TEAM 5

23520277: Trương Trọng Đạt
23520743: Lý Đăng Khoa
23520941: Nguyễn Nhữ Hoàng Minh

- Mergesort
- Cấu trúc priority queue (dùng heap)
- Bài tập

01

Merge Sort

02

Priority queue

03

Quiz

01

Merge Sort
(Sắp xếp trộn)

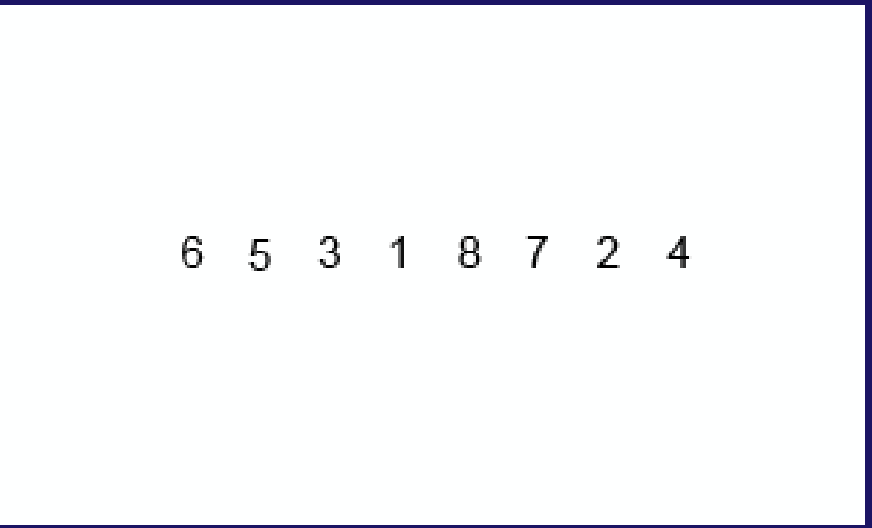
Giới Thiệu

-Được sáng lập vào 1945 bởi John von Neumann

-Hiệu suất cao và thông dụng

-Dựa vào lối thuật toán “chia để trị”

Ý Tưởng Thuật Toán



-Chia dãy chưa được sắp xếp thành các dãy con, mỗi dãy chứa một phần tử

-Liên tục ghép các dãy con để tạo các dãy con đã được sắp xếp cho đến khi chỉ còn một dãy, đó là dãy đã được sắp xếp

Các Bước Của Thuật Toán

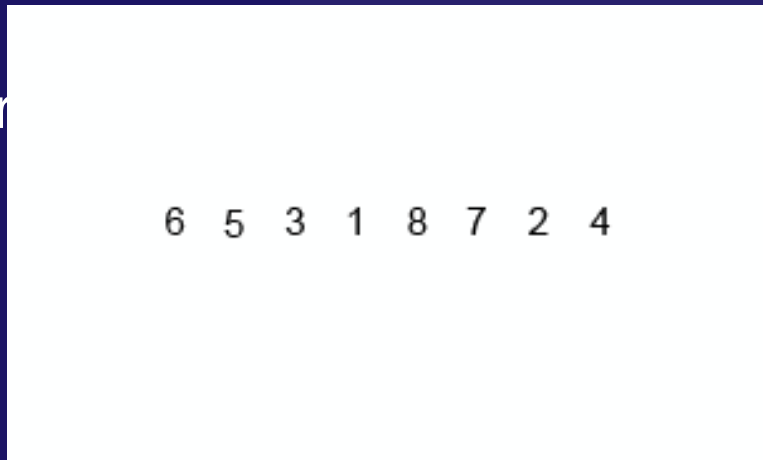
Step 1 Bắt đầu

Step 2 Tìm vị trí l,m,r của mảng

Step 3 Nếu mảng có nhiều hơn 1 phần tử:

```
m = (l+r)/2;
mergeSort(arr, l, m);
mergeSort(arr, m+1, r
merge(arr, l, m, r);
```

Step 4 Kết thúc



Code Của MergeSort

Phần Ghép (merge):

```
void merge(int arr[], int const l, int const m, int const r)
```

Bước 1:

```
int i, j, k;

//khởi tạo 2 mảng L <- arr[l...m] và R <- arr[m+1...r]
int const n1 = m - l + 1;
int const n2 = r - m;
auto *L= new int[n1], *R= new int[n2];

//copy vào 2 mảng L[n1], R[n2]
for (i = 0; i < n1; i++)
    L[i] = arr[l + i];
for (j = 0; j < n2; j++)
    R[j] = arr[m + 1 + j];
```

Code Của MergeSort

Bước 2:

```
i = 0; // vị trí đầu của mảng L[]
j = 0; // vị trí đầu của mảng R[]
k = l; // vị trí đầu của mảng arr[l...r]
```

Bước 3:

```
while (i < n1 && j < n2) {
    if (L[i] <= R[j]) {
        arr[k] = L[i];      i++;
    }
    else {
        arr[k] = R[j];      j++;
    }
    k++;
}
```

Code Của MergeSort

Bước 4:

```
//Copy những phần tử còn lại vào arr[]
while (i < n1)
{
    arr[k] = L[i];
    i++;
    k++;
}
while (j < n2)
{
    arr[k] = R[j];
    j++;
    k++;
}
```

Ví Dụ Quá Trình Ghép (Merge)


```
void merge(int arr[], int const l, int const m, int const r)
//l=0, m=3, r=5
```

Ví Dụ Quá Trình Ghép (Merge)

