Unit III
Stacks

Introduction to stack ADT:

A **stack** is a linear data structure in which insertion and deletion of elements take place only at one end, known as the **TOP** of the stack.

A stack follows the **Last-In, First-Out (LIFO)** principle. This means that the last element inserted into the stack is the first element to be removed.

A stack supports three basic operations:

- **Push** – Adds an element to the top of the stack.
- **Pop** – Removes the top element from the stack.
- **Peek (or Peep)** – Returns the value of the topmost element without removing it.

When elements are stored sequentially (for example, in an array from index 0 to TOP), deletion appears to occur by simply decreasing the TOP pointer. The actual data may remain in memory, but it is no longer considered part of the stack.

A stack can also be defined as an **Abstract Data Type (ADT)**.

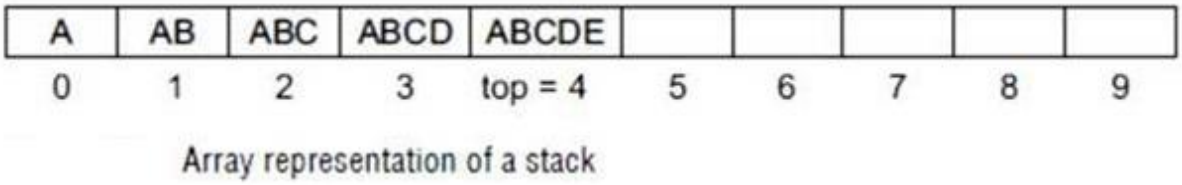
A stack of elements of a particular data type is a finite sequence of elements along with the following operations:

1. **Initialize** – Create an empty stack.
2. **isEmpty** – Check whether the stack is empty.
3. **isFull** – Check whether the stack is full (in case of array implementation).
4. **Push** – If the stack is not full, insert an element at the TOP.
5. **Peek** – If the stack is not empty, retrieve the element at the TOP without removing it.
6. **Pop** – If the stack is not empty, remove the element from the TOP.

Q. Explain Stack and stack ADT

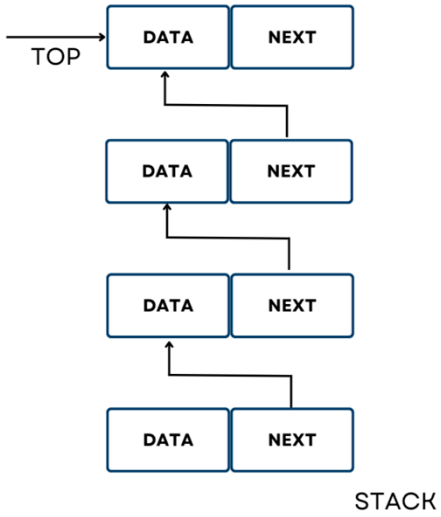
Implementation of stacks using array and Linked List,

In the computer's memory, stacks can be implemented using arrays or linked lists.



Array representation of a stack

The process of storing data onto a stack is called push() and removing data from stack is called pop(). The algorithm for implementing the stack ADT for each operation is as follows:




```
PUSH(STACK, TOP, VAL)
        BEGIN
        IF TOP = MAX - 1 THEN
        PRINT "STACK OVERFLOW"
        ELSE
        TOP = TOP + 1
        STACK[TOP] = VAL
        END IF
        END

POP(STACK, TOP)
        BEGIN
        IF TOP = -1 THEN
        PRINT "STACK UNDERFLOW"
        ELSE
        VAL = STACK[TOP]
        TOP = TOP - 1
        RETURN VAL
```

END IF
END

Things to remember while implementing stack on array:

Stack is empty when TOP = -1
Stack is full when TOP = MAX - 1
Increment TOP before insertion
Decrement TOP after removal
POP should return value

Q. Explain stack implementation on array and Linked list.

****if code related questions are asked in examination, students can write their own logics and even in LAB can have their own logical implementation of code keeping the concept in mind. The given code is just example. Students need not mug up the same. ****

Code in C:

