BATU-EXAM

Made by batuexams.com

at MET Bhujbal Knowledege City

Data Structures Department

The PDF notes on this website are the copyrighted property of batuexams.com.

All rights reserved.

DOWNLOADED FROM BATU-EXAMS.in

UNIT – II

Stack and Queue

Stacks and Queues

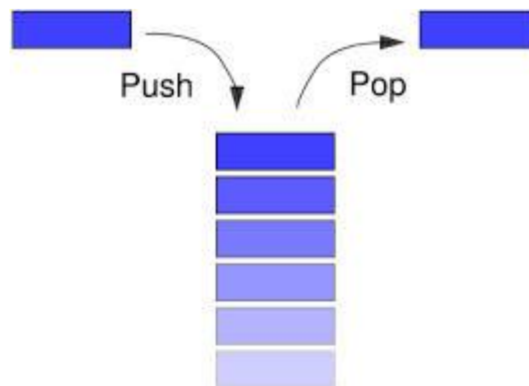
Stack is a container of objects that are inserted and removed according to the last-in first-out (LIFO) principle.

Queue is a container of objects (a linear collection) that are inserted and removed according to the first-in first-out (FIFO) principle.

Introduction

Stack:

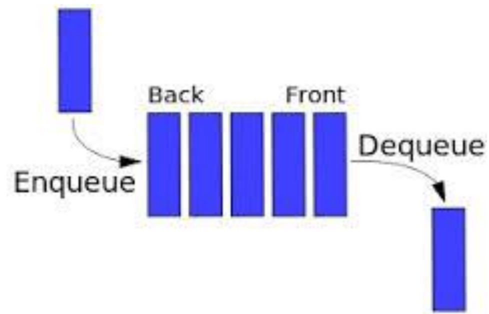
In the pushdown stacks only two operations are allowed: push the item into the stack and pop the item out of the stack. A stack is a limited access data structure - elements can be added and removed from the stack only at the top. push adds an item to the top of the stack, pop removes the item from the top. A helpful analogy is to think of a stack of books; you can remove only the top book, also you can add a new book on the top.



Queue:

An excellent example of a queue is a line of students in the food court of the UC. New additions to a line made to the back of the queue, while removal (or serving) happens in the front. In the queue only two operations are allowed enqueue and dequeue. Enqueue means to insert an item into the back of the queue, dequeue means removing the front item. The picture demonstrates the FIFO access. 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.

DOWNLOADED FROM BATU-EXAMS.in



FIFO & LILO and LIFO & FILO Principles

Queue: First In First Out (FIFO): The first object into a queue is the first object to leave the queue, used by a queue.

Stack: Last In First Out (LIFO): The last object into a stack is the first object to leave the stack, used by a stack

OR

Stack: First In Last Out (FILO): The first object or item in a stack is the last object or item to leave the stack.

Queue: Last In Last Out (LILO): The last object or item in a queue is the last object or item to leave the queue.

Stack C Code:

```
// Written in COP 3502 to illustrate an array implementation of a stack.
#include <stdio.h>
// The array will store the items in the stack, first in
// index 0, then 1, etc. top will represent the index
// to the top element in the stack. If the stack is
// empty top will be -1.

#define SIZE 10
#define EMPTY -1

struct stack {

    int items[SIZE];
    int top;
};
```

DOWNLOADED FROM BATU-EXAMS.in

```
void initialize(struct stack* stackPtr);
int full(struct stack* stackPtr);
int push(struct stack* stackPtr, int value);
int empty(struct stack* stackPtr);
int pop(struct stack* stackPtr);
int top(struct stack* stackPtr);

int main() {
    int i;
    struct stack mine;

    // Set up the stack and push a couple items, then pop one.
    initialize(&mine);
    push(&mine, 4);
    push(&mine, 5);
    printf("Popping %d\n", pop(&mine));

    // Push a couple more and test top.
    push(&mine, 22);
    push(&mine, 16);
    printf("At top now = %d\n", top(&mine));

    // Pop all three off.
    printf("Popping %d\n", pop(&mine));
    printf("Popping %d\n", pop(&mine));
    printf("Popping %d\n", pop(&mine));

    // Checking the empty function.
    if (empty(&mine))
        printf("The stack is empty as expected.\n");

    // Fill the stack.
    for (i = 0; i<10; i++)
        push(&mine, i);
```

DOWNLOADED FROM BATU-EXAMS.in

```
    // Check if full works.
    if (full(&mine))
        printf("This stack is full as expected.\n");

    // Pop everything back off.
    for (i = 0; i<10; i++)
        printf("popping %d\n",pop(&mine));

    system("PAUSE");
    return 0;
}

void initialize(struct stack* stackPtr) {
     stackPtr->top = -1;
}

// If the push occurs, 1 is returned. If the
// stack is full and the push can't be done, 0 is
// returned.
int push(struct stack* stackPtr, int value) {

    // Check if the stack is full.
    if (full(stackPtr))
        return 0;

    // Add value to the top of the stack and adjust the value of the top.
    stackPtr->items[stackPtr->top+1] = value;
    (stackPtr->top)++;
    return 1;
}

// Returns true iff the stack pointed to by stackPtr is full.
int full(struct stack* stackPtr) {
    return (stackPtr->top == SIZE - 1);
}
```

DOWNLOADED FROM BATU-EXAMS.in

```
// Returns true iff the stack pointed to by stackPtr is empty.
int empty(struct stack* stackPtr) {
    return (stackPtr->top == -1);
}

// Pre-condition: The stack pointed to by stackPtr is NOT empty.
// Post-condition: The value on the top of the stack is popped and returned.
// Note: If the stack pointed to by stackPtr is empty, -1 is returned.
int pop(struct stack* stackPtr) {

    int retval;

    // Check the case that the stack is empty.
    if (empty(stackPtr))
        return EMPTY;
    // Retrieve the item from the top of the stack, adjust the top and return
    // the item.
    retval = stackPtr->items[stackPtr->top];
    (stackPtr->top)--;
    return retval;
}

// Pre-condition: The stack pointed to by stackPtr is NOT empty.
// Post-condition: The value on the top of the stack is returned.
// Note: If the stack pointed to by stackPtr is empty, -1 is returned.
int top(struct stack* stackPtr) {

    // Take care of the empty case.
    if (empty(stackPtr))
        return EMPTY;

    // Return the desired item.
    return stackPtr->items[stackPtr->top];
}
```

DOWNLOADED FROM BATU-EXAMS.in

Queue C Code:

```
// Example of how to implement a queue with an array.
#include <stdio.h>

#define EMPTY -1
#define INIT_SIZE 10

// Stores our queue.
struct queue {
    int* elements;
    int front;
    int numElements;
    int queueSize;
};

void init(struct queue* qPtr);
int enqueue(struct queue* qPtr, int val);
int dequeue(struct queue* qPtr);
int empty(struct queue* qPtr);
int front(struct queue* qPtr);

int main() {

     int i;

    // Allocate space for our queue and initialize it.
    struct queue* MyQueuePtr = (struct queue*)malloc(sizeof(struct queue));
    init(MyQueuePtr);

    // Enqueue some items.
    enqueue(MyQueuePtr, 3);
    enqueue(MyQueuePtr, 7);
    enqueue(MyQueuePtr, 4);

    // Try one dequeue.
```

DOWNLOADED FROM BATU-EXAMS.in

```
    printf("Dequeue %d\n", dequeue(MyQueuePtr));

    // Enqueue one more item, then try several dequeues and one front.
    enqueue(MyQueuePtr, 2);
    printf("Dequeue %d\n", dequeue(MyQueuePtr));
    printf("At Front of Queue Now: %d\n", front(MyQueuePtr));
    printf("Dequeue %d\n", dequeue(MyQueuePtr));
    printf("Dequeue %d\n", dequeue(MyQueuePtr));

    // See if we can detect an empty queue.
    printf("Is empty: %d\n", empty(MyQueuePtr));

    // Try enqueuing and dequeuing again to make sure that our previous
    // operations didn't "corrupt" the queue.
    enqueue(MyQueuePtr, 5);
    enqueue(MyQueuePtr, 9);

    // Try lots of enqueues to test the dynamic capability of the queue.
    for (i=30; i<60; i++)
        enqueue(MyQueuePtr, i);

    // Dequeue everything.
    while (!empty(MyQueuePtr))
        printf("Dequeue %d\n", dequeue(MyQueuePtr));

    return 0;
}

// Pre-condition: qPtr points to a valid struct queue.
// Post-condition: The struct that qPtr points to will be set up to represent an
//                 empty queue.
void init(struct queue* qPtr) {

     // The front index is 0, as is the number of elements.
     qPtr->elements = (int*)malloc(sizeof(int)*INIT_SIZE);
     qPtr->front = 0;
```

DOWNLOADED FROM BATU-EXAMS.in

```
     qPtr->numElements = 0;
     qPtr->queueSize = INIT_SIZE;
}

// Pre-condition: qPtr points to a valid struct queue and val is the value to
//                enqueue into the queue pointed to by qPtr.
// Post-condition: If the operation is successful, 1 will be returned, otherwise
//                 no change will be made to the queue and 0 will be returned.

// Note: Right now, I don't know how to detect that the realloc failed, so 0
//       does not get returned.

int enqueue(struct queue* qPtr, int val) {

    int i;

    // Regular case where our queue isn't full.
    if (qPtr->numElements != qPtr->queueSize) {

        // Enqueue the current element. Note the use of mod for the wraparound
        // case. Edit the number of elements.
        qPtr->elements[(qPtr->front+qPtr->numElements)%qPtr->queueSize] = val;
        (qPtr->numElements)++;

        // Signifies a successful operation.
        return 1;
    }

    // Case where the queue is full, we must find more space before we enqueue.
    else {

         // Allocate more space!
         qPtr->elements = realloc(qPtr->elements, (qPtr->queueSize)*sizeof(int)*2)
;

         // Copy all of the items that are stored "before" front and copy them
         // into their new correct locations.
```

DOWNLOADED FROM BATU-EXAMS.in

```
        for (i=0; i<=qPtr->front-1; i++)
            qPtr->elements[i+qPtr->queueSize] = qPtr->elements[i];

        // Enqueue the new item, now that there is space. We are guaranteed that
        // no wraparound is necessary here.
        qPtr->elements[i+qPtr->queueSize] = val;

        // More bookkeeping: The size of the queue as doubled and the number of
        // elements has increased by one.
        (qPtr->queueSize) *= 2;
        (qPtr->numElements)++;

        return 1;
    }

}

// Pre-condition: qPtr points to a valid struct queue.
// Post-condition: If the queue pointed to by qPtr is non-empty, then the value
//                 at the front of the queue is deleted from the queue and
//                 returned. Otherwise, -1 is returned to signify that the queue
//                 was already empty when the dequeue was attempted.
int dequeue(struct queue* qPtr) {

    int retval;

    // Empty case.
    if (qPtr->numElements == 0)
        return EMPTY;

    // Store the value that should be returned.
    retval = qPtr->elements[qPtr->front];

    // Adjust the index to the front of the queue accordingly.
    qPtr->front = (qPtr->front + 1)% qPtr->queueSize;

```

DOWNLOADED FROM BATU-EXAMS.in

```
    // We have one fewer element in the queue.
    (qPtr->numElements)--;

    // Return the dequeued element.
    return retval;
}

// Pre-condition: qPtr points to a valid struct queue.
// Post-condition: returns true iff the queue pointed to by pPtr is empty.
int empty(struct queue* qPtr) {
    return qPtr->numElements == 0;
}

// Pre-condition: qPtr points to a valid struct queue.
// Post-condition: if the queue pointed to by qPtr is non-empty, the value
//                 stored at the front of the queue is returned. Otherwise,
//                 -1 is returned to signify that this queue is empty.
int front(struct queue* qPtr) {
    if (qPtr->numElements != 0)
        return qPtr->elements[qPtr->front];
    else
        return EMPTY;
}
```

Stack ADT:

A stack is an Abstract Data Type (ADT), commonly used in most programming languages. It is named stack as it behaves like a real-world stack, for example – a deck of cards or a pile of plates, etc.