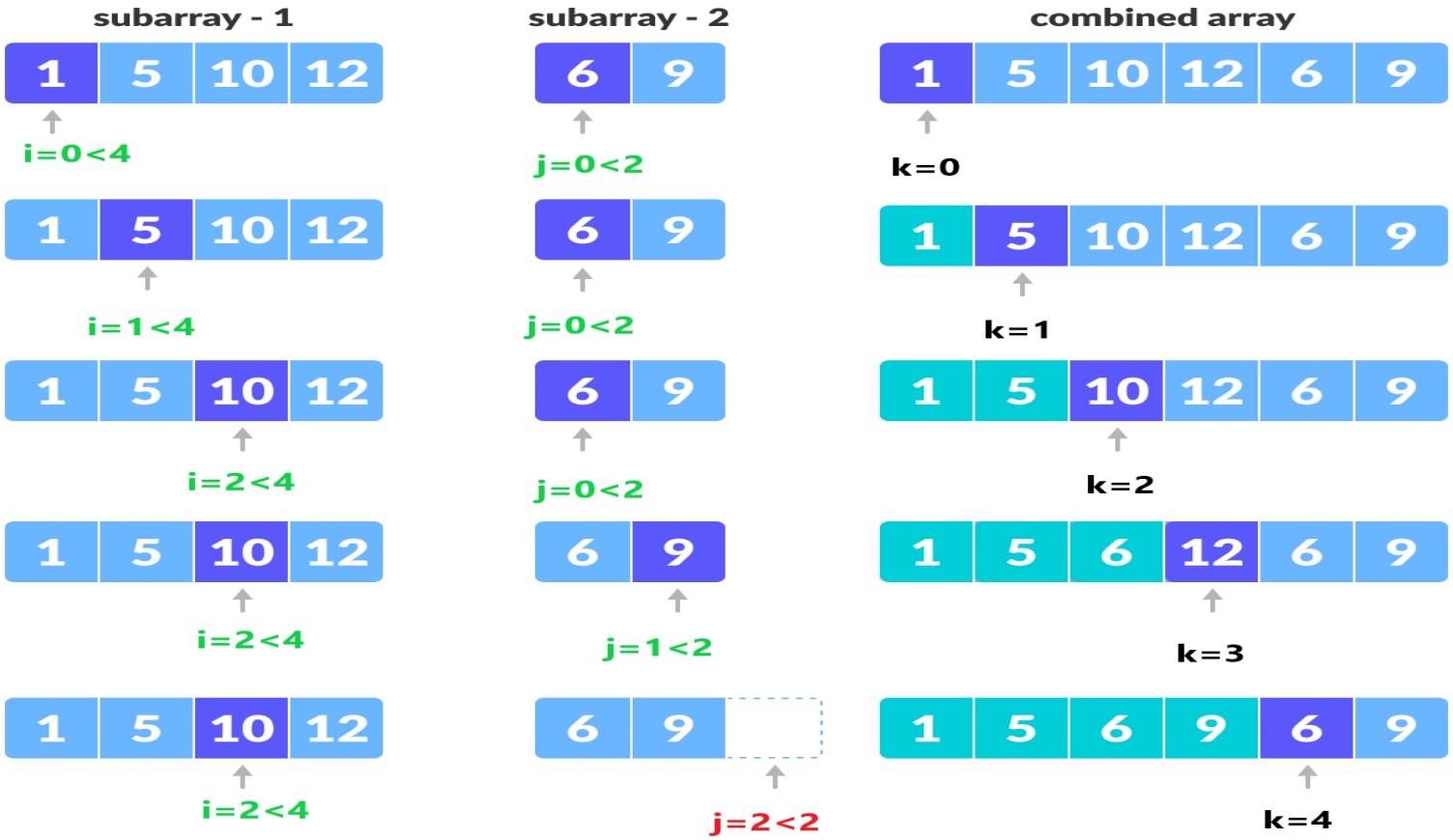
```
int const n1 = m - l + 1;              //3 - 0 + 1 = 4
int const n2 = r - m;                  //5 - 3 = 2
auto *L= new int[n1], *R= new int[n2]; //L[4] , R[2]

//copy vào 2 mảng L[n1], R[n2]
for (i = 0; i < n1; i++)
    L[i] = arr[l + i];      //L[0,1,2,3] = A[0,1,2,3] = [1,5,10,12]
for (j = 0; j < n2; j++)
    R[j] = arr[m + 1 + j]; //R[0,1] = A[4,5] = [6,9]
```

Bước 1:

A	1	5	10	12	6	9
L	1	5	10	12		
R	6	9				

Ví Dụ Quá Trình Ghép (Merge)

Bước 2:



Ví Dụ Quá Trình Ghép (Merge)

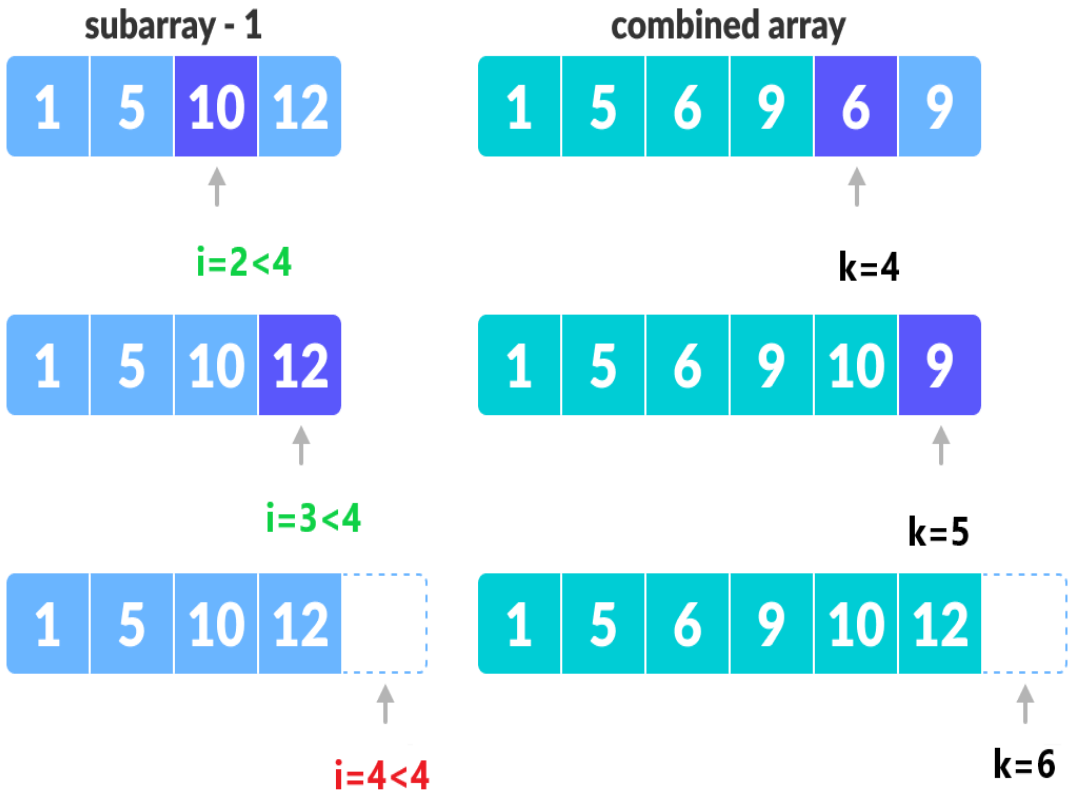
Bước 2:

```
while (i < n1 && j < n2) {
    if (L[i] <= R[j]) {
        arr[k] = L[i];      i++;
    }
    else {
        arr[k] = R[j];      j++;
    }
    k++;
}
```

Ví Dụ Quá Trình Ghép (Merge)

Bước 3:

```
while (i < n1)
{
    arr[k] = L[i];
    i++;
    k++;
}
```



Ví Dụ Quá Trình Ghép (Merge)

Bước 3:

```
void mergeSort(int arr[], int l, int r)
{
    if (l < r)
    {
        int m = (l+r) / 2;

        mergeSort(arr, l, m);
        mergeSort(arr, m + 1, r);
        merge(arr, l, m, r);
    }
}
```

Ví Dụ Quá Trình Ghép (Merge)

