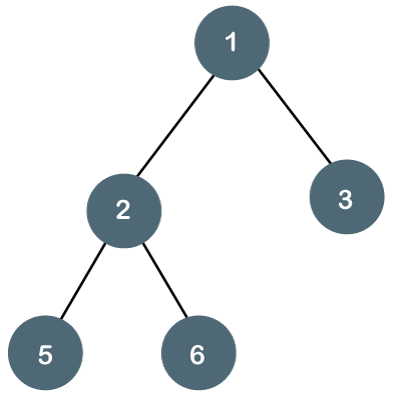
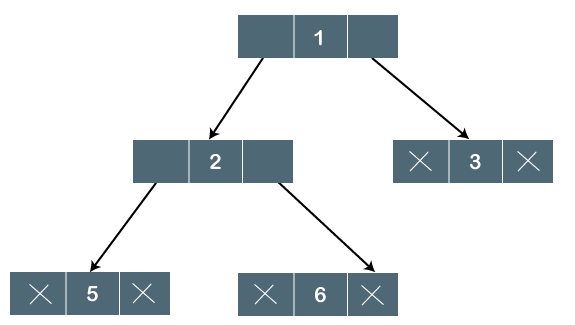
Binary Tree

The Binary tree means that the node can have maximum two children. Here, binary name itself suggests that 'two'; therefore, each node can have either 0, 1 or 2 children.

**Let's understand the binary tree through an example.**



The above tree is a binary tree because each node contains the utmost two children. The logical representation of the above tree is given below:



In the above tree, node 1 contains two pointers, i.e., left and a right pointer pointing to the left and right node respectively. The node 2 contains both the nodes (left and right node); therefore, it has two pointers (left and right). The nodes 3, 5 and 6 are the leaf nodes, so all these nodes contain **NULL** pointer on both left and right parts.

### **Properties of Binary Tree**

* At each level of i, the maximum number of nodes is 2i.
* The height of the tree is defined as the longest path from the root node to the leaf node. The tree which is shown above has a height equal to 3. Therefore, the maximum number of nodes at height 3 is equal to (1+2+4+8) = 15. In general, the maximum number of nodes possible at height h is (20 + 21 + 22+….2h) = 2h+1 -1.
* The minimum number of nodes possible at height h is equal to **h+1**.
* If the number of nodes is minimum, then the height of the tree would be maximum. Conversely, if the number of nodes is maximum, then the height of the tree would be minimum.

If there are 'n' number of nodes in the binary tree.

**The minimum height can be computed as:**

As we know that,

n = 2h+1 -1

n+1 = 2h+1

Taking log on both the sides,

log2(n+1) = log2(2h+1)

log2(n+1) = h+1

**h = log2(n+1) - 1**

**The maximum height can be computed as:**

As we know that,

n = h+1

**h= n-1**

**Binary Tree (Array implementation)**

 [Read](javascript:void(0))

 [Discuss](javascript:void(0))

 [Courses](javascript:void(0))

 [Practice](javascript:void(0))

Given an array that represents a tree in such a way that array indexes are values in tree nodes and array values give the parent node of that particular index (or node). The value of the root node index would always be -1 as there is no parent for root. Construct the standard linked representation of given Binary Tree from this given representation. [Do refer in order to understand how to construct binary tree from given parent array representation](https://www.geeksforgeeks.org/construct-a-binary-tree-from-parent-array-representation/).

**Ways to represent:**

Trees can be represented in two ways as listed below:

1. Dynamic Node Representation [(Linked Representation).](https://www.geeksforgeeks.org/binary-tree-set-1-introduction/)
2. Array Representation (Sequential Representation).

