Priority Queues & Heaps

Topics

1. Introduction to Priority Queues
2. Heaps (Binary Heap) & Implementation details of Heaps
3. Heap Sort & its Implementation
4. Usage details of Heaps in C++, Java & Python
5. Problems & Solutions
6. Important References

Introduction to Priority Queue

In computer science, a **priority queue** is an **abstract data type**, which is like a regular queue or stack data structure, but where additionally each element has a "priority" associated with it. In a priority queue, an element with high priority is served before an element with lower priority. If two elements have the same priority, they are served according to their order in the queue.

Queues are a standard mechanism for ordering tasks on a first-come, first-served basis. However, some tasks may be more important or timely than others (higher priority). Priority queues store tasks using a partial ordering based on priority and ensure highest priority task ahead of queue. Heaps are the underlying data structure of priority queues. This underlying heap can be following min heap properties, or max heap properties, depending on the application of the priority queue.

Priority Queue primarily supports the following three basic operations:

1. `getTop()` : Fetching the top priority element.
2. `insert()` : Insertion of an element.
3. `deleteTop()` : Deleting the top priority element.

Note: Here the top priority element can be a maximum priority or minimum priority element. It totally depends on the implementation of the priority queue.

Implementation details of a Priority Queue using different Data Structures

We can try and implement priority queue using different data structures. But all the implementations of the priority queue have a major flaw and that has to do with increased Time Complexity. It turns out that Heaps are naturally the best data structure to implement a priority queue, as shown in the following table below:

Data Structure	`getTop()`	`insert()`	`deleteTop()`
Unordered Array	$O(N)$	$O(1)$	$O(N)$
Ordered Array	$O(1)$	$O(N)$	$O(1)$
Unordered Linked List	$O(N)$	$O(1)$	$O(N)$
Ordered Linked List	$O(1)$	$O(N)$	$O(1)$
Balanced Binary Search Tree	$O(\log_2(N))$	$O(\log_2(N))$	$O(\log_2(N))$
Binary Heap	$O(1)$	$O(\log_2(N))$	$O(\log_2(N))$

Heap is the maximally efficient implementation, and in fact priority queues are often referred to as "**heaps**", regardless of how they may be implemented.
There are two kind of priority queue: **max-priority queue** and **min-priority queue**.
Max-heap is used for max-priority queue and Min-heap is used for min-priority queue.

Note: A *heap* data structure should not be confused with *the heap* which is a common name for the pool of memory from which dynamically allocated memory is allocated. The term was originally used only for the data structure.

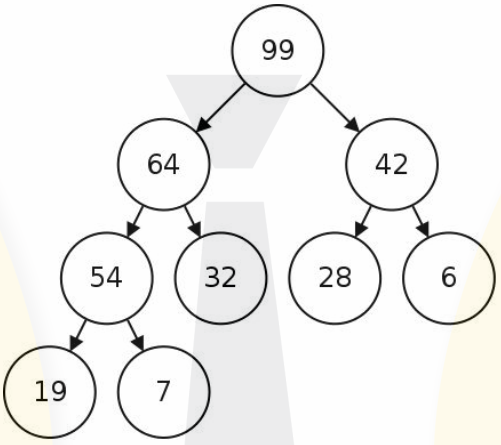
Applications of Priority Queue

- Priority Scheduling of Processes in OS
- Dijkstra's Algorithm for Single Source Shortest Path
- Prim's Algorithm for Minimum Spanning Tree
- Heap Sort

Heaps

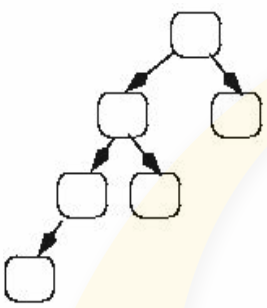
A **heap** is a binary tree (in which each node contains a Comparable key value), with two special properties:

A. The **ORDER** property: For every node **n**, the value in **n** is **greater than or equal to** the values in its **children** (and thus is also greater than or equal to all of the values in its subtrees).

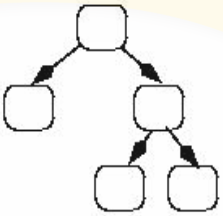


This heap is called a *max heap* because the root node is the maximum. We could also have a *min heap* where each node's value is **less than or equal to** that of its children. Here we will only talk about max heaps, though the same ideas apply to min heaps as well. Using a heap to represent a priority queue, we will always take the root item next as this is the one with the largest priority. Notice that there is no particular relationship between sibling nodes, the only relationship that matters is the parent to the child.

