CSCI 2270 – Data Structures

Recitation 9, October 2018

Priority Queues and Heaps

Objectives

1. Priority Queues
2. Priority Queue ADT
3. Heaps
4. Exercise
    a. Challenge 1 (Silver Badge)
    b. Challenge 2 (Gold Badge)

1. Priority Queues

In some situations we may need to find the minimum/ maximum element among a collection of elements. We can do this with the help of Priority Queue ADT. A priority queue ADT is a data structure that supports Insert and DeleteMin(returns and removes the minimum element) or DeleteMax(returns and removes the maximum element).

These operations are equivalent to EnQueue and DeQueue operations of a queue. The difference in that, in priority queues, the order in which the elements enter the queue may not be the same in which they were processed. An example application of a priority queue is job scheduling, which is prioritized instead of serving in first come first serve.

A priority queue is called an *ascending - priority queue*, if the item with smallest key has the highest priority. Similarly, a priority queue is said to be a *descending - priority* queue of the item with largest key has the highest priority.

2. Priority Queue ADT

The following operations make priority queues an ADT.

Main Priority Queues Operations
A priority queue is a container of elements, each having an associated key.

Priority Queues and Heaps

- Insert(key, data): Inserts data with key to the priority queue. Elements are ordered based on key.
- DeleteMin/ DeleteMax: Remove and return the element with the smallest/ largest key.
- GetMinimum/ GetMaximum: Return the element with the smallest/ largest key without deleting it.

Auxiliary Priority Queues Operation

- kth smallest/ kth largest: returns the kth smallest/ kth largest key in priority queue.
- Size: Returns number of elements in priority queue.
- Heap Sort: Sorts the elements in the priority queue based on priority(key).

Applications of Priority Queue

1. Data compression.
2. Shortest path algorithms

Priority Queue Implementations

There are many possible options of implementing a priority queue:
However the below 2 are the most used:

Ordered Array Implementation

Elements are inserted into array in a sorted order based on key field. Deletions are performed at only one end. Insertions complexity: O(n) DeleteMin complexity: O(1).

Binary Heap Implementation

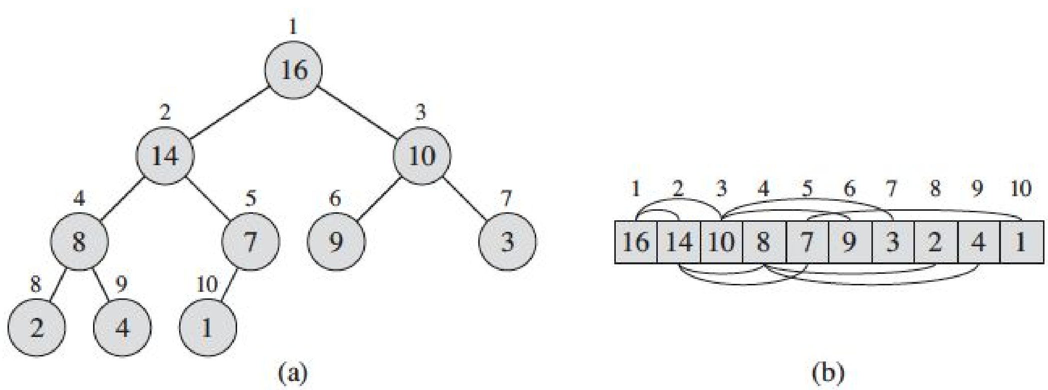
Insert and delete takes O(logn) complexity (Search takes O(n)) and finding the maximum or minimum element takes O(1).

3. Heaps

What is a Heap?

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.

The (binary) heap data structure can be thought of as an array object that we can view as a nearly complete binary tree. Each node of the tree corresponds to an element of the array. The tree is completely filled on all levels except possibly the lowest, which is filled from the left up to a point.



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

arr[i/2]	Returns the parent node
arr[2*i]	Returns the left child
arr[(2*i)+1]	Returns the right child

Priority Queues and Heaps

Since a heap of n elements is based on a complete binary tree, its height is theta(lg n). We will see that the basic operations on heaps run in time at most proportional to the height of the tree and thus take O(lg n) time.

