Definition and Classification of Heap

Priority Queues

Before introducing a Heap, let's first talk about a Priority Queue.

Wikipedia: a priority queue is an abstract data type similar to a regular queue or stack data structure in which each element additionally has a "priority" associated with it. In a priority queue, an element with high priority is served before an element with low priority.

In daily life, we would assign different priorities to tasks, start working on the task with the highest priority and then proceed to the task with the second highest priority. This is an example of a Priority Queue.

A common misconception is that a Heap is the same as a Priority Queue, which is not true. A priority queue is an abstract data type, while a Heap is a data structure. Therefore, a Heap is not a Priority Queue, but a way to implement a Priority Queue.

There are multiple ways to implement a Priority Queue, such as array and linked list. However, these implementations only guarantee $O(1)$time complexity for either insertion or deletion, while the other operation will have a time complexity of $O(N)$ On the other hand, implementing the priority queue with Heap will allow both insertion and deletion to have a time complexity of $O(\log N)$So, what is a Heap?

In this chapter, we will learn to:

1. Understand the Heap data structure.
2. Understand Max Heap and Min Heap.
3. Understand the insertion and deletion of a Heap.
4. Implement a Heap.

Definition of Heap

According to Wikipedia, a **Heap** is a special type of binary tree. A heap is a binary tree that meets the following criteria:

- Is a **complete binary tree**;
- The value of each node must be **no greater than (or no less than)** the value of its child nodes.

A Heap has the following properties:

- Insertion of an element into the Heap has a time complexity of $O(\log N)$;
- Deletion of an element from the Heap has a time complexity of $O(\log N)$
- The maximum/minimum value in the Heap can be obtained with $O(1)$ time complexity.

Classification of Heap

There are two kinds of heaps: **Max Heap** and **Min Heap**.

- Max Heap: Each node in the Heap has a value **no less than** its child nodes. Therefore, the top element (root node) has the **largest** value in the Heap.
- Min Heap: Each node in the Heap has a value **no larger than** its child nodes. Therefore, the top element (root node) has the **smallest** value in the Heap.

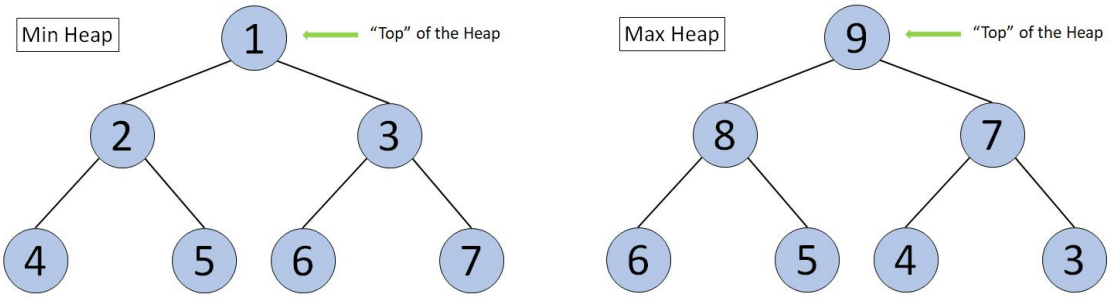

Diagram of a Min Heap and a Max Heap

Heap Insertion

Insertion means adding an element to the Heap. After inserting the element, the properties of the Heap should remain unchanged.

For max heap : 1) Should be a complete binary tree

2) root value should be greater than the children value

For min heap : 1)Should be a complete binary tree

3) root value should be less than the children values

Common applications of Heap

In most programming languages, Heaps are already built-in. Therefore, we usually do not need to implement a Heap from scratch. However, to use Heap adequately, we need to understand how Heap is commonly used.

In this chapter, we will learn how to:

- Construct a Max Heap and a Min Heap.
- Insert elements into a Heap.
- Get the top element of a Heap.
- Delete the top element from a Heap.
- Get the length of a Heap.
- Perform time and space complexity analysis for common applications that use a Heap.

Construct a Heap

Constructing a Heap means initializing an instance of a Heap. All methods of Heap need to be performed on an instance. Therefore, we need to initialize an instance before applying the methods. When creating a Heap, we can simultaneously perform the **heapify** operation. Heapify means converting a group of data into a Heap.

Time complexity: $O(N)$

Space complexity: $O(N)$

Inserting an Element

Insertion means inserting a new element into the Heap. Note that, after the new element is inserted, properties of the Heap are still maintained.

