Data Structures & 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

facebook.com/cs.MohamedAyman

linkedin.com/in/cs-MohamedAyman

github.com/cs-MohamedAyman

codeforces.com/profile/Mohamed_Ayman

Lecture 11

Binary Heap Tree

Array 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

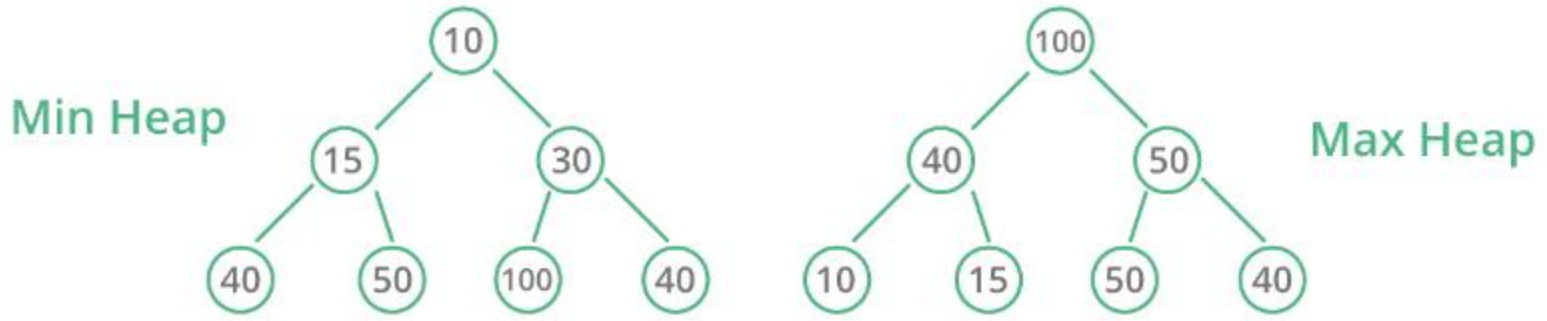
Let's
STARTUP

Lecture Agenda

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

- A Binary Heap is a Binary Tree with following properties.

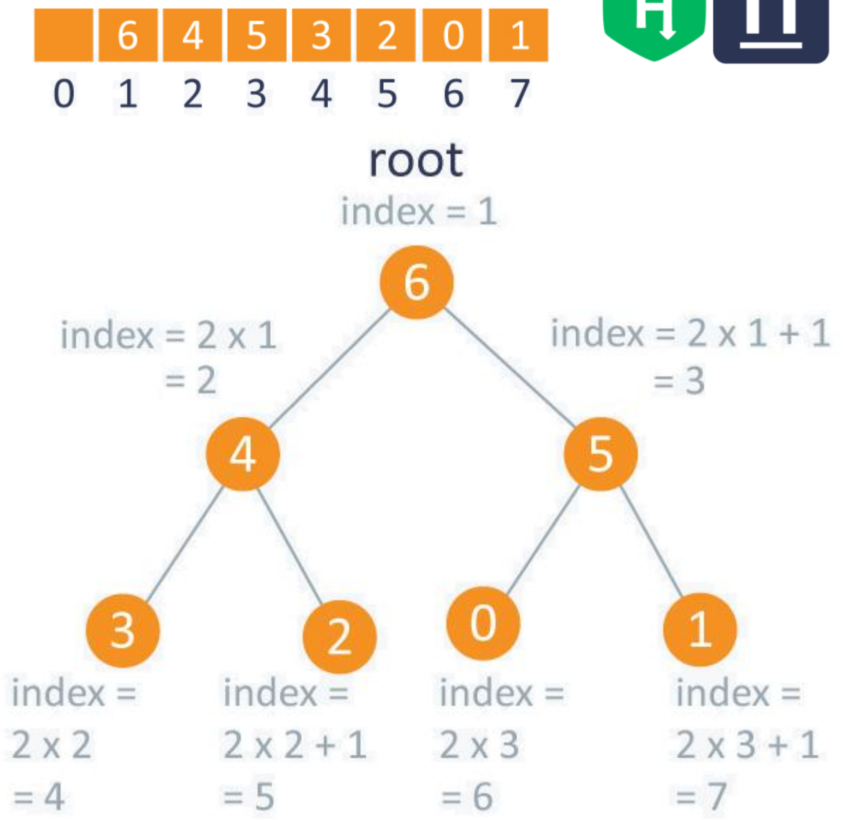
1) It's a complete tree (All levels are completely filled except possibly the last level and the last level has all keys as left as possible). This property of Binary Heap makes them suitable to be stored in an array.

2) A Binary Heap is either Min Heap or Max Heap. In a Min Binary Heap, the key at root must be minimum among all keys present in Binary Heap. The same property must be recursively true for all nodes in Binary Tree. Max Binary Heap is similar to Min Heap with the opposite definition.

- Heap data structures is a specialized binary tree based data structures. Heap is a binary tree with special characteristics. In a heap data structures, nodes are arranged based on their value. A heap data structures, sometime called as Binary Heap.

1) Property (Ordering): Node must be arranged in a order according to values based on Max heap or Min heap.

2) Property (Structural): All levels in a heap must full, expect last level and nodes must be filled from left to right strictly.

