

Step 1 – START ADD

Step 2 – get values of a & b

Step 3 – $c \leftarrow a + b$

Step 4 – display c

Step 5 – STOP

In design and analysis of algorithms, usually the second method is used to describe an algorithm. It makes it easy for the analyst to analyze the algorithm ignoring all unwanted definitions. He can observe what operations are being used and how the process is flowing.

Writing step numbers, is optional.

We design an algorithm to get a solution of a given problem. A problem can be solved in more than one ways.

one problem many solutions

Hence, many solution algorithms can be derived for a given problem. The next step is to analyze those proposed solution algorithms and implement the best suitable solution.

Algorithm Analysis

Efficiency of an algorithm can be analyzed at two different stages, before implementation and after implementation. They are the following –

A Priori Analysis – This is a theoretical analysis of an algorithm. Efficiency of an algorithm is measured by assuming that all other factors, for example, processor speed, are constant and have no effect on the implementation.

A Posterior Analysis – This is an empirical analysis of an algorithm. The selected algorithm is implemented using programming language. This is then executed on target computer machine. In this analysis, actual statistics like running time and space required, are collected.

We shall learn about a priori algorithm analysis. Algorithm analysis deals with the execution or running time of various operations involved. The running time of an operation can be defined as the number of computer instructions executed per operation.

Algorithm Complexity

Suppose X is an algorithm and n is the size of input data, the time and space used by the algorithm X are the two main factors, which decide the efficiency of X .

Time Factor – Time is measured by counting the number of key operations such as comparisons in the sorting algorithm.

Space Factor – Space is measured by counting the maximum memory space required by the algorithm.

The complexity of an algorithm $f(n)$ gives the running time and/or the storage space required by the algorithm in terms of n as the size of input data.

Space Complexity

Space complexity of an algorithm represents the amount of memory space required by the algorithm in its life cycle. The space required by an algorithm is equal to the sum of the following two components –

A fixed part that is a space required to store certain data and variables, that are independent of the size of the problem. For example, simple variables and constants used, program size, etc.

A variable part is a space required by variables, whose size depends on the size of the problem. For example, dynamic memory allocation, recursion stack space, etc.

Space complexity $S(P)$ of any algorithm P is $S(P) = C + SP(I)$, where C is the fixed part and $S(I)$ is the variable part of the algorithm, which depends on instance characteristic I . Following is a simple example that tries to explain the concept –

Algorithm: SUM(A, B)

Step 1 - START

Step 2 - $C \leftarrow A + B + 10$

Step 3 - Stop

Here we have three variables A, B, and C and one constant. Hence $S(P) = 1 + 3$. Now, space depends on data types of given variables and constant types and it will be multiplied accordingly.

Time Complexity

Time complexity of an algorithm represents the amount of time required by the algorithm to run to completion. Time requirements can be defined as a numerical function $T(n)$, where $T(n)$ can be measured as the number of steps, provided each step consumes constant time.

For example, addition of two n -bit integers takes n steps. Consequently, the total computational time is $T(n) = c * n$, where c is the time taken for the addition of two bits. Here, we observe that $T(n)$ grows linearly as the input size increases.

Data Structures - Asymptotic Analysis

Asymptotic analysis of an algorithm refers to defining the mathematical bound/limit of its run-time performance. Using asymptotic analysis, we can very well conclude the best case, average case, and worst case scenario of an algorithm.

Asymptotic analysis is input bound i.e., if there's no input to the algorithm, it is concluded to work in a constant time. Other than the "input" all other factors are considered constant.

Asymptotic analysis refers to computing the running time of any operation in mathematical units of computation. For example, the running time of one operation is computed as $f(n)$ and may be for another operation it is computed as $g(n^2)$. This means the first operation running time will increase linearly with the increase in n and the running time of the second operation will increase exponentially when n increases. Similarly, the running time of both operations will be nearly the same if n is significantly small.

Usually, the time required by an algorithm falls under three types –

Best Case – Minimum time required for program execution.

Average Case – Average time required for program execution.

Worst Case – Maximum time required for program execution.

Asymptotic Notations

Following are the commonly used asymptotic notations to calculate the running time complexity of an algorithm.

O Notation

Ω Notation

θ Notation

Big Oh Notation, O

The notation $O(n)$ is the formal way to express the upper bound of an algorithm's running time. It measures the worst case time complexity or the longest amount of time an algorithm can possibly take to complete.

