# Heap

by

Jane Alam Jan

# Introduction

Often there are problems like the following

*We are given a list of* ***n*** *elements; there are three types of operations.*

1) *Insert an item into the list*
2) *Find the minimum item in the list*
3) *Remove the minimum item from the list*

## Idea 1

This problem doesn't look that tough. We can easily store the list in an array or in a linked-list. But if we watch carefully, for adding an item, we can easily add the item in the end. But if we are asked to find the minimum item, or remove the minimum item, then we have to traverse the whole list.

So, if the array has **n** items and we are asked to find and remove the minimum item; then we will traverse the whole list. And thus, we need **n** operations for the first deletion. Since one item has been removed, so, we need `n – 1` operations for the second deletion. So, if we delete `n` items, we need `n + (n – 1) + (n - 2) + ... 1 = n*(n+1)/2` operations. Suppose that `n = 1000000` and imagine that the computer performs $10^9$ operations per second; this idea will take around 500 seconds. This is way too large. So, if we delete **k** times, the complexity of the idea will be **O(n*k)**.

## Idea 2

Another idea is, we can sort the array and can easily find and remove the minimum item, since it exists in the end. But still if we add items in the array, the array may lost its sorted property, and if we want to sort it after adding one or more items, then still the total time will be huge. For example you can sort the array in **O(n*log(n))** time, but if we add items `k` times, then the total time will be **O(n*log(n)*k)**, which can be huge. Say `k = 1000000`, and `n = 1000000`, then it's easy to show that it will work worse than the previous idea.

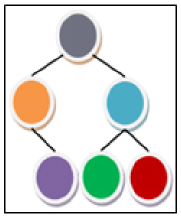
So, both the ideas may not work, since they can perform worse in some cases. So, using them is risky sometimes.

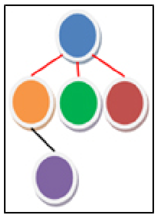
## Heap

Heap is a tree-based data structure. We keep the numbers in the tree maintaining some beautiful properties. That's why the time complexity to solve the given problem is sufficiently low. At first we will describe the idea, and we will show why it works well. And in second part we will describe the implementation idea, actually how to implement it easily.

### Idea

Before advancing, we should know about binary trees. A binary tree is a tree with the property that ***each node can have at most two children.***


**Binary Tree**


**Not a binary tree (3 children for root)**

Now for heap, we will maintain three properties

1) Each node can have at most two children (binary tree)
2) The number containing by a node will be less than the numbers containing by its children
3) When adding a node in the tree, we will add it in the upper leftmost free position

### Insertion

Suppose we have the following numbers – `4, 5, 3, 2, 8, 9, 6` and `1`. Now let's make the heap, using these numbers. The steps are noted below.

1) Initially there is no element, so, we can add `4` in the tree as the first node or root (fig 1).
2) Now, we have `5`, since the left most free position is the left child of node `4`, so we add `5` as the left child of `4` (property `3`) (fig 2).
3) Now, we have `3`, since the left most free position is the right child of node `4`, so we add `3` as the right child of `4` (fig 3.1). But `3` is less than `4`, which violates property `2`. So, we will swap `3` and `4` to maintain property `2` (fig 3.2).

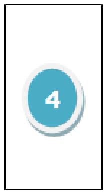


**Figure 1**

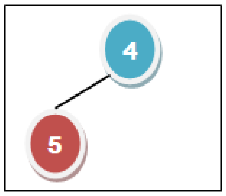


**Figure 2**

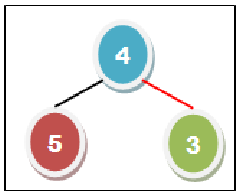


**Figure 3.1**

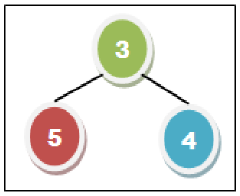


**Figure 3.2**

4) We have 2 now; the leftmost free position is the left child of the node 5. We add 2 as the left child of 5 (fig 4.1). But 2 can't be child of 5 (property 2), so we swap 2 and 5 (fig 4.2). Now, see that 2 can't be child of 3, so, we swap 2 and 3 (fig 4.3).