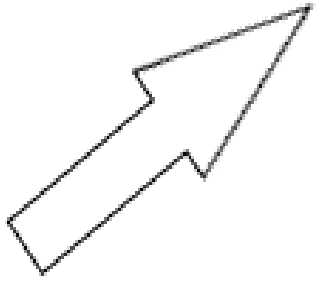
A[0...5]

```
57  void mergeSort(int arr[], int l, int r)
58  {                                   //l=0    r=5
59      if (l < r)
60      {
61          int m = (l+r) / 2; //m=( 0 + 5 ) / 2 = 2
62
63          mergeSort(arr, l, m); //A[0...2]
64          mergeSort(arr, m + 1, r);
65          merge(arr, l, m, r);
66      }
67  }
```

Ví Dụ Quá Trình Ghép (Merge)

A[0...2]

```
57  void mergeSort(int arr[], int l, int r)
58  {                                   //l=0    r=2
59      if (l < r)
60      {
61          int m = (l+r) / 2; //m=( 0 + 2 ) / 2 = 1
62
63          mergeSort(arr, l, m); //A[0...1]
64          mergeSort(arr, m + 1, r);
65          merge(arr, l, m, r);
66      }
```



A[0...5]

```
63          mergeSort(arr, l, m); //A[0...2]
64          mergeSort(arr, m + 1, r);
65          merge(arr, l, m, r);
```

Ví Dụ Quá Trình Ghép (Merge)

A[0...1]

```
57  void mergeSort(int arr[], int l, int r)
58  {                                   //l=0    r=1
59      if (l < r)
60      {
61          int m = (l+r) / 2; //m=( 0 + 1 ) / 2 = 0
62
63          mergeSort(arr, l, m); //A[0...0]
64          mergeSort(arr, m + 1, r);
65          merge(arr, l, m, r);
66      }
67  }
```

A[0...2]

```
63          mergeSort(arr, l, m); //A[0...1]
64          mergeSort(arr, m + 1, r);
65          merge(arr, l, m, r);
```

A[0...5]

Ví Dụ Quá Trình Ghép (Merge)

~~A[0...0]~~ -->2

```
57  void mergeSort(int arr[], int l, int r)
58  {                                   //l=0    r=0
59      if (l < r)  //(0 < 0 = false)
```

A[0...1]

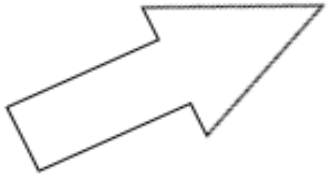
```
63          mergeSort(arr, l, m); //A[0...0]
64          mergeSort(arr, m + 1, r);
65          merge(arr, l, m, r);
```

A[0...2]

...

Ví Dụ Quá Trình Ghép (Merge)

~~A[1...1]~~ --> 0



```
57  void mergeSort(int arr[], int l, int r)
58  {                                   //l=1    r=1
59      if (l < r)  //(1 < 1 = false)
```

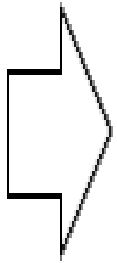
A[0...1]

```
57  void mergeSort(int arr[], int l, int r)
58  {                                   //l=0    r=1
59      if (l < r)
60      {
61          int m = (l+r) / 2; //m=( 0 + 1 ) / 2 = 0
62
63          mergeSort(arr, l, m); //A[0...0]
64          mergeSort(arr, m + 1, r); //A[1...1]
65          merge(arr, l, m, r);
66      }
67  }
```

Ví Dụ Quá Trình Ghép (Merge)

A[0...1]

```
63      mergeSort(arr, l, m); //A[0...0]
64      mergeSort(arr, m + 1, r); //A[1...1]
65      merge(arr, l, m, r);// (arr , 0 , 0 , 1)
```



[0 , 2]

Ví Dụ Quá Trình Ghép (Merge)

A[0...1] --> [0 , 2]

A[0...2]

```
57  void mergeSort(int arr[], int l, int r)
58  {                                //l=0    r=2
59      if (l < r)
60      {
61          int m = (l+r) / 2; //m=( 0 + 2 ) / 2 = 1
62
63          mergeSort(arr, l, m); //A[0...1]
64          mergeSort(arr, m + 1, r); //A[2...2]
65          merge(arr, l, m, r);
```

A[2...2] --> 3

```
57  void mergeSort(int arr[], int l, int r)
58  {                                //l=2    r=2
59      if (l < r)  //(2 < 2 = false)
```

A[0...5]

Ví Dụ Quá Trình Ghép (Merge)

~~A[0...2]~~ --> [0 , 2, 3]

```
57  void mergeSort(int arr[], int l, int r)
58  {                                   //l=2    r=2
59      if (l < r)  //(2 < 2 = false)
60      {
61          int m = (l+r) / 2; //m=( 0 + 2 ) / 2 = 1
62
63          mergeSort(arr, l, m); //A[0...1]
64          mergeSort(arr, m + 1, r); //A[2...2]
65          merge(arr, l, m, r);  //(arr, 0 , 1 , 2)
66      }
```

A[0...5]

Ví Dụ Quá Trình Ghép (Merge)

A[0...5]

```
57  void mergeSort(int arr[], int l, int r)
58  {                                   //l=0   r=5
59      if (l < r)
60      {
61          int m = (l+r) / 2; //m=( 0 + 5 ) / 2 = 2
62
63          mergeSort(arr, l, m); //A[0...2]
64          mergeSort(arr, m + 1, r); //A[3...5]
65          merge(arr, l, m, r);
```