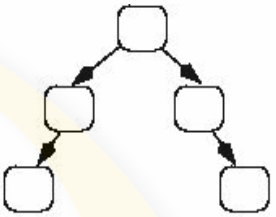
B. The **SHAPE** property (See the image below, and the following properties to understand the Shape property of a heap in a much deeper sense. A heap always follows **Complete Binary Tree Property**):
   1. All leaves are either at depth **`d`** or **`d-1`** (for some value **`d`**).
   2. All of the leaves at depth **`d-1`** are to the **right** of the leaves at depth **`d`**.
   3. (a). There is at most 1 node with just 1 child.
      (b). That child is the **left** child of its parent, and
      (c). It is the **rightmost** leaf at depth **`d`**.



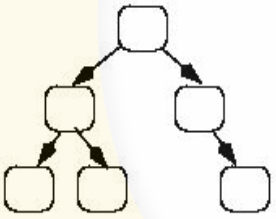
NO: violates shape property 1



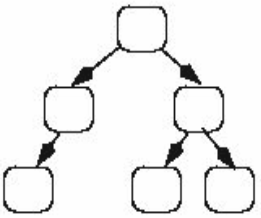
NO: violates shape property 2



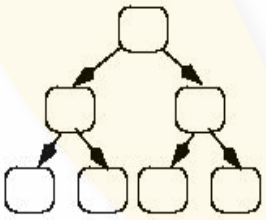
NO: violates shape property 3(a)



NO: violates shape property 3(b)

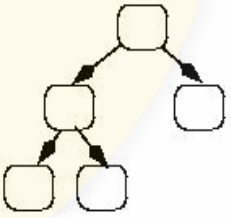


NO: violates shape property 3(c)



YES!

YES!



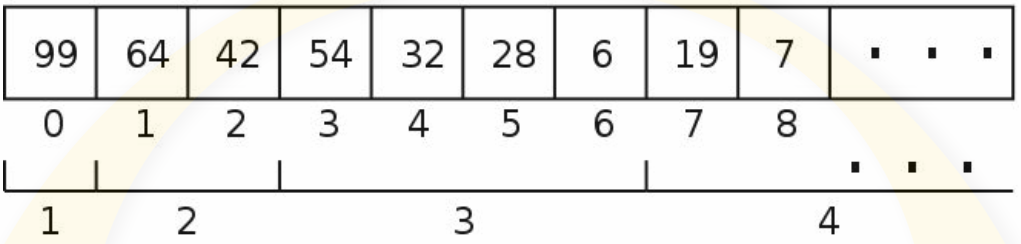
YES!

Implementation Details of Heaps

Heaps supports a number of different operations:

1. `create()` : Create an empty heap.
2. `insert()` : Add an element to the heap.
3. `getTop()` : Return the top element.
4. `deleteTop()` : Remove the top element from the heap.

How these are implemented depends on how the heap is implemented. We can implement trees using a linked structure as we did for binary search trees. However, we can also implement a tree with an array. Doing this with a heap, we will store the root in array location 0, then the two children of the root in the next two locations. The rule that the heap must be balanced and left-aligned implies that there is only one array location where each item can be stored.



a. To find the left child of a node, we can use this formula:

```
left_child(node) = node * 2 + 1
```

b. The right child is directly after the left child, so that formula is:

```
right_child(node) = node * 2 + 2
```

c. We can find the parent of any node fairly easily as well:

```
parent(node) = (node - 1) / 2
```

