Group 5

MERGE SORT
&
PRIORITY QUEUE

I. Merge Sort

1. Định nghĩa:

- Merge Sort là một thuật toán sắp xếp đệ quy
- Merge Sort hoạt động bằng cách chia mảng thành các phần nhỏ hơn sắp xếp từng phần đó rồi kết hợp chúng lại để có một dãy đã được sắp xếp

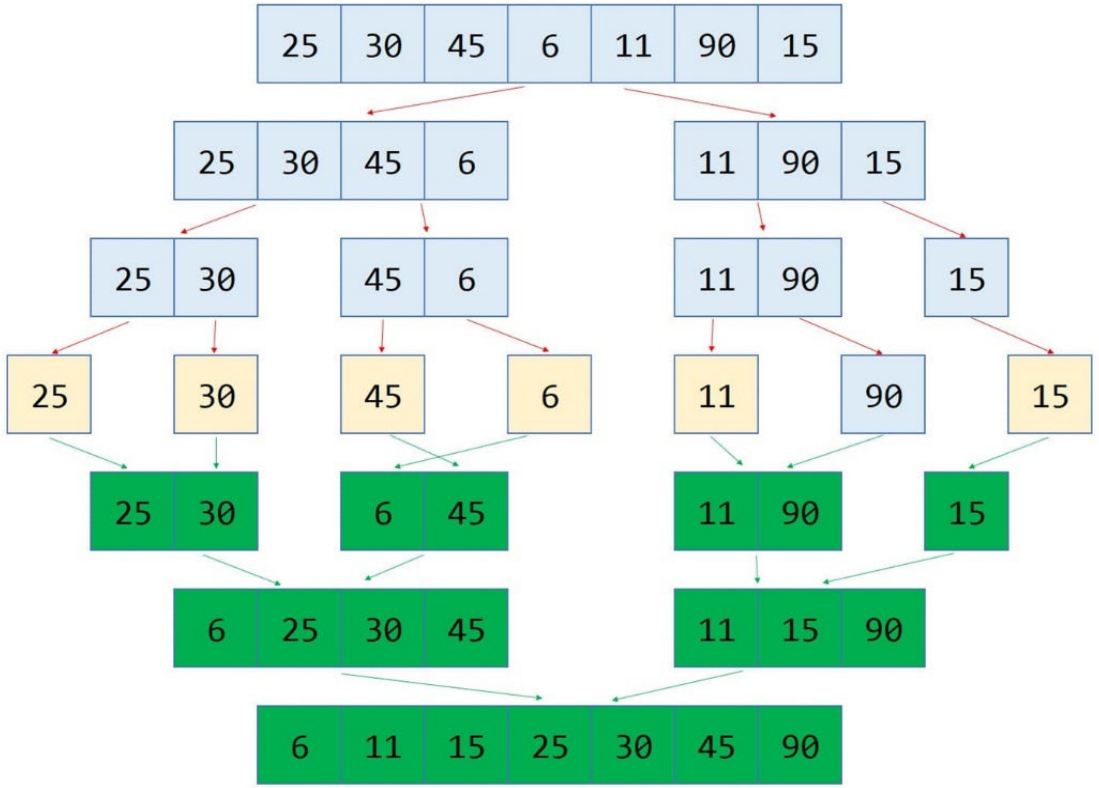
I. Merge Sort

1. Định nghĩa:

- Merge Sort là là một thuật toán sắp xếp đệ quy
- Merge Sort hoạt động bằng cách chia mảng thành các phần nhỏ hơn sắp xếp từng phần đó rồi kết hợp chúng lại để có một dãy đã được sắp xếp
- Các bước triển khai thuật toán Merge Sort:

