**FUNDAMENTALS OF DATA STRUCTURES (3 UNITS) - C**

Elementary data items. Structured data item, array, ordered list, sparse matrices, stacks, queues sequences. Trees, simple sorting and searching techniques. Tree structures and graphs structures; polish notation, storage management and garbage collection. Hash coding, recursive programming, use of macros.

**Data Structure Basics**

## **Data Definition**

Data Definition defines a particular data with the following characteristics.

* **Atomic** − Definition should define a single concept.
* **Traceable** − Definition should be able to be mapped to some data element.
* **Accurate** − Definition should be unambiguous.
* **Clear and Concise** − Definition should be understandable.

## **Data Object**

Data Object represents an object having a data.

## **Data Type**

Data type is a way to classify various types of data such as integer, string, etc. which determines the values that can be used with the corresponding type of data, the type of operations that can be performed on the corresponding type of data. There are two data types −

* Built-in Data Type
* Derived Data Type

### **Built-in Data Type**

Those data types for which a language has built-in support are known as Built-in Data types. For example, most of the languages provide the following built-in data types.

* Integers
* Boolean (true, false)
* Floating (Decimal numbers)
* Character and Strings

### **Derived Data Type**

Those data types which are implementation independent as they can be implemented in one or the other way are known as derived data types. These data types are normally built by the combination of primary or built-in data types and associated operations on them. For example −

* List
* Array
* Stack
* Queue

## **Basic Operations**

The data in the data structures are processed by certain operations. The particular data structure chosen largely depends on the frequency of the operation that needs to be performed on the data structure.

* Traversing
* Searching
* Insertion
* Deletion
* Sorting
* Merging

# Data Structures and Types

A data structure is a specialized format for organizing, processing, retrieving and storing data. There are several basic and advanced types of data structures, all designed to arrange data to suit a specific purpose. Data structures make it easy for users to access and work with the data they need in appropriate ways. Most importantly, data structures frame the organization of information so that machines and humans can better understand it.

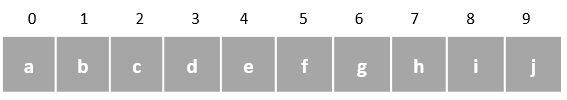
Data types are often confused as a type of data structures, but it is not precisely correct even though they are referred to as Abstract Data Types. Data types represent the nature of the data while data structures are just a collection of similar or different data types in one.

There are usually just two types of data structures −

* Linear
* Non-Linear

## **Linear Data Structures**

The data is stored in linear data structures sequentially. These are rudimentary structures since the elements are stored one after the other without applying any mathematical operations.



Linear data structures are usually easy to implement but since the memory allocation might become complicated, time and space complexities increase. Few examples of linear data structures include −

* Arrays
* Linked Lists
* Stacks
* Queues

Based on the data storage methods, these linear data structures are divided into two sub-types. They are − **static** and **dynamic** data structures.

### **Static Linear Data Structures**

In Static Linear Data Structures, the memory allocation is not scalable. Once the entire memory is used, no more space can be retrieved to store more data. Hence, the memory is required to be reserved based on the size of the program. This will also act as a drawback since reserving more memory than required can cause a wastage of memory blocks.

The best example for static linear data structures is an array.

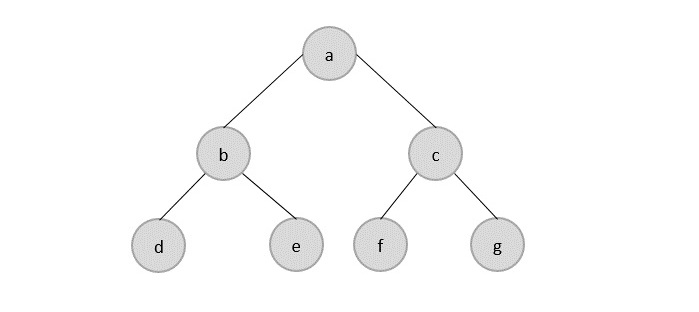
### **Dynamic Linear Data Structures**

In Dynamic linear data structures, the memory allocation can be done dynamically when required. These data structures are efficient considering the space complexity of the program.

Few examples of dynamic linear data structures include: linked lists, stacks and queues.

## **Non-Linear Data Structures**

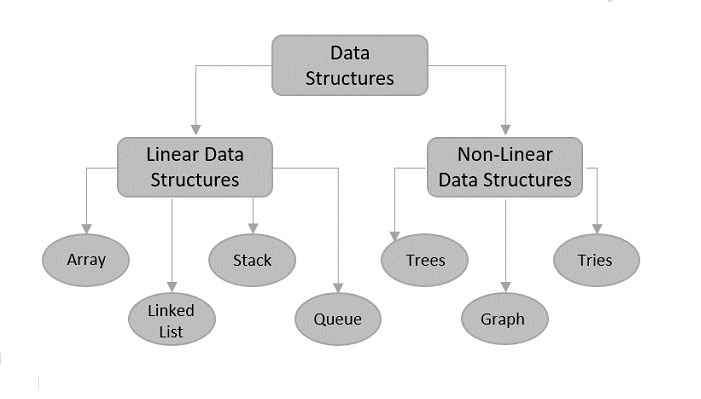
Non-Linear data structures store the data in the form of a hierarchy. Therefore, in contrast to the linear data structures, the data can be found in multiple levels and are difficult to traverse through.



However, they are designed to overcome the issues and limitations of linear data structures. For instance, the main disadvantage of linear data structures is the memory allocation. Since the data is allocated sequentially in linear data structures, each element in these data structures uses one whole memory block. However, if the data uses less memory than the assigned block can hold, the extra memory space in the block is wasted. Therefore, non-linear data structures are introduced. They decrease the space complexity and use the memory optimally.

Few types of non-linear data structures are −

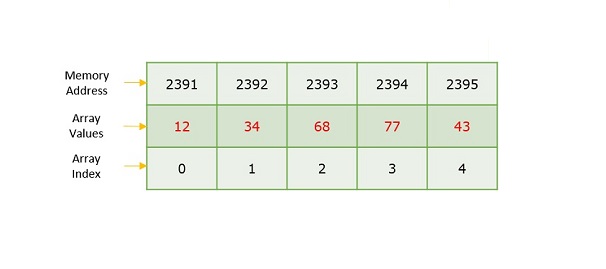
* Graphs
* Trees
* Tries
* Maps



# Array Data Structure

## **What is an Array?**

An array is a type of linear data structure that is defined as a collection of elements with same or different data types. They exist in both single dimension and multiple dimensions. These data structures come into picture when there is a necessity to store multiple elements of similar nature together at one place.



The difference between an array index and a memory address is that the array index acts like a key value to label the elements in the array. However, a memory address is the starting address of free memory available.

Following are the important terms to understand the concept of Array.

* **Element** − Each item stored in an array is called an element.
* **Index** − Each location of an element in an array has a numerical index, which is used to identify the element.

### **Syntax**

Creating an array in **C** and **C++** programming languages −

data\_type array\_name[array\_size]={elements separated by commas}

or,

data\_type array\_name[array\_size];

Creating an array in **Java** programming language −

data\_type[] array\_name = {elements separated by commas}

or,

data\_type array\_name = new data\_type[array\_size];

## **Need for Arrays**

Arrays are used as solutions to many problems from the small sorting problems to more complex problems like travelling salesperson problem. There are many data structures other than arrays that provide efficient time and space complexity for these problems, so what makes using arrays better? The answer lies in the random access lookup time.

Arrays provide **O(1)** random access lookup time. That means, accessing the 1st index of the array and the 1000th index of the array will both take the same time. This is due to the fact that array comes with a pointer and an offset value. The pointer points to the right location of the memory and the offset value shows how far to look in the said memory.

array\_name[index]

| |

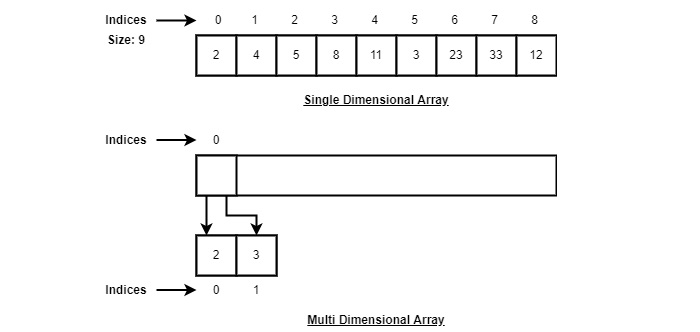
Pointer Offset

Therefore, in an array with 6 elements, to access the 1st element, array is pointed towards the 0th index. Similarly, to access the 6th element, array is pointed towards the 5th index.

## **Array Representation**

Arrays are represented as a collection of buckets where each bucket stores one element. These buckets are indexed from '0' to 'n-1', where n is the size of that particular array. For example, an array with size 10 will have buckets indexed from 0 to 9.

This indexing will be similar for the multidimensional arrays as well. If it is a 2-dimensional array, it will have sub-buckets in each bucket. Then it will be indexed as array\_name[m][n], where m and n are the sizes of each level in the array.



As per the above illustration, following are the important points to be considered.

* Index starts with 0.
* Array length is 9 which means it can store 9 elements.
* Each element can be accessed via its index. For example, we can fetch an element at index 6 as 23.

## **Basic Operations in Arrays**

The basic operations in the Arrays are insertion, deletion, searching, display, traverse, and update. These operations are usually performed to either modify the data in the array or to report the status of the array.

Following are the basic operations supported by an array.