Implementation of Binary Max-Heap:

```
#include <iostream>
#include <cstdio>
#include <cstdlib>
#include <algorithm>

using namespace std;

typedef struct heap {
    int *array;
    int capacity;
    int size;
} maxHeap;

// Heap Creation (Max Heap) : capacity
maxHeap Create(int capacity) {
    maxHeap h = (struct heap*)malloc(sizeof(struct heap));
    if(h == NULL) {
        cout << "Memory Error" << endl;
        return NULL;
    }
    h->size = 0;
    h->capacity = capacity;
    h->array = (int *)malloc(sizeof(int)*capacity);
    if(h->array == NULL) {
        cout << "Memory Error" << endl;
        return NULL;
    }
    return h;
}
```

```c
// Getting Left Child
int GetLeftChild(maxHeap h, int i) {
    int left = 2*i+1;
    if(h->size > left)
        return left;
    else
        return -1;
}

// Getting Right Child
int GetRightChild(maxHeap h, int i) {
    int right = 2*i+2;
    if(h->size > right)
        return right;
    else
        return -1;
}

// Getting Parent
int GetParent(maxHeap h, int i) {
    if(i == 0)
        return -1;
    else
        return h->array[(i-1)/2];
}

// Heapify : Maintaining Heap property
void heapify(maxHeap h, int i) {
    int l = GetLeftChild(h, i);
    int r = GetRightChild(h, i);
    if(l == -1 && r == -1)
        return;

    int max_index, temp;
    max_index = i;
    if(l != -1 && h->array[l] > h->array[i])
        max_index = l;
    if(r != -1 && h->array[r] > h->array[max_index])
        max_index = r;

    if(max_index != i) {
        temp = h->array[i];
        h->array[i] = h->array[max_index];
        h->array[max_index] = temp;
        heapify(h, max_index);
    }
}

// Getting Maximum Element (Max Heap) : O(1)
int GetMaximum(maxHeap h) {
    if(h->size == 0)
        return -1;
    return h->array[0];
}
```

```
// Insertion : O(logn)
void insert(maxHeap h, int val) {
    int i, temp;
    i = h->size;
    h->array[i] = val;
    h->size++;
    while(i != 0 && GetParent(h,i) < h->array[i]) {
        temp = GetParent(h,i);
        h->array[(i-1)/2] = h->array[i];
        h->array[i] = temp;
        i = (i-1)/2;
    }
}


// Deletion (Top Element) : O(logn)
void deletion(maxHeap h) {
    if(h->size == 0)
        cout << "Empty Heap" << endl;
    else {
        h->array[0] = h->array[h->size-1];
        h->size--;
        heapify(h,0);
    }
}


// Print Heap
void printHeap(maxHeap h) {
    for(int i = 0; i < h->size; i++) {
        cout << h->array[i] << " ";
    }
    cout << endl;
}

int main() {
    int capacity;
    cin >> capacity;
    maxHeap h = Create(capacity);
    insert(h,3);        insert(h,10);
    insert(h,7);        insert(h,9);
    print_heap(h);
    deletion(h);
    print_heap(h);
    insert(h,8);        insert(h,6);
    insert(h,45);       insert(h,45);
    print_heap(h);
    deletion(h);
    print_heap(h);
    return 0;
}
```

- **Implementation of a Binary Max Heap in Java.**
- **Implementation of a Binary Max Heap in Python.**

Heap Sort

Check the Pseudo code of Heapsort from here.
Time Complexity of Heapsort: $O(n \times \log_2(n))$

