Stack and Queue

CPE 112 Programming with Data Structures

Dr. Piyanit Wepulanon
Dr. Taweechai Nuntawisuttiwong

Computer Engineering
King Mongkut's University of Technology Thonburi

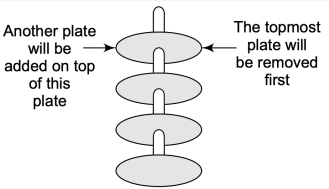
Overview

1. Stack
2. Array Representation of Stack
3. Linked List Representation of Stack
4. Applications of Stack
5. Queue
6. Array Representation of Queues
7. Linked List Representation of Queues
8. Types of Queues
9. Applicaitons of Queues

Stack

Stack

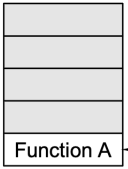
- Stack is an important data structure which stores its elements in an ordered manner.
- We will explain the concept of stacks using an analogy. You must have seen a pile of plates where one plate is placed on top of another as shown in Figure.
- Now, when you want to remove a plate, you remove the topmost plate first. Hence, you can add and remove an element (i.e., a plate) only at/from one position which is the topmost position.



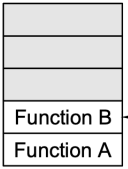
LIFO

Stack in Computer Science

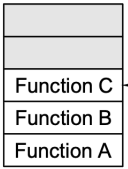
Function Calls



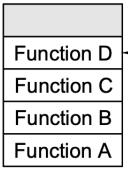
When A calls B, A is pushed on top of the system stack. When the execution of B is complete, the system control will remove A from the stack and continue with its execution.



When B calls C, B is pushed on top of the system stack. When the execution of C is complete, the system control will remove B from the stack and continue with its execution.



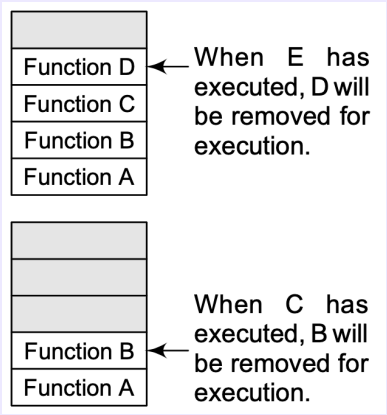
When C calls D, C is pushed on top of the system stack. When the execution of D is complete, the system control will remove C from the stack and continue with its execution.



When D calls E, D is pushed on top of the system stack. When the execution of E is complete, the system control will remove D from the stack and continue with its execution.

Stack in Computer Science

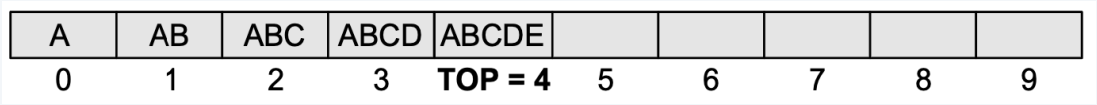
Function Calls



Array Representation of Stack

Array Representation of Stack

In the computer's memory, stacks can be represented as a linear array. Every stack has a variable called TOP associated with it, which is used to store the address of the topmost element of the stack. It is this position where the element will be added to or deleted from.



Operations on a Stack

- A stack supports three basic operations: push, pop, and peek.
- The push operation adds an element to the top of the stack.
- The pop operation removes the element from the top of the stack.
- The peek operation returns the value of the topmost element of the stack.

PUSH Operation

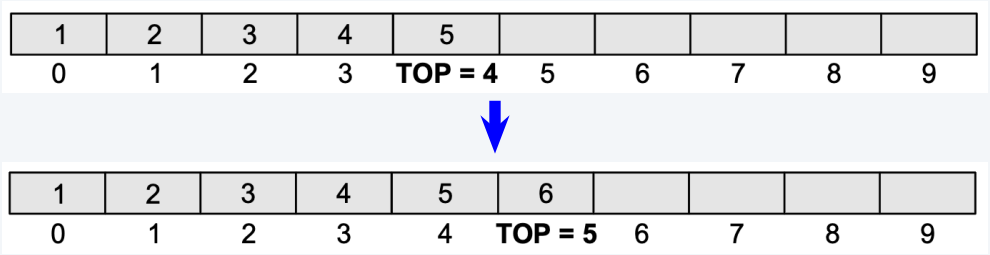
- The push operation is used to insert an element into the stack. The new element is added at the topmost position of the stack.
- However, before inserting the value, we must first check if TOP=MAX-1, becuase if that is the case, then the stack is full and no more insertions can be done.
- If an attempt is made to insert a value in a stack that is already full, an OVERFLOW message is printed.

