Queues

Topics

1. Introduction to Queue
2. Implementation using Arrays and Linked List
3. C++ STL Queue
4. Introduction to Dequeue
5. C++ STL Dequeue
6. Problems

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). A good example of 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 removing. In a stack we remove the item the most recently added; in a queue, we remove the item the least recently added.

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 following kind of scenarios:

1. When a resource is shared among multiple consumers. Examples include CPU scheduling, Disk Scheduling.
2. When data is transferred asynchronously (data not necessarily received at same rate as sent) between two processes. Examples include IO Buffers, pipes, file IO, etc.

Implementation
There are two ways to implement a queue:

- Using array
- Using linked list

Array implementation Of Queue
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);
}
```

```
// 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];
}

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 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;
}
```

```
// 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;
}

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;
}
```

Usage Details for C++

Declaration
queue<int> myQueue;

Functions:

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

More details at http://www.cplusplus.com/reference/queue/queue/.

Usage Details for Java can be found at
https://docs.oracle.com/javase/7/docs/api/java/util/Queue.html.

Introduction to Dequeue
Dequeue or Double Ended Queue is a generalized version of Queue data structure that allows insert and delete at both ends.

Dequeue primarily supports the following basic operations

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

In addition to the above operations, Dequeue also supports following operations

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

Applications of Dequeue:
Since Dequeue supports both stack and queue operations, it can be used as both. The Dequeue 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 both ends can be efficiently solved using Dequeue.

Usage Details for C++

Declaration
dequeue<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 dequeue: dq.size();
- Check Empty: dq.empty();

More details at http://www.cplusplus.com/reference/deque/deque/**.**

Usage Details for Java can be found at
https://docs.oracle.com/javase/7/docs/api/java/util/Deque.html.

Problems

1. Implement Queue using Stacks
   Solution: Use two stacks one for push and one for pop

2. Design Queue for getMin() in O(1) time complexity.
   Solution: One queue and One dequeue

3. Find maximum number in sliding window of k.
   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} maximum is 7, print 7, then
   next window is {7,2,1} maximum is 7, print 7, and so on.
   Final output is : { 7,7,7,3,4 }

   Solution: Use Dequeue 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 dequeue till the elements are smaller than the current element. Also, for every next window, remove the left out element from the dequeue.