Big O Notation

For example, for a function $f(n)$

$O(f(n)) = \{ g(n) : \text{there exists } c > 0 \text{ and } n_0 \text{ such that } f(n) \leq c \cdot g(n) \text{ for all } n > n_0. \}$

Omega Notation, Ω

The notation $\Omega(n)$ is the formal way to express the lower bound of an algorithm's running time. It measures the best case time complexity or the best amount of time an algorithm can possibly take to complete.

Omega Notation

For example, for a function $f(n)$

$\Omega(f(n)) \geq \{ g(n) : \text{there exists } c > 0 \text{ and } n_0 \text{ such that } g(n) \leq c \cdot f(n) \text{ for all } n > n_0. \}$

Theta Notation, θ

The notation $\theta(n)$ is the formal way to express both the lower bound and the upper bound of an algorithm's running time. It is represented as follows –

Theta Notation

$\theta(f(n)) = \{ g(n) \text{ if and only if } g(n) = O(f(n)) \text{ and } g(n) = \Omega(f(n)) \text{ for all } n > n_0. \}$

Common Asymptotic Notations

Following is a list of some common asymptotic notations –

constant	–	$O(1)$
logarithmic	–	$O(\log n)$
linear	–	$O(n)$
$n \log n$	–	$O(n \log n)$
quadratic	–	$O(n^2)$
cubic	–	$O(n^3)$
polynomial	–	$n^{O(1)}$
exponential	–	$2^{O(n)}$

Data Structures - Greedy Algorithms

An algorithm is designed to achieve optimum solution for a given problem. In greedy algorithm approach, decisions are made from the given solution domain. As being greedy, the closest solution that seems to provide an optimum solution is chosen.

Greedy algorithms try to find a localized optimum solution, which may eventually lead to globally optimized solutions. However, generally greedy algorithms do not provide globally optimized solutions.

Counting Coins

This problem is to count to a desired value by choosing the least possible coins and the greedy approach forces the algorithm to pick the largest possible coin. If we are provided coins of ₹ 1, 2, 5 and 10 and we are asked to count ₹ 18 then the greedy procedure will be –

1 – Select one ₹ 10 coin, the remaining count is 8

2 – Then select one ₹ 5 coin, the remaining count is 3

3 – Then select one ₹ 2 coin, the remaining count is 1

4 – And finally, the selection of one ₹ 1 coins solves the problem

Though, it seems to be working fine, for this count we need to pick only 4 coins. But if we slightly change the problem then the same approach may not be able to produce the same optimum result.

For the currency system, where we have coins of 1, 7, 10 value, counting coins for value 18 will be absolutely optimum but for count like 15, it may use more coins than necessary. For example, the greedy approach will use $10 + 1 + 1 + 1 + 1 + 1$, total 6 coins. Whereas the same problem could be solved by using only 3 coins ($7 + 7 + 1$)

Hence, we may conclude that the greedy approach picks an immediate optimized solution and may fail where global optimization is a major concern.

Examples

Most networking algorithms use the greedy approach. Here is a list of few of them –

Travelling Salesman Problem

Prim's Minimal Spanning Tree Algorithm

Kruskal's Minimal Spanning Tree Algorithm

Dijkstra's Minimal Spanning Tree Algorithm

Graph - Map Coloring

Graph - Vertex Cover

Knapsack Problem

Job Scheduling Problem

There are lots of similar problems that uses the greedy approach to find an optimum solution.

Data Structures - Divide and Conquer

In divide and conquer approach, the problem in hand, is divided into smaller sub-problems and then each problem is solved independently. When we keep on dividing the subproblems into even smaller sub-problems, we may eventually reach a stage where no more division is possible. Those "atomic" smallest possible sub-problem (fractions) are solved. The

solution of all sub-problems is finally merged in order to obtain the solution of an original problem.

Divide and Conquer

Broadly, we can understand divide-and-conquer approach in a three-step process.

Divide/Break

This step involves breaking the problem into smaller sub-problems. Sub-problems should represent a part of the original problem. This step generally takes a recursive approach to divide the problem until no sub-problem is further divisible. At this stage, sub-problems become atomic in nature but still represent some part of the actual problem.

Conquer/Solve

This step receives a lot of smaller sub-problems to be solved. Generally, at this level, the problems are considered 'solved' on their own.