PUSH Operation

Insert an element with value 6

Check if TOP=MAX-1

- If the condition is false,
  - Increment the value of TOP, and
  - Store the new element at the position given by stack[TOP].



PUSH Operation

```
Step 1: IF TOP = MAX-1
            PRINT "OVERFLOW"
            Goto Step 4
        [END OF IF]
Step 2: SET TOP = TOP + 1
Step 3: SET STACK[TOP] = VALUE
Step 4: END
```

```c
void push(int stack[], int val){
  if(top == MAX-1){
    printf("Overflow\n");
    return;
  }
  top++;
  stack[top] = val;
}
```

POP Operation

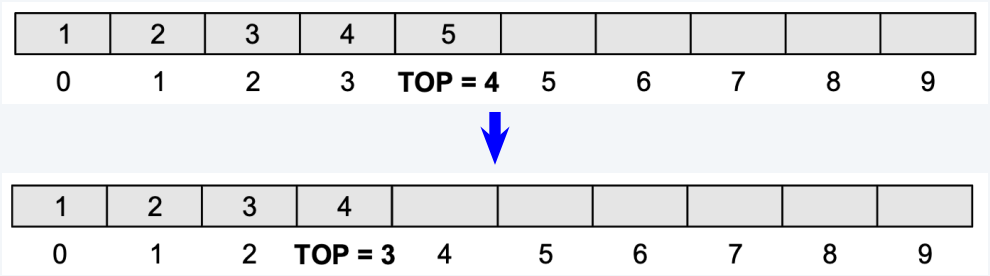
- The pop operation is used to delete the topmost element from the stack.
- However, before deleting the value, we must first check if TOP=NULL because if that is the case, then it means the stack is empty and no more deletions can be done.
- If an attempt is made to delete a value from a stack that is already empty, an UNDERFLOW message is printed.

POP Operation

Delete the topmost element

Check if TOP=NULL

- If the condition is false,
  - Decrement the value pointed by TOP



POP Operation

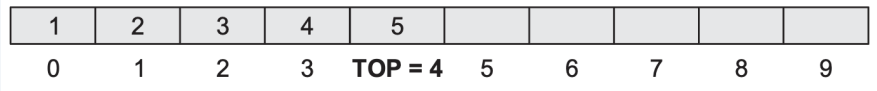
```
Step 1: IF TOP = NULL
            PRINT "UNDERFLOW"
            Goto Step 4
        [END OF IF]
Step 2: SET VAL = STACK[TOP]
Step 3: SET TOP = TOP - 1
Step 4: END
```

```c
int pop(int stack[]){
  int val;

  if(top == -1){
    printf("Underflow\n");
    return -1;
  } else{
    val = stack[top];
    top--;
    return val;
  }
}
```

PEEK Operation

- Peek is an operation that returns the value of the topmost element of the stack without deleting it from the stack.
- However, the Peek operation first checks if the stack is empty, i.e., if TOP = NULL, then an appropriate message is printed, else the value is returned.



PEEK Operation

```
Step 1: IF TOP = NULL
            PRINT "STACK IS EMPTY"
            Goto Step 3
Step 2: RETURN STACK[TOP]
Step 3: END
```

```c
int peek(int stack[]){
  if(top == -1){
    printf("Stack is empty.\n");
    return -1;
  }
  return stack[top];
}
```

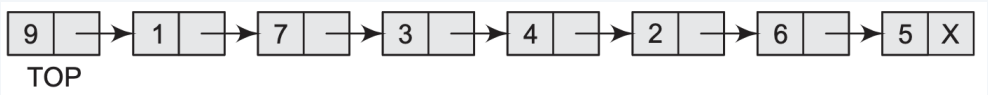
Linked List Representation of Stack

Linked List Representation of Stack

- The technique of creating a stack is easy, but the drawback is that the array must be declared to have some fixed size.
- In case the stack is a very small one or its maximum size is known in advance, then the array implementation of the stack gives an efficient implementation.
- But if the array size cannot be determined in advance, then the other alternative, i.e., linked representation, is used.

