**Chương 2 : Cấu trúc dữ liệu nâng cao**

**\*) Tập hợp (set)**

Khái niệm

◦ Các phần tử cùng kiểu dữ liệu

◦ Không trùng nhau

 Ví dụ

◦ Mảng a = {1, 2, 1, 1, 2}

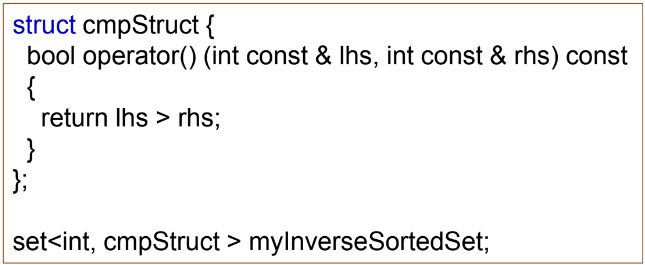
◦ Tập hợp s = {1, 2}

 Thứ tự các phần tử

◦ Mặc định: thứ tự tăng dần

◦ Thứ tự giảm dần: set<int, std::greater> s;

◦ Hoặc theo định nghĩa:



 Cài đặt: kiểm tra trùng khi chèn

◦ Lớp set

 Các phần tử được sắp thứ tự

 Sử dụng cây đỏ đen

 Thời gian truy cập một phần tử: O(log(n))

◦ Lớp unordered\_set

 Các phần tử không được sắp

 Được cài đặt bằng bảng băm

 Thời gian truy cập O(1)

 Cài đặt:

◦ Một số phương thức: insert, erase, swap, clear, size, find

◦ Duyệt phần tử

set<int>::iterator it;

for (it = s.begin(); it != s.end(); ++it)

cout << ' ' << \*it;

 Ứng dụng

◦ Bài toán liên quan tới tập hợp

◦ Ví dụ 1: Cho dãy n phần tử. Hãy đếm số phần tử khác nhau trong mảng.

set<int> s;

for (int i=0; i < n; i++)

s.insert(a[i]);

return s.size();

◦ Ví dụ 2: Một vector v được gọi là vector beautiful nếu một số trong v chỉ xuất hiện đúng một lần. Cần xoá ít nhất bao nhiêu phần tử trong v để v trở thành vector beautiful

+ Với v = [2, 3, 6, 3] thì vectorBeautiful(v) = 1

* chỉ cần xóa số 3.

+ Với v = [1, 2, 3] thì vectorBeautiful(v) = 0

Sử dụng set s, chèn tất cả phần tử trong v vào s.

Kết quả chính là v.size() - s.size().

Ví dụ 3: Viết chương trình liệt kê các số nguyên tố trong phạm vi từ 1 đến n theo phương pháp Eratosthene (không cần đến các phép nhân).

Thuật toán: xuất phát từ một tập các số nguyên từ 1 đến n. Mỗi lần gặp một số nguyên tố, ta sẽ loại trừ ra khỏi tập này tất cả các số là bội của số nguyên tố này.

**\*) Ánh xạ (MAP**)  
 Khái niệm

◦ Dữ liệu có kiểu cấu trúc liên kết

◦ Là sự kết hợp của khóa (key value) và ánh xạ của nó (mapped value)

◦ Giá trị các khóa

 Duy nhất (không trùng)

 Map

 Được sắp thứ tự tăng dần

 Unordered\_map

 Không được sắp

◦ Chú ý: trong Python, C# thì map là dictionary

 Cài đặt

◦ Trong C++

 map

 Dựa trên cây đỏ đen

 Khai báo: map<type 1, type 2> tien\_bien;

 Lấy giá trị khóa (key): tien\_bien.first

 Lấy giá trị (value): ten\_bien.second

 unordered\_map

 Dựa trên bảng băm (hash table)

 Khai báo: unordered\_map<type 1, type 2> map\_name;

◦Thay đổi thứ tự sắp xếp các phần tử của map

struct cmp {

bool operator() (char a, char b) {

return a > b;

}

};

.....

map <char, int, cmp> m;

◦ Các hàm cơ bản

 value = at(k): trả lại giá trị ứng với khóa k

 size(): số phần tử hiện tại

 max\_size(): số phần tử mà map có thể chứa

 empty(): kiểm tra map có rỗng không

 erase(const g): xóa khóa g khỏi map

 pair insert(keyvalue, mapvalue)

 clear(): xóa tất cả các phần tử

◦ Ví dụ

 Chèn dữ liệu

 map\_name. insert(pair<int, int>(160, 42));

 Xóa

 map\_name.erase(160);

 Duyệt

map<int, int>::iterator itr;

for (itr = map\_name.begin(); itr != map\_name.end(); ++itr) {

cout << itr.first << "t t" << itr.second << endl;

}

 Ứng dụng

◦ Tìm chủ thuê bao theo số điện thoại

◦ Tìm địa chỉ IP theo tên miền

◦ Tìm số lần vi phạm kỷ luật của một nhân viên

**\*)Hàng đợi(hai đầu, ưu tiên)  
- Hai đầu**

 Khái niệm

◦ Dữ liệu có thể thêm hoặc xóa ở cả đầu (front) và cuối (rear) của hàng đợi  sự kết hợp của cả stack và queue

 Cài đặt

◦ Cài đặt bằng mảng

◦ Cài đặt bằng danh sách móc nối kép

 Sử dụng lớp deque trong STL

◦ Khai báo: deque<type> dq\_data;

◦ Các phương thức:

 push\_front(const T& x)

 push\_back(const T& x)

 pop\_front()

 pop\_back()

 T& front()

 T& back()

 size\_type size()

 Ứng dụng

◦ Giải các bài toán trong đó các phần tử cần phải được thêm và loại bỏ ở hai đầu

◦ Lịch sử duyệt Web: Các trang Web được viếng thăm gần nhất được thêm vào một đầu và đầu kia loại bỏ các trang Web được viếng thăm cũnhất (khi vượt quá kích thước của hàng đợi)

◦ Lưu danh sách các thao tác Undo của phần mềm

* **Hàng đợi ưu tiên**

 Khái niệm

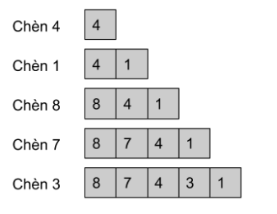
◦ Giống hàng đợi thông thường

◦ Thứ tự các phần tử phụ thuộc vào độ ưu tiên của phần

tử đó

◦ Ưu tiên nhất  đầu dãy

◦ Ví dụ: {4, 1, 8, 7, 3} với số càng lớn độ ưu tiên càng cao



 Cài đặt

◦ Sử dụng mảng thông thường

 Thao tác chèn phải kiểm tra độ ưu tiên  O(n)

 Theo tác xóa phải dồn mảng  O(n)

◦ Sử dụng mảng với cấu trúc đống (heap)

 Thao tác chèn, xóa  O(log(n))

◦ Sử dụng danh sách liên kết

 Thao tác chèn  O(n)

◦ Sử dụng thư viện C++ STL

 Khai báo: priority\_queue<type> pr\_data;

 Ứng dụng

◦ Cài đặt cấu trúc đống (heap)

◦ Thuật toán cây bao trùm tối thiểu Prim

◦ Thuật toán tìm đường đi ngắn nhất Dijkstra

◦ Sinh Huffman Code

◦ Cân bằng tải, xử lý ngắt trong hệ điều hành

◦ Đảm bảo chất lượng dịch vụ (mạng)

◦ Dùng để cài đặt thuật toán khác

**\*) Đống(heap) và ứng dụng**

 Khái niệm

◦ Cấu trúc dữ liệu dạng cây

◦ Tất cả các nút trong cây được sắp xếp theo một thứ tự nhất định, có thể là theo chiều tăng dần hoặc giảm dần

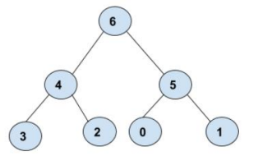
◦ Max heap:

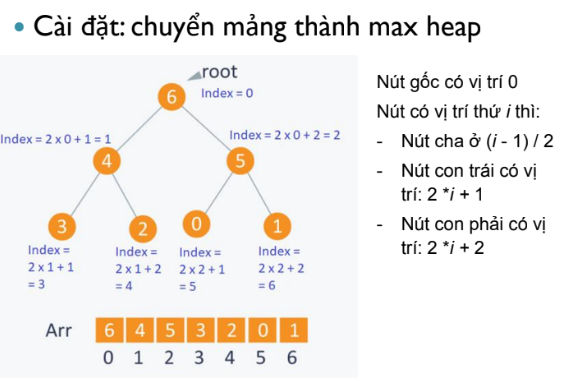
 Nút A là cha của nút B và giá trị của A lớn hơn B

 Áp dụng luật trên cho toàn bộ cây

◦ Binary Heap được ứng dụng phổ biến

 Một heap có 7 nút với giá trị: {6, 4, 5, 3, 2, 0, 1}





 Cài đặt: chuyển mảng thành max heap

◦ Bước 1: Lấy một phần tử từ mảng

◦ Bước 2: Kiểm tra xem con bên trái của nó có

phải max heap không

◦ Bước 3: Kiểm tra xem con bên phải của nó có

phải max heap không

◦ Bước 4: Kiểm tra xem chính node đó có phải

max heap không

 Cài đặt: chuyển mảng thành max heap

void heapify (int A[], int i) {

int largest; //Lưu chỉ số của phần tử lớn nhất

int left = 2\*i + 1; // Vị trí của con bên trái

int right = 2\*i +2; // Vị trí của con bên phải

if (left< N and A[left] > A[i]) // N là số phần tử trong mảng

largest = left;

else

largest = i;

if (right < N and A[right] > A[largest])

largest = right;

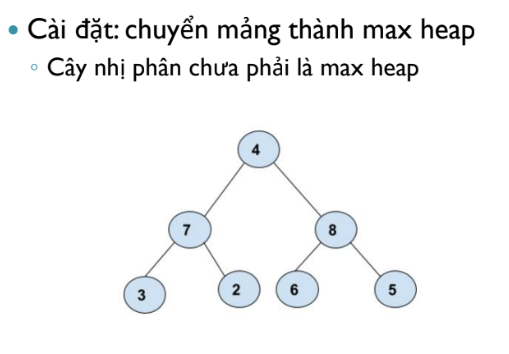
if (largest != i ) {

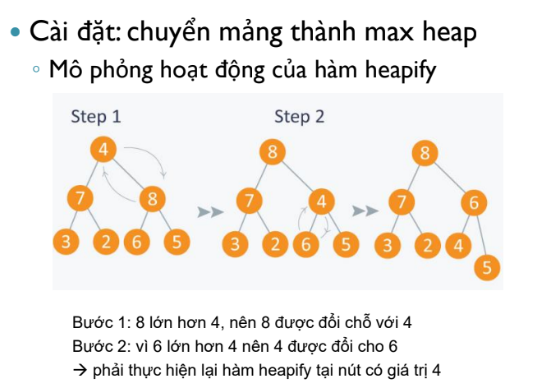
swap (A[i] , A[largest]);

heapify (A, largest);

}

}





 Cài đặt: chuyển mảng thành max heap

◦ Hàm heapify mới áp dụng cho một nút bất kỳ.