* **Traverse** − print all the array elements one by one.
* **Insertion** − Adds an element at the given index.
* **Deletion** − Deletes an element at the given index.
* **Search** − Searches an element using the given index or by the value.
* **Update** − Updates an element at the given index.
* **Display** − Displays the contents of the array.
* In C, when an array is initialized with size, then it assigns defaults values to its elements in following order.

|  |  |
| --- | --- |
| **Data Type** | **Default Value** |
| bool | false |
| char | 0 |
| int | 0 |
| float | 0.0 |
| double | 0.0f |
| void |  |
| wchar\_t | 0 |

## **Array - Insertion Operation**

In the insertion operation, we are adding one or more elements to the array. Based on the requirement, a new element can be added at the beginning, end, or any given index of array. This is done using input statements of the programming languages.

### **Algorithm**

Following is an algorithm to insert elements into a Linear Array until we reach the end of the array −

* 1. Start
* 2. Create an Array of a desired datatype and size.
* 3. Initialize a variable 'i' as 0.
* 4. Enter the element at ith index of the array.
* 5. Increment i by 1.
* 6. Repeat Steps 4 & 5 until the end of the array.
* 7. Stop

### **Example**

Here, we see a practical implementation of insertion operation, where we add data at the end of the array –

In C

#include <stdio.h>

int main(){

int LA[3] = {}, i;

printf("Array Before Insertion:\n");

for(i = 0; i < 3; i++)

printf("LA[%d] = %d \n", i, LA[i]);

printf("Inserting Elements.. \n");

printf("The array elements after insertion :\n"); // prints array values

for(i = 0; i < 3; i++) {

LA[i] = i + 2;

printf("LA[%d] = %d \n", i, LA[i]);

}

return 0;

}

### **Output**

Array Before Insertion:

LA[0] = 0

LA[1] = 0

LA[2] = 0

Inserting elements..

Array After Insertion:

LA[0] = 2

LA[1] = 3

LA[2] = 4

LA[3] = 5

LA[4] = 6

In C++

#include <iostream>

using namespace std;

int main(){

int LA[3] = {}, i;

cout << "Array Before Insertion:" << endl;

for(i = 0; i < 3; i++)

cout << "LA[" << i <<"] = " << LA[i] << endl;

//prints garbage values

cout << "Inserting elements.." <<endl;

cout << "Array After Insertion:" << endl; // prints array values

for(i = 0; i < 5; i++) {

LA[i] = i + 2;

cout << "LA[" << i <<"] = " << LA[i] << endl;

}

return 0;

}

In JAva

public class ArrayDemo {

public static void main(String []args) {

int LA[] = new int[3];

System.out.println("Array Before Insertion:");

for(int i = 0; i < 3; i++)

System.out.println("LA[" + i + "] = " + LA[i]); //prints empty array

System.out.println("Inserting Elements..");

// Printing Array after Insertion

System.out.println("Array After Insertion:");

for(int i = 0; i < 3; i++) {

LA[i] = i+3;

System.out.println("LA[" + i + "] = " + LA[i]);

}

}

}

In Python

# python program to insert element using insert operation

def insert(arr, element):

arr.append(element)

# Driver's code

if \_\_name\_\_ == '\_\_main\_\_':

# declaring array and value to insert

LA = [0, 0, 0]

x = 0

# array before inserting an element

print("Array Before Insertion: ")

for x in range(len(LA)):

print("LA", [x], " = " , LA[x])

print("Inserting elements....")

# array after Inserting element

for x in range(len(LA)):

LA.append(x);

LA[x] = x+1;

print("Array After Insertion: ")

for x in range(len(LA)):

print("LA", [x], " = " , LA[x])

## **Array - Deletion Operation**

In this array operation, we delete an element from the particular index of an array. This deletion operation takes place as we assign the value in the consequent index to the current index.

### **Algorithm**

Consider LA is a linear array with N elements and K is a positive integer such that K<=N. Following is the algorithm to delete an element available at the Kth position of LA.

1. Start

2. Set J = K

3. Repeat steps 4 and 5 while J < N

4. Set LA[J] = LA[J + 1]

5. Set J = J+1

6. Set N = N-1

7. Stop

### **Example**

Following are the implementations of this operation in various programming languages −

In C

#include <stdio.h>

void main(){

int LA[] = {1,3,5};

int n = 3;

int i;

printf("The original array elements are :\n");

for(i = 0; i<n; i++)

printf("LA[%d] = %d \n", i, LA[i]);

for(i = 1; i<n; i++) {

LA[i] = LA[i+1];

n = n - 1;

}

printf("The array elements after deletion :\n");

for(i = 0; i<n; i++)

printf("LA[%d] = %d \n", i, LA[i]);

}

In C++

#include <iostream>

using namespace std;

int main(){

int LA[] = {1,3,5};

int i, n = 3;

cout << "The original array elements are :"<<endl;

for(i = 0; i<n; i++) {

cout << "LA[" << i << "] = " << LA[i] << endl;

}

for(i = 1; i<n; i++) {

LA[i] = LA[i+1];

n = n - 1;

}

cout << "The array elements after deletion :"<<endl;

for(i = 0; i<n; i++) {

cout << "LA[" << i << "] = " << LA[i] <<endl;

}

}

In Java

public class ArrayDemo {

public static void main(String []args) {

int LA[] = new int[3];

int n = LA.length;

System.out.println("Array Before Deletion:");

for(int i = 0; i < n; i++) {

LA[i] = i + 3;

System.out.println("LA[" + i + "] = " + LA[i]);

}

for(int i = 1; i<n-1; i++) {

LA[i] = LA[i+1];

n = n - 1;

}

System.out.println("Array After Deletion:");

for(int i = 0; i < n; i++) {

System.out.println("LA[" + i + "] = " + LA[i]);

}

}

}

In python

#python program to delete the value using delete operation

if \_\_name\_\_ == '\_\_main\_\_':

# Declaring array and deleting value

LA = [0,0,0]

n = len(LA)

print("Array Before Deletion: ")

for x in range(len(LA)):

LA.append(x)

LA[x] = x + 3

print("LA", [x], " = " , LA[x])

# delete the value if exists

# or show error it does not exist in the list

for x in range(1, n-1):

LA[x] = LA[x+1]

n = n-1

print("Array After Deletion: ")

for x in range(n):

print("LA", [x], " = " , LA[x])

### **Output**

The original array elements are :

LA[0] = 1

LA[1] = 3

LA[2] = 5

The array elements after deletion :

LA[0] = 1

LA[1] = 5

## **Array - Search Operation**

Searching an element in the array using a key; The key element sequentially compares every value in the array to check if the key is present in the array or not.

### **Algorithm**

Consider LA is a linear array with N elements and K is a positive integer such that K<=N. Following is the algorithm to find an element with a value of ITEM using sequential search.

1. Start

2. Set J = 0

3. Repeat steps 4 and 5 while J < N

4. IF LA[J] is equal ITEM THEN GOTO STEP 6

5. Set J = J +1

6. PRINT J, ITEM

7. Stop

### **Example**

Following are the implementations of this operation in JAVA programming languages −

public class ArrayDemo{

public static void main(String []args){

int LA[] = new int[5];

System.out.println("Array:");

for(int i = 0; i < 5; i++) {

LA[i] = i + 3;

System.out.println("LA[" + i + "] = " + LA[i]);

}

for(int i = 0; i < 5; i++) {

if(LA[i] == 6)

System.out.println("Element " + 6 + " is found at index " + i);

}

}

}

In python

#python program to delete the value using delete operation

if \_\_name\_\_ == '\_\_main\_\_':

# Declaring array and deleting value

LA = [0,0,0]

n = len(LA)

print("Array Before Deletion: ")

for x in range(len(LA)):

LA.append(x)

LA[x] = x + 3

print("LA", [x], " = " , LA[x])

# delete the value if exists

# or show error it does not exist in the list

for x in range(1, n-1):

LA[x] = LA[x+1]

n = n-1

print("Array After Deletion: ")

for x in range(n):

print("LA", [x], " = " , LA[x])

### **Output**

The original array elements are :

LA[0] = 1

LA[1] = 3

LA[2] = 5

The array elements after deletion :

LA[0] = 1

LA[1] = 5

## **Array - Search Operation**

Searching an element in the array using a key; The key element sequentially compares every value in the array to check if the key is present in the array or not.

### **Algorithm**

Consider LA is a linear array with N elements and K is a positive integer such that K<=N. Following is the algorithm to find an element with a value of ITEM using sequential search.

1. Start

2. Set J = 0

3. Repeat steps 4 and 5 while J < N

4. IF LA[J] is equal ITEM THEN GOTO STEP 6

5. Set J = J +1

6. PRINT J, ITEM

7. Stop

### **Example**

Following are the implementations of this operation in JAVA programming languages −

public class ArrayDemo{

public static void main(String []args){

int LA[] = new int[5];

System.out.println("Array:");

for(int i = 0; i < 5; i++) {

LA[i] = i + 3;

System.out.println("LA[" + i + "] = " + LA[i]);

}

for(int i = 0; i < 5; i++) {

if(LA[i] == 6)

System.out.println("Element " + 6 + " is found at index " + i);

}

}

}

In Python

# Python code to iterate over a array using python

LA = [1, 3, 5, 7, 8]

# length of the elements

length = len(LA)

# Traversing the elements using For loop and range

# same as 'for x in range(len(array))'

print("Array elements are: ")

for x in range(length):

print("LA", [x], " = ", LA[x])

### **Output**

The original array elements are :

LA[0] = 1

LA[1] = 3

LA[2] = 5

LA[3] = 7

LA[4] = 8

## **Array - Update Operation**

Update operation refers to updating an existing element from the array at a given index.

### **Algorithm**

Consider LA is a linear array with N elements and K is a positive integer such that K<=N. Following is the algorithm to update an element available at the Kth position of LA.