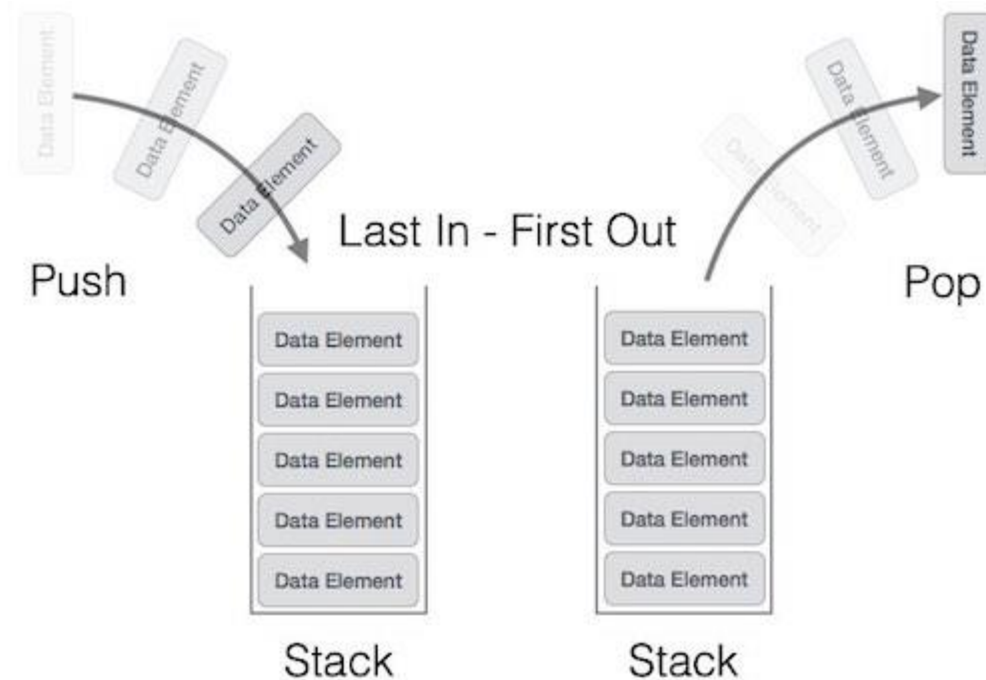
A real-world stack allows operations at one end only. For example, we can place or remove a card or plate from the top of the stack only. Likewise, Stack ADT allows all data operations at one end only. At any given time, we can only access the top element of a stack.

DOWNLOADED FROM BATU-EXAMS.in

This feature makes it LIFO data structure. LIFO stands for Last-in-first-out. Here, the element which is placed (inserted or added) last, is accessed first. In stack terminology, insertion operation is called **PUSH** operation and removal operation is called **POP** operation.

Stack Representation

The following diagram depicts a stack and its operations −



A stack can be implemented by means of Array, Structure, Pointer, and Linked List. Stack can either be a fixed size one or it may have a sense of dynamic resizing. Here, we are going to implement stack using arrays, which makes it a fixed size stack implementation.

Basic Operations

Stack operations may involve initializing the stack, using it and then de-initializing it. Apart from these basic stuffs, a stack is used for the following two primary operations −

- **push()** − Pushing (storing) an element on the stack.
- **pop()** − Removing (accessing) an element from the stack.

When data is PUSHed onto stack.

To use a stack efficiently, we need to check the status of stack as well. For the same purpose, the following functionality is added to stacks −

- **peek()** − get the top data element of the stack, without removing it.
- **isFull()** − check if stack is full.
- **isEmpty()** − check if stack is empty.

At all times, we maintain a pointer to the last PUSHed data on the stack. As this pointer always represents the top of the stack, hence named **top**. The **top** pointer provides top value of the stack without removing it.

DOWNLOADED FROM BATU-EXAMS.in

First, we should learn about procedures to support stack functions –

peek()

Algorithm of peek() function −

```
begin procedure peek
   return stack[top]
end procedure
```

Implementation of peek() function in C programming language −

Example

```
int peek() {
   return stack[top];
}
```

isfull()

Algorithm of isfull() function −

```
begin procedure isfull

   if top equals to MAXSIZE
      return true
   else
      return false
   endif

end procedure
```

Implementation of isfull() function in C programming language −

Example

```
bool isfull() {
   if(top == MAXSIZE)
      return true;
   else
      return false;
}
```

isempty()

Algorithm of isempty() function −

DOWNLOADED FROM BATU-EXAMS.in

```
begin procedure isempty

  if top less than 1
    return true
  else
    return false
  endif

end procedure
```

Implementation of isempty() function in C programming language is slightly different. We initialize top at -1, as the index in array starts from 0. So we check if the top is below zero or -1 to determine if the stack is empty. Here's the code −

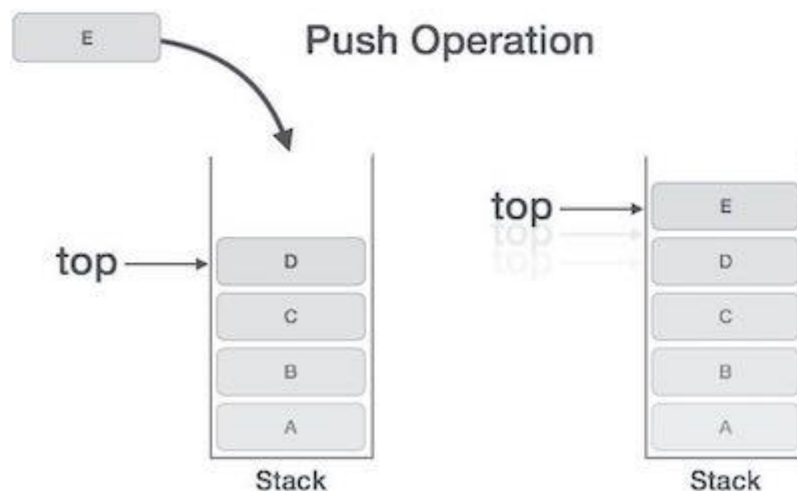
Example

```
bool isempty() {
  if(top == -1)
    return true;
  else
    return false;
}
```

Push Operation

The process of putting a new data element onto stack is known as a Push Operation. Push operation involves a series of steps −

- **Step 1** − Checks if the stack is full.
- **Step 2** − If the stack is full, produces an error and exit.
- **Step 3** − If the stack is not full, increments **top** to point next empty space.
- **Step 4** − Adds data element to the stack location, where top is pointing.
- **Step 5** − Returns success.



DOWNLOADED FROM BATU-EXAMS.in

If the linked list is used to implement the stack, then in step 3, we need to allocate space dynamically.

Algorithm for PUSH Operation

A simple algorithm for Push operation can be derived as follows −

```
begin procedure push: stack, data

   if stack is full
      return null
   endif

   top ← top + 1
   stack[top] ← data

end procedure
```

Implementation of this algorithm in C, is very easy. See the following code −

Example

```
void push(int data) {
   if(!isFull()) {
      top = top + 1;
      stack[top] = data;
   } else {
      printf("Could not insert data, Stack is full.\n");
   }
}
```

Pop Operation

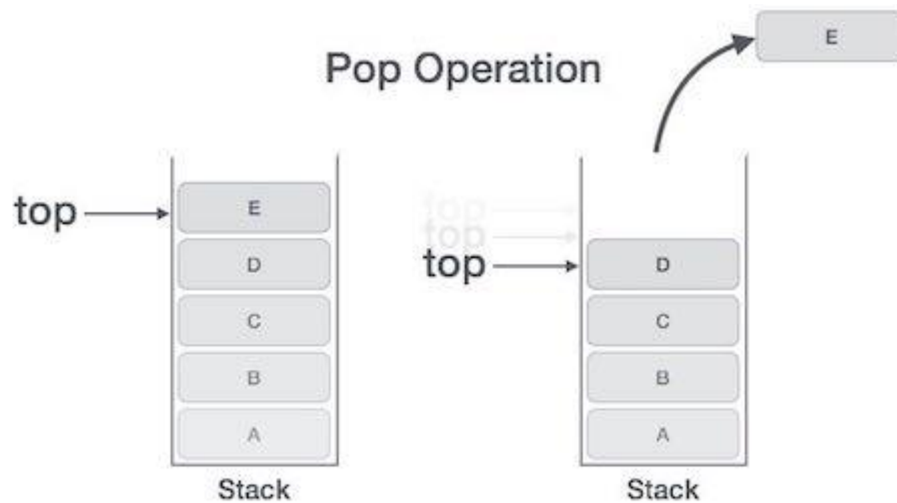
Accessing the content while removing it from the stack, is known as a Pop Operation. In an array implementation of pop() operation, the data element is not actually removed, instead **top** is decremented to a lower position in the stack to point to the next value. But in linked-list implementation, pop() actually removes data element and deallocates memory space.

A Pop operation may involve the following steps −

- **Step 1** − Checks if the stack is empty.
- **Step 2** − If the stack is empty, produces an error and exit.
- **Step 3** − If the stack is not empty, accesses the data element at which **top** is pointing.

DOWNLOADED FROM BATU-EXAMS.in

- **Step 4** − Decreases the value of top by 1.
- **Step 5** − Returns success.



Algorithm for Pop Operation

A simple algorithm for Pop operation can be derived as follows −

```
begin procedure pop: stack

  if stack is empty
    return null
  endif

  data ← stack[top]
  top ← top - 1
  return data

end procedure
```

Implementation of this algorithm in C, is as follows −

Example

```
int pop(int data) {


  if(!isempty()) {

    data = stack[top];

    top = top - 1;

    return data;

  } else {

    printf("Could not retrieve data, Stack is empty.\n");

  }

}
```

DOWNLOADED FROM BATU-EXAMS.in

Queue ADT:

Queue is an abstract data structure, somewhat like Stacks. Unlike stacks, a queue is open at both its ends. One end is always used to insert data (enqueue) and the other is used to remove data (dequeue). Queue follows First-In-First-Out methodology, i.e., the data item stored first will be accessed first.

A real-world example of queue can be a single-lane one-way road, where the vehicle enters first, exits first. More real-world examples can be seen as queues at the ticket windows and bus-stops.

Queue Representation

As we now understand that in queue, we access both ends for different reasons. The following diagram given below tries to explain queue representation as data structure −

As in stacks, a queue can also be implemented using Arrays, Linked-lists, Pointers and Structures. For the sake of simplicity, we shall implement queues using one-dimensional array.

Basic Operations

Queue operations may involve initializing or defining the queue, utilizing it, and then completely erasing it from the memory. Here we shall try to understand the basic operations associated with queues −

- **enqueue()** − add (store) an item to the queue.
- **dequeue()** − remove (access) an item from the queue.