I. Merge Sort

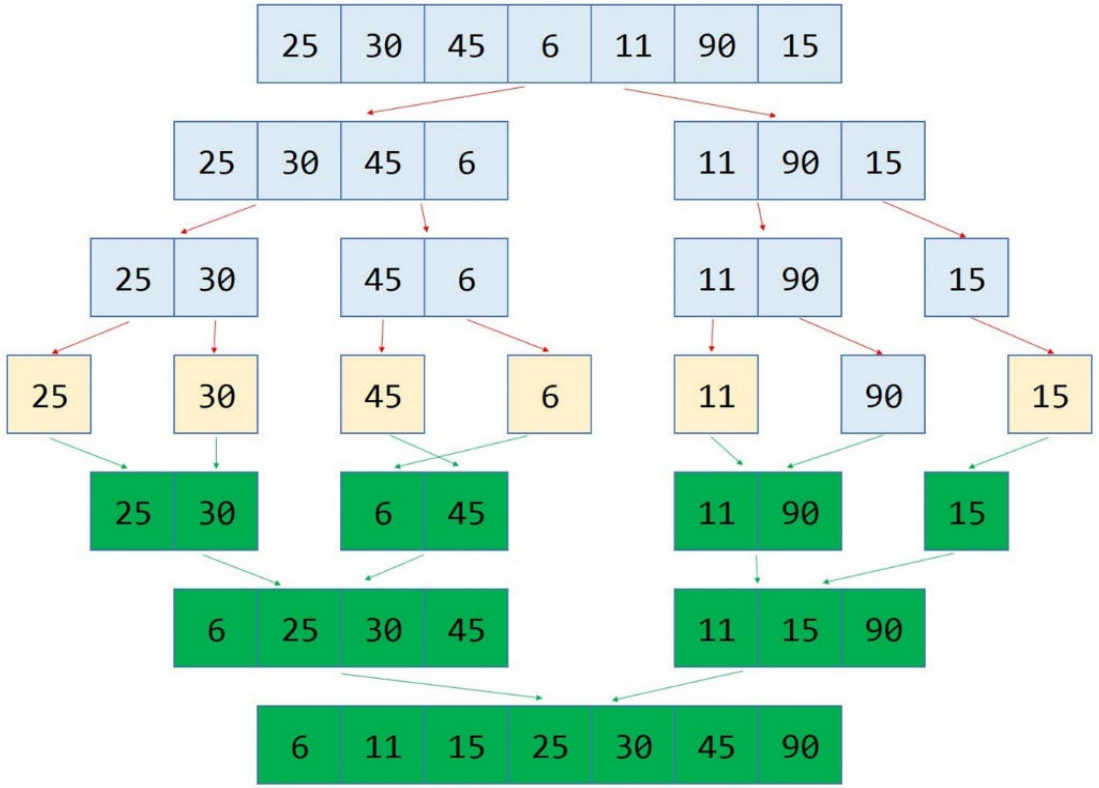
❖ Ví dụ minh họa:



- Độ phức tạp: **O(N log(N))**

I. Merge Sort

❖ Ví dụ minh họa:

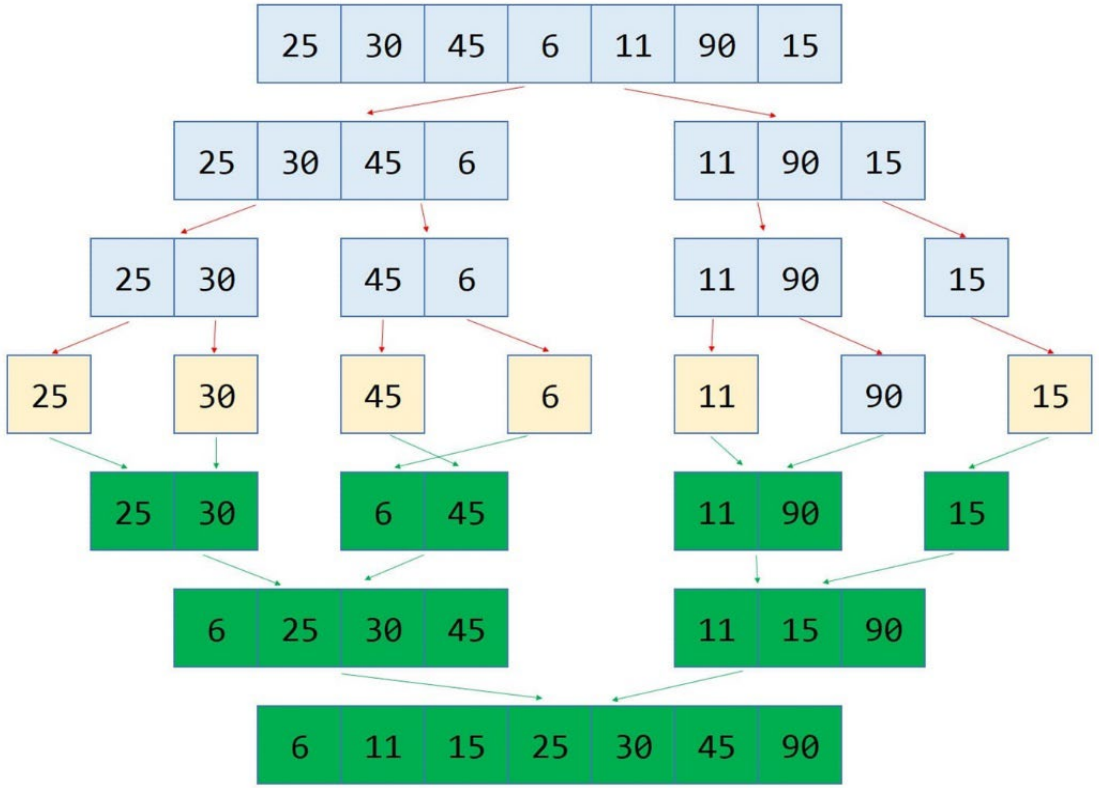


- Mảng ban đầu được lặp lại hành động chia cho tới khi kích thước các mảng sau chia là 1

▪ Độ phức tạp: **O(N log(N))**

I. Merge Sort

❖ Ví dụ minh họa:

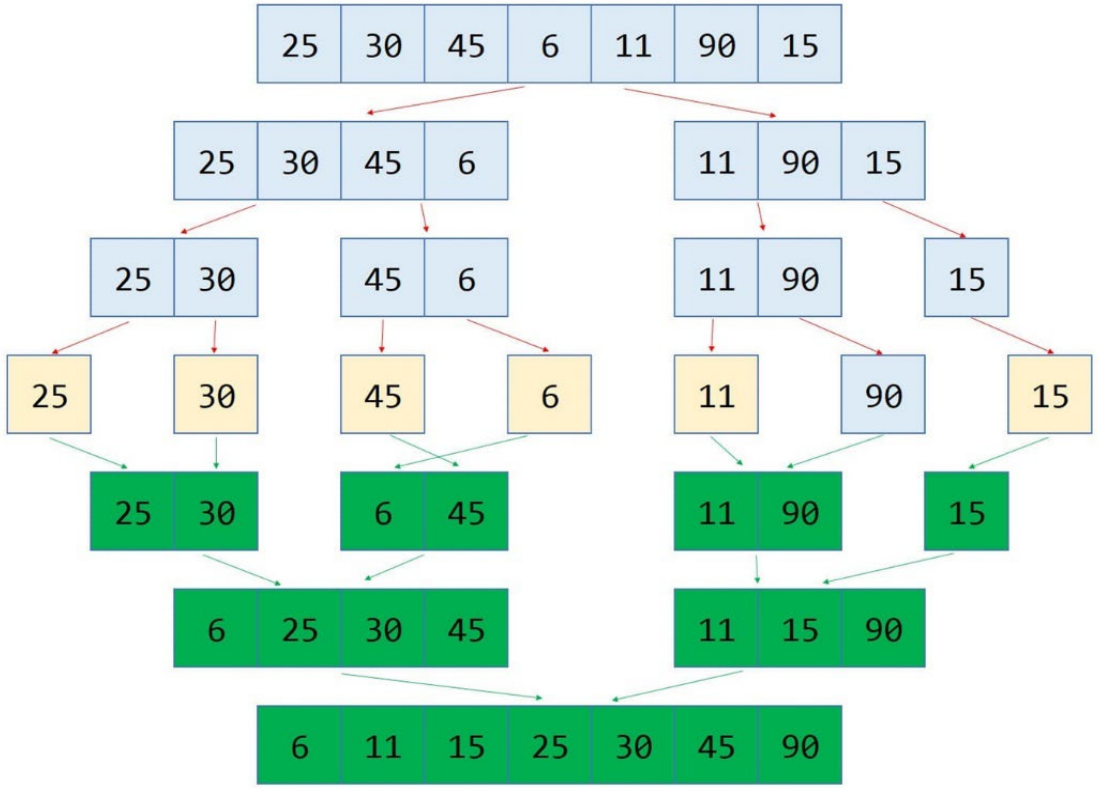


- Mảng ban đầu được lặp lại hành động chia cho tới khi kích thước các mảng sau chia là 1
- Khi kích thước các mảng con là 1, tiến trình gộp sẽ bắt đầu

- Độ phức tạp: **O(N log(N))**

I. Merge Sort

❖ Ví dụ minh họa:



- Mảng ban đầu được lặp lại hành động chia cho tới khi kích thước các mảng sau chia là 1
- Khi kích thước các mảng con là 1, tiến trình gộp sẽ bắt đầu
- Thực hiện gộp lại các mảng này cho tới khi hoàn thành và chỉ còn một mảng đã sắp xếp

▪ Độ phức tạp: **O(N log(N))**

I. Merge Sort

❖ **Triển khai thuật toán Merge Sort trong C++:**

```
6  void merge(int a[], int l, int m, int r){
7      vector<int> x(a + l, a + m + 1);
8      vector<int> y(a + m + 1, a + r + 1);
9      int i = 0, j = 0;
10     while(i < x.size() && j < y.size()){
11         if(x[i] <= y[j]){
12             a[l] = x[i]; ++l; ++i;
13         }
14         else{
15             a[l] = y[j]; ++l; ++j;
16         }
17     }
18     while(i < x.size()){
19         a[l] = x[i]; ++l; ++i;
20     }
21     while(j < y.size()){
22         a[l] = y[j]; ++l; ++j;
23     }
24 }
```

I. Merge Sort

❖ **Triển khai thuật toán Merge Sort trong C++:**

```cpp
void merge(int a[], int l, int m, int r){
    vector<int> x(a + l, a + m + 1);
    vector<int> y(a + m + 1, a + r + 1);
    int i = 0, j = 0;
    while(i < x.size() && j < y.size()){
        if(x[i] <= y[j]){
            a[l] = x[i]; ++l; ++i;
        }
        else{
            a[l] = y[j]; ++l; ++j;
        }
    }
    while(i < x.size()){
        a[l] = x[i]; ++l; ++i;
    }
    while(j < y.size()){
        a[l] = y[j]; ++l; ++j;
    }
}
```

```cpp
    while(i < x.size()){
        a[l] = x[i]; ++l; ++i;
    }
    while(j < y.size()){
        a[l] = y[j]; ++l; ++j;
    }
}

void mergeSort(int a[], int l, int r){
    if(l >= r) return;
    int m = (l + r) / 2;
    mergeSort(a, l, m);
    mergeSort(a, m + 1, r);
    merge(a, l, m ,r);
}
```

I. Merge Sort

❖ Ưu điểm của Merge Sort:

- **Tính ổn định** : Merge sort là một thuật toán sắp xếp ổn định, có nghĩa là nó duy trì thứ tự tương đối của các phần tử bằng nhau trong mảng đầu vào

I. Merge Sort

❖ Ưu điểm của Merge Sort:

- **Tính ổn định** : Merge sort là một thuật toán sắp xếp ổn định, có nghĩa là nó duy trì thứ tự tương đối của các phần tử bằng nhau trong mảng đầu vào

- **Hiệu suất được đảm bảo trong trường hợp xấu nhất** : Merge sort có độ phức tạp về thời gian trong cả trường hợp xấu nhất là O(n log(n)), có nghĩa là nó hoạt động tốt ngay cả trên tập dữ liệu lớn