Now, we are going to talk about the sequential representation of the trees.  In order to represent a tree using an array, the numbering of nodes can start either from 0–(n-1) or 1– n, consider the below illustration as follows:

**Illustration:**

A(0)

/ \

B(1) C(2)

/ \ \

D(3) E(4) F(6)

OR,

A(1)

/ \

B(2) C(3)

/ \ \

D(4) E(5) F(7)

**Procedure:**

***Note:****father, left\_son and right\_son are the values of indices of the array.*

**Case 1:** (0—n-1)

if (say)father=p;

then left\_son=(2\*p)+1;

and right\_son=(2\*p)+2;

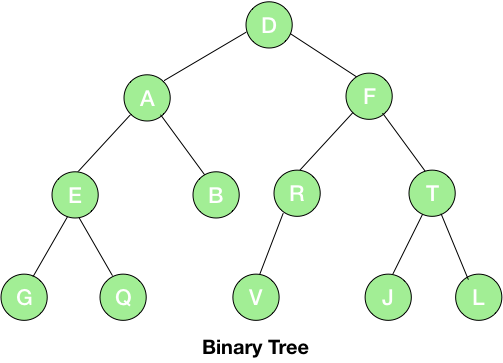
**Case 2:** 1—n

if (say)father=p;

then left\_son=(2\*p);

and right\_son=(2\*p)+1;

This post is about implementing a binary tree in C using an array. You can visit [Binary Trees](https://www.codesdope.com/blog/article/binary-trees) for the concepts behind binary trees. We will use array representation to make a binary tree in C and then we will implement **inorder**, **preorder** and **postorder** traversals in both the representations and then finish this post by **making a function to calculate the height of the tree**.



We will use the above tree for the array representation. As discussed in the post [Binary Trees](https://www.codesdope.com/blog/article/binary-trees), the array we will use to represent the above tree will be:

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| \0 | D | A | F | E | B | R | T | G | Q | \0 | \0 | V | \0 | J | L |

Let’s start with the first step and make an array.

**#include** *<stdio.h>*

*/\**

*D*

*/ \*

*/ \*

*/ \*

*A F*

*/ \ / \*

*/ \ / \*

*E B R T*

*/ \ / / \*

*G Q V J L*

*\*/*

**int** complete\_node = 15;

**char** tree[] = {'\0', 'D', 'A', 'F', 'E', 'B', 'R', 'T', 'G', 'Q', '\0', '\0', 'V', '\0', 'J', 'L'};

**int** main()

{

**return** 0;

}

int complete\_node = 15 – It is just a variable to keep the total number of nodes if the tree given is a complete binary tree.

char tree[ ] – It is the array which is storing the entire binary tree.

Now, we are ready with a binary tree and the next step is to make the functions to traverse over this binary tree. But before doing so, we need two more functions to get the left and the right children of any node. So, let’s make these functions first.

## **Function to Get Right and Left Children**

**int** get\_right\_child(**int** index)

{

**if**(tree[index]!='\0' && ((2\*index)+1)<=complete\_node)

**return** (2\*index)+1;

**return** -1;

}

tree[index]!='\0' – Checking if the current node is not null.

(2\*index)+1)<=complete\_node – We know that the right child of node ‘i’ is given by (2\*i)+1 but this value must lie within the number of elements in the array. And this condition checks the same.

return (2\*index)+1 – If both the above conditions are satisfied then we are returning the index of the right child.

return -1 – In case of failure, we are returning -1, a negative value to represent failure.

Similarly, we can make a function to get the left child of the tree by using the property that the left child of node ‘i’ of a complete binary tree is given by 2\*i.

**int** get\_left\_child(**int** index)

{

**if**(tree[index]!='\0' && (2\*index)<=complete\_node)

**return** 2\*index;

**return** -1;

}

We are now ready with the functions. So, let’s make the functions to traverse the binary tree.

## **Traversals in Binary Tree**

### **Preorder Traversal**

**void** preorder(**int** index)

{

**if**(index>0 && tree[index]!='\0')

{

printf(" %c\n",tree[index]);

preorder(get\_left\_child(index));

preorder(get\_right\_child(index));

}

}

if(index>0 && tree[index]!='\0') – We are first checking if the index given is valid or not because the functions we made above to get the children of the tree return -1 for an invalid result so, the condition index>0 checks the same. The condition tree[index]!='\0' checks if the node is not null.

printf(" %c\n",tree[index]) – We are first visiting the root.

preorder(get\_left\_child(index)) – Then we are visiting the left child

preorder(get\_right\_child(index)) – And at last, the right child.

These are explained in the [Binary Trees](https://www.codesdope.com/blog/article/binary-trees) post.

Similarly, we can write functions for the postorder and inorder traversals.

