Practice 11

Mayra Cristina Berrones Reyes

April 10, 2020

1 Introduction

A queue is a data structure similar to a real life scenario of a line in the groceries. The first person in the queue is the first person to pay their groceries. So viewing it as such, a queue follows the First In First Out (FIFO) rule. First item to go in is the first item that comes out too.

2 Priority queues

In this case, a Priority Queue (PQ) is a special type of queue, because the first item to go in may not necessarily be the first to go out. In PQ each item is comes with a certain priority, and it is served according to this priority. In programing this can be shown as an extra key to the item, for example, in the Dijkstra algorithm, the item can be the edge between two nodes, and the extra key the weight of using that edge. However, generally in many examples we find it as the value of the element itself is considered the assigned priority, like we see in any sorting algorithms. In this case the highest or the lowest value can be considered the highest priority element, depending on our needs.

The basic operations of a PQ can be described as:

- **Add**: Adds an item.
- **Remove**: Removes the highest priority item.
- **Peek**: Returns the highest priority item without removing it.
- **Size**: Returns the number of items in the PQ.
- **IsEmpty**: Returns whether the PQ has no items.

A PQ can be implemented using various data structures such as an array, a linked list, a heap data structure or a binary search tree. In this case the functions of add, remove and peek for each structure has different performances as we can see in Table 1.

Table 1: Performance of the different data structures for PQ in running time

	Peek	**Insert**	**Delete**
Array (sorted)	$\mathcal{O}(1)$	$\mathcal{O}(n)$	$\mathcal{O}(1)$
Array (unsorted)	$\mathcal{O}(n)$	$\mathcal{O}(1)$	$\mathcal{O}(n)$
Linked list (sorted)	$\mathcal{O}(n)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$
Linked list (unsorted)	$\mathcal{O}(1)$	$\mathcal{O}(n)$	$\mathcal{O}(n)$
Binary heap	$\mathcal{O}(1)$	$\mathcal{O}\log n$	$\mathcal{O}\log n$
Binary search tree	$\mathcal{O}(1)$	$\mathcal{O}\log n$	$\mathcal{O}\log n$

The most common data structure used for PQ is binary heap. We are going to show an example of sorting problem in each structure to show why the binary heap is the preferred method.

2.1 Arrays for PQ

Starting with the arrays. This data structure is the least favored one because, unlike all of the other structures, we have to have a fixed number of items in mind to be able to save the memory space. If we are working with an unsorted array, inserting items is very simple, since we can just add them to the end of the queue, so this is the complexity $\mathcal{O}(1)$.

Imagining that we have enough space for out items in the PQ, the next thing we need to do is de-queue our PQ. Here is where the complexity becomes $\mathcal{O}(n)$, because in the worst case scenario, to search the item with the highest priority we must do a linear scan to find it, and if it is at the end of the array, we need to pass all the other elements to get to it.

Same thing happens to remove, in this case, because there can not be an empty space between spaces in arrays, so the worst case scenario in this is if the highest priority item is at the beginning, since we have to shift all elements to cover the vacant space, that is also complexity $\mathcal{O}(n)$.

In the case of the sorted array, the insertion becomes more complex, because we can not just add the element at the end of the array, we must find where to insert the element. This in itself can be done in log time, but then we have to move all the other items to make space for the one we are inserting, which then becomes $\mathcal{O}(n)$.

Peek and remove become constant time $\mathcal{O}(1)$ because if the array is sorted, then the element with the highest priority is at the end. Removing it will not move the other items.