Few more functions are required to make the above-mentioned queue operation efficient. These are −

- **peek()** − Gets the element at the front of the queue without removing it.
- **isfull()** − Checks if the queue is full.
- **isempty()** − Checks if the queue is empty.

DOWNLOADED FROM BATU-EXAMS.in

In queue, we always dequeue (or access) data, pointed by **front** pointer and while enqueing (or storing) data in the queue we take help of **rear** pointer.

Let's first learn about supportive functions of a queue −

peek()

This function helps to see the data at the **front** of the queue. The algorithm of peek() function is as follows −

Algorithm

```
begin procedure peek

   return queue[front]

end procedure
```

Implementation of peek() function in C programming language −

Example

```
int peek() {
   return queue[front];
}
```

isfull()

As we are using single dimension array to implement queue, we just check for the rear pointer to reach at MAXSIZE to determine that the queue is full. In case we maintain the queue in a circular linked-list, the algorithm will differ. Algorithm of isfull() function −

Algorithm

```
begin procedure isfull

   if rear equals to MAXSIZE
      return true
   else
      return false
   endif

end procedure
```

Implementation of isfull() function in C programming language −

Example

```
bool isfull() {
   if(rear == MAXSIZE - 1)
      return true;
   else
      return false;
}
```

DOWNLOADED FROM BATU-EXAMS.in

isempty()

Algorithm of isempty() function −

Algorithm

```
begin procedure isempty

  if front is less than MIN  OR front is greater than rear
    return true
  else
    return false
  endif

end procedure
```

If the value of **front** is less than MIN or 0, it tells that the queue is not yet initialized, hence empty.

Here's the C programming code −

Example

```
bool isempty() {
  if(front < 0 || front > rear)
    return true;
  else
    return false;
}
```

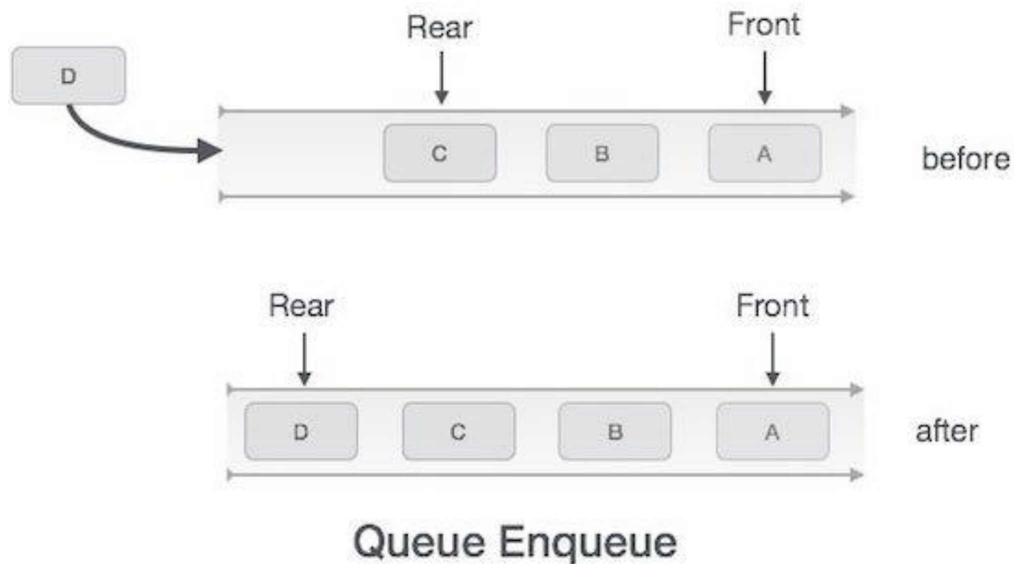
Enqueue Operation

Queues maintain two data pointers, **front** and **rear**. Therefore, its operations are comparatively difficult to implement than that of stacks.

The following steps should be taken to enqueue (insert) data into a queue −

- **Step 1** − Check if the queue is full.
- **Step 2** − If the queue is full, produce overflow error and exit.
- **Step 3** − If the queue is not full, increment **rear** pointer to point the next empty space.
- **Step 4** − Add data element to the queue location, where the rear is pointing.
- **Step 5** − return success.

DOWNLOADED FROM BATU-EXAMS.in



Sometimes, we also check to see if a queue is initialized or not, to handle any unforeseen situations.

Algorithm for enqueue operation

```
procedure enqueue(data)


  if queue is full
    return overflow
  endif


  rear ← rear + 1
  queue[rear] ← data
  return true


end procedure
```

Implementation of enqueue() in C programming language −

Example

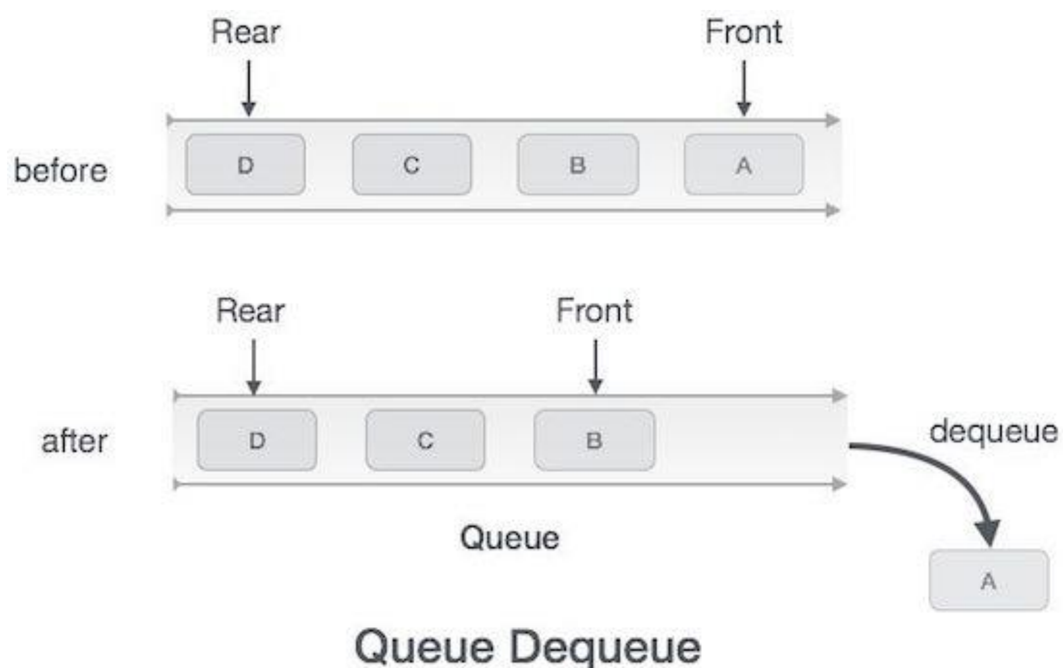
```
int enqueue(int data)
  if(isfull())
    return 0;


  rear = rear + 1;
  queue[rear] = data;
  return 1;
end procedure
```

DOWNLOADED FROM BATU-EXAMS.in

Dequeue Operation

Accessing data from the queue is a process of two tasks − access the data where **front** is pointing and remove the data after access. The following steps are taken to perform **dequeue** operation −

- **Step 1** − Check if the queue is empty.
- **Step 2** − If the queue is empty, produce underflow error and exit.
- **Step 3** − If the queue is not empty, access the data where **front** is pointing.
- **Step 4** − Increment **front** pointer to point to the next available data element.
- **Step 5** − Return success.



Algorithm for dequeue operation

```
procedure dequeue

  if queue is empty
    return underflow
  end if

  data = queue[front]
  front ← front + 1
  return true

end procedure
```

DOWNLOADED FROM BATU-EXAMS.in

Implementation of dequeue() in C programming language −

Example

```
int dequeue() {
  if(isempty())
    return 0;

  int data = queue[front];
  front = front + 1;

  return data;
}
```

Expression Parsing

The way to write arithmetic expression is known as a **notation**. An arithmetic expression can be written in three different but equivalent notations, i.e., without changing the essence or output of an expression. These notations are −

- Infix Notation
- Prefix (Polish) Notation
- Postfix (Reverse-Polish) Notation

These notations are named as how they use operator in expression. We shall learn the same here in this chapter.

Infix Notation

We write expression in **infix** notation, e.g. a - b + c, where operators are used **in**-between operands. It is easy for us humans to read, write, and speak in infix notation but the same does not go well with computing devices. An algorithm to process infix notation could be difficult and costly in terms of time and space consumption.

Prefix Notation

In this notation, operator is **prefix**ed to operands, i.e. operator is written ahead of operands. For example, +**ab**. This is equivalent to its infix notation **a** + **b**. Prefix notation is also known as **Polish Notation**.

Postfix Notation

This notation style is known as **Reversed Polish Notation**. In this notation style, the operator is **postfix**ed to the operands i.e., the operator is written after the operands. For example, **ab**+. This is equivalent to its infix notation **a** + **b**.

The following table briefly tries to show the difference in all three notations −

DOWNLOADED FROM BATU-EXAMS.in

Sr.No.	Infix Notation	Prefix Notation	Postfix Notation
1	a + b	+ a b	a b +
2	(a + b) ∗ c	∗ + a b c	a b + c ∗
3	a ∗ (b + c)	∗ a + b c	a b c + ∗
4	a / b + c / d	+ / a b / c d	a b / c d / +
5	(a + b) ∗ (c + d)	∗ + a b + c d	a b + c d + ∗
6	((a + b) ∗ c) - d	- ∗ + a b c d	a b + c ∗ d -

Parsing Expressions

As we have discussed, it is not a very efficient way to design an algorithm or program to parse infix notations. Instead, these infix notations are first converted into either postfix or prefix notations and then computed.

To parse any arithmetic expression, we need to take care of operator precedence and associativity also.

Precedence

When an operand is in between two different operators, which operator will take the operand first, is decided by the precedence of an operator over others. For example −

a + b * c ⇨ a + (b * c)

As multiplication operation has precedence over addition, b * c will be evaluated first. A table of operator precedence is provided later.

Associativity

Associativity describes the rule where operators with the same precedence appear in an expression. For example, in expression a + b − c, both + and – have the same precedence, then which part of the expression will be evaluated first, is determined by associativity of those operators. Here, both + and − are left associative, so the expression will be evaluated as **(a + b) − c**.

Precedence and associativity determine the order of evaluation of an expression. Following is an operator precedence and associativity table (highest to lowest) −

Sr.No.	Operator	Precedence	Associativity
1	Exponentiation ^	Highest	Right Associative
2	Multiplication (∗) & Division (/)	Second Highest	Left Associative
3	Addition (+) & Subtraction (−)	Lowest	Left Associative

DOWNLOADED FROM BATU-EXAMS.in

The above table shows the default behavior of operators. At any point of time in expression evaluation, the order can be altered by using parenthesis. For example −

In **a + b*c**, the expression part **b*c** will be evaluated first, with multiplication as precedence over addition. We here use parenthesis for **a + b** to be evaluated first, like **(a + b)*c**.

Postfix Evaluation Algorithm

We shall now look at the algorithm on how to evaluate postfix notation −

Step 1 − scan the expression from left to right.

Step 2 − if it is an operand push it to stack.

Step 3 − if it is an operator pull operand from stack and perform operation.

Step 4 − store the output of step 3, back to stack.

Step 5 − scan the expression until all operands are consumed.