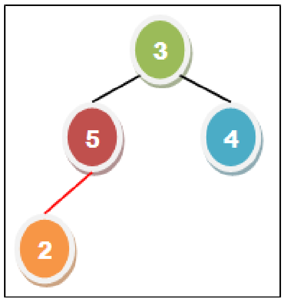


Figure 4.1

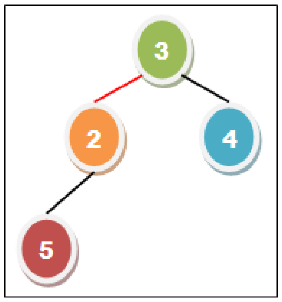


Figure 4.2

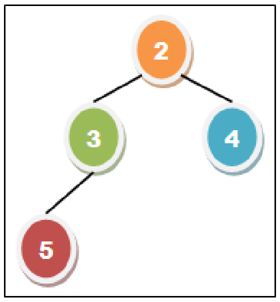


Figure 4.3

5) The leftmost free position for 8 is the right child of 3. So, we add it here (fig 5).
6) The leftmost free position for 9 is the left child of 4. So, we add it here (fig 6).
7) The leftmost free position for 6 is the right child of 4. So, we add it here (fig 7).

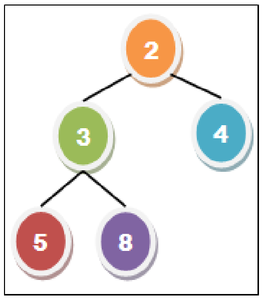


Figure 5

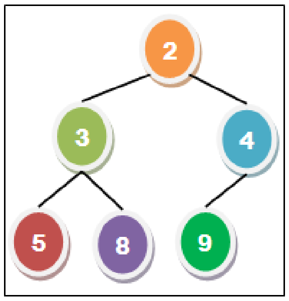


Figure 6

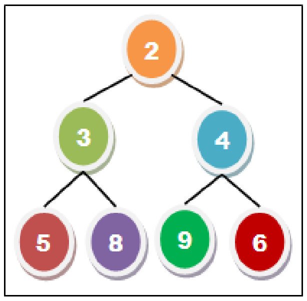


Figure 7

8) Finally we have 1. The leftmost free position for 1 is the left child of 5. We place it here (fig 8.1). But since 1 can't be a child of 5; we swap 1 and 5. Now is 1 is child of 3 (fig 8.2). Still its violating rule 1, thus we swap 1 and 3 (fig 8.3). Again 1 can't be child of 2, so we swap 1 and 2 (fig 8.4).

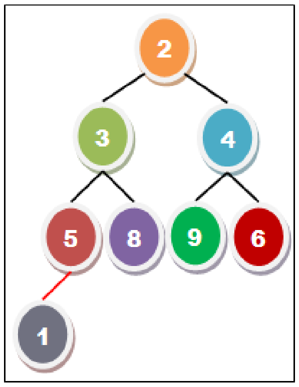


Figure 8.1

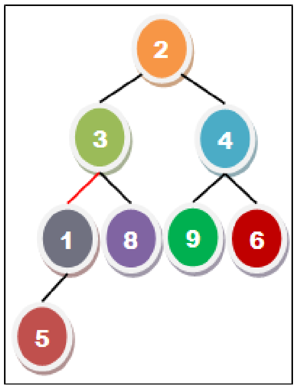


Figure 8.2

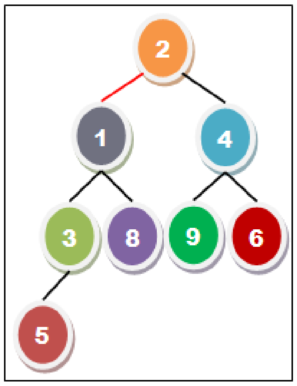


Figure 8.3

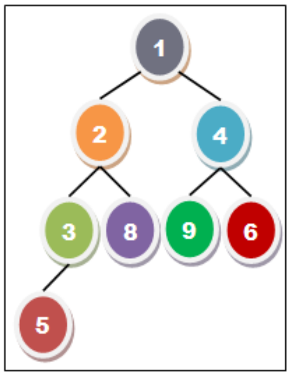
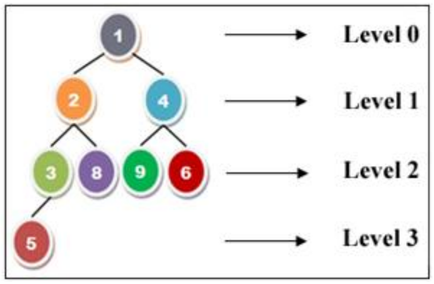


