Data Structures
Lab Manual 12

Topic: Heap

Session: Spring 2023

Faculty of Information Technology

UCP Lahore Pakistan

Heap

Objectives:

The objective of learning about heaps in data structures and algorithms (DSA) is to understand and utilize this fundamental data structure efficiently. A heap is a specialized tree-based data structure that satisfies the heap property, which allows for efficient access to the minimum or maximum element depending on the type of heap (min-heap or max-heap).

Here are the key objectives of learning about heaps in DSA:

1. Understanding heap properties: Learning about the heap property and how it differs for min-heaps and max-heaps is essential. This property ensures that the root element of the heap holds either the minimum or maximum value, allowing for quick retrieval.

2. Mastering heap operations: Learning how to perform operations on heaps is crucial. These operations include insertion of elements, deletion of elements (often the root), and heapify operations to maintain the heap property.

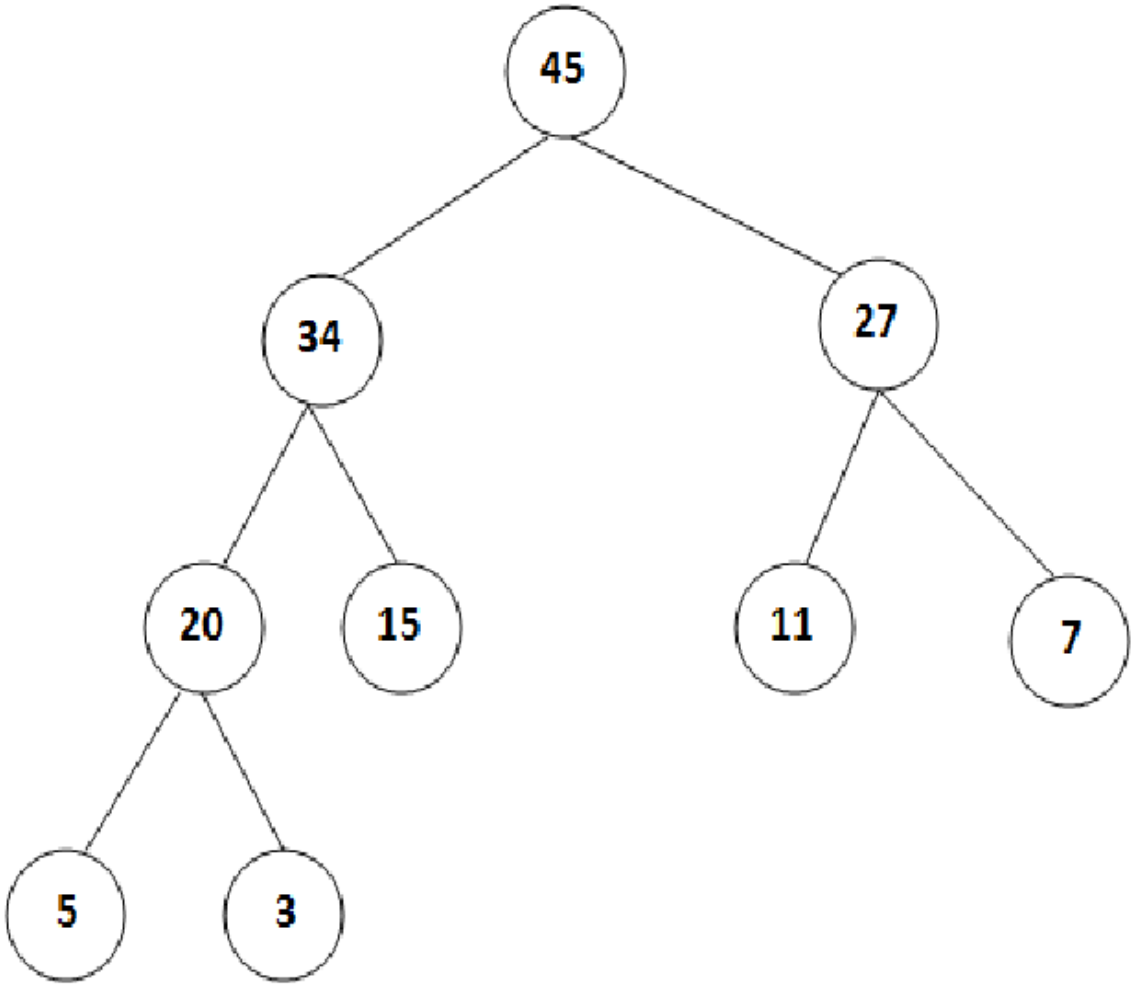
3. Efficiently solving problems: Heaps are commonly used in a variety of algorithms and data structures due to their efficient nature. By understanding heaps, you can apply them to solve problems like finding the kth largest or smallest element, implementing priority queues, and efficiently sorting data.