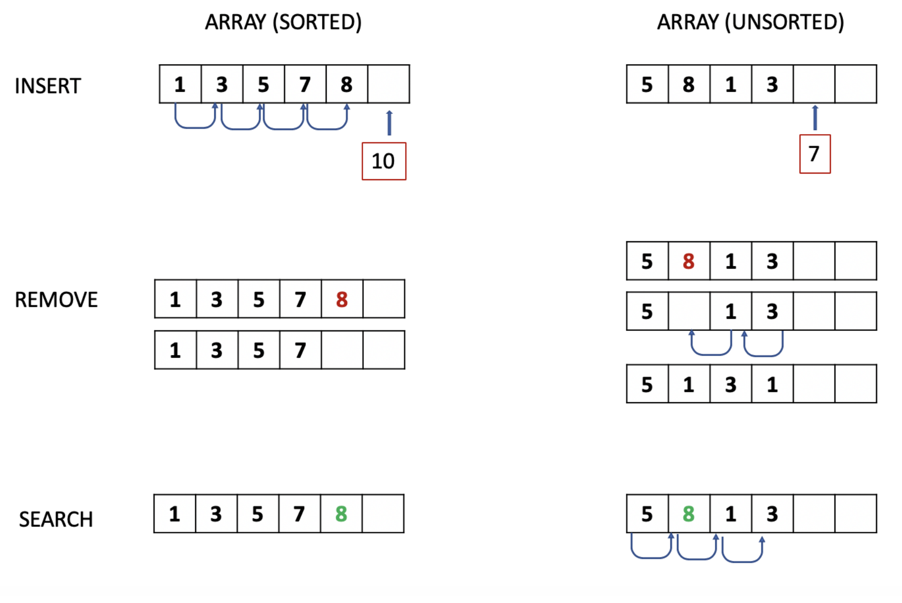

Figure 1: Examples of array features.

2.2 Linked lists for PQ

Like arrays the linked list structure is linear, but unlike arrays, this structure is not stored at a contiguous location in memory. This elements are linked using pointers. They also are dynamic, so they are not constrained by a fixed size.

For the unsorted and sorted linked list the same rules apply as the unsorted and sorted array. The difference is that in the unsorted linked list is in the remove part, where it has the chance to become log because it does not have to rearrange all the other items after removing one.

2.3 Binary search tree for PQ

The structure for a binary search tree (BST) has the following properties:

- The left child of a node contains only the node with lesser key.
- The right child contains the node with a greater key.
- There can not be duplicate nodes.

For this structure the insertion procedure is quite similar to searching. We start from the root node and in a recursive way we go down the tree following the properties of a BST. If the item is already on the tree, then we are done, since duplicates are not allowed.

For searching it can have a $\mathcal{O}(h)$ complexity because it depends on the size of the tree. This could even turn to a $\mathcal{O}(n)$ complexity, because a BST can degenerate to a linked list in certain cases.

This would not happen if we are working with a self balancing binary search tree such as a AVL tree or a red-black tree. In this case finding the minimum en maximum element will be $\mathcal{O}(1)$. Inserting and removing elements will be $\mathcal{O}\log n$ complexity, because this means that, in order to do this procedure, they do not have to go trough all of the elements in the tree, only a certain part.

2.4 Binary heap for PQ

Among all the other data structures, binary heap structure provides a efficient implementation of priority queues. Heap is a complete binary tree in which the root node is the smaller value of all the trees, in this case the convention is that the smallest element is the one with the highest priority.

It is simple to explain the search or peek procedure, because the highest priority element is always going to be on top. So the complexity is $\mathcal{O}(1)$. To insert a new item to the tree, we put it to the last space (the next leaf from left to right) and then we bubble up the element until it reaches its appropriate place in the tree. Since the tree we are working on is a complete tree, it has minimum height and the worst number of moves has $\mathcal{O}\log n$ complexity.

To begin the de-queueing we need to remove the element on top. Once this is done, we replace the root node with the last node of the tree (the last leaf from left to right) and then we bubble it down until it reaches its appropiate place in the tree. Since it is similar to the insertion, it also has a complexity of $\mathcal{O}\log n$.

