Data Structures and Algorithms

Prepared by: Mohamed Ayman

Algorithm Engineer at Valeo

Deep Learning Researcher and Teaching Assistant

at The American University in Cairo (AUC)

spring 2020

sw.eng.MohamedAyman@gmail.com

linkedin.com/in/cs-MohamedAyman

github.com/cs-MohamedAyman

codeforces.com/profile/Mohamed_Ayman

Lecture 11

Binary Heap Tree

Tree Based

Course Roadmap

Part 2: Non-Linear Data Structures

Lecture 8: Binary Tree

Lecture 9: Binary Search Tree

Lecture 10: Self Balancing Binary Search Tree

Lecture 11: Binary Heap Tree

Lecture 12: Hash Table

Lecture 13: Graph

Lecture 14: STL in C++ (Non-Linear Data Structures)

Lecture Agenda

We will discuss in this lecture the following topics

1- Introduction to Binary Heap Tree

2- Insertion Operation

3- Deletion Operation

4- Top Operation

5- Time Complexity & Space Complexity


Let's
START UP

Lecture Agenda

Section 1: Introduction to Binary Heap Tree

Section 2: Insertion Operation

Section 3: Deletion Operation

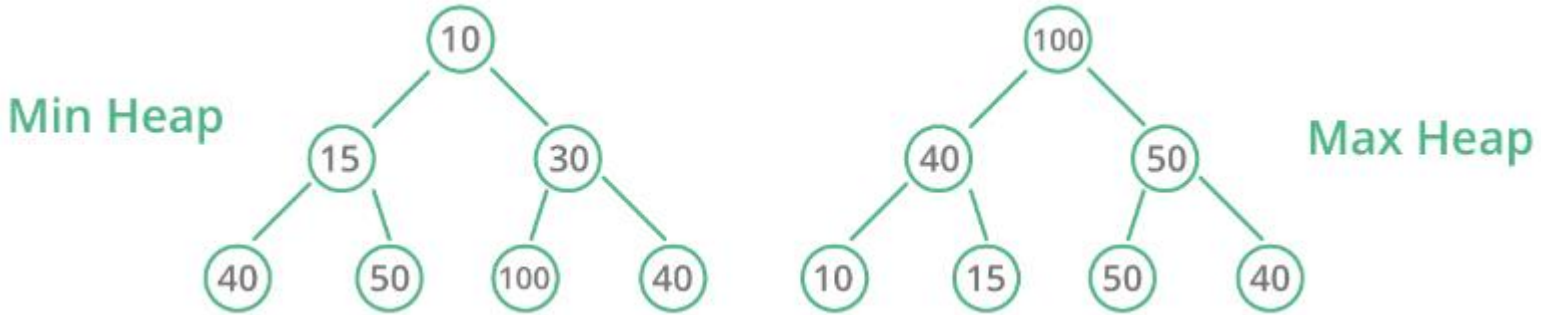
Section 4: Top Operation

Section 5: Time Complexity & Space Complexity

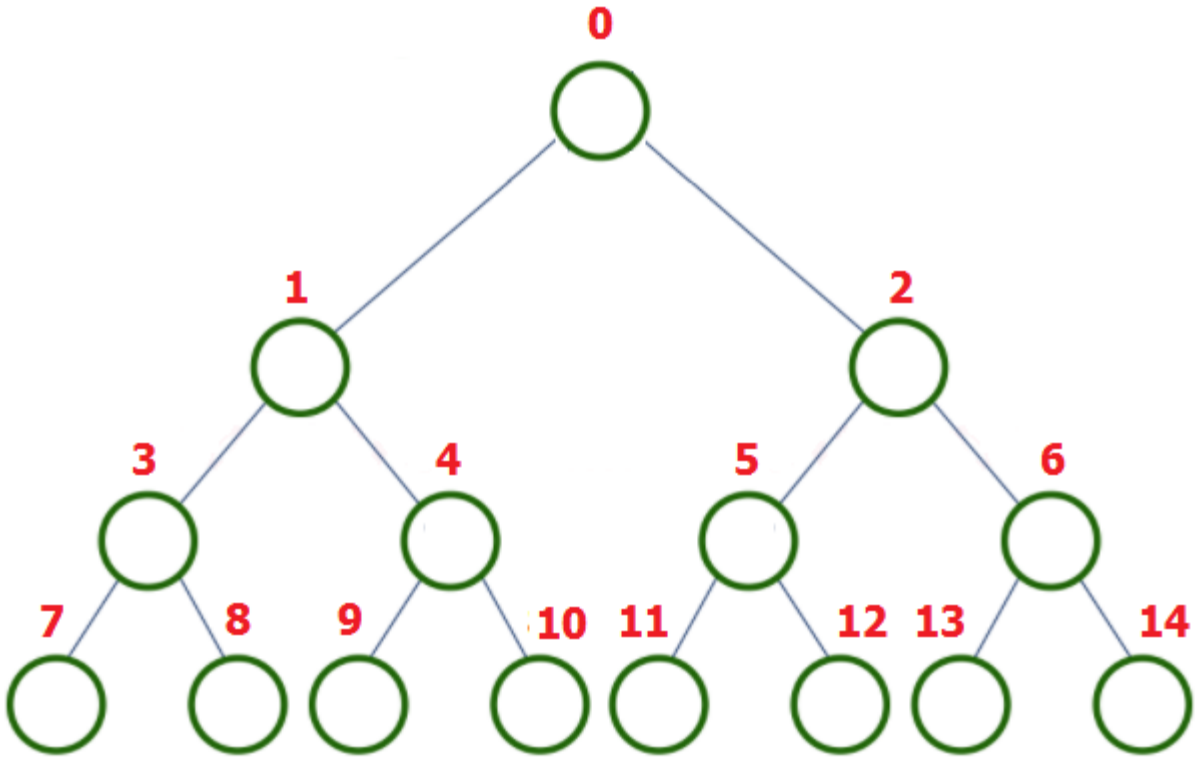
Introduction to Binary Heap Tree

- A heap is a specific tree based data structure in which all the nodes of tree are in a specific order. Let's say if X is a parent node of Y, then the value of X follows some specific order with respect to value of Y and the same order will be followed across the tree. The maximum number of children of a node in the heap depends on the type of heap. However in the more commonly used heap type, there are at most 2 children of a note and it's known as a Binary heap. In binary heap, if the heap is a complete binary tree with N nodes, then it has smallest possible height which is log2 N.
- There can be two types of heap:

- Max Heap: In this type of heap, the value of parent node will always be greater than or equal to the value of child node across the tree and the node with highest value will be the root node of the tree.
- Min Heap: In this type of heap, the value of parent node will always be smallest than or equal to the value of child node across the tree and the node with lowest value will be the root node of the tree.



