C++
Queues

CO658 Data Structures & Algorithms

Topics

- Introduction
- Circular Queues
- Operations

Introduction

- First in is First Out (FIFO).
- Examples : Printer Queue, Input Queue.
- No standard terminology.
  - **Insert**, put, add, enque.
  - **Remove**, delete, get or deque
  - **Rear** : tail, end or back.
  - **Front** : head.
- We'll use the terms Insert, Remove, Front & Rear
- Items are inserted at the rear of the queue
- Items are removed from the front of the queue
- Often the underlying data structure is an array

The Queue

- Two pointers keep track of Last In and First to go
  - **front** : item to remove next
  - **rear** : last inserted item
- As the items are stored in an array front & rear correspond to indexes.
- Pointers avoid having to move the elements within the array each time an operation is performed.

	Index	Item
front	0	9
	1	5
	2	3
rear	3	1
	4	

Remove()

	Index	Item
	0	
front	1	5
	2	3
rear	3	1
	4	

Circular Queue

- As rear == array size -1, no more items can be inserted although there may be free elements at the beginning of the array due to removals.
- In a Circular Queue rear and front wrap their values.
- Sometimes referred to as a Ring Buffer

	Index	Item
	0	
	1	
front →	2	3
	3	1
rear →	4	8

Insert(2)

	Index	Item
rear →	0	2
	1	
front →	2	3
	3	1
	4	8

Insert

front = 0
rear = -1

Index	Item
0	
1	
2	
3	
4	

Insert(9)

front = 0
rear = 0

Index	Item
0	9
1	
2	
3	
4	

Insert(5)

front = 0
rear = 1

Index	Item
0	9
1	5
2	
3	
4	

- Items inserted at the index identified by **rear**.
- On each insertion the **rear** is incremented and the item placed at rear's index.
- **Front** remains unchanged as this references the first element to be removed.

Remove

	Remove()	Remove()	Remove()	Remove()	Remove()
front = 0 rear = 4	front = 1 rear = 4	front = 2 rear = 4	front = 3 rear = 4	front = 4 rear = 4	front = 0 rear = 4

Index	Item
0	9
1	5
2	3
3	1
4	8

Index	Item
0	
1	5
2	3
3	1
4	8

Index	Item
0	
1	
2	3
3	1
4	8

Index	Item
0	
1	
2	
3	1
4	8

Index	Item
0	
1	
2	
3	
4	8

Index	Item
0	
1	
2	
3	
4	

- Items at the **front** index are removed.
- **Front** is incremented after each operation.
- Front wraps to the beginning

Insert (Wrapping)

front = 2
rear = 3

Index	Item
0	
1	
2	3
3	1
4	

Insert(4)

front = 2
rear = 4

Index	Item
0	
1	
2	3
3	1
4	4

Insert(2)

front = 2
rear = 0

Index	Item
0	2
1	
2	3
3	1
4	4

- If the last element of the array is full the rear index will wrap around to the beginning of the array (element 0).
- Check if the queue is full to avoid over writing items.

Activity

- Attempt exercise 1, 2 and 3
  - Implement a Circular Queue

Priority Queue

- Items inserted in order
- Items with lowest key at front therefore lowest number removed first.
- Performance : Insert O(n), Remove O(1)
- Applications include prioritizing the processing of Adjacent Nodes in Graph Algorithms. O/S processor allocation.

Insert

Index	Item
0	
1	
2	
3	
4	

Insert(5)

Index	Item
0	5
1	
2	
3	
4	

Insert(8)

Index	Item
0	5
1	8
2	
3	
4	

Insert(6)

Index	Item
0	5
1	6
2	8
3	
4	

Insert(3)

Index	Item
0	3
1	5
2	6
3	8
4	

Insert(10)

Index	Item
0	3
1	5
2	6
3	8
4	10

- Items inserted based on their key value.
- All elements with a greater key are pushed forward by one index to make room.
- Front is always index 0 (if not empty).

Conclusion

- A data structure that offers fast insertion and searching O(1).
- Relatively easy to implement.
- Usually based on arrays so difficult to expand.
- Performance can degrade as the array becomes full / Clustered
- No ability to visit the data in order.

Activity

- Attempt exercise 4, 5 and 6
  - Implement a Pirority Queue