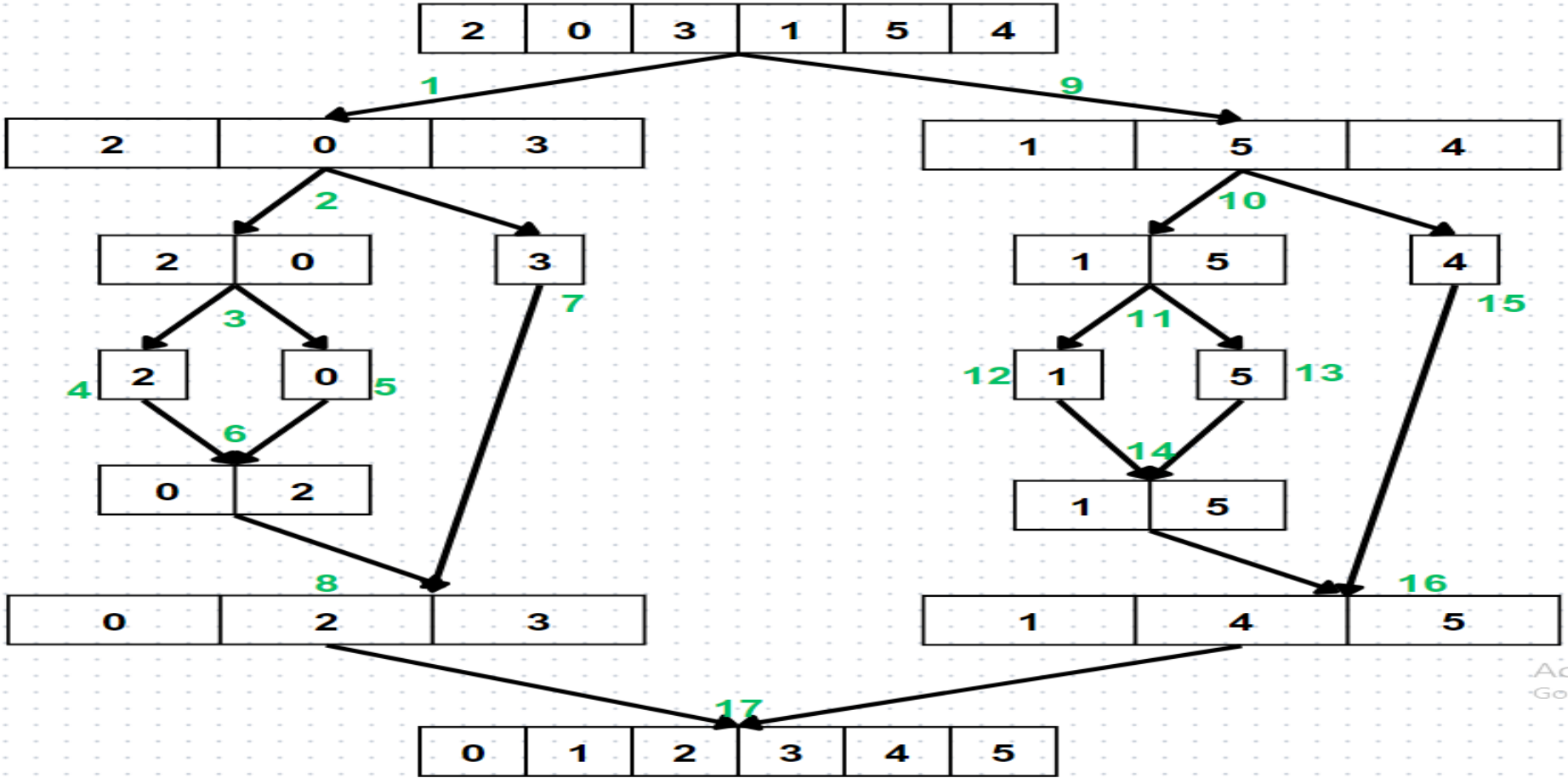
A[0...2] --> [0 , 2 , 3]

A[3...5]

...

...

Ví Dụ Quá Trình Ghép (Merge)



Ví Dụ Quá Trình Ghép (Merge)

```
void printArray(int arr[], int size)
{
    int i;
    for (i = 0; i < size; i++) {
        cout << arr[i] << " ";
    }
    cout << endl;
}
```

```
int main()
{
    int arr[] = {2,0,3,1,5,4};
    int arr_size = sizeof(arr) / sizeof(arr[0]);
    printArray(arr, arr_size); //in dãy vừa nhập

    mergeSort(arr, 0, arr_size - 1); // sắp xếp trộn dãy

    printArray(arr, arr_size);  // in dãy đã sắp xếp
```

Độ Phức Tạp

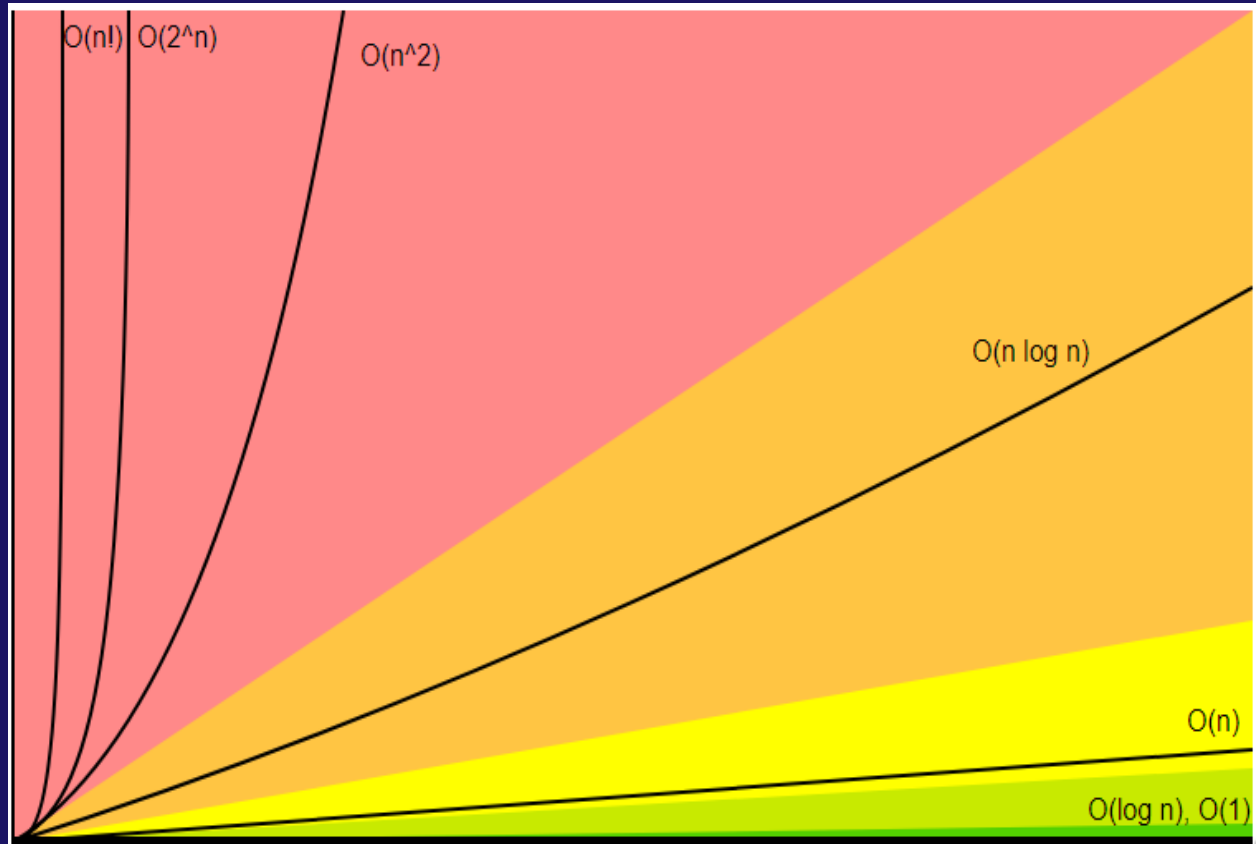
- Độ phức tạp thời gian: vì Merge Sort luôn phân chia mảng thành 2 phần với độ phức tạp O[log(n)] và cần O(n) để trộn chúng lại, nên:

+ Trường hợp tốt nhất: O(nlog(n)).