Figure 8.4

## Cost for Insertion

Now we are inserting numbers, but we may need some swap operations to maintain the heap properties. In worst case how many swap operations are needed? Before answering the question we should see the following part.

For convenience we partition the tree into some levels.



**Partitioning the tree into levels**

It's clear that in level 0, there can be only one node, level 1 can have two nodes, level 2 can have 4 nodes and level 3 can have 8 nodes (though in the picture, there is only one node in level 3, but we can surely add 7 more nodes here). So, we can surely say,

```
Level 0 contains at most 1 node = 2^0 node.
Level 1 contains at most 2 nodes = 2^1 nodes.
Level 2 contains at most 4 nodes = 2^2 nodes.
Level 3 contains at most 8 nodes = 2^3 nodes.
...
Level n contains at most 2^n nodes.
```

Three important facts should be noted here. They are

1. If a node (other than the root) is in level **`k`**, its parent is surely in level **`k - 1`**.
2. We can't add an item in level **`k`** unless level is **`k - 1`** full.
3. If a node is added in level **`k`**, then at most **`k`** swaps are need.

Actually the way we have built the tree, we are sure that fact 1 and 2 will always be maintained.

For the 3rd fact, imagine that we are trying to add a new item in level `3`. And in worst case, we would have to swap it with its parent which is in level `2`. After that we may need another swap with the current parent which lies in level `1`. And finally we may need another swap with its current parent which lies in level 0. So, at most 3 swaps are needed (see the 8th iteration of the heap building example given above).

Now if we add an item in a heap in level 4, then at most 4 swaps are needed. Because if a swap is possible, after 1 swap the node will be in level 3, after another possible swap it will go to level 2, and if another swap is possible, then it will surely go to level 1, and finally, if another swap is possible then it will go to level 0. After that no swaps can be needed. So, at most 4 swaps are needed. So, fact 3 is also correct.

Now suppose, there are **n** nodes in the tree, then what would be the highest level for the heap?

```
n = 1; highest level is 0, floor(log(n)) = 0
n = 2, 3; highest level is 1, floor(log(n)) = 1
n = 4 to 7; highest level is 2, floor(log(n)) = 2
n = 8 to 15; highest level is 3, floor(log(n)) = 3
...
So, for any n; highest level is floor(log (n))
```

[Here, **floor** function returns the integer part of a number, for example, `floor(1.99)=1`, `floor(7.00)=7`, the base for the **log** is 2].

For a tree which has **n** nodes, has **k** (=**floor**(log (n))) as the highest level, so, if we add a new item in the tree, if the **k**th level is not full, then in worst case we need **k** operations. If the **k**th level is full, then the new item will be added in (**k+1**)th level, so, in worst case we need **k+1** operations. For simplicity, we assume that, in worst case we need `log(n)` operations in both cases. So, the complexity for adding an item is **O(log(n))**. Thus, the complexity for adding **k** items is **O(k*logn)**.

## Finding Minimum Item

If we see the tree, we see that the root will always be the minimum item in the tree. Since we know that for heap, every child will be greater than its parent. So, inductively it's clear that the root will contain the minimum number; otherwise we have not maintained the heap property.

## Cost for Finding Minimum Item

Since the root can be found without any calculations. So, the complexity is O(1).

## Removing Minimum Item

So far we have seen some efficient ideas to add an item or finding the minimum item in the tree. Now we want to remove the minimum item from the tree efficiently.

Let's delete 3 minimum elements from the tree in figure 8.4. The steps are noted below.