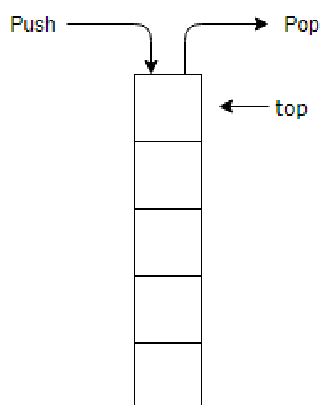
Step 6 − pop the stack and perform operation.

Implementation Of Stack and Queue Using Linked List

we will be discussing two data structures - Stack and Queue. We will discuss various I/O operations on these data structures and their implementation using another data structure, i.e., Linked List.

What is Stack?

A Stack is a linear data structure which allows adding and removing of elements in a particular order. New elements are added at the top of Stack. If we want to remove an element from the Stack, we can only remove the top element from Stack. Since it allows insertion and deletion from only one end and the element to be inserted last will be the element to be deleted first, hence it is called Last in First Out data structure (LIFO).



Stack Data Structure

DOWNLOADED FROM BATU-EXAMS.in

Here we will define three operations on Stack,

- Push - it specifies adding an element to the Stack. If we try to insert an element when the Stack is full, then it is said to be Stack Overflow condition.
- Pop - it specifies removing an element from the Stack. Elements are always removed from top of Stack. If we try to perform pop operation on an empty Stack, then it is said to be Stack Underflow condition.
- Peek - it will show the element on the top of Stack (without removing it).

Implementing Stack functionalities using Linked List

Stack can be implemented using both, arrays and linked list. The limitation in case of array is that we need to define the size at the beginning of the implementation. This makes our Stack static. It can also result in *"Stack overflow"* if we try to add elements after the array is full. So, to alleviate this problem, we use linked list to implement the Stack so that it can grow in real time.

First, we will create our Node class which will form our Linked List. We will be using this same Node class to implement the Queue also in the later part of this article.

```
1. internal class Node
2. {
3.     internal int data;
4.     internal Node next;
5.
6.     // Constructor to create a new node.Next is by default
   initialized as null
7.     public Node(int d)
8.     {
9.         data = d;
10.          next = null;
11.     }
12. }
```

Now, we will create our Stack Class. We will define a pointer, top, and initialize it to null. So, our LinkedListStack class will be –

```
1. internal class LinkListStack
2. {
3.     Node top;
4.
5.     public LinkListStack()
6.     {
7.         this.top = null;
8.     }
9.  }
```

DOWNLOADED FROM BATU-EXAMS.in

Push an element into Stack

Now, our Stack and Node class is ready. So, we will proceed to Push operation on Stack. We will add a new element at the top of Stack.

Algorithm

- Create a new node with the value to be inserted.
- If the Stack is empty, set the next of the new node to null.
- If the Stack is not empty, set the next of the new node to top.
- Finally, increment the top to point to the new node.

The time complexity for *Push* operation is O(1). The method for Push will look like this.

```
1. internal void Push(int value)
2. {
3.     Node newNode = new Node(value);
4.     if (top == null)
5.     {
6.         newNode.next = null;
7.     }
8.     else
9.     {
10.         newNode.next = top;
11.     }
12.     top = newNode;
13.     Console.WriteLine("{0} pushed to stack", value);
14. }
```

Pop an element from Stack

We will remove the top element from Stack.

Algorithm

- If the Stack is empty, terminate the method as it is Stack underflow.
- If the Stack is not empty, increment top to point to the next node.
- Hence the element pointed by top earlier is now removed.

The time complexity for Pop operation is O(1). The method for Pop will be like following.

```
1. internal void Pop()
2. {
3.     if (top == null)
4.     {
5.         Console.WriteLine("Stack Underflow. Deletion not p
ossible");
6.         return;
7.     }
8.
9.     Console.WriteLine("Item popped is {0}", top.data);
```

DOWNLOADED FROM BATU-EXAMS.in

```
10.      top = top.next;
11. }
```

Peek the element from Stack

The peek operation will always return the top element of Stack without removing it from Stack.

Algorithm

- If the Stack is empty, terminate the method as it is Stack underflow.
- If the Stack is not empty, return the element pointed by the top.

The time complexity for Peek operation is O(1). The Peek method will be like following.

```
1. internal void Peek()
2.  {
3.      if (top == null)
4.      {
5.          Console.WriteLine("Stack Underflow.");
6.          return;
7.      }
8.
9.      Console.WriteLine("{0} is on the top of Stack", this.
   top.data);
10.  }
```

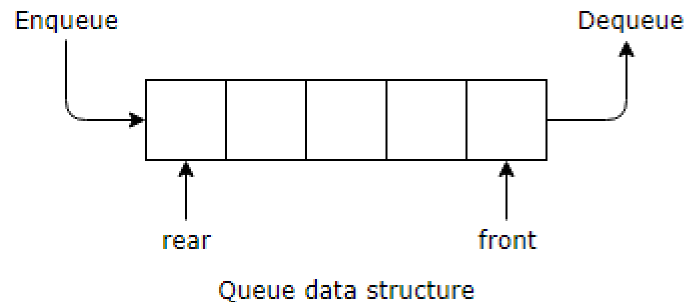
Uses of Stack

- Stack can be used to implement back/forward button in the browser.
- Undo feature in the text editors is also implemented using Stack.
- It is also used to implement recursion.
- Call and return mechanism for a method uses Stack.
- It is also used to implement backtracking.

DOWNLOADED FROM BATU-EXAMS.in

What is Queue?

A Queue is also a linear data structure where insertions and deletions are performed from two different ends. A new element is added from the rear of Queue and deletion of existing element occurs from the front. Since we can access elements from both ends and the element inserted first will be the one to be deleted first, hence Queue is called First in First Out data structure (FIFO).



Queue data structure

Here, we will define two operations on Queue.

- Enqueue - It specifies the insertion of a new element to the Queue. Enqueue will always take place from the rear end of the Queue.
- Dequeue - It specifies the deletion of an existing element from the Queue. Dequeue will always take place from the front end of the Queue.

Implementing Queue functionalities using Linked List

Similar to Stack, the Queue can also be implemented using both, arrays and linked list. But it also has the same drawback of limited size. Hence, we will be using a Linked list to implement the Queue.

The Node class will be the same as defined above in Stack implementation. We will define LinkedListQueue class as below.

```
1. internal class LinkListQueue
2. {
3.     Node front;
4.     Node rear;
5.
6.     public LinkListQueue()
7.     {
8.         this.front = this.rear = null;
9.     }
10.  }
```

Here, we have taken two pointers - rear and front - to refer to the rear and the front end of the Queue respectively and will initialize it to null.

DOWNLOADED FROM BATU-EXAMS.in

Enqueue of an Element

We will add a new element to our Queue from the rear end.

Algorithm

- Create a new node with the value to be inserted.
- If the Queue is empty, then set both front and rear to point to newNode.
- If the Queue is not empty, then set next of rear to the new node and the rear to point to the new node.

The time complexity for Enqueue operation is O(1). The Method for *Enqueue* will be like the following.

```
1. internal void Enqueue(int item)
2.  {
3.      Node newNode = new Node(item);
4.
5.      // If queue is empty, then new node is front and rear
    both
6.      if (this.rear == null)
7.      {
8.          this.front = this.rear = newNode;
9.      }
10.       else
11.       {
12.           // Add the new node at the end of queue and chan
   ge rear
13.           this.rear.next = newNode;
14.           this.rear = newNode;
15.       }
16.       Console.WriteLine("{0} inserted into Queue", item);

17.   }
```

Dequeue of an Element

We will delete the existing element from the Queue from the front end.

Algorithm

- If the Queue is empty, terminate the method.
- If the Queue is not empty, increment front to point to next node.
- Finally, check if the front is null, then set rear to null also. This signifies empty Queue.

The time complexity for Dequeue operation is O(1). The Method for *Dequeue* will be like following.

DOWNLOADED FROM BATU-EXAMS.in

```
1. internal void Dequeue()
2.  {
3.      // If queue is empty, return NULL.
4.      if (this.front == null)
5.      {
6.          Console.WriteLine("The Queue is empty");
7.          return;
8.      }
9.
10.       // Store previous front and move front one node ahea
   d
11.       Node temp = this.front;
12.       this.front = this.front.next;
13.
14.       // If front becomes NULL, then change rear also as N
   ULL
15.       if (this.front == null)
16.       {
17.           this.rear = null;
18.       }
19.
20.       Console.WriteLine("Item deleted is {0}", temp.data);

21.   }
```

Uses of Queue

- CPU scheduling in Operating system uses Queue. The processes ready to execute and the requests of CPU resources wait in a queue and the request is served on first come first serve basis.
- Data buffer - a physical memory storage which is used to temporarily store data while it is being moved from one place to another is also implemented using Queue.

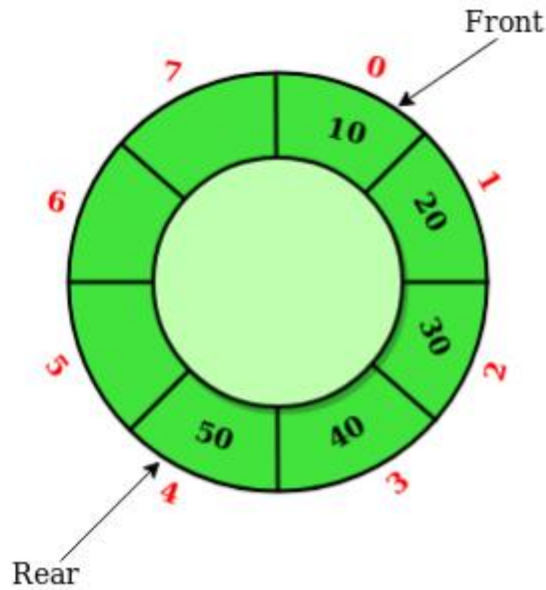
Implementation of Circular Queue

What is a Circular Queue?

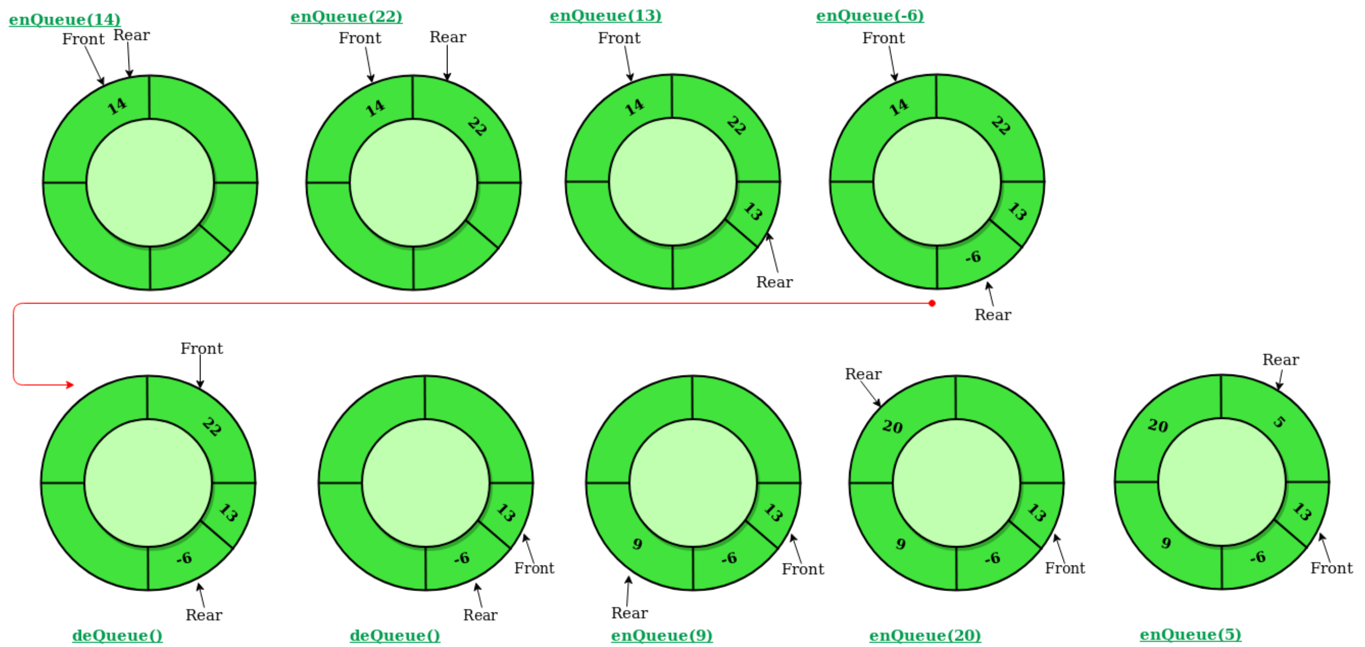
A Circular Queue is a special version of queue where the last element of the queue is connected to the first element of the queue forming a circle.