> Tip: Why do we start from index 1?
>
> On most computers, the **LEFT** procedure can compute **2*i** in one instruction by simply shifting the binary representation of i left by one bit position. Similarly, the RIGHT procedure can quickly compute **2*i+1** by shifting the binary representation of i left by one bit position and then adding in a 1 as the low-order bit. The **PARENT** procedure can compute **i/2** by shifting i right one bit position.
>
> *Source: Stack Overflow*

Types of Heaps

Min Heap

In this type of heap, the value of parent node will always be less than or equal to the value of child node across the tree and the node with lowest value will be the root node of tree.

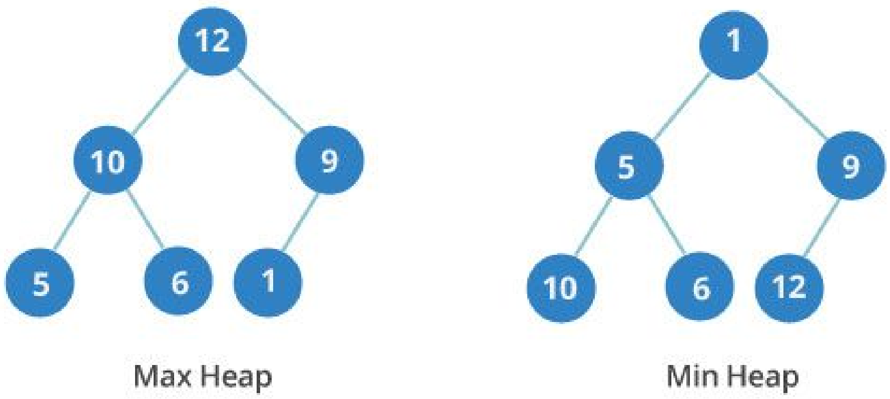
Max Heap

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.

CSCI 2270 – Data Structures

Recitation 9, October 2018

Priority Queues and Heaps



Heapifying an Element

Let's assume that we have a heap having some elements N which are stored in array Arr. The way to convert this array into a heap structure is the following. We pick a node in the array, check if the left subtree and the right subtree are max heaps, in themselves and the node itself is a max heap (it's value should be greater than all the child nodes)

To do this we will implement a function that can maintain the property of max heap. When it is called, the following function max_heapify assumes that the binary trees rooted at Left and Right of i are max-heaps, but that Arr[i] might be smaller than its children, thus violating the max-heap property. max_heapify lets the value at A[i] to "float down" in the max-heap so that the subtree rooted at index i obeys the max-heap property.

```
void max_heapify (int Arr[ ], int i, int N)
{
    int left = 2*i                //left child
    int right = 2*i +1            //right child
    if(left<= N and arr[left] > Arr[i] )
          largest = left;
    else
         largest = i;
    if(right <= N and arr[right] > arr[largest] )
        largest = right;
```

```
    if(largest != i )
    {
        swap (&Ar[i] , &Arr[largest]);
        max_heapify (Arr, largest,N);
    }
}
```

Building a Max Heap

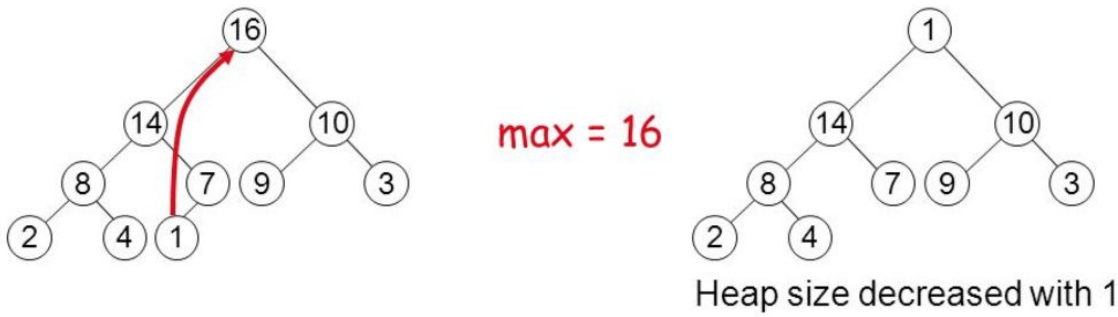
Now let's say we have N elements stored in the array Arr indexed from 1 to N. They are currently not following the property of max heap. So, we can use the procedure max-heapify in a bottom-up manner to convert the array into a max-heap.