Introduction to Binary Heap Tree



1st Parent: (i - 1) / 2

2nd Parent: ((i - 1) / 2 - 1) / 2

= (i/4 - 1/4 - 1/2)

= (i - 3) / 4

3rd Parent: ((i - 3) / 4 - 1) / 2

= (i/8 - 3/8 - 1/2)

= (i - 7) / 8

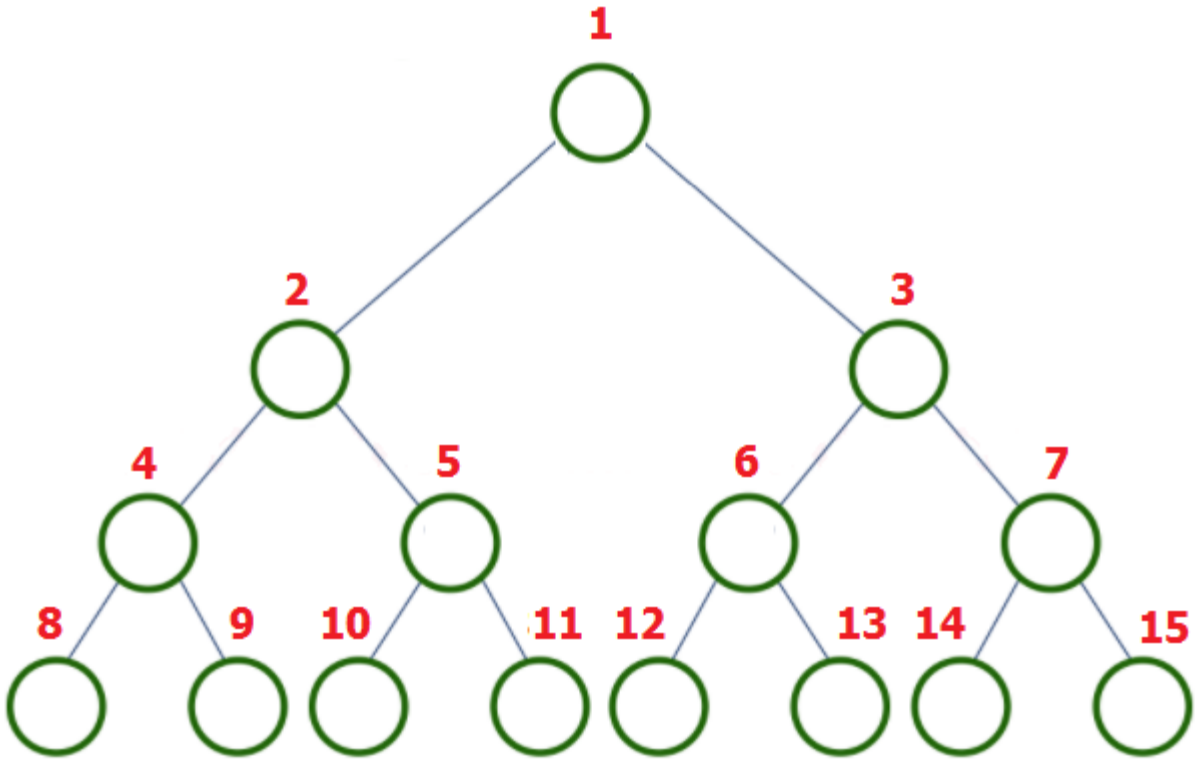
1st Parent: (i - 1) / 2

2nd Parent: (i - 3) / 4

3rd Parent: (i - 7) / 8

N^{th} Parent: (i - (2^x-1)) / 2^x

Introduction to Binary Heap Tree



1st Parent: i / 2

2nd Parent: (i / 2) / 2

= i / 4

3rd Parent: (i / 4) / 2

= i / 8

1st Parent: i / 2

2nd Parent: i / 4

3rd Parent: i / 8

N^{th} Parent: i / 2^x

Get Parent Node Method - Binary Heap Tree

```cpp
// This function gets the node based on the index
node* get_parent_node(node* curr, int idx, int target, int level) {
    // base case we reach a null node
    if (curr == NULL)
        return NULL;
    // return the target node
    if (target == 2*idx+1 || target == 2*idx+2)
        return curr;
    int child = (target - (1<<level) + 1) / (1<<level);
    // repeat the same definition of get node at left or right subtrees
    if (child == 2*idx+1)
        return get_parent_node(curr->left, child, target, level-1);
    if (child == 2*idx+2)
        return get_parent_node(curr->right, child, target, level-1);
    return NULL;
}
```

Max-Heap-Tree-Based.cpp

Min-Heap-Tree-Based.cpp

Lecture Agenda

- ✓ Section 1: Introduction to Binary Heap Tree
- **Section 2: Insertion Operation**
- Section 3: Deletion Operation
- Section 4: Top Operation
- Section 5: Time Complexity & Space Complexity

Insertion Operation - Max Heap

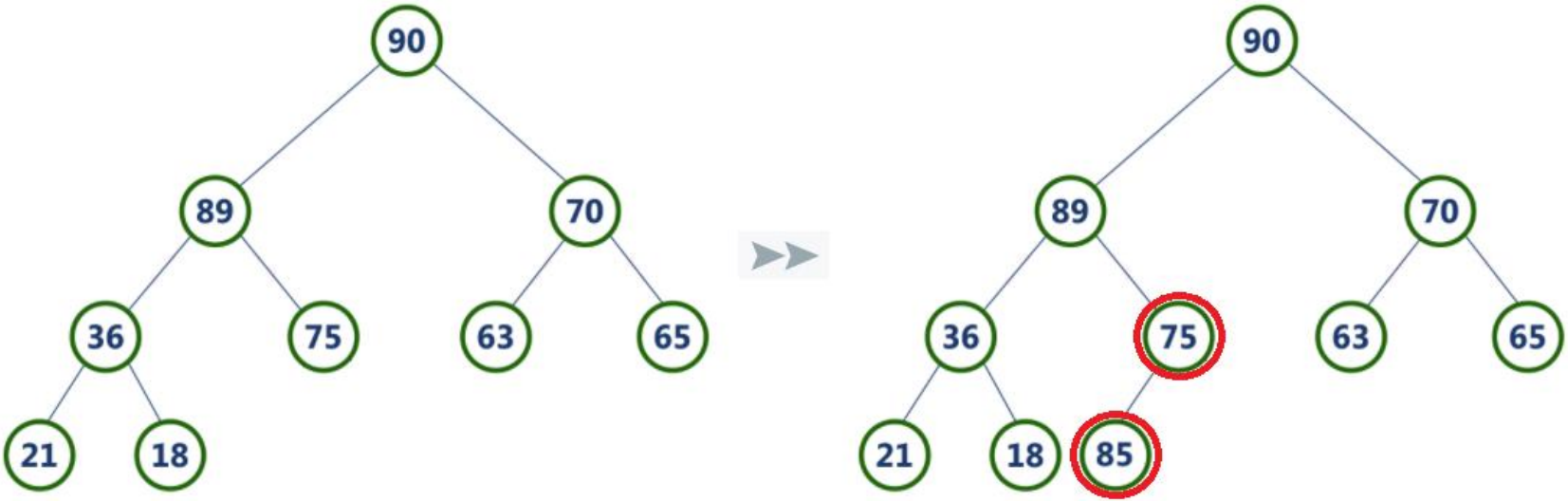
➢ Process of Insertion:

Elements can be inserted to the heap following an approach. The idea is to:

1- First increase the heap size by 1, so that it can store the new element.