The operations are performed based on FIFO (First In First Out) principle. It is also called **'Ring Buffer'**.

DOWNLOADED FROM BATU-EXAMS.in



In a normal Queue, we can insert elements until queue becomes full. But once queue becomes full, we cannot insert the next element even if there is a space in front of queue.



Operations on Circular Queue:

- **Front:** Get the front item from queue.
- **Rear:** Get the last item from queue.
- **enQueue(value)** This function is used to insert an element into the circular queue. In a circular queue, the new element is always inserted at Rear position.

DOWNLOADED FROM BATU-EXAMS.in

1. Check whether queue is Full – Check ((rear == SIZE-1 && front == 0) || (rear == front-1)).
2. If it is full then display Queue is full. If queue is not full then, check if (rear == SIZE – 1 && front != 0) if it is true then set rear=0 and insert element.

- **deQueue()** This function is used to delete an element from the circular queue. In a circular queue, the element is always deleted from front position.
    1. Check whether queue is Empty means check (front==-1).
    2. If it is empty then display Queue is empty. If queue is not empty then step 3
    3. Check if (front==rear) if it is true then set front=rear= -1 else check if (front==size-1), if it is true then set front=0 and return the element.

Steps to implement Circular Queue using Array:

1. Initialize an array queue of size n, where n is the maximum number of elements that the queue can hold.
2. Initialize two variables front and rear to -1.
3. To enqueue an element x onto the queue, do the following:
    - Increment rear by 1.
    - If rear is equal to n, set rear to 0.
    - If front is -1, set front to 0.
    - Set queue[rear] to x.
4. To dequeue an element from the queue, do the following:
    - Check if the queue is empty by checking if front is -1. If it is, return an error message indicating that the queue is empty.
    - Set x to queue[front].
    - If front is equal to rear, set front and rear to -1.
    - Otherwise, increment front by 1 and if front is equal to n, set front to 0.
    - Return x.

Implementation:

```
// C or C++ program for insertion and
// deletion in Circular Queue
#include<bits/stdc++.h>
using namespace std;

class Queue
{
    // Initialize front and rear
    int rear, front;

    // Circular Queue
    int size;
    int *arr;
```

DOWNLOADED FROM BATU-EXAMS.in

```
public:
    Queue(int s)
    {
       front = rear = -1;
       size = s;
       arr = new int[s];
    }

    void enQueue(int value);
    int deQueue();
    void displayQueue();
};


/* Function to create Circular queue */
void Queue::enQueue(int value)
{
    if ((front == 0 && rear == size-1) ||
            (rear == (front-1)%(size-1)))
    {
        printf("\nQueue is Full");
        return;
    }

    else if (front == -1) /* Insert First Element */
    {
        front = rear = 0;
        arr[rear] = value;
    }

    else if (rear == size-1 && front != 0)
    {
        rear = 0;
        arr[rear] = value;
    }

    else
    {
        rear++;
        arr[rear] = value;
    }
}

// Function to delete element from Circular Queue
int Queue::deQueue()
{
    if (front == -1)
    {
```

DOWNLOADED FROM BATU-EXAMS.in

```
        printf("\nQueue is Empty");
        return INT_MIN;
    }

    int data = arr[front];
    arr[front] = -1;
    if (front == rear)
    {
        front = -1;
        rear = -1;
    }
    else if (front == size-1)
        front = 0;
    else
        front++;

    return data;
}

// Function displaying the elements
// of Circular Queue
void Queue::displayQueue()
{
    if (front == -1)
    {
        printf("\nQueue is Empty");
        return;
    }
    printf("\nElements in Circular Queue are: ");
    if (rear >= front)
    {
        for (int i = front; i <= rear; i++)
            printf("%d ",arr[i]);
    }
    else
    {
        for (int i = front; i < size; i++)
            printf("%d ", arr[i]);

        for (int i = 0; i <= rear; i++)
            printf("%d ", arr[i]);
    }
}

/* Driver of the program */
int main()
{
    Queue q(5);
```

DOWNLOADED FROM BATU-EXAMS.in

```
    // Inserting elements in Circular Queue
    q.enQueue(14);
    q.enQueue(22);
    q.enQueue(13);
    q.enQueue(-6);

    // Display elements present in Circular Queue
    q.displayQueue();

    // Deleting elements from Circular Queue
    printf("\nDeleted value = %d", q.deQueue());
    printf("\nDeleted value = %d", q.deQueue());

    q.displayQueue();

    q.enQueue(9);
    q.enQueue(20);
    q.enQueue(5);

    q.displayQueue();

    q.enQueue(20);
    return 0;
}
```

Output

```
Elements in Circular Queue are: 14 22 13 -6
Deleted value = 14
Deleted value = 22
Elements in Circular Queue are: 13 -6
Elements in Circular Queue are: 13 -6 9 20 5
Queue is Full
```

Time Complexity: Time complexity of enQueue(), deQueue() operation is **O(1)** as there is no loop in any of the operation.

Applications:

1. **Memory Management:** The unused memory locations in the case of ordinary queues can be utilized in circular queues.
2. **Traffic system:** In computer-controlled traffic system, circular queues are used to switch on the traffic lights one by one repeatedly as per the time set.
3. **CPU Scheduling:** Operating systems often maintain a queue of processes that are ready to execute or that are waiting for a particular event to occur.

DOWNLOADED FROM BATU-EXAMS.in

Application of Stack:

Expression Evaluation:

While reading the expression from left to right, push the element in the stack if it is an operand. Pop the two operands from the stack, if the element is an operator and then evaluate it. Push back the result of the evaluation. Repeat it till the end of the expression.

Expression Conversion:

One of the applications of Stack is in the conversion of arithmetic expressions in high-level programming languages into machine readable form. An expression can be represented in infix, postfix and prefix and stack proves to be useful while converting one form to another.

Syntax Parsing:

Conversion from one form of the expression to another form needs a stack. Many compilers use a stack for parsing the syntax of expressions, program blocks etc. before translating into low level code.

Parenthesis Checking:

One of the most important applications of stacks is to check if the parentheses are balanced in each expression. The compiler generates an error if the parentheses are not matched.

String Reversal:

Reversing string is an operation of Stack by using it we can reverse any string.

Function Call:

The function call stack (often referred to just as the call stack or the stack) is responsible for maintaining the local variables and parameters during function execution.

Expression Evaluation

Evaluate an expression represented by a String. The expression can contain parentheses, you can assume parentheses are well-matched. For simplicity, you can assume only binary operations allowed are +, -, *, and /. Arithmetic Expressions can be written in one of three forms:

- ***Infix Notation:*** Operators are written between the operands they operate on, e.g., 3 + 4.

DOWNLOADED FROM BATU-EXAMS.in

- ***Prefix Notation:*** Operators are written before the operands, e.g., + 3 4.
- ***Postfix Notation:*** Operators are written after operands.

Infix Expressions are harder for Computers to evaluate because of the additional work needed to decide precedence. Infix notation is how expressions are written and recognized by humans and, generally, input to programs. Given that they are harder to evaluate, they are generally converted to one of the two remaining forms. A very well-known algorithm for converting an infix notation to a postfix notation is Shunting Yard Algorithm by Edgar Dijkstra.

This algorithm takes as input an Infix Expression and produces a queue that has this expression converted to postfix notation. The same algorithm can be modified so that it outputs the result of the evaluation of expression instead of a queue. The trick is using two stacks instead of one, one for operands, and one for operators.

```
1. While there are still tokens to be read in,
   1.1 Get the next token.
   1.2 If the token is:
       1.2.1 A number: push it onto the value stack.
       1.2.2 A variable: get its value and push onto the value stack.
       1.2.3 A left parenthesis: push it onto the operator stack.
       1.2.4 A right parenthesis:
         1 While the thing on top of the operator stack is not a
           left parenthesis,
             1 Pop the operator from the operator stack.
             2 Pop the value stack twice, getting two operands.
             3 Apply the operator to the operands, in the correct order.
             4 Push the result onto the value stack.
         2 Pop the left parenthesis from the operator stack and discard it.
       1.2.5 An operator (call it thisOp):
         1 While the operator stack is not empty, and the top thing on the
           operator stack has the same or greater precedence as thisOp,
           1 Pop the operator from the operator stack.
           2 Pop the value stack twice, getting two operands.
           3 Apply the operator to the operands, in the correct order.
           4 Push the result onto the value stack.
         2 Push thisOp onto the operator stack.
2. While the operator stack is not empty,
    1 Pop the operator from the operator stack.
    2 Pop the value stack twice, getting two operands.
```

DOWNLOADED FROM BATU-EXAMS.in

```
    3 Apply the operator to the operands, in the correct order.
    4 Push the result onto the value stack.
3. At this point the operator stack should be empty, and the value
   stack should have only one value in it, which is the final result.
```

Implementation:

It should be clear that this algorithm runs in linear time – each number or operator is pushed onto and popped from Stack only once.

```
// CPP program to evaluate a given
// expression where tokens are
// separated by space.
#include <bits/stdc++.h>
using namespace std;

// Function to find precedence of
// operators.
int precedence(char op){
    if(op == '+'||op == '-')
    return 1;
    if(op == '*'||op == '/')
    return 2;
    return 0;
}

// Function to perform arithmetic operations.
int applyOp(int a, int b, char op){
    switch(op){
        case '+': return a + b;
        case '-': return a - b;
        case '*': return a * b;
        case '/': return a / b;
    }
}

// Function that returns value of
// expression after evaluation.
int evaluate(string tokens){
    int i;

    // stack to store integer values.
    stack <int> values;

    // stack to store operators.
    stack <char> ops;
```

