# Group 51- Write an efficient algorithm to merge two given large Heaps of different sizes to create one large Heap. Do this merging using Heap Insertion approach where you insert elements of one Heap, one-by-one, into the other Heap. Trace the movement of elements in every step of the execution of your algorithm.

IHM2016501[1] IHM2016005[2] BIM2016004[3] IIM2016006[4] RIT2015050[5]

***Abstract*— We worked on an efficient algorithm to merge two given large heaps and we are tracing the movement of the elements in the above mentioned process by making use of linked lists.The result obtained is a heap which consists the elements of both the given heaps.**

## I. INTRODUCTION AND LITERATURE SURVEY

A Heap is a partially sorted binary tree. Although a heap is not completely in order, it conforms to a sorting principle: every node has a value less than or greater than either of its children. Additionally, a heap is a "complete tree" – a complete tree is one in which there are no gaps between leaves. $Min - Heap$ Where the value of the root node is less than or equal to either of its children. $Max - Heap$ Where the value of the root node is greater than or equal to either of its children.If we consider a Heap as a binary tree sorted in linear order by depth with the root node first; then the children of a node at index N are 2N+1 and 2N+2.

So,here in the question we are given two large heapsof given sizes and we have to write an efficient algorithm in-order to merge them to create a single large Heap. We use insertion approach to merge them.Here we insert elements of one heap into the other heap.We also need to track the movement of every element of the heap during thr execution of the heap.

## II. ALGORITHM DESIGN

This part of the report can be further divided into following sections.

### A. Input Format

In first few lines of the code we use $rand()$ function to generate a random number and assign it to the variable $n$ and $m$ which defines the size of the heaps.We then compare $n$ and $m$ and we see that $heapa$ has more number of elements that $heapb$.And then we use $rand()$ function to generate $n$,$m$ random numbers and store them in the respective heaps.So.we now have two heaps $heaps$,$heapb$ of sizes n,m filled with n,m randomly generated numbers.

- A structure $node$ has been used which has an pointerof type structure $node$ which can be used to pint to another structure $node$ and value of type integer which stores the value present index of the heap.
- We have a array $position[]$ of the above mentioned structure type.Which is useful in tracing the movement of the elements.
- We also have another array $indexes[]$ which stores the instanteneuos indexes of the final heap elements .
- The above mentioned Structure has been declared globally.

- Also the arrays heapa[],heapb[],indexes[], variables n,m are declared globally.

*B. Algorithm*

**Algorithm 1** prints the track of elements

```
1: Input head of linked list
2: procedure PRINT(head)
3:     temp ← head
4:     while temp ≠ NULL do
5:         print(temp → data)
6:         temp = temp → next
```

**Algorithm 2** adding a node to linked list

```
1: Input head of linked list, element inserted
2: Initialization : temp, temp1(structpointer)
3: procedure ADD(head, x)
4:     (temp → data) ← x
5:     if head = NULL then
6:         (temp → next) ← head
7:         head ← temp
8:         return head
9:     else
10:        temp1 ← head
11:        while temp1 → next ≠ NULL do
12:            temp1 ← (temp1 → next)
13:        (temp1 → next) ← temp
14:        (temp → next) ← NULL
15:        return head
```

**Algorithm 3** returns parent of given node

```
1: Input Index of node i
2: procedure P(i)
3:     return (i − 1)/2
```

**Algorithm 4** Swaps two position of heap

```
1: Input Two positions i and p
2: Initialization : temp
3: procedure SWAP(i, p)
4:     temp ← heapa[i]
5:     heapa[i] ← heapa[p]
6:     heapa[p] ← temp
7:     temp ← indexes[i]
8:     indexes[i] ← indexes[p]
9:     indexes[p] ← temp
```

**Algorithm 5** Function to build max heap

```
1: Input index i
2: procedure HEAPIFY(i)
3:     while i ≠ 0 AND heapa[p(i)] < heapa[i] do
4:         position[indexes[p(i)]] ← add(position[indexes[p(i)]], i)
5:         position[indexes[i]] ← add(position[indexes[i]], p(i))
6:         swap(i, p(i))
7:         i ← p(i)
```

**Algorithm 6** Merging two heaps

```
1: Input Two integers a and b
2: Initialization i,j,k
3: procedure MERGE(a, b)
4:     i ← a
5:     for k ← 0 to a do
6:         position[indexes[k]] ← add(position[k], k)
7:     for j ← m-1 to 0 do
8:         position[indexes[i]] ← add(position[indexes[i]], i)
9:         heapa[i] ← heapb[j]
10:        heapify(i)
11:        i ← i + 1
```