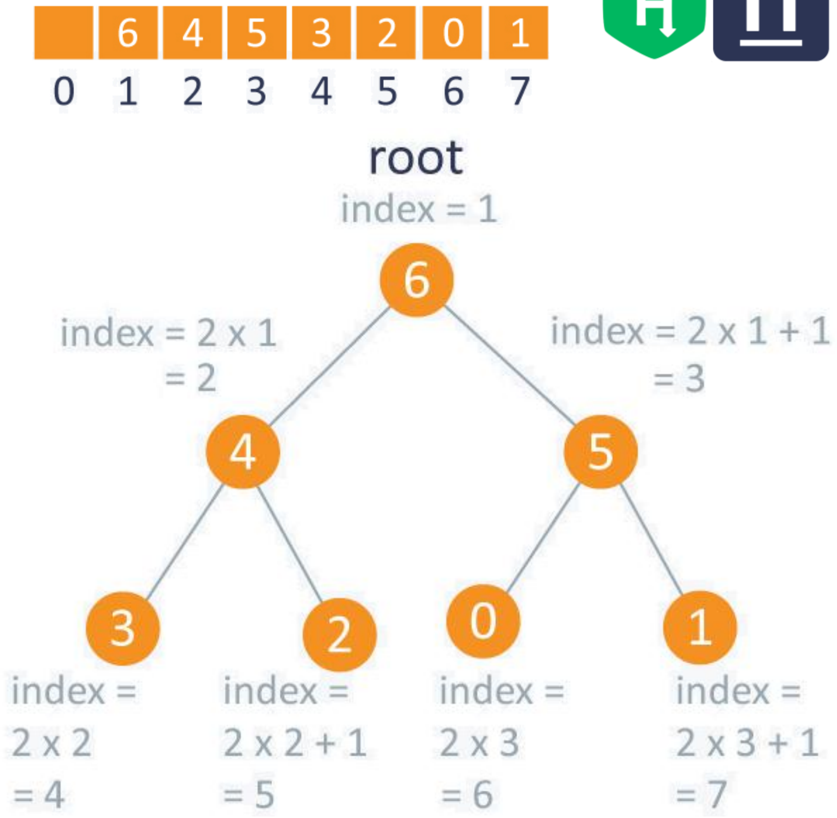


© Prepared by: Mohamed Ayman

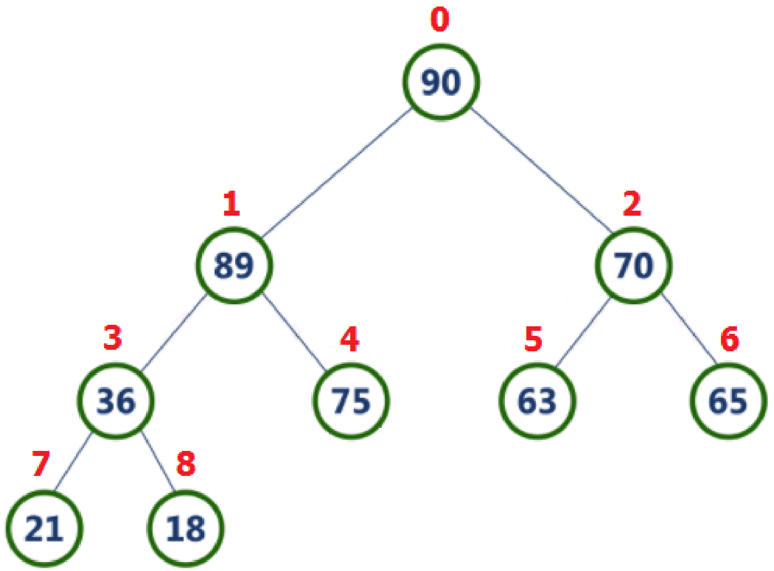
Introduction to Binary Heap Tree

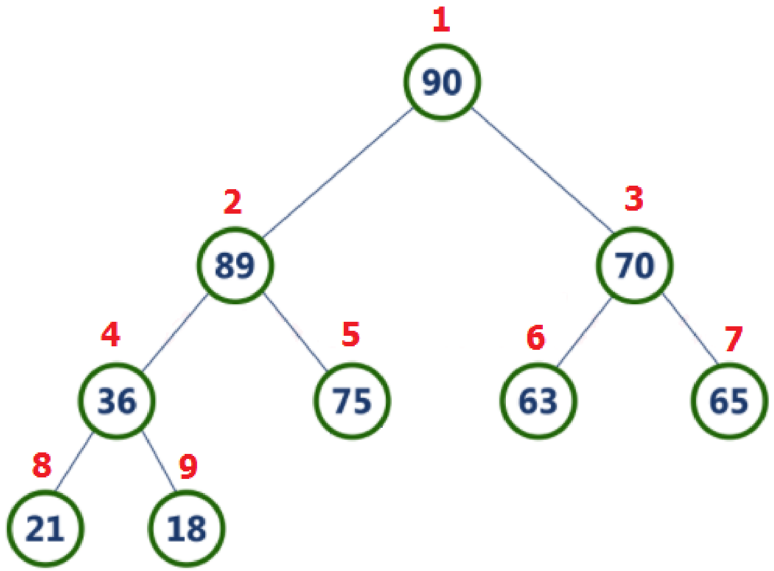
- Note: An array can be used to simulate a tree in the following way.

