Department of
Computer Science and Engineering

Title: Implementation of Priority Queue Using Heap

Data Structure Lab

CSE 106

Green University of Bangladesh

1 Objective(s)

- Implementation of Priority Queue Using Heap.
- To learn problem solving techniques using C.

2 Problem analysis

Priority queue is a type of queue in which every element has a key associated to it and the queue returns the element according to these keys, unlike the traditional queue which works on first come first serve basis. Thus, a max-priority queue returns the element with maximum key first whereas, a min-priority queue returns the element with the smallest key first

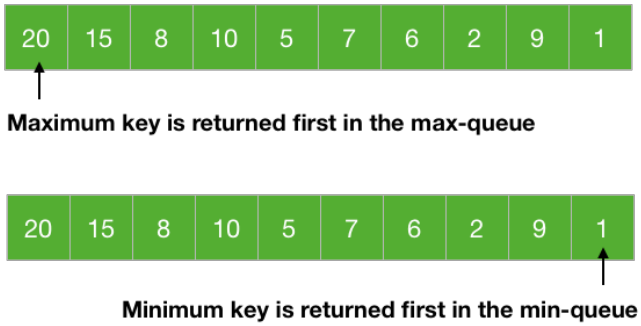
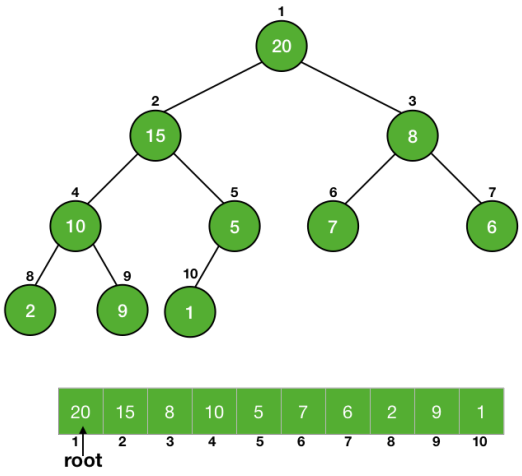


Figure 1: The max-queue and min-queue illustrated

There are mainly 4 operations we want from a priority queue:

- Insert → To insert a new element in the queue.
- Maximum/Minimum → To get the maximum and the minimum element from the max-priority queue and min-priority queue respectively.
- Extract Maximum/Minimum → To remove and return the maximum and the minimum element from the max-priority queue and min-priority queue respectively
- Increase/Decrease key → To increase or decrease key of any element in the queue.

A priority queue stores its data in a specific order according to the keys of the elements. So, inserting a new data must go in a place according to the specified order. This is what the insert operation does. The entire point of the priority queue is to get the data according to the key of the data and the Maximum/Minimum and Extract Maximum/Minimum does this for us. With these operations, we have fulfilled most of our demand of a priority queue i.e., to insert data into the queue and take data from the queue. But we may also face a situation in which we need to change the key of an element, so Increase/Decrease key is used to do that.

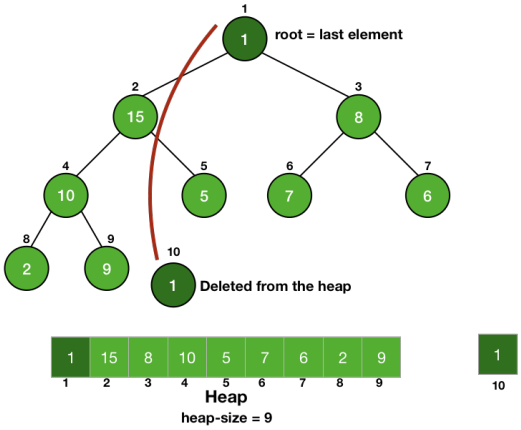


2.1 Maximum/Minimum

We know that the maximum (or minimum) element of a priority queue is at the root of the max-heap (or min-heap). So, we just need to return the element at the root of the heap.

2.2 Extract Maximum/Minimum

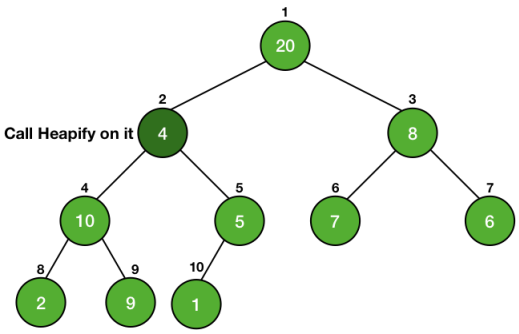
This is like the pop of a queue; we return the element as well as delete it from the heap. So, we have to return and delete the root of a heap. Firstly, we store the value of the root in a variable to return it later from the function and then we just make the root equal to the last element of the heap. Now the root is equal to the last element of the heap, we delete the last element easily by reducing the size of the heap by 1.