1. Start

2. Set LA[K-1] = ITEM

3. Stop

### **Example**

Following are the implementations of this operation in JAVA programming languages −

public class ArrayDemo {

public static void main(String []args) {

int LA[] = new int[5];

int item = 15;

System.out.println("The array elements are: ");

for(int i = 0; i < 5; i++) {

LA[i] = i + 2;

System.out.println("LA[" + i + "] = " + LA[i]);

}

LA[3] = item;

System.out.println("The array elements after updation are: ");

for(int i = 0; i < 5; i++)

System.out.println("LA[" + i + "] = " + LA[i]);

}

}

In Python

### **Output**

The original array elements are :

LA[0] = 1

LA[1] = 3

LA[2] = 5

LA[3] = 7

LA[4] = 8

The array elements after updation :

LA[0] = 1

LA[1] = 3

LA[2] = 10

LA[3] = 7

LA[4] = 8

## **Array - Display Operation**

This operation displays all the elements in the entire array using a print statement.

### **Algorithm**

Consider LA is a linear array with N elements. Following is the algorithm to display an array elements.

1. Start

2. Print all the elements in the Array

3. Stop

### **Example**

Following are the implementations of this operation in JAVA programming languages –

public class ArrayDemo {

public static void main(String []args) {

int LA[] = new int[5];

System.out.println("The array elements are: ");

for(int i = 0; i < 5; i++) {

LA[i] = i + 2;

System.out.println("LA[" + i + "] = " + LA[i]);

}

}

}

In Python

#Display operation using python

#Display operation using python

#Declaring array elements

LA = [2,3,4,5,6]

#Displaying the array

print("The array elements are: ")

for x in range(len(LA)):

print("LA", [x], " = " , LA[x])

### **Output**

The original array elements are :

LA[0] = 1

LA[1] = 3

LA[2] = 5

LA[3] = 7

LA[4] = 8

# Linked List Data Structure

## **What is Linked List?**

A linked list is a linear data structure which can store a collection of "nodes" connected together via links i.e. pointers. Linked lists nodes are not stored at a contiguous location, rather they are linked using pointers to the different memory locations. A node consists of the data value and a pointer to the address of the next node within the linked list.

A linked list is a dynamic linear data structure whose memory size can be allocated or de-allocated at run time based on the operation insertion or deletion, this helps in using system memory efficiently. Linked lists can be used to implment various data structures like a stack, queue, graph, hash maps, etc.



A linked list starts with a **head** node which points to the first node. Every node consists of data which holds the actual data (value) associated with the node and a next pointer which holds the memory address of the next node in the linked list. The last node is called the tail node in the list which points to **null** indicating the end of the list.

## **Linked Lists vs Arrays**

In case of arrays, the size is given at the time of creation and so arrays are of fixed lenghth where as Linked lists are dynamic in size and any number of nodes can be added in the linked lists dynamically. An array can accommodate similar types of data types where as linked lists can store various nodes of different data types.

## **Types of Linked List**

Following are the various types of linked list.

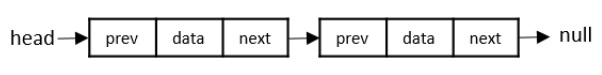
### **Singly Linked Lists**

Singly linked lists contain two "buckets" in one node; one bucket holds the data and the other bucket holds the address of the next node of the list. Traversals can be done in one direction only as there is only a single link between two nodes of the same list.



### **Doubly Linked Lists**

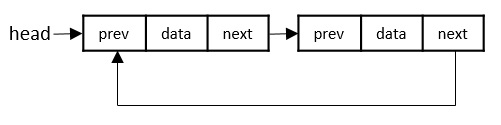
Doubly Linked Lists contain three "buckets" in one node; one bucket holds the data and the other buckets hold the addresses of the previous and next nodes in the list. The list is traversed twice as the nodes in the list are connected to each other from both sides.



### **Circular Linked Lists**

Circular linked lists can exist in both singly linked list and doubly linked list.

Since the last node and the first node of the circular linked list are connected, the traversal in this linked list will go on forever until it is broken.



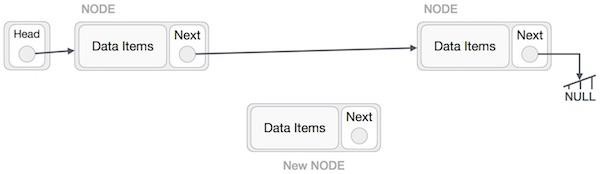
## **Basic Operations in Linked List**

The basic operations in the linked lists are insertion, deletion, searching, display, and deleting an element at a given key. These operations are performed on Singly Linked Lists as given below −

* **Insertion** − Adds an element at the beginning of the list.
* **Deletion** − Deletes an element at the beginning of the list.
* **Display** − Displays the complete list.
* **Search** − Searches an element using the given key.
* **Delete** − Deletes an element using the given key.

## **Linked List - Insertion Operation**

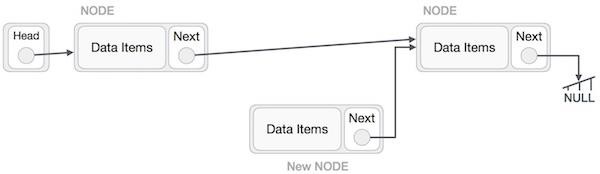
Adding a new node in linked list is a more than one step activity. We shall learn this with diagrams here. First, create a node using the same structure and find the location where it has to be inserted.



Imagine that we are inserting a node B (NewNode), between A (LeftNode) and C (RightNode). Then point B.next to C −

NewNode.next -> RightNode;

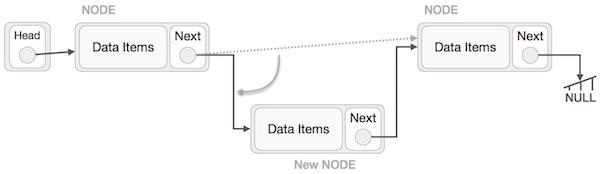
It should look like this −



Now, the next node at the left should point to the new node.

LeftNode.next -> NewNode;

This will put the new node in the middle of the two. The new list should look like this −



Insertion in linked list can be done in three different ways. They are explained as follows −

### **Insertion at Beginning**

In this operation, we are adding an element at the beginning of the list.

**Algorithm**

1. START

2. Create a node to store the data

3. Check if the list is empty

4. If the list is empty, add the data to the node and

assign the head pointer to it.

5. If the list is not empty, add the data to a node and link to the

current head. Assign the head to the newly added node.

6. END

**Example**

Following are the implementations of this operation in C programming languages −

#include <stdio.h>

#include <string.h>

#include <stdlib.h>

struct node {

int data;

struct node \*next;

};

struct node \*head = NULL;

struct node \*current = NULL;

// display the list

void printList(){

struct node \*p = head;

printf("\n[");

//start from the beginning

while(p != NULL) {

printf(" %d ",p->data);

p = p->next;

}

printf("]");

}

//insertion at the beginning

void insertatbegin(int data){

//create a link

struct node \*lk = (struct node\*) malloc(sizeof(struct node));

lk->data = data;

// point it to old first node

lk->next = head;

//point first to new first node

head = lk;

}

void main(){

int k=0;

insertatbegin(12);

insertatbegin(22);

insertatbegin(30);

insertatbegin(44);

insertatbegin(50);

printf("Linked List: ");

// print list

printList();

}

**Output**

Linked List:

[ 50 44 30 22 12 ]

### **Insertion at Ending**

In this operation, we are adding an element at the ending of the list.

**Algorithm**

1. START

2. Create a new node and assign the data

3. Find the last node

4. Point the last node to new node

5. END

**Example**

Following are the implementations of this operation in C programming languages −

#include <stdio.h>

#include <string.h>

#include <stdlib.h>

struct node {

int data;

struct node \*next;

};

struct node \*head = NULL;

struct node \*current = NULL;

// display the list

void printList(){

struct node \*p = head;

printf("\n[");

//start from the beginning

while(p != NULL) {

printf(" %d ",p->data);

p = p->next;

}

printf("]");

}

//insertion at the beginning

void insertatbegin(int data){

//create a link

struct node \*lk = (struct node\*) malloc(sizeof(struct node));

lk->data = data;

// point it to old first node

lk->next = head;

//point first to new first node

head = lk;

}

void insertatend(int data){

//create a link

struct node \*lk = (struct node\*) malloc(sizeof(struct node));

lk->data = data;

struct node \*linkedlist = head;

// point it to old first node

while(linkedlist->next != NULL)

linkedlist = linkedlist->next;

//point first to new first node

linkedlist->next = lk;

}

void main(){

int k=0;

insertatbegin(12);

insertatend(22);

insertatend(30);

insertatend(44);

insertatend(50);

printf("Linked List: ");

// print list

printList();

}

**Output**

Linked List:

[ 12 22 30 44 50 ]

### **Insertion at a Given Position**

In this operation, we are adding an element at any position within the list.

**Algorithm**

1. START

2. Create a new node and assign data to it

3. Iterate until the node at position is found

4. Point first to new first node

5. END

**Example**

Following are the implementations of this operation in C programming languages −

#include <stdio.h>

#include <string.h>

#include <stdlib.h>

struct node {

int data;

struct node \*next;

};

struct node \*head = NULL;

struct node \*current = NULL;

// display the list

void printList(){

struct node \*p = head;

printf("\n[");

//start from the beginning

while(p != NULL) {

printf(" %d ",p->data);

p = p->next;

}

printf("]");

}

//insertion at the beginning

void insertatbegin(int data){

//create a link

struct node \*lk = (struct node\*) malloc(sizeof(struct node));

lk->data = data;

// point it to old first node

lk->next = head;

//point first to new first node

head = lk;

}

void insertafternode(struct node \*list, int data){

struct node \*lk = (struct node\*) malloc(sizeof(struct node));

lk->data = data;

lk->next = list->next;

list->next = lk;

}

void main(){

int k=0;

insertatbegin(12);

insertatbegin(22);

insertafternode(head->next, 30);

printf("Linked List: ");

// print list

printList();

}

**Output**

Linked List:

[ 22 12 30 ]

## **Linked List - Deletion Operation**

Deletion is also a more than one step process. We shall learn with pictorial representation. First, locate the target node to be removed, by using searching algorithms.