4. Analyzing time complexity: Understanding the time complexity of heap operations helps in determining the efficiency of algorithms that use heaps. For example, heap sort is an efficient sorting algorithm with a time complexity of O(n log n).

5. Understanding applications: Heap data structures have numerous applications in various fields of computer science, such as graph algorithms (e.g., Dijkstra's algorithm), scheduling algorithms, memory management, and more. Learning about heaps opens up possibilities for applying them in real-world scenarios.

By achieving these objectives, you will gain a strong foundation in heaps, enabling you to design efficient algorithms and solve complex problems effectively.

Heap:
A Heap is a special Tree-based data structure in which the tree is a complete binary tree.



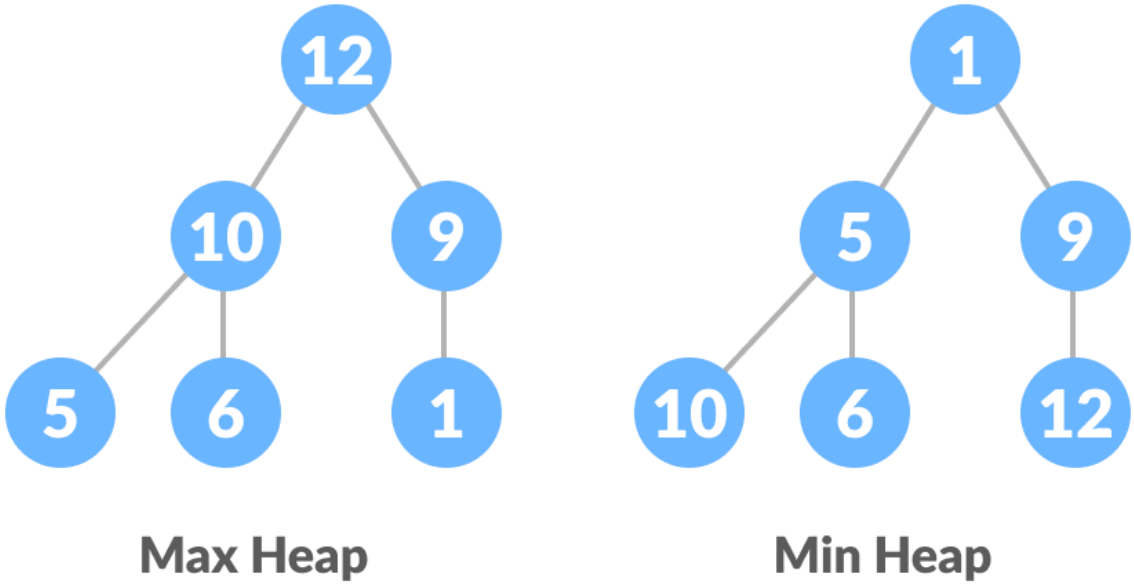
Heap Data Structure

A heap is a specialized tree-based data structure that satisfies the heap property. It is commonly implemented as a binary tree, although other variants exist, such as ternary heaps or Fibonacci heaps. The heap property ensures that the value of each node is either greater than or equal to (in a max-heap) or less than or equal to (in a min-heap) the values of its child nodes.

Types of Heap Data Structure
Generally, Heaps can be of two types:

1. **Max-Heap**: In a Max-Heap the key present at the root node must be greatest among the keys present at all of it's children. The same property must be recursively true for all sub-trees in that Binary Tree.

2. **Min-Heap**: In a Min-Heap the key present at the root node must be minimum among the keys present at all of it's children. The same property must be recursively true for all sub-trees in that Binary Tree.



In a binary heap, the structure is typically represented as an array, where the parent-child relationships are determined by the indices of the array elements. The root of the heap is stored at index 0, and for any element at index i, its left child is at index 2i+1, and its right child is at index 2i+2.

There are two key operations performed on a heap:

1. Insertion: To insert a new element into the heap, it is added at the next available position in the array and then "bubbled up" or "percolated up" to its correct position by repeatedly comparing it with its parent. If the heap property is violated, the elements are swapped until the property is restored.

2. Extraction: The extraction operation removes the root element of the heap, which is typically the minimum (in a min-heap) or maximum (in a max-heap). After removal, the last element in the array is moved to the root position, and then it is "bubbled down" or "percolated down" by comparing it with its children and swapping with the smaller (in a min-heap) or larger (in a max-heap) child until the heap property is satisfied.

Heaps are commonly used to implement priority queues, where the highest-priority element can be efficiently accessed and removed. They are also utilized in various algorithms, such as heap sort, which involves building a max-heap (or min-heap) from an array and repeatedly extracting the maximum (or minimum) element to achieve a sorted output. The efficiency of heap operations is crucial. The insertion and extraction operations typically have a time complexity of O(log n), where n is the number of elements in the heap. The heap property ensures that the minimum or maximum element can be accessed in constant time (O(1)).
Overall, heaps are valuable data structures that enable efficient access to the minimum or maximum element, making them essential for solving a variety of algorithmic problems.