2- Insert the new element at the end of the Heap.

3- This newly inserted element may distort the properties of Heap for its parents.
So, in order to keep the properties of Heap, heapify this newly inserted element following a bottom-up approach.



Insertion Operation - Max Heap

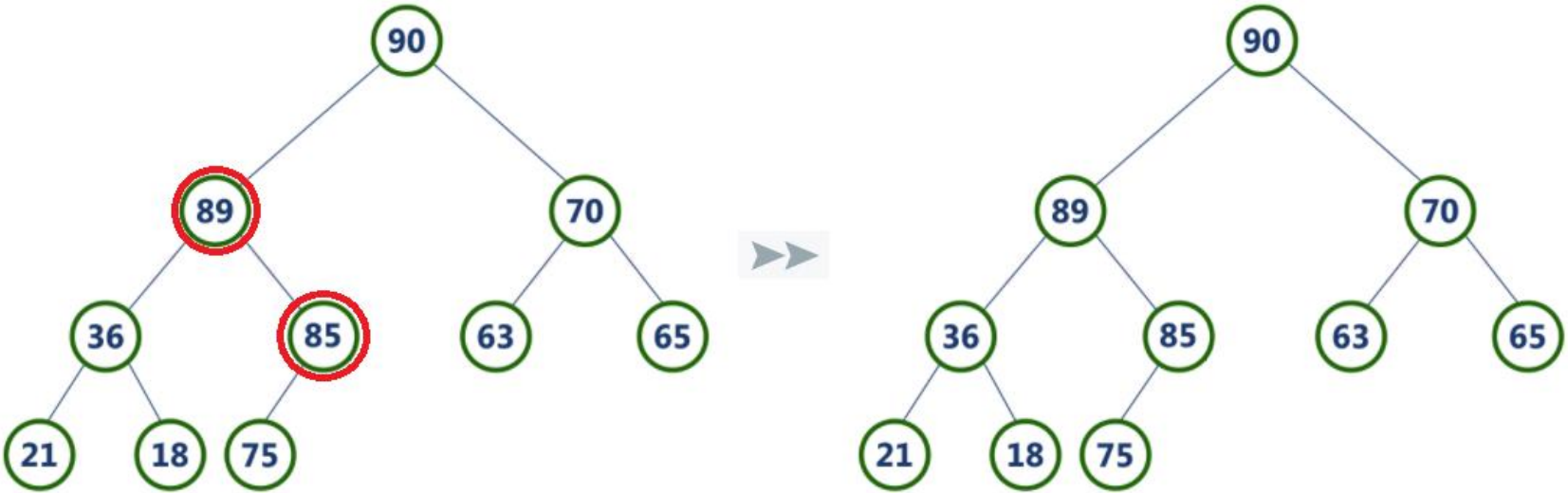
➢ Process of Insertion:

Elements can be inserted to the heap following an approach. The idea is to:

1- First increase the heap size by 1, so that it can store the new element.

2- Insert the new element at the end of the Heap.

3- This newly inserted element may distort the properties of Heap for its parents.
So, in order to keep the properties of Heap, heapify this newly inserted element following a bottom-up approach.



Insertion Operation - Max Heap

```
// This function inserts an element at the given index in the max-heap
void push(int item) {
    // update the size of the max-heap
    n = n + 1;
    // the first node in the heap tree
    if (n == 1) {
        node* new_node = new node();
        new_node->data = item;
        root = new_node;
        return;
    }
    // get the parent of the last node
    node* parent_last_node = get_parent_node(root, 0, n-1, log2(n)-1);
    // insert the new element
    node* new_node = new node();
    new_node->data = item;
```

Max-Heap-Tree-Based.cpp

Insertion Operation - Max Heap

```
    // get the parent of the last node
    node* parent_last_node = get_parent_node(root, 0, n-1, log2(n)-1);
    // insert the new element
    node* new_node = new node();
    new_node->data = item;
    if (parent_last_node->left == NULL)
        parent_last_node->left = new_node;
    else
        parent_last_node->right = new_node;
    new_node->parent = parent_last_node;
    // loop to shif the values of the parents till the root of this path
    node* curr = new_node;
    while (curr != root && curr->parent->data < curr->data) {
        swap(curr->data, curr->parent->data);
        curr = curr->parent;
    }
}
```

Max-Heap-Tree-Based.cpp

Insertion Operation - Min Heap

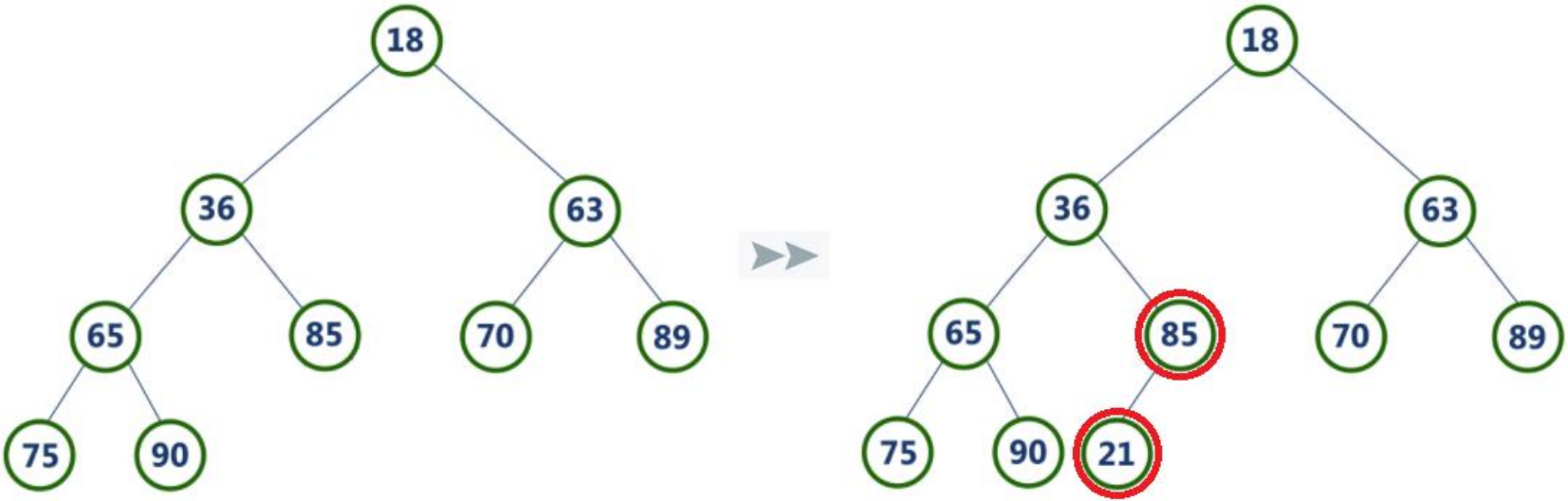
➢ Process of Insertion:

Elements can be inserted to the heap following an approach. The idea is to:

1- First increase the heap size by 1, so that it can store the new element.