I. Merge Sort

❖ Ưu điểm của Merge Sort:

- **Tính ổn định** : Merge sort là một thuật toán sắp xếp ổn định, có nghĩa là nó duy trì thứ tự tương đối của các phần tử bằng nhau trong mảng đầu vào

- **Hiệu suất được đảm bảo trong trường hợp xấu nhất** : Merge sort có độ phức tạp về thời gian trong cả trường hợp xấu nhất là O(n log(n)), có nghĩa là nó hoạt động tốt ngay cả trên tập dữ liệu lớn

- **Có thể song hóa** : Merge sort là một thuật toán có khả năng song song hóa một cách tự nhiên, có nghĩa là nó có thể dễ dàng song song hóa để tận dụng nhiều bộ xử lý hoặc luồng

I. Merge Sort

❖ Hạn chế của Merge Sort:

- **Độ phức tạp về không gian** : Merge sort yêu cầu bộ nhớ bổ sung để lưu trữ các mảng con đã hợp nhất trong quá trình sắp xếp

I. Merge Sort

❖ Hạn chế của Merge Sort:

- **Độ phức tạp về không gian** : Merge sort yêu cầu bộ nhớ bổ sung để lưu trữ các mảng con đã hợp nhất trong quá trình sắp xếp

- **Không đúng chỗ** : Merge sort không phải là thuật toán sắp xếp tại chỗ, có nghĩa là nó cần thêm bộ nhớ để lưu trữ dữ liệu đã sắp xếp

I. Merge Sort

❖ Hạn chế của Merge Sort:

- **Độ phức tạp về không gian** : Merge sort yêu cầu bộ nhớ bổ sung để lưu trữ các mảng con đã hợp nhất trong quá trình sắp xếp

- **Không đúng chỗ** : Merge sort không phải là thuật toán sắp xếp tại chỗ, có nghĩa là nó cần thêm bộ nhớ để lưu trữ dữ liệu đã sắp xếp

- **Không phải lúc nào cũng tối ưu cho các tập dữ liệu nhỏ** : Đối với các tập dữ liệu nhỏ, merge sort có độ phức tạp về thời gian cao hơn một số thuật toán sắp xếp khác, chẳng hạn như *insertion sort (sắp xếp chèn)*. Điều này có thể dẫn đến hiệu suất chậm hơn đối với các tập dữ liệu rất nhỏ

I. Merge Sort

II. Priority Queue

1. Định nghĩa:

- **Hàng đợi ưu tiên:** là loại hàng đợi sắp xếp các phần tử *dựa trên giá trị ưu tiên* của chúng. Các phần tử có giá trị ưu tiên cao hơn thường được truy xuất trước các phần tử có giá trị ưu tiên thấp hơn.

II. Priority Queue

1. Định nghĩa:

- **Hàng đợi ưu tiên:** là loại hàng đợi sắp xếp các phần tử *dựa trên giá trị ưu tiên* của chúng. Các phần tử có giá trị ưu tiên cao hơn thường được truy xuất trước các phần tử có giá trị ưu tiên thấp hơn.

- Trong hàng đợi ưu tiên, mỗi phần tử có một giá trị ưu tiên gắn liền với nó. Khi bạn thêm một phần tử vào hàng đợi, nó sẽ được chèn vào vị trí dựa trên giá trị ưu tiên của nó.

II. Priority Queue

1. Định nghĩa:

- **Hàng đợi ưu tiên:** là loại hàng đợi sắp xếp các phần tử *dựa trên giá trị ưu tiên* của chúng. Các phần tử có giá trị ưu tiên cao hơn thường được truy xuất trước các phần tử có giá trị ưu tiên thấp hơn.

- Trong hàng đợi ưu tiên, mỗi phần tử có một giá trị ưu tiên gắn liền với nó. Khi bạn thêm một phần tử vào hàng đợi, nó sẽ được chèn vào vị trí dựa trên giá trị ưu tiên của nó.

- Một số cách triển khai hàng đợi ưu tiên, bao gồm sử dụng: Mảng, danh sách liên kết, vùng heap, cây tìm kiếm nhị phân.

