Queue

Topics

1. Introduction to Queue
2. Implementation of Queue using Arrays
3. Implementation of Queue using Linked List
4. Details of Inbuilt Queue library in C++, Java & Python
5. Introduction to Deque
6. Details of Inbuilt Deque library in C++, Java & Python
7. Problems & Solutions

Introduction

Like Stack, Queue is a linear structure which follows a particular order in which the operations are performed. The order is First In First Out [FIFO] or Last In Last Out [LILO]. A good example of a queue is any queue of consumers for a resource where the consumer that came first is served first. The difference between stacks and queues is in the order of removing elements/items. In a stack most recently added item is removed/popped; in a queue, we remove/dequeue the least recently added item.

Queue primarily supports the following basic operations:

1. **Enqueue:** Adds an item to the queue. If the queue is full, then it is said to be an Overflow condition.
   Complexity : O(1)

2. **Dequeue:** Removes an item from the queue. The items are popped in the same order in which they are pushed. If the queue is empty, then it is said to be an Underflow condition.
   Complexity : O(1)

3. **Front:** Get the front item from queue.
   Complexity : O(1)

4. **Rear:** Get the last item from queue.
   Complexity : O(1)

Applications of Queue

Queue is used when things don't have to be processed immediately, but have to be processed in First In First Out order like Breadth First Search. This property of Queue makes it also useful in the following scenarios:

1. When a single resource is shared among multiple consumers, the consumers have to wait in the Queue to access the resource. Examples include: CPU scheduling, Disk Scheduling, etc.
2. When data is transferred asynchronously (data not necessarily received at same rate as sent) between two processes, the data sits in a Queue, until all the fragments of the data is available and then the data is re-structured. Examples include: IO Buffers, pipes, file IO, packet routing over internet etc.

Implementation

There are two ways to implement a queue:

- Using array
- Using linked list

Array implementation of Queue in C/C++

For implementing queue, we need to keep track of two indices, front and rear. We enqueue an item at the rear and dequeue an item from front. If we simply increment front and rear indices, then there may be problems, front may reach end of the array. The solution to this problem is to increase front and rear in circular manner

```
#include <stdio.h>
#include <stdlib.h>
#include <limits.h>

// A structure to represent a queue
struct Queue {
	int front, rear, size;
	unsigned capacity;
	int* array;
};

// Function to create a queue of given capacity. It initializes size of
queue as 0
struct Queue* createQueue(unsigned capacity) {
	struct Queue* queue = (struct Queue*) malloc(sizeof(struct Queue));
	queue->capacity = capacity;
	queue->front = queue->size = 0;
	queue->rear = capacity - 1;  // This is important, see the enqueue
	queue->array = (int*) malloc(queue->capacity * sizeof(int));
	return queue;
}

// Queue is full when size becomes equal to the capacity
int isFull(struct Queue* queue) {
	return (queue->size == queue->capacity);
}

// Queue is empty when size is 0
int isEmpty(struct Queue* queue) {
	return (queue->size == 0);
}
```

```c
// Function to add an item to the queue. It changes rear and size
void enqueue(struct Queue* queue, int item) {
	if (isFull(queue))
		Return;
	queue->rear = (queue->rear + 1)%queue->capacity;
	queue->array[queue->rear] = item;
	queue->size = queue->size + 1;
	printf("%d enqueued to queue\n", item);
}

// Function to remove an item from queue. It changes front and size
int dequeue(struct Queue* queue) {
	if (isEmpty(queue))
		return INT_MIN;
	int item = queue->array[queue->front];
	queue->front = (queue->front + 1)%queue->capacity;
	queue->size = queue->size - 1;
	return item;
}

// Function to get front of queue
int front(struct Queue* queue) {
	if (isEmpty(queue))
		return INT_MIN;
	return queue->array[queue->front];
}

// Function to get rear of queue
int rear(struct Queue* queue) {
	if (isEmpty(queue))
		return INT_MIN;
	return queue->array[queue->rear];
}

// Driver Program
int main() {
	struct Queue* queue = createQueue(1000);

	enqueue(queue, 10);
	enqueue(queue, 20);
	enqueue(queue, 30);
	enqueue(queue, 40);

	printf("%d dequeued from queue\n", dequeue(queue));
	printf("Front item is %d\n", front(queue));
	printf("Rear item is %d\n", rear(queue));

	return 0;
}
```

