COMPARATIVE STUDY OF PRIORITY QUEUE IMPLEMENTATIONS

Dhananjay Gupta[1], Avani Kothari[2], Neetesh Gupta[3], Akshay Khandelwal[4]

[1,2,3,4] *Senior Undergraduate, Department of Information Technology, Shri Vaishnav Institute of Technology and Science, M.P., India*

ABSTRACT

In this paper, we analyze and compare various available ways to implement priority queues. A Priority queue is an abstract data type which is like a regular queue, but where additionally each element has a priority key associated to it [1]. The key serves as a way of sorting out the elements of the priority queue. One can imagine a case of vehicles running on the road. Certain vehicles like ambulances need more priority than any other. VIP vehicles come second and so on. Here, this signifies the relevance of priority. Similarly in operating systems jobs may be scheduled as per their importance this can be achieved through priority queues. The primary focus of this paper is to analyze various available ways of implementing priority queues, which could later form as basis when devising a new approach or method to improve the efficiency of priority queues. We compare the priority queues on the basis of their running time efficiencies and compare their running time bounds in the worst cases. This paper also forms the basis for our next work, where we show, how the concept of memory pooling can be effectively used in improving the efficiency of the priority queues.

Keyword: - *Priority queues, heap tree, Fibonacci heap, comparative study, complexity analysis.*
.

1. INTRODUCTION

A priority queue is a data structure for maintaining a set S of elements, each with an associated value called a key. A priority queue must at least support the following operations [1]:

INSERT (S, x) inserts the element x into set S, which is equivalent to the operation S = S U {x}.

MAXIMUM(S) returns the element of S with the largest key.

EXTRACT-MAX(S) removes and returns the element of S with the largest key.

More advance implementation of priority queues may support more complicated operations, pull lowest priority element, inspecting the first few highest- or lowest-priority elements, clearing the queue, clearing subsets of the queue, performing a batch insert, merging two or more queues into one, incrementing priority of any element, etc.

An efficient algorithm is one which executes in less time and requires less memory space. In today's times space complexity (defined in terms of input given at a time based on memory required) is not of much concern because of good storage system with very large memory can always be attached and a program with size greater than the memory can be executed (example- virtual memory). This brings us to a conclusion that time is much more important concern than space. A user is always expecting an output within zero seconds of providing the input. However, this is not practically possible. The aim of this work is to summarize and compare the running time, worst case efficiencies of already available advanced implementations of priority queues.

1.1 Big Oh Notation, O

The notation O(n) is the formal way to express the upper bound of an algorithm's running time. It measures the worst case time complexity or the longest amount of time an algorithm can possibly take to complete.

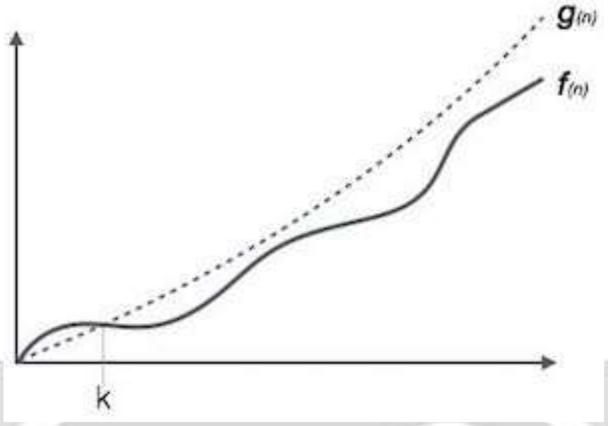

Figure1: Big O notation's graphical representation.

For example, for a function f(n)
$O(f(n)) = \{ g(n) : \text{there exists } c > 0 \text{ and no } n_o \text{ such that } f(n) \leq c.g(n) \text{ for all } n > n_0 \}$

Following is the list of some common asymptotic notations:

Constant	–	O(1)
Logarithmic	–	O(log n)
Linear	–	O(n)
n log n	–	O(n log n)
Quadratic	–	$O(n^2)$
Cubic	–	$O(n^3)$
Polynomial	–	$n^{O(1)}$
Exponential	–	$2^{O(n)}$

Table -1: Common Asymptotic Notations

1.2 Data Flow of Priority Queue Operations

The following figure 2. shows the relative flow of data in priority queue data structure. Data as input is inserted. If the queue is full, an overflow condition is observed. Remove operation remove elements and if the queue becomes empty and underflow condition is observed.