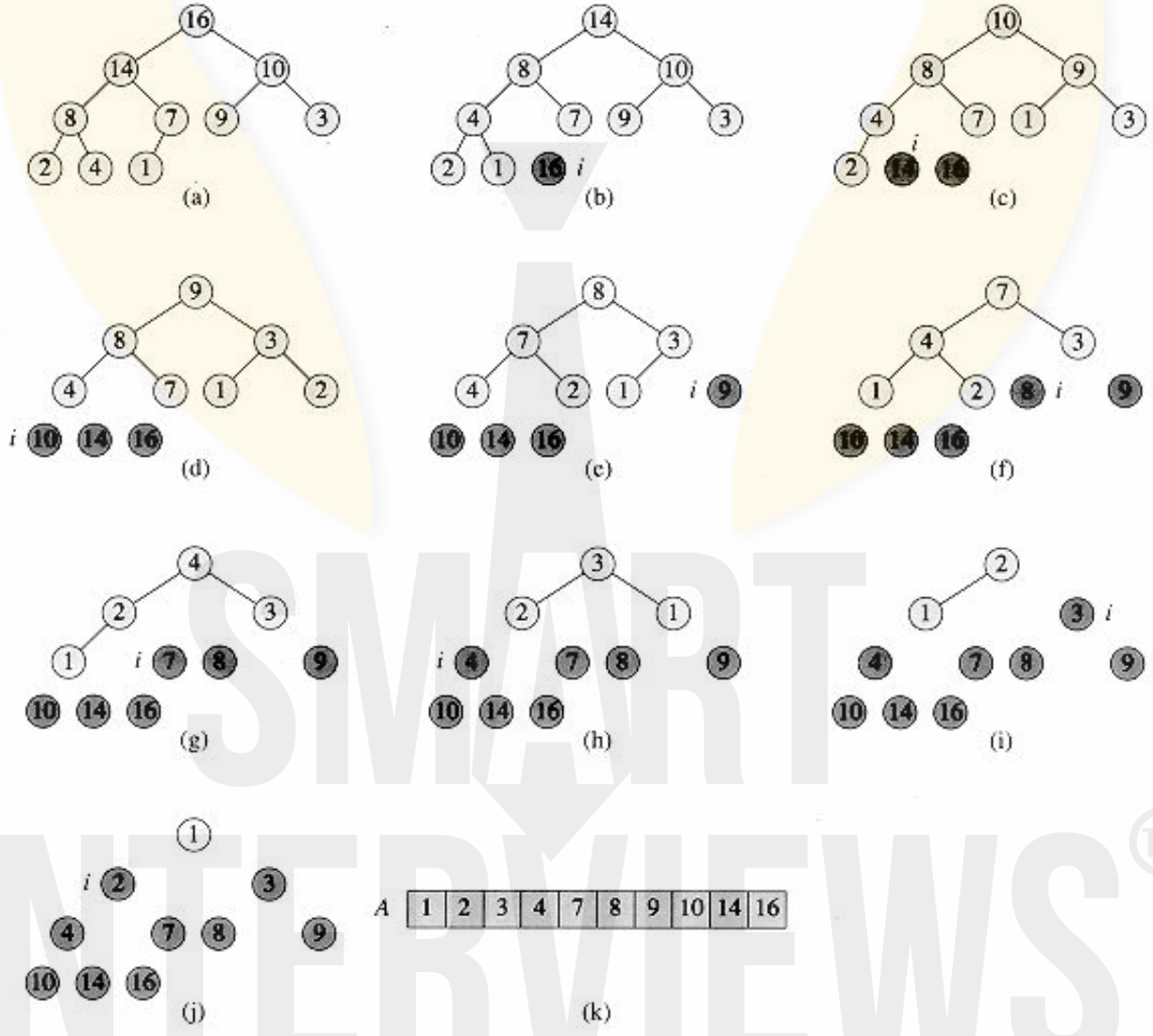
Implementation of Heap Sort in C/C++:

```
#include <iostream>
#include <cstdio>
using namespace std;

void print(int a[], int low, int high) {
   for(int i = low; i <= high; i++)
       cout<< a[i] <<" ";
   cout << endl;
}

void buildmaxheap(int a[],int size) {
   for(int i = (size-1)/2; i >= 0; i--)
       heapify(a, i, size);
}

void heapify(int a[], int i, int size) {
   int left = 2*i+1;
   int right = 2*i+2;
   int max_index = i;
   if(left > size-1 && right > size-1)
            return;
   if(left <= size-1) {
       if(a[left] > a[max_index])
          max_index = left;
    }

   if(right <= size-1) {
       if(a[right] > a[max_index])
          max_index = right;
   }

   if(max_index != i) {
       int temp = a[i];
       a[i] = a[max_index];
       a[max_index] = temp;
       heapify(a, max_index, size);
    }
}

void heapsort(int a[], int h_size) {
   buildmaxheap(a, h_size);
   for(int i = h_size-1; i > 0; i--) {
       int temp = a[h_size-1];
       a[h_size-1] = a[0];
       a[0] = temp;
       h_size--;
       heapify(a, 0, h_size);
   }
}
```

```
int main() {
    int n;
    cin >> n;
    int a[n+1];
    for(int i = 0; i < n; i++)
        cin >> a[i];
    heapsort(a, n);
    print(a, 0, n-1);
    return 0;
}
```

- **Implementation of Heap Sort in Java.**
- **Implementation of Heap Sort in Python.**

Example Visualization of Heap Sort



Usage Details of Heaps in C++, Java & Python

Declaration:

- `priority_queue<int> max_pq;` `//max heap`
- `priority_queue<int, vector<int>, greater<int>> min_pq;` `//min heap`

Functions

- Push new element : `pq.push();`
- Get top element (max/min) : `pq.top();`
- Delete top element : `pq.pop();`
- Check if its empty : `pq.empty();`
- Get size : `pq.size();`

More details at **C++ - `std::priority_queue`**.