Linked List implementation of Queue in C/C++

In a Queue data structure, we maintain two pointers, front and rear. The front points the first item of queue and rear points to last item.

```
#include <stdlib.h>
#include <stdio.h>

// A linked list (LL) node to store a queue entry
struct QNode {
	int key;
	struct QNode *next;
};

// The queue, front stores the front node of LL and rear stores the last
node of LL
struct Queue {
	struct QNode *front, *rear;
};

// A utility function to create a new linked list node.
struct QNode* newNode(int k) {
	struct QNode *temp = (struct QNode*)malloc(sizeof(struct QNode));
	temp->key = k;
	temp->next = NULL;
	return temp;
}

// A utility function to create an empty queue
struct Queue *createQueue() {
	struct Queue *q = (struct Queue*)malloc(sizeof(struct Queue));
	q->front = q->rear = NULL;
	return q;
}

// The function to add a key k to q
void enQueue(struct Queue *q, int k) {
	// Create a new LL node
	struct QNode *temp = newNode(k);

	// If queue is empty, then new node is front and rear both
	if (q->rear == NULL) {
		q->front = q->rear = temp;
		return;
	}

	// Add the new node at the end of queue and change rear
	q->rear->next = temp;
	q->rear = temp;
}
```

```c
// Function to remove a key from given queue q
struct QNode *deQueue(struct Queue *q) {
	// If queue is empty, return NULL.
	if (q->front == NULL)
		return NULL;

	// Store previous front and move front one node ahead
	struct QNode *temp = q->front;
	q->front = q->front->next;

	// If front becomes NULL, then change rear also as NULL
	if (q->front == NULL)
		q->rear = NULL;
	return temp;
}

// Driver Program
int main() {
	struct Queue *q = createQueue();
	enQueue(q, 10);
	enQueue(q, 20);
	deQueue(q);
	deQueue(q);
	enQueue(q, 30);
	enQueue(q, 40);
	enQueue(q, 50);
	struct QNode *n = deQueue(q);
	if (n != NULL)
		printf("Dequeued item is %d", n->key);
	return 0;
}
```

- **Array Implementation & Linked List Implementation of Queue in Java.**
- **Array/List Implementation & Linked List Implementation of Queue in Python.**

Details of Inbuilt Queue

- C++

  Declaration

  ```
  queue<int> myQueue;
  ```

  Functions:

  - Push new element: `myQueue.push();`
  - Front element: `myQueue.front();`
  - Pop front element: `myQueue.pop();`
  - Last (rear) element: `myQueue.back();`
  - Size of the queue: `myQueue.size();`
  - Check Empty: `myQueue.empty();`

  More details at **CPPReference - `Queue`.**

- Library Details for **Java's Queue Class: Java Docs & How to use Queue Class in Java.**

- Library Details for **Python's Queue Class: Python Docs & How to use Queue Class in Python.**

Introduction to Deque

Deque or **Double Ended Queue** is a generalized version of Queue data structure that allows insertion/enqueue and deletion/dequeue at both ends of the Queue.

Deque primarily supports the following basic operations

1. **`insertFront()`:** Adds an item at the front of Deque.
2. **`insertLast()` :** Adds an item at the rear of Deque.
3. **`deleteFront()`:** Deletes an item from the front of Deque.
4. **`deleteLast()` :** Deletes an item from the rear of Deque.

In addition to the above operations, Deque also supports the following operations:

1. **`getFront()` :** Gets the front item from the Deque.
2. **`getRear()` :** Gets the last item from the Deque.
3. **`isEmpty()` :** Checks whether the Deque is empty or not.
4. **`isFull()` :** Checks whether the Deque is full or not.

Applications of Deque:
Since Deque supports both stack and queue operations, it can be used as both. The Deque data structure supports clockwise and anticlockwise rotations in O(1) time, which can be useful in certain applications. Also, the problems where elements need to be removed and/or added at both ends, in that scenario, Deque can be efficiently used to solve those problems.

Details of Inbuilt Deque

- **C++**

  Declaration:

  ```
  deque<int> dq;
  ```

  Functions:

  - Push at back: `dq.push_back();`
  - Delete from back: `dq.pop_back();`
  - Push at front: `dq.push_front();`
  - Delete from front: `dq.pop_front();`
  - Size of the deque: `dq.size();`
  - Check Empty: `dq.empty();`

  More details at **CPPReference - `Deque`.**

- Library Details for **Java's Queue Class: Java Docs & How to use Deque Class in Java.**

- Library Details for **Python's Queue Class: Python Docs** & **THow to use Deque Class in Python.**

Problems & Solutions

1. Implement Queue using Stacks
   Hint: Use two stacks, one for pushing and another for popping.

2. Design Queue for **`getMin()`** in O(1) time complexity.
   Hint: Use a queue and a deque.

3. Find maximum number in each window of k in an array of size N.
   Example : If array is `{1, 5, 7, 2, 1, 3, 4}` and `k=3`, then
   first window is `{1, 5, 7}` so maximum is 7, print 7, then
   next window is `{5, 7, 2}` so maximum is 7, print 7, then
   next window is `{7, 2, 1}` so maximum is 7, print 7, and so on.
   Final output is : `{7, 7, 7, 3, 4}`
   Solution: Use a deque for maintaining the elements of the window, with the front storing the maximum element for the current window. For the new element of the next window, remove elements from the back of the deque till the elements are smaller than the current element. Also, for every new element inclusion into the window, remove the left out element from the previous window and check the maximum in the new deque window.