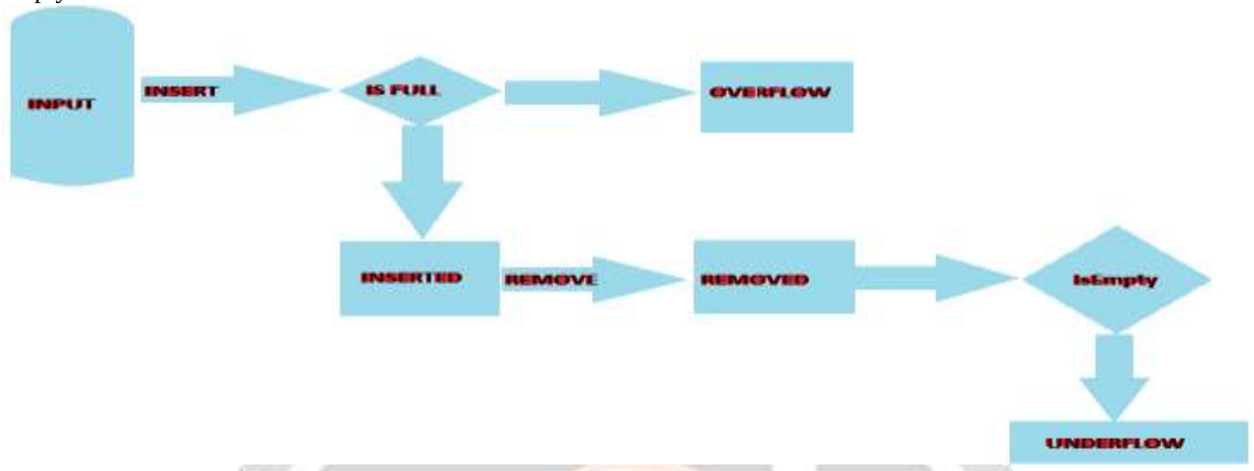

Figure 2: Diagrammatic view of Data flow in priority queues.

2. Various Available Implementations of Priority Queues

2.1 Simple Implementation Using Number System

In The simple implementation of priority queues we have performed all main operations insert, delete and display using a very simple logic. Here, we have used two variables rear and front to track the elements of priority queue on every operation. We use simple number system to check the working and output of the code. Every element is compared to previous one in terms of its priority (which is also provided by the user) therefore there is no consideration of optimization in this code. Comparing and swapping (in case of higher priority) makes this code very general and inefficient for real time data or large data.

2.1.1 Complexity Analysis

Complexity of a code is generally calculated by analyzing the no. of assignments or comparison in a statement multiplied to no. of times that statement is supposed to run in a single call. Here our main focus is on worst case complexity therefore we will look for maximum no. of times a statement can be executed.

Insert- while inserting an element it is compared to the element at front end and if the priority of element at front is higher than current element that it is swapped with it. For this a- for loop can run maximum rear*rear times and rear can have maximum value n (no. of elements). So the worst case complexity is O(n^2)

Delete-deletion simply takes place from front end so it is O(1).

Overall complexity = O(n^2)

2.2 Extended Simple Implementation Using Arrays

There are several possible implementations of priority queues one simplest is through arrays. In this algorithm we have tried to optimize the operations we used in previous algorithm .Arrays are linear data structures use to store data sequentially of a particular size. The highest-priority element is requested, search through all elements for the one with the highest priority. Another approach is to add code for *insert* to move larger entries one position to the right, thus keeping the entries in the array in order (as in insertion sort). Thus the largest item is always at the end, and the code for *remove the maximum* in the priority queue is the same as for *pop* in the stack.

2.2.1 Complexity Analysis

In this code (insertion) insertion of new data is followed by shifting the existing data in right hence giving the complexity of order n this not very efficient as it shifting and comparing takes considerable time and the overall complexity is affected.