Linked List Representation of Stack

- The storage requirement of linked representation of the stack with n elements is O(n), and the typical time requirement for the operations is O(1).
- In a linked stack, every node has two parts—one that stores data and another that stores the address of the next node.
- The START pointer of the linked list is used as TOP. All insertions and deletions are done at the node pointed by TOP. If TOP = NULL, then it indicates that the stack is empty.

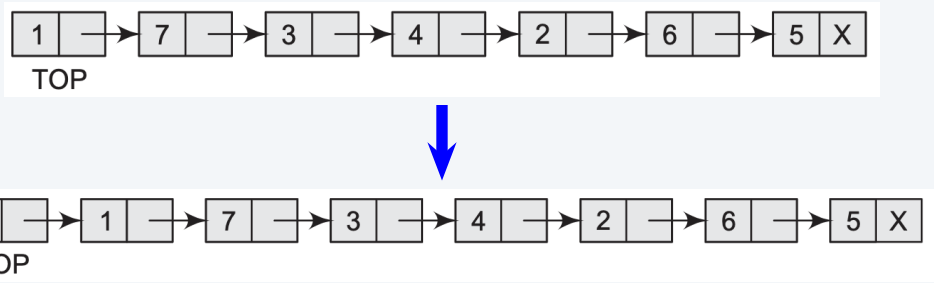


Operations on a Linked List

- A linked stack supports three basic operations: push, pop, and peek.
- The push operation adds an element to the top of the stack.
- The pop operation removes the element from the top of the stack.
- The peek operation returns the value of the topmost element of the stack.

PUSH Operation

Insert an element with value 9



PUSH Operation

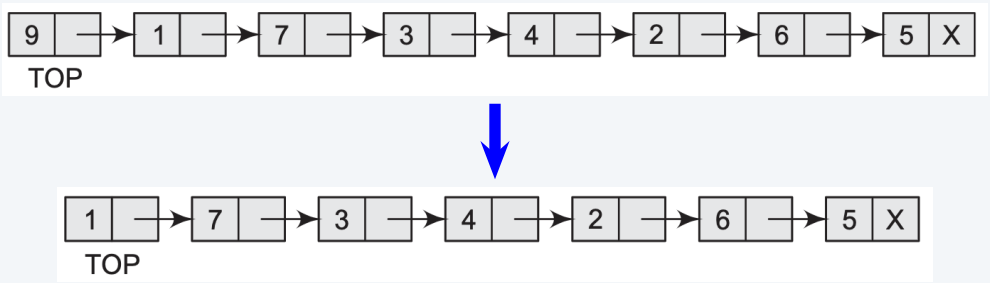
```
Step 1: Allocate memory for the new
        node and name it as NEW_NODE
Step 2: SET NEW_NODE -> DATA = VAL
Step 3: IF TOP = NULL
            SET NEW_NODE -> NEXT = NULL
            SET TOP = NEW_NODE
        ELSE
            SET NEW_NODE -> NEXT = TOP
            SET TOP = NEW_NODE
        [END OF IF]
Step 4: END
```

```c
void push(node **top,int val){
  node *newNode;

  newNode = (node*)malloc(sizeof(node));
  newNode->data = val;
  if(*top == NULL){
    newNode->next = NULL;
    *top = newNode;
  } else{
    newNode->next = *top;
    *top = newNode;
  }
}
```

POP Operation

Delete the topmost element



POP Operation

```
Step 1: IF TOP = NULL
            PRINT "UNDERFLOW"
            Goto Step 5
        [END OF IF]
Step 2: SET PTR = TOP
Step 3: SET TOP = TOP -> NEXT
Step 4: FREE PTR
Step 5: END
```

```c
void pop(node **top){
  node *ptr;

  if(*top == NULL){
    printf("Underflow\n");
    return;
  }
  ptr = *top;
  *top = (*top)->next;
  free(ptr);
}
```