+ Trường hợp xấu nhất: O(nlog(n)).

+ Trường hợp trung bình: O(nlog(n)).

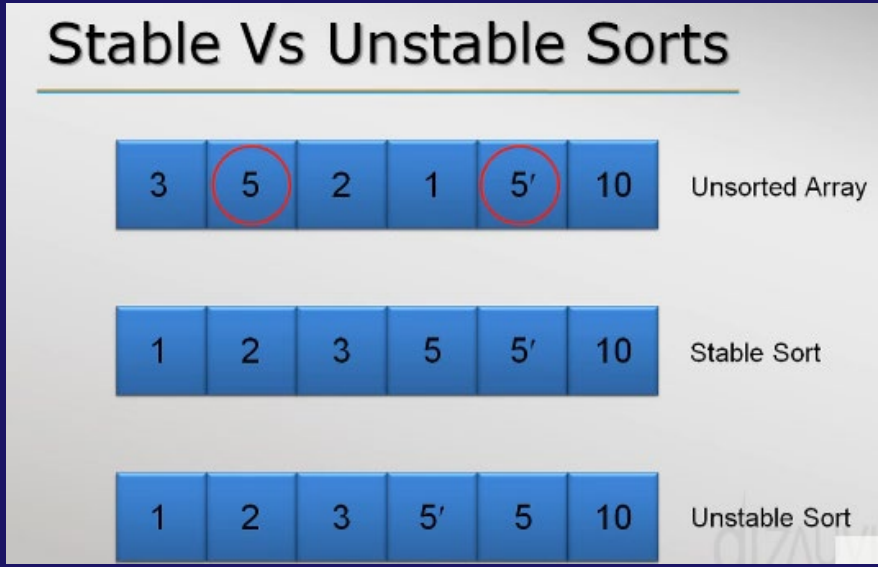
- Độ phức tạp không gian: O(n), vì n phần tử được phân vào n không gian con.



Ưu Điểm Và Nhược Điểm

Ưu điểm:

- Hiệu quả về mặt thời gian, đặc biệt là với các danh sách dữ liệu lớn.
- Hiệu quả cho sắp xếp ngoại (external sorting)
- Là một dạng sắp xếp ổn định (stable sort)



Nhược điểm:

- Có thể chậm hơn thuật toán sắp xếp khác khi lượng dữ liệu nhỏ
- Mặc dù mảng đã được sắp xếp, toàn bộ các bước vẫn chạy
- Cần thêm bộ nhớ để chứa dãy con

Ứng Dụng

- Sắp xếp các danh sách lớn.

VD: Các từ trong từ điển, sách trong thư viện theo tên sách/ tên tác giả, dữ liệu người dùng trong mạng xã hội

- Tận dụng bộ nhớ ngoài cho việc sắp xếp
- Đếm các cặp nghịch thế trong mảng (inversion count of an array)
- Kết hợp với các thuật toán sắp xếp khác như insertion sort

02

Priority queue

Định Nghĩa Priority Queue

Priority Queue là một cấu trúc dữ liệu mà các phần tử được quản lý sẽ có độ ưu tiên khác nhau gắn với từng phần tử. Phần tử có thứ tự ưu tiên cao hơn trong Priority Queue sẽ được xếp lên trước và truy vấn trước.

Đơn giản hơn, Priority Queue là một cấu trúc cho phép nó tự động sắp xếp các phần tử của nó.

Định Nghĩa Priority Queue

Một Priority Queue sẽ có các chức năng cơ bản sau:

- Thêm một phần tử vào tập quản lý
- Lấy ra phần tử có ưu tiên cao nhất
- Loại bỏ một phần tử
- Trả về giá trị của phân tử có độ ưu tiên cao nhất
- Thay đổi độ ưu tiên của một phần tử cho trước

Định Nghĩa Binary Heap

Binary Heap là 1 cây nhị phân có đủ 2 tính chất sau:

- Là cây nhị phân hoàn chỉnh.

- Binary Heap phải là Max Heap hoặc Min Heap. Trong Min Heap, phần tử ở node cha phải là nhỏ hơn hoặc bằng tất cả các phần tử ở node con. Max Heap tương tự như MinHeap.

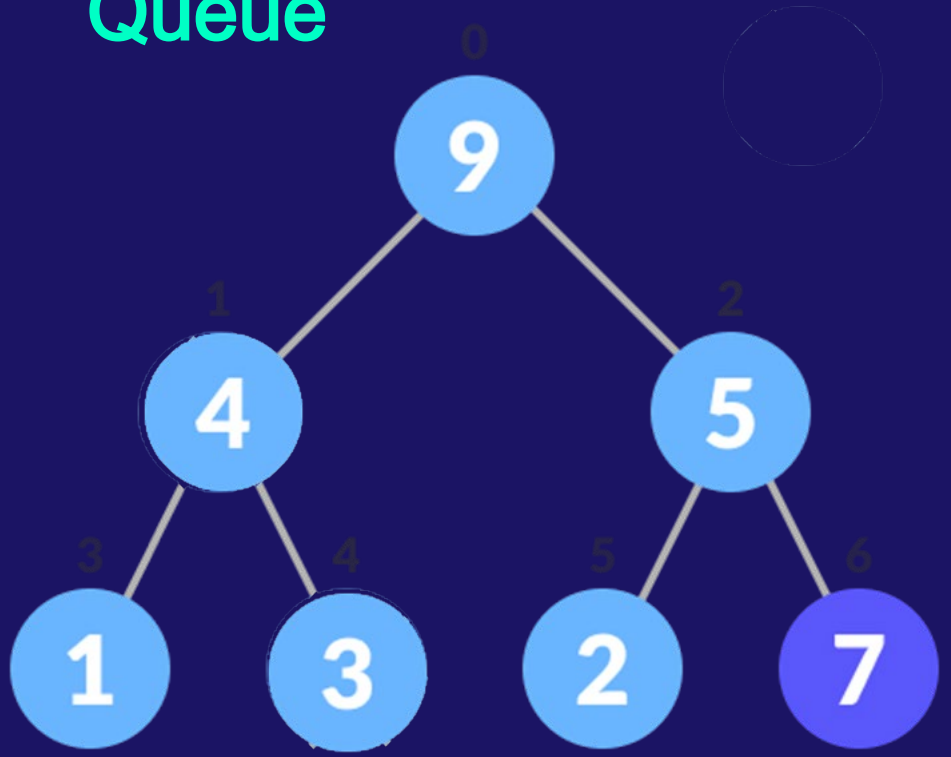
Ứng Dụng

- Được dùng trong các Grapth algorithm:
  - Thuật toán đường đi ngắn nhất Dijkstra.
  - Thuật toán cây bao trùm tối thiểu Prim.
- Bài toán tìm kiếm: Tìm phần tử nhỏ/lớn thứ k.
- Trí tuệ nhân tạo: thuật toán tìm kiếm A*.
- Tất cả các ứng dụng của Queue có liên quan đến mức độ ưu tiên.
- Triển khai Stack.
- Cân bằng tải (load balancing), quản lý các ngắt, thời gian chờ (interrupt handling) trong hệ điều hành.
- Lập lịch CPU.
- Nén dữ liệu: thuật toán Huffman Coding.
- Mô phỏng sự kiện: khách hàng xếp hàng.