Merge/Combine

When the smaller sub-problems are solved, this stage recursively combines them until they formulate a solution of the original problem. This algorithmic approach works recursively and conquer & merge steps work so close that they appear as one.

Examples

The following computer algorithms are based on divide-and-conquer programming approach –

Merge Sort

Quick Sort

Binary Search

Strassen's Matrix Multiplication

Closest pair (points)

There are various ways available to solve any computer problem, but the mentioned are a good example of divide and conquer approach.

Data Structures - Dynamic Programming

Dynamic programming approach is similar to divide and conquer in breaking down the problem into smaller and yet smaller possible sub-problems. But unlike, divide and conquer, these sub-problems are not solved independently. Rather, results of these smaller sub-problems are remembered and used for similar or overlapping sub-problems.

Dynamic programming is used where we have problems, which can be divided into similar sub-problems, so that their results can be re-used. Mostly, these algorithms are used for optimization. Before solving the in-hand sub-problem, dynamic algorithm will try to examine the results of the previously solved sub-problems. The solutions of sub-problems are combined in order to achieve the best solution.

So we can say that –

The problem should be able to be divided into smaller overlapping sub-problem.

An optimum solution can be achieved by using an optimum solution of smaller sub-problems.

Dynamic algorithms use Memoization.

Comparison

In contrast to greedy algorithms, where local optimization is addressed, dynamic algorithms are motivated for an overall optimization of the problem.

In contrast to divide and conquer algorithms, where solutions are combined to achieve an overall solution, dynamic algorithms use the output of a smaller sub-problem and then try to optimize a bigger sub-problem. Dynamic algorithms use Memoization to remember the output of already solved sub-problems.

Example

The following computer problems can be solved using dynamic programming approach –

Fibonacci number series

Knapsack problem

Tower of Hanoi

All pair shortest path by Floyd-Warshall

Shortest path by Dijkstra

Project scheduling

Dynamic programming can be used in both top-down and bottom-up manner. And of course, most of the times, referring to the previous solution output is cheaper than recomputing in terms of CPU cycles.

Data Structures & Algorithm Basic Concepts

This chapter explains the basic terms related to data structure.

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 Algorithms - Arrays

Array is a container which can hold a fix number of items and these items should be of the same type. Most of the data structures make use of arrays to implement their algorithms. 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.

Array Representation

Arrays can be declared in various ways in different languages. For illustration, let's take C array declaration.

Array Declaration

Arrays can be declared in various ways in different languages. For illustration, let's take C array declaration.

Array Representation

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

Index starts with 0.

Array length is 10 which means it can store 10 elements.

Each element can be accessed via its index. For example, we can fetch an element at index 6 as 9.

Basic Operations

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.

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
---------	---

Traverse Operation

This operation is to traverse through the elements of an array.

Example

Following program traverses and prints the elements of an array:

```
#include <stdio.h>

main() {
    int LA[] = {1,3,5,7,8};
    int item = 10, k = 3, n = 5;
    int i = 0, j = n;
    printf("The original array elements are :\n");
    for(i = 0; i<n; i++) {
        printf("LA[%d] = %d \n", i, LA[i]);
    }
}
```

When we compile and execute the above program, it produces the following result –

Output

The original array elements are :

LA[0] = 1

LA[1] = 3

LA[2] = 5

LA[3] = 7

LA[4] = 8

Insertion Operation

Insert operation is to insert one or more data elements into an array. Based on the requirement, a new element can be added at the beginning, end, or any given index of array.

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

Example

Following is the implementation of the above algorithm –

Live Demo

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

```
main() {  
    int LA[] = {1,3,5,7,8};  
    int item = 10, k = 3, n = 5;  
    int i = 0, j = n;  
  
    printf("The original array elements are :\n");  
  
    for(i = 0; i<n; i++) {  
        printf("LA[%d] = %d \n", i, LA[i]);  
    }  
  
    n = n + 1;
```

```
while( j >= k) {  
    LA[j+1] = LA[j];  
    j = j - 1;  
}
```

```
LA[k] = item;
```

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

```
for(i = 0; i<n; i++) {  
    printf("LA[%d] = %d \n", i, LA[i]);  
}  
}
```

When we compile and execute the above program, it produces the following result –

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 insertion :

$LA[0] = 1$

$LA[1] = 3$

$LA[2] = 5$

$LA[3] = 10$

$LA[4] = 7$

$LA[5] = 8$

For other variations of array insertion operation [click here](#)

Deletion Operation

Deletion refers to removing an existing element from the array and re-organizing all elements of an array.

