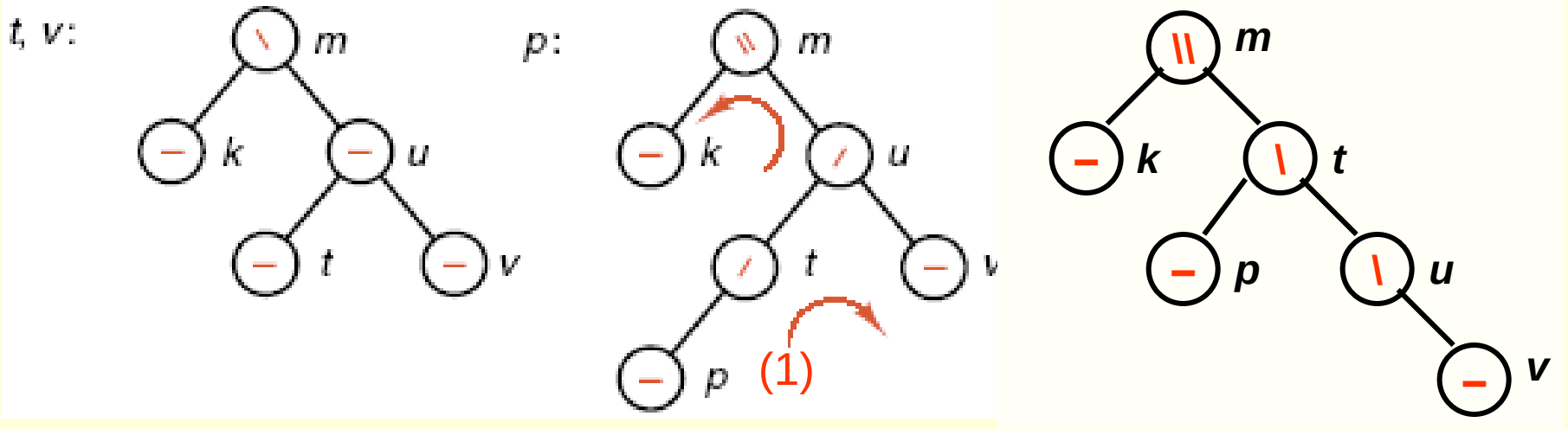




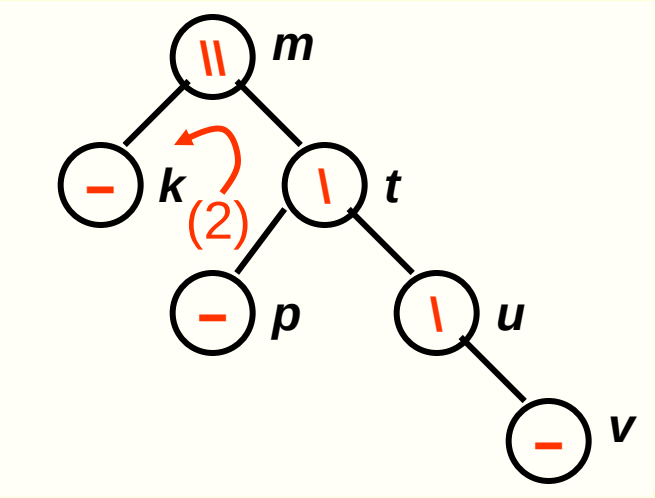


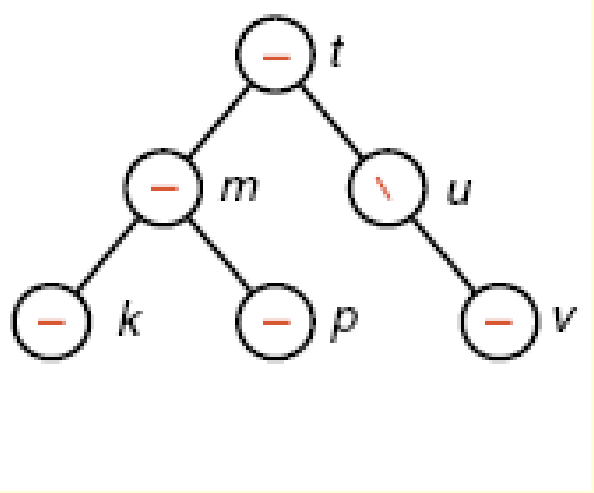
Ví dụ 2 thêm vào cây AVL



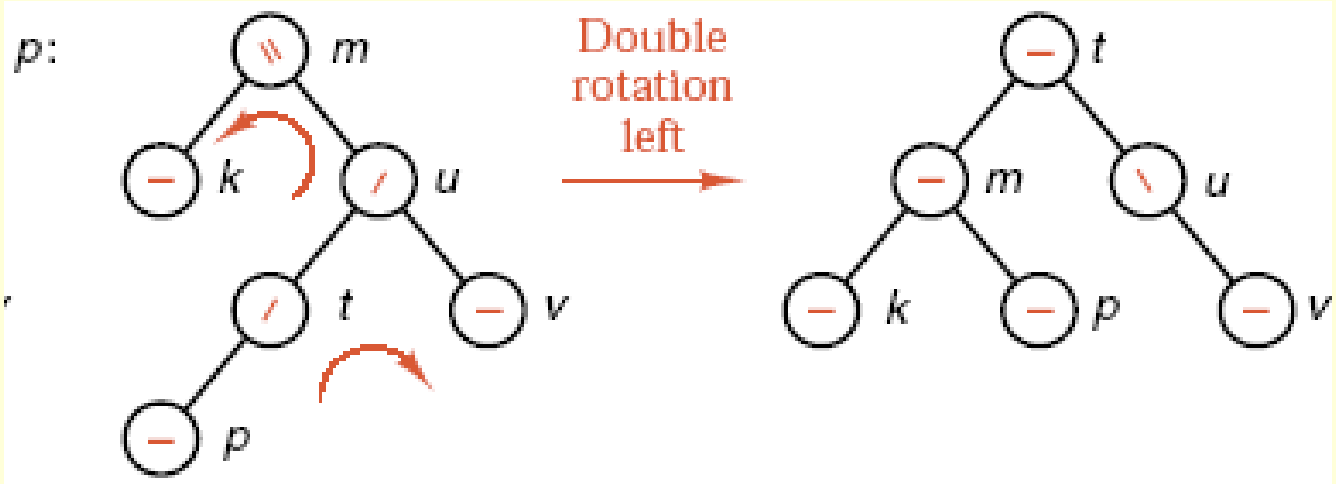


Ví dụ 2 thêm vào cây AVL (tt.)

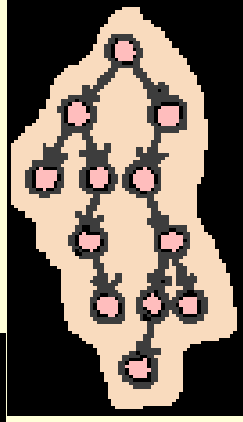


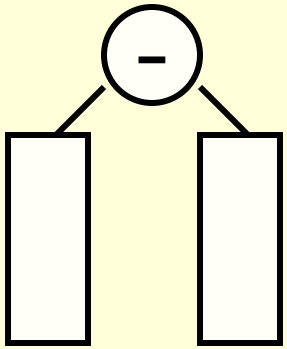


Viết gọn

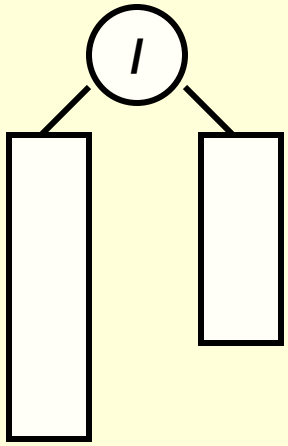


Các trạng thái khi thêm vào

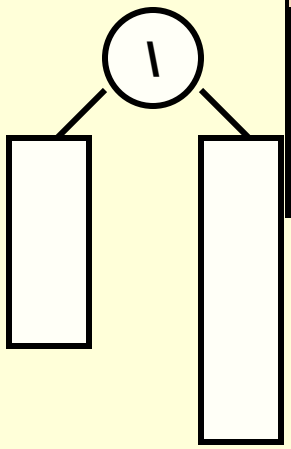




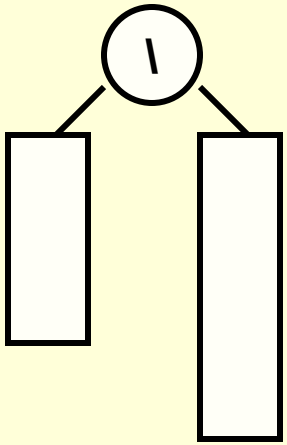
Thêm vào bên phải và làm bên phải cao lên



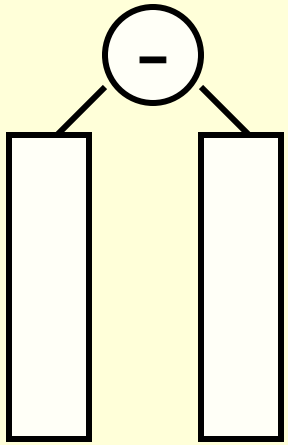
Thêm vào bên phải và làm bên phải cao lên



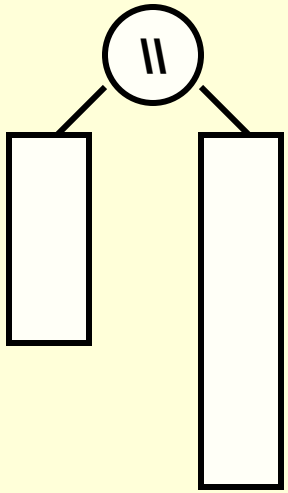
Thêm vào bên phải và làm bên phải cao lên