Cài Đặt Priority Queue Bằng Binary Heap

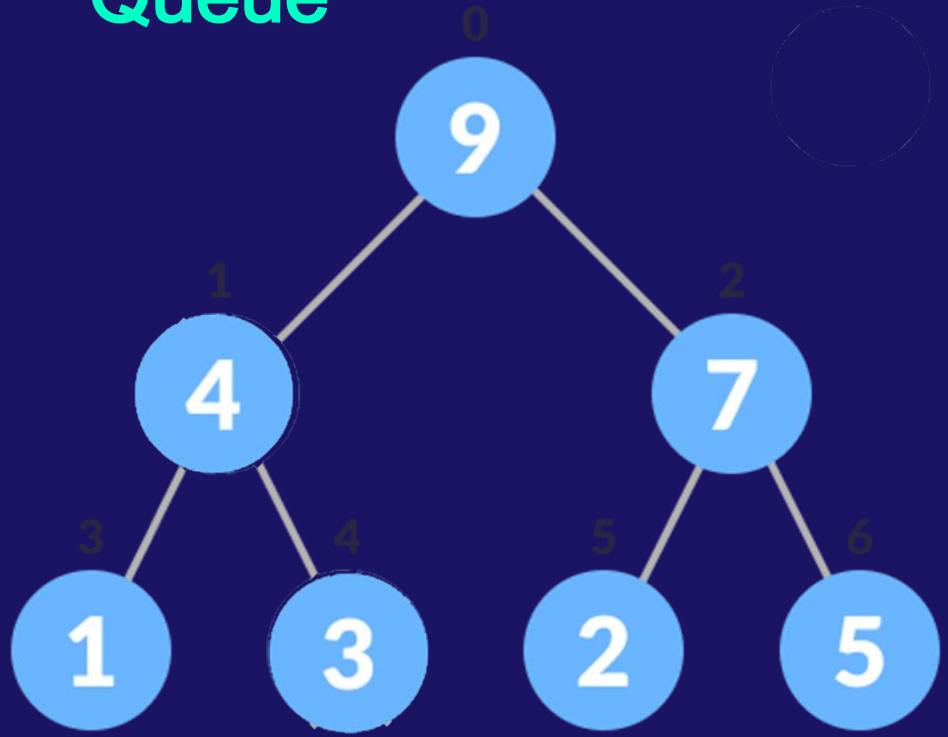
1/Chèn 1 Phần Tử Vào Priority Queue

Chèn phần tử mới vào cuối cây:



1/Chèn 1 Phần Tử Vào Priority Queue

Khôi phục tính chất Heap cho Priority Queue:



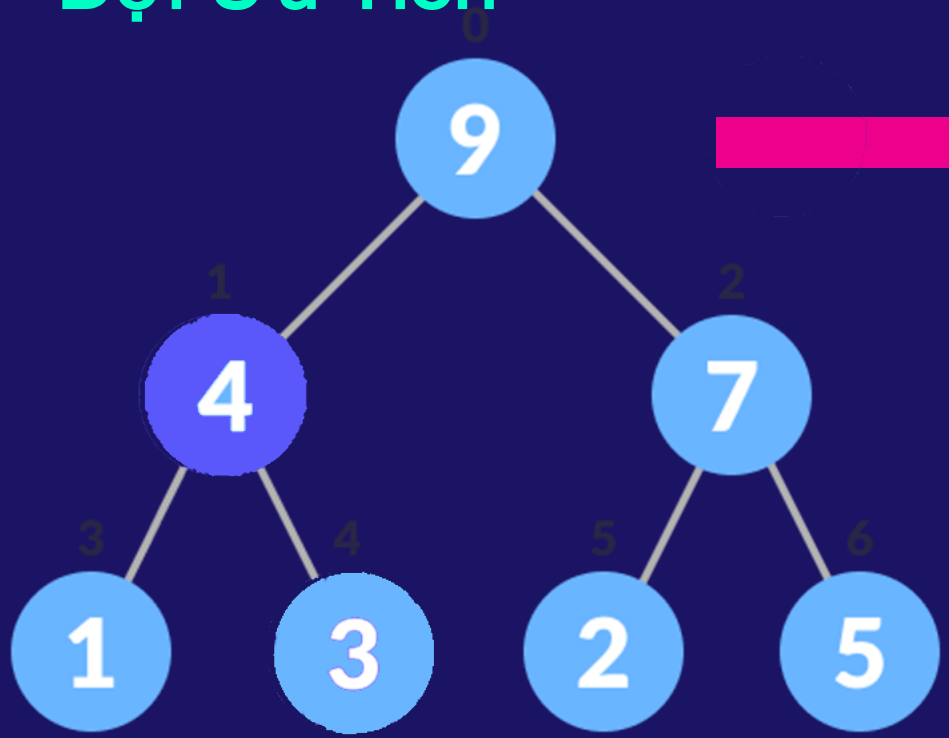
1/Chèn 1 Phần Tử Vào Priority Queue

Mã giả:

```
If there is no node,

  create a newNode.

else (a node is already present)

  insert the newNode at the end (last node from left to right.)



heapify the array
```