DOWNLOADED FROM BATU-EXAMS.in

```
    for(i = 0; i < tokens.length(); i++){

        // Current token is a whitespace,
        // skip it.
        if(tokens[i] == ' ')
            continue;

        // Current token is an opening
        // brace, push it to 'ops'
        else if(tokens[i] == '('){
            ops.push(tokens[i]);
        }

        // Current token is a number, push
        // it to stack for numbers.
        else if(isdigit(tokens[i])){
            int val = 0;

            // There may be more than one
            // digits in number.
            while(i < tokens.length() &&
                        isdigit(tokens[i]))
            {
                val = (val*10) + (tokens[i]-'0');
                i++;
            }

            values.push(val);

            // right now the i points to
            // the character next to the digit,
            // since the for loop also increases
            // the i, we would skip one
            //  token position; we need to
            // decrease the value of i by 1 to
            // correct the offset.
              i--;
        }

        // Closing brace encountered, solve
        // entire brace.
        else if(tokens[i] == ')')
        {
            while(!ops.empty() && ops.top() != '(')
            {
                int val2 = values.top();
                values.pop();
```

DOWNLOADED FROM BATU-EXAMS.in

```
                int val1 = values.top();
                values.pop();

                char op = ops.top();
                ops.pop();

                values.push(applyOp(val1, val2, op));
            }

            // pop opening brace.
            if(!ops.empty())
               ops.pop();
        }

        // Current token is an operator.
        else
        {
            // While top of 'ops' has same or greater
            // precedence to current token, which
            // is an operator. Apply operator on top
            // of 'ops' to top two elements in values stack.
            while(!ops.empty() && precedence(ops.top())
                                >= precedence(tokens[i])){
                int val2 = values.top();
                values.pop();

                int val1 = values.top();
                values.pop();

                char op = ops.top();
                ops.pop();

                values.push(applyOp(val1, val2, op));
            }

            // Push current token to 'ops'.
            ops.push(tokens[i]);
        }
    }

    // Entire expression has been parsed at this
    // point, apply remaining ops to remaining
    // values.
    while(!ops.empty()){
        int val2 = values.top();
        values.pop();
```

DOWNLOADED FROM BATU-EXAMS.in

```
        int val1 = values.top();
        values.pop();

        char op = ops.top();
        ops.pop();

        values.push(applyOp(val1, val2, op));
    }

    // Top of 'values' contains result, return it.
    return values.top();
}

int main() {
    cout << evaluate("10 + 2 * 6") << "\n";
    cout << evaluate("100 * 2 + 12") << "\n";
    cout << evaluate("100 * ( 2 + 12 )") << "\n";
    cout << evaluate("100 * ( 2 + 12 ) / 14");
    return 0;
}
```

Output

22

212

1400

100

Time Complexity: O(n)
Space Complexity: O(n)

Write down the algorithm to convert infix notation into postfix.

OR

Write down the algorithm to evaluate the infix expression.

Algorithm for Infix To Postfix Conversion:

1. Create an empty stack and an empty postfix output string/stream.
2. Scan the infix input string/stream left to right.
3. If the current input token is an operand, simply append it to the output string (note the examples above that the operands remain in the same order)
4. If the current input token is an operator, pop off all operators that have equal or higher precedence and append them to the output string; push the operator onto the stack. The order of popping is the order in the output.
5. If the current input token is '(', push it onto the stack

DOWNLOADED FROM BATU-EXAMS.in

6. If the current input token is ')', pop off all operators and append them to the output string until a '(' is popped; discard the '('.
7. If the end of the input string is found, pop all operators, and append them to the output string.

Convert following infix expression into postfix expression A + (B*C + D)/E.

(A+(B*C+D)/E)

Symbols	Stack	Postfix string P
(	(	
A	(	A
+	(+	A
(	(+(	A
B	(+(	AB
*	(+(*	AB
C	(+(*	ABC
+	(+(+	ABC*
D	(+(+	ABC*D
)	(+	ABC*D+
/	(+/	ABC*D+
E	(+/	ABC*D+E
)		ABC*D+E/+

DOWNLOADED FROM BATU-EXAMS.in

Consider the following infix expression a convert into reverse polish notation using stack.

Expression=A + (B*C - (D/E ^ F) * H)

Symbols	Stack	Postfix
(	(	
A	(	A
+	(+	A
(	(+(	A
B	(+(	AB
*	(+(*	AB
C	(+(*	ABC
-	(+(-	ABC*
(	(+(-(	ABC*
D	(+(-(	ABC*D
/	(+(-(/	ABC*D
E	(+(-(/	ABC*DE
^	(+(-(/^	ABC*DE
F	(+(-(/^	ABC*DEF
)	(+(-	ABC*DEF^/
*	(+(-*	ABC*DEF^/
H	(+(-*	ABC*DEF^/H
)	(+	ABC*DEF^/H*-
)		ABC*DEF^/H*-+

Postfix Expression.

Write down the algorithm to evaluate the postfix expression.
OR
Write down the algorithm to convert postfix to infix.

DOWNLOADED FROM BATU-EXAMS.in

Postfix: If an operator appears before operand in the expression, then expression is known as Postfix operation.

Infix: If an operator is in between every pair of operands in the expression then expression is known as Infix operation.

Algorithm to Convert Expression from Postfix to Infix:

1. If a character is operand, push it to stack.
2. If a character is an operator,
3. pop operand from the stack, say it's s1.
4. pop operand from the stack, say it's s2.
5. perform (s2 operator s1) and push it to stack.
6. Once the expression iteration is completed, initialize the result string, and pop out from the stack and add it to the result.
7. Return the result.

Consider the following arithmetic expression written in infix notation:

i) E = (A + B) * C + D / (B + A * C) + D

ii) E = A / B ^ C + D * E - A * C

Convert the above expression into postfix and prefix notation.

i) Postfix Expression:

Q13)

Symbols	Stack	Postfix
(	(	
(	((	
A	((	A
+	((+	A
B	((+	AB
)	(	AB+
*	(*	AB+
C	(*	AB+C
+	(+	AB+C*
D	(+	AB+C*D
/	(+/	AB+C*D
(	(+/(	AB+C*D
B	(+/(	AB+C*DB
+	(+/(+	AB+C*DB
A	(+/(+	AB+C*DBA
*	(+/(+*	AB+C*DBA
C	(+/(+*	AB+C*DBAC
)	(+/	AB+C*DBAC*+
+	(+	AB+C*DBAC*+/+
D	(+	AB+C*DBAC*+/+D
)		ABC+C*DBAC*+/+D+

Postfix Expression

DOWNLOADED FROM BATU-EXAMS.in

Recursion

What is Recursion?

The process in which a function calls itself directly or indirectly is called recursion and the corresponding function is called a recursive function. Using a recursive algorithm, certain problems can be solved quite easily. Examples of such problems are Towers of Hanoi (TOH), Inorder/Preorder/Postorder Tree Traversals, DFS of Graph, etc. A recursive function solves a particular problem by calling a copy of itself and solving smaller subproblems of the original problems. Many more recursive calls can be generated as and when required. It is essential to know that we should provide a certain case in order to terminate this recursion process. So, we can say that every time the function calls itself with a simpler version of the original problem.

Need of Recursion

Recursion is an amazing technique with the help of which we can reduce the length of our code and make it easier to read and write. It has certain advantages over the iteration technique which will be discussed later. A task that can be defined with its similar subtask, recursion is one of the best solutions for it. For example, The Factorial of a number.

Properties of Recursion:

- Performing the same operations multiple times with different inputs.
- In every step, we try smaller inputs to make the problem smaller.
- Base condition is needed to stop the recursion otherwise infinite loop will occur.

Algorithm: Steps

The algorithmic steps for implementing recursion in a function are as follows:

Step1 - Define a base case: Identify the simplest case for which the solution is known or trivial. This is the stopping condition for the recursion, as it prevents the function from infinitely calling itself.

Step2 - Define a recursive case: Define the problem in terms of smaller subproblems. Break the problem down into smaller versions of itself and call the function recursively to solve each subproblem.

Step3 - Ensure the recursion terminates: Make sure that the recursive function eventually reaches the base case and does not enter an infinite loop.

Step4 - Combine the solutions: Combine the solutions of the subproblems to solve the original problem.

DOWNLOADED FROM BATU-EXAMS.in

A Mathematical Interpretation

Let us consider a problem that a programmer must determine the sum of first n natural numbers, there are several ways of doing that, but the simplest approach is simply to add the numbers starting from 1 to n. So, the function simply looks like this,

approach (1) – Simply adding one by one.
f(n) = 1 + 2 + 3 +……..+ n

but there is another mathematical approach of representing this,

approach (2) – Recursive adding
f(n) = 1 n=1
f(n) = n + f(n-1) n>1

There is a simple difference between the approach (1) and approach(2) and that is in **approach(2)** the function " **f( )** " itself is being called inside the function, so this phenomenon is named recursion, and the function containing recursion is called recursive function, at the end, this is a great tool in the hand of the programmers to code some problems in a lot easier and efficient way.

How are recursive functions stored in memory?
Recursion uses more memory, because the recursive function adds to the stack with each recursive call and keeps the values there until the call is finished. The recursive function uses LIFO (LAST IN FIRST OUT) Structure just like the stack data structure.

What is the base condition in recursion?
In the recursive program, the solution to the base case is provided and the solution to the bigger problem is expressed in terms of smaller problems.

```
int fact(int n)
{
    if (n < = 1) // base case
        return 1;
    else
        return n*fact(n-1);
}
```

In the above example, the base case for n < = 1 is defined and the larger value of a number can be solved by converting to a smaller one till the base case is reached.

How is a particular problem solved using recursion?
The idea is to represent a problem in terms of one or more smaller problems and add one or more base conditions that stop the recursion. For example, we compute factorial n if we know the factorial of (n-1). The base case for factorial would be n = 0. We return 1 when n = 0.

DOWNLOADED FROM BATU-EXAMS.in

Why Stack Overflow error occurs in recursion?
If the base case is not reached or not defined, then the stack overflow problem may arise. Let us take an example to understand this.

```
int fact(int n)
{
    // wrong base case (it may cause
    // stack overflow).
    if (n == 100)
        return 1;

    else
        return n*fact(n-1);
}
```

If fact(10) is called, it will call fact(9), fact(8), fact(7), and so on but the number will never reach 100. So, the base case is not reached. If the memory is exhausted by these functions on the stack, it will cause a stack overflow error.

What is the difference between direct and indirect recursion?
A function fun is called direct recursive if it calls the same function fun. A function fun is called indirect recursive if it calls another function say fun_new and fun_new calls fun directly or indirectly.

```
// An example of direct recursion
void directRecFun()
{
    // Some code....

    directRecFun();

    // Some code...
}

// An example of indirect recursion
void indirectRecFun1()
{
    // Some code...

    indirectRecFun2();

    // Some code...
```

DOWNLOADED FROM BATU-EXAMS.in

```
}
void indirectRecFun2()
{
    // Some code...

    indirectRecFun1();

    // Some code...
}
```

How is memory allocated to different function calls in recursion?