◦ Cần thực hiện cho toàn bộ heap

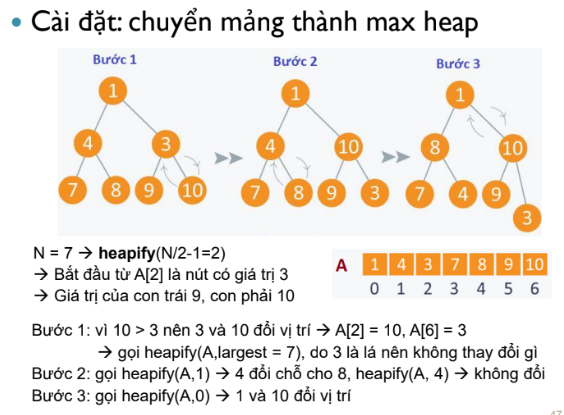
 Không cần áp dụng cho nút lá vì chúng không có con

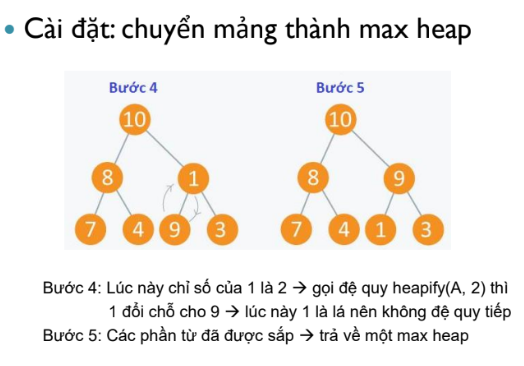
 Chỉ cần áp dụng đối với các nút trong

* có chỉ số từ 0 đến N / 2 - 1

void run\_heapify(int A[]) {

for(int i = N / 2 - 1 ; i >= 0 ; i-- ) { heapify (A, i) ;}}



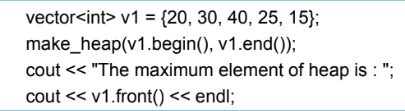


 Cài đặt: sử dụng thư viện STL trong C++

◦ Hàm make\_heap(): chuyển cấu trúc dạng range

sang heap

◦ Ví dụ:



◦ Hàm push\_heap(), pop\_heap(), is\_heap()

* **ứng dụng**

◦ Sắp xếp vung đống (heap sort)

◦ Hàng đợi ưu tiên (priority queue)

◦ Các thuật toán trên đồ thị

 Tìm đường đi ngắn nhất Dijkstra

 Cây bao trùm tối thiểu Prim

◦ Sắp xếp vun đống (heap sort)

 Sắp xếp các phần tử trong mảng A[] theo thứ tự

tăng dần

 Sử dụng max heap với ý tưởng sau:

 Tạo đống -> Nút gốc có giá trị lớn nhất -> lưu lại

 Thay thế nó bằng nút có giá trị nhỏ hơn trong cây->gọi hàm heapiffy để tìm được giá trị lớn thứ hai

 Cứ tiếp tục như vậy để tìm được giá trị lớn thứ ba, thứ tư, ...

◦ Sắp xếp vung đống (heap sort), gồm các bước

 Bước 1: Tạo max heap các phần tử trong mảng A sử

dụng hàm heapify

 Bước 2: Có A[0] là phần tử lớn nhất  đổi chỗ A[0]

cho phần tử cuối mảng (thường có giá trị nhỏ nhất)

 Bước 3: Tạo max heap cho các phần tử trong mảng

trừ phần tử cuối (là A[0] đã đúng vị trí)

 Kích thước của heap giảm đi 1

 Bước 4: Lặp lại bước 2 và bước 3 cho tới khi các

phần tử trong mảng đã được sắp

Độ phức tạp: O(NlogN)

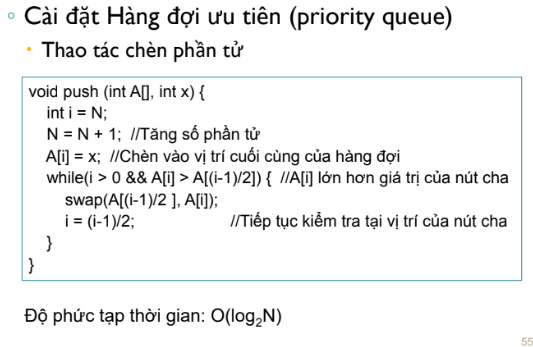
◦ Cài đặt Hàng đợi ưu tiên (priority queue)

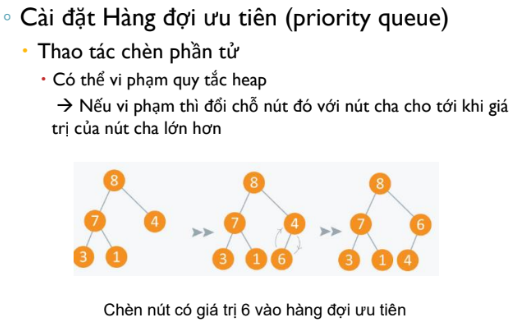
 Giả sử dùng mảng A để lưu các phần tử

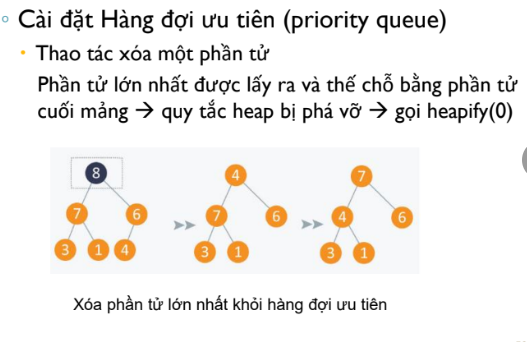
 Số phần tử hiện có là N

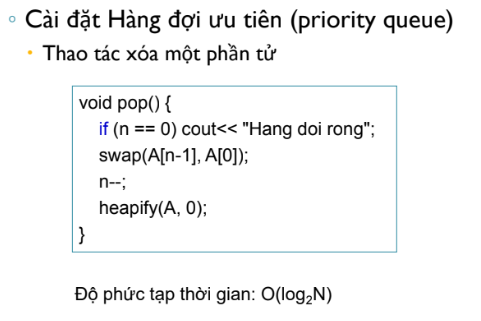
 Giá trị càng lớn thì độ ưu tiên càng cao

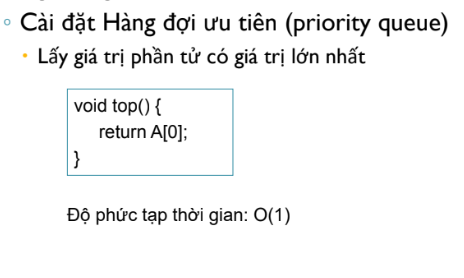
 Sử dụng cấu trúc max heap











**\*) Cây tìm kiếm nhị phân**

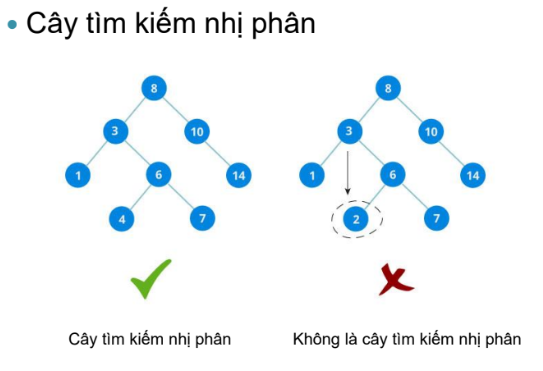
 Cây tìm kiếm nhị phân

◦ Là cây có các ràng buộc sau

 Giá trị của tất cả nút ở cây con bên trái phải nhỏ hơn hoặc bằng giá trị của nút gốc

 Giá trị của tất cả nút ở cây con bên phải phải lớn hơn giá trị của nút gốc

 Tất cả các cây con (trái và phải) đều phải đảm bảo 2 tính chất trên



◦ Ý nghĩa

 Được gọi là cây nhị phân vì mỗi nút của cây chỉ có tối đa hai con

 Một cấu trúc dữ liệu hiệu quả để xây dựng một danh sách mà dữ liệu trên đó được sắp xếp

 Được gọi là cây tìm kiếm nhị phân vì có thể được sử dụng để tìm kiếm một phần tử trong thời gianO(log2(n))

◦ Thêm phần tử

 Vị trí của các nút được thêm vào sẽ luôn nút lá

 Nếu nút hiện tại là NULL thì đó là vị trí cần thêm ->thêm và kết thúc

 Nếu giá trị cần thêm nhỏ hơn hoặc bằng giá trị của gốc hiện tại, gọi đệ quy chèn vào cây con bên trái

 Nếu giá trị cần thêm lớn hơn giá trị của gốc hiện tại,gọi đệ quy chèn vào cây con bên phải

◦ Tìm phần tử

 Nếu giá trị cần tìm bằng giá trị của nút hiện tại, trả về true và kết thúc

 Nếu giá trị cần tìm nhỏ hơn giá trị của nút hiện tại,gọi đệ quy tìm ở cây con bên trái

 Nếu giá trị cần tìm lớn hơn giá trị của nút hiện tại,gọi đệ quy tìm ở cây con bên phải

 Nếu tìm đến hết cây mà không thấy, trả về false và kết thúc

◦ Xóa phần tử

 Trường hợp 1: nút cần xóa là lá  xóa khỏi cây

 Trường hợp 2: nút cần xóa có một con

 Giải phóng nút bị xóa

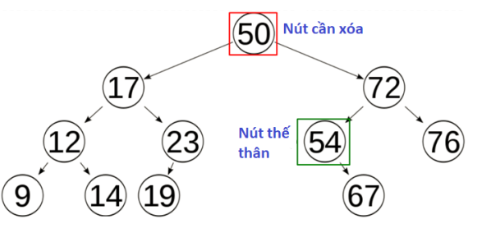
 Liên kết trực tiếp cây con duy nhất của nó với cha của nút bị xóa.