Time complexity: $O(\log N)$

Space complexity: $O(1)$

Getting the Top Element of the Heap

The top element of a Max heap is the maximum value in the Heap, while the top element of a Min Heap is the smallest value in the Heap. The top element of the Heap is the most important element in the Heap.

Time complexity: $O(1)$

Space complexity: $O(1)$

Deleting the top element

Note that, after deleting the top element, the properties of the Heap will still hold. Therefore, the new top element in the Heap will be the maximum (for Max Heap) or minimum (for Min Heap) of the current Heap.

Time complexity: $O(\log N)$

Space complexity: $O(1)$

Getting the Length of a Heap

The length of the Heap can be used to determine the size of the current heap, and it can also be used to determine if the current Heap is empty. If there are no elements in the current Heap, the length of the Heap is zero.

Time complexity: $O(1)$

Space complexity: $O(1)$

Overall Time And Space Complexity of a heap

Space and Time Complexity

Heap method	Time complexity	Space complexity
Construct a Heap	O(N)	O(N)
Insert an element	O(log N)	O(1)
Get the top element	O(1)	O(1)
Delete the top element	O(log N)	O(1)
Get the size of a Heap	O(1)	O(1)

N is the number of elements in the heap.

Application of Heap

Heap is a commonly used data structure in computer science. In this chapter, we will cover several applications of Heap.

1. Heap Sort
2. The Top-K problem
3. The K-th element

Heap Sort

Heap Sort sorts a group of unordered elements using the Heap data structure.

The sorting algorithm using a **Min Heap** is as follows:

1. Heapify all elements into a Min Heap.
2. Record and delete the top element.
3. Put the top element into an array T that stores all sorted elements. Now, the Heap will remain a Min Heap.
4. Repeat steps 2 and 3 until the Heap is empty. The array T will contain all elements sorted in ascending order.

The sorting algorithm using a **Max Heap** is as follows:

1. Heapify all elements into a Max Heap.
2. Record and delete the top element.
3. Put the top element into an array T that stores all sorted elements. Now, the Heap will remain a Max Heap.
4. Repeat steps 2 and 3 until the Heap is empty. The array T will contain all elements sorted in descending order.

Complexity Analysis:

Let N be the total number of elements.

Time complexity: $O(N\log N)$

Space complexity: $O(N)$

Inshort:

Using a heap to obtain a sorted array involves converting the unsorted array into a heap, then popping the elements from the heap one at a time and adding them to the new sorted array. The following video will walk through this process step by step for a Min Heap to obtain an array sorted in ascending order. The process for a Max Heap would be the same, except that the sorted array would be in descending order.

The Top K Problem

The Top K Problem - Approach 1

Use the Heap data structure to obtain Top K's largest or smallest elements.

Solution of the Top K largest elements:

1. Construct a Max Heap.
2. Add all elements into the Max Heap.
3. Traversing and deleting the top element (using pop() or poll() for instance), and store the value into the result array T.
4. Repeat step 3 until we have removed the K largest elements.

Solution of the Top K smallest elements:

1. Construct a Min Heap.
2. Add all elements into the Min Heap.
3. Traversing and deleting the top element (using pop() or poll() for instance), and store the value into the result array T.
4. Repeat step 3 until we have removed the K smallest elements.

Complexity Analysis:

Time complexity: $O(K \log N + N)$

- Steps one and two require us to construct a Max Heap which requires $O(N)$time using the previously discussed heapify method. Each element removed from the heap requires $O(\log N)$time; this process is repeated Ktimes. Thus the total time complexity is $O(K \log N + N)$

Space complexity: $O(N)$

- After step 2, the heap will store all N elements.

The Top K Problem - Approach 2

Use the **Heap** data structure to obtain Top K's largest or smallest elements.

Solution of the Top K largest elements:

1. Construct a Min Heap with size K.
2. Add elements to the Min Heap one by one.
3. When there are K elements in the "Min Heap", compare the current element with the top element of the Heap:
4. If the current element is no larger than the top element of the Heap, drop it and - proceed to the next element.
5. If the current element is larger than the Heap's top element, delete the Heap's top element, and add the current element to the Min Heap.
6. Repeat Steps 2 and 3 until all elements have been iterated.

Now the K elements in the Min Heap are the K largest elements.