When any function is called from main(), the memory is allocated to it on the stack. A recursive function calls itself, the memory for a called function is allocated on top of memory allocated to the calling function and a different copy of local variables is created for each function call. When the base case is reached, the function returns its value to the function by whom it is called, and memory is de-allocated and the process continues.
Let us take the example of how recursion works by taking a simple function.

```cpp
// A C++ program to demonstrate working of
// recursion
#include <bits/stdc++.h>
using namespace std;

void printFun(int test)
{
    if (test < 1)
        return;
    else {
        cout << test << " ";
        printFun(test - 1); // statement 2
        cout << test << " ";
        return;
    }
}

// Driver Code
int main()
{
    int test = 3;
    printFun(test);
}
```

Output
3 2 1 1 2 3

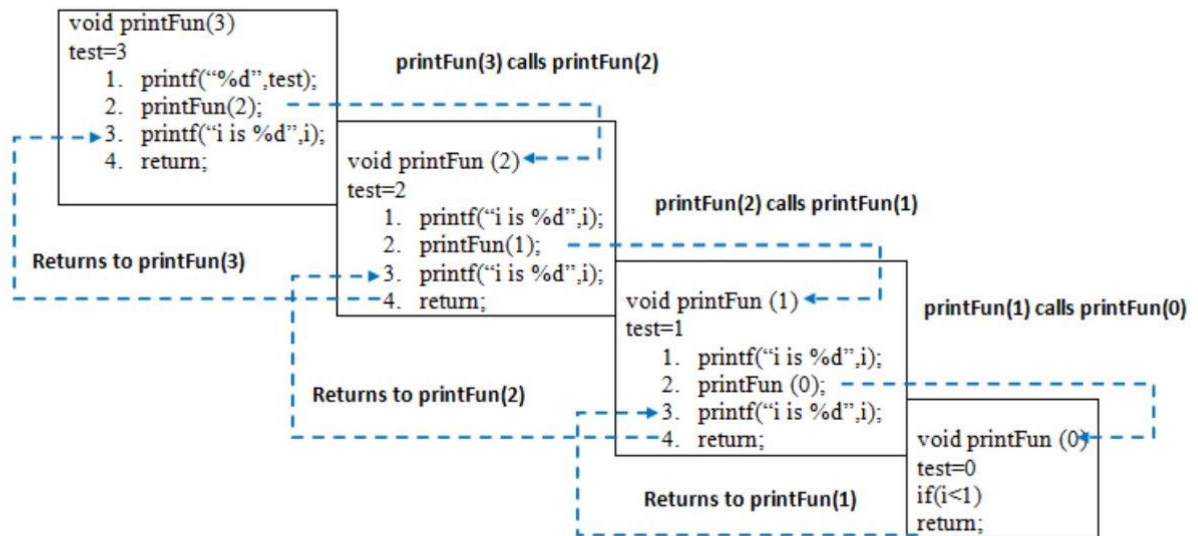
Output:
3 2 1 1 2 3

DOWNLOADED FROM BATU-EXAMS.in

Time Complexity: O (1)
Auxiliary Space: O (1)

When **printFun(3)** is called from main(), memory is allocated to **printFun(3)** and a local variable test is initialized to 3 and statement 1 to 4 are pushed on the stack as shown in below diagram. It first prints '3'. In statement 2, **printFun(2)** is called and memory is allocated to **printFun(2)** and a local variable test is initialized to 2 and statement 1 to 4 are pushed into the stack.

Similarly, **printFun(2)** calls **printFun(1)** and **printFun(1)** calls **printFun(0)**. **printFun(0)** goes to if statement and it return to **printFun(1)**. The remaining statements of **printFun(1)** are executed and it returns to **printFun(2)** and so on. In the output, values from 3 to 1 are printed and then 1 to 3 are printed. The memory stack has been shown in below diagram.



Recursion VS Iteration

SR No.	Recursion	Iteration
1)	Terminates when the base case becomes true.	Terminates when the condition becomes false.
2)	Used with functions.	Used with loops.
3)	Every recursive call needs extra space in the stack memory.	Every iteration does not require any extra space.
4)	Smaller code size.	Larger code size.

DOWNLOADED FROM BATU-EXAMS.in

Basic understanding of Recursion.

Problem 1: Write a program and recurrence relation to find the Fibonacci series of n where n>2.

Mathematical Equation:

n if n == 0, n == 1.

fib(n) = fib(n-1) + fib(n-2) otherwise.

Recurrence Relation:

T(n) = T(n-1) + T(n-2) + O (1)

Recursive program:

Input: n = 5

Output:

Fibonacci series of 5 numbers is: 0 1 1 2 3.

Implementation:

```
// C code to implement Fibonacci series
#include <stdio.h>

// Function for fibonacci
int fib(int n)
{
    // Stop condition
    if (n == 0)
        return 0;

    // Stop condition
    if (n == 1 || n == 2)
        return 1;

    // Recursion function
    else
        return (fib(n - 1) + fib(n - 2));
}

// Driver Code
int main()
{
    // Initialize variable n.
    int n = 5;
    printf("Fibonacci series "
           "of %d numbers is: ",
           n);
```

DOWNLOADED FROM BATU-EXAMS.in

```
    // for loop to print the fibonacci series.
    for (int i = 0; i < n; i++) {
        printf("%d ", fib(i));
    }
    return 0;
}
```

Output
Fibonacci series of 5 numbers is: 0 1 1 2 3.

Time Complexity: $O(2^n)$

Auxiliary Space: O(n)

Here is the recursive tree for input 5 which shows a clear picture of how a big problem can be solved into smaller ones.

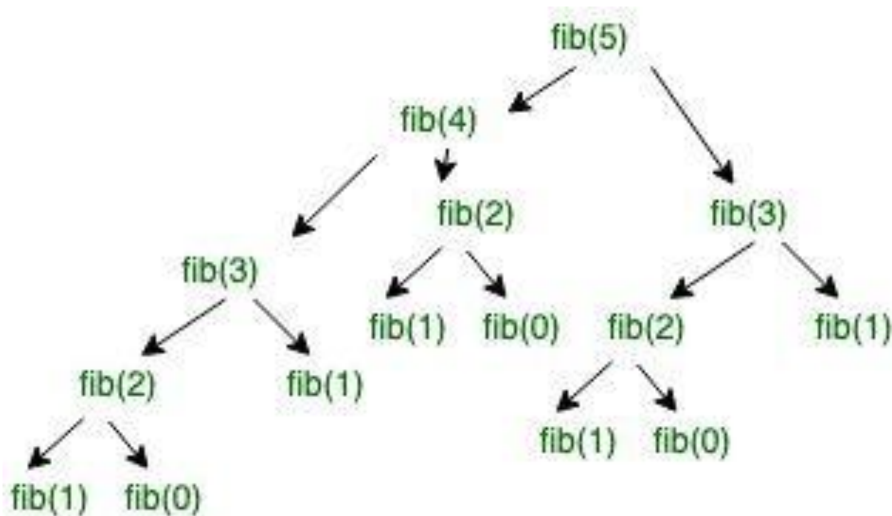
fib(n) is a Fibonacci function. The time complexity of the given program can depend on the function call.

*fib(n) -> level CBT (UB) -> 2^n-1 nodes -> 2^n function call -> 2^n*O(1) -> T(n) = O(2^n)*

For Best Case.

T(n) = θ(2^n\2)

Working:



DOWNLOADED FROM BATU-EXAMS.in

Example: Real Applications of Recursion in real problems

Recursion is a powerful technique that has many applications in computer science and programming. Here are some of the common applications of recursion:

- **Tree and graph traversal**: Recursion is frequently used for traversing and searching data structures such as trees and graphs. Recursive algorithms can be used to explore all the nodes or vertices of a tree or graph in a systematic way.
- **Sorting algorithms**: Recursive algorithms are also used in sorting algorithms such as quicksort and merge sort. These algorithms use recursion to divide the data into smaller subarrays or sub lists, sort them, and then merge them back together.
- **Divide-and-conquer algorithms**: Many algorithms that use a divide-and-conquer approach, such as the binary search algorithm, use recursion to break down the problem into smaller subproblems.
- **Fractal generation**: Fractal shapes and patterns can be generated using recursive algorithms. For example, the Mandelbrot set is generated by repeatedly applying a recursive formula to complex numbers.
- **Backtracking algorithms**: Backtracking algorithms are used to solve problems that involve making a sequence of decisions, where each decision depends on the previous ones. These algorithms can be implemented using recursion to explore all possible paths and backtrack when a solution is not found.
- **Memoization**: Memoization is a technique that involves storing the results of expensive function calls and returning the cached result when the same inputs occur again. Memoization can be implemented using recursive functions to compute and cache the results of subproblems.

What are the disadvantages of recursive programming over iterative programming?

Note that both recursive and iterative programs have the same problem-solving powers, i.e., every recursive program can be written iteratively and vice versa is also true. The recursive program has greater space requirements than the iterative program as all functions will remain in the stack until the base case is reached. It also has greater time requirements because of function calls and returns overhead.
Moreover, due to the smaller length of code, the codes are difficult to understand, and hence extra care must be practiced while writing the code. The computer may run out of memory if the recursive calls are not properly checked.

What are the advantages of recursive programming over iterative programming?

Recursion provides a clean and simple way to write code. Some problems are inherently recursive like tree traversals, Tower of Hanoi, etc. For such problems, it is preferred to write recursive code. We can write such codes also iteratively with the help of a stack data structure. For example, refer Inorder Tree Traversal without Recursion, Iterative Tower of Hanoi.

DOWNLOADED FROM BATU-EXAMS.in

Summary of Recursion:

- There are two types of cases in recursion i.e., recursive case and a base case.
- The base case is used to terminate the recursive function when the case turns out to be true.
- Each recursive call makes a new copy of that method in the stack memory.
- Infinite recursion may lead to running out of stack memory.
- Examples of Recursive algorithms: Merge Sort, Quick Sort, Tower of Hanoi, Fibonacci Series, Factorial Problem, etc.

Priority Queue

A **priority queue** is a type of queue that arranges elements based on their priority values. Elements with higher priority values are typically retrieved before elements with lower priority values.

In a priority queue, each element has a priority value associated with it. When you add an element to the queue, it is inserted in a position based on its priority value. For example, if you add an element with a high priority value to a priority queue, it may be inserted near the front of the queue, while an element with a low priority value may be inserted near the back.

There are several ways to implement a priority queue, including using an array, linked list, heap, or binary search tree. Each method has its own advantages and disadvantages, and the best choice will depend on the specific needs of your application.

Priority queues are often used in real-time systems, where the order in which elements are processed can have significant consequences. They are also used in algorithms to improve their efficiencies, such as **Dijkstra's algorithm** for finding the shortest path in a graph and the A* search algorithm for pathfinding.

Properties of Priority Queue

So, a priority Queue is an extension of the **queue** ***with the following properties.***

- Every item has a priority associated with it.
- An element with high priority is dequeued before an element with low priority.
- If two elements have the same priority, they are served according to their order in the queue.

In the below priority queue, an element with a maximum ASCII value will have the highest priority. The elements with higher priority are served first.

DOWNLOADED FROM BATU-EXAMS.in

Priority Queue

Initial Queue = { }

Operation	Return value	Queue Content
insert (C)		C
insert (O)		C O
insert (D)		C O D
remove max	O	C D
insert (I)		C D I
insert (N)		C D I N
remove max	N	C D I
insert (G)		C D I G

How is Priority assigned to the elements in a Priority Queue?

In a priority queue, generally, the value of an element is considered for assigning the priority.

For example, the element with the highest value is assigned the highest priority and the element with the lowest value is assigned the lowest priority. The reverse case can also be used i.e., the element with the lowest value can be assigned the highest priority. Also, the priority can be assigned according to our needs.

Operations of a Priority Queue:

A typical priority queue supports the following operations:

1) Insertion in a Priority Queue

When a new element is inserted in a priority queue, it moves to the empty slot from top to bottom and left to right. However, if the element is not in the correct place, then it will be compared with the parent node. If the element is not in the correct order, the elements are swapped. The swapping process continues until all the elements are placed in the correct position.

2) Deletion in a Priority Queue

As you know that in a max heap, the maximum element is the root node. And it will remove the element which has maximum priority first. Thus, you remove the root node from the queue. This removal creates an empty slot, which will be further filled with new insertion. Then, it compares the newly inserted element with all the elements inside the queue to maintain the heap invariant.