Chiều cao cây tăng

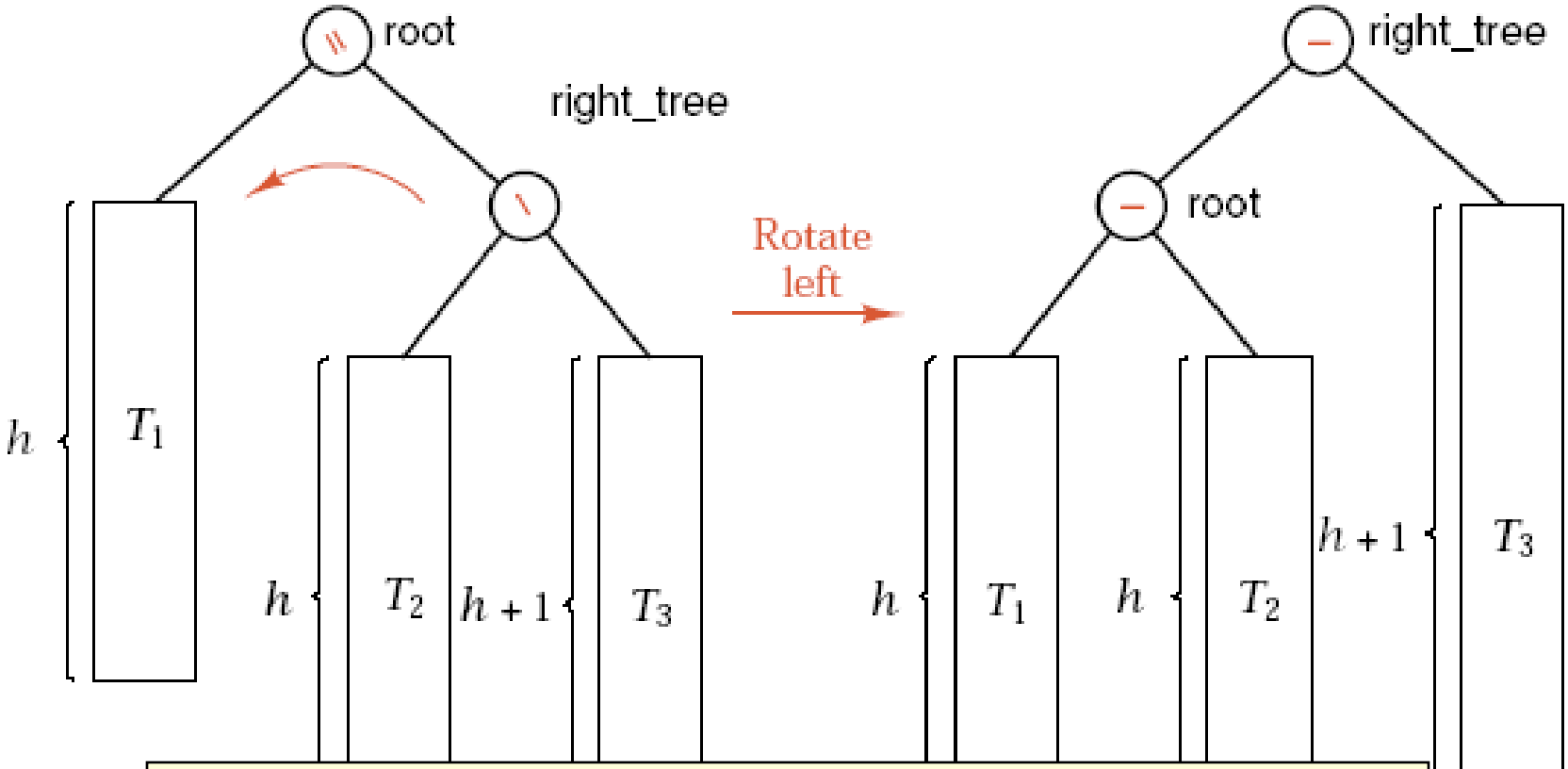


Chiều cao cây không đổi



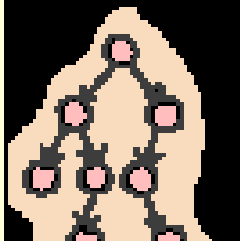
Mất cân bằng bên phải

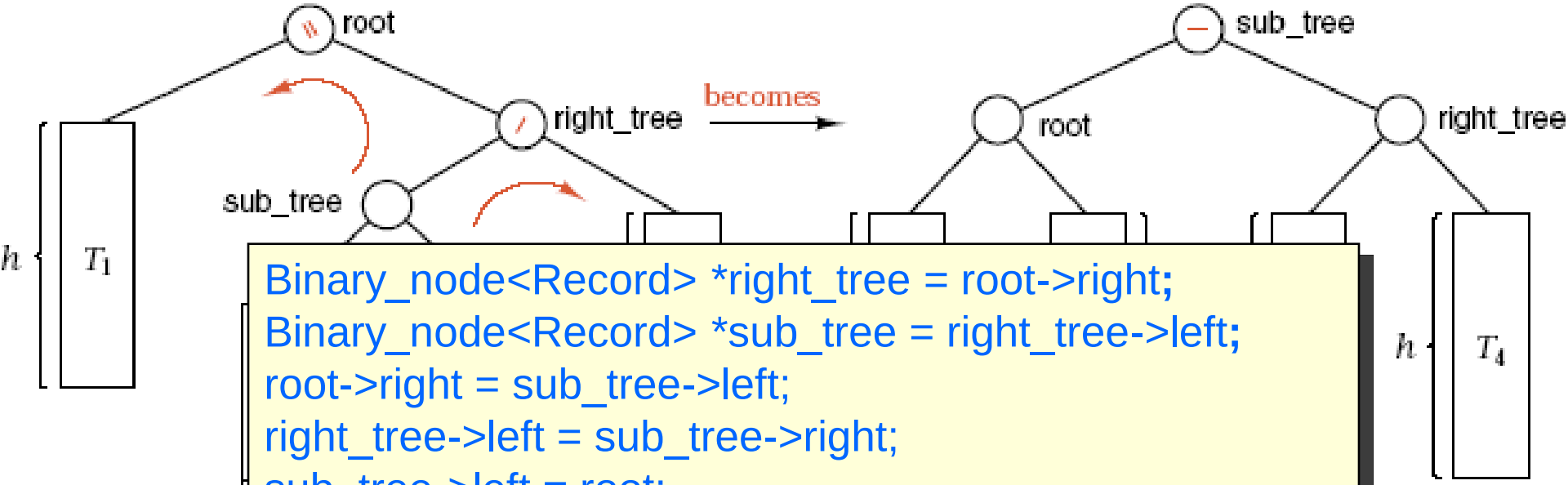
Cân bằng cây AVL – Quay đơn




```
Binary_node<Record> *right_tree = root->right;
root->right = right_tree->left;
right_tree->left = root;
root = right_tree;
```

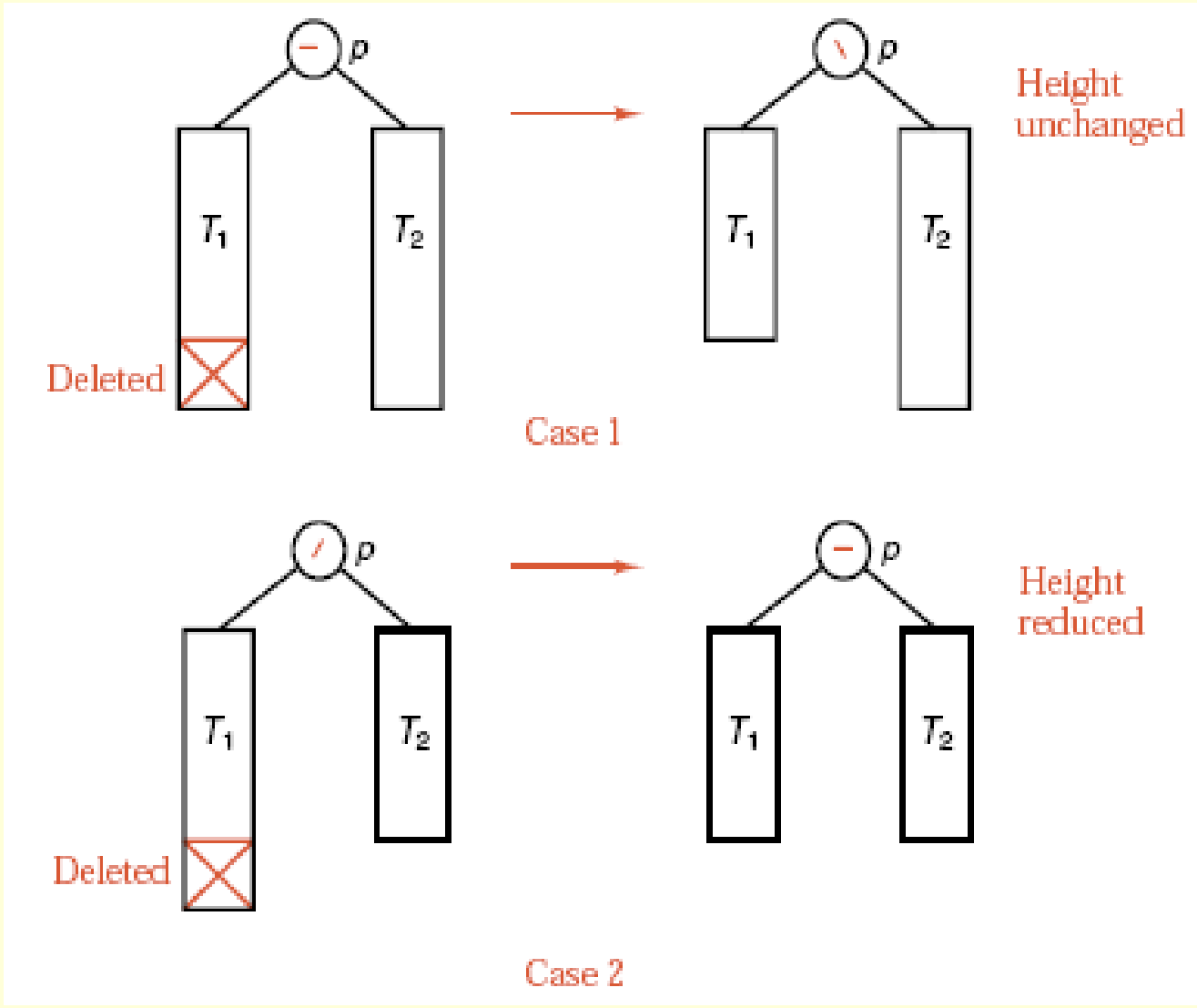
Cân bằng cây AVL – Quay kép



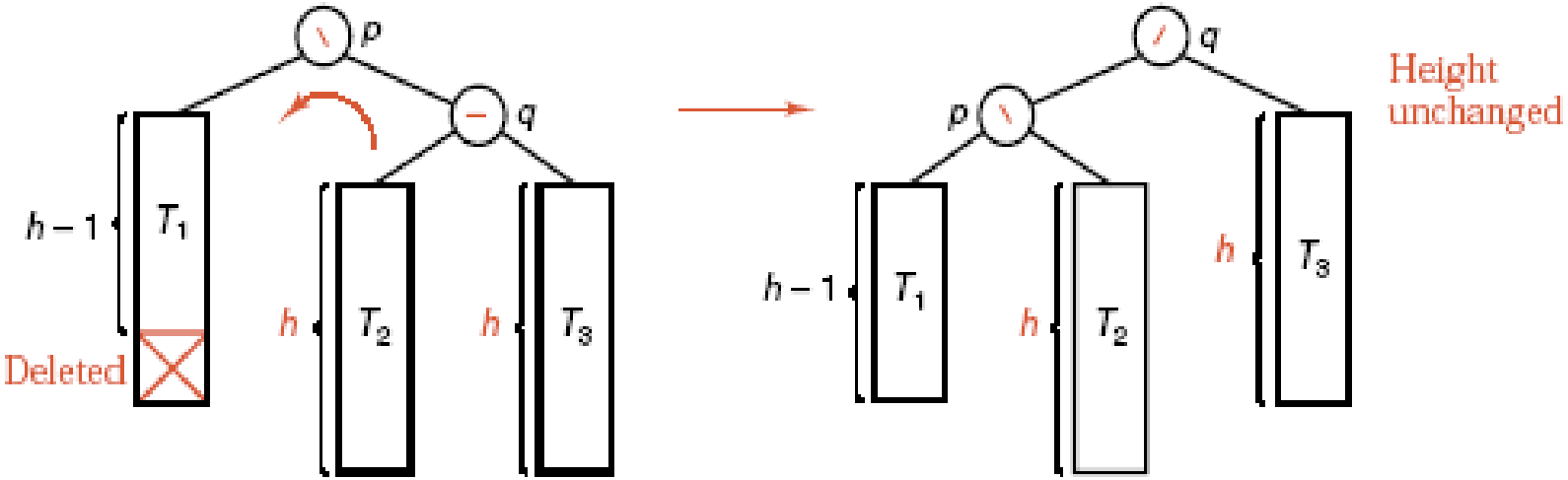



```
Binary_node<Record> *right_tree = root->right;
Binary_node<Record> *sub_tree = right_tree->left;
root->right = sub_tree->left;
right_tree->left = sub_tree->right;
sub_tree->left = root;
sub_tree->right = right_tree;
root = sub_tree;

Hoặc:

rotate_right(right_tree);
rotate_left(root);
```