Implementation code of stack on array:

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

#define MAX 5

int stack[MAX];
int top = -1;

/* Function Prototypes */
void push(int value);
int pop();
int peek();
void display();

int main()
{
   int choice, value;

   while(1)
   {
      printf("\n----- STACK MENU -----\n");
      printf("1. PUSH\n");
```

```
        printf("2. POP\n");
        printf("3. PEEK\n");
        printf("4. DISPLAY\n");
        printf("0. EXIT\n");
        printf("Enter your choice: ");
        scanf("%d", &choice);

        switch(choice)
        {
            case 1:
                printf("Enter value to push: ");
                scanf("%d", &value);
                push(value);
                break;

            case 2:
                value = pop();
                if(value != -1)
                    printf("Popped element: %d\n", value);
                break;

            case 3:
                value = peek();
                if(value != -1)
                    printf("Top element: %d\n", value);
                break;

            case 4:
                display();
                break;

            case 0:
                exit(0);

            default:
                printf("Invalid choice!\n");
        }
    }

    return 0;
}

/* Push Operation */
```

```
void push(int value)
{
   if(top == MAX - 1)
   {
      printf("Stack Overflow!\n");
   }
   else
   {
      top++;
      stack[top] = value;
      printf("Element %d pushed successfully.\n", value);
   }
}

/* Pop Operation */
int pop()
{
   if(top == -1)
   {
      printf("Stack Underflow!\n");
      return -1;
   }
   else
   {
      return stack[top--];
   }
}

/* Peek Operation */
int peek()
{
   if(top == -1)
   {
      printf("Stack is Empty!\n");
      return -1;
   }
   else
   {
      return stack[top];
   }
}

/* Display Stack */
```

```
void display()
{
   if(top == -1)
   {
      printf("Stack is Empty!\n");
   }
   else
   {
      printf("Stack elements: ");
      for(int i = top; i >= 0; i--)
      {
         printf("%d ", stack[i]);
      }
      printf("\n");
   }
}
```

Implementation code of stack on linked list:

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

/* Define structure */
struct Node
{
   int data;
   struct Node* next;
};

struct Node* top = NULL;

/* Function Prototypes */
void push(int value);
int pop();
int peek();
void display();

int main()
{
   int choice, value;

   while(1)
```

```
{
    printf("\n----- STACK MENU (Linked List) -----\n");
    printf("1. PUSH\n");
    printf("2. POP\n");
    printf("3. PEEK\n");
    printf("4. DISPLAY\n");
    printf("0. EXIT\n");
    printf("Enter your choice: ");
    scanf("%d", &choice);

    switch(choice)
    {
        case 1:
            printf("Enter value to push: ");
            scanf("%d", &value);
            push(value);
            break;

        case 2:
            value = pop();
            if(value != -1)
                printf("Popped element: %d\n", value);
            break;

        case 3:
            value = peek();
            if(value != -1)
                printf("Top element: %d\n", value);
            break;

        case 4:
            display();
            break;

        case 0:
            exit(0);

        default:
            printf("Invalid choice!\n");
    }
}

return 0;
```

```
}

/* Push Operation */
void push(int value)
{
   struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));

   if(newNode == NULL)
   {
      printf("Stack Overflow! Memory not available.\n");
      return;
   }

   newNode->data = value;
   newNode->next = top;
   top = newNode;

   printf("Element %d pushed successfully.\n", value);
}

/* Pop Operation */
int pop()
{
   if(top == NULL)
   {
      printf("Stack Underflow! Stack is empty.\n");
      return -1;
   }

   struct Node* temp = top;
   int value = temp->data;

   top = temp->next;
   free(temp);

   return value;
}

/* Peek Operation */
int peek()
{
   if(top == NULL)
   {
```

```
        printf("Stack is Empty!\n");
        return -1;
    }

    return top->data;
}

/* Display Stack */
void display()
{
    if(top == NULL)
    {
        printf("Stack is Empty!\n");
        return;
    }

    struct Node* temp = top;
    printf("Stack elements: ");

    while(temp != NULL)
    {
        printf("%d ", temp->data);
        temp = temp->next;
    }

    printf("\n");
}
```

Literally, the stack implementation on linked list is very similar to normal linked list, but in normal linked list, the next head holds the first node and each node links to next node and in stack implementation, the top points to last node and the link pointer points to previous node.