2- Insert the new element at the end of the Heap.

3- This newly inserted element may distort the properties of Heap for its parents.
So, in order to keep the properties of Heap, heapify this newly inserted element following a bottom-up approach.



Insertion Operation - Min Heap

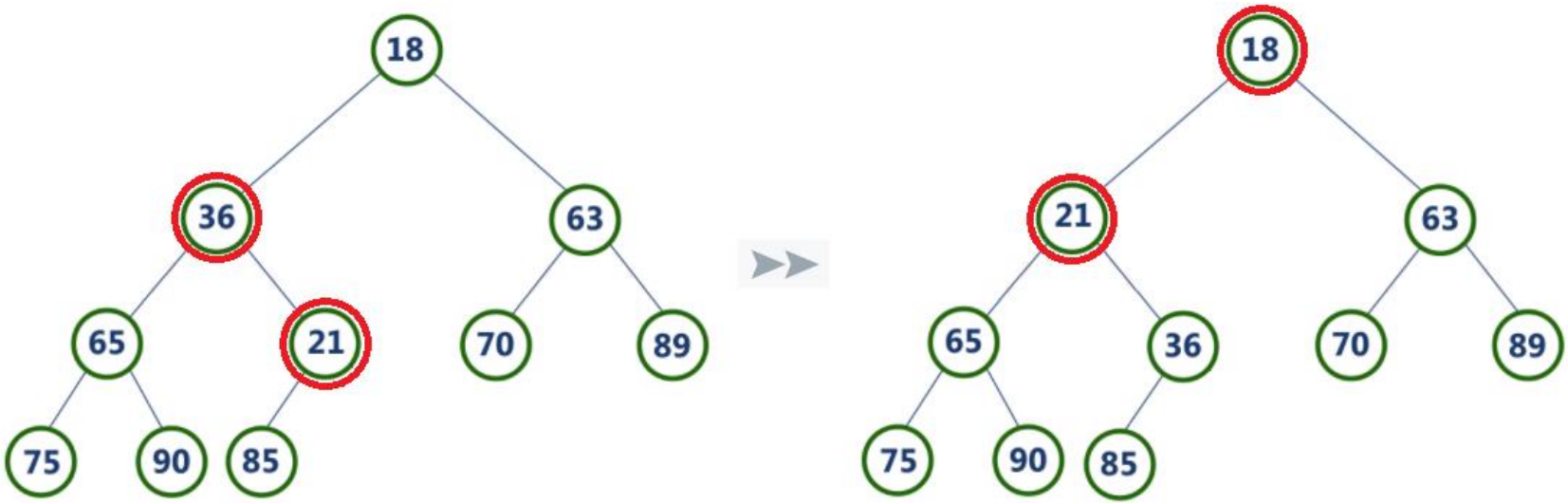
➢ Process of Insertion:

Elements can be inserted to the heap following an approach. The idea is to:

1- First increase the heap size by 1, so that it can store the new element.

2- Insert the new element at the end of the Heap.

3- This newly inserted element may distort the properties of Heap for its parents.
So, in order to keep the properties of Heap, heapify this newly inserted element following a bottom-up approach.



Insertion Operation - Min Heap

```
// This function inserts an element at the given index in the min-heap
void push(int item) {
    // update the size of the min-heap
    n = n + 1;
    // the first node in the heap tree
    if (n == 1) {
        node* new_node = new node();
        new_node->data = item;
        root = new_node;
        return;
    }
    // get the parent of the last node
    node* parent_last_node = get_parent_node(root, 0, n-1, log2(n)-1);
    // insert the new element
    node* new_node = new node();
    new_node->data = item;
```

Min-Heap-Tree-Based.cpp

Insertion Operation - Min Heap

```
    // get the parent of the last node
    node* parent_last_node = get_parent_node(root, 0, n-1, log2(n)-1);
    // insert the new element
    node* new_node = new node();
    new_node->data = item;
    if (parent_last_node->left == NULL)
        parent_last_node->left = new_node;
    else
        parent_last_node->right = new_node;
    new_node->parent = parent_last_node;
    // loop to shif the values of the parents till the root of this path
    node* curr = new_node;
    while (curr != root && curr->parent->data > curr->data) {
        swap(curr->data, curr->parent->data);
        curr = curr->parent;
    }
}
```

Min-Heap-Tree-Based.cpp

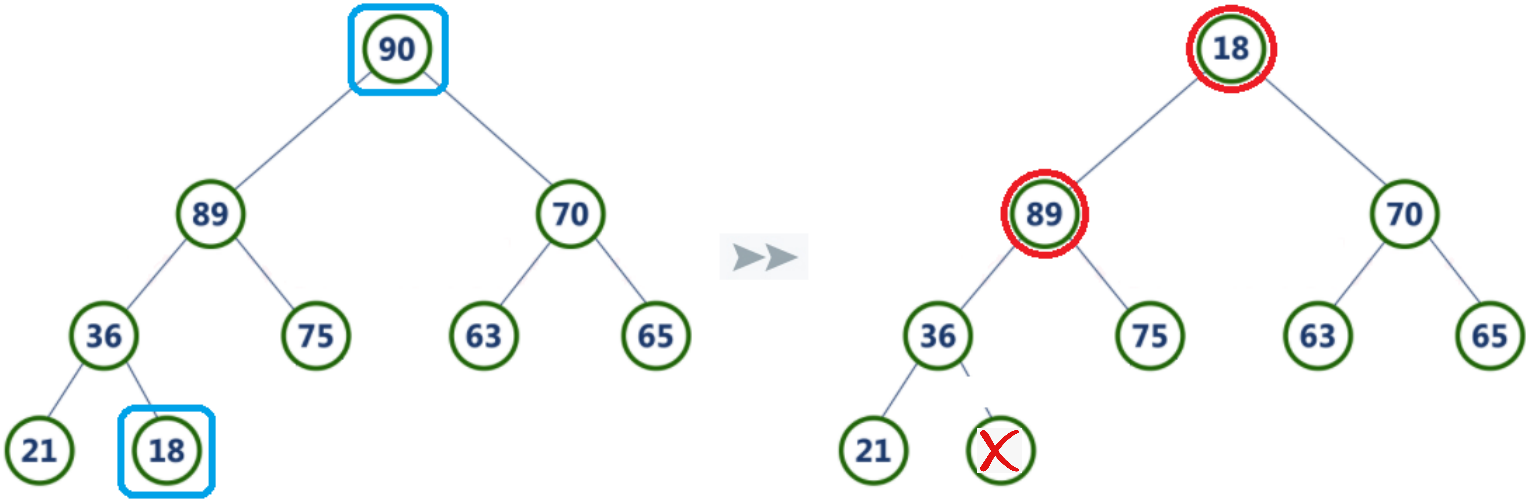
Lecture Agenda

- ✔ Section 1: Introduction to Binary Heap Tree
- ✔ Section 2: Insertion Operation
- **Section 3: Deletion Operation**
- Section 4: Top Operation
- Section 5: Time Complexity & Space Complexity