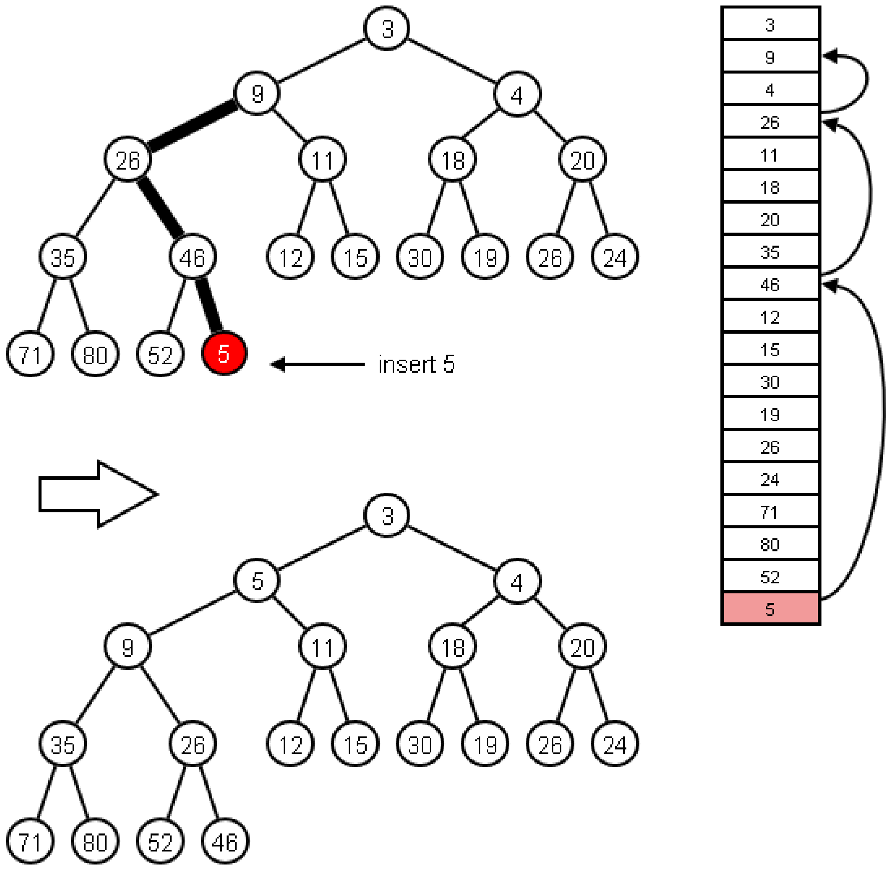

Figure 2: Examples of heap insertion from [1].