Operations of Heap Data Structure:

- **Heapify:** a process of creating a heap from an array.
- **Insertion:** process to insert an element in existing heap time complexity O(log N).
- **Deletion:** deleting the top element of the heap or the highest priority element, and then organizing the heap and returning the element with time complexity O(log N).
- **Peek:** to check or find the first (or can say the top) element of the heap.

Applications of Heap Data Structure:

- **Priority Queues:** Priority queues can be efficiently implemented using Binary Heap because it supports insert(), delete() and extractmax(), decreaseKey() operations in **O(log N)** time.
- **Binomial Heap** and **Fibonacci Heap** are variations of Binary Heap. These variations perform union also in **O(log N)** time which is an O(N) operation in Binary Heap.
- **Order statistics:** The Heap data structure can be used to efficiently find the kth smallest (or largest) element in an array.

Implementation of Heap Data Structure:
The following code shows the implementation of a max-heap.
Let's understand the maxHeapify function in detail:
maxHeapify is the function responsible for restoring the property of the Max Heap. It arranges the node i, and its subtrees accordingly so that the heap property is maintained.

1. Suppose we are given an array, arr[] representing the complete binary tree. The left and the right child of ith node are in indices 2*i+1 and 2*i+2.
2. We set the index of the current element, i, as the 'MAXIMUM'.
3. If arr[2 * i + 1] > arr[i], i.e., the left child is larger than the current value, it is set as 'MAXIMUM'.
4. Similarly if arr[2 * i + 2] > arr[i], i.e., the right child is larger than the current value, it is set as 'MAXIMUM'.
5. Swap the 'MAXIMUM' with the current element.
6. Repeat steps 2 to 5 till the property of the heap is restored.

```
#include <iostream>
```

```cpp
using namespace std;

// Function to maintain the max heap property
void maxHeapify(int arr[], int n, int i) {
    int largest = i;              // Initialize the largest as the root
    int left = 2 * i + 1;         // Left child index
    int right = 2 * i + 2;        // Right child index

    // Check if the left child is larger than the root
    if (left < n && arr[left] > arr[largest])
        largest = left;

    // Check if the right child is larger than the current largest
    if (right < n && arr[right] > arr[largest])
        largest = right;

    // If the largest is not the root, swap the largest with the root
    if (largest != i) {
        swap(arr[i], arr[largest]);

        // Recursively call maxHeapify on the affected subtree
        maxHeapify(arr, n, largest);
    }
}

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

int main() {
    int arr[] = { 4, 10, 3, 5, 1 };
    int n = sizeof(arr) / sizeof(arr[0]);

    cout << "Array before heapify: ";
    printArray(arr, n);

    maxHeapify(arr, n, 0);

    cout << "Array after heapify: ";
    printArray(arr, n);

    return 0;
}
```

Lab Task

Task 1

Heap is a special case of *balanced* binary tree data structure where the root-node key is compared with its children and arranged accordingly. If α has child node β then,

key(α) ≥ key(β)

As the value of parent is greater than that of child, this property generates Max Heap. Based on this criteria, a heap can be of two types:

- **Min-Heap** − Where the value of the root node is less than or equal to either of its children.
- **Max-Heap** − Where the value of the root node is greater than or equal to either of its children.

Note: A *balanced* binary tree is one in which no leaf nodes are 'too far' from the root. For example, one definition of balanced could require that all leaf nodes have a depth that differ by at most 1.

For your task, create an abstract class named **Heap** with the following:

Attributes:

- ✓ int *arr;
- ✓ int capacity;
- ✓ int current;

Functions:

virtual void insert(int) = 0;

Inserts a new key of type int in the Heap.

Note: Write constructor and destructor as well.

Task 2

Using the abstract **Heap** class, make a child class named **MinHeap** having following functionalities:

- ✓ void insert(int key) -> inserts a new key of type int in the Heap maintaining the Min Heap property.
- ✓ int getMin() -> returns the root element of Min Heap. Time Complexity of this operation is O(1).

Note: You can write a print() function as well which prints the array through which heap is implemented for verification purposes.

Task 3

Using the abstract **Heap** class, make a child class named **MaxHeap** having following functionalities:

- ✓ void insert(int key) -> inserts a new key of type int in the Heap maintaining the Max Heap property.
- ✓ int getMax() -> returns the root element of Max Heap. Time Complexity of this operation is O(1).

Note: You can write a print() function as well which prints the array through which heap is implemented for verification purposes.

Task 4

Convert **MaxHeap** into **MinHeap** in linear time use already build **MaxHeap**.