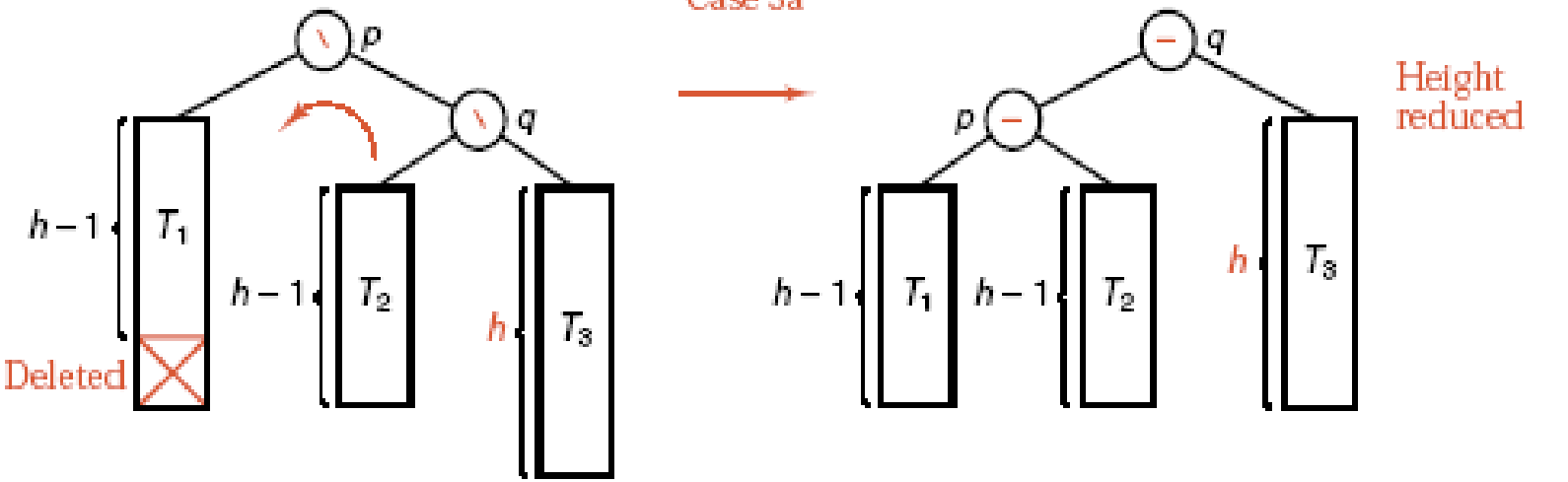
Các trạng thái khi xóa node



Các trạng thái khi xóa node (tt.)

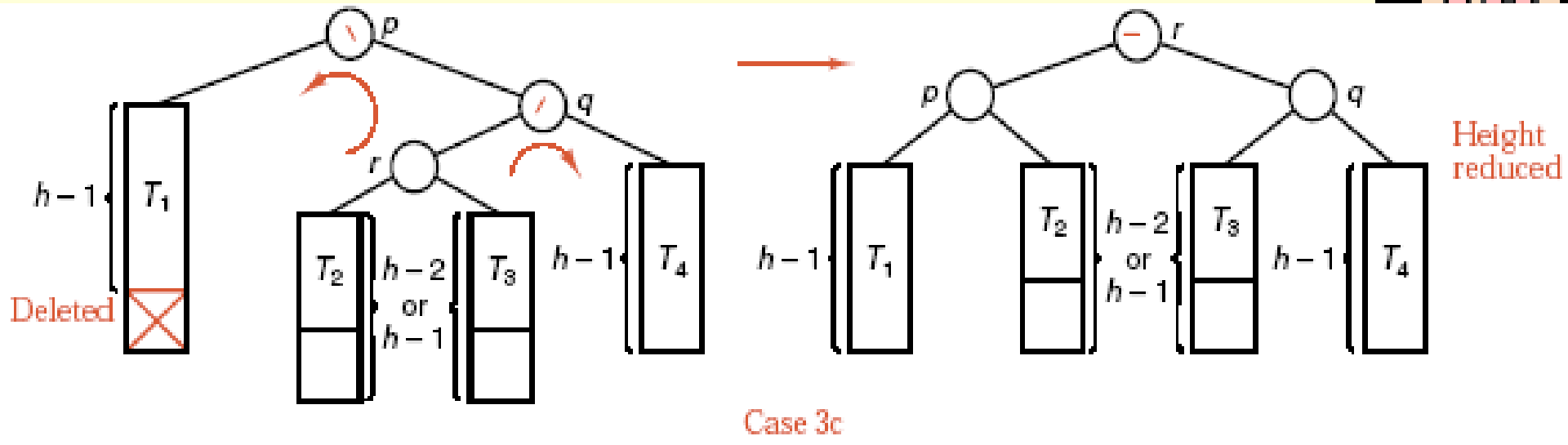


Case 3a

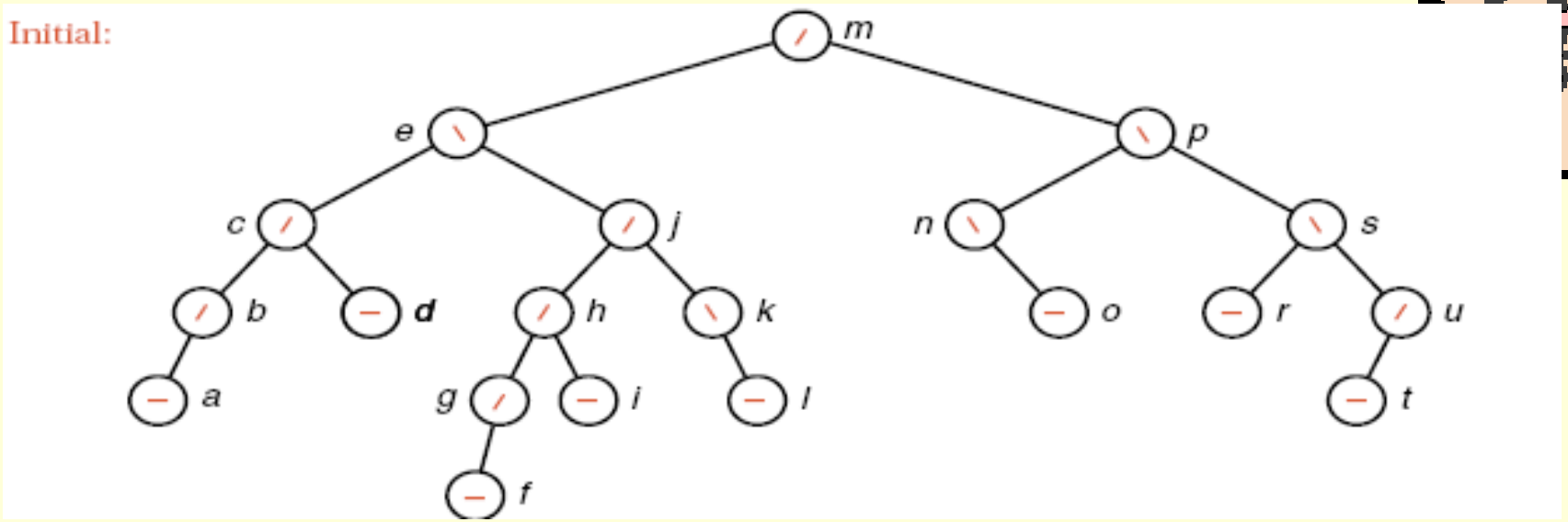


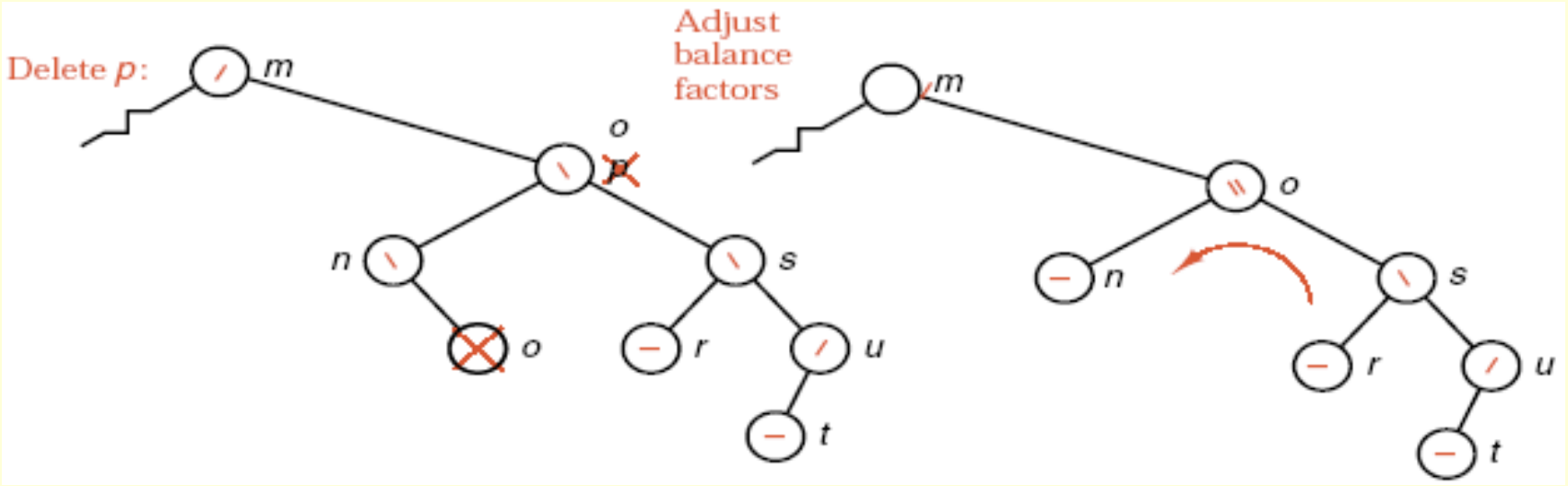
Case 3b

Các trạng thái khi xóa node (tt.)