II. Priority Queue

❖ Bài toán: Cho *n* ván gỗ có chiều dài lần lượt là $a_1, a_2, \ldots, a_n$ đơn vị độ dài $0 < a_i < 10^5$. Ta có thể *ghép* hai ván gỗ có chiều dài a_i và a_j thành một ván gỗ mới có chiều dài là $a_i + a_j$ và tốn chi phí là $a_i + a_j$. Hỏi *chi phí nhỏ nhất* để ghép ván gỗ đã cho thành một ván gỗ duy nhất là bao nhiêu?

Ví dụ:

Input	Output
6 1 2 8 2 7 6	60

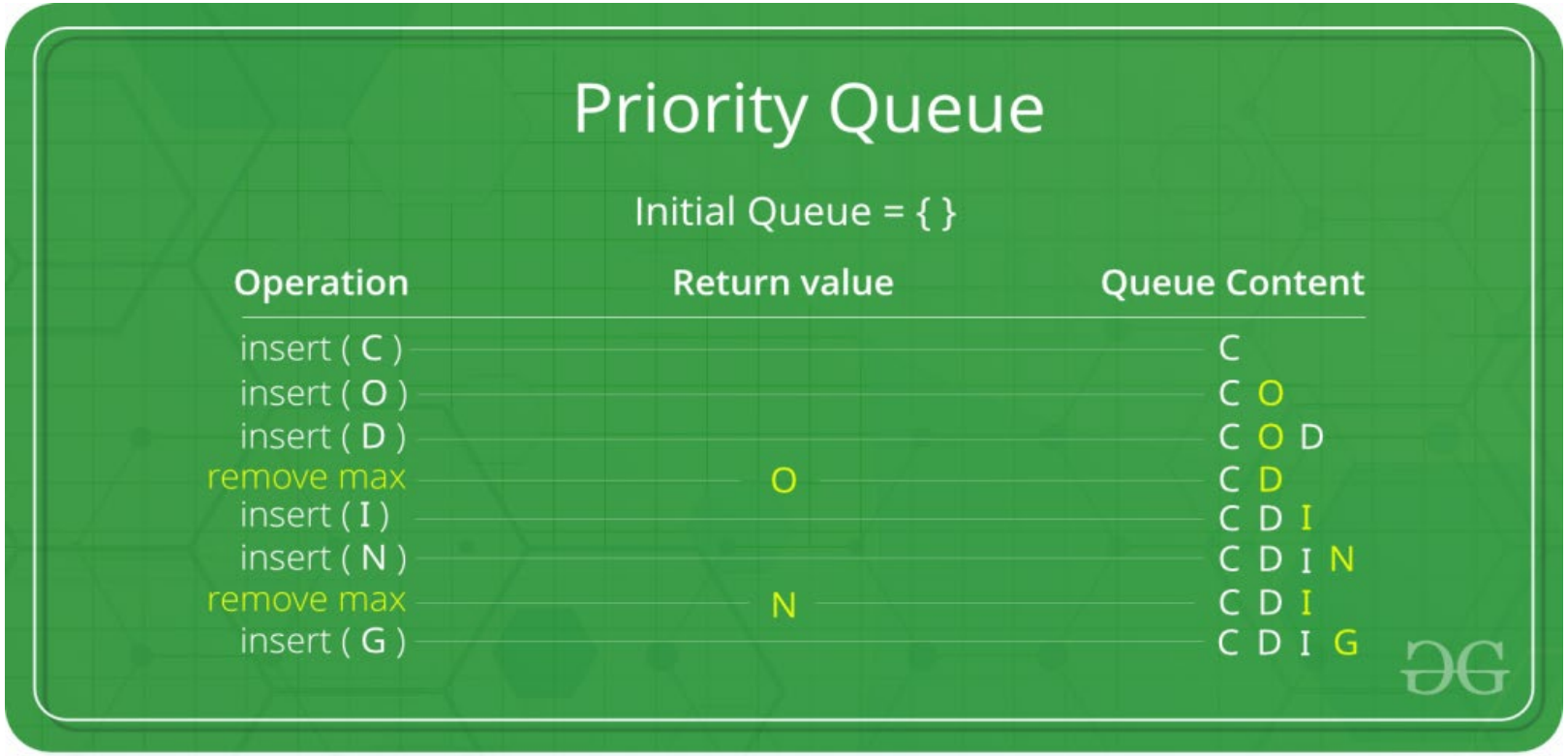
2. Thuộc tính của hàng đợi ưu tiên:

Hàng đợi ưu tiên là phần mở rộng của hàng đợi với các thuộc tính sau:

- Mỗi mục có một mức độ ưu tiên liên quan đến nó.
- Phần tử có mức độ ưu tiên cao sẽ được xếp hàng trước phần tử có mức độ ưu tiên thấp.
- Nếu hai phần tử có cùng mức độ ưu tiên thì chúng sẽ được phân phát theo thứ tự trong hàng đợi.

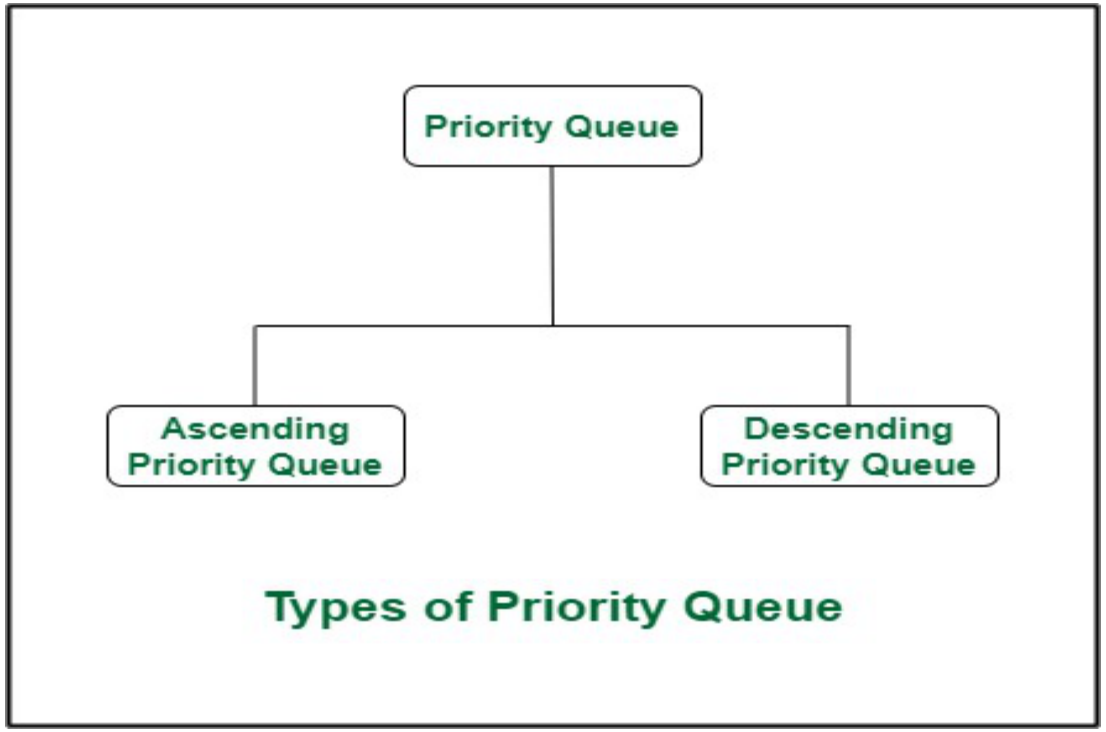
2. Thuộc tính của hàng đợi ưu tiên:

Trong hàng đợi ưu tiên bên dưới, **phần tử có giá trị ASCII tối đa sẽ có mức độ ưu tiên cao nhất**. Các phần tử có mức độ ưu tiên cao hơn sẽ được phục vụ trước.