### **Postorder Traversal**

**void** postorder(**int** index)

{

**if**(index>0 && tree[index]!='\0')

{

postorder(get\_left\_child(index));

postorder(get\_right\_child(index));

printf(" %c\n",tree[index]);

}

}

### **Inorder Traversal**

**void** inorder(**int** index)

{

**if**(index>0 && tree[index]!='\0')

{

inorder(get\_left\_child(index));

printf(" %c\n",tree[index]);

inorder(get\_right\_child(index));

}

}

Let’s test these function in the main function.

**#include** *<stdio.h>*

*/\**

*D*

*/ \*

*/ \*

*/ \*

*A F*

*/ \ / \*

*/ \ / \*

*E B R T*

*/ \ / / \*

*G Q V J L*

*\*/*

*// variable to store maximum number of nodes*

**int** complete\_node = 15;

*// array to store the tree*

**char** tree[] = {'\0', 'D', 'A', 'F', 'E', 'B', 'R', 'T', 'G', 'Q', '\0', '\0', 'V', '\0', 'J', 'L'};

**int** get\_right\_child(**int** index)

{

*// node is not null*

*// and the result must lie within the number of nodes for a complete binary tree*

**if**(tree[index]!='\0' && ((2\*index)+1)<=complete\_node)

**return** (2\*index)+1;

*// right child doesn't exist*

**return** -1;

}

**int** get\_left\_child(**int** index)

{

*// node is not null*

*// and the result must lie within the number of nodes for a complete binary tree*

**if**(tree[index]!='\0' && (2\*index)<=complete\_node)

**return** 2\*index;

*// left child doesn't exist*

**return** -1;

}

**void** preorder(**int** index)

{

*// checking for valid index and null node*

**if**(index>0 && tree[index]!='\0')

{

printf(" %c ",tree[index]); *// visiting root*

preorder(get\_left\_child(index)); *//visiting left subtree*

preorder(get\_right\_child(index)); *//visiting right subtree*

}

}

**void** postorder(**int** index)

{

*// checking for valid index and null node*

**if**(index>0 && tree[index]!='\0')

{

postorder(get\_left\_child(index)); *//visiting left subtree*

postorder(get\_right\_child(index)); *//visiting right subtree*

printf(" %c ",tree[index]); *//visiting root*

}

}

**void** inorder(**int** index)

{

*// checking for valid index and null node*

**if**(index>0 && tree[index]!='\0')

{

inorder(get\_left\_child(index)); *//visiting left subtree*

printf(" %c ",tree[index]); *//visiting root*

inorder(get\_right\_child(index)); *// visiting right subtree*

}

}

**int** main()

{

printf("Preorder:\n");

preorder(1);

printf("\nPostorder:\n");

postorder(1);

printf("\nInorder:\n");

inorder(1);

    printf("\n");

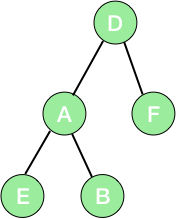
**return** 0;

}

**Output**:  
Preorder:  
 D  A  E  G  Q  B  F  R  V  T  J  L   
Postorder:  
 G  Q  E  B  A  V  R  J  L  T  F  D   
Inorder:  
 G  E  Q  A  B  D  V  R  F  J  T  L

This post is about implementing a binary tree in C. You can visit [Binary Trees](https://www.codesdope.com/blog/article/binary-trees) for the concepts behind binary trees. We will use linked representation to make a binary tree in C and then we will implement **inorder**, **preorder** and **postorder** traversals and then finish this post by **making a function to calculate the height of the tree**.

The [binary tree](https://www.codesdope.com/blog/article/binary-trees) we will be using in this post is:



So, let’s make a node using a structure in C.

**struct** node

{

**char** data; *//node will store character*

**struct** node \*right\_child; *// right child*

**struct** node \*left\_child; *// left child*

};

char data – The data which we are going to store in this node is of character type.

struct node \*right\_child – ‘right\_child’ is a pointer to the node (our defined structure). Thus, we will use this to link the current node with its right child.

struct node \*left\_child – Similarly, we will use this to link the current node with its left child.

Now, we have a structure of our node and we need a function to create new nodes to make a tree. So, let’s write it.