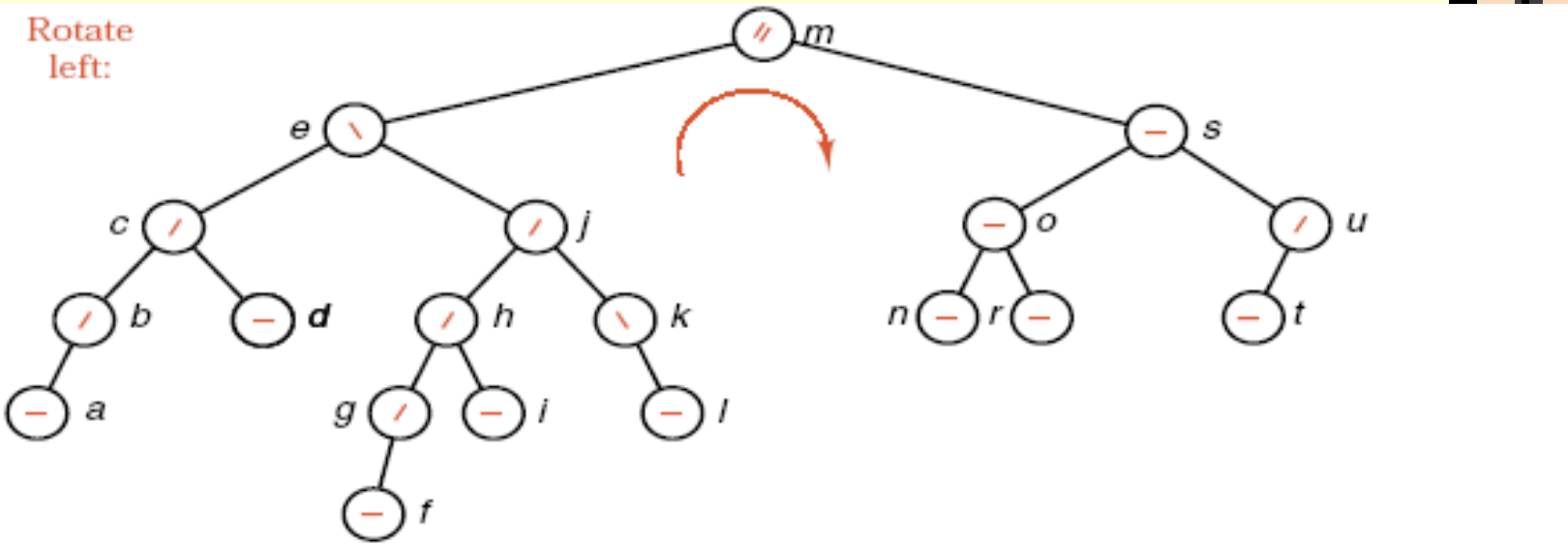


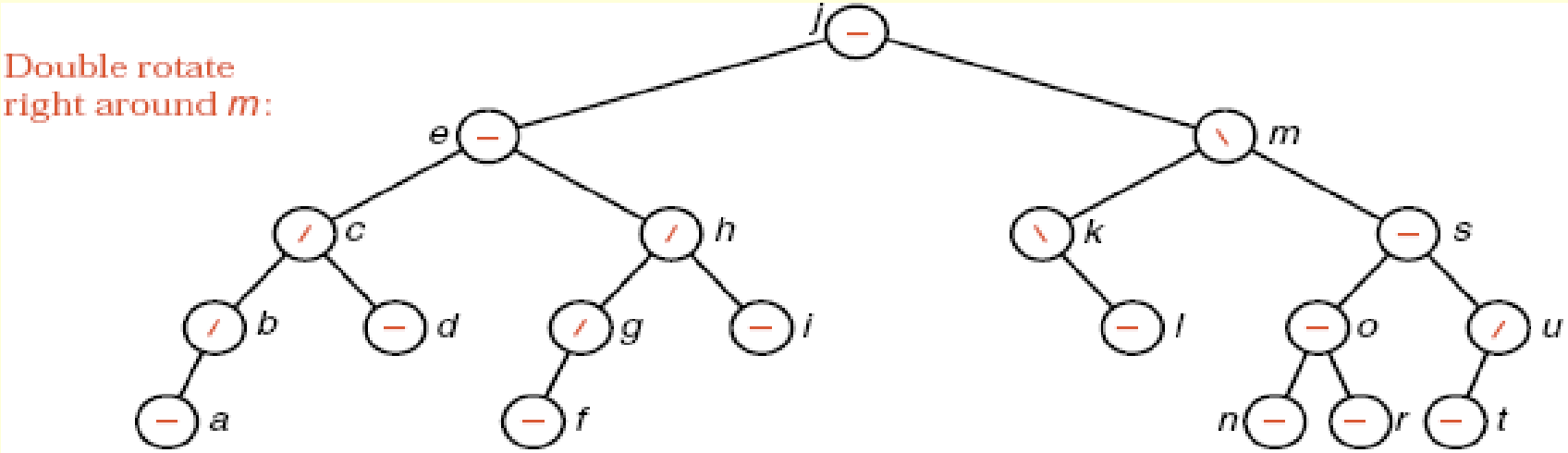
Ví dụ xóa node của cây AVL





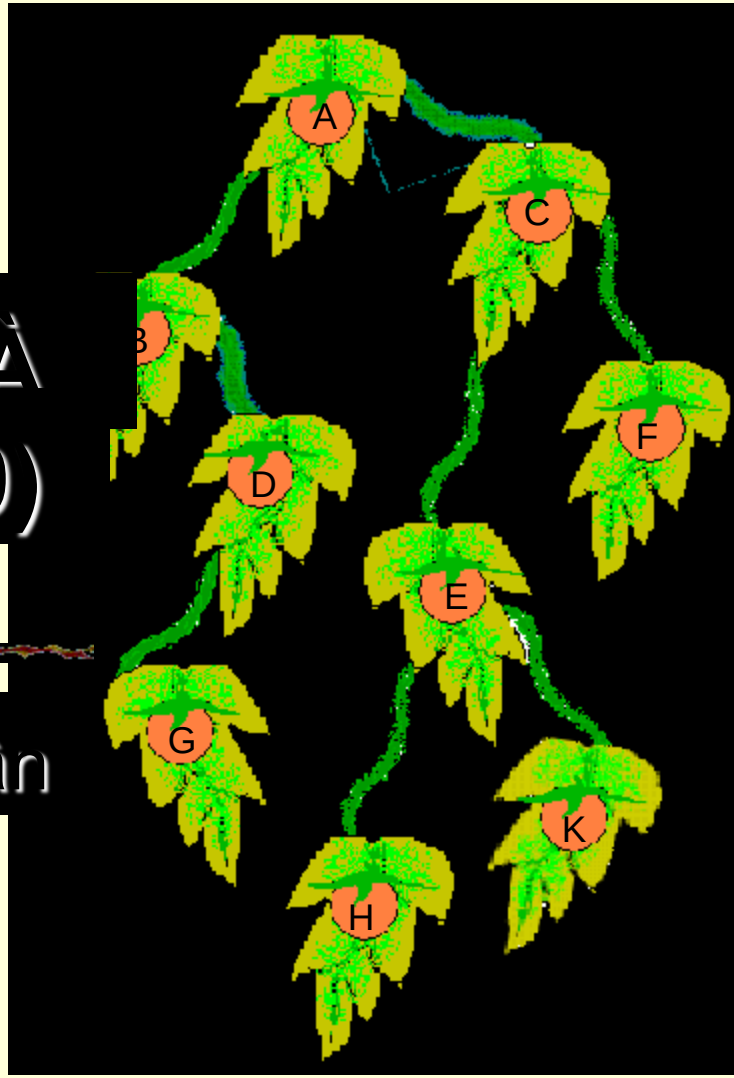
Ví dụ xóa node của cây AVL (tt.)



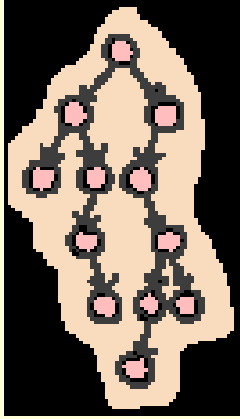


CẤU TRÚC DỮ LIỆU VÀ GIẢI THUẬT (501040)