1) The root contains the minimum element, so 1 is the minimum element. At first we remove the bottom rightmost element (or you can say the last element) from the tree and copy its content to the root. So, we remove 1 from the tree and copy 5 to root (fig 9.1).

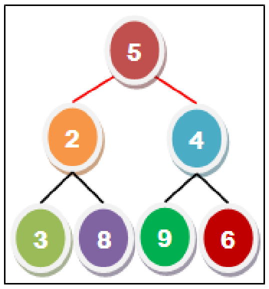


**Figure 9.1**

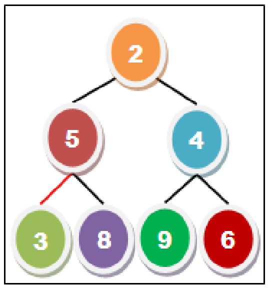


**Figure 9.2**

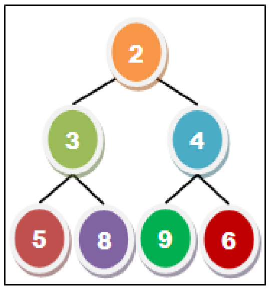


**Figure 9.3**

2) As we can see that the tree has lost its heap property 2 (that is the value of a node should be less than the value of its children). So, from 2 and 4, we pick the minimum one, that is 2, and we swap it with 5 (fig 9.2).
3) Still the tree is not a heap since, 3 is less than 5, this we swap them (fig 9.3). Now it's a heap.
4) Let's delete the next minimum number, that is 2, so, we can do the same procedure that is we will delete 2 and we replace it by 6 (fig 10.1).

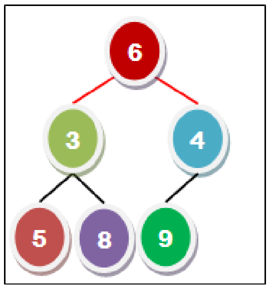


**Figure 10.1**

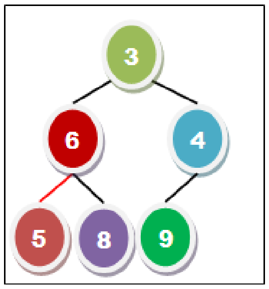


**Figure 10.2**

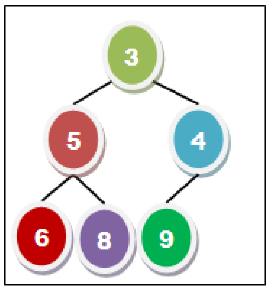


**Figure 10.3**

5) Since it's not a heap, we swap 6 with its minimum child that is 3 (fig 10.2).
6) Still there is a conflict, because 5 is smaller than 6, thus we swap 6 and 5 (fig 10.3). And it's a heap.
7) Let's delete the current minimum that is 3. So, we replace 3 by 9 (fig 11.1).
8) And swap 9 with its minimum child 4, and we get a heap (fig 11.2).

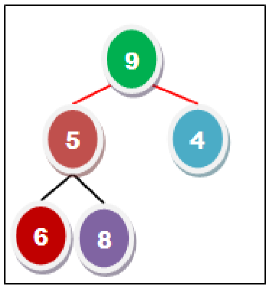


**Figure 11.1**

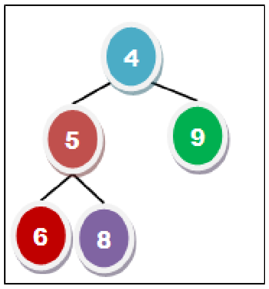


**Figure 11.2**

## Cost for Deleting Minimum Item

As we can see that, for a deletion we need to swap some elements. Let this number be **K**. Each time after a swap we can see that the number goes to the next level, that means the level is increased by 1, and since the number of levels is `log(n)`, So, the complexity for the deletion of the minimum number would be `O(log(n))`.

## The Way to Code

So, we now want to code this structure. At first we number the nodes in the following way.