Deletion Operation - Max Heap

➢ Process of Deletion:

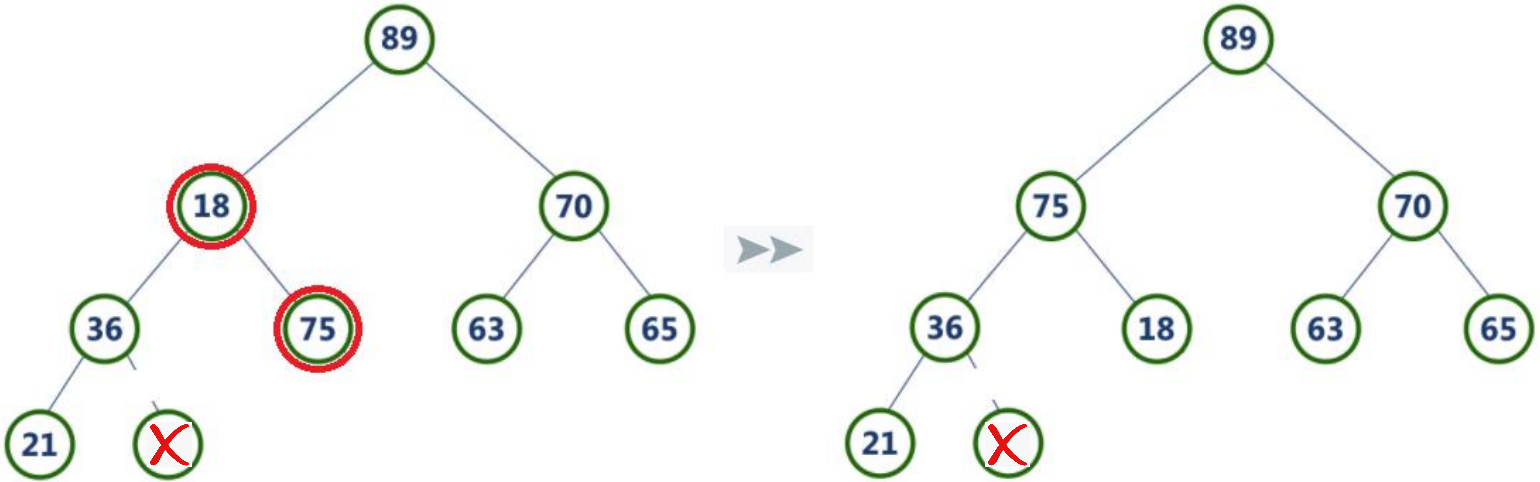
1- Since deleting an element at any intermediary position in the heap can be costly,
So we can simply replace the element to be deleted by the last element and delete the last element of the Heap.
2- Replace the root or element to be deleted by the last element.
3- Delete the last element from the Heap.
4- Since, the last element is now placed at the position of the root node.
So, it may not follow the heap property. Therefore, heapify the last node placed at the position of root.



Deletion Operation - Max Heap

➢ Process of Deletion:

1- Since deleting an element at any intermediary position in the heap can be costly,
So we can simply replace the element to be deleted by the last element and delete the last element of the Heap.
2- Replace the root or element to be deleted by the last element.
3- Delete the last element from the Heap.
4- Since, the last element is now placed at the position of the root node.
So, it may not follow the heap property. Therefore, heapify the last node placed at the position of root.



Deletion Operation - Max Heap

```
// This function deletes an element at index idx in the max-heap
void pop() {
    // check if the max-heap is empty
    if (n == 0)
        return;
    // update the size of the max-heap
    n = n - 1;
    // the first node in the heap tree
    if (n == 0) {
        node* temp = root;
        delete(temp);
        root = NULL;
        return;
    }
    // get the last node
    node* parent_last_node = get_parent_node(root, 0, n, log2(n+1)-1);
```

Max-Heap-Tree-Based.cpp

Deletion Operation - Max Heap

```
// get the last node
node* parent_last_node = get_parent_node(root, 0, n, log2(n+1)-1);
// delete the root of the max-heap by replace it with the last element
if (parent_last_node->right != NULL) {
    swap(root->data, parent_last_node->right->data);
    node* temp = parent_last_node->right;
    delete(temp);
    parent_last_node->right = NULL;
}
else {
    swap(root->data, parent_last_node->left->data);
    node* temp = parent_last_node->left;
    delete(temp);
    parent_last_node->left = NULL;
}
```

Max-Heap-Tree-Based.cpp

Deletion Operation - Max Heap

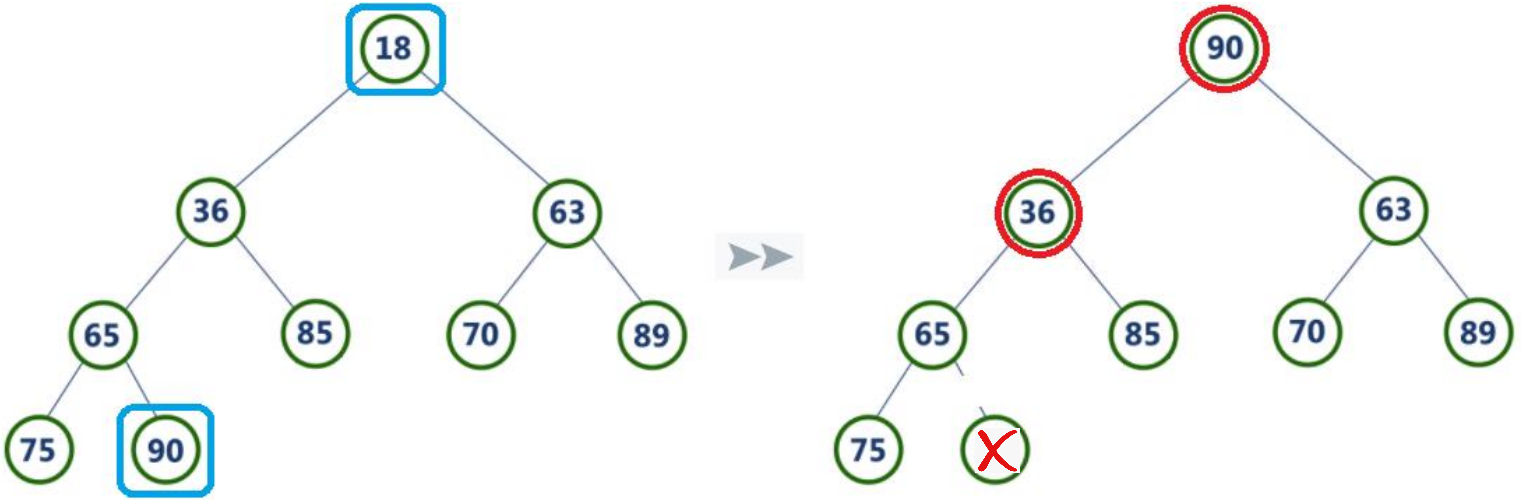
```
    // loop to shif the values of the parents till the root of this path
    node* curr = root;
    node* prev = NULL;
    while (curr != prev) {
        // store the prev index
        prev = curr;
        // get the left and right children
        node* left_child = curr->left;
        node* right_child = curr->right;
        // get the index of the maximum child
        if (left_child != NULL && left_child->data > curr->data)
            curr = left_child;
        if (right_child != NULL && right_child->data > curr->data)
            curr = right_child;
        // swap the old parent with the maximum child
        swap(prev->data, curr->data);
    }
}
```