Q. Explain stack implementation on array with code.
Q. Explain stack implementation on linked list with code.

Application of stacks

Following is the various Applications of Stack in Data Structure:

- Evaluation of Arithmetic Expressions
- Backtracking

- Delimiter Checking
- Reverse a Data
- Processing Function Calls

Evaluation of Arithmetic Expressions

A stack is a very effective data structure for evaluating arithmetic expressions in programming languages. An arithmetic expression consists of operands and operators.

Evaluation of expressions based on precedence of operators and converting the expression from infix into either pre-fix notation or post-fix notation can be very flexibly done using stacks.

Backtracking

Backtracking is another application of Stack. It is a recursive algorithm that is used for solving the optimization problem. For example the back button in a browser.

Delimiter Checking

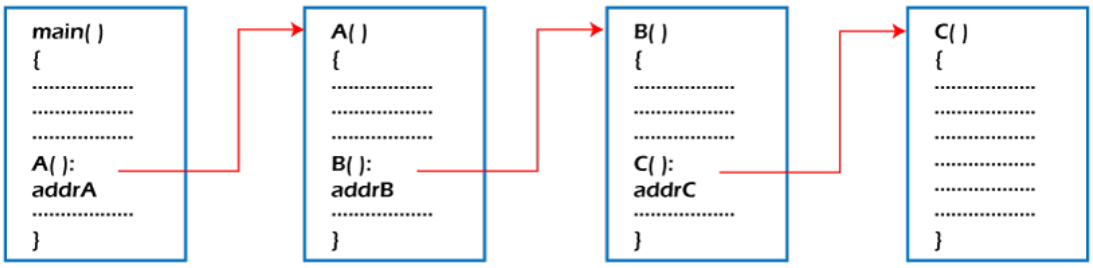
The common application of Stack is delimiter checking, i.e., parsing that involves analyzing a source program syntactically. It is also called parenthesis checking. When the compiler translates a source program written in some programming language such as C, C++ to a machine language, it parses the program into multiple individual parts such as variable names, keywords, etc. By scanning from left to right. The main problem encountered while translating is the unmatched delimiters. We make use of different types of delimiters include the parenthesis checking (,), curly braces {,} and square brackets [,], and common delimiters /* and */. Every opening delimiter must match a closing delimiter, i.e., every opening parenthesis should be followed by a matching closing parenthesis. Also, the delimiter can be nested. The opening delimiter that occurs later in the source program should be closed before those occurring earlier. To perform a delimiter checking, the compiler makes use of a stack. When a compiler translates a source program, it reads the characters one at a time, and if it finds an opening delimiter it places it on a stack. When a closing delimiter is found, it pops up the opening delimiter from the top of the Stack and matches it with the closing delimiter.

Reverse a Data

To reverse a given set of data, we need to reorder the data so that the first and last elements are exchanged, the second and second last element are exchanged, and so on for all other elements.

Processing Function Calls:

Stack plays an important role in programs that call several functions in succession. Suppose we have a program containing three functions: A, B, and C. function A invokes function B, which invokes the function C.



Function call

When we invoke function A, which contains a call to function B, then its processing will not be completed until function B has completed its execution and returned. Similarly for function B and C. So we observe that function A will only be completed after function B is completed and function B will only be completed after function C is completed. Therefore, function A is first to be started and last to be completed. To conclude, the above function activity matches the last in first out behavior and can easily be handled using Stack.

