Data Structures
Spring 2009

Kishore Kothapalli

Chapter 5

Priority Queue

5.1 Introduction

While servicing jobs is best done by placing the waiting jobs in a queue and processing them in a first-in-first-out manner is easier to comprehend, there are several scenarios where an out-of-order processing is also necessary. Consider for example the scheduler in an operating system. If it is a multi-tasking OS, such as Unix, meaning that several jobs can be in the state of execution at the same time, then the scheduler has to decide which job to run at any given time. Here, instead of a FIFO rule, it makes perfect sense to run the shorter jobs first before longer ones. This is a very popular scheduling rule, called SJF, and is known to minimize the total waiting time experienced by the jobs.

Thus, what is required is a way to identify the next job to be processed from the queue not based on FIFO but some other rule. This rule is referred to as priority in CS terms. Hence, we are seeking a data structure to support operations such as inserting and item and deleting the item with the highest priority at the very least. In this chapter, we'll study very efficient solutions to this problem.

5.1.1 Operations

The data structure we are seeking should support the following operations.

- `Insert`(x) : Insert the element with priority x.
- `DeleteMin`(x) : Remove the element with the least value.

Note that though we are trying to remove the job with the highest priority from the queue currently, the name of the operation is misleading. But imagine that jobs with small value of x have more priority than jobs with bigger values of x.

There exist other useful operations that are useful in implementing various graph and greedy algorithms. We will define them later after understanding these basic operations.

5.2 A Binary Heap

Let us consider some alternatives. If we use a simple linked list, then `insert` can be supported in $O(1)$ time but `DeleteMin` requires $O(n)$ time as one has to scan the entire list. If the linked list is however maintained as a sorted list, then `DeleteMin` can be supported in $O(1)$ time, but the cost of `insert` increases to $O(n)$ in the worst case.

A binary search tree is another alternative. Here one can use AVL trees to ensure that the depth of the tree is $O(\log n)$. Both the operations run in $O(\log n)$ time, but this is an overkill as BST supports lots of other operations that are not relevant to us.

Fortunately, there exists a simple and elegant strategy that does not even involve pointers and links.

Consider a binary tree of n nodes where the leaf nodes are spread across at most two levels. This also suggests that the depth of the tree is $\leq \lceil \log n \rceil$. Another way to specify this tree is to require the condition that children are added in a left-to-right manner. Thus, the tree is completely occupied by nodes with the exception of the last level where nodes are filled from left to right. Such a tree is also called as a *complete binary tree*. This, a complete binary tree of height h has between 2^h and $2^{h+1} - 1$ nodes. The following figure shows a complete binary tree of height 3.

FIGURE TO COME HERE

We will use a completely binary tree as our data structure. Of course, we promised that our solution does not require any pointers. We will now show that a complete binary tree of n nodes can be implemented as an array of size n thereby doing away with any pointer manipulation.

Consider an array of size n with indices ranging from 1 to n. The element at index 1 will be the root of the complete binary tree and the left child of a node at index i will be stored at index $2i$ and the right child of a node at i will be stored at index $2i + 1$, if $2i, 2i + 1 \leq n$. Otherwise, the node does not have the left or right child correspondingly. The above tree can thus be represented by the following array.

FIGURE FROM PAGE 3 BOTTOM HERE

Thus, the left child of node 15 and index 5 is 3 at index 10 and the right child is node 5 at index 11.

Thus, our data structure, a complete binary tree, can be implemented as an array of size n. The only disadvantage of this choice is that at times it is difficult to know the size of the array, n, in advance. Evevn to remedy this, dynamic variants are known. So, in the following we will not worry about the size n of the array.

we will call our data structure as the *binary heap* or *heap* for short. For our data structure, we are only interested in finding the element of least value. This, it is reasonable to expect that the root contains the minimum element. Since each subtree of the root could also be viewed as our data structure, we can also expect that the root of the left subtree contain the minimum element among the elements in the left subtree, and the root of the right subtree contain the minimum element among the elements of the right subtree. We call this the *heap property* and is written below for clarity.

Definition 5.2.1 (Heap Property) *In a heap, the value at any node will be smaller than the value of any node in that subtree.*

Equivalently, the value at any node x is smaller than the value at its parent. The following figure shows an example of a heap.

FIGURE FROM PAGE 4

Thus, while implementing our operations, we have to keep the structure as a complete binary tree and maintain the heap property. We will now use how our operations can be implemented.

5.2.1 Operations

Let us see how to implement the `insert` operation.

Insert:

to insert a node with value x to an existing heap of k elements, we proceed as follows. To satisfy the conditions of a complete binary tree, we will add x as the element at index $k + 1$ in the array. As an

example, to insert element 20 in the heap shown in the following figure, we add 20 as the right child of node 50.

FIGURE HERE

Now the insertion may violate the heap property as the above example shows. To restore the heap property, we have to push the inserted element towards the root. This process is called as `ShuffleUp`. In the Shuffleup operation, the value of a position is compared with that the value of its parent. If the value is smaller than its parent, then they are swapped. This is done until either value is bigger than the value at its parent or we have reached the root.

Let us illustrate this operation by continuing our example. Since $20 < 50$, nodes 20 and 50 are swapped. Since $20 < 21$, we perform another swap and since $12 < 20$, we stop with 20 being the left child of 12.

3 PICTURES HERE FROM TOP OF PAGE 6

When `ShuffleUp` finishes, we also satisfy the heap property. Thus, the insertion procedure completes. Since a binary heap of n elements has height of $O(\log n)$, an insert operation can finish in $O(\log n)$ time. The pseudocode of the operation is given below.

Procedure Insert(x)
begin
$H(k+1) = x$; $i = k+1$; done = false;
while $i \neq 1$ and not done
begin
    parent = $\lfloor i/2 \rfloor$;
    if H(parent) $> H(i)$ then
        swap(H(parent), $H(i)$);
        $i =$ parent;
    else done = true;
end-while
end

Now, let us look at the main operation of DeleteMin(). The operation DeleteMin is similar to that of insert. In a heap, the minimum element is always available at the root. So it can be located easily. The difficult part is to physically delete the element. Lazy deletion is not an option here. Since the number of elements after deletion has to go down by 1, the element placed at index n presently has to be relocated eleswhere. This is done by first relocating it to the root, satisfying the complete binary tree property.

FIGURE FROM PAGE TOP OF PAGE 7 HERE

This relocation may however violate the heap property. To restore the heap property, we perform what is knnown as Shuffle-Down. In Shuffle-Down, analogous to Shuffle-Up, we move the element down the tree until its value is at most the value of any of its children. Every time the value at a node X is bigger than the value of either of its children, the smallest child and X are swapped. The procedure is illustrated by continuing our example.

FIGURE FROM BOTTOM OF PAGE 7 HERE.

At the end of the Shuffle-Down we have ensured that the smallest element is now again at the root and the value at any node is smaller than that of its children. This satisfies the heap property.

Since a heap of n nodes has a height of $O(\log n)$, DeleteMin takes $O(\log n)$ time in the worst case. The pseudocode for the operation is given below.

Procedure DeleteMin(H)
begin
    $\min = H(1)$;
    $H(1) = H(n)$;
    Shuffle-Down($H(1)$);
    return $\min$;
end. Procedure ShuffleDown(x)
begin
    $i =$index(x);
    while $(i \leq n)$ and element $> H(2i)$ or element $> H(2i+1)$ do
        min = $\min\{H(2i), H(2i+1)\}$;
        swap($H(i), \min$);
        $i =$index(min);
    End-while;
End.