Algorithm

Consider LA is a linear array with N elements and K is a positive integer such that $K \leq N$. 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 is the implementation of the above algorithm –

Live Demo

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

```
void main() {
```

```
    int LA[] = {1,3,5,7,8};
```

```
    int k = 3, n = 5;
```

```
    int i, j;
```

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

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

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

```
    }
```

```
    j = k;
```

```
    while( j < n) {
```

```
        LA[j-1] = LA[j];
```

```
        j = j + 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]);
}
}
```

When we compile and execute the above program, it produces the following result –

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 deletion :

LA[0] = 1

LA[1] = 3

LA[2] = 7

LA[3] = 8

Search Operation

You can perform a search for an array element based on its value or its index.

Algorithm

Consider LA is a linear array with N elements and K is a positive integer such that $K \leq N$. 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 is the implementation of the above algorithm –

Live Demo

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

```
void main() {  
    int LA[] = {1,3,5,7,8};  
    int item = 5, n = 5;  
    int i = 0, j = 0;  
  
    printf("The original array elements are :\n");
```



```
for(i = 0; i<n; i++) {  
    printf("LA[%d] = %d \n", i, LA[i]);  
}
```

```
while( j < n){  
    if( LA[j] == item ) {  
        break;  
    }
```

```
    j = j + 1;  
}
```

```
printf("Found element %d at position %d\n", item, j+1);  
}
```

When we compile and execute the above program, it produces the following result –

Output

The original array elements are :

LA[0] = 1

LA[1] = 3

LA[2] = 5

LA[3] = 7

LA[4] = 8

Found element 5 at position 3

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 $K \leq N$. Following is the algorithm to update an element available at the Kth position of LA.

1. Start
2. Set $LA[K-1] = \text{ITEM}$
3. Stop

Example

Following is the implementation of the above algorithm –

Live Demo

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

```
void main() {  
    int LA[] = {1,3,5,7,8};  
    int k = 3, n = 5, item = 10;  
    int i, j;  
  
    printf("The original array elements are :\n");
```

```
for(i = 0; i<n; i++) {  
    printf("LA[%d] = %d \n", i, LA[i]);  
}  
  
LA[k-1] = item;  
  
printf("The array elements after updation :\n");  
  
for(i = 0; i<n; i++) {  
    printf("LA[%d] = %d \n", i, LA[i]);  
}  
}
```

When we compile and execute the above program, it produces the following result –

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

Data Structure and Algorithms - Linked List

A linked list is a sequence of data structures, which are connected together via links.

Linked List is a sequence of links which contains items. Each link contains a connection to another link. Linked list is the second most-used data structure after array. Following are the important terms to understand the concept of 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.

LinkedList – A Linked List contains the connection link to the first link called First.

Linked List Representation

Linked list can be visualized as a chain of nodes, where every node points to the next node.

Linked List

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

Linked List contains a link element called first.

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.

Last link carries a link as null to mark the end of the list.

Types of Linked List

Following are the various types of linked list.

Simple Linked List – Item navigation is forward only.

Doubly Linked List – Items can be navigated forward and backward.

Circular Linked List – Last item contains link of the first element as next and the first element has a link to the last element as previous.

Basic Operations

Following are the basic operations supported by a list.

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.

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.

Linked List Insertion

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 –

Linked List Insertion

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

```
LeftNode.next -> NewNode;
```

Linked List Insertion

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

Linked List Insertion

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 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.

Linked List Deletion

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

```
LeftNode.next -> TargetNode.next;
```

Linked List Deletion

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;

Linked List Deletion

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.

Linked List Deletion

Reverse 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.

Linked List Reverse Operation

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 –

Linked List Reverse Operation

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.

Linked List Reverse Operation

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.

Linked List Reverse Operation

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

Linked List Reverse Operation

The linked list is now reversed. To see linked list implementation in C programming language, please [click here](#).

Data Structure - Doubly Linked List

Doubly Linked List is a variation of Linked list in which navigation is possible in both ways, either forward and 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.

LinkedList – A Linked List contains the connection link to the first link called First and to the last link called Last.

