Queue

A queue is a linear data structure that stores data in an order known as the First In First Out order. This property is helpful in certain programming cases where the data needs to be ordered.

Queue

Queue

Lasindi [Publidomain]

Queues can be visualised like a real-life queue of people. A person may join the queue from the end and may leave tit from the front. The first person to enter leaves first.

A ticket counter can be an example where the people standing in the queue get their tickets one by one and leave the queue.

Operations in a Queue

The two primary operations in a queue are the enqueue and the dequeue operation:

Enqueue Operation

The Enqueue is used to add an element to the queue. The element always gets added to the end of the current queue items.

Dequeue Operation

The Dequeue is used to remove an element from the queue. The element always gets removed from the front of the queue.

The front and rear pointer

To efficiently add or remove data from the queue, two special pointers are used which keep track of the first and last element in the queue. These pointers update continuously and keep a check on the overflow and underflow conditions.

The front pointer always points to the position where an element would be dequeued next. The rear pointer always points to the position where an element would be enqueued next.

Overflow and Underflow Conditions

A queue may have a limited space depending on the

implementation. We must implement check conditions to see if we are not adding or deleting elements more than it can maximum support.

The underflow condition checks if there exists any item before popping from the queue. An empty one cannot be dequeued further.

```
if(front == rear)
    underflow condition
```

The overflow condition checks if the queue is full (or more memory is available) before enqueueing any element. This prevents any error if more space cannot be allocated for the next item.

```
if(rear == SIZE-1)
    overflow condition
```

Creating a queue

A queue can be created using both an array or through a linked list. For simplicity, we will create a queue with an array.

Create a one dimensional array with the above defined SIZE.

(int queue[SIZE])

Define two integer variables front and rear and initialize both with '-1'. (nit front = -1, rear = -1)

```
#define SIZE 10
int queue[SIZE];
int front = -1, rear = -1;
```

Enqueue Operation

Check whether the queue is FULL (rear == SIZE - 1).

If it is FULL, then display an error and terminate the function.

If it is NOT FULL, then increment the rear value by one (rear++)and set queue[rear] = value.

```
void enQueue(int value) {
if(rear == SIZE-1)
printf("\nOverflow. Queue is Full.");
else{
if(front == -1)
front = 0;
rear++;
```

```
queue[rear] = value;
printf("\nInsertion was successful");
    }
}
```

Dequeue Operation

Check whether the queue is EMPTY. (front == rear)

If it is EMPTY, then display an error and terminate the function.

If it is NOT EMPTY, then increment the front value by one (front++)Then display the queue[front] as the deleted element.

Then check whether both front and rear are equal (front == rear)if it TRUE, then set both front and rear to '-1' (roht = rear = -1).

```
void deQueue() {
if(front == rear)
printf("\nUnderflow. Queue is Empty.");
else{
printf("\nDeleted item is: %d", queue[front]);
front++;
if(front == rear)
```

```
front = rear = -1;
    }
}
```

Variations of a queue

A queue can have some variations which make it useful in certain situations:

Double-Ended queue (Deque)

In a standard queue, insertion can only be done from the back and deletion only from the front. A double-ended queue allows for insertion and deletion from both ends.

Circular Queue (Circular Buffer)

A circular queue uses a single, fixed-size buffer as if it were connected end-to-end like a circle.

Circular Queue with both the pointers

Circular Queue with both the pointers

Cburnett, derivative work: Pluke [CC BY-SA 3.0]

This is an efficient implementation for a queue that has fixed maximum size. There is no shifting involved and the whole

queue can be used for storing all the elements.

Priority Queue

A priority queue assigns a priority to each element in the queue. This priority determines which elements are to be deleted and processed first. There can be different criteria's for the priority queue to assign priorities.

An element with the highest priority gets processed first. If there exist two elements with the same priority, then the order of which the element was inserted is considered.

Queue Complexity

Access

An arbitrary element in a queue can only be accessed by continuously shifting the front element. The time complexity is hence $O(n)$.

Search

Similarly, searching an element will involve continuously shifting the front element off the queue until the required element is

found. The time complexity is hence O(n).

Insertion

Inserting an element is only possible at the rear. There is no interaction needed with the rest of the elements. It is hence an O(1) operation.

Deletion

Similar to insertion, deleting an element is only possible from the front of the queue. There is no interaction needed with the rest of the elements. It is hence an O(1) operation.

Space Required

A queue only takes the space used to store the elements of the data type specified. This means that for storing n elements, the space required is O(n).

Applications of Queues in Programming

CPU Scheduling: Various CPU scheduling algorithms make use of this data structure to implement multiprocessing.

Synchronization during data transfer: Asynchronous data