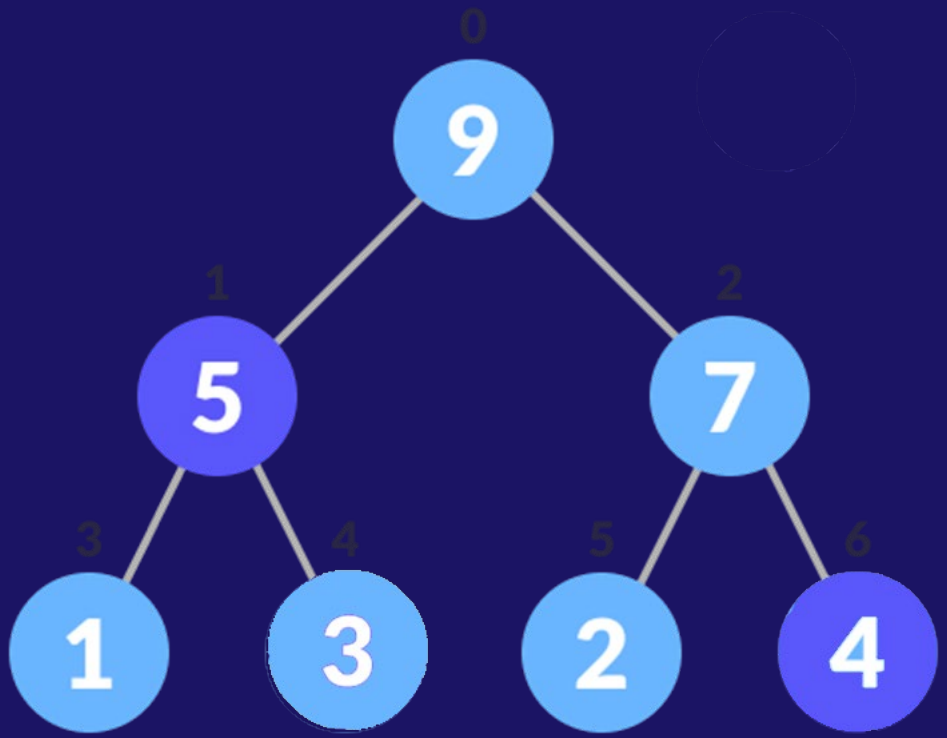
2/Xóa Một Phần Tử Khỏi Hàng Đợi Ưu Tiên

Chọn phần tử cần xóa:



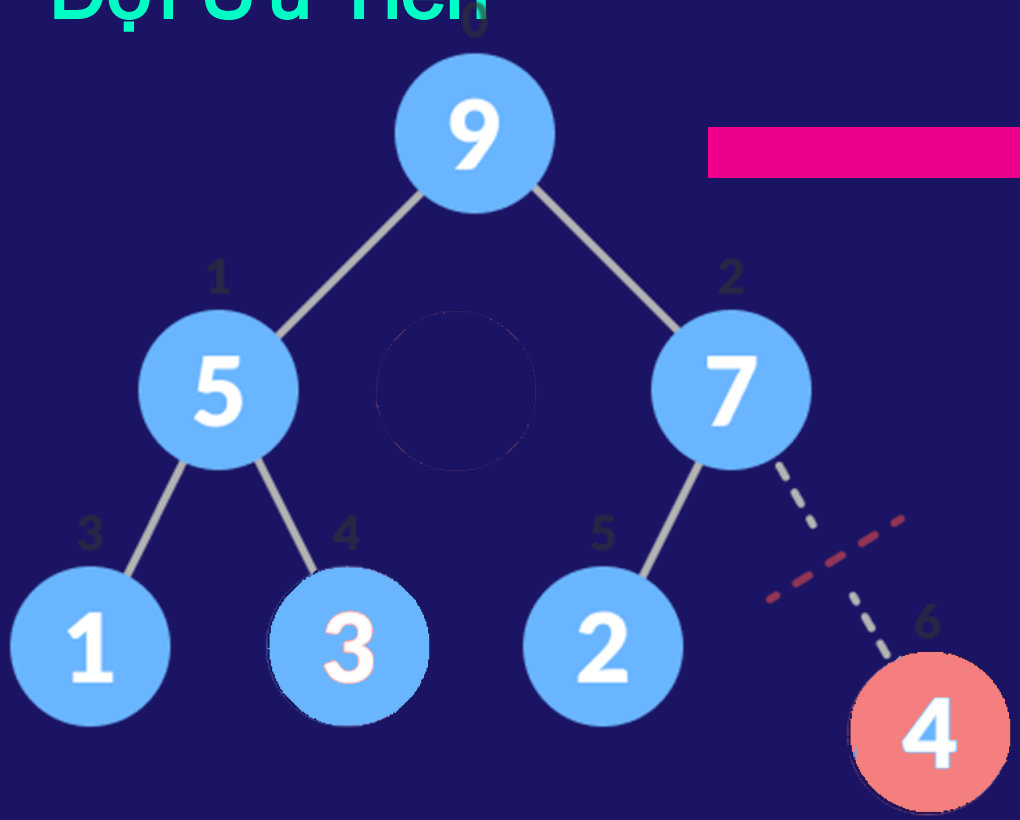
2/Xóa Một Phần Tử Khỏi Hàng Đợi Ưu Tiên

Hoán đổi nó với phần tử cuối cùng:



2/Xóa Một Phần Tử Khỏi Hàng Đợi Ưu Tiên

Xóa phần tử cuối cùng:



2/Xóa Một Phần Tử Khỏi Hàng Đợi Ưu Tiên

Mã giả:

```
If nodeToBeDeleted is the lastLeafNode
   remove the node
Else swap nodeToBeDeleted with the lastLeafNode
   remove noteToBeDeleted


heapify the array
```

3/Tìm Giá Trị Có Độ Ưu Tiên Cao Nhất

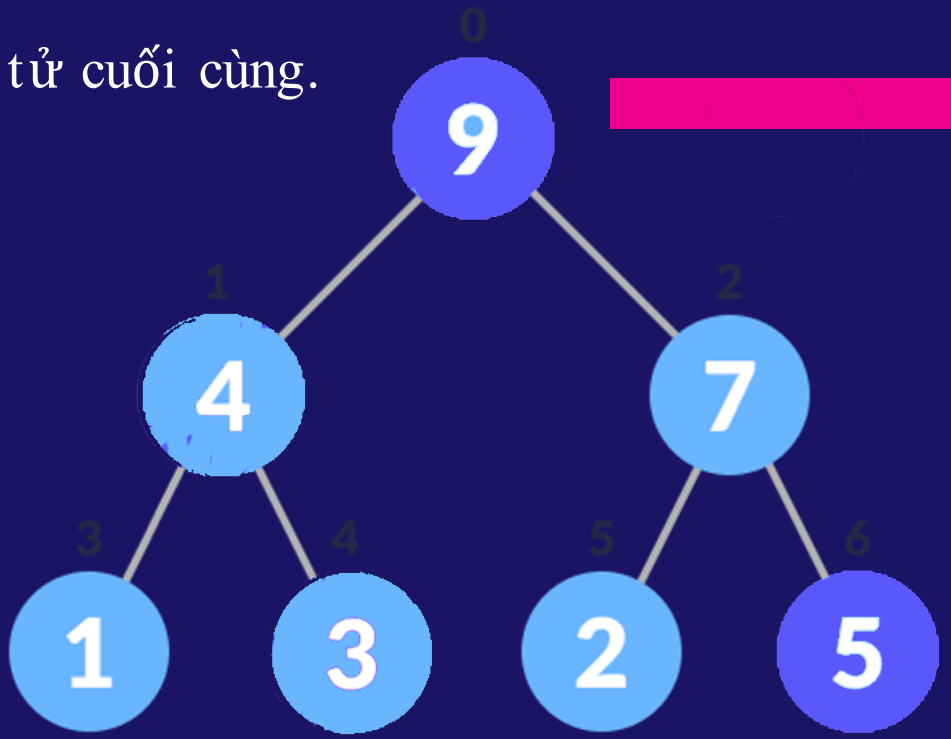
In ra giá trị đầu tiên của Hàng đợi ưu tiên:

Mã giả:

```
return rootNode
```

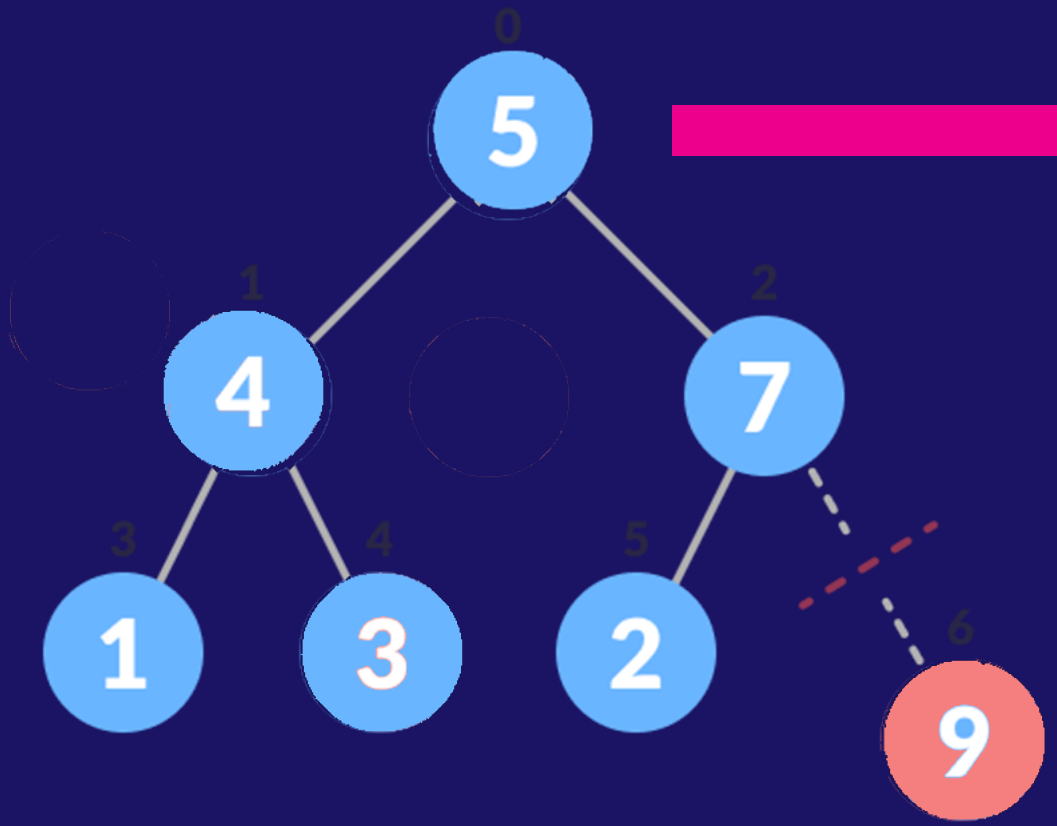
4/Lấy Ra Phần Tử Có Độ Ưu Tiên Cao Nhất

Bước 1: Hoán đổi nó với phần tử cuối cùng.



4/Lấy Ra Phần Tử Có Độ Ưu Tiên Cao Nhất

Bước 2: Lưu giá trị sau đó xóa phần tử cuối cùng.



Cài Đặt Trong C++

```cpp
#include <iostream>
#include <vector>
using namespace std;

void swap(int* a, int* b) {
    int temp = *b;
    *b = *a;
    *a = temp;
}
void heapify(vector<int>& hT, int i) {
    int size = hT.size();
    int largest = i;
    int l = 2 * i + 1;
    int r = 2 * i + 2;
    if (l < size && hT[l] > hT[largest])
        largest = l;
    if (r < size && hT[r] > hT[largest])
        largest = r;
    if (largest != i) {
        swap(&hT[i], &hT[largest]);
        heapify(hT, largest);
    }
}
```

```cpp
void insert(vector<int>& hT, int newNum) {
    int size = hT.size();
    if (size == 0) {
        hT.push_back(newNum);
    }
    else {
        hT.push_back(newNum);
        for (int i = size / 2 - 1; i >= 0; i--) {
            heapify(hT, i);
        }
    }
}
```

```cpp
void deleteNode(vector<int>& hT, int num) {
    int size = hT.size();
    int i;
    for (i = 0; i < size; i++) {
        if (num == hT[i])
            break;
    }
    swap(&hT[i], &hT[size - 1]);

    hT.pop_back();
    for (int i = size / 2 - 1; i >= 0; i--) {
        heapify(hT, i);
    }
}

void printArray(vector<int>& hT) {
    for (int i = 0; i < hT.size(); ++i)
        cout << hT[i] << " ";
    cout << "\n";
}
```

Priority Queue
Trong STL

Cú Pháp Khởi Tạo

Khai báo thư viện: `#include <queue>`

Khai báo priority queue: `priority_queue<Type, Container, Functional>;`

Type: kiểu dữ liệu của priority queue.

Container: kiểu dữ liệu chứa (bắt buộc là kiểu mảng, ví dụ vector,deque..., chú ý không sử dụng list), nếu không khai báo thì mặc định là vector.

Functional: tham trị phụ chỉ thứ tự ưu tiên cho các phần tử (less, greater..).

Cú Pháp Khởi Tạo

Ví dụ: `priority_queue <int, vector<int>, less<int>> q;`

Chú ý: Trong trường hợp chỉ khai báo type, bỏ qua các thành phần còn lại thì mặc định là giảm dần.
(Ví dụ: `priority_queue <int> q;`)

Các Hàm Với priority_queue

empty()

```
q.empty();
```

size()

```
q.size();
```

push()

```
q.push(n);
```

top()

```
q.top();
```

pop()

```
q.pop();
```

swap()

```
q1.swap(q2);
```

Các Hàm Với priority_queue

```cpp
#include<iostream>
#include <queue>
#include <functional>

using namespace std;
int main()
{
    priority_queue<int> a;

    priority_queue<int, vector<int>, greater<int> > c;
    priority_queue<string> b;

    for (int i = 0; i < 5; i++)
    {
        a.push(i);
        c.push(i);
    }
    while (!a.empty())
    {
        cout << a.top() << ' ';
        a.pop();
    }
    cout << endl;
    while (!c.empty())
    {
        cout << c.top() << ' ';
        c.pop();
    }
    cout << endl;

    b.push("abc");
    b.push("abcd");
    b.push("cbd");
    while (!b.empty())
    {
        cout << b.top() << ' ';
        b.pop();
    }
    cout << endl;
    return 0;
}
```

03

QUIZ

THANKS FOR YOUR ATTENTION!

CREDITS: This presentation template was created by Slidesgo, including icons by Flaticon, and infographics &images by Freepik.