Doubly Linked List Representation

Doubly Linked List

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 two link fields called next and prev.

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

Following are the basic operations supported by a list.

Insertion – Adds an element at the beginning of the list.

Deletion – Deletes an element at the beginning of the list.

Insert Last – Adds an element at the end of the list.

Delete Last – Deletes an element from the end of the list.

Insert After – Adds an element after an item 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.

Insertion Operation

Following code demonstrates the insertion operation at the beginning of a doubly linked list.

Example

```
//insert link at the first location
```

```
void insertFirst(int key, int data) {
```

```
    //create a link
```

```
    struct node *link = (struct node*) malloc(sizeof(struct node));
```

```
    link->key = key;
```

```
    link->data = data;
```

```

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;

//point first to new first link
head = link;
}

```

Deletion Operation

Following code demonstrates the deletion operation at the beginning of a doubly linked list.

Example

```

//delete first item
struct node* deleteFirst() {

    //save reference to first link
    struct 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;
}
```

Insertion at the End of an Operation

Following code demonstrates the insertion operation at the last position of a doubly linked list.

Example

```
//insert link at the last location
void insertLast(int key, int data) {

    //create a link
    struct node *link = (struct node*) malloc(sizeof(struct node));
    link->key = key;
    link->data = data;
```

```
if(isEmpty()) {  
    //make it the last link  
    last = link;  
} else {  
    //make link a new last link  
    last->next = link;  
  
    //mark old last node as prev of new link  
    link->prev = last;  
}  
  
//point last to new last node  
last = link;  
}  
  
To see the implementation in C programming language, please  
click here.
```

Data Structure - 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.

Singly Linked List as Circular Linked List

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.

Doubly Linked List as Circular Linked List

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

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.

Insertion Operation

Following code demonstrates the insertion operation in a circular linked list based on single linked list.

Example

```
//insert link at the first location
void insertFirst(int key, int data) {
    //create a link
    struct node *link = (struct node*) malloc(sizeof(struct 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;
    }
}
```



```
}
```

Deletion Operation

Following code demonstrates the deletion operation in a circular linked list based on single linked list.

```
//delete first item
struct node * deleteFirst() {
    //save reference to first link
    struct 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 List Operation

Following code demonstrates the display list operation in a circular linked list.

```
//display the list
void printList() {
    struct node *ptr = head;
    printf("\n[ ");

    //start from the beginning
    if(head != NULL) {
        while(ptr->next != ptr) {
            printf("(%d,%d) ",ptr->key,ptr->data);
            ptr = ptr->next;
        }
    }

    printf(" ]");
}
```

To know about its implementation in C programming language, please click [here](#).

Data Structure and Algorithms - Stack

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.

Stack Example

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.

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 –

Stack Representation

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 actually removing it.

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 —

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.

Stack Push Operation

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

$\text{top} \leftarrow \text{top} + 1$

$\text{stack}[\text{top}] \leftarrow \text{data}$

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.

Step 4 – Decreases the value of top by 1.

Step 5 – Returns success.

Stack Pop Operation

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  $\leftarrow$  stack[top]
```

```
    top  $\leftarrow$  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");
}
}

```

For a complete stack program in C programming language, please click [here](#).

Data Structure - 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 prefixed 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 postfixed 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 Precedence

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

To see the implementation in C programming language, please [click here](#).

Data Structure and Algorithms - Queue

Queue is an abstract data structure, somewhat similar to 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.

Queue Example

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 –

Queue Example

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.

In queue, we always dequeue (or access) data, pointed by front pointer and while enqueueing (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;
```

```
}
```

```
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.

Insert Operation

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

```

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.

Remove Operation

Algorithm for dequeue operation

procedure dequeue

```
if queue is empty
    return underflow
end if
```

```
data = queue[front]
```

```
front ← front + 1
```

```
return true
```

```
end procedure
```

Implementation of dequeue() in C programming language –

Example

```
int dequeue() {  
    if(isempty())  
        return 0;  
  
    int data = queue[front];  
    front = front + 1;  
  
    return data;  
}
```

For a complete Queue program in C programming language, please click [here](#).

Data Structure and Algorithms Linear Search

Linear search is a very simple search algorithm. In this type of search, a sequential search is made over all items one by one. Every item is checked and if a match is found then that particular item is returned, otherwise the search continues till the end of the data collection.