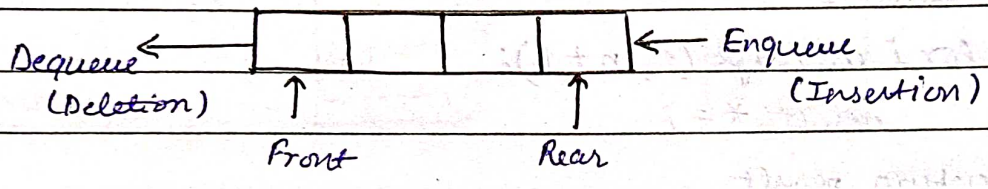
Date:
Page: 50

Queue :-

A queue is an ordered list which enables insert operations to be performed at one end called rear and delete operations to be performed at another end called Front.

First in First Out or last in last out.



Application of Queue :-

1. Waiting list for single shared resource like, printer, disk, CPU.
2. Queues maintain playlist in media player like add or remove.
3. Queues are used in OS for handling process priority.

Complexity :-

Time Complexity →

Worst Case :-

Traversal :- O(n)

Search :- O(n)

Insert :- O(1)

Delete :- O(1)

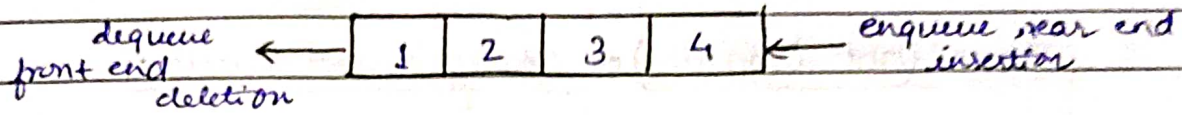
Space Complexity →

Worst Case :- O(n)

Types of Queue :-

Simple or Linear Queue :-

→ Insertion from one end and deletion from another.
Strictly follows FIFO

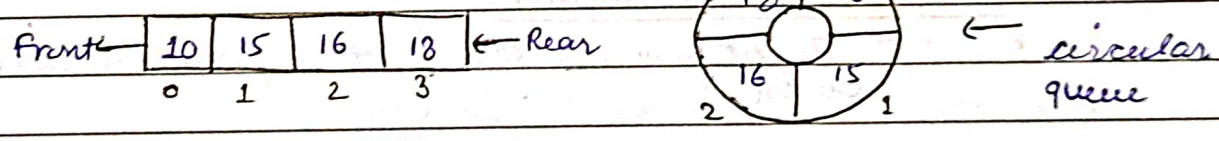


Drawback :-

→ If first three elements are removed i.e. 1,2,3 then we cannot add more elements even after space is left.

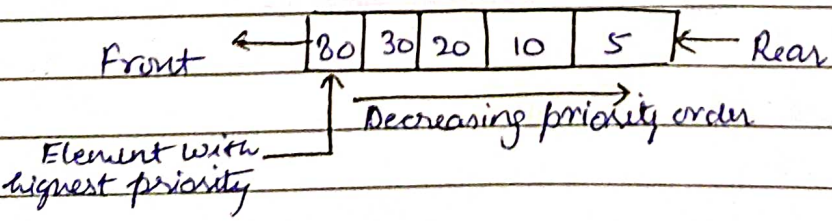
Circular Queue :-

→ Similar to the linear queue but last element is connected to first.
Better memory utilization.



Priority Queue :-

→ Special Queue elements are based on their priority.
→ every element has priority associated with it.
→ If same priority then will arrange according to FIFO order.
→ For CPU scheduling algorithms.

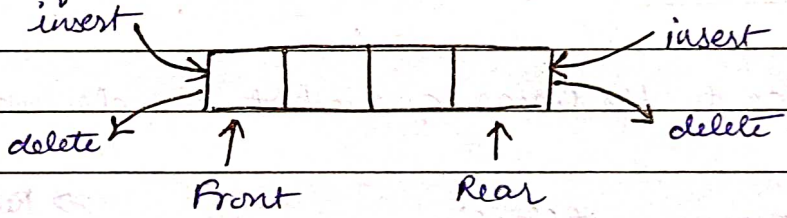


Ascending Priority Queue :-
→ sequence is same but smallest element priority will be deleted first.

Descending Priority Queue :-
→ Sequence is same but largest element priority will be deleted first.

Deque (not Dequeue) or Doubly Ended Queue :-

→ Insertion and deletion can be done from both ends.
→ Deque is used for palindrome checker like if we read the string from both ends, the string would be same.
→ Deque is used both as stack and queue as insertion and deletion from both ends.