Q. Explain applications of stacks.

Polish Notations – Converting Infix to Post Fix Notation

Polish notation refers to a method of writing arithmetic expressions without using parentheses while still maintaining the correct order of operations. There are two types of Polish notation: Prefix (Polish Notation), in which the operator is written before its operands, and Postfix (Reverse Polish Notation), in which the operator is written after its operands. Infix notation is the common form in which operators are written between operands, such as A + B. However, infix

expressions require parentheses and precedence rules to remove ambiguity. In postfix notation, expressions do not require parentheses and can be evaluated easily using a stack.

The conversion of an infix expression to postfix is performed using a stack data structure. During the conversion process, operands are directly added to the postfix expression, whereas operators are pushed onto the stack according to their precedence. Operators such as exponentiation (^) have the highest precedence, followed by multiplication (*) and division (/), and then addition (+) and subtraction (-). When an operator is encountered, operators from the stack with higher or equal precedence are popped and added to the postfix expression before pushing the current operator onto the stack. Parentheses are handled separately: a left parenthesis is pushed onto the stack, and when a right parenthesis is encountered, operators are popped from the stack and added to the postfix expression until the left parenthesis is removed. After scanning the entire infix expression, any remaining operators in the stack are popped and appended to the postfix expression. The postfix notation has advantages such as eliminating the need for parentheses, simplifying evaluation, and allowing efficient computation using a stack.

Example Conversion

Convert:
A + B * C

Step-by-step:

Symbol	Stack	Postfix
A		A
+	+	A
B	+	AB
*	+ *	AB
C	+ *	ABC
End		ABC*+

Evaluation of Post Fix Notation:

Evaluation of Postfix Notation is performed using a stack data structure. In postfix (Reverse Polish) notation, the operator appears after its operands, which eliminates the need for parentheses and precedence rules during evaluation. The evaluation process involves scanning the postfix expression from left to right. If an operand is encountered, it is pushed onto the stack. If an operator is encountered, the required number of operands (usually two for binary operators) are popped from the stack. The operator is then applied to these operands in the correct order, and the result is pushed back onto the stack. This process continues until the entire expression is scanned. After the complete traversal of the postfix expression, the final result remains at the top of the stack, which represents the evaluated value of the expression.

For example, consider the postfix expression **2 3 * 5 4 * + 9 -**. First, push 2 and 3 onto the stack. When the operator * is encountered, pop 3 and 2, multiply them to obtain 6, and push 6 back onto the stack. Next, push 5 and 4. When * is encountered again, pop 4 and 5, multiply them to obtain 20, and push 20. When the + operator appears, pop 20 and 6, add them to obtain 26, and push 26. Then push 9. When the - operator is encountered, pop 9 and 26, subtract 9 from 26 to obtain 17, and push 17 onto the stack. The final value left in the stack is 17, which is the result of the postfix expression.

```
#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>   // for isdigit()

#define MAX 100

int stack[MAX];
int top = -1;

/* Push function */
void push(int value)
{
   stack[++top] = value;
}

/* Pop function */
int pop()
{
   return stack[top--];
}
```

```
/* Main function */
int main()
{
    char postfix[MAX];
    int i, operand1, operand2, result;

    printf("Enter postfix expression: ");
    scanf("%s", postfix);

    for(i = 0; postfix[i] != '\0'; i++)
    {
        /* If operand, push to stack */
        if(isdigit(postfix[i]))
        {
            push(postfix[i] - '0');   // Convert char to int
        }
        else
        {
            /* Operator encountered */
            operand2 = pop();
            operand1 = pop();

            switch(postfix[i])
            {
                case '+':
                    result = operand1 + operand2;
                    break;

                case '-':
                    result = operand1 - operand2;
                    break;

                case '*':
                    result = operand1 * operand2;
                    break;

                case '/':
                    result = operand1 / operand2;
                    break;

                default:
                    printf("Invalid operator\n");
```

```
                exit(1);
            }

            push(result);
        }
    }

    printf("Result of postfix expression = %d\n", pop());

    return 0;
}
```

Q. Explain evaluation of postfix notation
Q. Write code for evaluation of postfix notation

Queues

Introduction to Queue ADT:

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.

Analogies using concept of queues:

People waiting for a bus. The first person standing in the line will be the first one to get into the bus.