Max-Heap-Tree-Based.cpp

Deletion Operation - Min Heap

➢ Process of Deletion:

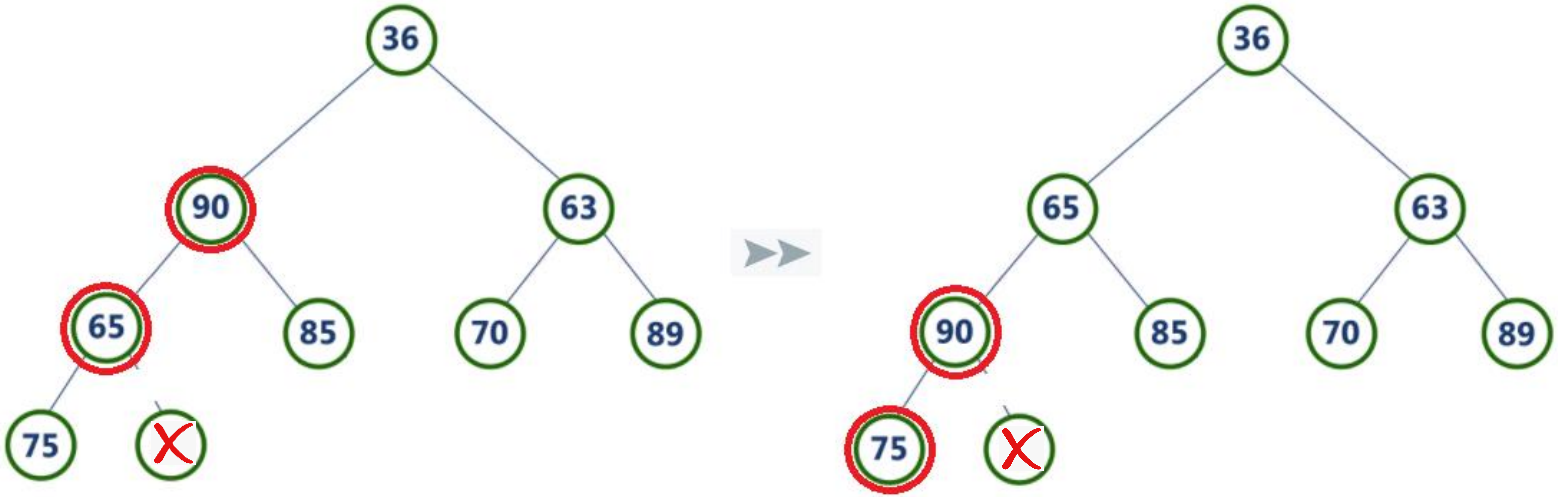
1- Since deleting an element at any intermediary position in the heap can be costly,
So we can simply replace the element to be deleted by the last element and delete the last element of the Heap.
2- Replace the root or element to be deleted by the last element.
3- Delete the last element from the Heap.
4- Since, the last element is now placed at the position of the root node.
So, it may not follow the heap property. Therefore, heapify the last node placed at the position of root.



Deletion Operation - Min Heap

➢ Process of Deletion:

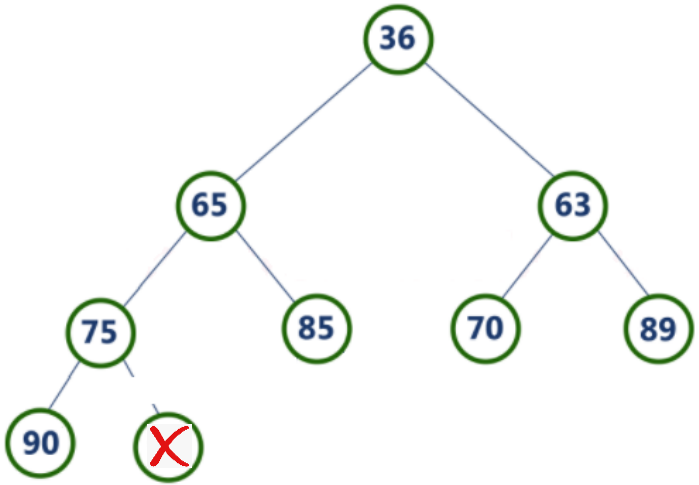
1- Since deleting an element at any intermediary position in the heap can be costly,
So we can simply replace the element to be deleted by the last element and delete the last element of the Heap.
2- Replace the root or element to be deleted by the last element.
3- Delete the last element from the Heap.
4- Since, the last element is now placed at the position of the root node.
So, it may not follow the heap property. Therefore, heapify the last node placed at the position of root.



Deletion Operation - Min Heap

- Process of Deletion:

1- Since deleting an element at any intermediary position in the heap can be costly,
So we can simply replace the element to be deleted by the last element and delete the last element of the Heap.
2- Replace the root or element to be deleted by the last element.
3- Delete the last element from the Heap.
4- Since, the last element is now placed at the position of the root node.
So, it may not follow the heap property. Therefore, heapify the last node placed at the position of root.