- If we are storing one element at index i in array Arr, then its parent will be stored at index i/2 (unless its a root, as root has no parent) and can be access by Arr[i/2],
- and its left child can be accessed by Arr[2 * i] and its right child can be accessed by Arr[2 * i +1]. Index of root will be 1 in an array.



Introduction to Binary Heap Tree





Left:	2 * i + 1	Left:	2 * i
Right:	2 * i + 2	Right:	2 * i + 1
Parent:	(i - 1) / 2	Parent:	i / 2

Applications of Binary Heap Tree

- Applications of Binary Heap Tree:

1) Heap Sort: It uses Binary Heap to sort an array. We can use heaps in sorting the elements in a specific order in efficient time. Let's say we want to sort elements of array Arr in ascending order. We can use max heap to perform this operation.

Idea: We build the max heap of elements stored in Arr, and the maximum element of Arr will always be at the root of the heap. Leveraging this idea we can sort an array in the following manner.

2) Priority Queue: Priority queues can be efficiently implemented using Binary Heap because it supports insert(), delete() and extractmax(), decreaseKey() operations.

Binomial Heap and Fibonacci Heap are variations of Binary Heap. These variations perform union also efficiently. it is similar to queue where we insert an element from the back and remove an element from front, but with a one difference that the logical order of elements in the priority queue depends on the priority of the elements. The element with highest priority will be moved to the front of the queue and one with lowest priority will move to the back of the queue. Thus it is possible that when you enqueue an element at the back in the queue, it can move to front because of its highest priority.

3) Graph Algorithms: The priority queues are especially used in Graph Algorithms like Dijkstra's Shortest Path and Prim's Minimum Spanning Tree.

Reserve Method - Binary Heap Tree

```
// This function updates the capacity of the heap
void reserve(int new_capacity) {
    // Initialize a new heap with the new capacity
    int* temp = new int[new_capacity];
    // copy the elements in the current heap to the new heap
    for (int i = 0; i < n; i++)
        temp[i] = arr[i];
    // delete the old heap
    delete[] arr;
    // set the temp heap with new capacity to be the heap
    arr = temp;
    // set the current capacity of the heap to be the new capacity
    capacity = new_capacity;
}
```

Max-Heap-Array-Based.cpp

Min-Heap-Array-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

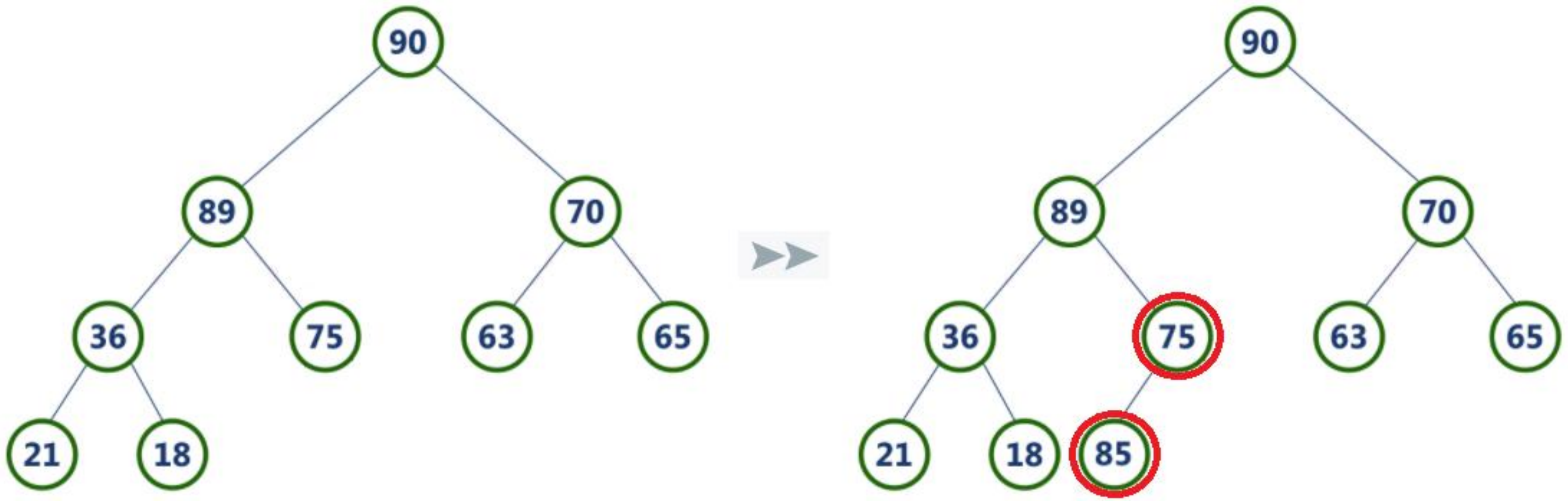
➢ Process of Insertion:

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

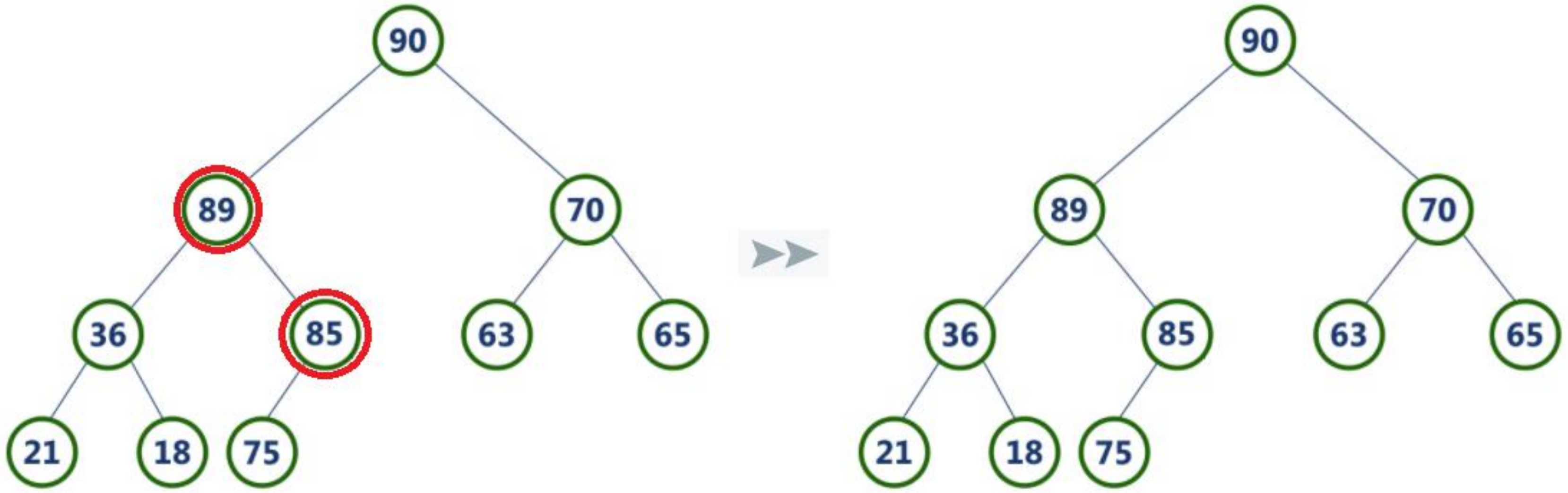
➢ Process of Insertion:

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) {
    // check if we need to update the capacity of the max-heap
    if (n == capacity)
        reserve(2 * capacity + 1);
    // insert the new element
    arr[n] = item;
    // loop to shif the values of the parents till the root of this path
    int i = n;
    while (i > 0 && arr[(i-1) / 2] < arr[i]) {
        swap(arr[(i-1) / 2], arr[i]);
        i = (i-1) / 2;
    }
    // update the size of the max-heap
    n = n + 1;
}
```

Max-Heap-Array-Based.cpp

Insertion Operation - Min Heap

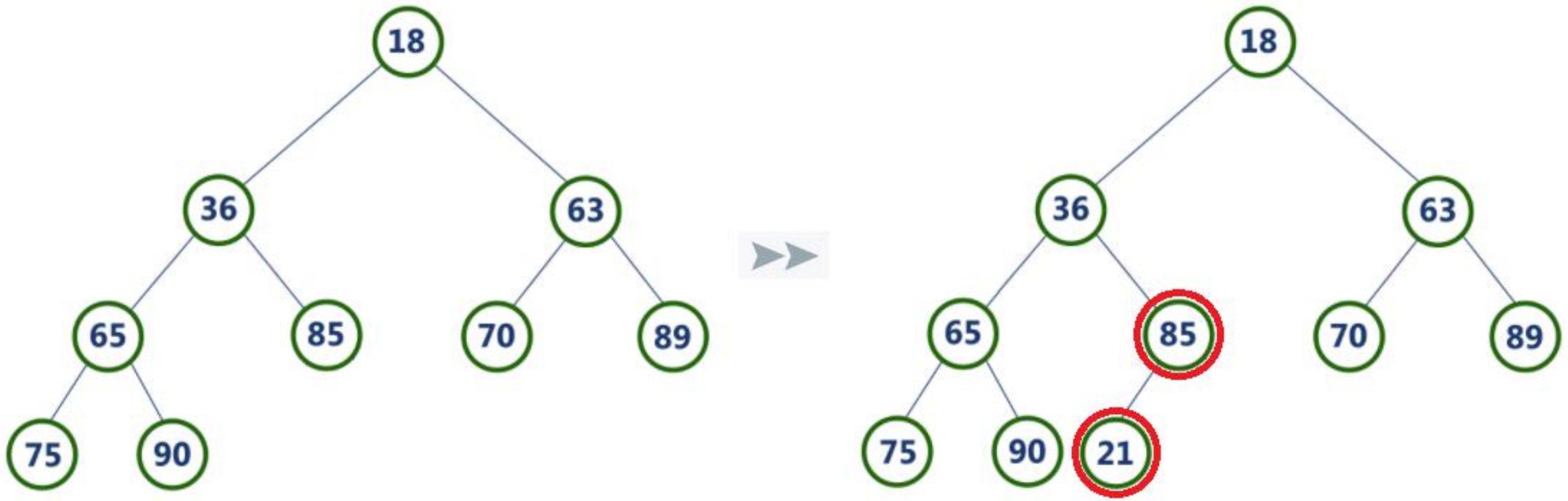
➢ Process of Insertion:

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

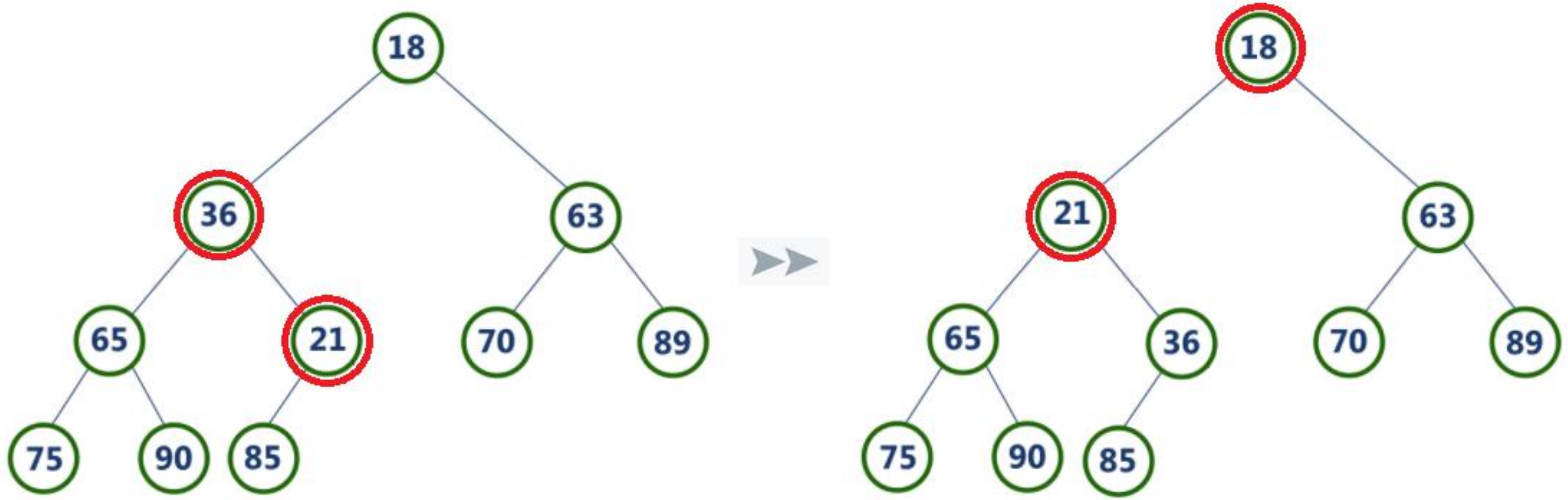
- Process of Insertion:

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) {
    // check if we need to update the capacity of the min-heap
    if (n == capacity)
        reserve(2 * capacity + 1);
    // insert the new element
    arr[n] = item;
    // loop to shif the values of the parents till the root of this path
    int i = n;
    while (i > 0 && arr[(i-1) / 2] > arr[i]) {
        swap(arr[(i-1) / 2], arr[i]);
        i = (i-1) / 2;
    }
    // update the size of the min-heap
    n = n + 1;
}
```

Min-Heap-Array-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

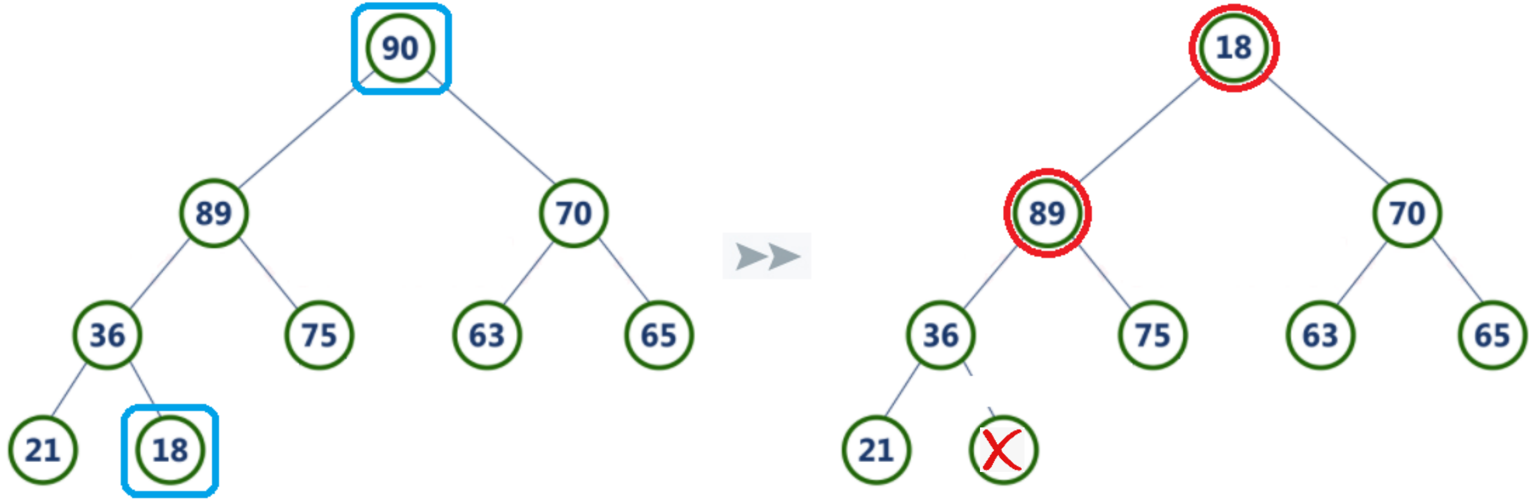
➢ Process of Deletion:

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.

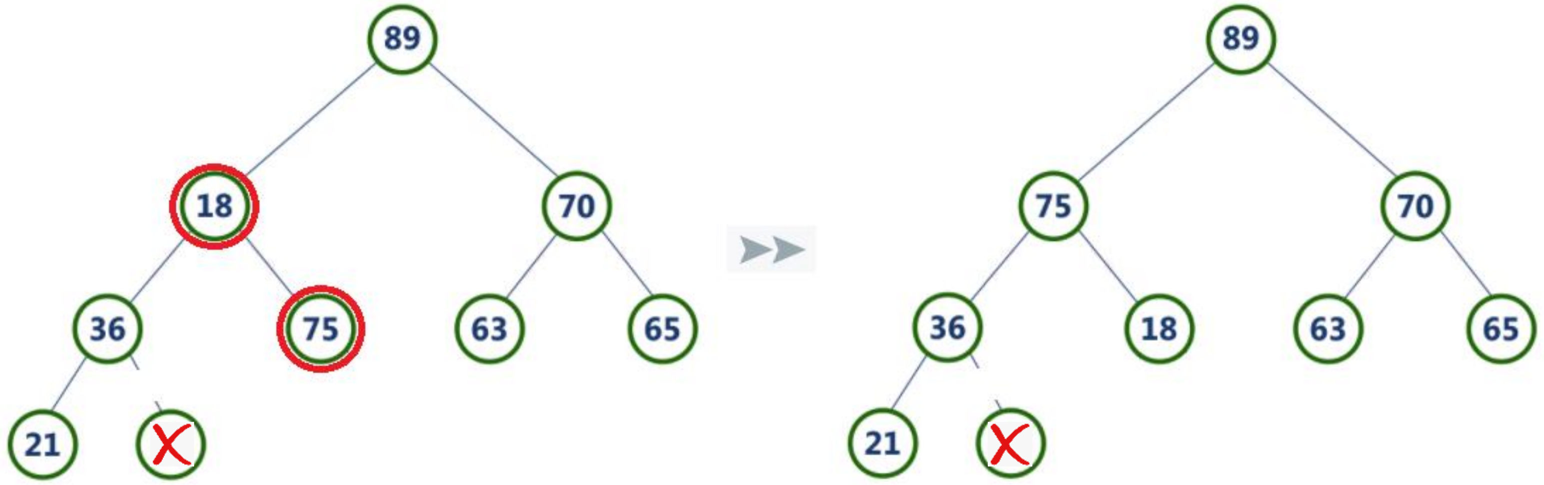
4- Since, the last element is now placed at the position of the root node.
So, it may not follow the heap property. Therefore, heapify the last node placed at the position of root.