3. Các loại hàng đợi ưu tiên:

- Hàng đợi ưu tiên **thứ tự giảm dần**
- Hàng đợi ưu tiên **thứ tự tăng dần**



Các loại hàng đợi ưu tiên

4. Cài đặt và thao tác đối với hàng đợi ưu tiên:

❖ Cài đặt:

```
#include <queue>

priority_queue<Type, Container, Functional>
```

Trong đó:

- Type chỉ kiểu dữ liệu
- **Container** chỉ kiểu dữ liệu chứa (bắt buộc là kiểu mảng)
- Tham trị **Functional** là phần tham trị phụ chỉ thứ tự ưu tiên cho các phần tử (less, greater)

4. Cài đặt và thao tác đối với hàng đợi ưu tiên:

❖ Các thao tác:

- **empty()** : Thao tác này kiểm tra xem **Priority_Queue** trống hay không. Nếu nó trống, trả về **true**, ngược lại là **false**. Nó không chứa bất kì tham số nào.

```
q.empty()
```

- **size()** : Phương thức này cung cấp số phần tử của **Priority_queue**. Nó trả về giá trị là một số nguyên. Nó không chứa bất kỳ tham số nào.

```
q.size()
```

4. Cài đặt và thao tác đối với hàng đợi ưu tiên:

❖ Các thao tác:

- **push()** : Phương thức này chèn phần tử vào hàng đợi ưu tiên **Priority_Queue**. Đầu tiên, phần tử được thêm vào cuối hàng đợi và đồng thời các phần tử tự sắp xếp lại với mức độ ưu tiên. Nó có giá trị chứa trong tham số.

```
q.push(3) // chèn phần tử 3 vào q
```

- **pop()** : Phương thức này xóa phần tử trên cùng (mức độ ưu tiên cao nhất) khỏi hàng đợi ưu tiên. Nó không có bất kỳ tham số nào.

```
q.pop()
```

4. Cài đặt và thao tác đối với hàng đợi ưu tiên:

❖ Các thao tác:

- **top()** : Phương thức này cung cấp phần tử trên cùng từ vùng chứa hàng đợi ưu tiên. Nó không có bất kỳ tham số nào.

```
q.top()
```

- **swap()** : Phương thức này hoán đổi các phần tử của một **Priority_queue** với một **Priority_queue** (có cùng kích thước và kiểu). Nó nhận hàng đợi ưu tiên trong một tham số có các giá trị cần được hoán đổi.

```
q.swap(qu)
```

4. Cài đặt và thao tác đối với hàng đợi ưu tiên:

❖ Các thao tác:

- **emplace()** : Phương thức này thêm một phần tử mới vào một vùng chứa ở đầu hàng đợi ưu tiên. Nó nhận giá trị trong một tham số.

```
q.emplace(3) // thêm phần tử 3
```

5. Ứng dụng:

- Lập lịch CPU
- Triển khai ngăn xếp
- Nén dữ liệu bằng mã Huffman
- Mô phỏng theo sự kiện, chẳng hạn như khách hàng đang xếp hàng chờ.
- Tìm phần tử lớn nhất/nhỏ thứ K.
- Các thuật toán đồ thị như thuật toán Đường đi ngắn nhất của Dijkstra,

  Cây kéo dài tối thiểu của Brim,…

6.1. Ưu điểm của hang đợi ưu tiên:

- Giúp truy cập các phần tử một cách nhanh hơn
- Việc sắp xếp các phần tử trong Hàng đợi ưu tiên được thực hiện một cách linh hoạt
- Các thuật toán hiệu quả có thể được thực hiện, chẳng hạn như thuật toán Dijkstra
- Bao gồm trong các hệ thống thời gian thực

6.2. Nhược điểm của hang đợi ưu tiên:

- Độ phức tạp cao
- Tiêu thụ bộ nhớ cao
- Nó không phải lúc nào cũng là cấu trúc dữ liệu hiệu quả nhất
- Thứ tự các phần tử được truy xuất có thể khó dự đoán hơn so với các cấu trúc dữ liệu khác

A
MINION
thanks!

Thank you for listening