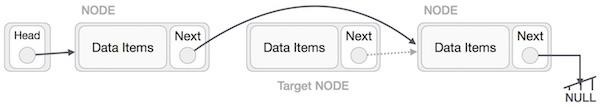
The left (previous) node of the target node now should point to the next node of the target node −

LeftNode.next -> TargetNode.next;

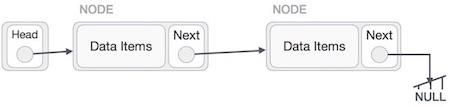


This will remove the link that was pointing to the target node. Now, using the following code, we will remove what the target node is pointing at.

TargetNode.next -> NULL;



We need to use the deleted node. We can keep that in memory otherwise we can simply deallocate memory and wipe off the target node completely.



Similar steps should be taken if the node is being inserted at the beginning of the list. While inserting it at the end, the second last node of the list should point to the new node and the new node will point to NULL.

Deletion in linked lists is also performed in three different ways. They are as follows −

### **Deletion at Beginning**

In this deletion operation of the linked, we are deleting an element from the beginning of the list. For this, we point the head to the second node.

**Algorithm**

1. START

2. Assign the head pointer to the next node in the list

3. END

**Example**

Following are the implementations of this operation in C programming languages −

#include <stdio.h>

#include <string.h>

#include <stdlib.h>

struct node {

int data;

struct node \*next;

};

struct node \*head = NULL;

struct node \*current = NULL;

// display the list

void printList(){

struct node \*p = head;

printf("\n[");

//start from the beginning

while(p != NULL) {

printf(" %d ",p->data);

p = p->next;

}

printf("]");

}

//insertion at the beginning

void insertatbegin(int data){

//create a link

struct node \*lk = (struct node\*) malloc(sizeof(struct node));

lk->data = data;

// point it to old first node

lk->next = head;

//point first to new first node

head = lk;

}

void deleteatbegin(){

head = head->next;

}

int main(){

int k=0;

insertatbegin(12);

insertatbegin(22);

insertatbegin(30);

insertatbegin(40);

insertatbegin(55);

printf("Linked List: ");

// print list

printList();

deleteatbegin();

printf("\nLinked List after deletion: ");

// print list

printList();

}

**Output**

Linked List:

[ 55 40 30 22 12 ]

Linked List after deletion:

[ 40 30 22 12 ]

### **Deletion at Ending**

In this deletion operation of the linked, we are deleting an element from the ending of the list.

**Algorithm**

1. START

2. Iterate until you find the second last element in the list.

3. Assign NULL to the second last element in the list.

4. END

**Example**

Following are the implementations of this operation in C programming languages −

#include <stdio.h>

#include <string.h>

#include <stdlib.h>

struct node {

int data;

struct node \*next;

};

struct node \*head = NULL;

struct node \*current = NULL;

// display the list

void printList(){

struct node \*p = head;

printf("\n[");

//start from the beginning

while(p != NULL) {

printf(" %d ",p->data);

p = p->next;

}

printf("]");

}

//insertion at the beginning

void insertatbegin(int data){

//create a link

struct node \*lk = (struct node\*) malloc(sizeof(struct node));

lk->data = data;

// point it to old first node

lk->next = head;

//point first to new first node

head = lk;

}

void deleteatend(){

struct node \*linkedlist = head;

while (linkedlist->next->next != NULL)

linkedlist = linkedlist->next;

linkedlist->next = NULL;

}

void main(){

int k=0;

insertatbegin(12);

insertatbegin(22);

insertatbegin(30);

insertatbegin(40);

insertatbegin(55);

printf("Linked List: ");

// print list

printList();

deleteatend();

printf("\nLinked List after deletion: ");

// print list

printList();

}

**Output**

Linked List:

[ 55 40 30 22 12 ]

Linked List after deletion:

[ 55 40 30 22 ]

### **Deletion at a Given Position**

In this deletion operation of the linked, we are deleting an element at any position of the list.

**Algorithm**

1. START

2. Iterate until find the current node at position in the list.

3. Assign the adjacent node of current node in the list

to its previous node.

4. END

**Example**

Following are the implementations of this operation in C programming languages

#include <stdio.h>

#include <string.h>

#include <stdlib.h>

struct node {

int data;

struct node \*next;

};

struct node \*head = NULL;

struct node \*current = NULL;

// display the list

void printList(){

struct node \*p = head;

printf("\n[");

//start from the beginning

while(p != NULL) {

printf(" %d ",p->data);

p = p->next;

}

printf("]");

}

//insertion at the beginning

void insertatbegin(int data){

//create a link

struct node \*lk = (struct node\*) malloc(sizeof(struct node));

lk->data = data;

// point it to old first node

lk->next = head;

//point first to new first node

head = lk;

}

void deletenode(int key){

struct node \*temp = head, \*prev;

if (temp != NULL && temp->data == key) {

head = temp->next;

return;

}

// Find the key to be deleted

while (temp != NULL && temp->data != key) {

prev = temp;

temp = temp->next;

}

// If the key is not present

if (temp == NULL) return;

// Remove the node

prev->next = temp->next;

}

void main(){

int k=0;

insertatbegin(12);

insertatbegin(22);

insertatbegin(30);

insertatbegin(40);

insertatbegin(55);

printf("Linked List: ");

// print list

printList();

deletenode(30);

printf("\nLinked List after deletion: ");

// print list

printList();

}

**Output**

Linked List:

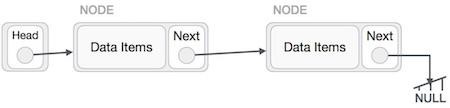
[ 55 40 30 22 12 ]

Linked List after deletion:

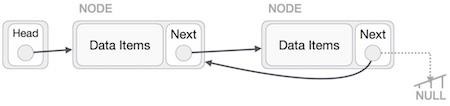
[ 55 40 22 12 ]

## **Linked List - Reversal Operation**

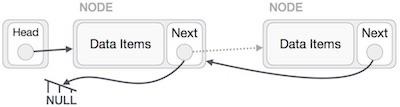
This operation is a thorough one. We need to make the last node to be pointed by the head node and reverse the whole linked list.



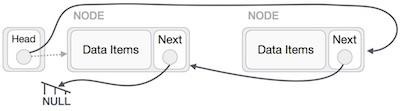
First, we traverse to the end of the list. It should be pointing to NULL. Now, we shall make it point to its previous node −



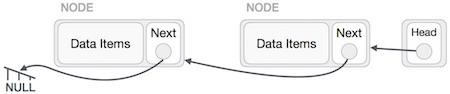
We have to make sure that the last node is not the last node. So we'll have some temp node, which looks like the head node pointing to the last node. Now, we shall make all left side nodes point to their previous nodes one by one.



Except the node (first node) pointed by the head node, all nodes should point to their predecessor, making them their new successor. The first node will point to NULL.



We'll make the head node point to the new first node by using the temp node.



### **Algorithm**

Step by step process to reverse a linked list is as follows −

1. START

2. We use three pointers to perform the reversing:

prev, next, head.

3. Point the current node to head and assign its next value to

the prev node.

4. Iteratively repeat the step 3 for all the nodes in the list.

5. Assign head to the prev node.

### **Example**

Following are the implementations of this operation in C programming languages −

#include <stdio.h>

#include <string.h>

#include <stdlib.h>

struct node {

int data;

struct node \*next;

};

struct node \*head = NULL;

struct node \*current = NULL;

// display the list

void printList(){

struct node \*p = head;

printf("\n[");

//start from the beginning

while(p != NULL) {

printf(" %d ",p->data);

p = p->next;

}

printf("]");

}

//insertion at the beginning

void insertatbegin(int data){

//create a link

struct node \*lk = (struct node\*) malloc(sizeof(struct node));

lk->data = data;

// point it to old first node

lk->next = head;

//point first to new first node

head = lk;

}

void reverseList(struct node\*\* head){

struct node \*prev = NULL, \*cur=\*head, \*tmp;

while(cur!= NULL) {

tmp = cur->next;

cur->next = prev;

prev = cur;

cur = tmp;

}

\*head = prev;

}

void main(){

int k=0;

insertatbegin(12);

insertatbegin(22);

insertatbegin(30);

insertatbegin(40);

insertatbegin(55);

printf("Linked List: ");

// print list

printList();

reverseList(&head);

printf("\nReversed Linked List: ");

printList();

}

**Output**

Linked List:

[ 55 40 30 22 12 ]

Reversed Linked List:

[ 12 22 30 40 55 ]

## **Linked List - Search Operation**

Searching for an element in the list using a key element. This operation is done in the same way as array search; comparing every element in the list with the key element given.