Chương 11: Cây đa phân

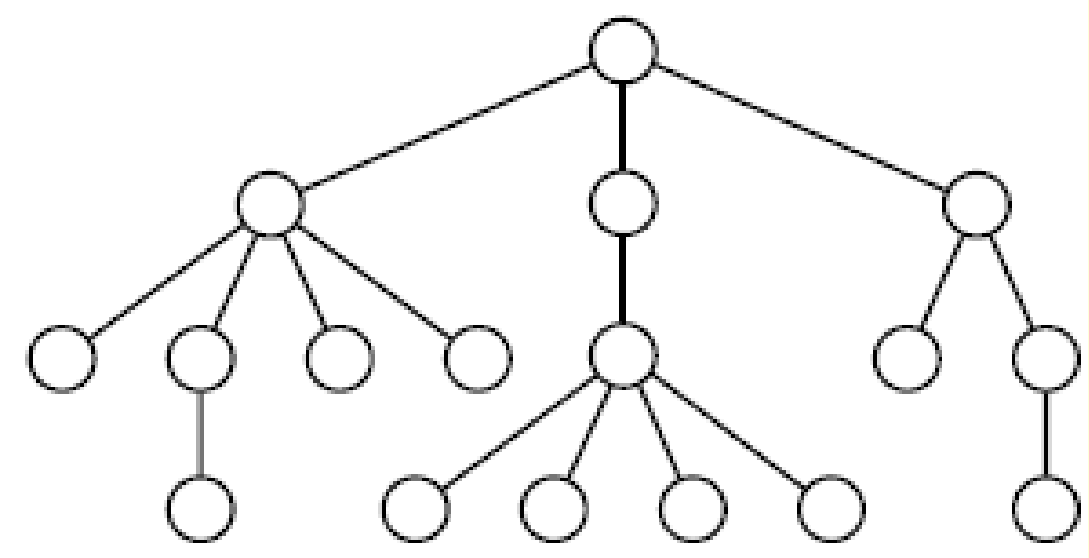


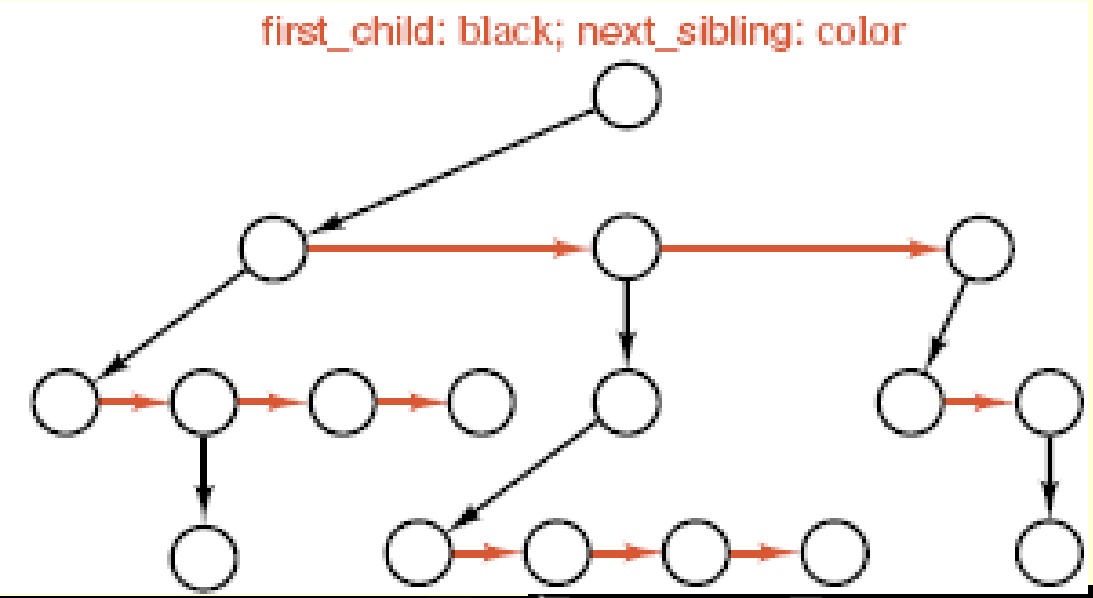
Định nghĩa



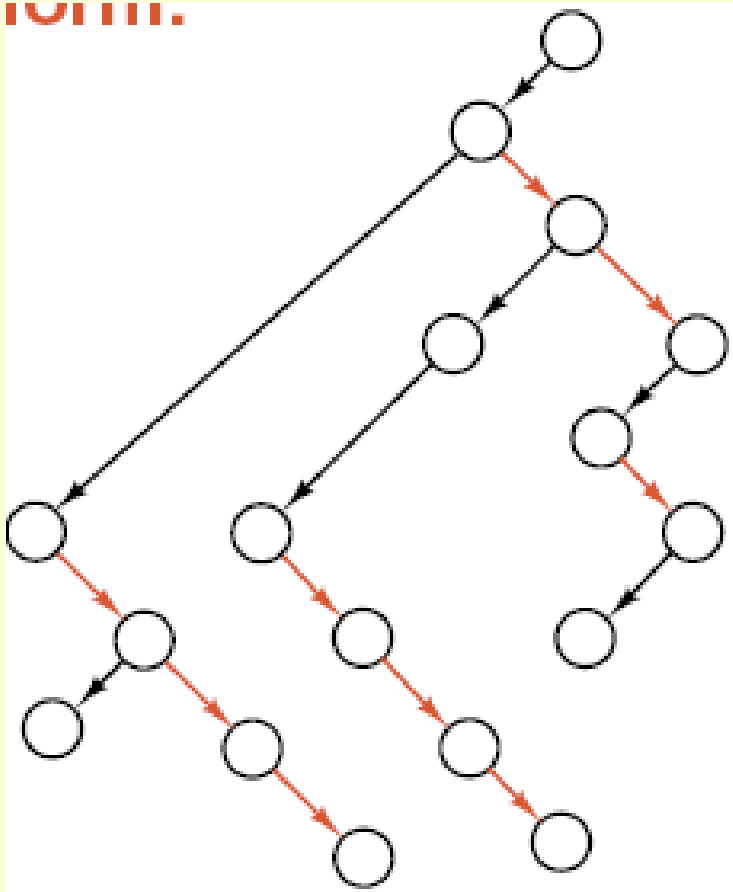
- Cây đa phân
  - Cây rỗng
  - Hoặc có một node gọi là gốc (root) và nhiều cây con.
- Biểu diễn:
  - Mỗi node gồm có nhiều nhánh con
  - Mỗi node có 2 liên kết first_child và next_sibling
  - Dùng cây nhị phân

Biểu diễn





Biểu diễn dạng nhị phân

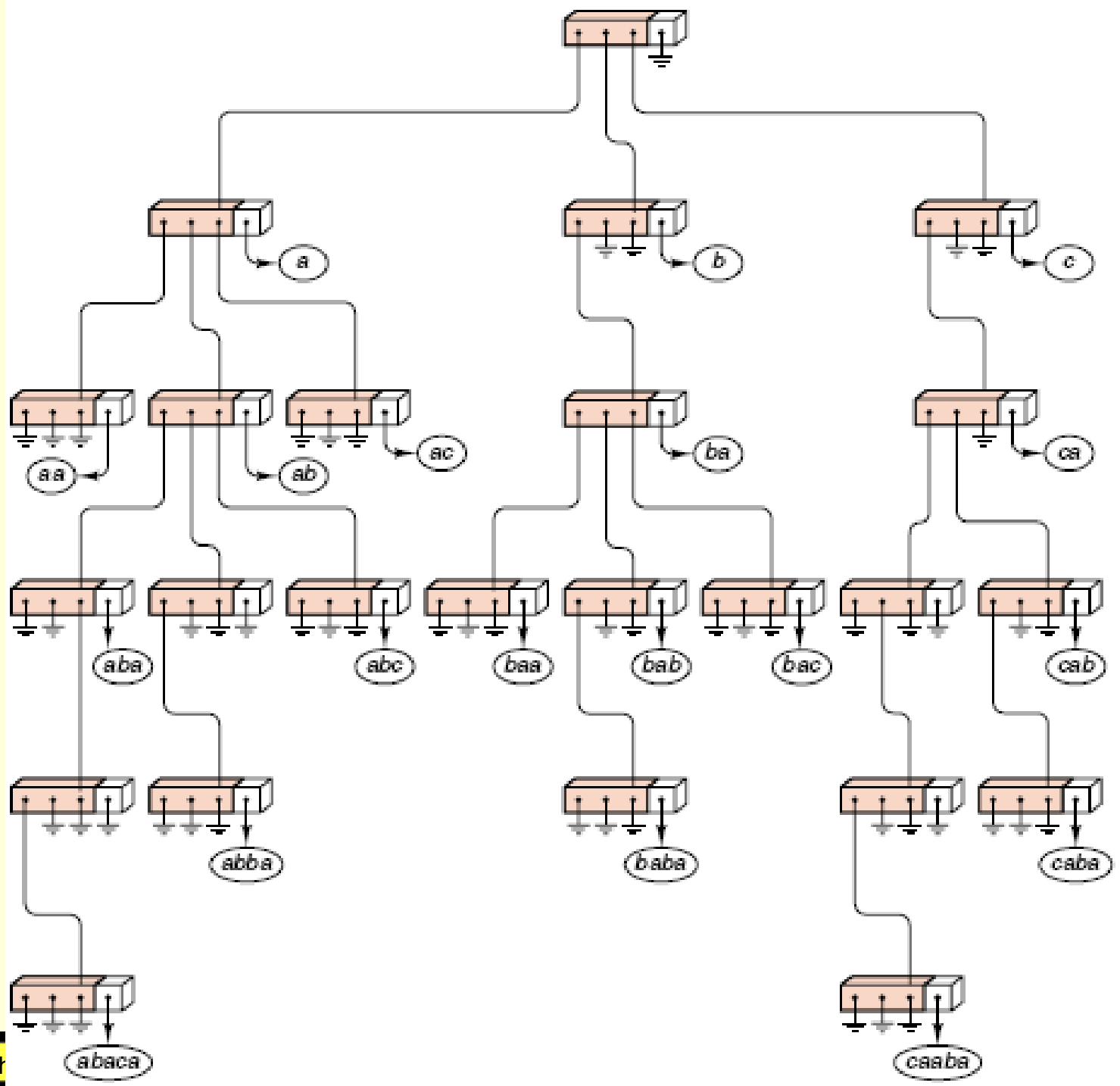


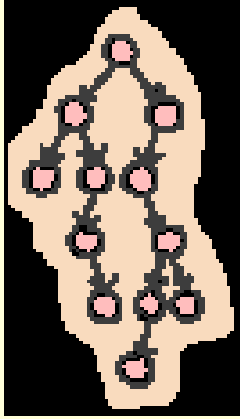
first_child (left) links: black
next_sibling (right) links: color

Nhị phân:

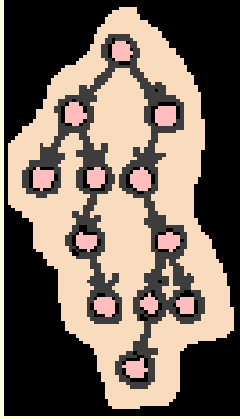
left = black

right = color


a
b
c
aa
ab
ac
ba
ca
aba
abc
baa
bab
bac
cab
abba
baba
caba
abaca
caaba



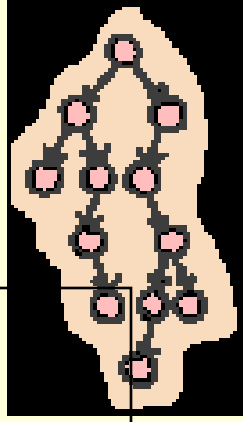
Thiết kế Trie



```
class Trie {
public: // Add method prototypes here.
private: // data members
    Trie_node *root;
};
```

```
const int num chars = 28;

struct Trie_node {
    // data members
    Record *data;
    Trie_node *branch[num_chars];
    // constructors
    Trie_node( );
};
```

Giải thuật tìm kiếm trên Trie

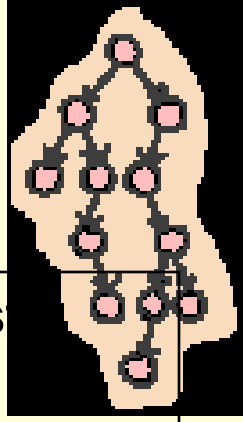


Algorithm trie_search
Input: target là khóa cần tìm
Output: mẫu tin tìm thấy

1. Bắt đầu tìm từ node root với ký tự đầu tiên của target
2. **while** (còn node để tìm và chưa xét hết chuỗi target)
   2.1. Nhảy đến node con tương ứng tùy theo ký tự từ target
   2.2. Xét ký tự kế tiếp trong chuỗi target
3. **if** (có node và dữ liệu của nó khác rỗng)
   3.1. **return** dữ liệu của node này
4. **return** not_present

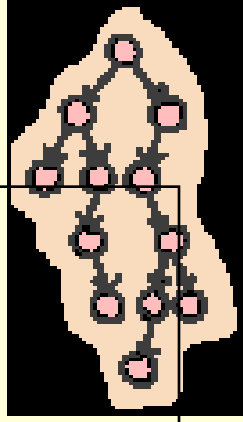
End trie_search

Mã C++ tìm kiếm trên Trie



```
Error_code Trie :: trie_search(const Key &target, Record &x) cons
    int position = 0;
    char next_char;
    Trie_node *location = root;
    while (location != NULL &&
                        (next_char = target.key_letter(position)) !=‘ ’) {
        location = location->branch[alphabetic order(next char)];
        position++;
    }
    if (location != NULL && location->data != NULL) {
        x = *(location->data);
        return success;
    }
    else
        return not present;
}
```

Giải thuật thêm vào Trie



Algorithm trie_insert

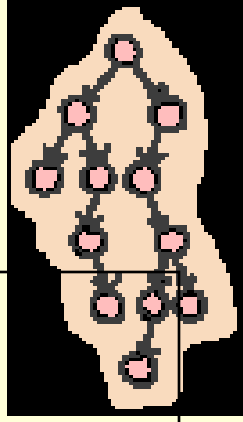
Input: new_entry là dữ liệu cần thêm vào

Output: cây sau khi thêm vào dữ liệu mới

1. **if** (cây rỗng)
   1.1. Thêm node mới vào đây
   1.2. Kết thúc
2. Bắt đầu từ node root và ký tự đầu tiên trong khóa của new_entry
3. **while** (vẫn chưa xét hết chuỗi của khóa của new_entry)
   3.1. next_char là ký tự hiện tại trên khóa
   3.2. **if** (node con tại vị trí next_char không có)
      //Tìm và thêm các node trung gian không có dữ liệu vào
      3.2.1. Thêm một node có dữ liệu rỗng vào đây
   3.3. Nhảy đến node con tương ứng với vị trí của next_char
   3.4. Xét ký tự kế tiếp của khóa
4. Thêm dữ liệu vào node hiện tại

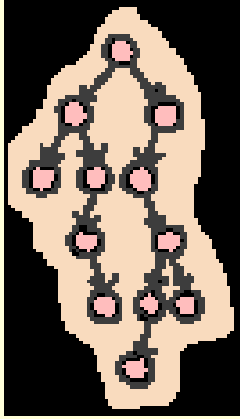
End trie_insert

Mã C++ thêm vào Trie



```
Error_code Trie :: insert(const Record &new_entry) {
    Error_code result = success;
    if (root == NULL) root = new Trie_node;
    int position = 0;
    char next_char;
    Trie_node *location = root;
    while ((next char = new entry.key letter(position)) != ' ') {
        int next_position = alphabetic_order(next_char);
        if (location->branch[next_position] == NULL)
            location->branch[next_position] = new Trie_node;
        location = location->branch[next_position];
        position++;  }
    if (location->data != NULL) result = duplicate_error;
    else location->data = new Record(new_entry);
return result;
}
```