**Algorithm 7** Main Function

```
1: Initialize n, m
2: procedure MAIN                          ▷ main function
3:     if n ≥ m then
4:         for i ← 0 to n do
5:             scan(heapa[i])
6:             indexes[i] ← i
7:         for i ← 0 to m do
8:             scan(heapb[i])
9:             indexes[i + n] ← i + n
10:        merge(n, m)
11:    else
12:        for i ← 0 to n do
13:            scan(heapa[i])
14:        for i ← 0 to m do
15:            scan(heapb[i])
16:        merge(m, n)
```

### C. Output Format

As mentioned in the question we are merging the given heaps and then we are tracing the path of each element.We will first output the tracing of the elements and then we print the new heap which has been formed my merging the given arrays.

### D. Comparison with the Algorithmn of Previous Group

Now we will compare the above mentioned algorithmn with the algorithmn of previous group and highlight the differences.

*a) Input :* First we take the input i.e, the length of the two large heaps respectively and store them in n,m and then the two large heaps heapa and heapb accordingly.And then we will call the merge function taking the lengths of the heaps as the attributes.We can see that both the algorithmns have the same process for taking in the input.

*b) Merge Function:* The actually difference in the algorithmns occurs in this $Merge$ Function.The Merge function is used to merge both the heaps.In our approach we insert the last element of heapb into heapa i.e,inserting in the end of heapa and then heapify heapa to form a new heap.Hence the heapa contains n+1 elements and heapb contains m-1 elements and heapb remains as a heap as we are deleting its last element.But,whereas in the approach followed by previous group the first element of heapb is added into heapa i.e insertion in the end of heapa but then in this case we have to heapify heapb also as when we remove the first element of heapb it is no longer a heap(But the previous group didnt heapify heapb ).This increases complexity as we have two(heapa,heapb) heaps to heapify instead of one(heapa).

**Algorithm 8** Merging two heaps(previous group)

1: **Input** *Two integers a and b*
2: **Initialization** *i,j,k*
3: **procedure** MERGE($a, b$)
4: **for** i $\leftarrow$ 0 to a **do**
5: $track[index\_arr[i]] \leftarrow InsertAtEnd(index\_arr[i], i)$
6: $index \leftarrow i$
7: **for** j $\leftarrow$ 0 to b-1 **do**
8: $track[index\_arr[index]] \leftarrow InsertAtEnd(track[index\_arr[index]], index)$
9: $heapa[index] \leftarrow heapb[j]$
10: $heapify(index)$
11: $index \leftarrow index + 1$

*c) Other Functions:* The other Functions in the algorithmn include $heapify$-It is used to heapify the heaps i.e, build the max heap,$print$-prints the track of the specified element,$swap$-it swaps the position of the two heaps,$p$(parent)-returns the parent of given node,$add$-it inserts a node to its specified parent.All the above functions has the same functioning in both the algorithmns.

*d) Tracing the elements:* In both the algorithmns an array is specified to store the initial indexes of the heaps and the tracing of the elements is done with the help of linked lists and a position[] array where the nodes are added to the corresponding indexes when ever swapping takes place.

## III. ANALYSIS

### A. Analysis

*1) Best Case Analysis:* The algorithm will have its best case when the elements of larger heap is equal to that of the elements of the smaller heap and all elements in the heapa are equal, in this case the moment we add an elements into heapa we don't need to heapify it as it is already a heap so in the algorithm will not enter the while loop (running $log_2(n+m)n$ times) in the heapify function hence it will execute in linear time. Best Case time complexity proportional to, $O(n+m)$.

$$time_{best} \propto 3n + 3m + 3k(m+n) + 30$$

So,time complexity in best case is $O(n+m)$

*2) Worst Case Analysis :* Worst case occurs when all the elements of heapa and heapb are different , sizes of both the heaps are equal and all the elements of heapb are greater than heapa.Since Inserting an element in heap and making it a maxheap takes $(log_2(n))$ instructions(number of elements in heap is n), so we this can maximize when both n and m are equal, then only it will for maximum time. Worst Case proportional to $O(mlog_2(m+n))$

$$time_{worst} \propto 3n+3m+3k(m+n)+m(4+log(n+m)*(25+10k))+30$$

So,Worst case time complexity is $O(mlog_2(n+m))$

*3) Average Case Analysis :* We randomly generate both the heaps and merge them.

