Priority Queues (with code in C, C++, and Java)

Introduction

Regular queue follows a First In First Out (FIFO) order to insert and remove an item. Whatever goes in first, comes out first. However, in a priority queue, an item with the highest priority comes out first. Therefore, the FIFO pattern is no longer valid.

Every item in the priority queue is associated with a priority. It does not matter in which order we insert the items in the queue, the item with higher priority must be removed before the item with the lower priority. If the two items have same priorities, the order of removal is undefined and it depends upon the implementation.

The real world example of a priority queue would be a line in a bank where there is a special privilege for disabled and old people. The disabled people have the highest priority followed by elderly people and the normal person has the lowest priority.

Operations on a priority queue

Just like the regular queue, priority queue as an abstract data type has following operations.

1. **EnQueue**: EnQueue operation inserts an item into the queue. The item can be inserted at the end of the queue or at the front of the queue or at the middle. The item must have a priority.
2. **DeQueue**: DeQueue operation removes the item with the highest priority from the queue.
3. **Peek**: Peek operation reads the item with the highest priority.

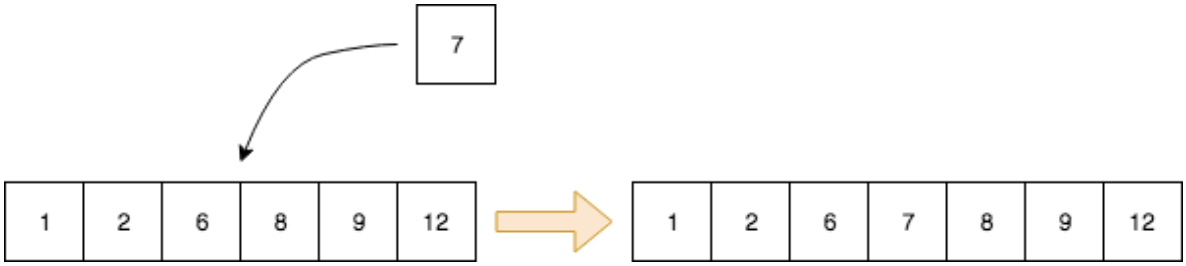
The complexity of these operation depends upon the underlying data structure being used.

Implementation

A priority queue can be implemented using data structures like arrays, linked lists, or heaps. The array can be ordered or unordered.

Using an ordered array

The item is inserted in such a way that the array remains ordered i.e. the largest item is always in the end. The insertion operation is illustrated in figure 1.



(

Fig 1: Insertion operation in an ordered array

The item with priority 7 is inserted between the items with priorities 6 and 8. We can insert it at the end of the queue. If we do so the array becomes unordered. Since we must scan through the queue in order to find the appropriate position to insert the new item, the worst-case complexity of this operation is O(n). Since the item with the highest priority is always in the last position, the dequeue and peek operation takes a constant time. The C program below implements the enqueue and dequeue operation using an ordered array.

```
// insert an item at the appropriate position of the
// queue so that the queue is always ordered
void enqueue(int item) {
        // Check if the queue is full
        if (n == MAX_SIZE - 1) {
                printf("%s\n", "ERROR: Queue is full");
                return;
        }

        int i = n - 1;
        while (i >= 0 && item < queue[i]) {
                queue[i + 1] = queue[i];
                i--;
        }
        queue[i + 1] = item;
        n++;
}

// remove the last element in the queue
int dequeue() {
        int item;
        // Check if the queue is empty
        if (n == 0) {
                printf("%s\n", "ERROR: Queue is empty");
                return -999999;
        }
        item = queue[n - 1];
        n = n - 1;
        return item;
}
```

Using an unordered array

We insert the item at the end of the queue. While inserting, we do not maintain the order. The complexity of this operation is O(1). Since the queue is not ordered, we need to search through the queue for the item with maximum priority. Once we remove this item, we need to move all the items after it one step to the left. The dequeue operation is illustrated in figure 2.

(

Fig 2: Dequeue operation in an unordered array

It is obvious that the complexity of dequeue and peek operation is O(n). The following C program implements the priority queue using an unordered array.

```
// insert an item at the rear of the queue
void enqueue(int item) {
        // Check if the queue is full
        if (n == MAX_SIZE - 1) {
                printf("%s\n", "ERROR: Queue is full");
                return;
        }
        queue[n++] = item;
}

// removes the item with the maximum priority
// search the maximum item in the array and replace it with
// the last item
int dequeue() {
        int item;
        // Check if the queue is empty
        if (n == 0) {
                printf("%s\n", "ERROR: Queue is empty");
                return -999999;
        }
        int i, max = 0;
        // find the maximum priority
        for (i = 1; i < n; i++) {
                if (queue[max] < queue[i]) {
                        max = i;
                }
        }
        item = queue[max];

        // replace the max with the last element
        queue[max] = queue[n - 1];
        n = n - 1;
        return item;
}
```

Using a linked list

The linked can be ordered or unordered just like the array. In the ordered linked list, we can insert item so that the items are sorted and the maximum item is always at the end (pointed by head or tail). The complexity of enqueue operation is O(n) and dequeue and peek operation is O(1).