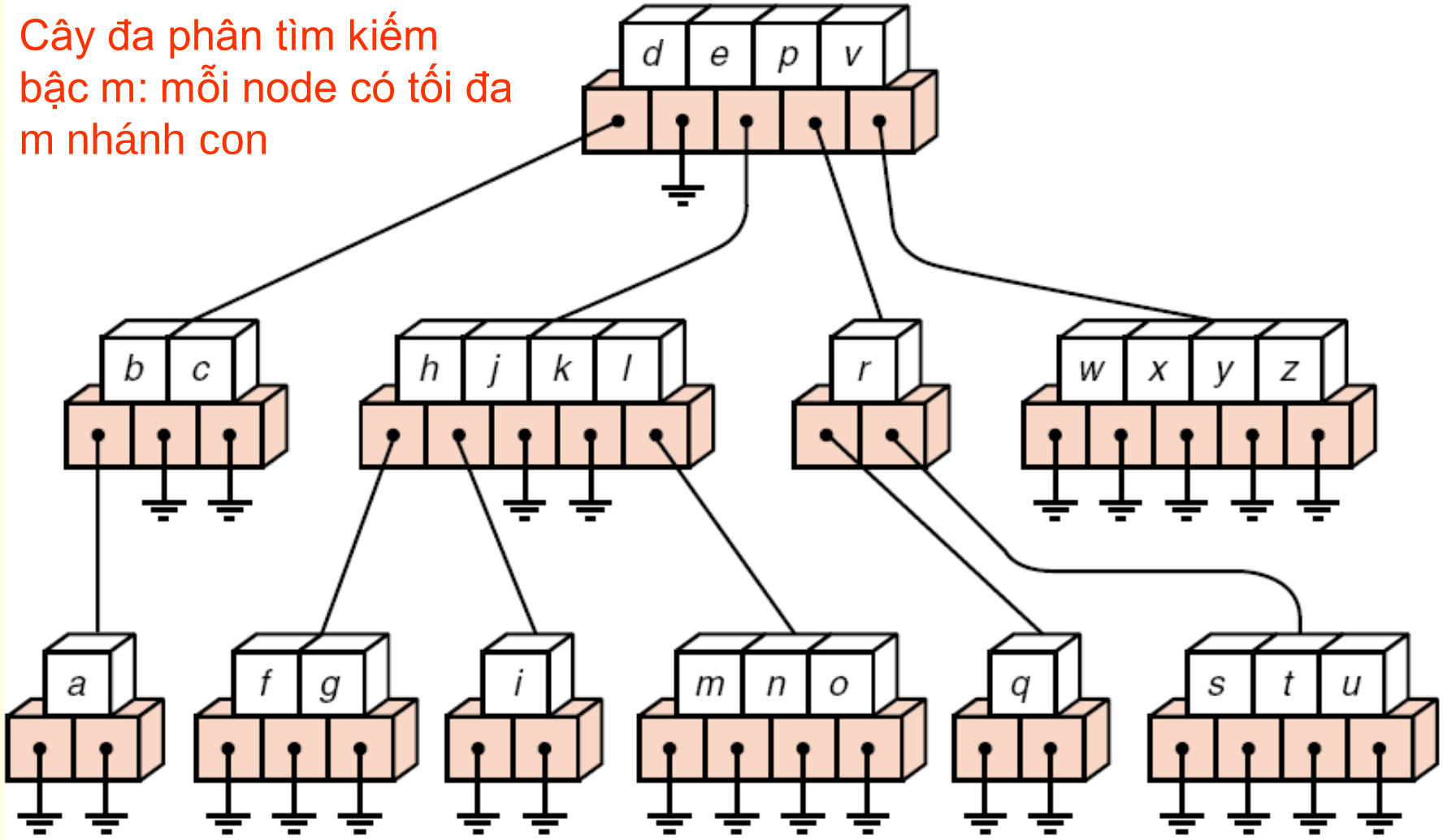
Đánh giá trie



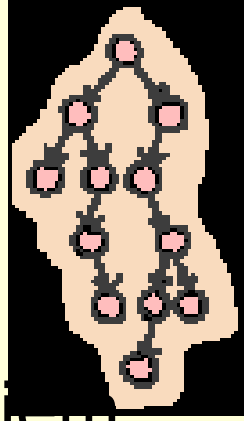
- Tìm kiếm: Lần so sánh = chiều dài khóa
- Từ điển tiếng Anh 100.000 từ, chiều dài tối đa 15 ký tự:
  - Trie: Số lần so sánh tối đa = 15
  - Tìm nhị phân = k*lg (100.000) = 17k (k: chiều dài trung bình của từ tiếng Anh)

Cây đa phân tìm kiếm

Cây đa phân tìm kiếm bậc m: mỗi node có tối đa m nhánh con

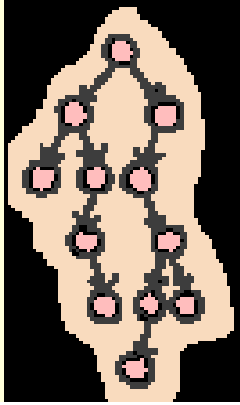


Cây đa phân cân bằng (B-tree)

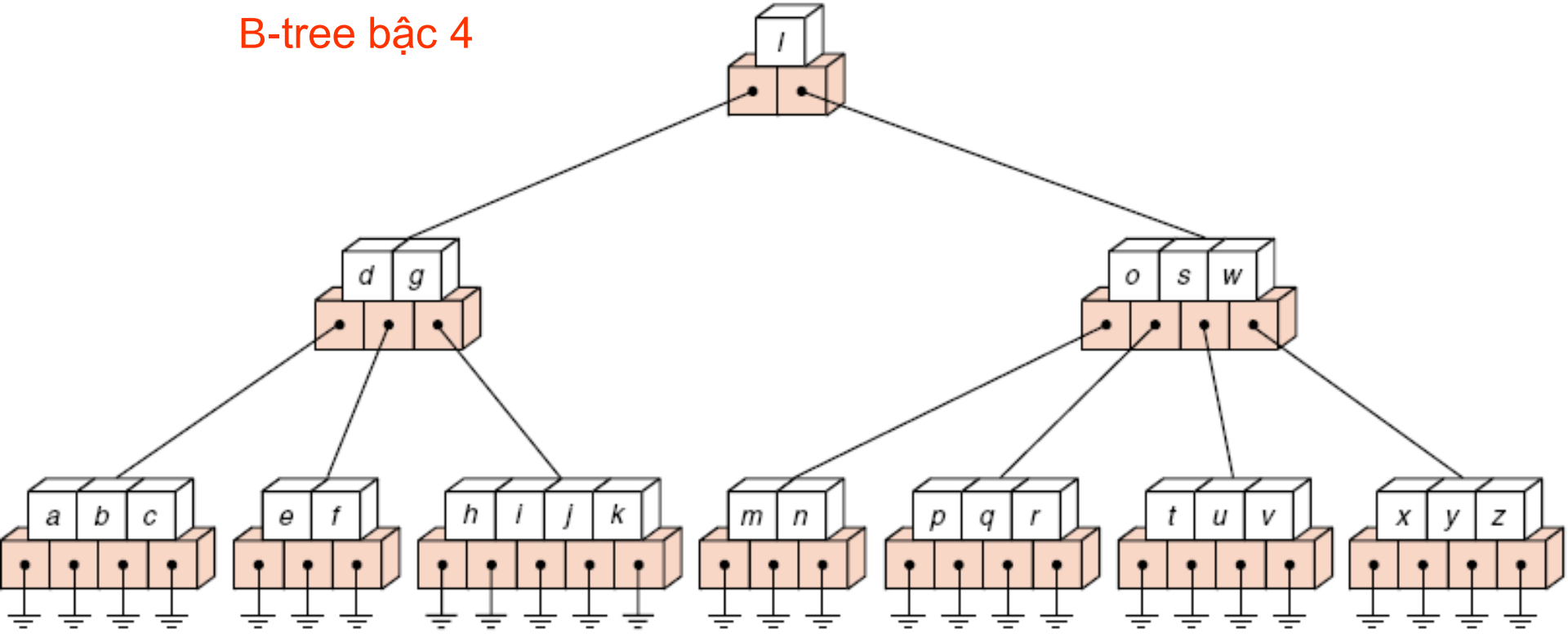


- Một *B-tree* bậc *m* là một cây đa phân tìm kiếm bậc m:
  - 1. Tất cả các node lá ở cùng một mức.
  - 2. Tất cả các node trung gian trừ node gốc có tối đa *m* nhánh con và tối thiểu *m/2* nhánh con (khác rỗng).
  - 3. Số khóa của mỗi node trung gian ít hơn một so với số nhánh con và phân chia các khóa trong các nhánh con theo cách của cây tìm kiếm.
  - 4. Node gốc có tối đa *m* nhánh con, tối thiểu là 2 nhánh con khi node gốc *không là node lá* hoặc không có nhánh con khi cây chỉ có node gốc.

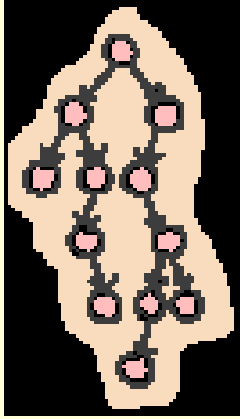
Ví dụ B-tree



B-tree bậc 4



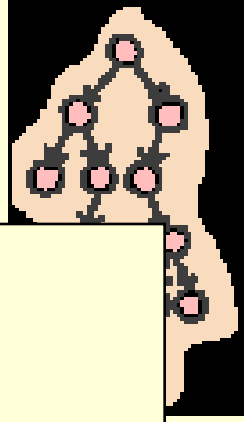
Thiết kế B-tree



```
template <class Record, int order>
class B_tree {
public: // Add public methods.
private: // data members
    B_node<Record, order> *root;
    // Add private auxiliary functions here.
};
```

```
template <class Record, int order>
struct B_node {
    // data members:
    int count;
    Record data[order − 1];
    B_node<Record, order> *branch[order];
    // constructor:
    B_node( );
};
```

Giải thuật tìm kiếm trên B-tree

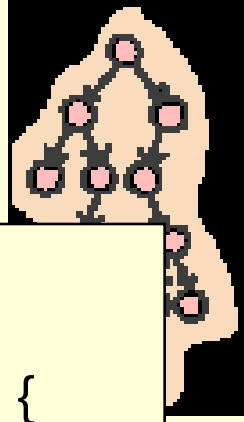


Algorithm search_B_tree
Input: subroot là gốc của cây và target là khóa cần tìm
Output: dữ liệu tìm thấy

1. **if** (cây rỗng)
   1.1. **return** not_present
2. **else**
   2.1. Tìm target trên dữ liệu của subroot
   2.2. **if** (tìm thấy)
      2.2.1. **return** dữ liệu tìm thấy
   2.3. **else**
      //Tìm không thấy sẽ ngừng tại vị trí có khóa vừa lớn hơn khóa cần
      //tìm, ở đó có liên kết đến nhánh con gồm các khóa nhỏ hơn nó.
      2.3.1. Nhảy đến nhánh con của vị trí không tìm thấy
      2.3.1. **Call** search_B_tree với nhánh con mới