Solution of the Top K smallest elements:

1. Construct a Max Heap with size K.
2. Add elements to the Max Heap one by one.
3. When there are K elements in the "Max Heap", compare the current element with the top element of the Heap:
4. If the current element is no smaller than the top element of the Heap, drop it and proceed to the next element.
5. If the current element is smaller than the top element of the Heap, delete the top element of the Heap, and add the current element to the Max Heap.
6. Repeat Steps 2 and 3 until all elements have been iterated.

Now the K elements in the Max Heap are the K smallest elements.

Complexity Analysis:

Time complexity: $O(N \log K)$

- Steps one and two will require $O(K \log K)$ time if the elements are added one by one to the heap, however using the heapify method, these two steps could be accomplished in $O(K)$ time. Steps 3 and 4 will require $O(\log K)$ time each time an element must be replaced in the heap. In the worst-case scenario, this will be done
- $N - K$ times. Thus the total time complexity is $O((N - K) \log K + K \log K)$which simplifies to $O(N \log K)$.

Space complexity: $O(K)$

- The heap will contain at most K elements at any given time.

Similar to top k problem

he K-th Element

The K-th Element - Approach 1

Use the Heap data structure to obtain the K-th largest or smallest element.

Solution of the K-th largest element:

1. Construct a Max Heap.
2. Add all elements into the Max Heap.
3. Traversing and deleting the top element (using pop() or poll() for instance).
4. Repeat Step 3 K times until we find the K-th largest element.

Solution of the K-th smallest element:

1. Construct a Min Heap.
2. Add all elements into the Min Heap.
3. Traversing and deleting the top element (using pop() or poll() for instance).

4. Repeat Step 3 K times until we find the K-th smallest element.

Complexity Analysis:

Let N be the total number of elements.

Time complexity: $O(K \log N + N)$

- Steps one and two require us to construct a Max Heap which requires $O(N)$ time using the previously discussed heapify method. Each element removed from the heap requires $O(\log N)$time; this process is repeated Ktimes. Thus the total time complexity is $O(K \log N + N)$

Space complexity: $O(N)$

- After step 2, the heap will store all Nelements.

The K-th Element - Approach 2

Use the **Heap** data structure to obtain the K-th largest or smallest element.

Solution of the K-th largest element:

1. Construct a Min Heap with size K.
2. Add elements to the Min Heap one by one.
3. When there are K elements in the "Min Heap", compare the current element with the top element of the Heap:
    1. If the current element is not larger than the top element of the Heap, drop it and proceed to the next element.
    2. If the current element is larger than the Heap's top element, delete the Heap's top element, and add the current element to the "Min Heap".
4. Repeat Steps 2 and 3 until all elements have been iterated.

Now the top element in the Min Heap is the K-th largest element.

Solution of the K-th smallest element:

1. Construct a Max Heap with size K.
2. Add elements to the Max Heap one by one.
3. When there are K elements in the Max Heap, compare the current element with the top element of the Heap:

    1. If the current element is not smaller than the top element of the Heap, drop it and proceed to the next element;
    2. If the current element is smaller than the top element of the Heap, delete the top element of the Heap, and add the current element to the Max Heap.

4. Repeat Steps 2 and 3 until all elements have been iterated. Now the top element in the Max Heap is the K smallest element.

Complexity Analysis:

Time complexity: $O(N \log K)$

- Steps one and two will require $O(K \log K)$ time if the elements are added one by one to the heap, however using the heapify method, these two steps could be accomplished in $O(K)$time. Steps 3 and 4 will require $O(\log K)$ time each time an element must be replaced in the heap. In the worst-case scenario, this will be done $N - K$ times. Thus the total time complexity is $O((N - K) \log K + K \log K)$ which simplifies to $O(N \log K)$

Space complexity: $O(K)$

- The heap will contain at most K elements at any given time.

Implementation of Heap Using STL

```
class heap{
private:
    vector<int>nums;
    int size;
public:
    heap(vector<int>nu){
        nums = nu;
        size = nu.size();
    }

    void build(){
        make_heap(nums.begin(),nums.end());
    }
    void push(int a){
        nums.push_back(a);
        push_heap(nums.begin(),nums.end());
    }
    void pop(){
        pop_heap(nums.begin(),nums.end());
        nums.pop_back();
    }
    // After using sorting  method
    // it is no longer heap
    void sort(){
        sort_heap(nums.begin(),nums.end());
    }
    void print(){
        for(auto it : nums){
            cout << it << " ";
        }
        cout << endl;}
};
```