Deletion Operation - Min Heap

```
// This function deletes an element at index idx in the min-heap
void pop() {
    // check if the min-heap is empty
    if (n == 0)
        return;
    // update the size of the min-heap
    n = n - 1;
    // the first node in the heap tree
    if (n == 0) {
        node* temp = root;
        delete(temp);
        root = NULL;
        return;
    }
    // get the last node
    node* parent_last_node = get_parent_node(root, 0, n, log2(n+1)-1);
```

Min-Heap-Tree-Based.cpp

Deletion Operation - Min Heap

```
// get the last node
node* parent_last_node = get_parent_node(root, 0, n, log2(n+1)-1);
// delete the root of the min-heap by replace it with the last element
if (parent_last_node->right != NULL) {
    swap(root->data, parent_last_node->right->data);
    node* temp = parent_last_node->right;
    delete(temp);
    parent_last_node->right = NULL;
}
else {
    swap(root->data, parent_last_node->left->data);
    node* temp = parent_last_node->left;
    delete(temp);
    parent_last_node->left = NULL;
}
```

Min-Heap-Tree-Based.cpp

Deletion Operation - Min Heap

```
    // loop to shif the values of the parents till the root of this path
    node* curr = root;
    node* prev = NULL;
    while (curr != prev) {
        // store the prev index
        prev = curr;
        // get the left and right children
        node* left_child = curr->left;
        node* right_child = curr->right;
        // get the index of the minimum child
        if (left_child != NULL && left_child->data < curr->data)
            curr = left_child;
        if (right_child != NULL && right_child->data < curr->data)
            curr = right_child;
        // swap the old parent with the minimum child
        swap(prev->data, curr->data);
    }
}
```

Min-Heap-Tree-Based.cpp

Lecture Agenda

- ✔ Section 1: Introduction to Binary Heap Tree
- ✔ Section 2: Insertion Operation
- ✔ Section 3: Deletion Operation
- **Section 4: Top Operation**
- Section 5: Time Complexity & Space Complexity

Top Operation

- ➢ Extract Maximum (Max-Heap):
- In this operation, the maximum element will be returned and the last element of heap will be placed at index 1 and max_heapify will be performed on node 1 as placing last element on index 1 will violate the property of max-heap.
- ➢ Extract Minimum (Min-Heap):
- In this operation, the minimum element will be returned and the last element of heap will be placed at index 1 and min_heapify will be performed on node 1 as placing last element on index 1 will violate the property of min-heap.

Top Operation - Max Heap

```
// This function gets the value of the root in the max-heap
int get_root() {
    // check if the max-heap is empty
    // to return the smallest integer value as an invalid value
    if (n == 0)
        return INT_MIN;
    // otherwise return the real value
    else
        return root->data;
}
```

Max-Heap-Tree-Based.cpp

Top Operation - Min Heap

```
// This function gets the value of the root in the min-heap
int get_root() {
    // check if the min-heap is empty
    // to return the biggest integer value as an invalid value
    if (n == 0)
        return INT_MAX;
    // otherwise return the real value
    else
        return root->data;
}
```

Min-Heap-Tree-Based.cpp

Functionality Testing - Max Heap

- Initialize a global struct

```cpp
#include <bits/stdc++.h>
using namespace std;

// A binary heap node
struct node {
    int data;
    node* left;
    node* right;
    node* parent;
};

// Initialize a global pointer for root and size of the heap
node* root;
int n;
```

- In the Main function:

```cpp
cout << "Max-Heap root: " << get_root() << '\n';
```

- Expected Output:

```
Max-Heap root: -2147483648
```

Max-Heap-Tree-Based.cpp

Functionality Testing - Max Heap

- In the Main function:

```
cout << "Max-Heap root: " << get_root() << '\n';
push(10);
cout << "Max-Heap root: " << get_root() << '\n';
push(20);
cout << "Max-Heap root: " << get_root() << '\n';
push(15);
cout << "Max-Heap root: " << get_root() << '\n';
push(30);
cout << "Max-Heap root: " << get_root() << '\n';
push(25);
cout << "Max-Heap root: " << get_root() << '\n';
```

- Expected Output:

```
Max-Heap root: -2147483648
Max-Heap root: 10
Max-Heap root: 20
Max-Heap root: 20
Max-Heap root: 30
Max-Heap root: 30
```

Max-Heap-Tree-Based.cpp

Functionality Testing - Max Heap

- In the Main function:

```
while (n > 0) {
    pop();
    cout << "Max-Heap root has been deleted and the current root is: ";
    cout << get_root() << '\n';
}
cout << "Max-Heap is empty now\n";
```

- Expected Output:

```
Max-Heap root has been deleted and the current root is: 25
Max-Heap root has been deleted and the current root is: 20
Max-Heap root has been deleted and the current root is: 15
Max-Heap root has been deleted and the current root is: 10
Max-Heap root has been deleted and the current root is: -2147483648
Max-Heap is empty now
```

Max-Heap-Tree-Based.cpp

Functionality Testing - Max Heap

- In the Main function:

```
for (int i = 1; i <= 16; i++) {
    push(i);
    cout << "Max-Heap root: " << get_root() << '\n';
}
```

- Expected Output:

```
Max-Heap root: 1
Max-Heap root: 2
Max-Heap root: 3
Max-Heap root: 4
Max-Heap root: 5
Max-Heap root: 6
Max-Heap root: 7
Max-Heap root: 8
Max-Heap root: 9
Max-Heap root: 10
Max-Heap root: 11
Max-Heap root: 12
Max-Heap root: 13
Max-Heap root: 14
Max-Heap root: 15
Max-Heap root: 16
```

Max-Heap-Tree-Based.cpp

Functionality Testing - Max Heap

- In the Main function:

```
while (n > 0) {
    pop();
    cout << "Max-Heap root has been deleted and the current root is: ";
    cout << get_root() << '\n';
}
cout << "Max-Heap is empty now\n";
```

- Expected Output:

```
Max-Heap root has been deleted and the current root is: 15
Max-Heap root has been deleted and the current root is: 14
Max-Heap root has been deleted and the current root is: 13
Max-Heap root has been deleted and the current root is: 12
Max-Heap root has been deleted and the current root is: 11
Max-Heap root has been deleted and the current root is: 10
Max-Heap root has been deleted and the current root is: 9
Max-Heap root has been deleted and the current root is: 8
```

Max-Heap-Tree-Based.cpp

Functionality Testing - Max Heap

- In the Main function:

```
while (n > 0) {
    pop();
    cout << "Max-Heap root has been deleted and the current root is: ";
    cout << get_root() << '\n';
}
cout << "Max-Heap is empty now\n";
```

- Expected Output:

```
Max-Heap root has been deleted and the current root is: 7
Max-Heap root has been deleted and the current root is: 6
Max-Heap root has been deleted and the current root is: 5
Max-Heap root has been deleted and the current root is: 4
Max-Heap root has been deleted and the current root is: 3
Max-Heap root has been deleted and the current root is: 2
Max-Heap root has been deleted and the current root is: 1
Max-Heap root has been deleted and the current root is: -2147483648
Max-Heap is empty now
```

Max-Heap-Tree-Based.cpp

Functionality Testing - Min Heap

- Initialize a global struct

```cpp
#include <bits/stdc++.h>
using namespace std;

// A binary heap node
struct node {
    int data;
    node* left;
    node* right;
    node* parent;
};

// Initialize a global pointer for root and size of the heap
node* root;
int n;
```

- In the Main function:

```cpp
cout << "Min-Heap root: " << get_root() << '\n';
```

- Expected Output:

```
Min-Heap root: 2147483647
```

Min-Heap-Tree-Based.cpp

Functionality Testing - Min Heap

- In the Main function:

```
cout << "Min-Heap root: " << get_root() << '\n';
push(25);
cout << "Min-Heap root: " << get_root() << '\n';
push(30);
cout << "Min-Heap root: " << get_root() << '\n';
push(15);
cout << "Min-Heap root: " << get_root() << '\n';
push(20);
cout << "Min-Heap root: " << get_root() << '\n';
push(10);
cout << "Min-Heap root: " << get_root() << '\n';
```