Applications of Stack

Applications of Stack

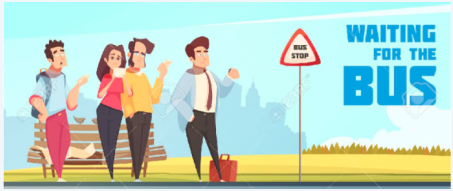
We will discuss typical problems where stacks can be easily applied for a simple and efficient solution. The topics that will be discussed in this section include the following:

- Reversing a list
- Parentheses checker
- Conversion of an infix expression into a postfix expression
- Evaluation of a postfix expression
- Conversion of an infix expression into a prefix expression
- Evaluation of a prefix expression
- Recursion
- Tower of Hanoi

Queue

Queue

- In all these examples, we see that the element at the first position is served first. Same is the case with queue data structure.
- A queue is a FIFO (First-In, First-Out) data structure in which the element that is inserted first is the first one to be taken out.
- The elements in a queue are added at one end called the REAR and removed from the other end called the FRONT. Queues can be implemented by using either arrays or linked lists. Next, we will see how queues are implemented using each of these data structures.

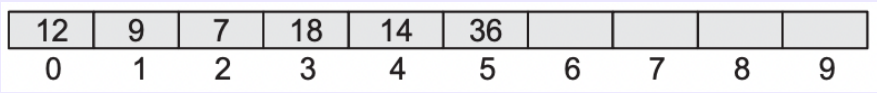


Array Representation of Queues

Array Representation of Queues

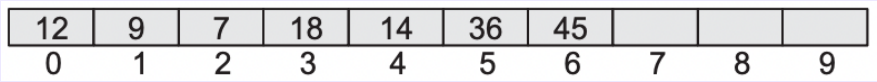
Queues can be easily represented using linear arrays. As stated earlier, every queue has front and rear variables that point to the position from where deletions and insertions can be done, respectively.

Queue

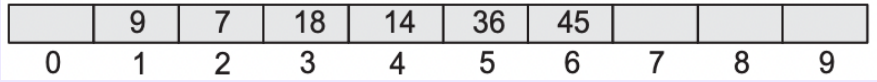


Array Representation of Queues

Queue after insertion of a new element



Queue after deletion of an element



Operations on Queues

- A queue supports three basic operations: insert, and delete.
- The insert operation adds an element to the rear of the queue.
- The delete operation removes the element from the front of the queue.

INSERT Operation

```
Step 1: IF REAR = MAX-1
            Write OVERFLOW
            Goto step 4
        [END OF IF]
Step 2: IF FRONT = -1 and REAR = -1
            SET FRONT = REAR = 0
        ELSE
            SET REAR = REAR + 1
        [END OF IF]
Step 3: SET QUEUE[REAR] = NUM
Step 4: EXIT
```

```
void insertQ(int queue[], int num){
  if(rear == MAX - 1){
    printf("OVERFLOW\n");
    return;
  }
  if(front == -1 && rear == -1){
    front = rear = 0;
  } else{
    rear = rear + 1;
  }
  queue[rear] = num;
}
```

DELETE Operation

```
Step 1: IF FRONT = -1 OR FRONT > REAR
            Write UNDERFLOW
        ELSE
            SET VAL = QUEUE[FRONT]
            SET FRONT = FRONT + 1
        [END OF IF]
Step 2: EXIT
```

```c
int deleteQ(int queue[]){
  int val;

  if(front == -1 || front > rear){
    printf("UNDERFLOW\n");
    return -1;
  } else{
    val = queue[front];
    front++;
    return val;
  }
}
```

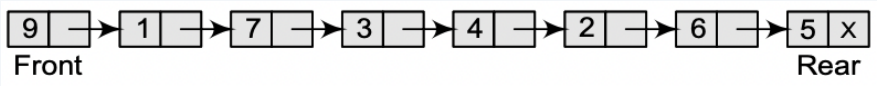
Linked List Representation of Queues

Linked List Representation of Queues

- We have seen how a queue is created using an array. Although this technique of creating a queue is easy, its drawback is that the array must be declared to have some fixed size.
- If we allocate space for 50 elements in the queue and it hardly uses 20– 25 locations, then half of the space will be wasted.
- And in case we allocate less memory locations for a queue that might end up growing large and large, then a lot of re-allocations will have to be done, thereby creating a lot of overhead and consuming a lot of time.
- In case the queue is a very small one or its maximum size is known in advance, then the array implementation of the queue gives an efficient implementation. But if the array size cannot be determined in advance, the other alternative, i.e., the linked representation is used.