**struct** node\* new\_node(**char** data)

{

**struct** node \*p; *// node*

p = malloc(**sizeof**(**struct** node)); *// allocating space to p*

p->data = data; *// assinging data to p*

p->left\_child = NULL; *// making children NULL*

p->right\_child = NULL;

**return**(p); *// returning the newly made node*

}

struct node \*p – ‘\*p’ is a node and ‘p’ is a pointer to the node. It means that p will store the address of the node. Please note that we have just declared a node here and no memory has been yet assigned to store data. Thus, the next task is to allocate the memory to the node and we will do it using the [malloc function](https://www.codesdope.com/c-dynamic-memory/).

p = malloc(sizeof(struct node)) – We are allocating a space in the memory to the node and storing the address of the memory in the variable ‘p’.

p->data = data – Now, we have our node and we also have a space in memory allocated to it. Here, we are just storing data in it.

p->left\_child = NULL – We are making the left child of the node null.

p->right\_child = NULL – Similarly, we also made the right child null.

return(p) – We are just returning the address of the node we created.

## **Traversals in a Binary Tree**

Now, we have made our node and a function to add new nodes. Thus, the next task is to make the tree described in the above picture and implement **inorder**, **postorder** and **preorder** traversals to it. So, let’s first make the tree in the main function.

**int** main()

{

**struct** node \*root; *//new structure*

root = new\_node('D'); *// making a new node*

*/\**

*\_\_\_\_*

*| D |*

*|\_\_\_\_|*

*\*/*

root->left\_child = new\_node('A'); *//left child of root*

*/\**

*\_\_\_\_*

*| D |*

*/|\_\_\_\_|*

*\_\_\_\_ /*

*| A |*

*|\_\_\_\_|*

*\*/*

root->right\_child = new\_node('F'); *//right child of root*

*/\**

*\_\_\_\_*

*| D |*

*/|\_\_\_\_|\*

*\_\_\_\_ / \_\\_\_\_*

*| A | | F |*

*|\_\_\_\_| |\_\_\_\_|*

*\*/*

root->left\_child->left\_child = new\_node('E'); *// new node*

*/\**

*\_\_\_\_*

*| D |*

*/|\_\_\_\_|\*

*\_\_\_\_ / \_\\_\_\_*

*| A | | F |*

*/|\_\_\_\_| |\_\_\_\_|*

*\_\_\_\_/*

*| E |*

*|\_\_\_\_|*

*\*/*

root->left\_child->right\_child = new\_node('B'); *// new node*

*/\**

*\_\_\_\_*

*| D |*

*/|\_\_\_\_|\*

*\_\_\_\_ / \_\\_\_\_*

*| A | | F |*

*/|\_\_\_\_|\ |\_\_\_\_|*

*\_\_\_\_/ \_\\_\_\_*

*| E | | B |*

*|\_\_\_\_| |\_\_\_\_\_|*

*\*/*

}

You can learn the concepts behind the traversals from the post [Binary Trees](https://www.codesdope.com/blog/article/binary-trees).

### **Preorder Traversal**

**void** preorder(**struct** node \*root)

{

**if**(root!=NULL) *// checking if the root is not null*

{

printf(" %c ", root->data); *// printing data at root*

preorder(root->left\_child); *// visiting left child*

preorder(root->right\_child); *// visiting right child*

}

}

In preorder traversal, we first visit the root and then the left subtree and lastly the right subtree. We are doing the same here.

printf(" %c ", root->data) – We are first visiting the root (of the main tree or subtree) or the current node then we will visit its left subtree and then the right subtree.

preorder(root->left\_child) – Then we are visiting the left subtree.

preorder(root->right\_child) – And lastly the right subtree.

### **Postorder Traversal**

**void** postorder(**struct** node \*root)

{

**if**(root!=NULL) *// checking if the root is not null*

{

postorder(root->left\_child); *// visiting left child*

postorder(root->right\_child); *// visiting right child*

printf(" %c ", root->data); *// printing data at root*

}

}

In postorder traversal, we first visit the left subtree and then the right and lastly the node.