Usage Details of `java.util.PriorityQueue` in Java can be found here.
Tutorial on how to use `java.util.PriorityQueue` in Java.

Usage Details of `heapq.PriorityQueue` in Python can be found here.
Tutorial on how to use `heapq.PriorityQueue` in Python.

Problems & Solutions

1. Given an array of n integers, devise an algorithm to get median of the sub-array **0 to *i*** $\forall i$ from 0 to n-1.

   Example:
   Let's assume array is `[5, 2, 3, 4]`
   `M(i)` represents the median of array from 0 to i
   `M(0) = 5`, Sorted Array: `[5]`
   `M(1) = (2+5)/2 = 3`, Sorted Array: `[2, 5]`
   `M(2) = 3`, Sorted Array: `[2, 3, 5]`
   `M(3) = (3+4)/2 = 3`, Sorted Array: `[2, 3, 4, 5]`

   Solution:
   a. Observe that in order to find the median of a sub-array [0, i], we only require the center two elements of the sorted array [0, i],
   b. In other words, we need the max element from the smaller half elements of the array and the min element from the larger half elements of the array.
   c. Use max-heap to store the smaller half and min-heap to store the larger half.
   d. The two heaps should differ in size by at max 1 at any point of time:
   `|size(min_heap) - size(max_heap)| <= 1`

2. Given an array of ***n*** elements, where *each element* is *at most* ***k*** *away* from its *actual position* in the *sorted version* of the array (**k**-sorted array), devise an algorithm that sorts the given **k**-sorted array in $O(n \times \log_2(k))$ time.
   Solution:
   a. Use Min-heap of size **k+1**, and insert the 1st **k+1** elements of the given array in the heap. This heap is bound to have the smallest element of the array, as the given array is **k**-sorted
   b. ∀**i** (**i** > **k+1**), insert **ar[i]** in the Min-heap, and do a **delMin()** to get the next element of the sorted version of the array.

3. Given an array of integers, find **k** smallest elements in the array.
   Solution
   a. Create **k** sized Max-heap using the first **k** elements of the array. In short, we are assuming that the first **k** elements are the **k** smallest elements in the array.
   b. ∀**i** (**i** > **k**), do the following:

```
if ar[i] < getTop(max_heap):
      delMax(); insert(ar[i]);
```

4. There are given ***n*** ropes of different lengths, we need to connect these ropes into one rope. The cost to connect two ropes is equal to sum of their lengths. We need to connect the ropes with minimum cost.
   Solution: The solution follows greedy approach:
   a. Take the two smallest length ropes, connect them and add it back to your list of ropes. Repeat until you have a single connected rope.
   b. This can be solved by using a Min-heap. Create a Min-heap and insert all lengths into the Min-heap. Perform two **delMin()** operations, add the two ropes and push the new rope length back into the min-heap. Repeat until heap size becomes 1.

5. Given **k** sorted arrays of size ***n*** each, merge them and print the sorted output.
   Solution:
   a. Create a Min-heap of size **k** and insert 1st element of all the **k** arrays into the heap.
   b. Repeat following step ***n*×*k*** times:
      i. Get minimum element from heap (minimum is always at root) and print it, and track the array number to which this extracted element belongs (say **m**).
      ii. If array **m** has no element left, then go back to the above step**,** otherwise insert next element from the array **m** in the Min-heap.

6. Given a row and column wise sorted matrix of size $n \times n$, print all elements in sorted order.
   Example:

1	5	10	20
2	8	15	30
12	15	18	35
20	25	30	40

   Solution:
   a. Create a Min-heap with the first row of the matrix.
   b. Repeat following step $n \times n$ times.
      i. Get minimum element from heap (minimum is always at root) and print it, and track the column number to which this extracted element belongs (say `m`).
      ii. If column `m` has no element left, then go back to the above step**,** otherwise insert next element from the column `m` in the Min-heap.

Important References

1. **Heap Sort Pseudocode**
2. **Difference between Strict, Complete & Full Binary Trees**
3. **java.util.PriorityQueue usage with Custom Objects in Java**
4. **Application of Priority Queue: Huffman Coding**