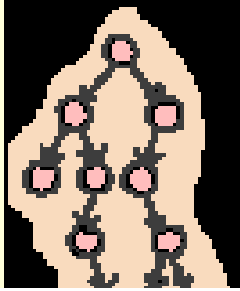
End search_B_tree

Mã C++ tìm kiếm trên B-tree

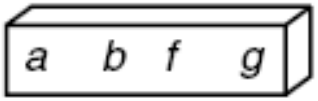


```
template <class Record, int order>
Error_code B_tree<Record, order> :: recursive_search_tree
                        (B_node<Record, order> *current, Record &target) {
    Error_code result = not_present;
    int position = 0;
    if (current != NULL) {
        while (position < current->count && target > current->data[position])
            position++;
        if (position < current->count && target == current->data[position])
            result = success;
        if (result == not_present)
            result =recursive_search_tree(current->branch[position], target);
        else
            target = current->data[position];
    }
    return result;
}
```

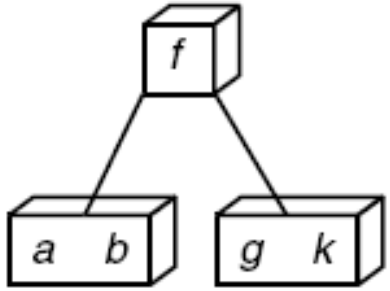
Thêm vào B-tree



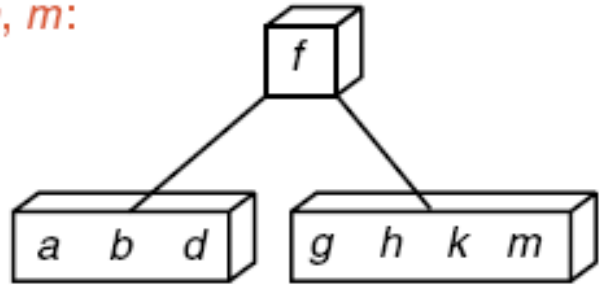
1. *a*, *g*, *f*, *b*:



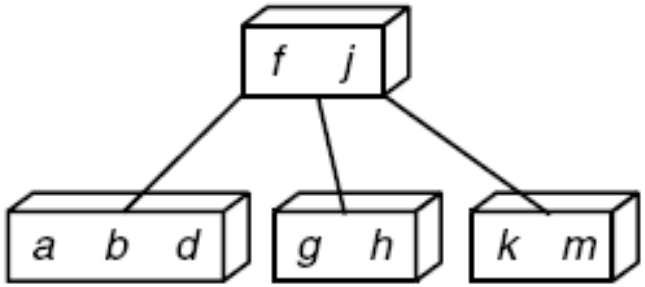
2. *k*:



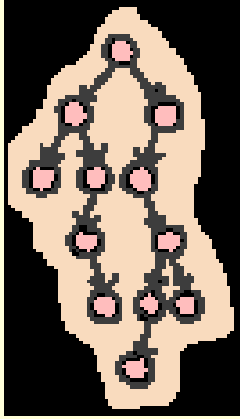
3. *d*, *h*, *m*:



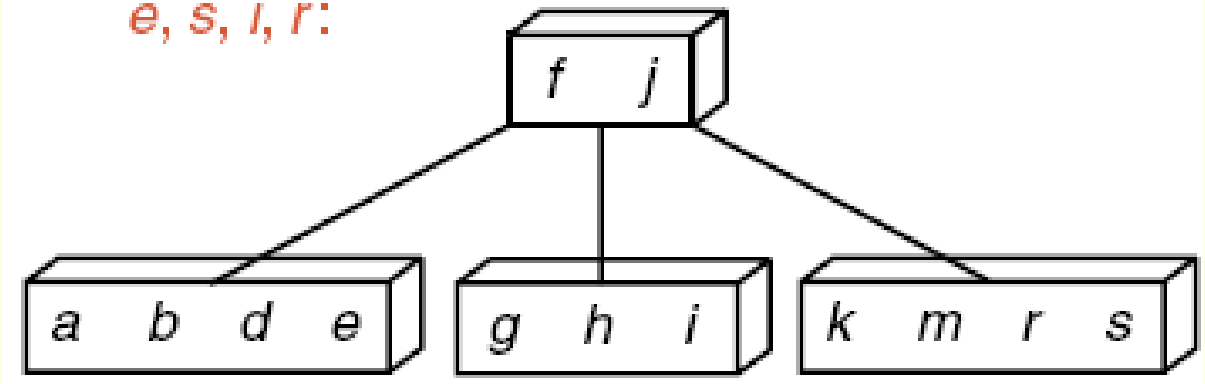
4. *j*:



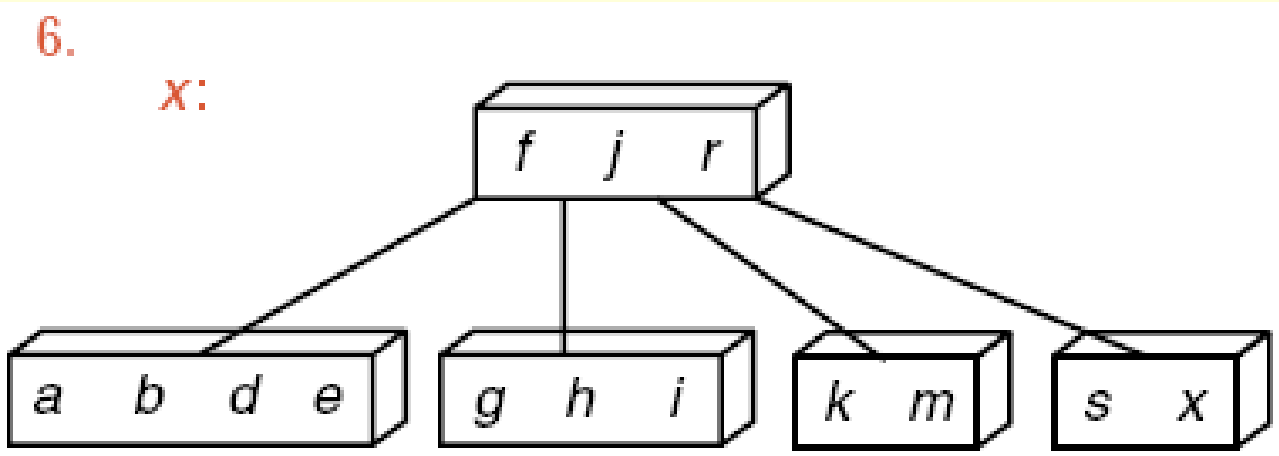
Thêm vào B-tree (tt.)



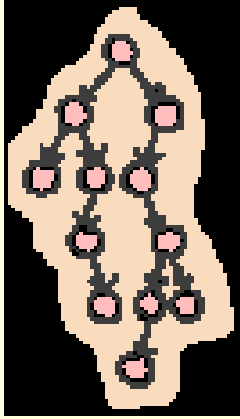
5. *e, s, i, r*:



6. *x*:

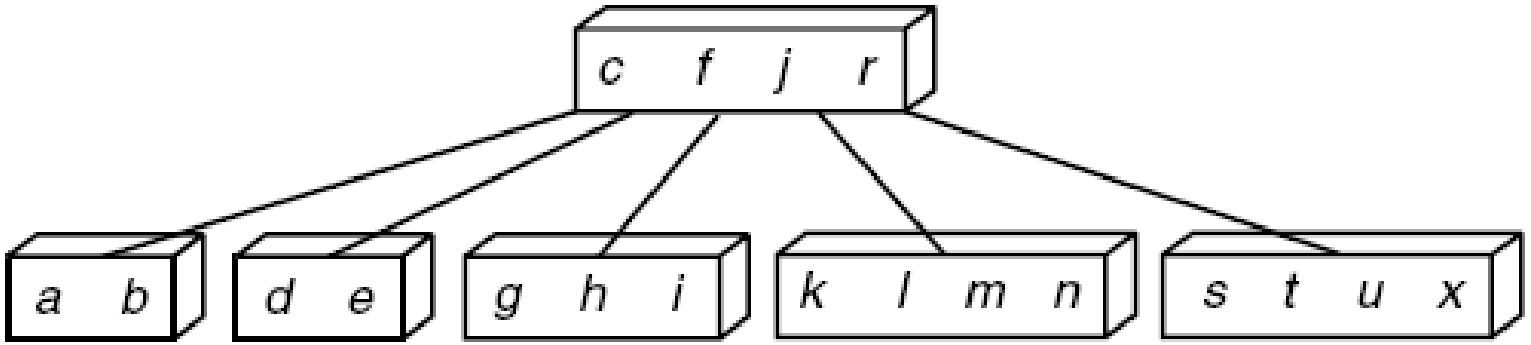


Thêm vào B-tree (tt.)

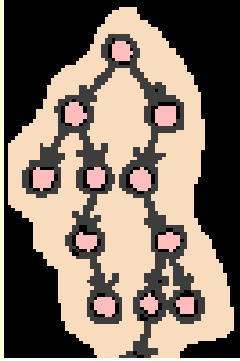


7.

c, l, n, t, u:

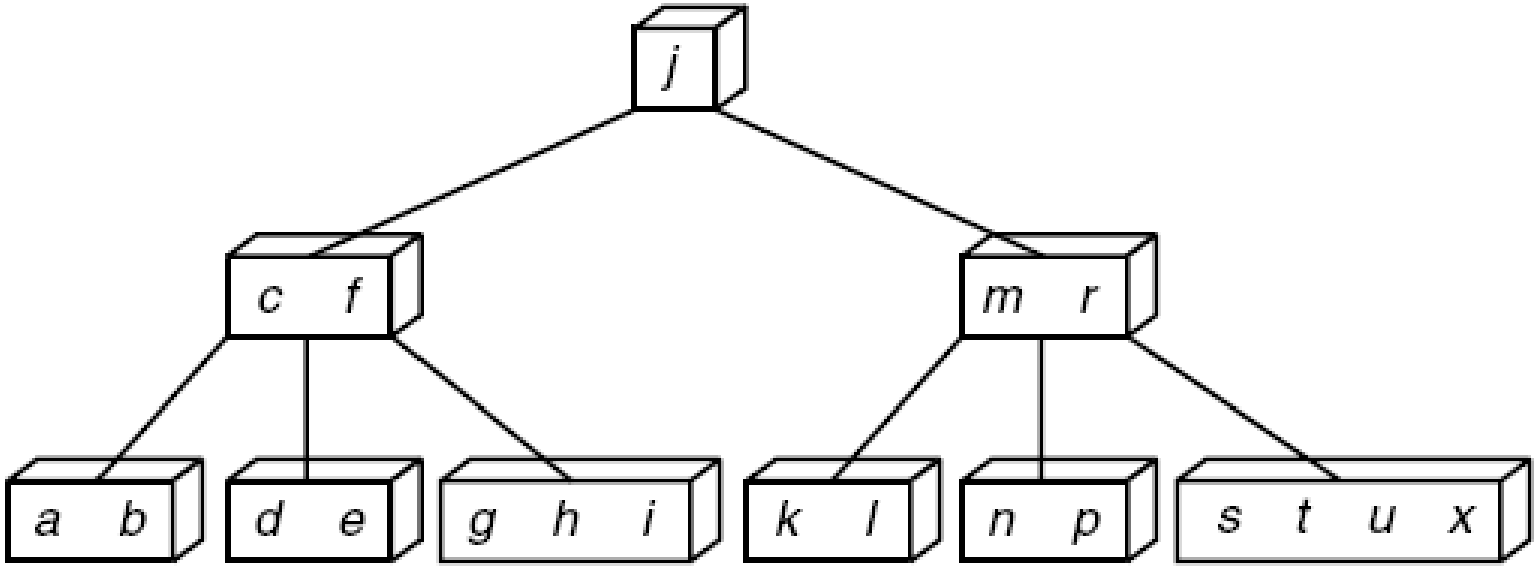


Thêm vào B-tree (tt.)

