Lab Assignment 2

Heap is a data structure (Binary Tree) with the following properties.

→ It is a complete Binary Tree. All the levels except may be last level is full and last level is filled from left to right.

Implemented using arrays, linked lists etc.

Insertion in a Heap: When we insert a node in a heap, we expand the heap and then move the key up the heap to maintain heap property.

Algorithm:

```
heapInsert (A, data)
{
  heap_size = heap_size + 1
  Decrease Key (A, heap_size, data)
}

Decrease Key (A, i, data)
{
   A[i] = data
   while ( i > 1 && A [parent(i)] > A[i])
      {
        exchange A[i] and A [parent(i)]
        i = parent (i);

   }
   end.
```

Time complexity:

When a node is inserted at a level of h height.

Considering the worst case.

→ Adding a node $= O(1)$

→ Swapping the nodes (Bottom to top heapify) $= O(H)$

Total $= O(1) + O(H) = O(H)$

$\because$ Heap is a complete binary tree $\therefore$ $h = \log_2(N)$ where N = Total no. of nodes

$\therefore$ Overall complexity $= O(H) = O(\log_2 N)$.

re
inplace

Deletion in a heap: In a heap, the element with the highest priority ^?
(the root node) is deleted followed by the next.

Algorithm:

```
delete Min (int A[])
{
    min = A[1];
    A[1] = A[heap-size]
    heap-size = heap-size - 1
    heapify (A, 1)
    return min
}
```

Time complexity:

If node is to be deleted from heap of H height considering the worst case of time complexity, then:

→ Swapping root node with last element = $O(1)$
A[1]   A[heap_size]

→ Heapify at the root node (Top to bottom heapify) = $O(H)$

→ Total Complexity = $O(H+1) = O(H)$

∵ Heap is complete binary tree ∴ $H = \log_2 n$ where n is total no. of nodes

∴ Total = $O(H) = O(\log_2 n)$

Heapify: One of the important step for manipulating heap. Used when heap property is violated at ith node. In this we checked the children of ith node where heap property is violated. Then select the smallest child node and swap it ~~whit~~ with the ith node. Then check the violation. This is repeated till heap is created.

rithm:

```
heapify ( A[] , i)
{
  smallest = i
  l = 2 * i
  r = 2 * i + 1
  if (l ≤ heap_size and A[l] < A[smallest])
        smallest = l
   if (r ≤ heap_size and A[r] < A[smallest])
        smallest = r
   if ( smallest is not i)
        Exchange A[i] with A[smallest]
        heapify ( A, smallest)
  }
```

Time complexity:

In heapify function, we walk through the tree from top to bottom.

Now, the height of binary tree $= H = \log_2 N$ where N is number of nodes.

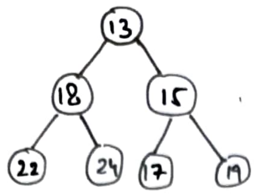
i.e if elements are doubled, then H increases by one.

$\therefore$ Complexity $= O(H) = O(\log_2 N)$.

For best case, worst case and average case.

Min Heap: In a min-heap, the value of a particular node is smaller than or equal to the value of its children.

E.g.



Algorithm:

For **insert**: One by one.

```
heap Insert (A, data)
{
  heap_size = heap_size + 1
  Decrease Key (A, heap_size, data)
}

DecreaseKey (A, i, data)
{
  A[i] = data
  while (i > 1 && A[parent(i)] > A[i])
  {
    exchange A[i] and A[parent(i)]
    i = parent(i)
  }
```

For **delete**: Deleting the minimum element (The root node)

```
delete Min (int A[ ])
{
  min = A[i]
  A[1] = A[heap_size]
  heap_size -= 1;
  heapify (A, 1)
  return min
}
```

For **heapify()**

```
heapify ( A , i)
{
   smallest = i.
   l = 2 x i
   r = 2 x i + 1
   if ( l ≤ heap_size and A[l] < A[smallest])
        smallest = l
    if (r ≤ heap_size and A[r] < A[smallest])
        smallest = r
    if ( smallest is not i)
        Exchange A[i] with A[smallest]
        heapify (A , smallest)
}
```

Time complexity:

As discussed earlier:

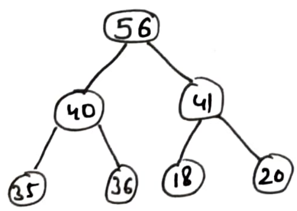
For Insert = $O(\log_2 N)$

For Delete = $O(\log_2 N)$

For Heapify = $O(\log N)$

Max Heap: In a max heap, the value of a particular node is greater than the value of their children.