Input Restricted Queue :-
→ Input from one end, deletion from both ends (insertion)

Output Restricted Queue :-
→ Output from one end, insertion from both ends (deletion)

P.T.O

Create Queue in Python :-

```
class Queue :
    def __init__(self, capacity):
        self.capacity = capacity
        self.queue = [None] * capacity
        self.front = self.rear = -1
```

Operations in queue :-

1. **Enqueue :-** Used to insert element at the rear end of queue.

```
def enqueue (self, item):
    if self.is_full():
        print("Queue Overflow")
    else:
        if self.is_empty():
            self.front = self.rear = 0
        else:
            self.rear = (self.rear + 1) % self.capacity
        self.queue[self.rear] = item
```

2. **Dequeue :-** Used to delete element from front.

```
def dequeue (self):
    if self.is_empty():
        print("Queue underflow")
        return None
    else:
        removed_element = self.queue[self.front]
```

```
        if self.front == self.rear:
            self.front = self.rear = -1
        else:
            self.front = (self.front + 1) % self.capacity
        return removed element
```

3. Peek :- Displays the front most element.

```
def peek(self):
    if self.is_empty():
        print("Queue is empty")
        return None
    else:
        return self.queue[self.front]
```

4. Queue Overflow (is full) :- Check if queue is full

```
def is_full(self):
    return (self.rear + 1) % self.capacity == self.front
```

5. Queue Underflow (is empty) :- Check if queue is empty

```
def is_empty(self):
    return self.front == -1
```

P.T.O

6. Searching :- Search for item in queue.

```
def search (self, target)
    current_index = self.front
    while current_index != (self.rear+1) % self.capacity:
        if self.queue[current_index] == target:
            print(f"element {target} found in the
                queue at index {current index}")
        current_index = {current_index+1)%self.capacity
    print(f"Element {target} not found in the queue")
    return False
```

Queue using linked list :-

```
class Node:
    def __init__(self, data):
        self.data = data
        self.next = None

class Queue:
    def __init__(self):
        self.rear = None

    def enqueue (self, data):
        new_node = Node(data)
        if self.rear is None:
            self.rear = new_node
        else:
            newNode.next = self.rear
            self.rear = New_Node
```

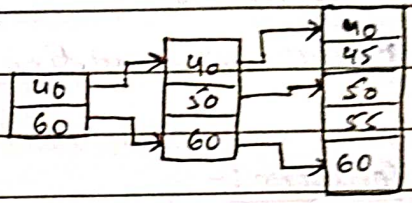
```
def display(self):
    current = self.rear
    while current:
        print(current.data, end=" ")
        current = current.next
    print()

queue = Queue()
queue.enqueue(1)
queue.enqueue(2)

queue.display()
```

Output :-

2 1

Indexed Sequential search :-

→ An index is created initially contain specific groups of Records.
→ Partial indexing is faster.
→ User requests for specific records triggers a search starting within the indexed group.
→ The index is sorted, aiding in efficient search operations.
→ The each element in index points to a block of elements in the array or another expanded index.




```
def index_s_s(arr, key, index_size):
    n = len(arr)
    block_size = n // index_size

    index = [i * block_size for i in range(index_size)]

    block_number = key // block_size

    start = index[block_number]
    end = min(index[block_number+1] if block_number < index_size - 1 else
                                                        n, n)

    for i in range(start, end):
        if arr[i] == key:
            return i

    return -1
array = [10, 20, 30, 40, 50, 60, 70]
Key_to_search = 60
index_size = 3
result = index_s_s(array, Key_to_search, index_size)
print(result)
```

Hashing Concepts :-

Hashing involves converting variable-size input into a fixed-size output using hash functions, determining storage location in a data structure.

Need :-

1. Growing internet data needs effective storage.
2. Arrays, offer O(1) storage, have O(logn) search time inefficient for large data set.
3. Boucles constant-time storage and retrieval.

Collision :-

→ Hashing may lead to collisions (two keys producing same value).
→ Requires collision handling technique.

Advantages :-

Ideal for key-value data structure
Quick access for constant time complexity
Efficient for insertion, deletion, searching
Requires less memory
Performs well with large datasets.
Essential for secure data storage.

Date:
Page: 50

Collisions :- When two or more keys have same hash value.

Collision Resolution techniques :-

1. Seperate Chaining
2. Open address Chaining
   - ↳ Quadratic Probing
   - ↳ Double Hashing
   - ↳ Linear Probing

1. Seperate Chaining :-

→ Creates linked list in slots with collision
→ Adds new keys to the list
→ Suitable for uncertain key counts

Advantage →
→ Easy implementation
→ Allows additional elements

Disadvantage →
→ Poor cache performance
→ High memory wastage

Time Complexity →
Worst Case - searching : O(n)
Worst Case - deletion : O(n)

2. Open Addressing :-

→ Resolves collision without external structures.
→ Hash table size is not less than key count.

Technique :-

a) Linear Probing :-
→ Continues linear search for the next slot
→ Simple but may cluster.

Diadvantage :-

→ Cluster formation

→ Time Consuming for ~~extra~~ empty slot

b) Quadratic Probing :-

→ Probes for i^2 nd slot in it iteration

→ Better Cache Performance

c) Double Hashing :-

→ Uses different hashing algorithm.

→ No clustering, poores cache performance.