 Trường hợp 3: nút cần xóa có đủ hai con

 Tìm nút con trái nhất của cây con bên phải của nút cần xóa (Left most)

 Cập nhật giá trị của nút cần xóa bằng giá trị của nút trái nhất

 Gọi đệ quy xóa nút Left most khỏi cây (rơi vào trường hợp 1 hoặc trường hợp 2)



◦ Duyệt cây -> Thứ tự duyệt được đặt tên phụ thuộc vào vị trí của nút gốc trong quá trìnhduyệt

 Theo thứ tự trước: Gốc -> Trái -> Phải

 Theo thứ tự giữa: Trái ->Gốc -> Phải

 Thứ tự sau: Trái -> Phải -> Gốc

◦ Duyệt cây: Theo thứ tự trước

 Ghé thăm nút gốc

 Gọi đệ quy duyệt qua cây con bên trái

 Gọi đệ quy duyệt qua cây con bên phải

void PreOrder(node\_t\* root) {

if(root != NULL) {

cout<<root->data;

PreOrder(root->left);

PreOrder(root->right);

}

}

◦ Duyệt cây: Theo thứ tự giữa

 Gọi đệ quy duyệt qua cây con bên trái

 Ghé thăm nút gốc

 Gọi đệ quy duyệt qua cây con bên phải

void InOrder(node\_t\* root){

if (root != NULL) {

InOrder(root->left);

cout<<root->data;

InOrder(root->right);

}

}

◦ Duyệt cây: Theo thứ tự sau

 Gọi đệ quy duyệt qua cây con bên trái

 Gọi đệ quy duyệt qua cây con bên phải

 Ghé thăm nút gốc

void PostOrder(node\_t\* root){

if(root != NULL) {

PostOrder(root->left);

PostOrder(root->right);

cout<<root->data;

}

}

**Chương 3: Sắp xếp, tìm kiếm nâng cao**

**\*) Sắp xếp trong Counting sort(sxep dem phân phối)**

◦ Counting sort là một thuật toán sắp xếp cực nhanh một mảng các phần tử mà mỗi phần tử là các số nguyên không âm hoặc là một danh sách các ký tự được ánh xạ về dạng số để sắp xếp theo bảng chữ cái

◦ Counting sort không dựa vào so sánh

 Ý tưởng của Counting sort

◦ Dựa vào các khóa trong một khoảng cụ thể

◦ Nó làm việc dựa vào việc đếm số phần tử có giá trị khóa khác biệt (một kiểu băm). Sau đó thực hiện việc tính toán số học vị trí của từng phần tử trong chuỗi đầu ra

 Minh họa Counting sort

◦ Để đơn giản, giả sử dữ liệu trong khoảng 0 đến 9

◦ Dữ liệu đầu vào: 1, 4, 1, 2, 7, 5, 2

Bước 1: Đếm số lần xuất hiện của từng phần tử trong mảng cần sắp

Index: 0 1 2 3 4 5 6 7 8 9

Count: 0 2 2 0 1 1 0 1 0 0

Bước 2: Sửa mảng count sao cho chỉ số của mỗi phần tử lưu tổng số lần đếm trước đó -> vị trí của chúng trong mảng đầu ra

Index: 0 1 2 3 4 5 6 7 8 9

Count: 0 2 4 4 5 6 6 7 7 7

Bước 3: Xuất từng phần tử trong chuỗi đầu vào tại ví trí là tổng số lần đếm trừ đi 1 và trừ số lần đếm đi 1.

Xử lý dữ liệu đầu vào: 1, 4, 1, 2, 7, 5, 2. Vị trí của 1 là (2-1) = 1 thì đặt số 1 vào chỉ số 1 trong mảng đầu ra. Gặp số 1 tiếp theo thì giảm chỉ số của nó đi 1 và đặt số 1 vào chỉ số 0

◦ Độ phức tạp thời gian: O(n+k), trong đó n là số phần tử trong dãy đầu vào và k là khoảng dữ liệu đầu vào

◦ Độ phức tạp không gian: O(n+k)

◦ Nhược điểm:

 Độ phức tạp không gian lớn cho việc sắp xếp số lượng nhỏ các phần tử nhưng có khoảng giá trị (khoảng cách từ số nhỏ nhất và số lớn nhất) lớn vì nó đòi hỏi một lượng lớn không gian không được sử dụng

 Điều gì xảy ra nếu các phần tử trong khoảng tử 1 đến n2?

Khi đó counting sort sẽ có độ phức tạp thời gian là O(n2)

**\*) Sắp xếp theo cơ số (Radix sort)**

◦ Một thuật toán sắp xếp không so sánh

◦ Dựa trên nguyên tắc phân loại thư của bưu điện nên nó còn có tên là Postman sort

◦ Bưu điện chuyển lượng lớn thư đến các địa phương khác nhau -> tổ chức phân loại thư theo phân cấp

 Các thư đến cùng một tỉnh/thành phố sẽ được đưa vào 1 lô

 Các bưu điện tỉnh phân các thư theo lô cùng quận, huyện

 Cứ như vậy, các thư sẽ đến được tay người nhận thư

◦ Radix sort dựa trên ý tưởng nếu một danh sách đã được sắp xếp hoàn chỉnh thì từng phần tử cũng sẽ được sắp xếp hoàn chỉnh dựa trên giá trị của các phần tử đó

◦ Để sắp xếp dãy a1, a2, ..., an thuật toán Radix sort thực hiện như sau:

 Giả sử mỗi phần tử ai trong dãy a1, a2, ..., an là một số nguyên có tối đa m chữ số

 Phân loại các phần tử lần lượt theo các chữ số hàng đơn vị,hàng chục, hàng trăm, ... tương tự việc phân loại thư theo tỉnh thành, quận/huyện, phường/xã, ...

 Các bước thực hiện sắp xếp theo cơ số

◦ Bước 1: k = 0; //(k = 0: hàng đơn vị; k = 1: hàng chục, ...), k là chữ số hiện thời (tức là sắp xếp các số hàng đơn vị trước)

◦ Bước 2: Khởi tạo 10 lô B0, B1, ... , B9 rỗng; //Tạo các lô chứa các loại chữ số khác nhau (từ 0 đến 9)

◦ Bước 3:

for (i = 0; i < n; i++) //Lặp từ phần tử a0 đến an-1

Ðặt ai vào lô Bt với t là chữ số thứ k của ai;

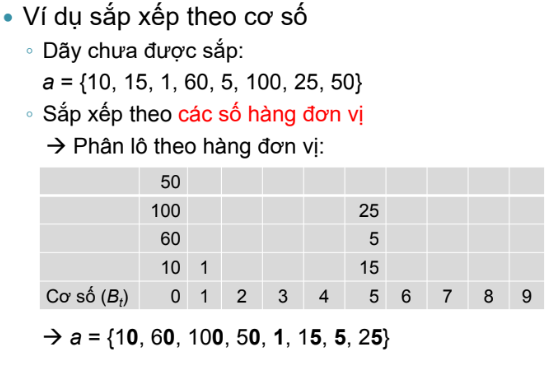
◦ Bước 4: Nối B0, B1, .. , B9 lại (theo đúng trình tự) thành a

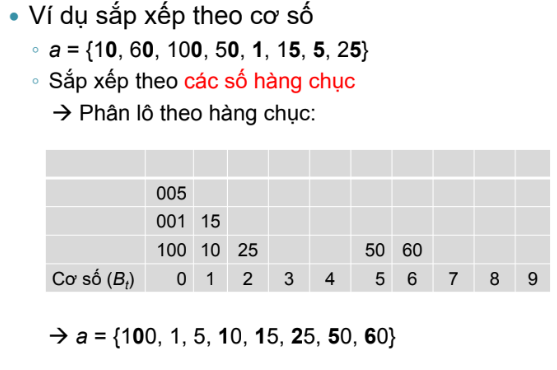
◦ Bước 5:

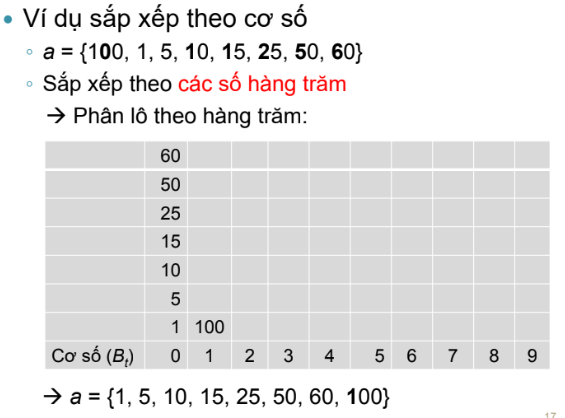
k = k + 1; //k = 0  chuyển lên hàng chục, ...

Nếu k < m thì trở lại bước 2.