### **Algorithm**

1 START

2 If the list is not empty, iteratively check if the list

contains the key

3 If the key element is not present in the list, unsuccessful

search

4 END

### **Example**

Following are the implementations of this operation in C programming languages

#include <stdio.h>

#include <string.h>

#include <stdlib.h>

struct node {

int data;

struct node \*next;

};

struct node \*head = NULL;

struct node \*current = NULL;

// display the list

void printList(){

struct node \*p = head;

printf("\n[");

//start from the beginning

while(p != NULL) {

printf(" %d ",p->data);

p = p->next;

}

printf("]");

}

//insertion at the beginning

void insertatbegin(int data){

//create a link

struct node \*lk = (struct node\*) malloc(sizeof(struct node));

lk->data = data;

// point it to old first node

lk->next = head;

//point first to new first node

head = lk;

}

int searchlist(int key){

struct node \*temp = head;

while(temp != NULL) {

if (temp->data == key) {

return 1;

}

temp=temp->next;

}

return 0;

}

void main(){

int k=0;

insertatbegin(12);

insertatbegin(22);

insertatbegin(30);

insertatbegin(40);

insertatbegin(55);

printf("Linked List: ");

// print list

printList();

int ele = 30;

printf("\nElement to be searched is: %d", ele);

k = searchlist(30);

if (k == 1)

printf("\nElement is found");

else

printf("\nElement is not found in the list");

}

### **Output**

Linked List:

[ 55 40 30 22 12 ]

Element to be searched is: 30

Element is found

## **Linked List - Traversal Operation**

The traversal operation walks through all the elements of the list in an order and displays the elements in that order.

### **Algorithm**

1. START

2. While the list is not empty and did not reach the end of the list,

print the data in each node

3. END

### **Example**

Following are the implementations of this operation in C programming languages

#include <stdio.h>

#include <string.h>

#include <stdlib.h>

struct node {

int data;

struct node \*next;

};

struct node \*head = NULL;

struct node \*current = NULL;

// display the list

void printList(){

struct node \*p = head;

printf("\n[");

//start from the beginning

while(p != NULL) {

printf(" %d ",p->data);

p = p->next;

}

printf("]");

}

//insertion at the beginning

void insertatbegin(int data){

//create a link

struct node \*lk = (struct node\*) malloc(sizeof(struct node));

lk->data = data;

// point it to old first node

lk->next = head;

//point first to new first node

head = lk;

}

void main(){

int k=0;

insertatbegin(12);

insertatbegin(22);

insertatbegin(30);

printf("Linked List: ");

// print list

printList();

}

### **Output**

Linked List:

[ 30 22 12 ]

## **Linked List - Complete implementation**

Following are the complete implementations of Linked List in C programming languages

#include <stdio.h>

#include <string.h>

#include <stdlib.h>

struct node {

int data;

struct node \*next;

};

struct node \*head = NULL;

struct node \*current = NULL;

// display the list

void printList(){

struct node \*p = head;

printf("\n[");

//start from the beginning

while(p != NULL) {

printf(" %d ",p->data);

p = p->next;

}

printf("]");

}

//insertion at the beginning

void insertatbegin(int data){

//create a link

struct node \*lk = (struct node\*) malloc(sizeof(struct node));

lk->data = data;

// point it to old first node

lk->next = head;

//point first to new first node

head = lk;

}

void insertatend(int data){

//create a link

struct node \*lk = (struct node\*) malloc(sizeof(struct node));

lk->data = data;

struct node \*linkedlist = head;

// point it to old first node

while(linkedlist->next != NULL)

linkedlist = linkedlist->next;

//point first to new first node

linkedlist->next = lk;

}

void insertafternode(struct node \*list, int data){

struct node \*lk = (struct node\*) malloc(sizeof(struct node));

lk->data = data;

lk->next = list->next;

list->next = lk;

}

void deleteatbegin(){

head = head->next;

}

void deleteatend(){

struct node \*linkedlist = head;

while (linkedlist->next->next != NULL)

linkedlist = linkedlist->next;

linkedlist->next = NULL;

}

void deletenode(int key){

struct node \*temp = head, \*prev;

if (temp != NULL && temp->data == key) {

head = temp->next;

return;

}

// Find the key to be deleted

while (temp != NULL && temp->data != key) {

prev = temp;

temp = temp->next;

}

// If the key is not present

if (temp == NULL) return;

// Remove the node

prev->next = temp->next;

}

int searchlist(int key){

struct node \*temp = head;

while(temp != NULL) {

if (temp->data == key) {

return 1;

}

temp=temp->next;

}

return 0;

}

void main(){

int k=0;

insertatbegin(12);

insertatbegin(22);

insertatend(30);

insertatend(44);

insertatbegin(50);

insertafternode(head->next->next, 33);

printf("Linked List: ");

// print list

printList();

deleteatbegin();

deleteatend();

deletenode(12);

printf("\nLinked List after deletion: ");

// print list

printList();

insertatbegin(4);

insertatbegin(16);

printf("\nUpdated Linked List: ");

printList();

k = searchlist(16);

if (k == 1)

printf("\nElement is found");

else

printf("\nElement is not present in the list");

}

### **Output**

Linked List:

[ 50 22 12 33 30 44 ]

Linked List after deletion:

[ 22 33 30 ]

Updated Linked List:

[ 16 4 22 33 30 ]

Element is found

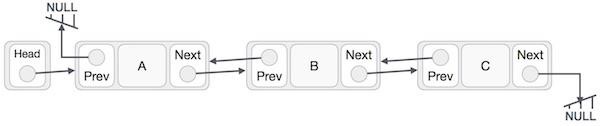
# Doubly Linked List Data Structure

## **What is Doubly Linked List?**

Doubly Linked List is a variation of Linked list in which navigation is possible in both ways, forward as well as backward easily as compared to Single Linked List. Following are the important terms to understand the concept of doubly linked list.

* **Link** − Each link of a linked list can store a data called an element.
* **Next** − Each link of a linked list contains a link to the next link called Next.
* **Prev** − Each link of a linked list contains a link to the previous link called Prev.
* **Linked List** − A Linked List contains the connection link to the first link called First and to the last link called Last.

## **Doubly Linked List Representation**



As per the above illustration, following are the important points to be considered.

* Doubly Linked List contains a link element called first and last.
* Each link carries a data field(s) and a link field called next.
* Each link is linked with its next link using its next link.
* Each link is linked with its previous link using its previous link.
* The last link carries a link as null to mark the end of the list.

## **Basic Operations in Doubly Linked List**

Following are the basic operations supported by a list.

* **Insertion** − Adds an element at the beginning of the list.
* **Insert Last** − Adds an element at the end of the list.
* **Insert After** − Adds an element after an item of the list.
* **Deletion** − Deletes an element at the beginning of the list.
* **Delete Last** − Deletes an element from the end of the list.
* **Delete** − Deletes an element from the list using the key.
* **Display forward** − Displays the complete list in a forward manner.
* **Display backward** − Displays the complete list in a backward manner.

## **Doubly Linked List - Insertion at the Beginning**

In this operation, we create a new node with three compartments, one containing the data, the others containing the address of its previous and next nodes in the list. This new node is inserted at the beginning of the list.

### **Algorithm**

1. START

2. Create a new node with three variables: prev, data, next.

3. Store the new data in the data variable

4. If the list is empty, make the new node as head.

5. Otherwise, link the address of the existing first node to the

next variable of the new node, and assign null to the prev variable.

6. Point the head to the new node.

7. END

### **Example**

Following are the implementations of this operation in JAVA programming languages

//Java code for doubly linked list

import java.util.\*;

class Node {

public int data;

public int key;

public Node next;

public Node prev;

public Node(int data, int key) {

this.data = data;

this.key = key;

this.next = null;

this.prev = null;

}

}

public class Main {

//this link always point to first Link

static Node head = null;

//this link always point to last Link

static Node last = null;

static Node current = null;

// is list empty

public static boolean is\_empty() {

return head == null;

}

//display the doubly linked list

public static void print\_list() {

Node ptr = head;

while (ptr != null) {

System.out.println("(" + ptr.key + "," + ptr.data + ")");

ptr = ptr.next;

}

}

//insert link at the first location

public static void insert\_first(int key, int data) {

//create a link

Node link = new Node(data, key);

if (is\_empty()) {

//make it the last link

last = link;

} else {

//update first prev link

head.prev = link;

}

//point it to old first link

link.next = head;

//point first to new first link

head = link;

}

public static void main(String[] args) {

insert\_first(1, 10);

insert\_first(2, 20);

insert\_first(3, 30);

insert\_first(4, 1);

insert\_first(5, 40);

insert\_first(6, 56);

System.out.println("Doubly Linked List: ");

print\_list();

}

}

### **Output**

Doubly Linked List: (6,56)(5,40)(4,1)(3,30)(2,20)(1,10)

## **Doubly Linked List - Insertion at the End**

In this insertion operation, the new input node is added at the end of the doubly linked list; if the list is not empty. The head will be pointed to the new node, if the list is empty.

### **Algorithm**

1. START

2. If the list is empty, add the node to the list and point

the head to it.

3. If the list is not empty, find the last node of the list.

4. Create a link between the last node in the list and the

new node.

5. The new node will point to NULL as it is the new last node.

6. END

### **Example**

Following are the implementations of this operation in JAVA programming languages

import java.util.\*;

class Node {

public int data;

public int key;

public Node next;

public Node prev;

public Node(int data, int key) {

this.data = data;

this.key = key;

this.next = null;

this.prev = null;

}

}

public class Main {

static Node head = null;

static Node last = null;

static Node current = null;

public static boolean isEmpty() {

return head == null;

}

public static void printList() {

Node ptr = head;

while (ptr != null) {

System.out.print("(" + ptr.key + "," + ptr.data + ") ");

ptr = ptr.next;

}

}

public static void insertFirst(int key, int data) {

Node link = new Node(data, key);

if (isEmpty()) {

last = link;

} else {

head.prev = link;

}

link.next = head;

head = link;

}

public static void insertLast(int key, int data) {

Node link = new Node(data, key);

if (isEmpty()) {

last = link;

} else {

last.next = link;

link.prev = last;

}

last = link;

}

public static void main(String[] args) {

insertFirst(1,10);

insertFirst(2,20);

insertFirst(3,30);

insertFirst(4,1);

insertLast(5,40);

insertLast(6,56);

System.out.print("Doubly Linked List: ");

printList();

}

}

### **Output**

Doubly Linked List: (4,1) (3,30) (2,20) (1,10) (5,40) (6,56)

## **Doubly Linked List - Deletion at the Beginning**

This deletion operation deletes the existing first nodes in the doubly linked list. The head is shifted to the next node and the link is removed.