3) Peek in a Priority Queue

This operation helps to return the maximum element from Max Heap or the minimum element from Min Heap without deleting the node from the priority queue.

DOWNLOADED FROM BATU-EXAMS.in

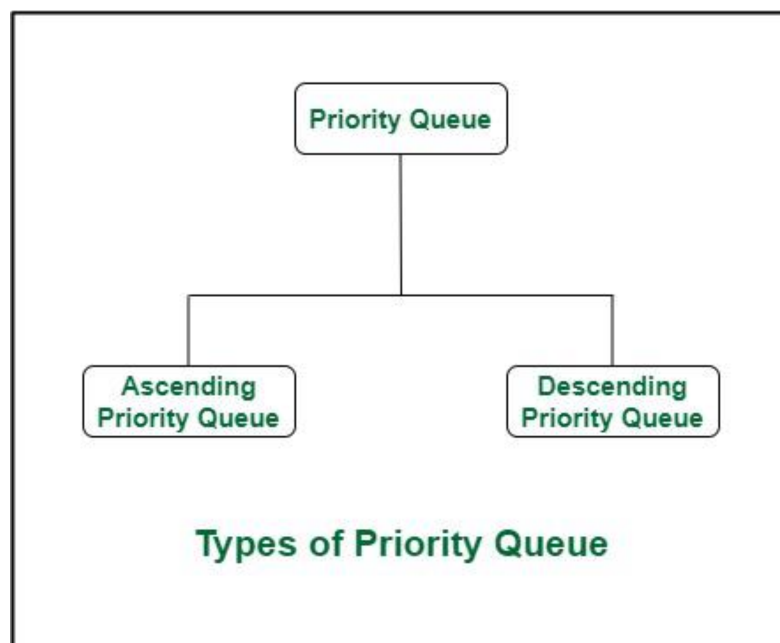
Types of Priority Queue:

1) Ascending Order Priority Queue

As the name suggests, in ascending order priority queue, the element with a lower priority value is given a higher priority in the priority list. For example, if we have the following elements in a priority queue arranged in ascending order like 4,6,8,9,10. Here, 4 is the smallest number, therefore, it will get the highest priority in a priority queue and so when we dequeue from this type of priority queue, 4 will remove from the queue and dequeue returns 4.

2) Descending order Priority Queue

The root node is the maximum element in a max heap, as you may know. It will also remove the element with the highest priority first. As a result, the root node is removed from the queue. This deletion leaves an empty space, which will be filled with fresh insertions in the future. The heap invariant is then maintained by comparing the newly inserted element to all other entries in the queue.



Difference between Priority Queue and Normal Queue?

There is no priority attached to elements in a queue, the rule of first-in-first-out(FIFO) is implemented whereas, in a priority queue, the elements have a priority. The elements with higher priority are served first.

How to Implement Priority Queue?

Priority queue can be implemented using the following data structures:

DOWNLOADED FROM BATU-EXAMS.in

- Arrays
- Linked list
- Heap data structure
- Binary search tree

1) Implement Priority Queue Using Array:

A simple implementation is to use an array of the following structure.

```
struct item {
    int item;
    int priority;
}
```

- **enqueue():** This function is used to insert new data into the queue.
- **dequeue():** This function removes the element with the highest priority from the queue.
- **peek()/top():** This function is used to get the highest priority element in the queue without removing it from the queue.

Arrays	enqueue()	dequeue()	peek()
Time Complexity	O(1)	O(n)	O(n)

```
// C++ program to implement Priority Queue
// using Arrays
#include <bits/stdc++.h>
using namespace std;

// Structure for the elements in the
// priority queue
struct item {
    int value;
    int priority;
};

// Store the element of a priority queue
item pr[100000];

// Pointer to the last index
int size = -1;

// Function to insert a new element
// into priority queue
void enqueue(int value, int priority)
{
    // Increase the size
    size++;
```

DOWNLOADED FROM BATU-EXAMS.in

```
    // Insert the element
    pr[size].value = value;
    pr[size].priority = priority;
}

// Function to check the top element
int peek()
{
    int highestPriority = INT_MIN;
    int ind = -1;

    // Check for the element with
    // highest priority
    for (int i = 0; i <= size; i++) {

        // If priority is same choose
        // the element with the
        // highest value
        if (highestPriority == pr[i].priority && ind > -1
            && pr[ind].value < pr[i].value) {
            highestPriority = pr[i].priority;
            ind = i;
        }
        else if (highestPriority < pr[i].priority) {
            highestPriority = pr[i].priority;
            ind = i;
        }
    }

    // Return position of the element
    return ind;
}

// Function to remove the element with
// the highest priority
void dequeue()
{
    // Find the position of the element
    // with highest priority
    int ind = peek();

    // Shift the element one index before
    // from the position of the element
    // with highest priority is found
    for (int i = ind; i < size; i++) {
        pr[i] = pr[i + 1];
    }
```

DOWNLOADED FROM BATU-EXAMS.in

```
    // Decrease the size of the
    // priority queue by one
    size--;
}

// Driver Code
int main()
{
    // Function Call to insert elements
    // as per the priority
    enqueue(10, 2);
    enqueue(14, 4);
    enqueue(16, 4);
    enqueue(12, 3);

    // Stores the top element
    // at the moment
    int ind = peek();

    cout << pr[ind].value << endl;

    // Dequeue the top element
    dequeue();

    // Check the top element
    ind = peek();
    cout << pr[ind].value << endl;

    // Dequeue the top element
    dequeue();

    // Check the top element
    ind = peek();
    cout << pr[ind].value << endl;

    return 0;
}
```

Output

16

14

12

DOWNLOADED FROM BATU-EXAMS.in

2) Implement Priority Queue Using Linked List:

In a LinkedList implementation, the entries are sorted in descending order based on their priority. The highest priority element is always added to the front of the priority queue, which is formed using linked lists. The functions like **push()**, **pop()**, and **peek()** are used to implement a priority queue using a linked list and are explained as follows:

- **push():** This function is used to insert new data into the queue.
- **pop():** This function removes the element with the highest priority from the queue.
- **peek() / top():** This function is used to get the highest priority element in the queue without removing it from the queue.

Linked List	push()	pop()	peek()
Time Complexity	O(n)	O(1)	O(1)

```
// C++ code to implement Priority Queue
// using Linked List
#include <bits/stdc++.h>
using namespace std;

// Node
typedef struct node {
    int data;

    // Lower values indicate
    // higher priority
    int priority;

    struct node* next;

} Node;

// Function to create a new node
Node* newNode(int d, int p)
{
    Node* temp = (Node*)malloc(sizeof(Node));
    temp->data = d;
    temp->priority = p;
    temp->next = NULL;

    return temp;
}

// Return the value at head
int peek(Node** head) { return (*head)->data; }

// Removes the element with the
```

DOWNLOADED FROM BATU-EXAMS.in

```
// highest priority form the list
void pop(Node** head)
{
    Node* temp = *head;
    (*head) = (*head)->next;
    free(temp);
}

// Function to push according to priority
void push(Node** head, int d, int p)
{
    Node* start = (*head);

    // Create new Node
    Node* temp = newNode(d, p);

    // Special Case: The head of list has
    // lesser priority than new node
    if ((*head)->priority < p) {

        // Insert New Node before head
        temp->next = *head;
        (*head) = temp;
    }
    else {

        // Traverse the list and find a
        // position to insert new node
        while (start->next != NULL
               && start->next->priority > p) {
            start = start->next;
        }

        // Either at the ends of the list
        // or at required position
        temp->next = start->next;
        start->next = temp;
    }
}

// Function to check is list is empty
int isEmpty(Node** head) { return (*head) == NULL; }

// Driver code
int main()
{

    // Create a Priority Queue
```

DOWNLOADED FROM BATU-EXAMS.in

```
    // 7->4->5->6
    Node* pq = newNode(4, 1);
    push(&pq, 5, 2);
    push(&pq, 6, 3);
    push(&pq, 7, 0);

    while (!isEmpty(&pq)) {
        cout << " " << peek(&pq);
        pop(&pq);
    }
    return 0;
}
```

Output

6 5 4 7

3) Implement Priority Queue Using Heaps:

Binary Heap is generally preferred for priority queue implementation because heaps provide better performance compared to arrays or LinkedList. Considering the properties of a heap, The entry with the largest key is on the top and can be removed immediately. It will, however, take time O(log n) to restore the heap property for the remaining keys. However if another entry is to be inserted immediately, then some of this time may be combined with the O(log n) time needed to insert the new entry. Thus, the representation of a priority queue as a heap proves advantageous for large n, since it is represented efficiently in contiguous storage and is guaranteed to require only logarithmic time for both insertions and deletions. Operations on Binary Heap are as follows:

- **insert(p):** Inserts a new element with priority p.
- **extractMax():** Extracts an element with maximum priority.
- **remove(i):** Removes an element pointed by an iterator i.
- **getMax():** Returns an element with maximum priority.
- **changePriority(i, p):** Changes the priority of an element pointed by **i to p**.

Binary Heap	**insert()**	**remove()**	**peek()**
Time Complexity	O(log n)	O(log n)	O(1)

4) Implement Priority Queue Using Binary Search Tree:

A Self-Balancing Binary Search Tree like AVL Tree, Red-Black Tree, etc. can also be used to implement a priority queue. Operations like peek(), insert() and delete() can be performed using BST.

Binary Search Tree	**peek()**	**insert()**	**delete()**
Time Complexity	O(1)	O(log n)	O(log n)

DOWNLOADED FROM BATU-EXAMS.in

Applications of Priority Queue:

- CPU Scheduling
- Graph algorithms like Dijkstra's shortest path algorithm, Prim's Minimum Spanning Tree, etc.
- Stack Implementation
- All queue applications where priority is involved.
- Data compression in Huffman code
- Event-driven simulation such as customers waiting in a queue.
- Finding Kth largest/smallest element.

Advantages of Priority Queue:

- It helps to access the elements in a faster way. This is because elements in a priority queue are ordered by priority, one can easily retrieve the highest priority element without having to search through the entire queue.
- The ordering of elements in a Priority Queue is done dynamically. Elements in a priority queue can have their priority values updated, which allows the queue to dynamically reorder itself as priorities change.
- Efficient algorithms can be implemented. Priority queues are used in many algorithms to improve their efficiency, such as Dijkstra's algorithm for finding the shortest path in a graph and the A* search algorithm for pathfinding.
- Included in real-time systems. This is because priority queues allow you to quickly retrieve the highest priority element, they are often used in real-time systems where time is of the essence.

Disadvantages of Priority Queue:

- High complexity. Priority queues are more complex than simple data structures like arrays and linked lists and may be more difficult to implement and maintain.
- High consumption of memory. Storing the priority value for each element in a priority queue can take up additional memory, which may be a concern in systems with limited resources.
- It is not always the most efficient data structure. In some cases, other data structures like heaps or binary search trees may be more efficient for certain operations, such as finding the minimum or maximum element in the queue.
- At times it is less predictable: This is because the order of elements in a priority queue is determined by their priority values, the order in which elements are retrieved may be less predictable than with other data structures like stacks or queues, which follow a first-in, first-out (FIFO) or last-in, first-out (LIFO) order.

BATU-EXAM

Made by batuexams.com

at MET Bhujbal Knowledege City

The PDF notes on this website are the copyrighted property of batuexams.com.

All rights reserved.