Ngược lại: Dừng







**\*) Sắp xếp ngoài**

**- Trộn Run**

◦ Thuật toán sắp xếp tập tin bằng phương pháp trộn Run có thể tóm lược như sau:

◦ Input: f0 là tập tin cần sắp thứ tự

◦ Ouput: f0 là tập tin đã được sắp thứ tự

◦ Gọi f1, f2 là 2 tập tin trộn

◦ Trong đó f0, f1, f2 có thể là các tập tin văn bản thường (text file) hoặc các tập tin nhị phân

 Phương pháp trộn Run – Minh họa

◦ Bước 1:

 Giả sử các phần tử trên f0 là:

24 12 67 33 58 42 11 34 29 31

 Khởi tạo f1, f2 rỗng

 Thực hiện phân bố m = 1 phần tử từ f0 lần lượt vào f1, f2:

f1: 24 67 58 11 29

f2: 12 33 42 34 31

 Trộn f1,f2 vào f0

f0: 12 24 33 67 42 58 11 34 29 31

◦ Bước 2:

 Phân bố m = 2 \*m = 2 phần tử từ f0 vào f1, f2

 f0: 12 24 33 67 42 58 11 34 29 31

 f1: 12 24 42 58 29 31

 f2: 33 67 11 34

 Trộn f1, f2 thành f0:

 f0: 12 24 33 6711 34 42 58 29 31

Bước 3:

 Tương tự bước 2, phân bố m = 2 \* m = 4 phần tử từ f0 vào f1, f2

f0: 12 24 33 67 11 34 42 58 29 31

f1: 12 24 33 67 29 31

f2: 11 34 42 58

 Trộn f1, f2 thành f0:

f0: 11 12 24 33 34 42 58 67 29 31

◦ Bước 4:

 Phân bố m = 2 \* m = 8 phần tử lần lượt từ f0 vào f1, f2

f1: 11 12 24 33 34 42 58 67

f2: 29 31

 Trộn f1, f2 thành f0:

f0: 11 12 24 29 31 33 34 42 58 67

◦ Bước 5:

 Lặp lại tương tự các bước trên cho tới khi chiều dài m của run cần phân bố lớn hơn chiều dài N của f0 thì dừng

 Phương pháp trộn Run – Cài đặt

m = 1

while (m < số phần tử của f0)

{

Chia [Distribute] m phần tử của f0 lần lượt cho f1 và f2

Trộn [Merge] f1 và f2 lần lượt vào f0

m = m \* 2

}

Phương pháp trộn Run – Đánh giá

◦ Đánh giá

 Cần ít nhất N không gian trống trên đĩa để hoạt động

 Số bước log2N (vì mỗi lần xử lý 1 dãy tăng gấp 2)

 Mỗi bước:

 Phân phối (Distribute): Copy N lần

 Trộn (Merge): Copy N lần, so sánh N/2 lần

 Tổng cộng:

 Copy: 2N \*log2 N

 So sánh: N/2 \* log2 N

 Hạn chế:

 Không tận dụng được dữ liệu đã được sắp bộ phận

 Độ dài dãy con xử lý ở bước k <= 2k

* **Trộn tự nhiên Run**

 Khái niệm Run

◦ Run là một dãy liên tiếp các phần tử được sắp thứ

tự

◦ Ví dụ: 2 4 7 12 55 là một Run

◦ Chiều dài Run chính là số phần tử trong Run

◦ Chẳng hạn Run ở ví dụ trên có chiều dài là 5

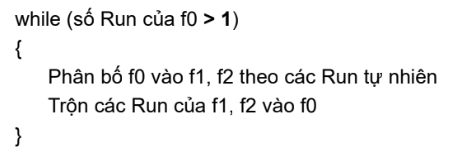
 Phương pháp trộn tự nhiên – Đặc điểm

◦ Trong phương pháp trộn ở trên, thuật toán chưa tận dụng được chiều dài cực đại của các Run trước khi phân bố -> chưa tối ưu

◦ Đặc điểm của phương pháp trộn tự nhiên là tận dụng chiều dài “tự nhiên” của các Run ban đầu

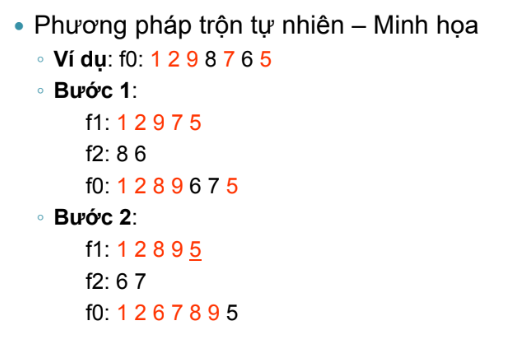
-> Nghĩa là thực hiện việc trộn các Run có độ dài cực đại với nhau cho tới khi dãy chỉ còn một Run duy nhất -> dãy đã được sắp

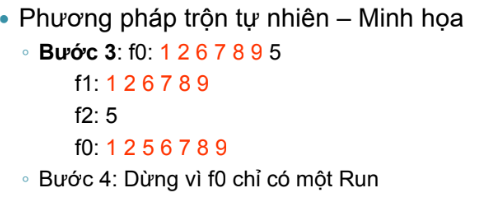
 Phương pháp trộn tự nhiên – Thuật toán



◦ Phân bố: Chia xoay vòng dữ liệu của file f0 cho f1 và f2, mỗi lần 1 run cho đến khi file f0 hết

◦ Trộn: Trộn từng cặp run của f1 và f2 tạo thành run mới trên f0





**\*) Bảng băm và ứng dụng**

**- Bảng băm**

◦ Bảng băm là một loại cấu trúc dữ liệu được dùng để chứa cặp key/value

◦ Chỉ số được tính toán bởi hàm băm -> giúp cho việc chèn hoặc tìm kiếm dữ liệu được dễ dàng hơn

◦ Với một bảng băm có một hàm băm được thực hiện tốt thì trong trường hợp lý tưởng, việc tìm kiếm các phần tử trong bảng có thời gian là O(1)

◦ Ví dụ: Đếm số lần xuất hiện của các ký tự trong chuỗi: “ababcd”

◦ Cách thông thường: duyệt qua tất cả các phần tử của chuỗi và đếm số lần xuất hiện của từng ký tự

◦ Cách tiếp cận tốt hơn:

 Sử dụng kỹ thuật băm bằng cách dùng một mảng có kích thước là 26 để chứa giá trị là số lần xuất hiện của một ký tự trong chuỗi

 Dùng hàm băm để tính toán ra chỉ số của ký tự trong chuỗi

 Duyệt qua toàn bộ chuỗi và tăng giá trị của phần tử mảng có chỉ số tương ứng bằng với chỉ số của ký tự vừa được tính ở bước trên

- **Ứng dụng**

◦ Được sử dụng để cài đặt một số cấu trúc dữ liệu trong C++ (unordered\_set, unordered\_map), Java,C#, Python

◦ Với các bài toán đặc thù -> sẽ phải tự viết hàm băm riêng và xây dựng cấu trúc dữ liệu bảng băm cho phù hợp

◦ Bảng băm thường được ứng dụng trong

 Lập chỉ mục Cơ sở dữ liệu

 Tổ chức bộ nhớ đệm

 Biểu diễn các đối tượng: Perl, Python, JavaScript và Ruby

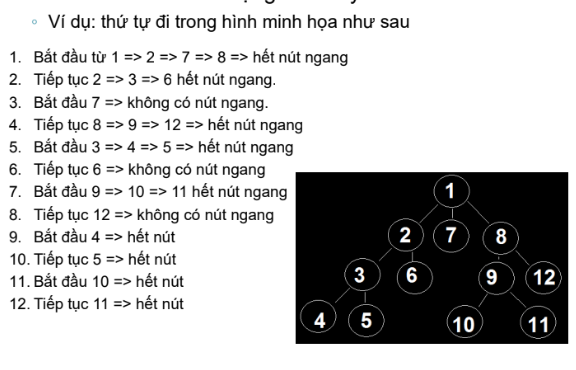
 Áp dụng trong các thuật toán để tăng tốc độ tính toán

**\*) Tìm kiếm theo chiều rộng và theo chiều sâu trên cây**

**- Theo chiều rộng**

◦Thuật toán tìm kiếm theo chiều rộng (Breadth First Search - BFS) là một thuật toán duyệt hoặc tìm kiếm một phần tử trên một cấu trúc dữ liệu dạng cây hay một đồ thị

◦ BFS sẽ ưu tiên theo chiều ngang, nghĩa là duyệt từ trái qua phải hết rồi mới duyệt tiếp xuống dưới cho từng phần tử



Thuật toán

B1. Tạo một queue rỗng bfqueue

B2. Enqueue nút gốc vào bfqueue

B3. Vòng lặp while khi queue không rỗng

B3.1. Gán nút tạm current bằng phần tử đầu tiên của queue

B3.2. Dequeue phần tử đầu tiên từ bfqueue

B3.3. In ra dữ liệu của nút current

B3.4. Enqueue các con của current vào bfqueue từ trái qua phải

◦ Các tính chất thuật toán BFS

 Sử dụng cấu trúc dữ liệu hàng đợi để lưu trữ các nút

 Là cấu trúc dữ liệu dạng dạng cây hay đồ thị

 Độ phức tạp thời gian là: O(|V| + |E|) với V và E lần lượt là số đỉnh và số cạnh duyệt qua

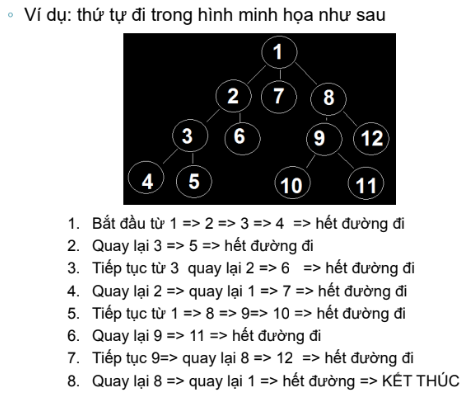
 Độ phức tạp không gian là O(|V|)

* **Theo chiều sâu**

◦ Thuật toán tìm kiếm theo chiều sâu (Depth First Search - DFS) là một thuật toán duyệt hoặc tìm kiếm một phần tử trên một cấu trúc dữ liệu dạng cây hay một đồ thị

◦ DFS bắt đầu đi từ một đỉnh của cây, sau đó từ đỉnh đó tìm ra nút đầu tiên và cứ tìm các nút tiếp theo cho đến khi không đi được nữa thì lại quay về nút trước đó và tìm sang nút bên cạnh để đi tiếp

◦ DFS giống như đi trong mê cung, khi gặp ngõ cụt thì qua trở lại và thử đi theo một hướng khác



Thuật toán

Khởi tạo một ngăn xếp S rỗng để lưu các nút

Với mỗi đỉnh u, khởi tạo u.visited = false

Đưa nút gốc (nút được viếng thăm đầu tiên) vào S

while S khác rỗng

Lấy ra phần tử u khỏi ngăn xếp S

If u.visited = false then

u.visited = true

Với mỗi hàng xóm chưa được viếng thăm w của u

Đưa w vào S

Kết thúc tiến trình khi tất cả các nút đã được viếng thăm

◦ Các tính chất thuật toán DFS

 Là cấu trúc dữ liệu dạng đồ thị, hay dạng cây

 Độ phức tạp thời gian là: O(|V| + |E|) với V và E lần lượt là số đỉnh và số cạnh duyệt qua

 Độ phức tạp không gian là O(|V|)

* **Ưu và ngược điểm của BFS và DFS**

◦ Các thuật toán BFS và DFS là tìm kiếm mù, tức là tìm kiếm không theo sự hướng dẫn nào, phạm vi tìm kiếm được phát triển liên tục cho đến khi gặp trạng thái đích

◦ BFS đảm bảo sẽ luôn tìm được nghiệm nếu bài toán có nghiệm, trong khi DFS không phải lúc nào cũng đảm bảo tìm thấy nghiệm

◦ DFS càng ngày càng đi sâu hơn theo nhánh đang duyệt. Vậy chuyện gì sẽ xảy ra nếu nhánh đó là nhánh vô hạn?