- Expected Output:

```
Min-Heap root: 2147483647
Min-Heap root: 25
Min-Heap root: 25
Min-Heap root: 15
Min-Heap root: 15
Min-Heap root: 10
```

Min-Heap-Tree-Based.cpp

Functionality Testing - Min Heap

- In the Main function:

```
while (n > 0) {
    pop();
    cout << "Min-Heap root has been deleted and the current root is: ";
    cout << get_root() << '\n';
}
cout << "Min-Heap is empty now\n";
```

- Expected Output:

```
Min-Heap root has been deleted and the current root is: 15
Min-Heap root has been deleted and the current root is: 20
Min-Heap root has been deleted and the current root is: 25
Min-Heap root has been deleted and the current root is: 30
Min-Heap root has been deleted and the current root is: 2147483647
Min-Heap is empty now
```

Min-Heap-Tree-Based.cpp

Functionality Testing - Min Heap

- In the Main function:

```
for (int i = 16; i >= 1; i--) {
    push(i);
    cout << "Min-Heap root: " << get_root() << '\n';
}
```

- Expected Output:

```
Min-Heap root: 16
Min-Heap root: 15
Min-Heap root: 14
Min-Heap root: 13
Min-Heap root: 12
Min-Heap root: 11
Min-Heap root: 10
Min-Heap root: 9
Min-Heap root: 8
Min-Heap root: 7
Min-Heap root: 6
Min-Heap root: 5
Min-Heap root: 4
Min-Heap root: 3
Min-Heap root: 2
Min-Heap root: 1
```

Min-Heap-Tree-Based.cpp

Functionality Testing - Min Heap

- In the Main function:

```
while (n > 0) {
    pop();
    cout << "Min-Heap root has been deleted and the current root is: ";
    cout << get_root() << '\n';
}
cout << "Min-Heap is empty now\n";
```

- Expected Output:

```
Min-Heap root has been deleted and the current root is: 2
Min-Heap root has been deleted and the current root is: 3
Min-Heap root has been deleted and the current root is: 4
Min-Heap root has been deleted and the current root is: 5
Min-Heap root has been deleted and the current root is: 6
Min-Heap root has been deleted and the current root is: 7
Min-Heap root has been deleted and the current root is: 8
Min-Heap root has been deleted and the current root is: 9
```

Min-Heap-Tree-Based.cpp

Functionality Testing - Min Heap

- In the Main function:

```
while (n > 0) {
    pop();
    cout << "Min-Heap root has been deleted and the current root is: ";
    cout << get_root() << '\n';
}
cout << "Min-Heap is empty now\n";
```

- Expected Output:

```
Min-Heap root has been deleted and the current root is: 10
Min-Heap root has been deleted and the current root is: 11
Min-Heap root has been deleted and the current root is: 12
Min-Heap root has been deleted and the current root is: 13
Min-Heap root has been deleted and the current root is: 14
Min-Heap root has been deleted and the current root is: 15
Min-Heap root has been deleted and the current root is: 16
Min-Heap root has been deleted and the current root is: 2147483647
Min-Heap is empty now
```

Min-Heap-Tree-Based.cpp

Lecture Agenda

- ✓ Section 1: Introduction to Binary Heap Tree
- ✓ Section 2: Insertion Operation
- ✓ Section 3: Deletion Operation
- ✓ Section 4: Top Operation

Section 5: Time Complexity & Space Complexity

Time Complexity & Space Complexity

➢ Time Analysis

	Worst Case	Average Case
• Insert	Θ(log n)	θ(log n)
• Delete	Θ(log n)	θ(log n)
• Top	Θ(1)	θ(1)

Lecture Agenda

- ✓ Section 1: Introduction to Binary Heap Tree
- ✓ Section 2: Insertion Operation
- ✓ Section 3: Deletion Operation
- ✓ Section 4: Top Operation
- ✓ Section 5: Time Complexity & Space Complexity

Practice

Practice

1- Check if a given array represents a binary heap
2- Connect n ropes with minimum cost
3- Check if a given binary tree is heap
4- Heap sort by iterative way
5- Maximum k sum combinations from two arrays
6- Maximum distinct elements after removing k elements
7- Maximum difference between two subsets of m elements
8- Height of a complete binary tree (or heap) with n nodes
9- Print all nodes less than a value x in a min heap
10- Median of stream of running integers using STL
11- Largest triplet product in a stream
12- Find k numbers with most occurrences in the given array
13- Convert BST to min heap
14- Merge two binary max heaps
15- K-th largest sum contiguous subarray

Practice

16- Minimum product of k integers in an array of positive Integers
17- Leaf starting point in a binary heap data structure
18- Convert min heap to max heap
19- Check if the tree is a min-heap, given level order traversal of a binary tree
20- Rearrange characters in a string such that no two adjacent are same
21- Sum of all elements between k1'th and k2'th smallest elements
22- Minimum sum of two numbers formed from digits of an array
23- Find k'th largest element in a stream
24- Find k largest (or smallest) elements in an array
25- Median in a stream of integers
26- Connect n ropes with minimum cost
27- Check if the Tree is a Min-Heap, given level order traversal of a Binary Tree
28- K'th Smallest/Largest element in Unsorted Array
29- Median of Stream of Running Integers
30- Sum of all elements between k1'th and k2'th smallest elements

Assignment

Implement STL Priority Queue

- Priority queues are a type of container adaptors, specifically designed such that its first element is always the greatest of the elements it contains, according to some strict weak ordering criterion.
- This context is similar to a heap, where elements can be inserted at any moment, and only the max heap element can be retrieved (the one at the top in the priority queue).
- Priority queues are implemented as container adaptors, which are classes that use an encapsulated object of a specific container class as its underlying container, providing a specific set of member functions to access its elements. Elements are popped from the "back" of the specific container, which is known as the top of the priority queue.
- Priority queues are a type of container adapters, specifically designed such that the first element of the queue is the greatest of all elements in the queue and elements are in non increasing order(hence we can see that each element of the queue has a priority{fixed order}).

More Info: cplusplus.com/reference/queue/priority_queue/
More Info: geeksforgeeks.org/priority-queue-in-cpp-stl/
More Info: en.cppreference.com/w/cpp/container/priority_queue

Implement STL Priority Queue

- Member functions: (constructor) Construct priority queue (public member function)
  (empty) Test whether container is empty (public member function)
  (size) Return size (public member function)
  (top) Access top element (public member function)
  (push) Insert element (public member function)
  (pop) Remove top element (public member function)
  (swap) Swap contents (public member function)

More Info: cplusplus.com/reference/queue/priority_queue/
More Info: en.cppreference.com/w/cpp/container/priority_queue


DO
MORE.