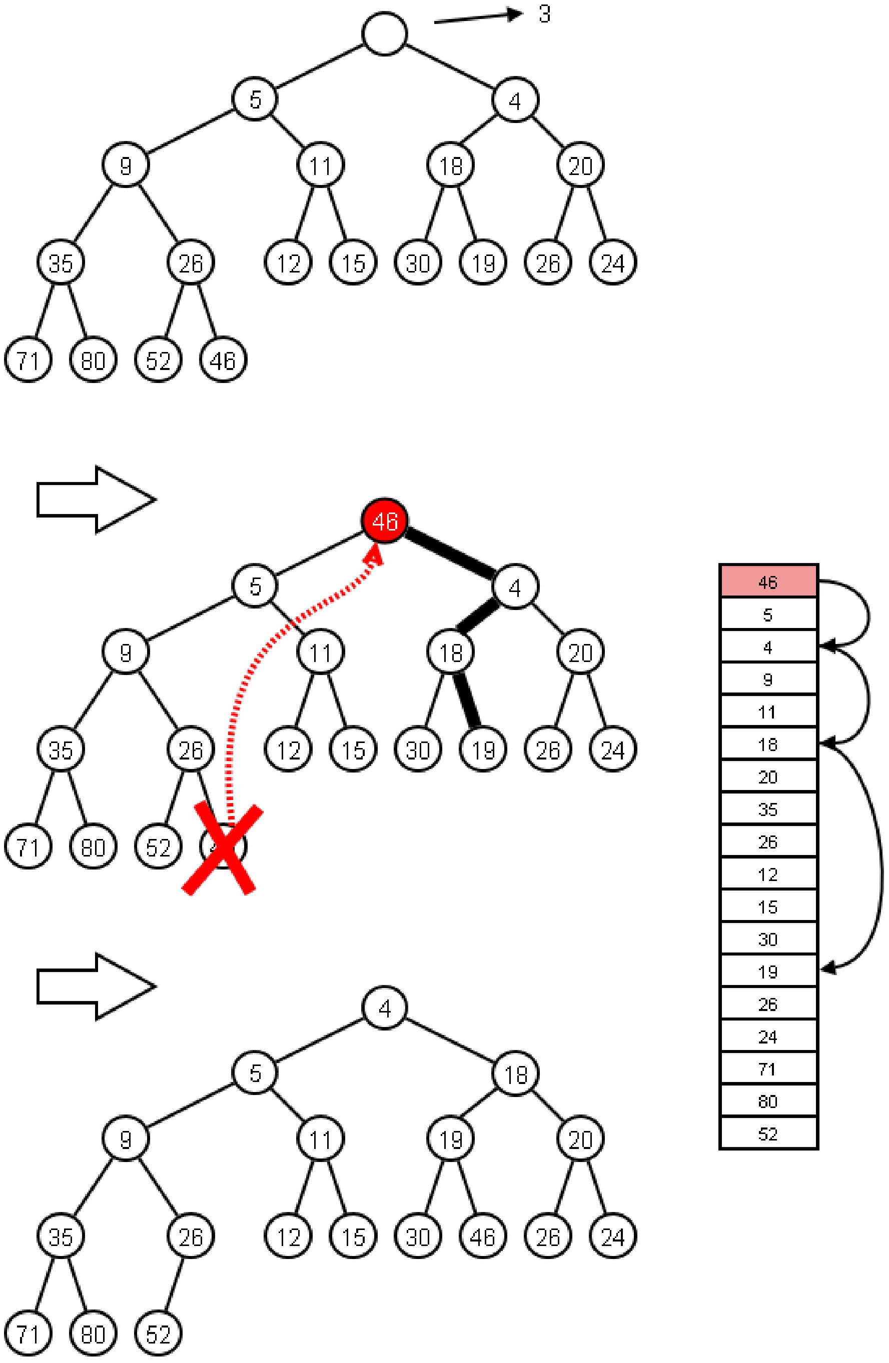

Figure 3: Examples of heap delete from [1]

3 Conclusions

As we stated before, the binary heap structure is the preferred method to solve priority queues. In this case, as in many other practices we have done, it depends on the features of your problem to decide which structure to use.

If we had few elements in our queue, it would be easier to implement a simple array or a linked list. In this case, the array has the advantage to have random access to its elements, and it has more variety of options to use a sorting algorithm, since a linked list can only use a few efficiently. In the case of an unknown set of elements, a linked list would be more appropriated, since has a dynamic size and it is easier to perform the delete and insert tasks.

However, if the number of elements to sort becomes to great, the $\mathcal{O}(n)$ complexity becomes a problem, and the $\mathcal{O}\log n$ complexity of the trees becomes more efficient. In this case I had a hard time deciding which one is best, because from all that I read they seemed to have similar advantages, but when I had the chance to visualize the BST without the self balancing feature, I realized why the heap method is more commonly used. It also seems more cleaner to me in the way that it arranges its nodes.

In Figure 4 I made a BST and a binary heap with the same sequence of numbers (4, 5, 8, 11, 12,6, 9, 7) and that is what I meant when I said the binary heap looks cleaner than BST without the self balancing.

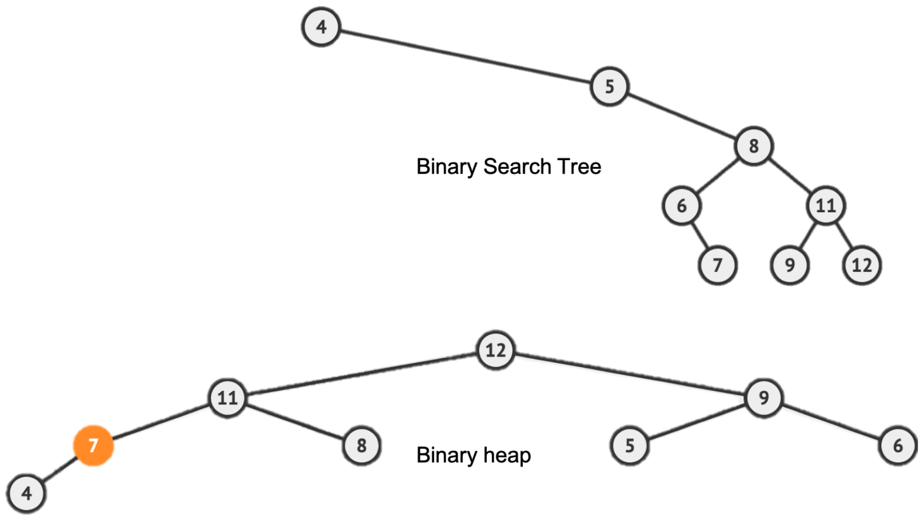

Figure 4: Difference of structure of Binary search tree and binary heap.

References

[1] Priority queues `https://cs.lmu.edu/~ray/notes/pqueues/`

[2] Data structures - Priority queues `https://www.programiz.com/dsa/priority-queue`

[3] Linked lists `https://www.geeksforgeeks.org/linked-list-set-1-introduction/`

[4] Binary Search trees `https://www.geeksforgeeks.org/binary-search-tree-set-1-search-and-insertion/`

[5] Binary heaps `https://www.geeksforgeeks.org/binary-heap/`

[6] Binary search tree visualization `https://visualgo.net/en/bst`

[7] Binary trees `https://www.cs.cmu.edu/~adamchik/15-121/lectures/Trees/trees.html`