Doing this, we have disturbed the heap property of the root but we have not touched any of its children, so they are still heaps. So, we can call Heapify on the root to make the tree a heap again.

2.3 Increase/Decrease key

Whenever we change the key of an element, it must change its position to go in a place of correct order according to the new key. If the heap is a max-heap and we are decreasing the key, then we just need to check if the key became smaller than any of its children or not. If the new key is smaller than any of its children, then it is violating the heap property, so we will call Heapify on it.



In the case of increasing the key of an element in a max-heap, we might make it greater than the key of its parent and thus violating the heap property. In this case, we swap the values of the parent and the node and this is done until the parent of the node becomes greater than the node itself.

2.4 Insert

The insert operation inserts a new element in the correct order according to its key. We just insert a new element at the last of the heap and increase the heap size by 1. Since it is the last element, so we first give a very large value (infinity) in the case of min-heap and a very less value (-inf) in the case of max-heap. Then we just change the key of the element by using the Increase/Decrease key operation.

3 Algorithm

Algorithm 1: Maximum of heap

Input: Element

```
/* Algorithm for getting Maximum                                    */
1 MAXIMUM(A)
2 return A[1]
```

Algorithm 2: Extract Maximum of heap

Input: Element

```
/* Algorithm for Extract Maximum operation                          */
1 EXTRACT-MAXIMUM(A)
2 max = A[1]
3 A[1] = A[heap size]
4 heap size = heap size - 1
5 MAX-HEAPIFY(A, 1)
6 return max
```

Algorithm 3: Increase Key of Heap

Input: Element, Index, Key

```
/* Algorithm for Increase Key                                       */
1 INCREASE-KEY(A, i, key)
2 A[i] = key
3 while i > 1 and A[Parent(i) < A[i] do
4 |   swap(A[i], A[Parent(i)])
5 |   i = Parent(i)
6 end
```

Algorithm 4: Decrease Key of Heap

Input: Element, Index, Key

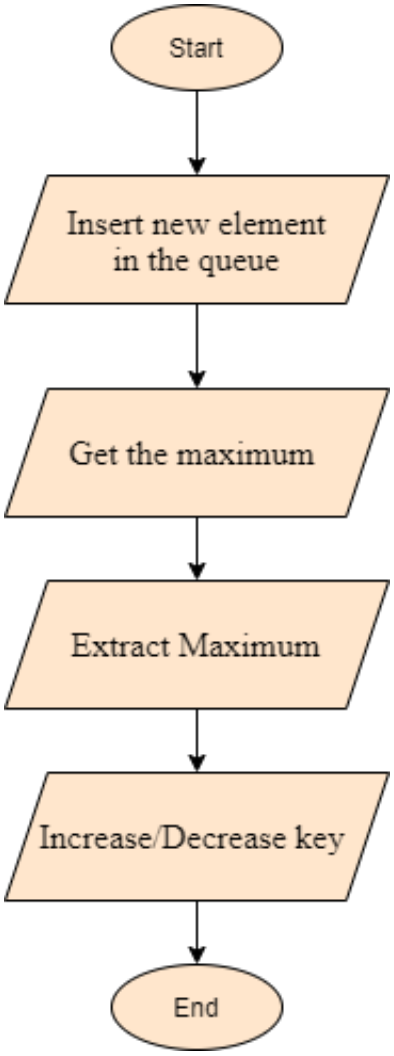
```
/* Algorithm for Decrease Key                                       */
1 DECREASE-KEY(A, i, key)
2 A[i] = key
3 MAX-HEAPIFY(A, i)
```

Algorithm 5: Decrease Key of Heap

Input: Element, Index, Key

```
/* Algorithm for Decrease Key                                       */
1 DECREASE-KEY(A, i, key)
2 A[i] = key
3 MAX-HEAPIFY(A, i)
```

4 Flowchart



5 Implementation in C

```c
#include <stdio.h>

int tree_array_size = 20;
int heap_size = 0;
const int INF = 100000;

void swap( int *a, int *b ) {
    int t;
    t = *a;
    *a = *b;
    *b = t;
}

//function to get right child of a node of a tree
int get_right_child(int A[], int index) {
    if((((2*index)+1) < tree_array_size) && (index >= 1))
        return (2*index)+1;
    return -1;
}