On the other hand removal is of order 1. Because the element with highest priority is always found at the front end of the queue and can be easily removed. The overall complexity is calculated by adding up the complexities obtained by inserting and deleting the data.

Insert-while inserting an element, (from right) it is compared from the element to its right and this is done till the 0th index of array is reached hence giving the complexity O(n).

Delete-deletion simply takes place from one end so it is O(1).

Overall complexity = O(n)

2.3 Linked List Implementation

In the link lists implementation we use link list data structures to implement priority queues. The core implementation of link lists can be seen in c/c++. Link lists are more dynamic data structures and are useful when fast retrieval of dynamic data is needed [5]. In this research we have also found that one way of inserting and extracting the elements of priority queues is by link lists. However on comparing it with other implementations it can be inferred that it is efficient for few operations but not fully optimized among all the implementations.

2.3.1 Complexity Analysis

Insert-while inserting an element, the entire link list is traversed since in link lists data is inserted sequentially giving the complexity of O(n).

Delete-data is popped from one end directly (by maintaining the pointers) hence deletion if of O(1)

Overall complexity = O(n)

2.4 Priority Queues Using Heaps

A heap is a specific tree based nonlinear data structure in which all the nodes of tree are in a specific order. Let’s say if X is a parent node of Y, then the value of X follows some specific order with respect to value of Y and the same order will be followed across the tree. In the more commonly used heap type, there are at most 2 children of a node and it's known as a Binary heap [6].
This is the usual implementation of priority queues.

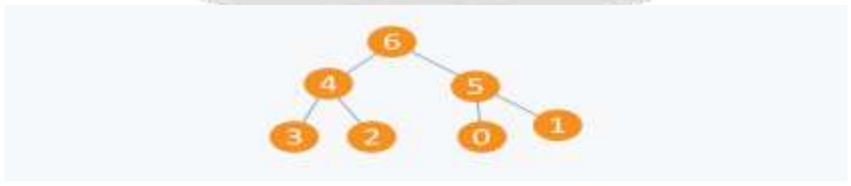

Figure 3: Diagrammatic view of a heap tree (Max.).

Suppose there are N Jobs in a queue to be done, and each job has its own priority. The job with maximum priority will get completed first than others. At each instant we are completing a job with maximum priority and at the same time we are also interested in inserting a new job in the queue with its own priority. This task can be very easily executed using a heap by considering N jobs as N nodes of the tree.

As you can see in the figure-3, we can use an array to store the nodes of the tree. Let's say we have 7 elements with values {6, 4, 5, 3, 2, 0, 1}.

Note: An array can be used to simulate a tree in the following way. If we are storing one element at index 'i' in array 'Ar', then its parent will be stored at index i/2 (unless it's a root, as root has no parent) and can be access by Ar[i/2], and its left child can be accessed by Ar[2 * i] and its right child can be accessed by Ar[2 * i +1]. Index of root will be 1 in an array.

2.4.1 Max Heap: In this type of heap, the value of parent node will always be greater than or equal to the value of child node across the tree and the node with highest value will be the root node of the tree.

2.4.2 Implementation:

Let's assume that we have a heap having some elements which are stored in array Arr. The way to convert this array into a heap structure is the following. We pick a node in the array, check if the left sub-tree and the right sub-tree are max heaps in themselves and the node itself is a max heap (it's value should be greater than all the child nodes)

```
void max_heapify (int Arr[ ], int i, int N){
   int left = 2*i                  //left child
   int right = 2*i +1          //right child
   if(left<= N and Arr[left] > Arr[i] )
   largest = left;
   else
       largest = i;
   if(right <= N and Arr[right] > Arr[largest] )
      largest = right;
   if(largest != i ){
      swap (Ar[i] , Arr[largest]);
      max_heapify (Arr, largest,N);
   }  }
```

we have N elements stored in the array Arr indexed from 1 to N. They are currently not following the property of max heap. So we can use max-heapify function to make a max heap out of the array.