### **Algorithm**

1. START

2. Check the status of the doubly linked list

3. If the list is empty, deletion is not possible

4. If the list is not empty, the head pointer is

shifted to the next node.

5. END

### **Example**

Following are the implementations of this operation in JAVA programming languages −

//Java code for doubly linked list

import java.util.\*;

class Node {

public int data;

public int key;

public Node next;

public Node prev;

public Node(int data, int key) {

this.data = data;

this.key = key;

this.next = null;

this.prev = null;

}

}

public class Main {

//this link always point to first Link

public static Node head = null;

//this link always point to last Link

public static Node last = null;

//this link always point to current Link

public static Node current = null;

//is list empty

public static boolean isEmpty() {

return head == null;

}

//display the doubly linked list

public static void printList() {

Node ptr = head;

while (ptr != null) {

System.out.print("(" + ptr.key + "," + ptr.data + ") ");

ptr = ptr.next;

}

}

//insert link at the first location

public static void insertFirst(int key, int data) {

//create a link

Node link = new Node(data, key);

if (isEmpty()) {

//make it the last link

last = link;

} else {

//update first prev link

head.prev = link;

}

//point it to old first link

link.next = head;

head = link;

}

//delete the first item

public static Node deleteFirst() {

//save reference to first link

Node tempLink = head;

//if only one link

if (head.next == null) {

last = null;

} else {

head.next.prev = null;

}

head = head.next;

//return the deleted link

return tempLink;

}

public static void main(String[] args) {

insertFirst(1, 10);

insertFirst(2, 20);

insertFirst(3, 30);

insertFirst(4, 1);

insertFirst(5, 40);

insertFirst(6, 56);

System.out.print("Doubly Linked List: \n");

printList();

System.out.print("\nList after deleting first record: \n");

deleteFirst();

printList();

}

}

### **Output**

Doubly Linked List:

(6,56) (5,40) (4,1) (3,30) (2,20) (1,10)

List after deleting first record:

(5,40) (4,1) (3,30) (2,20) (1,10)

## **Doubly Linked List - Complete Implementation**

Following are the complete implementations of Doubly Linked List in JAVA programming languages −

class Node {

int data;

int key;

Node next;

Node prev;

public Node(int key, int data) {

this.key = key;

this.data = data;

this.next = null;

this.prev = null;

}

}

class DoublyLinkedList {

Node head;

Node last;

boolean isEmpty() {

return head == null;

}

void displayForward() {

Node ptr = head;

System.out.print("[ ");

while (ptr != null) {

System.out.print("(" + ptr.key + "," + ptr.data + ") ");

ptr = ptr.next;

}

System.out.println("]");

}

void displayBackward() {

Node ptr = last;

System.out.print("[ ");

while (ptr != null) {

System.out.print("(" + ptr.key + "," + ptr.data + ") ");

ptr = ptr.prev;

}

System.out.println("]");

}

void insertFirst(int key, int data) {

Node link = new Node(key, data);

if (isEmpty()) {

last = link;

} else {

head.prev = link;

}

link.next = head;

head = link;

}

void insertLast(int key, int data) {

Node link = new Node(key, data);

if (isEmpty()) {

last = link;

} else {

last.next = link;

link.prev = last;

}

last = link;

}

Node deleteFirst() {

if (isEmpty()) {

return null;

}

Node tempLink = head;

if (head.next == null) {

last = null;

} else {

head.next.prev = null;

}

head = head.next;

return tempLink;

}

Node deleteLast() {

if (isEmpty()) {

return null;

}

Node tempLink = last;

if (head.next == null) {

head = null;

} else {

last.prev.next = null;

}

last = last.prev;

return tempLink;

}

Node delete(int key) {

Node current = head;

Node previous = null;

if (head == null) {

return null;

}

while (current.key != key) {

if (current.next == null) {

return null;

} else {

previous = current;

current = current.next;

}

}

if (current == head) {

head = head.next;

} else {

current.prev.next = current.next;

}

if (current == last) {

last = current.prev;

} else {

current.next.prev = current.prev;

}

return current;

}

boolean insertAfter(int key, int newKey, int data) {

Node current = head;

if (head == null) {

return false;

}

while (current.key != key) {

if (current.next == null) {

return false;

} else {

current = current.next;

}

}

Node newLink = new Node(newKey, data);

if (current == last) {

newLink.next = null;

last = newLink;

} else {

newLink.next = current.next;

current.next.prev = newLink;

}

newLink.prev = current;

current.next = newLink;

return true;

}

}

public class Main {

public static void main(String[] args) {

DoublyLinkedList dll = new DoublyLinkedList();

dll.insertFirst(1, 10);

dll.insertFirst(2, 20);

dll.insertFirst(3, 30);

dll.insertFirst(4, 1);

dll.insertFirst(5, 40);

dll.insertFirst(6, 56);

System.out.println("List (First to Last):");

dll.displayForward();

System.out.println();

System.out.println("List (Last to First):");

dll.displayBackward();

System.out.println("List, after deleting first record:");

dll.deleteFirst();

dll.displayForward();

System.out.println("List, after deleting last record:");

dll.deleteLast();

dll.displayForward();

System.out.println("List, insert after key(4):");

dll.insertAfter(4, 7, 13);

dll.displayForward();

System.out.println("List, after delete key(4):");

dll.delete(4);

dll.displayForward();

}

}

### **Output**

List (First to Last):

[ (6, 56) (5, 40) (4, 1) (3, 30) (2, 20) (1, 10) ]

List (Last to First):

[ (1, 10) (2, 20) (3, 30) (4, 1) (5, 40) (6, 56) ]

List, after deleting first record:

[ (5, 40) (4, 1) (3, 30) (2, 20) (1, 10) ]

List, after deleting last record:

[ (5, 40) (4, 1) (3, 30) (2, 20) ]

List, insert after key(4):

[ (5, 40) (4, 1) (7, 13) (3, 30) (2, 20) ]

List, after delete key (4):

[ (5, 40) (7, 13) (3, 30) (2, 20) ]

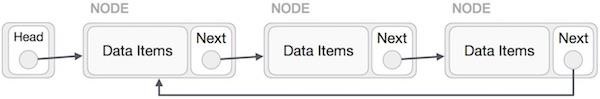
# Circular Linked List Data Structure

## **What is Circular Linked List?**

**Circular Linked List** is a variation of Linked list in which the first element points to the last element and the last element points to the first element. Both Singly Linked List and Doubly Linked List can be made into a circular linked list.

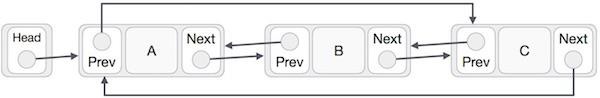
### **Singly Linked List as Circular**

In singly linked list, the next pointer of the last node points to the first node.



### **Doubly Linked List as Circular**

In doubly linked list, the next pointer of the last node points to the first node and the previous pointer of the first node points to the last node making the circular in both directions.



As per the above illustration, following are the important points to be considered.

* The last link's next points to the first link of the list in both cases of singly as well as doubly linked list.
* The first link's previous points to the last of the list in case of doubly linked list.

## **Basic Operations in Circular Linked List**

Following are the important operations supported by a circular list.

* **insert** − Inserts an element at the start of the list.
* **delete** − Deletes an element from the start of the list.
* **display** − Displays the list.

## **Circular Linked List - Insertion Operation**

The insertion operation of a circular linked list only inserts the element at the start of the list. This differs from the usual singly and doubly linked lists as there is no particular starting and ending points in this list. The insertion is done either at the start or after a particular node (or a given position) in the list.

### **Algorithm**

1. START

2. Check if the list is empty

3. If the list is empty, add the node and point the head

to this node

4. If the list is not empty, link the existing head as

the next node to the new node.

5. Make the new node as the new head.

6. END

### **Example**

Following are the implementations of this operation in JAVA programming languages −

//Java program for circular link list

import java.util.\*;

class Node {

int data;

int key;

Node next;

}

public class Main {

static Node head = null;

static Node current = null;

static boolean isEmpty() {

return head == null;

}

//insert link at the first location

static void insertFirst(int key, int data) {

//create a link

Node link = new Node();

link.key = key;

link.data = data;

if (isEmpty()) {

head = link;

head.next = head;

} else {

//point it to old first node

link.next = head;

//point first to new first node

head = link;

}

}

//display the list

static void printList() {

Node ptr = head;

System.out.print("\n[ ");

//start from the beginning

if (head != null) {

while (ptr.next != ptr) {

System.out.print("(" + ptr.key + "," + ptr.data + ") ");

ptr = ptr.next;

}

}

System.out.print(" ]");

}

public static void main(String[] args) {

insertFirst(1, 10);

insertFirst(2, 20);

insertFirst(3, 30);

insertFirst(4, 1);

insertFirst(5, 40);

insertFirst(6, 56);

System.out.print("Circular Linked List: ");

//print list

printList();

}

}

### **Output**

Circular Linked List:

[ (6,56) (5,40) (4,1) (3,30) (2,20) ]

## **Circular Linked List - Deletion Operation**

The Deletion operation in a Circular linked list removes a certain node from the list. The deletion operation in this type of lists can be done at the beginning, or a given position, or at the ending.