E.g



Algorithm:

For insert :

```
Heap_Insert ( A , data)
{
   heap_size + = 1;
   Decreasekey ( A , heap_size, data)
}
```

Decrease key (A, ~~heap-size~~ i , data)

{

A[i] = data

while (i > 1 and A[parent(i)] < A[i]) {

exchange A[i] and A[parent (i)]

i = parent(i)

}

}

For <u>delete</u>: Removal of Maximum element of the heap (i.e root node)

deleteMax (A[])

{

max = A[1]

A[1] = A[heap-size]

heap-size -= 1

heapify (A, 1)

return max

}

For <u>heapify</u>:

heapify (A, i)

{

~~smallest~~ largest = i

l = 2 x i

r = 2 x i + 1

if (l ≤ heap-size and A[i] > A[~~smallest~~ largest])

~~smallest = l~~ largest = l

if (r ≤ heap-size and A[i] > A[~~smallest~~ largest])

~~smallest = r~~ largest = r

if (smallest is not i) {

Exchange A[i] with A[largest]

heapify (A, largest)

}

}

e Complexity:

As discussed earlier:

For insert : $O(\log_2 N)$ For one element

For delete : $O(\log_2 N)$

For heapify : $O(\log N)$

Overall insert time complexity for inserting N elements = $O(N \log_2 N)$

Heap Sort: Heap Sort is an inplace algorithm, as comparison based algorithm based on the Binary Heap. It is an unstable algorithm but can be modified to stable. In heap sort, we first create a max heap, remove elements till heap ~~element~~ is empty (i.e every time we extract the maximum element).

Algorithm:

```
heapSort ( A)
{
  build _MaxHeap (A)
  for (i = n ; i >= 2 ; i++)
  {
   ① exchange A[1] and A[i]
   ② heap-size -= 1
   ③ heapify (A, 1)
  }
}
```

Time Complexity:

For heapify function, the worst case time complexity is $O(\log n)$

Now the for loop runs for $n-1$ times

consider ①, ②, ③ Time complexity

$= O(\text{constant})$

∴ Total time complexity :

$(n - \cancel{1})(\log_2 n + c)$

$= O(n \log n)$

Priority Queue: Priority Queue is a special queue data structure where every element ~~has~~ ~~at~~ is associated with a priority.

2 types: Ascending Priority Queue: The element with smallest value has highest priority

Descending Priority Queue: The element with largest value has highest priority.

3 operations: Enqueue(): Insert an element in queue

Dequeue(): Remove element of the highest priority

Peek(): Return the element of highest priority.

Can be implemented using array, heaps, linked list and BST.

Algorithm:

1. For enqueue:

```
Enqueue( A , data)
{
   if (heap_size == 0) {
      heap_size ++
      A[heap_size] = data
   }
   else
   {
      array[heap_size++] = data
      for (i = size/2 down to 1)
         heapify(A, i)
   }
}
```

2. For dequeue: Basically refers to removing the maximum or minimum element (Highest priority)

```
for (i = 1 to heap-size){
    if (data == A[i])
        break;
}

Exchange A[i] with A[heap-size]
heap-size -= 1
for (i = heap-size/2 down to 1)
    heapify (A, i)
}
```

3. For <u>peek()</u>: Returning the element of highest priority (Maximum element in Max Heap, minimum element in min heap)

```
peek (A[ ]) {
    return A[1]
}
```

<u>Time Complexity:</u>

For insertion: $O(\log n)$

For deletion: $O(\log n)$

For peek: $O(1)$