Linked List Representation of Queues

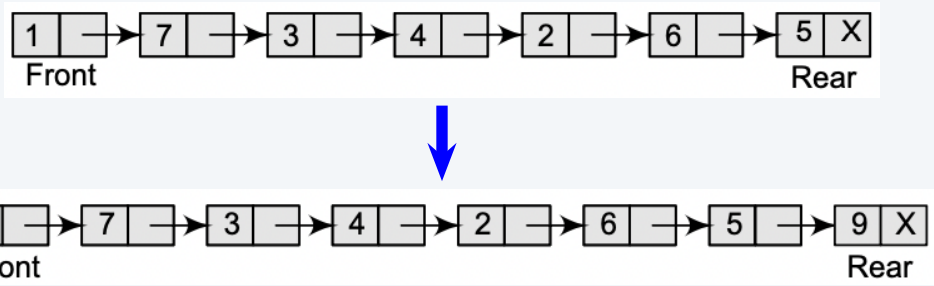
- The storage requirement of linked representation of a queue with n elements is O(n) and the typical time requirement for operations is O(1).
- In a linked queue, every element has two parts, one that stores the data and another that stores the address of the next element.



Operations on Linked Queues

- A queue has two basic operations: insert and delete.
- The insert operation adds an element to the end of the queue, and the delete operation removes an element from the front or the start of the queue.
- Apart from this, there is another operation peek which returns the value of the first element of the queue.

INSERT Operation



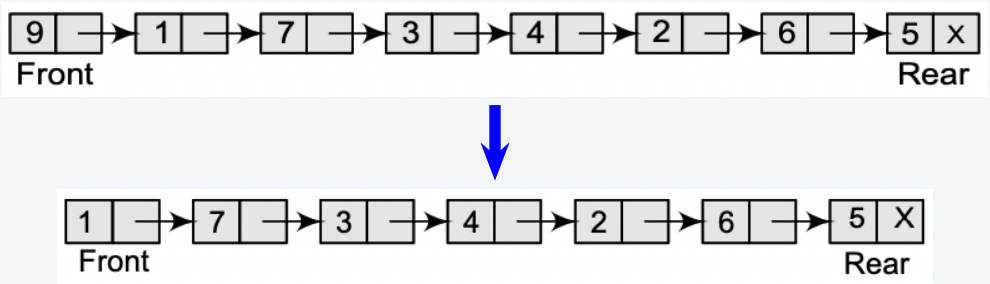
INSERT Operation

```
Step 1: Allocate memory for the new node and name
        it as PTR
Step 2: SET PTR -> DATA = VAL
Step 3: IF FRONT = NULL
            SET FRONT = REAR = PTR
            SET FRONT -> NEXT = REAR -> NEXT = NULL
        ELSE
            SET REAR -> NEXT = PTR
            SET REAR = PTR
            SET REAR -> NEXT = NULL
        [END OF IF]
Step 4: END
```

```
queue insertQ(queue q, int val){
  node *ptr;

  ptr = (node*)malloc(sizeof(node));
  ptr->data = val;
  if(q.front == NULL){
    q.front = ptr;
    q.rear = ptr;
    q.front->next = q.rear->next = NULL;
  } else{
    q.rear->next = ptr;
    q.rear = ptr;
    q.rear->next = NULL;
  }
  return q;
}
```

DELETE Operation



DELETE Operation

```
Step 1: IF FRONT = NULL
            Write "Underflow"
            Go to Step 5
        [END OF IF]
Step 2: SET PTR = FRONT
Step 3: SET FRONT = FRONT -> NEXT
Step 4: FREE PTR
Step 5: END
```

```
queue deleteQ(queue q){
  node *ptr;

  if(q.front == NULL)
    printf("Underflow\n");
  ptr = q.front;
  printf("Delete [%d].\n",q.front->data);
  q.front = q.front->next;
  free(ptr);
  return q;
}
```