Cars lined at a toll bridge. The first car to reach the bridge will be the first to leave.

Taking a Tickets in the Cinema Theater Queue. The First Person will collect the Ticket and Enter into the Hall.

The different operations done on queue data structure are generally called qinsert() or enqueue() and qdelete() or dequeue().

Queue as an Abstract data Type:

A Queue can also be defined as ADT. A queue of elements of a particular data type is a finite (limited/ known count) sequence of elements with below specified operations:

Initialize the queue to be empty

Determine whether the queue is empty or not

Determine whether the queue is full or not

If queue is not full, then add an element / node through the rear pointer of queue. And this operation is called qinsert() / enqueu()

If queue is not empty, then remove an element from front , called qdelete() / deque().

```
procedure Enqueue(element)
    if rear = MAX-1 then      // if queue is full
    call QUEUE_FULL;
    Else                      // queue is not full
    rear=  rear + 1;          //forward rear pointer by 1.
    Q[rear]<- element;        // store element at rear end
    End If;
```

```
procedure Dequeue()
    if front=-1 and rear=-1 then      //if queue empty
    call Queue is Empty;
    else if front==rear               //if rear and front both are equal
    set rear =-1 and front=-1
    else                              //if queue is not empty
    front=front+1;                    //forward front pointer by 1
    End if;
```

** but a simple queue has a problem that unless it is fully emptied, it cannot be used fully.

Q. Give an introduction to queues.

Implementation of Queues using array and Linked List,

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

#define MAX 5
```

```
int queue[MAX];
int front = -1, rear = -1;

/* Enqueue */
void enqueue(int value)
{
   if (rear == MAX - 1)
   {
      printf("Queue Overflow\n");
      return;
   }

   if (front == -1)
      front = 0;

   rear++;
   queue[rear] = value;
}

/* Dequeue */
int dequeue()
{
   if (front == -1 || front > rear)
   {
      printf("Queue Underflow\n");
      return -1;
   }

   int value = queue[front];
   front++;

   return value;
}

/* Display */
void display()
{
   if (front == -1 || front > rear)
   {
      printf("Queue is Empty\n");
      return;
   }
```

```
    printf("Queue elements: ");
    for (int i = front; i <= rear; i++)
        printf("%d ", queue[i]);

    printf("\n");
}

int main()
{
    enqueue(10);
    enqueue(20);
    enqueue(30);

    display();

    printf("Deleted element: %d\n", dequeue());

    display();

    return 0;
}
```

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

/* Define Node */
struct Node
{
    int data;
    struct Node* next;
};

struct Node* front = NULL;
struct Node* rear = NULL;

/* Enqueue */
void enqueue(int value)
{
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));

    if (newNode == NULL)
```

```
    {
        printf("Memory Overflow\n");
        return;
    }

    newNode->data = value;
    newNode->next = NULL;

    if (rear == NULL)
    {
        front = rear = newNode;
    }
    else
    {
        rear->next = newNode;
        rear = newNode;
    }
}

/* Dequeue */
int dequeue()
{
    if (front == NULL)
    {
        printf("Queue Underflow\n");
        return -1;
    }

    struct Node* temp = front;
    int value = temp->data;

    front = front->next;

    if (front == NULL)
        rear = NULL;

    free(temp);

    return value;
}

/* Display */
void display()
```

```
{
    if (front == NULL)
    {
        printf("Queue is Empty\n");
        return;
    }

    struct Node* temp = front;
    printf("Queue elements: ");

    while (temp != NULL)
    {
        printf("%d ", temp->data);
        temp = temp->next;
    }

    printf("\n");
}

int main()
{
    enqueue(10);
    enqueue(20);
    enqueue(30);

    display();

    printf("Deleted element: %d\n", dequeue());

    display();

    return 0;
}
```

Array Queue	Linked List Queue
Fixed size	Dynamic size
May cause overflow	Overflow only if memory full
Uses front & rear indices	Uses front & rear pointers
Possible wasted space	Efficient memory use

Application of Queues

A queue is a linear data structure that follows the **First-In, First-Out (FIFO)** principle. The element inserted first is removed first. Queues are widely used in computer science and real-life systems where tasks must be processed in the order they arrive.