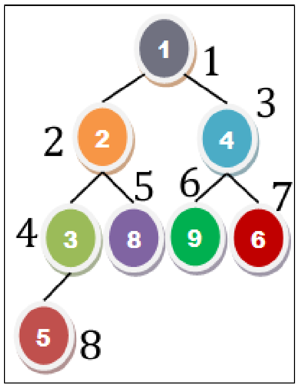


**Figure: 12**

If we see carefully, it's clear that if a node's index is **`p`**, its left child's index is **`2*p`**, and its right child's index is **`2*p+1`**. So, we will use an array to store the numbers. For example, for fig 12 the array will be:

```
index          1 2 3 4 5 6 7 8
content        1 2 4 3 8 9 6 5
```

For example, in index 3, we kept 4, and, so, its left child will be the number in index 2*3=6, and the right child will be the in index 2*3+1=7. As we can see that in index 6, we have 9, and in index 7 we have 6, which are the left and right child of 4 respectively.

And again, we can say that if a node's index is **p**, its parent's index will be **p/2** (integer division). For example, in index 5 we have 8, so, its parent's index will be 5/2=2 which is 2, and in index 2 there is 2, which is 8's parent.

```
#include <stdio.h>
#include <algorithm>
using namespace std;

const int NN = 100; // heap size

struct heap {
    int myarray[NN+1]; // myarray to store the numbers as heap, 1 indexed
    int n;  // the number of nodes in my array
    heap() { // constructor
        clear(); // we clear the heap
    }
    void clear() { // initialize the heap
        n = 0; // initially there are no nodes in the heap
    }
    void insert( int K ) { // inserting an element K in the heap
        if( n == NN ) { // the heap is full
            printf("cannot insert any more element, the heap is full\n");
            return;
        }
        ++n; // so, we have a new element, we increased n before adding
             // the element because we start from index 1
        myarray[n] = K; // inserted the element at the rightmost position
        int p = n; // for keeping the current position
        while( p > 1 ) { // p = 1 means we are on the root, and its a heap
            int pr = p / 2; // pr is the parent of p
            if( myarray[pr] > myarray[p] ) { // parent is greater than child
                swap( myarray[pr], myarray[p] );
                p = pr; // now the new position of the current element is pr
            }
            else break; // otherwise its a heap, so we can stop here
        }
    }
    int remove() { // removing the minimum element from the heap
        if( n == 0 ) { // is the heap is empty
            printf("The heap is empty, cannot delete.\n");
            return -1;
        }
        int K = myarray[1]; // first element in the heap is the minimum
        myarray[1] = myarray[n]; // brought the last element in 1st position
        n--; // as we removed one element, now we need to maintain the heap

        int p = 1; // as we moved the rightmost element in index 1
        while( 2 * p <= n ) { // means p has at least one child, if 2*p > n
                              // we are sure that p is in the last level
            int ch = 2 * p; // contains the index of the child
            if( 2 * p + 1 <= n ) { // right child exists
                if( myarray[ch] > myarray[ch+1] ) // right child is smaller
                                                  // than left child
                    ch++; // ch contains the index of the right child
            }
            if( myarray[p] > myarray[ch] ) { // so, current node is larger
                                             // than its child
                swap( myarray[p], myarray[ch] );
                p = ch; // new position of the current element
            }
            else break; //current node is smaller than its children, so heap
        }
        return K; // as we stored the minimum element in K
    }
    void print() { // printing the heap
        printf("Number of elements: %d\n", n);
        for( int i = 1; i <= n; i++ ) printf("%d ", myarray[i]);
        printf("\n");
    }
};

int main() {
    heap A; // created a heap

    A.clear();

    // testing
    A.insert(4);
    A.print();

    A.insert(5);
    A.print();

    A.insert(3);
    A.print();

    A.insert(2);
    A.print();

    A.insert(8);
    A.print();

    A.insert(9);
    A.print();

    A.insert(6);
    A.print();

    A.insert(1);
    A.print();

    printf("Deleting %d\n", A.remove());
    A.print();
    printf("Deleting %d\n", A.remove());
    A.print();
    printf("Deleting %d\n", A.remove());
    A.print();
    return 0;
}
```

## Applications

We use heap for Dijkstra. And it’s quite use useful for some problems. And the same idea is used for Segment trees. And it’s a compact structure that means all the elements are saved consecutively in the array and no position is wasted (we can forget 0th index).

## Related Problems

1293 – Document Analyzer