### **Algorithm**

1. START

2. If the list is empty, then the program is returned.

3. If the list is not empty, we traverse the list using a

current pointer that is set to the head pointer and create

another pointer previous that points to the last node.

4. Suppose the list has only one node, the node is deleted

by setting the head pointer to NULL.

5. If the list has more than one node and the first node is to

be deleted, the head is set to the next node and the previous

is linked to the new head.

6. If the node to be deleted is the last node, link the preceding

node of the last node to head node.

7. If the node is neither first nor last, remove the node by

linking its preceding node to its succeeding node.

8. END

### **Example**

Following are the implementations of this operation in JAVA programming languages −

//Java program for circular linked list

import java.util.\*;

public class Main {

static class Node {

int data;

int key;

Node next;

}

static Node head = null;

static Node current = null;

static boolean isEmpty() {

return head == null;

}

//insert link at the first location

static void insertFirst(int key, int data) {

//create a link

Node link = new Node();

link.key = key;

link.data = data;

if (isEmpty()) {

head = link;

head.next = head;

} else {

//point it to old first node

link.next = head;

//point first to new first node

head = link;

}

}

//delete first item

static Node deleteFirst() {

//save reference to first link

Node tempLink = head;

if (head.next == head) {

head = null;

return tempLink;

}

//mark next to first link as first

head = head.next;

//return the deleted link

return tempLink;

}

//display the list

static void printList() {

Node ptr = head;

//start from the beginning

if (head != null) {

while (ptr.next != ptr) {

System.out.printf("(%d,%d) ", ptr.key, ptr.data);

ptr = ptr.next;

}

}

}

public static void main(String[] args) {

insertFirst(1, 10);

insertFirst(2, 20);

insertFirst(3, 30);

insertFirst(4, 1);

insertFirst(5, 40);

insertFirst(6, 56);

System.out.print("Circular Linked List: ");

//print list

printList();

deleteFirst();

System.out.print("\nList after deleting the first item: ");

printList();

}

}

### **Output**

Circular Linked List: (6,56) (5,40) (4,1) (3,30) (2,20)

List after deleting the first item: (5,40) (4,1) (3,30) (2,20)

## **Circular Linked List - Displaying the List**

The Display List operation visits every node in the list and prints them all in the output.

### **Algorithm**

1. START

2. Walk through all the nodes of the list and print them

3. END

### **Example**

Following are the implementations of this operation in JAVA programming languages −

//Java program for circular link list

import java.util.\*;

class Node {

int data;

int key;

Node next;

}

public class Main {

static Node head = null;

static Node current = null;

static boolean isEmpty() {

return head == null;

}

//insert link at the first location

static void insertFirst(int key, int data) {

//create a link

Node link = new Node();

link.key = key;

link.data = data;

if (isEmpty()) {

head = link;

head.next = head;

} else {

//point it to old first node

link.next = head;

//point first to new first node

head = link;

}

}

//display the list

static void printList() {

Node ptr = head;

System.out.print("\n[ ");

//start from the beginning

if (head != null) {

while (ptr.next != ptr) {

System.out.print("(" + ptr.key + "," + ptr.data + ") ");

ptr = ptr.next;

}

}

System.out.print(" ]");

}

public static void main(String[] args) {

insertFirst(1, 10);

insertFirst(2, 20);

insertFirst(3, 30);

insertFirst(4, 1);

insertFirst(5, 40);

insertFirst(6, 56);

System.out.print("Circular Linked List: ");

//print list

printList();

}

}

### **Output**

Circular Linked List:

[ (6,56) (5,40) (4,1) (3,30) (2,20) ]

## **Circular Linked List - Complete Implementation**

Following are the complete implementations of Circular Linked List in JAVA programming languages −

class Node {

int data;

int key;

Node next;

Node(int key, int data) {

this.key = key;

this.data = data;

this.next = null;

}

}

public class LinkedList {

private Node head;

private Node current;

boolean isEmpty() {

return head == null;

}

int length() {

int length = 0;

//if list is empty

if (head == null) {

return 0;

}

current = head.next;

while (current != head) {

length++;

current = current.next;

}

return length;

}

//insert link at the first location

void insertFirst(int key, int data) {

//create a link

Node link = new Node(key, data);

if (isEmpty()) {

head = link;

head.next = head;

} else {

//point it to old first node

link.next = head;

//point first to new first node

head = link;

}

}

//delete first item

Node deleteFirst() {

if (head.next == head) {

//save reference to first link

Node tempLink = head;

head = null;

return tempLink;

}

Node tempLink = head;

//mark next to first link as first

head = head.next;

//return the deleted link

return tempLink;

}

//display the list

void printList() {

Node ptr = head;

System.out.print("\n[ ");

//start from the beginning

if (head != null) {

while (ptr.next != ptr) {

System.out.print("(" + ptr.key + "," + ptr.data + ") ");

ptr = ptr.next;

}

}

System.out.print(" ]");

}

public static void main(String[] args) {

LinkedList linkedList = new LinkedList();

linkedList.insertFirst(1, 10);

linkedList.insertFirst(2, 20);

linkedList.insertFirst(3, 30);

linkedList.insertFirst(4, 1);

linkedList.insertFirst(5, 40);

linkedList.insertFirst(6, 56);

System.out.print("Original List: ");

linkedList.printList();

//print list

while (!linkedList.isEmpty()) {

Node temp = linkedList.deleteFirst();

System.out.println("\nDeleted value: (" + temp.key + "," + temp.data + ")");

}

System.out.print("\nList after deleting all items: ");

linkedList.printList();

}

}

### **Output**

Original List:

[ (6,56) (5,40) (4,1) (3,30) (2,20) ]

Deleted value: (6,56)

Deleted value: (5,40)

Deleted value: (4,1)

Deleted value: (3,30)

Deleted value: (2,20)

Deleted value: (1,10)

List after deleting all items:

[ ]

# Stack Data Structure

## **What is a Stack?**

A stack is a **linear data structure** where elements are stored in the LIFO (Last In First Out) principle where the last element inserted would be the first element to be deleted. A stack is an Abstract Data Type (ADT), that is popularly used in most programming languages. It is named stack because it has the similar operations as the real-world stacks, for example − a pack of cards or a pile of plates, etc.

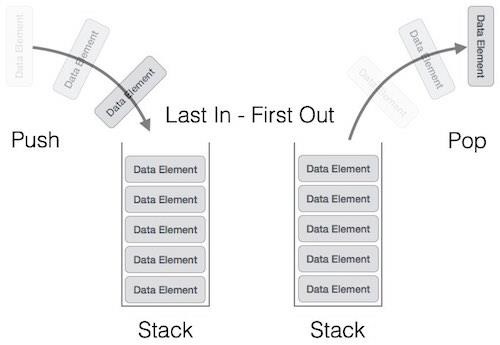


Stack is considered a complex data structure because it uses other data structures for implementation, such as Arrays, Linked lists, etc.

## **Stack Representation**

A stack allows all data operations at one end only. At any given time, we can only access the top element of a stack.

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 on Stacks**

Stack operations are usually performed for initialization, usage and, de-initialization of the stack ADT.

The most fundamental operations in the stack ADT include: push(), pop(), peek(), isFull(), isEmpty(). These are all built-in operations to carry out data manipulation and to check the status of the stack.

Stack uses pointers that always point to the topmost element within the stack, hence called as the **top** pointer.

## **Stack Insertion: push()**

The push() is an operation that inserts elements into the stack. The following is an algorithm that describes the push() operation in a simpler way.

### **Algorithm**

1. Checks if the stack is full.

2. If the stack is full, produces an error and exit.

3. If the stack is not full, increments top to point next

empty space.

4. Adds data element to the stack location, where top

is pointing.

5. Returns success.

### **Example**

Following are the implementations of this operation in JAVA programming languages −

public class Demo{

final static int MAXSIZE = 8;

static int stack[] = new int[MAXSIZE];

static int top = -1;

public static int isfull(){

if(top == MAXSIZE)

return 1;

else

return 0;

}

public static int push(int data){

if(isfull() != 1) {

top = top + 1;

stack[top] = data;

} else {

System.out.print("Could not insert data, Stack is full.\n");

}

return data;

}

public static void main(String[] args){

int i;

push(44);

push(10);

push(62);

push(123);

push(15);

System.out.print("\nStack Elements: ");

// print stack data

for(i = 0; i < MAXSIZE; i++) {

System.out.print(stack[i] + " ");

}

}

}

### **Output**

Stack Elements: 44 10 62 123 15 0 0 0

**Note** − In Java we have used to built-in method **push()** to perform this operation.

## **Stack Deletion: pop()**

The *pop()* is a data manipulation operation which removes elements from the stack. The following pseudo code describes the pop() operation in a simpler way.

### **Algorithm**

1. Checks if the stack is empty.

2. If the stack is empty, produces an error and exit.

3. If the stack is not empty, accesses the data element at

which top is pointing.