One of the most important applications of a queue is in **CPU scheduling** in operating systems. Processes waiting for CPU execution are placed in a ready queue, and the scheduler selects them in FIFO order. Queues are also used in **printer spooling**, where print jobs are stored in a queue and printed one by one in the order they are received.

Another major application is in **I/O buffering**. When data is transferred between devices operating at different speeds, a queue acts as a buffer to temporarily store data until it is processed. Queues are also used in **keyboard input buffering**, where characters typed by the user are stored before being processed by the system.

In networking, queues are used in **packet scheduling**. Data packets arriving at a router are stored in a queue and transmitted in order. In simulation systems, queues are used to model real-world waiting lines such as ticket counters, bank counters, and customer service centers.

Queues are also important in algorithms. They are used in **Breadth-First Search (BFS)** traversal of graphs and trees. BFS explores nodes level by level using a queue. Additionally, queues are used in interrupt handling systems, task scheduling, call center systems, and real-time systems where requests must be processed in the order they arrive.

Thus, queues play a vital role in operating systems, networking, simulations, and algorithm design wherever ordered processing is required.

Q. Write about applications of queues.

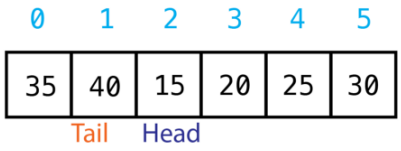
Circular Queues:

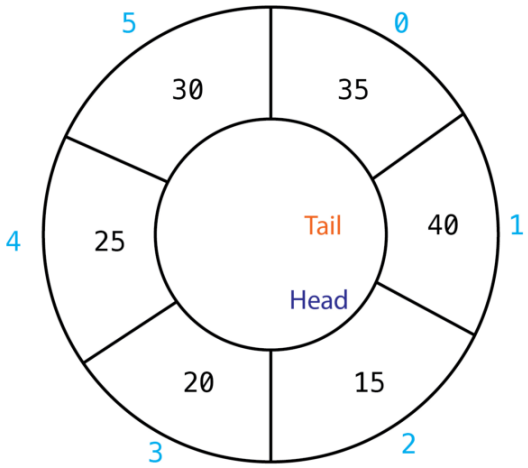
A simple queue when implemented, the logic or algorithm is such that, unless the queue is fully emptied, it cannot be used fully. For example if a queue has five elements and 4 elements are stored in that and then 3 are

removed, though there are 4 empty places in the queue, the algorithm / logic permits only 1 values to be enqueued.

To make use of the simple queue fully, each time the dequeue (delete queue) is done, all the elements are to be moved to its previous locations. This "moving elements" each time to its previous locations give burden on system.

To overcome this problem, the circular queues are implemented. In this the logic is implemented such that, the rear pointer moves to first location if first is empty so that the queue is logically formed as a circle.



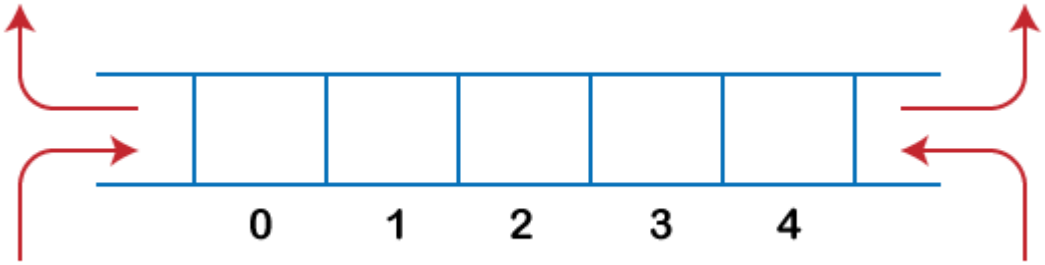



```
Q = circularQueue(6)
Q.Enqueue(5)
Q.Enqueue(10)
Q.Enqueue(15)
Q.Enqueue(20)
Q.Enqueue(25)
Q.Enqueue(30)
Q.Dequeue()
Q.Dequeue()
Q.Enqueue(35)
Q.Enqueue(40)
```

Q. Explain circular queues.

De-queues:

In the queue, the insertion takes place from one end while the deletion takes place from another end. The end at which the insertion occurs is known as the rear end whereas the end at which the deletion occurs is known as front end. The dequeue stands for Double Ended Queue.