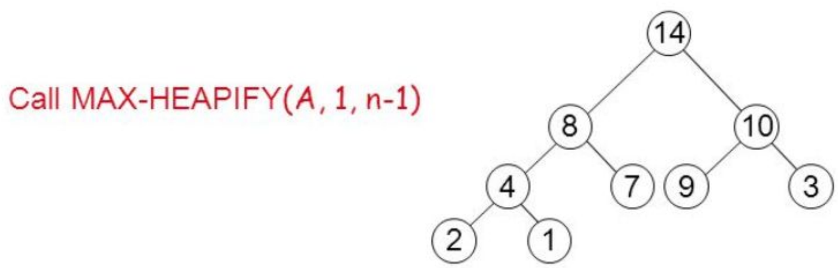
Lets observe a property which states: A N element heap stored in an array has leaves indexed by N/2+1, N/2+2 , N/2+3 …. upto N. From the above property we observed that elements from Arr[N/2+1] to Arr[N] are leaf nodes,and each node is a 1 element heap. We can use max_heapify function in a bottom up manner on remaining nodes, so that we can cover each node of tree.

```
void build_maxheap (int Arr[ ])
{
    for(int i = N/2 ; i >= 1 ; i-- )
    {
        max_heapify (Arr, i, N) ;
    }
}
```

Extract Max

The maximum element can be found at the root, which is the first element of the array. We return and remove the root and replace it with the last element of the heap In this operation. Since the last element of heap will be placed at index 1, it will violate the property of max-heap. Therefore, we need to perform max_heapify on node 1 to restore the heap property.
This procedure will become more clear from the following example of extracting max=16 from the following heap:





Inserting an Element

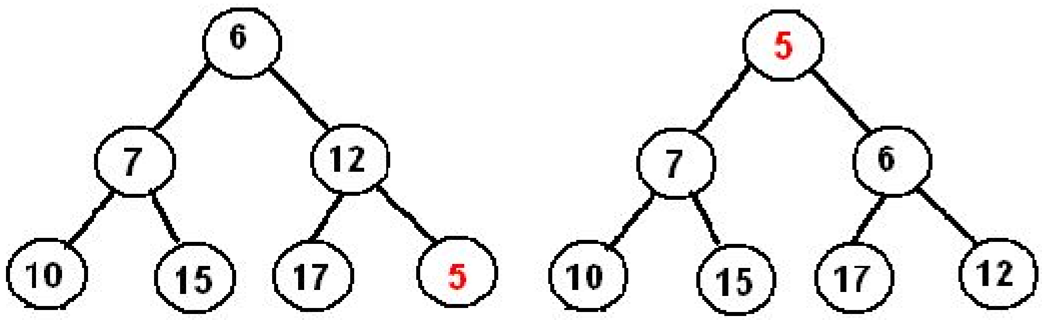
The new element is initially appended to the end of the heap (as the last element of the array). The heap property is repaired by comparing the added element with its parent and moving the added element up a level (swapping positions with the parent). This process is called "percolation up". The comparison is repeated until the parent is larger than or equal to the percolating element.

While we explained the above operations on Max-heap, we can have similar operations on Min-Heap as well; For example - Let's see how insertion into a MIN-HEAP looks like.

CSCI 2270 – Data Structures

Recitation 9, October 2018

Priority Queues and Heaps



	6	7	12	10	15	17	5
	6	7	5	10	15	17	12
	5	7	6	10	15	17	12
0	1	2	3	4	5	6	7

CSCI 2270 – Data Structures

Recitation 9, October 2018

Priority Queues and Heaps

Exercise

A. Silver Badge Problem

1. Download the **MinHeap.hpp** and **HeapMain.cpp** files from moodle.
2. Complete the **extractMin** functionality.
3. Call heapify on each call after extracting the smallest value from the heap.

B. Gold Badge Problem

1. You are given k number of **sorted linkedlists**.
2. Now using the above **minHeap** solution, merge all the lists and return a **single sorted linkedlist.**