4. Decreases the value of top by 1.

5. Returns success.

### **Example**

Following are the implementations of this operation in JAVA programming languages −

public class Demo{

final static int MAXSIZE = 8;

public static int stack[] = new int[MAXSIZE];

public static int top = -1;

/\* Check if the stack is empty \*/

public static int isempty(){

if(top == -1)

return 1;

else

return 0;

}

/\* Check if the stack is full\*/

public static int isfull(){

if(top == MAXSIZE)

return 1;

else

return 0;

}

/\* Function to delete from the stack \*/

public static int pop(){

int data = 0;

if(isempty() != 1) {

data = stack[top];

top = top - 1;

return data;

} else {

System.out.print("Could not retrieve data, Stack is empty.");

}

return data;

}

/\* Function to insert into the stack \*/

public static int push(int data){

if(isfull() != 1) {

top = top + 1;

stack[top] = data;

} else {

System.out.print("\nCould not insert data, Stack is full.\n");

}

return data;

}

/\* Main function \*/

public static void main(String[] args){

push(44);

push(10);

push(62);

push(123);

push(15);

System.out.print("Stack Elements: ");

// print stack data

for(int i = 0; i < MAXSIZE; i++) {

System.out.print(stack[i] + " ");

}

/\*printf("Element at top of the stack: %d\n" ,peek());\*/

System.out.print("\nElements popped: ");

// print stack data

while(isempty() != 1) {

int data = pop();

System.out.print(data + " ");

}

}

}

### **Output**

Stack Elements: 44 10 62 123 15 0 0 0

Elements popped: 15 123 62 10 44

**Note** − In Java we are using the built-in method pop().

## **Retrieving topmost Element from Stack: peek()**

The *peek()* is an operation retrieves the topmost element within the stack, without deleting it. This operation is used to check the status of the stack with the help of the top pointer.

### **Algorithm**

1. START

2. return the element at the top of the stack

3. END

### **Example**

Following are the implementations of this operation in JAVA programming languages −

public class Demo{

final static int MAXSIZE = 8;

public static int stack[] = new int[MAXSIZE];

public static int top = -1;

/\* Check if the stack is full \*/

public static int isfull(){

if(top == MAXSIZE)

return 1;

else

return 0;

}

/\* Function to return the topmost element in the stack \*/

public static int peek(){

return stack[top];

}

/\* Function to insert into the stack \*/

public static int push(int data){

if(isfull() != 1) {

top = top + 1;

stack[top] = data;

} else {

System.out.print("Could not insert data, Stack is full.");

}

return data;

}

/\* Main function \*/

public static void main(String[] args){

push(44);

push(10);

push(62);

push(123);

push(15);

System.out.print("Stack Elements: ");

// print stack data

for(int i = 0; i < MAXSIZE; i++) {

System.out.print(stack[i] + " ");

}

System.out.print("\nElement at top of the stack: " + peek());

}

}

### **Output**

Stack Elements: 44 10 62 123 15 0 0 0

Element at top of the stack: 15

## **Verifying whether the Stack is full: isFull()**

The *isFull()* operation checks whether the stack is full. This operation is used to check the status of the stack with the help of top pointer.

### **Algorithm**

1. START

2. If the size of the stack is equal to the top position of the stack,

the stack is full. Return 1.

3. Otherwise, return 0.

4. END

### **Example**

Following are the implementations of this operation in JAVA programming languages −

import java.io.\*;

public class StackExample {

private int arr[];

private int top;

private int capacity;

StackExample(int size) {

arr = new int[size];

capacity = size;

top = -1;

}

public boolean isEmpty() {

return top == -1;

}

public boolean isFull() {

return top == capacity - 1;

}

public void push(int key) {

if (isFull()) {

System.out.println("Stack is Full\n");

return;

}

arr[++top] = key;

}

public static void main (String[] args) {

StackExample stk = new StackExample(5);

stk.push(1); // inserting 1 in the stack

stk.push(2);

stk.push(3);

stk.push(4);

stk.push(5);

System.out.println("Stack full: " + stk.isFull());

}

}

### **Output**

Stack full: true

## **Verifying whether the Stack is empty: isEmpty()**

The *isEmpty()* operation verifies whether the stack is empty. This operation is used to check the status of the stack with the help of top pointer.

### **Algorithm**

1. START

2. If the top value is -1, the stack is empty. Return 1.

3. Otherwise, return 0.

4. END

### **Example**

Following are the implementations of this operation in JAVA programming languages −

public class Demo{

final static int MAXSIZE = 8;

static int stack[] = new int[MAXSIZE];

static int top = -1;

/\* Check if the stack is empty \*/

public static int isempty(){

if(top == -1)

return 1;

else

return 0;

}

/\* Main function \*/

public static void main(String[] args){

boolean res = isempty() == 1 ? true : false;

System.out.print("Stack empty: " + res);

}

}

### **Output**

Stack empty: true

## **Stack Complete implementation**

Following are the complete implementations of Stack in JAVA programming languages −

public class Demo{

final static int MAXSIZE = 8;

public static int stack[] = new int[MAXSIZE];

public static int top = -1;

/\* Check if the stack is empty \*/

public static int isempty(){

if(top == -1)

return 1;

else

return 0;

}

/\* Check if the stack is full \*/

public static int isfull(){

if(top == MAXSIZE)

return 1;

else

return 0;

}

/\* Function to return the topmost element in the stack \*/

public static int peek(){

return stack[top];

}

/\* Function to delete from the stack \*/

public static int pop(){

int data = 0;

if(isempty() != 1) {

data = stack[top];

top = top - 1;

return data;

} else

System.out.print("Could not retrieve data, Stack is empty.");

return data;

}

/\* Function to insert into the stack \*/

public static int push(int data){

if(isfull() != 1) {

top = top + 1;

stack[top] = data;

} else

System.out.print("Could not insert data, Stack is full.");

return data;

}

/\* Main function \*/

public static void main(String[] args){

push(44);

push(10);

push(62);

push(123);

push(15);

System.out.print("Element at top of the stack: " + peek());

System.out.print("\nElements: ");

// print stack data

while(isempty() != 1) {

int data = pop();

System.out.print(data + " ");

}

boolean res1 = isfull() == 1 ? true : false;

boolean res2 = isempty() == 1 ? true : false;

System.out.print("\nStack full: " + res1);

System.out.print("\nStack empty: " + res2);

}

}

### **Output**

Element at top of the stack: 15

Elements: 15 123 62 10 44

Stack full: false

Stack empty: true

## **Stack Implementation in C**

Click to check the implementation of [Stack Program using C](https://www.tutorialspoint.com/data_structures_algorithms/stack_program_in_c.htm)

# Expression Parsing in Data Structure

An expression is any word or group of words or symbols that generates a value on evaluation. Parsing expression means analyzing the expression for its words or symbols depending on a particular criterion. Expression parsing is a term used in a programming language to evaluate arithmetic and logical expressions.

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 −

|  |  |  |  |
| --- | --- | --- | --- |
| **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 −

Operator Precendence

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 determines 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 |

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

## **Expression Parsing - Complete implementation**

Following are the complete implementations of Expression Parsing (Conversion from infix notations to postfix notations) in C programming languages −

#include<stdio.h>

#include<string.h>

#include<ctype.h>

//char stack

char stack[25];

int top = -1;

void push(char item) {

stack[++top] = item;

}

char pop() {

return stack[top--];

}

//returns precedence of operators

int precedence(char symbol) {

switch(symbol) {

case '+':

case '-':

return 2;

break;

case '\*':

case '/':

return 3;

break;

case '^':

return 4;

break;

case '(':

case ')':

case '#':

return 1;

break;

}

}

//check whether the symbol is operator?

int isOperator(char symbol) {

switch(symbol) {

case '+':

case '-':

case '\*':

case '/':

case '^':

case '(':

case ')':

return 1;

break;

default:

return 0;

}

}

//converts infix expression to postfix

void convert(char infix[],char postfix[]) {

int i,symbol,j = 0;

stack[++top] = '#';

for(i = 0;i<strlen(infix);i++) {

symbol = infix[i];

if(isOperator(symbol) == 0) {

postfix[j] = symbol;

j++;

} else {

if(symbol == '(') {

push(symbol);

} else {

if(symbol == ')') {

while(stack[top] != '(') {

postfix[j] = pop();

j++;

}

pop(); //pop out (.

} else {

if(precedence(symbol)>precedence(stack[top])) {

push(symbol);

} else {

while(precedence(symbol)<=precedence(stack[top])) {

postfix[j] = pop();

j++;

}

push(symbol);

}

}

}

}

}

while(stack[top] != '#') {

postfix[j] = pop();

j++;

}

postfix[j]='\0'; //null terminate string.

}

//int stack

int stack\_int[25];

int top\_int = -1;

void push\_int(int item) {

stack\_int[++top\_int] = item;

}

char pop\_int() {

return stack\_int[top\_int--];

}

//evaluates postfix expression

int evaluate(char \*postfix){

char ch;

int i = 0,operand1,operand2;

while( (ch = postfix[i++]) != '\0') {

if(isdigit(ch)) {

push\_int(ch-'0'); // Push the operand

} else {

//Operator,pop two operands

operand2 = pop\_int();

operand1 = pop\_int();

switch(ch) {

case '+':

push\_int(operand1+operand2);

break;

case '-':

push\_int(operand1-operand2);

break;

case '\*':

push\_int(operand1\*operand2);

break;

case '/':

push\_int(operand1/operand2);

break;

}

}

}

return stack\_int[top\_int];

}

void main() {

char infix[25] = "1\*(2+3)",postfix[25];

convert(infix,postfix);

printf("Infix expression is: %s\n" , infix);

printf("Postfix expression is: %s\n" , postfix);

printf("Evaluated expression is: %d\n" , evaluate(postfix));

}

### **Output**

Infix expression is: 1\*(2+3)

Postfix expression is: 123+\*

Evaluated expression is: 5

## **Expression Parsing Using Stack**

We can use different data structures to implement expression parsing. Check the implementation of [Expression Parsing using Stack](https://www.tutorialspoint.com/data_structures_algorithms/expression_parsing_using_statck.htm)