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;
    // delete the root of the max-heap by replace it with the last element
    swap(arr[0], arr[n]);
    // loop to shif the values of the parents till the root of this path
    int i = 0, j = -1;
    while (i != j) {
        // store the prev index
        j = i;
        // get the index of the left and right children
        int left_idx  = 2*i + 1;
        int right_idx = 2*i + 2;
```

Max-Heap-Array-Based.cpp

Deletion Operation - Max Heap

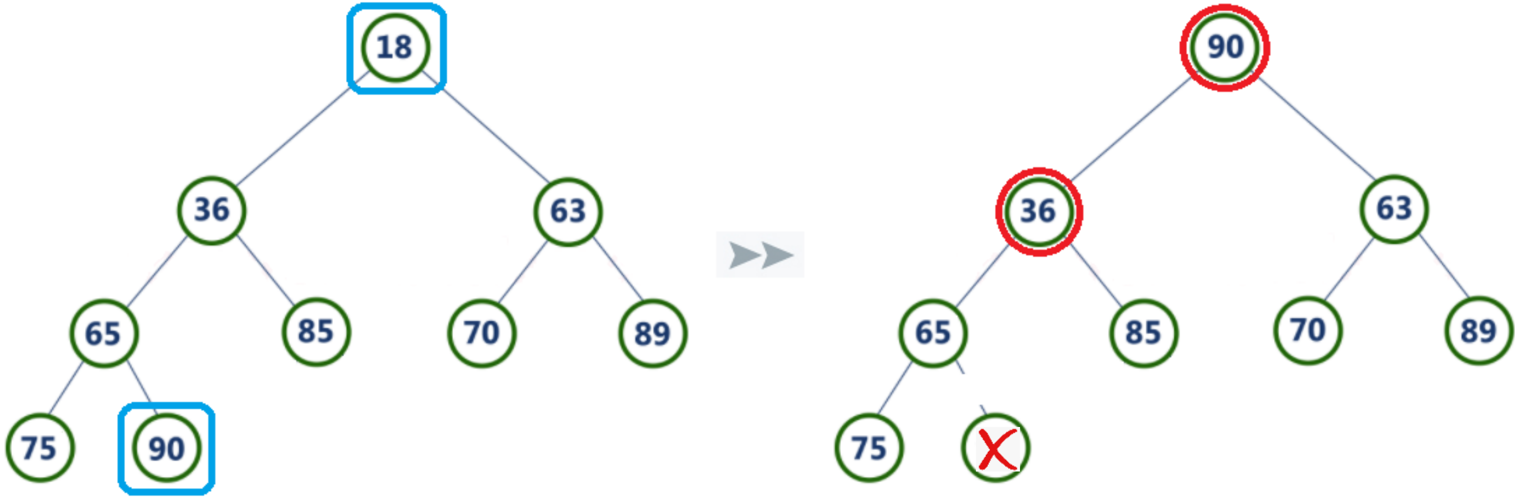
```
    while (i != j) {
        // store the prev index
        j = i;
        // get the index of the left and right children
        int left_idx  = 2*i + 1;
        int right_idx = 2*i + 2;
        // get the index of the maximum child
        if (left_idx < n && arr[left_idx] > arr[i])
            i = left_idx;
        if (right_idx < n && arr[right_idx] > arr[i])
            i = right_idx;
        // swap the old parent with the maximum child
        swap(arr[i], arr[j]);
    }
    // check if we need to update the capacity of the max-heap
    if (n < capacity / 2)
        reserve(capacity / 2);
}
```

Max-Heap-Array-Based.cpp

Deletion Operation - Min Heap

- Process of Deletion:

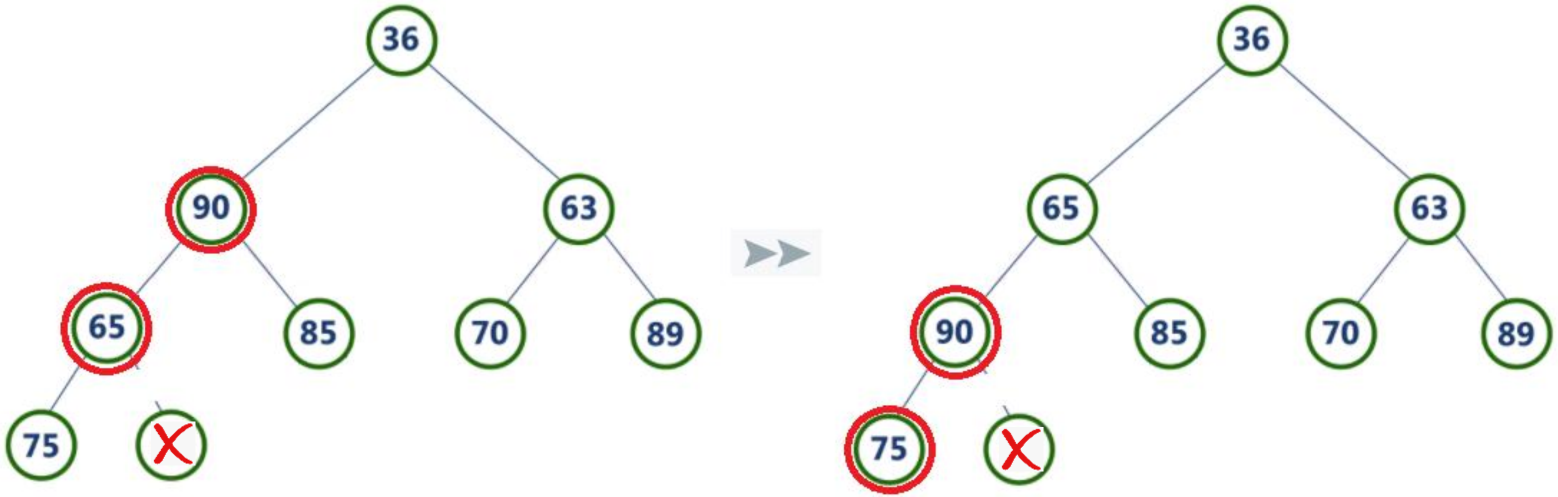
1- Since deleting an element at any intermediary position in the heap can be costly,
So we can simply replace the element to be deleted by the last element and delete the last element of the Heap.
2- Replace the root or element to be deleted by the last element.
3- Delete the last element from the Heap.
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:

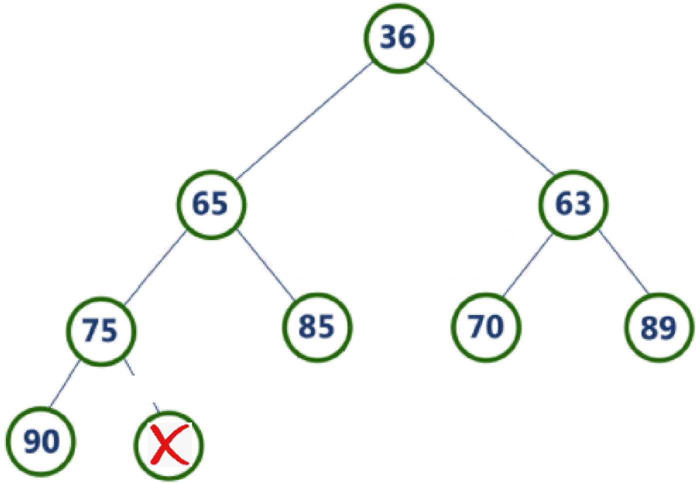
1- Since deleting an element at any intermediary position in the heap can be costly,
So we can simply replace the element to be deleted by the last element and delete the last element of the Heap.
2- Replace the root or element to be deleted by the last element.
3- Delete the last element from the Heap.
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;
    // delete the root of the min-heap by replace it with the last element
    swap(arr[0], arr[n]);
    // loop to shif the values of the parents till the root of this path
    int i = 0, j = -1;
    while (i != j) {
        // store the prev index
        j = i;
        // get the index of the left and right children
        int left_idx  = 2*i + 1;
        int right_idx = 2*i + 2;
```