```

```
21 //function to get left child of a node of a tree
22 int get_left_child(int A[], int index) {
23     if(((2*index) < tree_array_size) && (index >= 1))
24         return 2*index;
25     return -1;
26 }
27
28 //function to get the parent of a node of a tree
29 int get_parent(int A[], int index) {
30     if ((index > 1) && (index < tree_array_size)) {
31         return index/2;
32     }
33     return -1;
34 }
35
36 void max_heapify(int A[], int index) {
37     int left_child_index = get_left_child(A, index);
38     int right_child_index = get_right_child(A, index);
39
40     // finding largest among index, left child and right child
41     int largest = index;
42
43     if ((left_child_index <= heap_size) && (left_child_index>0)) {
44         if (A[left_child_index] > A[largest]) {
45             largest = left_child_index;
46         }
47     }
48
49     if ((right_child_index <= heap_size && (right_child_index>0))) {
50         if (A[right_child_index] > A[largest]) {
51             largest = right_child_index;
52         }
53     }
54
55     // largest is not the node, node is not a heap
56     if (largest != index) {
57         swap(&A[index], &A[largest]);
58         max_heapify(A, largest);
59     }
60 }
61
62 void build_max_heap(int A[]) {
63     int i;
64     for(i=heap_size/2; i>=1; i--) {
65         max_heapify(A, i);
66     }
67 }
68
69 int maximum(int A[]) {
70     return A[1];
71 }
72
73 int extract_max(int A[]) {
74     int maxm = A[1];
75     A[1] = A[heap_size];
76     heap_size--;
77     max_heapify(A, 1);
78     return maxm;
```

```
}

void increase_key(int A[], int index, int key) {
    A[index] = key;
    while((index>1) && (A[get_parent(A, index)] < A[index])) {
        swap(&A[index], &A[get_parent(A, index)]);
        index = get_parent(A, index);
    }
}

void decrease_key(int A[], int index, int key) {
    A[index] = key;
    max_heapify(A, index);
}

void insert(int A[], int key) {
    heap_size++;
    A[heap_size] = -1*INF;
    increase_key(A, heap_size, key);
}

void print_heap(int A[]) {
    int i;
    for(i=1; i<=heap_size; i++) {
        printf("%d  ",A[i]);
    }
    printf("\n");
}

int main() {
    int A[tree_array_size];
    insert(A, 20);
    insert(A, 15);
    insert(A, 8);
    insert(A, 10);
    insert(A, 5);
    insert(A, 7);
    insert(A, 6);
    insert(A, 2);
    insert(A, 9);
    insert(A, 1);

    printf("Priority Queue : ");
    print_heap(A);
    printf("\n");
    increase_key(A, 5, 22);
    printf("Priority queue after increase key :");

    print_heap(A);
    printf("\n");
    decrease_key(A, 1, 13);
    printf("Priority queue after decrease key :");
    print_heap(A);
    printf("\n");
    printf("Maximum: %d\n\n", maximum(A));
    printf("Extract max:%d\n\n", extract_max(A));
    printf("Priority Queue : ");
    print_heap(A);
```

```
137
138     return 0;
139 }
```

6 Input/Output (Compilation, Debugging & Testing)

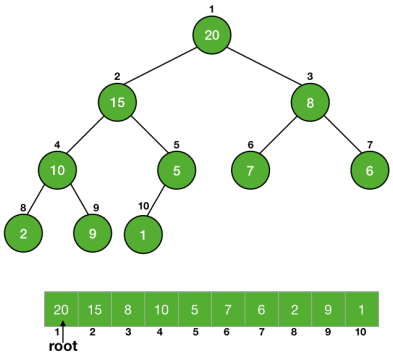
Output:
Priority Queue : 20 15 8 10 5 7 6 2 9 1
Priority queue after increase key : 22 20 8 10 15 7 6 2 9 1
Priority queue after decrease key : 20 15 8 10 13 7 6 2 9 1
Maximum: 20
Extract max: 20
Priority Queue : 15 13 8 10 1 7 6 2 9

7 Discussion & Conclusion

Based on the focused objective(s) to understand about the Priority Queue, the additional lab exercise made me more confident towards the fulfilment of the objectives(s).

8 Lab Task (Please implement yourself and show the output to the instructor)

1. Implement the following Max-Priority Queue Using Heap shown in the following figure and perform the operations.



2. Extract the Max item from the Queue and show the result.
3. Increase key 22 at position 5 (index 4) and show the result.
4. Decrease key 13 at position 1 (index 0) and show the result.

9 Lab Exercise (Submit as a report)

- Implement Min-Priority Queue Using Heap

10 Policy

Copying from internet, classmate, seniors, or from any other source is strongly prohibited. 100% marks will be *deducted* if any such copying is detected.