◦ Trong nhiều trường hợp DFS tìm được kết quả nhanh hơn BFS, không gian bộ nhớ cần cho việc lưu trữ cũng ít hơn

◦ Đối với các chiến lược tìm kiếm mù đó là việc bùng nổ tổ hợp trong quá trình tìm kiếm. Càng về sau, số lượng trạng thái chờ để được xét duyệt trở nên rất lớn khiến cho không gian lưu trữ cũng lớn theo

 Ứng dụng của các thuật toán tìm kiếm trên cây

◦ Tìm kiếm trong không gian trạng thái

 Các nút của cây chính là các trạng thái của không gian trạng thái

 Nút gốc là trạng thái ban đầu. Quá trình tìm kiếm sẽ ngừng khi gặp tập các trạng thái kết thúc (các trạng thái đích)

 Mỗi chiến lược tìm kiếm sẽ có cách duyệt cây tìm kiếm khác nhau

**Chương 4: Đồ thị**

**\*) Tìm kiếm theo chiều rộng và chiều sâu**

**- theo chiều rộng**

◦ Thăm tất cả các đỉnh kề với đỉnh u trước khi tiếp tục với các đỉnh khác-> sử dụng một cấu trúc dữ liệu hàng đợi (queue) để lưu trữ thông tin trung gian thu được trong quá trình tìm kiếm

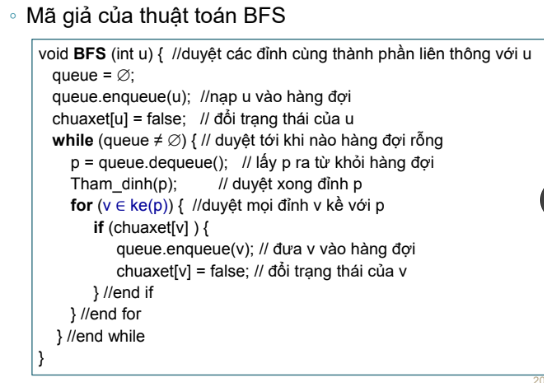
1. Thêm đỉnh gốc vào queue và đánh dấu đỉnh gốc.

2. Nếu queue chưa rỗng, lấy ra đỉnh u đầu tiên khỏi queue. Xét các đỉnh v kề với đỉnh u

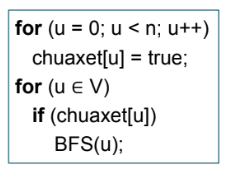
Nếu đỉnh v đã được đánh dấu thì bỏ qua.

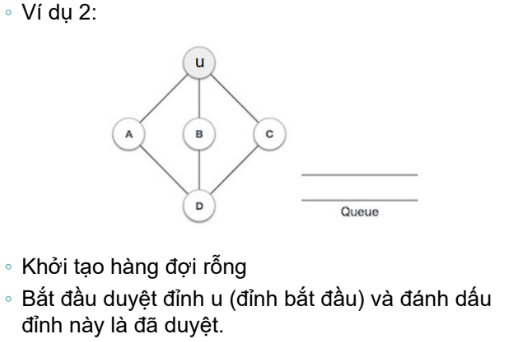
Nếu v chưa được đánh dấu thì thêm đỉnh v vào queue và đánh dấu đỉnh v.

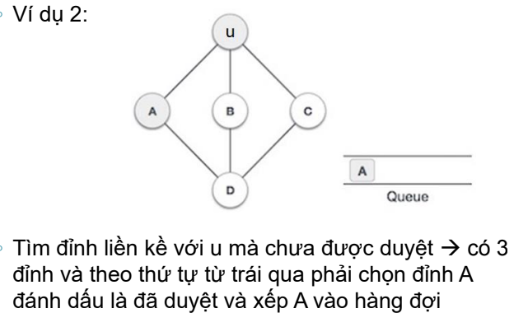
3. Nếu queue rỗng, dừng quá trình tìm kiếm

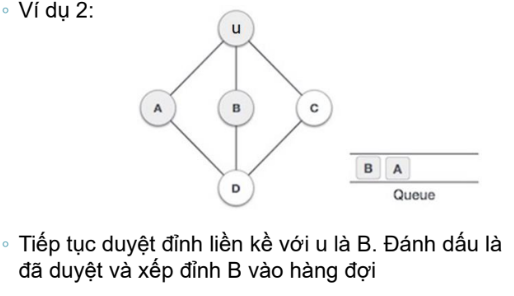


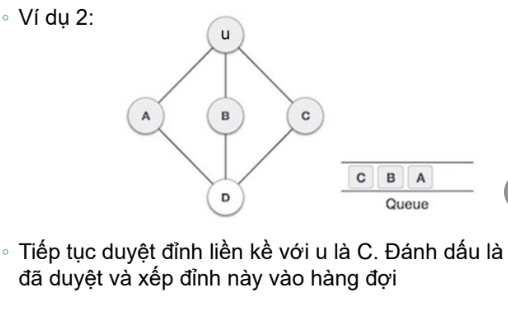
Để thăm tất cả các đỉnh của đồ thị thuộc các thành phần liên thông, thực hiện đoạn chương trình sau

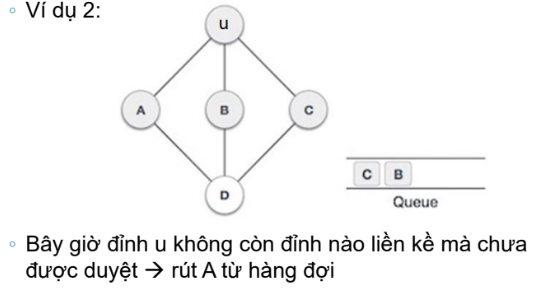


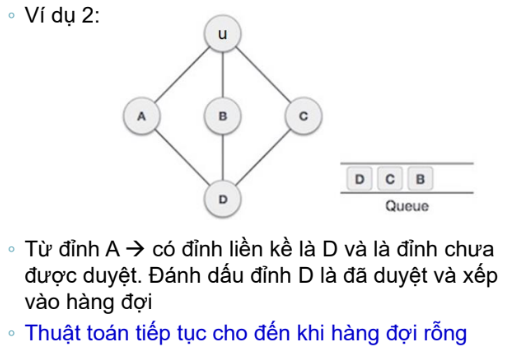












◦ Phân tích thuật toán:

 Không gian: 2|V|, với V là tập các đỉnh của đồ thị và |V| là số đỉnh của đồ thị

 Thời gian: O(|E| + |V|), với V và E là tập các đỉnh và cung của đồ thị vì trong trường hợp xấu nhất, mỗi đỉnh và cung của đồ thị được thăm đúng một lần.

O(|E| + |V|) nằm trong khoảng từ O(|V|) đến O(|V|2), tùy theo số cung của đồ thị

◦ Ưu điểm

 Xét duyệt tất cả các đỉnh để trả về kết quả

 Nếu số đỉnh là hữu hạn, thuật toán chắc chắn tìm ra kết quả

◦ Nhược điểm

 Mang tính chất vét cạn, không nên áp dụng nếu duyệt số đỉnh quá lớn

 Mang tính chất mù quáng vì duyệt tất cả đỉnh mà không chú ý đến thông tin trong các đỉnh để duyệt hiệu quả, dẫn đến duyệt qua các đỉnh không cần thiết

◦ Ứng dụng:

 Tìm các thành phần liên thông của đồ thị hoặc kiểm tra đồ thị hai phía

 Tìm đường đi ngắn nhất và cây bao trùm tối thiểu trong đồ thị không trọng số

 Tìm tất cả các nút lân cận của một mạng ngang hàng

 Crawlers in Search Engines: bắt đầu từ một trang nguồn và lần theo các liên kết từ trang nguồn đó, và làm tương tự trong các trang được lần tới

 Các trang website mạng xã hội: tìm các người dùng kết nối tới một người dùng cụ thể với khoảng cách k

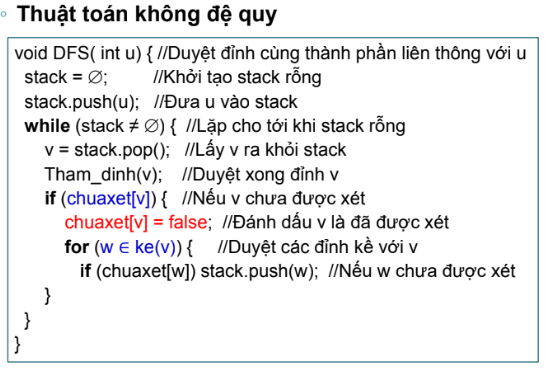
 Hệ thống định vị GPS: tìm các địa điểm lân cận

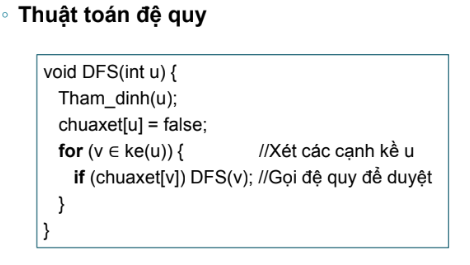
 Quảng bá thông tin mạng (Broadcasting in Network)

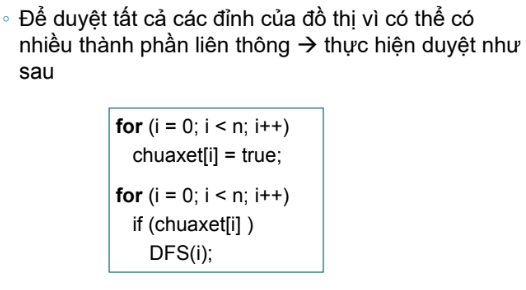
* **Theo chiều sâu**

◦ Thăm một đỉnh kề của u, sau đó thăm tiếp một đỉnh kề của đỉnh kề đó -> Sử dụng ngăn xếp (stack) để ghi nhớ đỉnh liền kề để bắt đầu việc tìm kiếm