Types of Queues

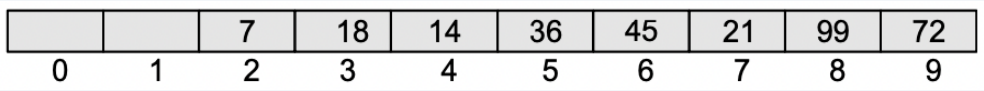
Types of Queues

A queue data structure can be classified into the following types:

- Circular Queue
- Deque
- Priority Queue
- Multiple Queue

Circular Queue

- In linear queues, we have discussed so far that insertions can be done only at one end called the REAR and deletions are always done from the other end called the FRONT.
- Suppose we want to insert a new element in the queue shown in figure. Even though there is space available, the overflow condition still exists because the condition REAR = MAX – 1 still hold true. This is a major drawback of a linear queues.



Circular Queue

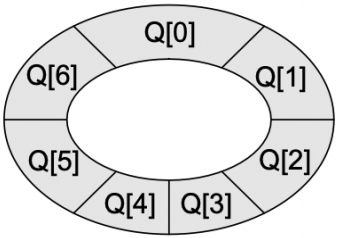
To resolve this problem, we have two solutions.

First Solution

Shift the elements to the left so that the vacant space can be occupied and utilized efficiently. But this can be very time-consuming, especially when the queue is quite large.

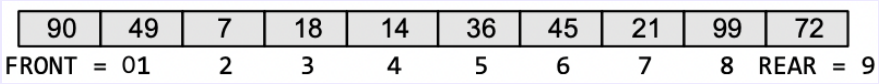
Second Solution

Use a circular queue. In the circular queue, the first index comes right after the last index. Conceptually, you can think of a circular queue as shown in figure.

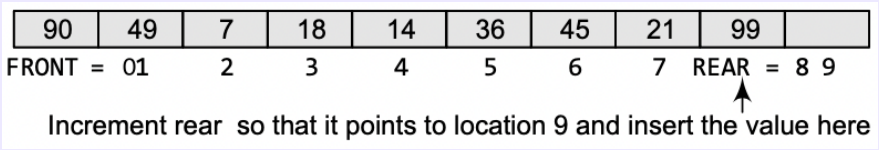


Circular Queue

Full queue

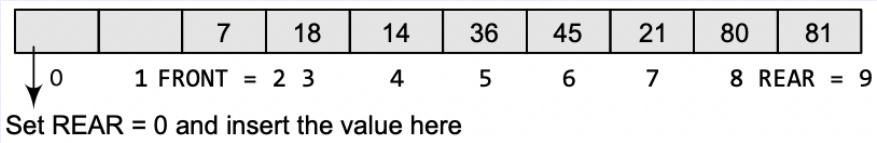


Queue with vacant locations



INSERT Operation

Inserting an element in a circular queue



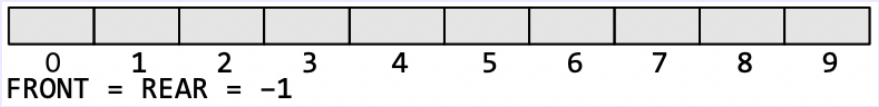
INSERT Operation

```
Step 1: IF FRONT = 0 and Rear = MAX - 1
            Write "OVERFLOW”
            Goto step 4
        [End OF IF]
Step 2: IF FRONT = -1 and REAR = -1
            SET FRONT = REAR = 0
        ELSE IF REAR = MAX - 1 and FRONT != 0
            SET REAR = 0
        ELSE
            SET REAR = REAR + 1
        [END OF IF]
Step 3: SET QUEUE[REAR] = VAL
Step 4: EXIT
```