2.4.2.1 Efficient Approach:

We can use heaps to implement the priority queue. It will take O(log N) time to insert and delete each element in the priority queue. Based on heap structure, priority queue also has two types max- priority queue and min - priority queue. Let's focus on Max Priority Queue. Max Priority Queue is based on the structure of max heap and can perform following operations:

maximum(Arr) : It returns maximum element from the Arr.
extract_maximum (Arr) - It removes and return the maximum element from the Arr.
increase_val (Arr, i , val) - It increases the key of element stored at index i in Arr to new value **val**.
insert_val (Arr, val) - It inserts the element with value **val** in Arr.

Implementation:

length = number of elements in Arr.

2.4.2.1.1 Maximum :

```
 int maximum(int Arr[ ]){
        return Arr[ 1 ];  //as the maximum element is the root element in the max heap.
    }
```
Complexity: O(1)

2.4.2.1.2 Extract Maximum:

In this operation, the maximum element will be returned and the last element of heap will be placed at index 1 and max_heapify will be performed on node 1 as placing last element on index 1 will violate the property of max-heap.

```
int extract_maximu-m (int Arr[ ]){
    if(length == 0){
cout<< "Can't remove element as queue is empty";
        return -1;
    }
    int max = Arr[1];
    Arr[1] = Arr[length];
    length = length -1;
    max_heapify(Arr, 1);
    return max;
}
```
Complexity: O(log N).

2.4.2.1.3 Increase Value:

In case increasing value of any node, may violate the property of max-heap, so we will swap the parent's value with the node's value until we get a larger value on parent node.

```
void increase_value (int Arr[ ], int i, int val){
    if(val < Arr[ i ]){
        cout<<"New value is less than current value, can't be inserted" <<endl;
        return;
    }
    Arr[ i ] = val;
    while( i > 1 and Arr[ i/2 ] < Arr[ i ]){
        swap|(Arr[ i/2 ], Arr[ i ]);
        i = i/2;
    }}
```
Complexity: O(log N).

2.4.2.1.4Insert Value:

```
void insert_value (int Arr[ ], int val){
    length = length + 1;
    Arr[ length ] = -1;  //assuming all the numbers greater than 0 are to be inserted in queue.
    increase_val (Arr, length, val);
}
```
Complexity:

Insert-As discussed above, like heaps we can use priority queues in scheduling of jobs. When there are N jobs in queue, each having its own priority. If the job with maximum priority will be completed first and will be removed from the queue, we can use priority queue's operation extract_maximum here. If at every instant we have to add a new job in the queue, we can use **insert_value** operation as it will insert the element in O(log (N)) and will also maintain the property of max heap, this is because insertion requires a comparison with element of above levels which can go till height of heap tree i.e. log(N).

Delete- Deletion is performed from the root therefore deletion from root followed by the re heapification each time takes time equal to height of the heap tree giving complexity of O(log(N)).

Hence the overall complexity is O(log(N)).

2.5 Fibonacci Heap:

Fibonacci heap is a specialized data structure used mainly for implementing priority queues in computer science. It consists of a patching of heap ordered trees. It has better running time than many other priority queue data structures including the ones which are discussed above. Michael L. Fredman and Robert E. Tarjan developed this. They named it so after the Fibonacci numbers, which are used in their running time analysis.