◦ Thuật toán tiếp tục cho tới khi gặp được đỉnh cầntìm hoặc tới một nút không có con. Khi đó thuật toán quay lui về đỉnh vừa mới tìm kiếm ở bước trước







◦ Ứng dụng:

 Xác định các thành phần liên thông của đồ thị

 Xác định các thành phần liên thông mạnh của đồ thị có hướng

 Kiểm tra một đồ thị có phải là đồ thị phẳng hay không

 Cây bao trùm tối thiểu trên đồ thị có trọng số

**\*) Tìm các thành phần liên thông**

◦ Cho đồ thị vô hướng G = (V, E), tìm tất cả những thành phần liên thông của G

◦ Áp dụng thuật toán BFS hoặc DFS cho đỉnh nguồn cho trước u, tìm tất cả những đỉnh cùng thành phần liên thông của u

◦ Nhận xét:

 Nếu đồ thị liên thông thì số thành phần liên thông của nó là 1 -> tương đương với phép duyệt theo thủ tục BFS hoặc DFS được gọi đến đúng một lần (BFS & DFS không sử dụng đệ quy)

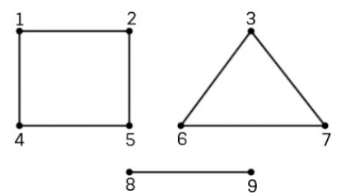
 Nếu đồ thị không liên thông thì số thành phần liên thông lớn hơn 1

 Có thể tách chúng thành những đồ thị con liên thông

 Trong phép duyệt đồ thị, số thành phần liên thông của nó bằng số lần gọi tới thủ tục BFS hoặc DFS

 Nếu đỉnh i được duyệt thuộc thành phần liên thông thứ j = so\_tplt, ta ghi nhận chuaxet[i] = so\_tplt

 Các đỉnh cùng thành phần liên thông nếu chúng có cùng giá trị trong mảng chuaxet[]



◦ Thuật toán

void BFS (int u) { //Duyệt các đỉnh cùng thành phần liên thông với u

queue = ; //Khởi tạo queue rỗng

queue.enqueue(u); //nạp u vào hàng đợi

so\_tplt++;

chuaxet[u] = so\_tplt; //so\_tplt là biến toàn cục

while (queue ≠ ) {

p = queue.dequeue(); // lấy p ra khỏi queue

for (v ∈ ke(p)) { // v thuộc tập đỉnh kề của p

if (chuaxet[v] ) {

queue.enqueue(v); //nạp v vào hàng đợi

chuaxet[v] = so\_tplt; //v có cùng thành phần liên thông với p

}

}

}

}

void Tim\_tp\_lienthong() {

for (i = 0; i < n; i++)

chuaxet[i] = 0;

for (i = 0; i < n; i++) //Duyệt các đỉnh cùng TPLT với i

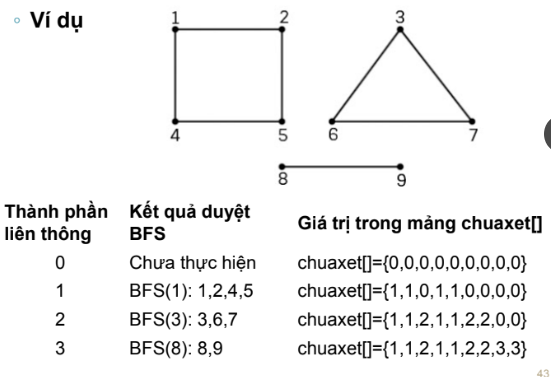
if (chuaxet[i] == 0) {

so\_tplt++;

BFS(i);

}

}



**\*) Đường đi ngắn nhất**

**- Bellman ford**

◦ Tìm các đường đi ngắn nhất nguồn đơn trong một

đồ thị có hướng có trọng số

◦ Một số cung có thể có trọng số âm

◦ Đơn giản hơn Dijkstra, phù hợp với các hệ phân tán

◦ Độ phức tạp về thời gian là O(VE)

◦ Thuật toán

Bước 1. Gán tất cả các đỉnh của đồ thị với giá trị khoảng cách là vô cùng. Giá trị của đỉnh nguồn là 0. Tạo mảng dist[] với kích thước |V| có giá trị là vô cùng (INF), đỉnh nguồn có giá trị 0

Bước 2. Thực hiện |V| - 1 lần các công việc sau

Với mỗi cạnh uv, thực hiện

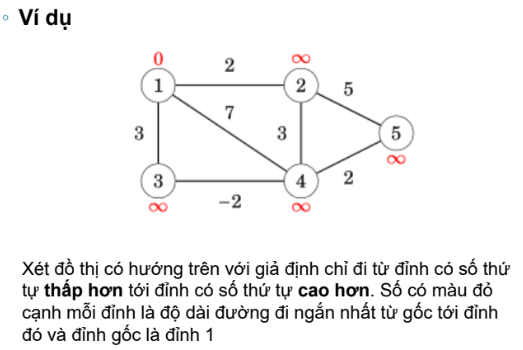
If dist[u]  INF && dist[v] > dist[u] + trọng số của uv then

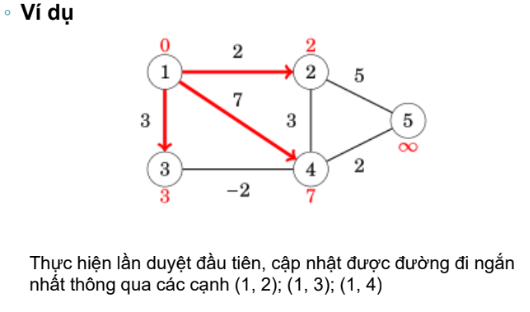
dist[v] = dist[u] + trọng số của uv

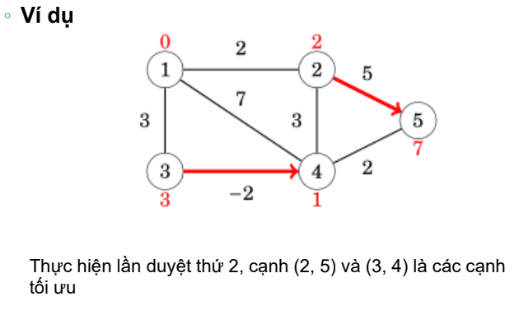
Bước 3. Báo cáo có chu trình âm. Với mỗi cạnh uv, thực hiện

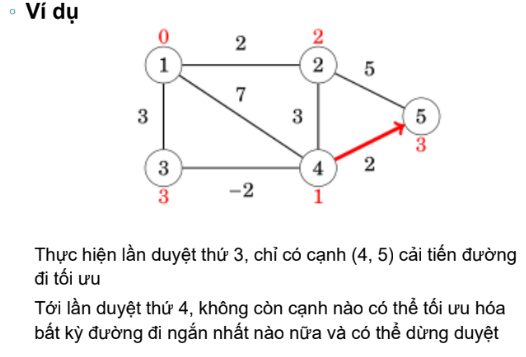
If dist[v] > dist[u] + trọng số của uv then “Đồ thị bao gồm chu trình âm”

Bước 2 đảm bảo khoảng cách ngắn nhất nếu đồ thị không có chu trình số âm. Nếu thực hiện lặp tất cả các cạnh thêm một lần nữa và thu được đường đi ngắn hơn đối với bất kỳ đỉnh nào thì có một chu trình âm.









* **Floyd-Warshall**

◦ Giải bài toán đường đi ngắn nhất cho mọi cặp đỉnh trong đồ thị có cạnh mang trọng số (dương hoặc âm) dựa trên khái niệm các Đỉnh trung gian

◦ Lưu ý là trong đồ thị không được có chu trình nào có tổng trọng số các cạnh là âm

◦ Độ phức tạp về thời gian: O(V3)

◦ Các bước của thuật toán với đồ thị N đỉnh

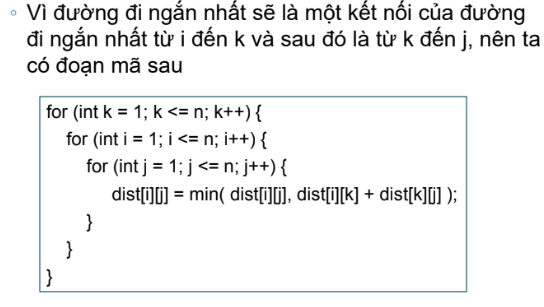
Bước 1. Khởi tạo mảng hai chiều dist[][] chứa đường đi ngắn nhất giữa hai đỉnh bất kỳ, có giá trị là vô cùng nếu không có đường nối trực tiếp, ngược lại 🡪 trọng số của đường nối đó

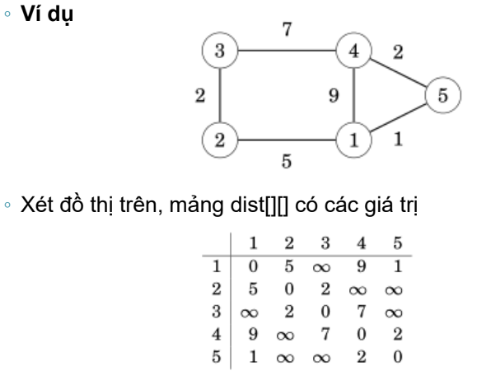
Bước 2. Tìm tất cả các đường đi ngắn nhất trực tiếp giữa hai cặp đỉnh, sau đó sử dụng k = 1 đỉnh đầu tiên làm đỉnh trung gian, sử dụng k = 2 đỉnh đầu tiên làm đỉnh trung gian, ..., cho đến khi sử dụng tất cả N đỉnh làm đỉnh trung gian

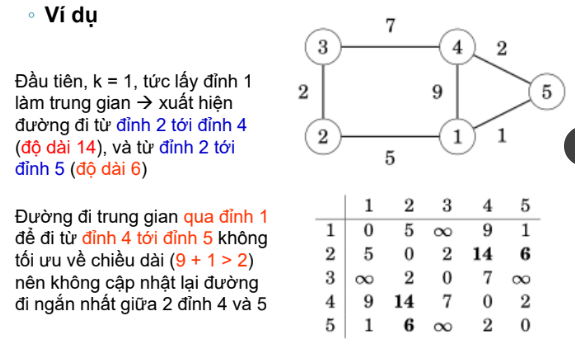
Bước 3. Tối thiểu hóa các đường đi ngắn nhất giữa hai cặp đỉnh bất kỳ trong bước trước. Với hai đỉnh (i, j) bất kỳ, đường đi ngắn nhất là:

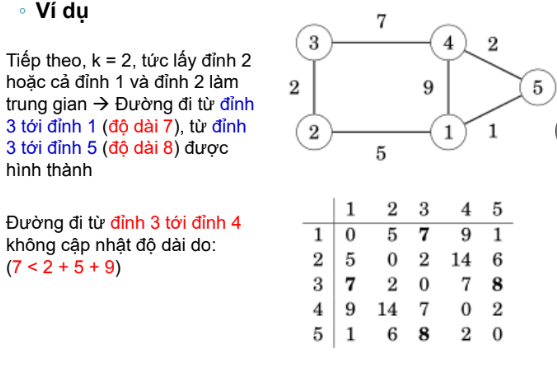
min(dist[i][j], dist[i][k] + dist[k][j] )

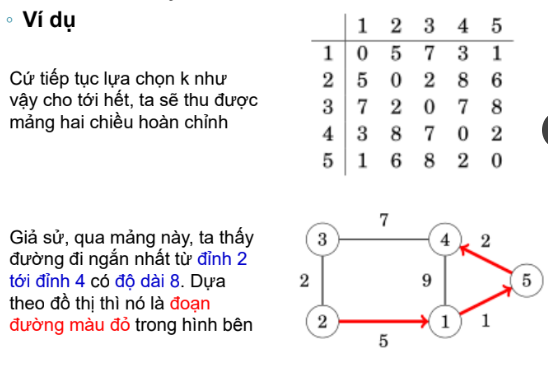
trong đó, dist[i][k] là đường đi ngắn nhất chỉ sử dụng k đỉnh trung gian đầu tiên, dist[k][j] là đường đi ngắn nhất giữa cặp đỉnh k và j









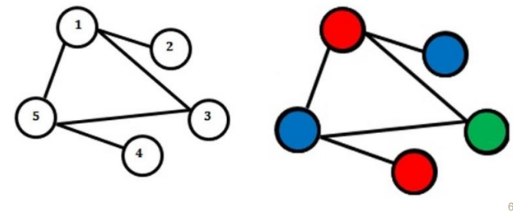


**\*) Tô màu đồ thị**

 Bài toán tô màu đồ thị

◦ Tô màu (đỉnh) đồ thị là việc thực hiện gán màu cho mỗi đỉnh của đồ thị, sao cho hai đỉnh kề nhau không cùng một màu và số màu được sử dụng là ít nhất

◦ Số màu ít nhất có thể sử dụng để tô màu đồ thị được gọi là sắc số (chromatic number) của đồ thị đó



 Thuật toán tham lam cơ bản tô màu đồ thị

◦ Bài toán tô màu đồ thị với số màu tối thiểu thuộc lớp bài toán NP-đầy đủ

◦ Thuật toán tham lam cơ bản không sử dụng nhiều hơn d + 1 màu với d là bậc lớn nhất của đồ thị

◦ Thuật toán

B1. Tô màu đỉnh đầu tiên với màu thứ nhất.

B2. Thực hiện các thao tác sau với V - 1 đỉnh còn lại.

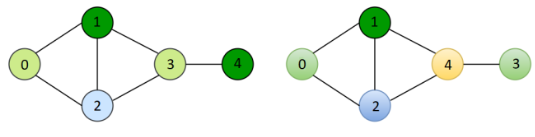
Xem xét đỉnh được chọn hiện thời v và tô màu đỉnh v với màu được đánh số thấp nhất mà chưa được sử dụng để tô bất kỳ đỉnh nào liền kề với v trước đó. Nếu tất cả các màu đã được sử dụng trước đó xuất hiện trên đỉnh liền kề với v, gán một màu mới cho nó.

◦ Phân tích thuật toán

◦ Độ phức tạp về thời gian: O(V2 + E)

◦ Không phải lúc nào cũng sử dụng số màu ít nhất

◦ Số màu được sử dụng đôi khi phụ thuộc vào thứ tự các đỉnh được tô -> thứ tự các đỉnh được tô là quan trọng -> cần tìm cách sắp xếp các đỉnh được tô để thuật toán làm việc hiệu quả hơn



 Thuật toán Welsh Powell

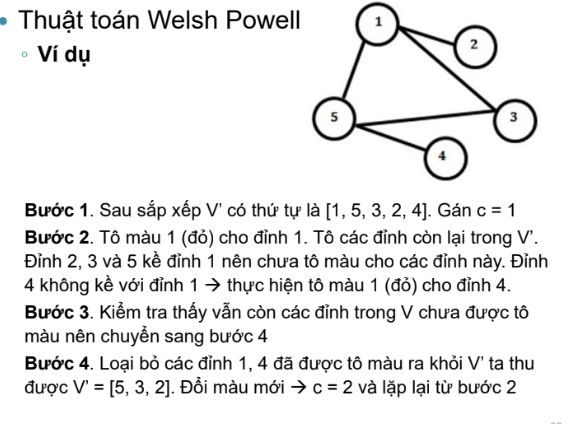
◦ Ý tưởng: bắt đầu với các đỉnh có bậc lớn nhất ->xử lý các đỉnh có khả năng xung đột sớm nhất có thể

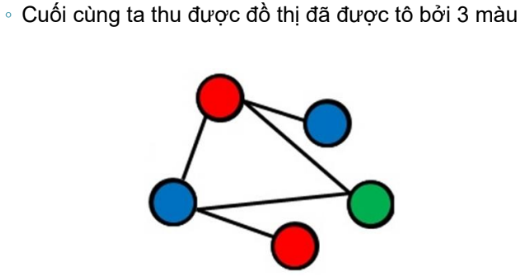
Bước 1. Tính giá trị bậc của các đỉnh trong V. Lập danh sách V’ = [v1, v2, ...,vn] là các đỉnh của đồ thị được sắp xếp theo thứ tự bậc giảm dần: d(v1) > d(v2) > ... > d(vn). Gán màu c = 1

Bước 2. Tô màu c cho đỉnh đầu tiên trong danh sách V’. Duyệt lần lượt các đỉnh khác trong V’ và chỉ tô màu c cho các đỉnh không kề đỉnh đã có màu c.

Bước 3. Kiểm tra nếu tất cả các đỉnh trong V đã được tô màu thì thuật toán kết thúc, đồ thị đã sử dụng c màu để tô. Ngược lại, nếu vẫn còn đỉnh chưa được tô thì chuyển sang bước 4.

Bước 4. Loại khỏi danh sách V’ các đỉnh đã được tô màu. Các đỉnh trong V’ vẫn theo thứ tự bậc giảm dần. Gán c = c + 1 và quay lại bước 2





 Ứng dụng

◦ Lập lịch và xếp thời khóa biểu

◦ Gán tần số cho trạm mobile

◦ Kiểm tra một đồ thị là đồ thị phân đôi hay không

◦ Tô màu bản đồ

**\*) Cây tối thiểu bao trùm Prim**

◦ Là thuật toán theo chiến lược tham lam

◦ Bắt đầu với MST rỗng

◦ Duy trì hai tập đỉnh, tập thứ nhất là các đỉnh nằm trong MST và tập thứ hai chưa được đưa vào MST

◦ Mỗi bước xem xét các cạnh nối hai tập đỉnh và chọn đỉnh có trọng số nhỏ nhất

◦ Đưa cạnh được chọn vào MST

◦ Độ phức tạp về thời gian là O(|V|2)

 Thuật toán Prim

Bước 1. Khởi tạo cấu trúc mstSet lưu các đỉnh đã nằm trong MST

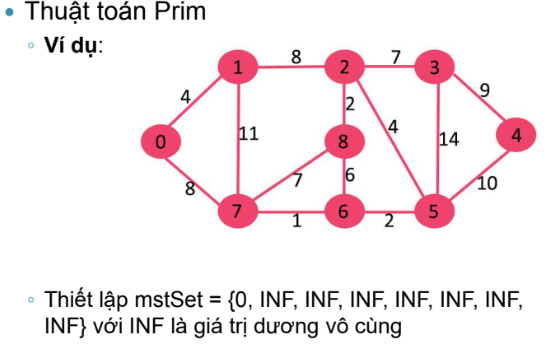
Bước 2. Gán khóa cho các đỉnh giá trị dương vô cùng, đỉnh được chọn đầu tiên có khóa là 0

Bước 3. Lặp cho đến khi mstSet chứa tất cả các đỉnh

Bước 3.1. Chọn đỉnh u chưa nằm trong mstSet và có giá trị khóa nhỏ nhất

Bước 3.2. Thêm u vào mstSet

Bước 3.3. Cập nhật giá trị khóa của tất cả các đỉnh kề u. Với mỗi cạnh v kề u, nếu trọng số của cạnh u-v nhỏ hơn giá trị khóa trước đó của v thì cập nhật giá trị khóa là trọng số của u-v

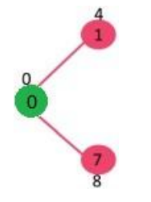


Chọn đỉnh 0 (có khóa nhỏ nhất), thêm vào mstSet.

* mstSet = {0}

Cập nhật khóa của tất cả các đỉnh kề 0 là 1 và 7

* Khóa của các đỉnh 1 và 7 thành 4 và 8

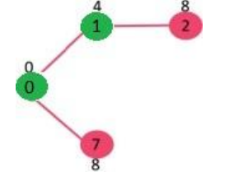


Chọn đỉnh có khóa nhỏ nhất không nằm trong MST.

* Đỉnh 1 được chọn và thêm vào mstSet
* mstSet = {0, 1}

Cập nhật khóa của tất cả các đỉnh kề 1 là 8

* Khóa của đỉnh 2 thành 8

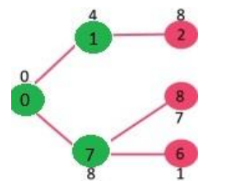


Chọn đỉnh có khóa nhỏ nhất không nằm trong MST.

* Chọn đỉnh 2 hoặc 7, chọn 7 và thêm vào mstSet
* mstSet = {0, 1, 7}

Cập nhật khóa của tất cả các đỉnh kề 7 là 6 và 8

* Khóa của các đỉnh 6 và 8 thành 1 và 7

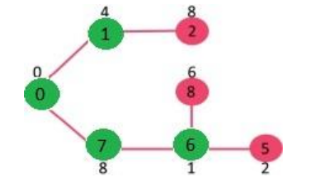


Chọn đỉnh có khóa nhỏ nhất không nằm trong MST.