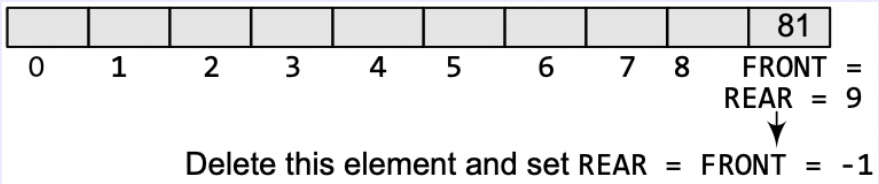
```
void insertQ(int queue[], int val){
  if(front == 0 && rear == MAX-1){
    printf("OVERFLOW");
    return;
  }
  if(front == -1 && rear == -1){
    front = rear = 0;
  } else if(rear == MAX - 1 && front != 0){
    rear = 0;
  } else {
    rear++;
  }
  queue[rear] = val;
}
```

DELETE Operation

Empty Queue

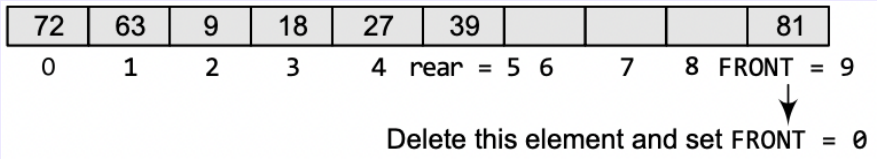


Queue with a single element



DELETE Operation

Queue where FRONT = MA-1 before deletion



DELETE Operation

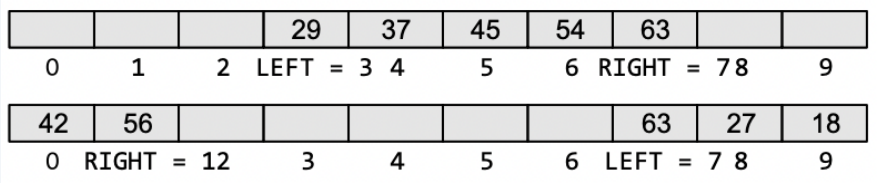
```
Step 1: IF FRONT = -1
            Write "UNDERFLOW"
            Goto Step 4
        [END of IF]
Step 2: SET VAL = QUEUE[FRONT]
Step 3: IF FRONT = REAR
            SET FRONT = REAR = -1
         ELSE
            IF FRONT = MAX -1
                  SET FRONT = 0
            ELSE
                  SET FRONT = FRONT + 1
            [END of IF]
        [END OF IF]
Step 4: EXIT
```

```c
int deleteQ(int queue[]){
  int val;

  if(front == -1){
    printf("UNDERFLOW");
    return -1;
  }
  val = queue[front];
  if(front == rear)
    front = rear = -1;
  else{
    if(front == MAX - 1)
      front = 0;
    else
      front++;
  }
  return val;
}
```

Deques

- A deque (pronounced as ‘deck’ or ‘dequeue’) is a list in which the elements can be inserted or deleted at either end.
- It is also known as a head-tail linked list because elements can be added to or removed from either the front (head) or the back (tail) end.
- However, no element can be added and deleted from the middle. In the computer’s memory, a deque is implemented using either a circular array or a circular doubly linked list.



Priority Queue

A priority queue is a data structure in which each element is assigned a priority. The priority of the element will be used to determine the order in which the elements will be processed. The general rules of processing the elements of a priority queue are

- An element with higher priority is processed before an element with a lower priority.
- Two elements with the same priority are processed on a first-come- first-served (FCFS) basis.

Multiple Queue

- When we implement a queue using an array, the size of the array must be known in advance. If the queue is allocated less space, then frequent overflow conditions will be encountered.
- To deal with this problem, the code will have to be modified to reallocate more space for the array.
- In case we allocate a large amount of space for the queue, it will result in sheer wastage of the memory. Thus, there lies a tradeoff between the frequency of overflows and the space allocated.
- So a better solution to deal with this problem is to have multiple queues or to have more than one queue in the same array of sufficient size.

Applicaitons of Queues

Applicaitons of Queues

- Queues are widely used as waiting lists for a single shared resource like printer, disk, CPU.
- Queues are used to transfer data asynchronously (data not necessarily received at same rate as sent) between two processes (IO buffers), e.g., pipes, file IO, sockets.
- Queues are used as buffers on MP3 players and portable CD players, iPod playlist.
- Queues are used in Playlist for jukebox to add songs to the end, play from the front of the list.
- Queues are used in operating system for handling interrupts. When programming are real-time system that can be interrupted.