2.5.1 Implementation [2]:

```
Make-Fibonacci-Heap()
n[H] := 0
min[H] := NIL
return H

Fibonacci-Heap-Minimum(H)
return min[H]

Fibonacci-Heap-Link(H,y,x)
remove y from the root list of H
make y a child of x
degree[x] := degree[x] + 1
mark[y] := FALSE

CONSOLIDATE(H)
for i:=0 to D(n[H])
   Do A[i] := NIL
for each node w in the root list of H
  do x:= w
    d:= degree[x]
    while A[d] <> NIL
             do y:=A[d]
               if key[x]>key[y]
                      then exchange x<->y
               Fibonacci-Heap-Link(H, y, x)
               A[d]:=NIL
        d:=d+1
    A[d]:=x
min[H]:=NIL
for i:=0 to D(n[H])
  do if A[i]<> NIL
      then add A[i] to the root list of H
         if min[H] = NIL or key[A[i]]<key[min[H]]
           then min[H]:= A[i]

Fibonacci-Heap-Union(H1,H2)
H := Make-Fibonacci-Heap()
min[H] := min[H1]
Concatenate the root list of H2 with the root list of H
if (min[H1] = NIL) or (min[H2] <> NIL and min[H2] < min[H1])
  then min[H] := min[H2]
```

```
n[H] := n[H1] + n[H2]
free the objects H1 and H2
return H
```

```
Fibonacci-Heap-Insert(H,x)
degree[x] := 0
p[x] := NIL
child[x] := NIL
left[x] := x
right[x] := x
mark[x] := FALSE
concatenate the root list containing x with root list H
if min[H] = NIL or key[x]<key[min[H]]
            then min[H] := x
n[H]:= n[H]+1
```

```
Fibonacci-Heap-Extract-Min(H)
z:= min[H]
if x <> NIL
            then for each child x of z
                        do add x to the root list of H
                          p[x]:= NIL
               remove z from the root root list of H
               if z = right[z]
           then min[H]:=NIL
           else min[H]:=right[z]
              CONSOLIDATE(H)
        n[H] := n[H]-1
return z
```

```
Fibonacci-Heap-Decrease-Key(H,x,k)
if k > key[x]
  then error "new key is greater than current key"
key[x] := k
y := p[x]
if y <> NIL and key[x]<key[y]
  then CUT(H, x, y)
            CASCADING-CUT(H,y)
if key[x]<key[min[H]]
  then min[H] := x
```

```
CUT(H,x,y)
Remove x from the child list of y, decrementing degree[y]
Add x to the root list of H
p[x]:= NIL
mark[x]:= FALSE
```

```
CASCADING-CUT(H,y)
z:= p[y]
if z <> NIL
  then if mark[y] = FALSE
      then mark[y]:= TRUE
      else CUT(H, y, z)
               CASCADING-CUT(H, z)
```

Fibonacci-Heap-Delete*(H,x)*
Fibonacci-Heap-Decrease-Key(*H*,*x*,-infinity)
Fibonacci-Heap-Extract-Min(*H*)

2.5.2 Complexity Analysis:

Operation find minimum is now trivial because we keep the pointer to the node containing it. It does not change the potential of the heap, therefore both actual and amortized cost are constant. merge is implemented simply by concatenating the lists of tree roots of the two heaps. This can be done in constant time and the potential does not change, leading again to constant amortized time. Operation insert works by creating a new heap with one element and doing merge. This takes constant time, and the potential increases by one, because the number of trees increases. The amortized cost is thus still constant.

3. CONCLUSION

By analyzing all notations asymptotically for the worst case behavior it can be seen that heap implementation of priority queue has the best running time complexity. It can be concluded that for a large data set priority queues using heap can give results in no more than O(log n) time, making it much more efficient than any other way of implementation of priority queues. It has been established now that priority queues work best with heaps when being used upon large volume of data. We shall now work on memory pooling with heaps to better improve the overall efficiency of priority queues.

Type of implementation	Complexity		
	insert	delete	display
Simple implementation	O(n^2)	O(1)	O(1)
Extended	O(n)	O(1)	O(1)
Link Lists	O(n)	O(1)	O(n)
Heaps	O(log n)	O(log n)	O(log n)
Fibonacci Heap	O(1)	O(log n)	O(1)

Table -2: Comparative Analysis

4. REFERENCES

[1]. INTRODUCTION TO ALGORITHMS (3rd edition) by Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein.
[2]. http://www.cse.yorku.ca/~aaw/Jason/FibonacciHeapAlgorithm.html
[3]. J. W. J. Williams, Algorithm 232: Heap sort, Communications of the ACM 7, 6(1964), 347-348
[4]. Data Structures and algorithms analysis- Mark Allen Weiss, Florida International University.
[5]. http://stackoverflow.com/questions/24475724/c-linked-list-with-priority-queue
[6]. https://www.cs.princeton.edu/~rs/AlgsDS07/06PriorityQueues.pdf