Min-Heap-Array-Based.cpp

Deletion Operation - Min Heap

```cpp
    while (i != j) {
        // store the prev index
        j = i;
        // get the index of the left and right children
        int left_idx  = 2*i + 1;
        int right_idx = 2*i + 2;
        // get the index of the minimum child
        if (left_idx < n && arr[left_idx] < arr[i])
            i = left_idx;
        if (right_idx < n && arr[right_idx] < arr[i])
            i = right_idx;
        // swap the old parent with the minimum child
        swap(arr[i], arr[j]);
    }
    // check if we need to update the capacity of the min-heap
    if (n < capacity / 2)
        reserve(capacity / 2);
}
```

Min-Heap-Array-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 arr[0];
}
```

Max-Heap-Array-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 arr[0];
}
```

Min-Heap-Array-Based.cpp

Functionality Testing - Max Heap

- Initialize a global array

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

// Initialize a heap with dynamic length
int n;
int capacity;
int* arr;
```

- In the Main function:

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

- Expected Output:

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

Max-Heap-Array-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-Array-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-Array-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-Array-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-Array-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-Array-Based.cpp

Functionality Testing - Min Heap

- Initialize a global array

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

// Initialize a heap with dynamic length
int n;
int capacity;
int* arr;
```

- In the Main function:

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

- Expected Output:

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

Min-Heap-Array-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-Array-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-Array-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-Array-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-Array-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-Array-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(n)	θ(log n)
• Delete	Θ(n)	θ(log n)
• Top	Θ(1)	θ(1)

Why is Heap Preferred over BST for Priority Queue?

➢ A Self Balancing Binary Search Tree like AVL Tree, Red-Black Tree, etc can also support above operations with same time complexities.

1. Finding minimum and maximum are not naturally $\Theta(1)$, but can be easily implemented in $\Theta(1)$ by keeping an extra pointer to minimum or maximum and updating the pointer with insertion and deletion if required. With deletion we can update by finding in-order predecessor or successor.

2. Inserting an element is naturally $\Theta(\log n)$.

3. Removing maximum or minimum are also $\Theta(\log n)$.

4. Decrease key can be done in $\Theta(\log n)$ by doing a deletion followed by insertion.

➢ So why is Binary Heap Preferred for Priority Queue? Since Binary Heap is implemented using arrays, there is always better locality of reference and operations are more cache friendly. Although operations are of same time complexity, constants in Binary Search Tree are higher.

- We can build a Binary Heap in $\Theta(n)$ time. Self Balancing BSTs require $\Theta(n \log n)$ time to construct.
- Binary Heap doesn't require extra space for pointers and Binary Heap is easier to implement.
- There are variations of Binary Heap like Fibonacci Heap that can support insert and decrease-key in $\Theta(1)$ time.

Why is Heap Preferred over BST for Priority Queue?

- Is Binary Heap always better? Although Binary Heap is for Priority Queue, BSTs have their own advantages and the list of advantages is in-fact bigger compared to binary heap.
- Searching an element in self-balancing BST is Θ(log n) which is Θ(n) in Binary Heap.
- We can print all elements of BST in sorted order in Θ(n) time, but Binary Heap requires Θ(n log n) time.
- Advantage of Binary Heap Is Binary Heap
- Its flexibility. That's because memory in this structure can be allocated and removed in any particular order. Slowness in the heap can be compensated if an algorithm is well designed and implemented.
- Garbage collection run on the heap memory to free the memory used by object. they doesn't have reference. any object created in the heap space has global access .
- It helps. to find minimum number and maximum number. heat widely used because it is very efficient.
- Disadvantage of Binary Heap Is Binary Heap
- Memory management is more complex in heap memory because it is used globally. heap memory is divided into two parts-old generation and young generation etc. at java garbage collection. It take to much time in execution compare then other.

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:

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.