* Chọn đỉnh 6 và thêm vào mstSet
* mstSet = {0, 1, 7, 6}

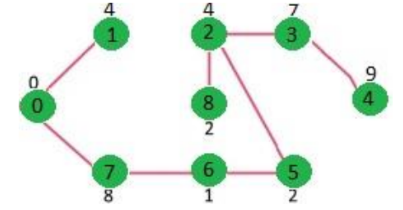
Cập nhật khóa của tất cả các đỉnh kề 6 là 5 và 8

* Khóa của các đỉnh 5 và 8 thành 2 và 6



Tiếp túc các bước chọn cạnh như vậy cho đến khi tất cả các đỉnh của đồ thị nằm trong mstSet

Cuối cùng ta thu được cây bao trùm nhỏ nhất



 Ứng dụng

◦ Thiết kế mạng: điện thoại, lưới điện, cáp TV, mạng máy tính, đường xá, ...

◦ Sử dụng trong các thuật toán tìm nghiệm xấp xỉ giải các bài toán NP-khó như bài toán người du lịch, cây steiner nhỏ nhất, ...

**Chương 5: Xâu kí tự**

**\*) Thuật toán sắp xếp sâu**

**- Phương pháp LDS(least significant – digit first)**

◦ Đặc điểm:

 Thuật ngữ digit được sử dụng thay cho character do xâu ký tự được mã hóa dựa trên bảng mã số (bảng mã ASCII có 256 số)

 Xem xét các ký tự từ phải qua trái (xem xét từ ít quan trọng trước)

 Chỉ áp dụng để sắp xếp các xâu ký tự có độ dài bằng nhau

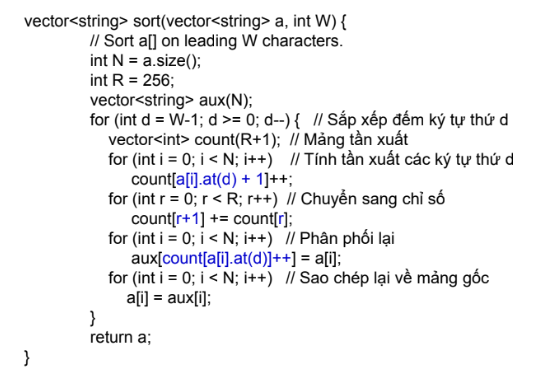
◦ Các bước: sắp xếp lần lượt từng ký tự từ phải qua trái sử dụng phương pháp sắp xếp đếm phân phối (counting sort):

 Tính tần xuất của các ký tự cùng ở vị trí d trong các xâu và lưu vào mảng count[]

 Chuyển các tần xuất sang chỉ số mảng bằng cách cộng dồn các tần xuất từ trái qua phải của mảng count[]

 Phân phối lại các xâu ký tự ban đầu vào mảng trung gian theo chỉ số mảng count[] thu được ở bước trên

 Sao chép lại mảng trung gian vào mảng ban đầu



◦ Phân tích:

 Sử dụng ~7WN + 3WR lần truy cập mảng, trong đó W là độ dài của một xâu, N là số xâu, R là kích thước bảng mã ASCII

 Không gian sử dụng thêm là N + R

 Độ phức tạp về thời gian: NW

**-Phương pháp MSD(most significant- digit first)**

◦ Sắp xếp các xâu ký tự có độ dài khác nhau

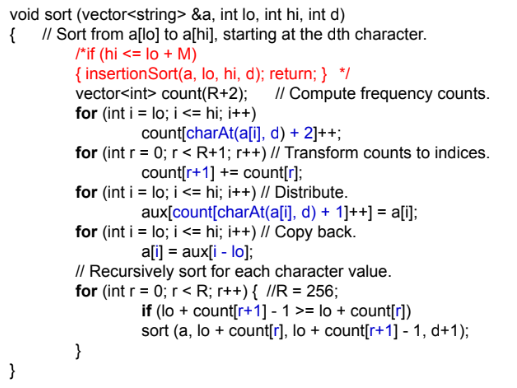
◦ Các ký tự được xem xét từ trái qua phải

◦ Ý tưởng:

 Xâu bắt đầu với ký tự a sẽ đứng trước xâu bắt bầu bằng ký tự b, ...

 Sử dụng phương pháp sắp xếp đếm phân phối (counting sort) để sắp xếp ký tự đầu tiên của các xâu, sau đó sắp xếp đệ quy trên các mảng con ứng với mỗi ký tự đầu tiên (các xâu của mảng con đã loạt bỏ ký tự đầu tiên)

 MSD phân hoạch mảng thành các mảng con để có thể sắp xếp chúng độc lập



◦ Phân tích:

 Thuật toán MSD phụ thuộc vào số ký tự trong bảng ký tự, ta đang xem xét bảng ASCII với 256 ký tự. Nếu xem xét bảng ký tự Unicode thì có 65536 ký tự  MSD ngốn nhiều thời gian và không bộ nhớ.

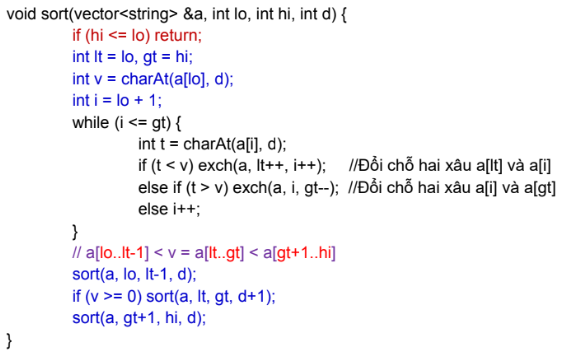
 Việc chia nhỏ mảng thành các mảng con đem lại hiệu quả vì chỉ thực hiện sắp xếp trên các mảng con đó. Tuy nhiên, khi các mảng con quá nhỏ sẽ ảnh hưởng đến hiệu suất của thuật toán MSD

 Các xâu con giống nhau (equal keys): số lượng lớn sẽ ảnh hưởng đến hiệu suất của MSD do phải gọi đệ quy không cần thiết cho mọi ký tự của các xâu con giống nhau này. Trường hợp xấu nhất đối với MSD là tất các cả các xâu giống nhau

**-Sắp xếp nhanh(3 way string quick sort)**

◦ Phân hoạch ba đường dựa trên ký tự đầu tiên của các chuỗi trong mảng

◦ Chỉ chuyển đến ký tự tiếp theo trên mảng con ở giữa (ký tự đầu tiên bằng với ký tự dùng để phân hoạch)



◦ Đặc điểm

 Chia mảng thành ba phần nên hạn chế được số lượng phân hoạch rỗng

 Xử lý tốt trường hợp có nhiều xâu bằng nhau hoặc có tiền tố chung dài hoặc trên vùng có nhiều mảng con

 Không sử dụng không gian dư thừa cho bảng mã hóa ký tự

**\*) Cây tiền tố**

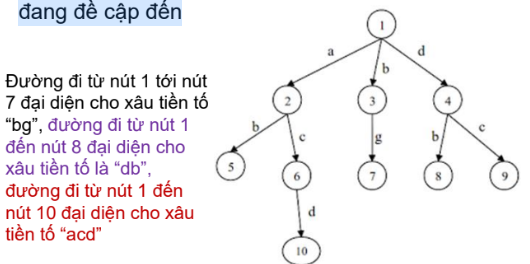
◦ Xử lý tiền tố trong một tập các xâu là một trong những vấn đề thường được đề cập trong các bài toán xử lý xâu

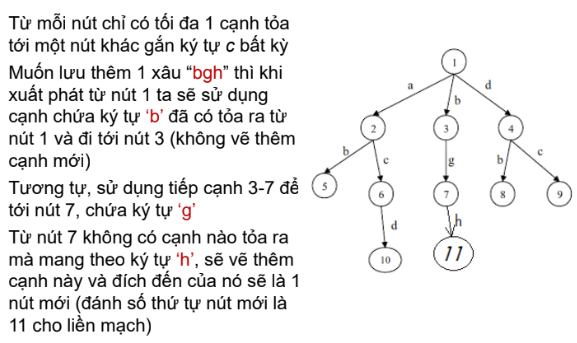
◦ Trong nhiều trường hợp, cấu trúc cây tiền tố (Trie) được coi là một giải pháp hiệu quả

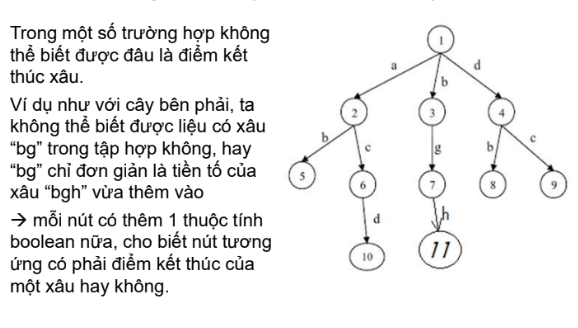
◦ Bản chất của cấu trúc Trie là một cây có gốc

◦ Nút gốc không chứa thông tin, nhưng mỗi cạnh nối hai nút trên cây tương ứng với một ký tự

◦ Đường đi từ nút gốc tới một nút bất kỳ trên cây này cho biết tập đang xét có chứa xâu tiền tố biểu diễn bởi tập các cạnh thể hiện đường đi từ nút gốc tới nút đang đề cập đến







◦ Có ba thao tác chính

 Thêm một xâu S vào cây. Độ phức tạp là O(|S|)

 Xóa một xâu S khỏi cây. Độ phức tạp là O(|S|)

 Kiểm tra xem một xâu S có tồn tại trong tập hợp dưới dạng một xâu hoàn chỉnh hoặc một tiền tố hay không. Độ phức tạp O(|S|)

◦ Ưu điểm

 Cài đặt đơn giản, dễ nhớ

 Tiết kiệm bộ nhớ: Khi số lượng khóa lớn và các khóa có độ dài nhỏ thì trie tiết kiệm bộ nhớ hơn do các phần đầu giống nhau của các khoá chỉ được lưu một lần. Ưu điểm này có ứng dụng rất lớn, chẳng hạn trong từ điển

 Dựa vào tính chất của cây trie, có thể thực hiện một số thao tác liên quan đến thứ tự từ điển như sắp xếp, tìm một khóa có thứ tự từ điển nhỏ nhất và lớn hơn một khóa cho trước, ...