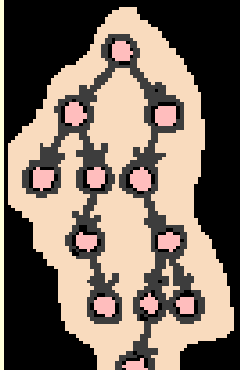


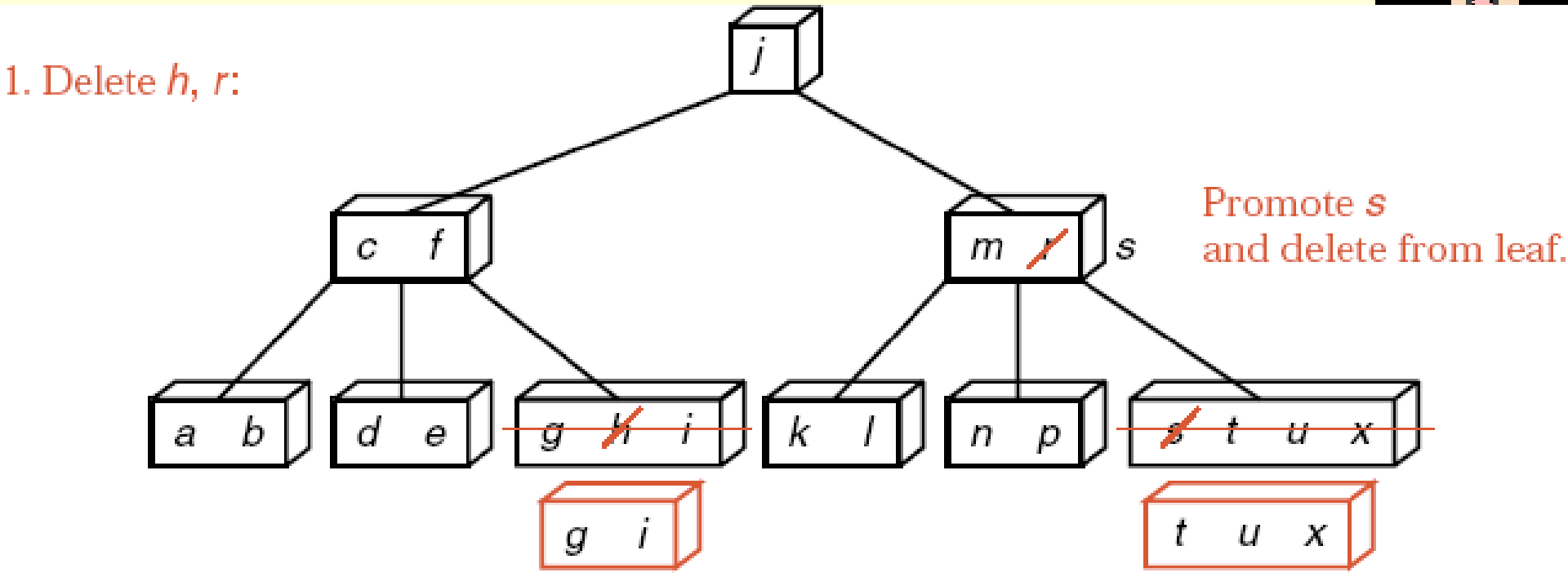
8.

p:

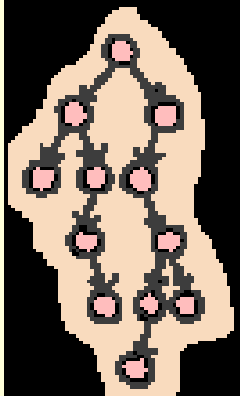


Xóa giá trị trên B-tree

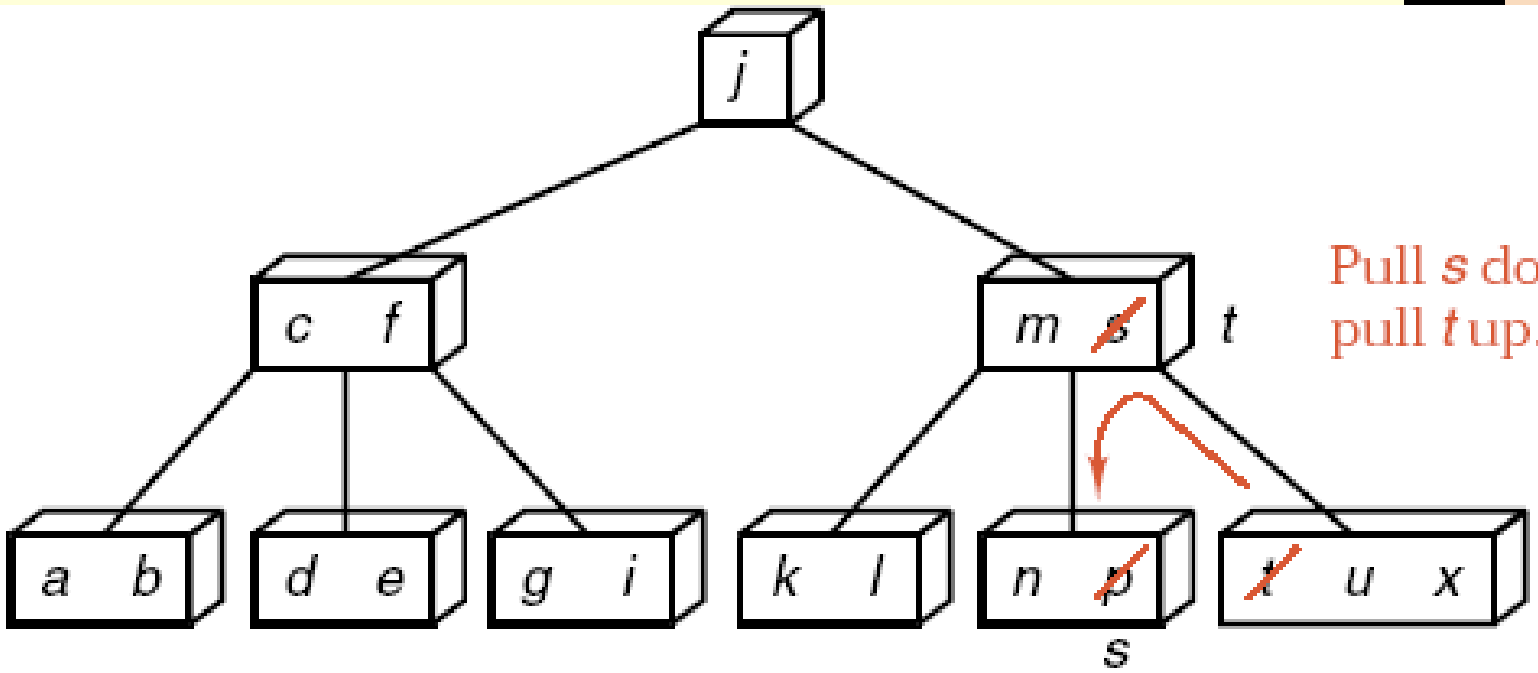




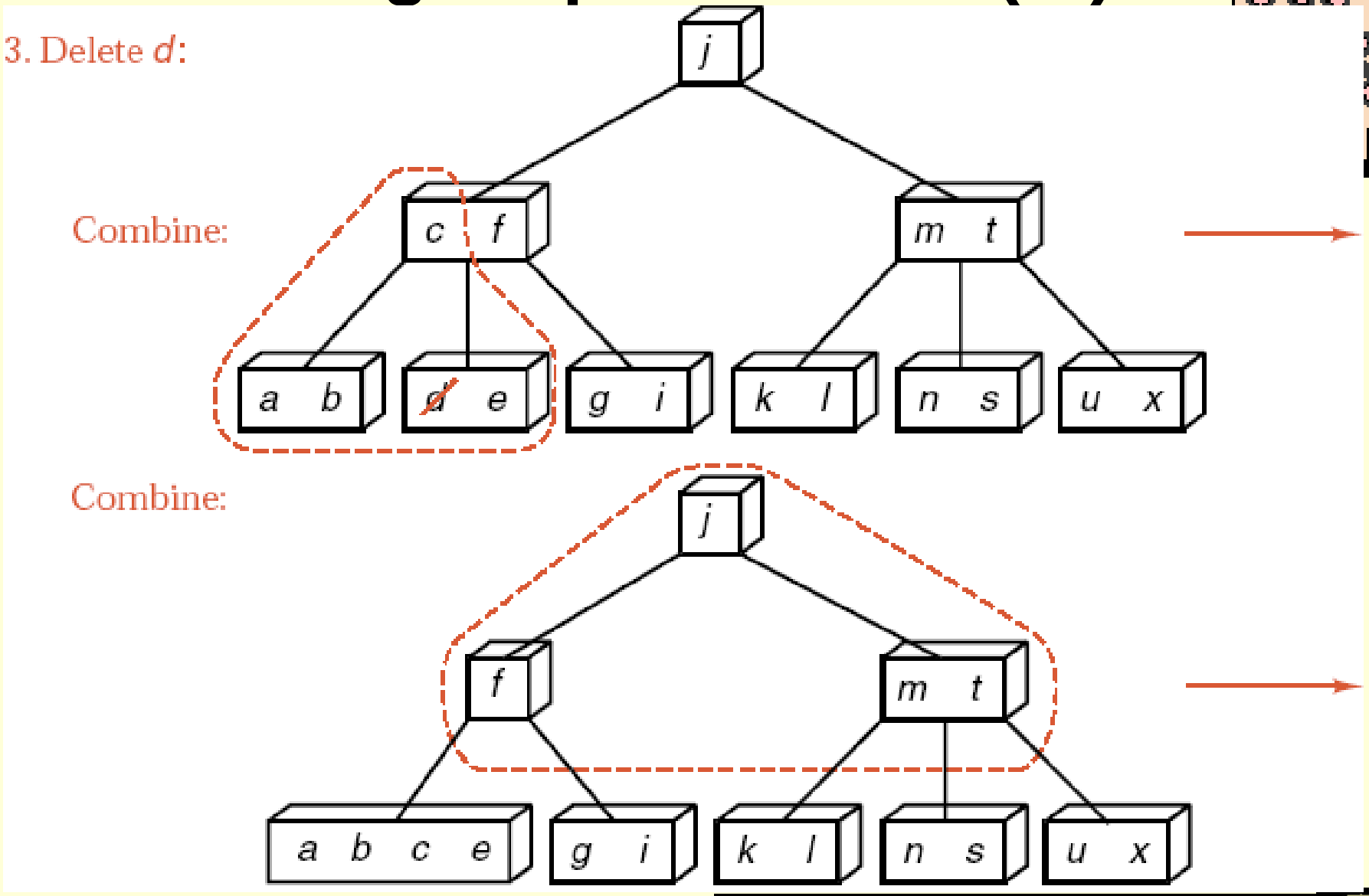
Xóa giá trị trên B-tree (tt.)



2. Delete *p*:



Xóa giá trị trên B-tree (tt.)



Xóa giá trị trên B-tree (tt.)