$$time_{average} \propto mlog_2(n+m)$$

So,Average case time complexity is $O(mlog_2(n+m))$

### B. Comparing the Analysis of the Algorithmns

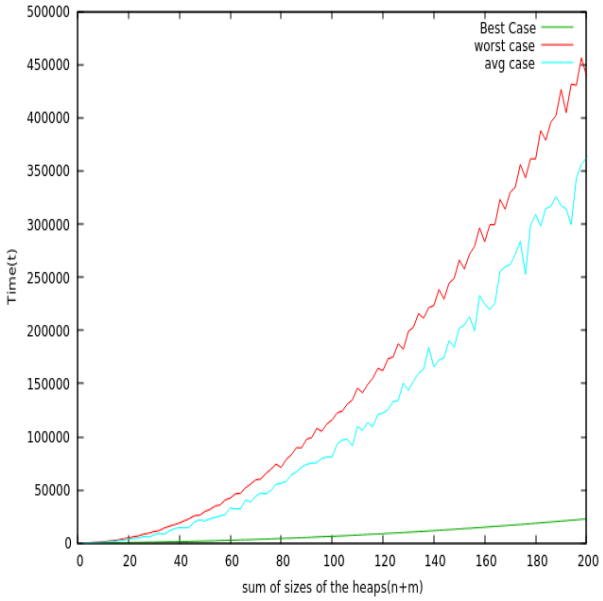
*a) Best Case:* The Best case complexity of previous group is proportional to $O(n+m+mlog_2m)$ as they have to heapify the heapb also where as our Best Case time complexity proportional to, $O(n+m)$.

*b) Worst and Average Case:* The Worst case and Average complexity of previous group is proportional to $O(n+m+mlog_2m+mlog_2(m+n))$ i.e, it is proportional to $O(mlog_2(m(m+n)))$ as they have to heapify the heapb also where as our Best Case and Average case time complexity proportional to, $O(mlog_2(m+n)+m+n)$ i.e, it is proportional to $O(mlog_2(m+n))$

## IV. EXPERIMENTALSTUDY

The best way to study an algorithm is by graphs and profiling.

*1) Graphs-Time-Complexity:* From the graph it is clear that in the best case obtained is almost linear i.e, it is proportional to $O(n+m)$ but the worst case is proportional to $O(mlog_2(n+m))$(but it is slightly deviated), we are getting slight deviations because of the additional complexities because of tracing the elements.




*2) Profiling:*

$$NumberOfComputations$$

| $n+m$ | $t_{avg}$ | $t_{best}$ | $t_{worst}$ |
|---|---|---|---|
| 2 | 76 | 31 | 76 |
| 4 | 221 | 64 | 221 |
| 10 | 810 | 187 | 1203 |
| 20 | 3831 | 472 | 4790 |
| 50 | 20769 | 1927 | 29679 |
| 70 | 44053 | 3397 | 59387 |
| 80 | 56295 | 4282 | 71123 |
| 200 | 363189 | 22702 | 439519 |

- We can also see that time increases as the value of n+m increases and $time_{worst} \propto mlog_2(n+m)$,$time_{avg} \propto mlog_2(n+m)$,$time_{best} \propto n+m$.

## V. DISCUSSIONS

### *A. Understanding the Use Of File Handling and generation of random numbers*

In this context the file handling is used to generate the random test cases through which analysis and experimentation of algorithm becomes easier for handling different test cases. We have used $srand()$ and $rand()$ to generate the random values to fill the matrix with the help of $< time.h >$ header file.

## VI. CONCLUSION

- By making the proper analysis of the algorithm and optimizing the code we can conclude that the best case algorithm will occur when the elements of larger heap is equal to that of the elements of the smaller heap and all elements in the heapa(larger heap) are equal and the time complexity for it will be linear which is $O(n + m)$ where n and m are the size of the heaps.And in worst case occurs when all the elements of heapa and heapb are different , sizes of both the heaps are equal and all the elements of heapb are greater than heapa and the time complexity for the worst case of the algorithm is $O(mlog_2(n + m))$. The average case lies in between these two and it is more inclined towards the worst case. Thus, by implementing the code designed for the given algorithm we have developed a optimized mechanism to merge two heaps into one. In this way we can extending the above problem for n heaps.

## REFERENCES


[1] H. Kopka and P. W. Daly, *A Guide to LaTeX*, 3rd ed. Harlow, England: Addison-Wesley, 1999.

[2] Introduction to Algorithms English By Thomas H. Cormen , By Charles E. Leiserson , Ronald L. Rivest ,Clifford Stein(3rd edition)