In an unordered linked list, we can insert the item at the end of the queue in constant time. The dequeue operation takes linear time (O(n)) because we need to search through the queue in order to find the item with maximum priority.

Using a binary heap

In above implementations using arrays and linked lists, one operation always takes linear time i.e. O(n). Using binary heaps (https://algorithmtutor.com/Data-Structures/Tree/Binary-Heaps/), we can make all the operations faster (in logarithmic time). Please read about the binary heaps before using them in a priority queue.

Using a binary heap, the enqueue operation is insert operation in the heap. It takes O(log n) time in the worst case. Similarly, the dequeue operation is the extract-max or remove-max operation which also takes O(log n) time. The peek operation is a constant time operation. In this way, the binary heap makes the priority queue operations a way faster.

The C, C++, and Java implementation of a priority queue using the binary heap is given below.

C C++ Java

```c
// C implementation of a max priority queue
#include < stdio.h >

        #define MAX_SIZE 15

// returns the index of the parent node
int parent(int i) {
        return (i - 1) / 2;
}

// return the index of the left child
int left_child(int i) {
        return 2 * i + 1;
}

// return the index of the right child
int right_child(int i) {
        return 2 * i + 2;
}

void swap(int * x, int * y) {
        int temp = * x;
        * x = * y;
        * y = temp;
}

// insert the item at the appropriate position
void enqueue(int a[], int data, int * n) {
        if ( * n >= MAX_SIZE) {
                printf("%s\n", "The heap is full. Cannot insert");
                return;
```

```
        }
        // first insert the time at the last position of the array
        // and move it up
        a[ * n] = data;
        * n = * n + 1;

        // move up until the heap property satisfies
        int i = * n - 1;
        while (i != 0 && a[parent(i)] < a[i]) {
                swap( & a[parent(i)], & a[i]);
                i = parent(i);
        }
}

// moves the item at position i of array a
// into its appropriate position
void max_heapify(int a[], int i, int n) {
        // find left child node
        int left = left_child(i);

        // find right child node
        int right = right_child(i);

        // find the largest among 3 nodes
        int largest = i;

        // check if the left node is larger than the current node
        if (left <= n && a[left] > a[largest]) {
                largest = left;
        }

        // check if the right node is larger than the current node
        if (right <= n && a[right] > a[largest]) {
                largest = right;
        }

        // swap the largest node with the current node
        // and repeat this process until the current node is larger than
        // the right and the left node
        if (largest != i) {
                int temp = a[i];
                a[i] = a[largest];
                a[largest] = temp;
                max_heapify(a, largest, n);
        }

}

// returns the  maximum item of the heap
int get_max(int a[]) {
        return a[0];
}

// deletes the max item and return
int dequeue(int a[], int * n) {
        int max_item = a[0];

        // replace the first item with the last item
        a[0] = a[ * n - 1];
```

```
        * n = * n - 1;

        // maintain the heap property by heapifying the
        // first item
        max_heapify(a, 0, * n);
        return max_item;
}

// prints the heap
void print_heap(int a[], int n) {
        int i;
        for (i = 0; i < n; i++) {
                printf("%d\n", a[i]);
        }
        printf("\n");
}

int main() {
        int n = 10;
        int a[MAX_SIZE];
        insert(a, 55, & n);
        insert(a, 56, & n);
        insert(a, 57, & n);
        insert(a, 58, & n);
        insert(a, 100, & n);
        print_heap(a, n);
        return 0;
}
```

Complexity

The complexity of the operations of a priority queue using arrays, linked list, and the binary heap is given in the table below.

Data Structure	EnQueue	DeQueue	Peek
Ordered Array	O(n)	O(1)	O(1)
Unordered Array	O(1)	O(n)	O(n)
Ordered Linked List	O(n)	O(1)	O(1)
Unordered Linked List	O(1)	O(n)	O(n)
Binary Heap	O(log n)	O(log n)	O(1)

array (/tags/array/) binary heap (/tags/binary-heap/)
priority queue (/tags/priority-queue/) queue (/tags/queue/)

Binary Heaps (/Data-Structures/Tree/Binary-Heaps/)

Binomial Heaps (/Data-Structures/Tree/Binomial-Heaps/)

Contact Us (https://goo.gl/forms/qNqf8R99NZkKnKMD3) About Us