Deque is a linear data structure in which the insertion and deletion operations are performed from both ends. We can say that deque is a generalized version of the queue.

In DEQUEUEs, the insert can be done either from front or from rear and also the deleting can be done either from front or rear.

But the DEQUEUE operation is not considered as a standard QUEUE operation, as the dequeues break the basic FIFO rule of queues.

The following are the six functions that we have used in the below program:

- enqueue_front(): It is used to insert the element from the front end.
- enqueue_rear(): It is used to insert the element from the rear end.
- dequeue_front(): It is used to delete the element from the front end.
- dequeue_rear(): It is used to delete the element from the rear end.
- getfront(): It is used to return the front element of the deque.
- getrear(): It is used to return the rear element of the deque.

Q. Write about dequeues.

Priority Queue and Heap :

A priority queue is an abstract data type that behaves similarly to the normal queue except that each element has some priority, i.e., the element with the highest priority would come first in a priority queue. The priority of the elements in a priority queue will determine the order in which elements are removed from the priority queue.

The priority queue supports only comparable elements, which means that the elements are either arranged in an ascending or descending order.

For example, suppose we have some values like 1, 3, 4, 8, 14, 22 inserted in a priority queue with an ordering imposed on the values is from least to the

greatest. Therefore, the 1 number would be having the highest priority while 22 will be having the lowest priority.

The characteristics are :

- Every element in a priority queue has some priority associated with it.
- An element with the higher priority will be deleted before the deletion of the lesser priority.
- If two elements in a priority queue have the same priority, they will be arranged using the FIFO principle.

If the values of the priority queue are 1, 3, 4, 8, 14, 22

All the values are arranged in ascending order. Now, we will observe how the priority queue will look after performing the following operations:

- poll(): This function will remove the highest priority element from the priority queue. In the above priority queue, the '1' element has the highest priority, so it will be removed from the priority queue.
- add(2): This function will insert '2' element in a priority queue. As 2 is the smallest element among all the numbers so it will obtain the highest priority.
- poll(): It will remove '2' element from the priority queue as it has the highest priority queue.
- add(5): It will insert 5 element after 4 as 5 is larger than 4 and lesser than 8, so it will obtain the third highest priority in a priority queue.

Implementation of Priority Queue

The priority queue can be implemented in four ways that include arrays, linked list, heap data structure and binary search tree. The heap data structure is the most efficient way of implementing the priority queue, so we will implement the priority queue using a heap data structure in this topic. Now, first we understand the reason why heap is the most efficient way among all the other data structures.

Q. Write about priority queues.

Applications of priority queues:

When the graph is stored in the form of adjacency list or matrix, priority queue can be used to extract minimum efficiently when implementing Dijkstra’s algorithm.

Prim’s algorithm: It is used to implement Prim’s Algorithm to store keys of nodes and extract minimum key node at every step.

Data compression : It is used in Huffman codes which is used to compresses data.
Artificial Intelligence : A* Search Algorithm : The A* search algorithm finds the shortest path between two vertices of a weighted graph, trying out the most promising routes first. The priority queue (also known as the fringe) is used to keep track of unexplored routes, the one for which a lower bound on the total path length is smallest is given highest priority.

Heap Sort : Heap sort is typically implemented using Heap which is an implementation of Priority Queue.

Operating systems: It is also use in Operating System for load balancing (load balancing on server), interrupt handling.

Q. Write about applications of priority queues.