### **Inorder Traversal**

**void** inorder(**struct** node \*root)

{

**if**(root!=NULL) *// checking if the root is not null*

{

inorder(root->left\_child); *// visiting left child*

printf(" %c ", root->data); *// printing data at root*

inorder(root->right\_child);*// visiting right child*

}

}

We first visit the left subtree and then root and lastly the right subtree in inorder traversal.

## **Height of a Node or Binary Tree**

Height of a node is 1+ greater heights among the left subtree and the right subtree. Also, the height of a leaf node or a null node is 0. Thus, we first write a function to identify a leaf node.

### **Function to Identify Leaves in Binary Tree**

**int** is\_leaf(**struct** node \*a)

{

**if**(a->right\_child==NULL && a->left\_child==NULL)

**return** 1;

**return** 0;

}

Checking for a leaf node is simple. If both the children of a node are null then it is a leaf node. We are checking the same with – if(a->right\_child==NULL && a->left\_child==NULL).

Now, we are ready to write a function to get the height of any node of tree.

*// function to return maximum of two numbers*

**int** get\_max(**int** a, **int** b)

{

**return** (a>b) ? a : b;

}

*//function to get the height of a tree or node*

**int** get\_height(**struct** node \*a)

{

**if**(a==NULL || is\_leaf(a)) *// height will be 0 if the node is leaf or null*

**return** 0;

*//height of a node will be 1+ greater among height of right subtree and height of left subtree*

**return**(get\_max(get\_height(a->left\_child), get\_height(a->right\_child)) + 1);

}

‘get\_max’ is a function to determine the greater number of the two numbers passed to it.

‘get\_height’ is the function to calculate the height of the tree. We are first checking for a null node or leaf node with if(a==NULL || is\_leaf(a)). In both cases, the height will be 0. Else, the height will be 1+maximum among the heights of left and the right subtrees – get\_max(get\_height(a->left\_child), get\_height(a->right\_child)) + 1.

Let’s implement the above concepts and see the result.

**#include** *<stdio.h>*

**#include** *<stdlib.h>*

*/\**

*D*

*/ \*

*/ \*

*/ \*

*A F*

*/ \*

*/ \*

*E B*

*\*/*

**struct** node

{

**char** data; *//node will store character*

**struct** node \*right\_child; *// right child*

**struct** node \*left\_child; *// left child*

};

**struct** node\* new\_node(**char** data)

{

**struct** node \*p; *// node*

p = malloc(**sizeof**(**struct** node)); *// allocating space to p*

p->data = data; *// assinging data to p*

p->left\_child = NULL; *// making children NULL*

p->right\_child = NULL;

**return**(p); *// returning the newly made node*

}

**void** preorder(**struct** node \*root)

{

**if**(root!=NULL) *// checking if the root is not null*

{

printf(" %c ", root->data); *// printing data at root*

preorder(root->left\_child); *// visiting left child*

preorder(root->right\_child); *// visiting right child*

}

}

**void** postorder(**struct** node \*root)

{

**if**(root!=NULL) *// checking if the root is not null*

{

postorder(root->left\_child); *// visiting left child*

postorder(root->right\_child); *// visiting right child*

printf(" %c ", root->data); *// printing data at root*

}

}

**void** inorder(**struct** node \*root)

{

**if**(root!=NULL) *// checking if the root is not null*

{

inorder(root->left\_child); *// visiting left child*

printf(" %c ", root->data); *// printing data at root*

inorder(root->right\_child);*// visiting right child*

}

}

**int** is\_leaf(**struct** node \*a)

{

**if**(a->right\_child==NULL && a->left\_child==NULL)

**return** 1;

**return** 0;

}

*// function to return maximum of two numbers*

**int** get\_max(**int** a, **int** b)

{

**return** (a>b) ? a : b;

}

*//function to get the height of a tree or node*

**int** get\_height(**struct** node \*a)

{

**if**(a==NULL || is\_leaf(a)) *// height will be 0 if the node is leaf or null*

**return** 0;

*//height of a node will be 1+ greater among height of right subtree and height of left subtree*

**return**(get\_max(get\_height(a->left\_child), get\_height(a->right\_child)) + 1);

}

**int** main()

{

**struct** node \*root; *//new structure*

root = new\_node('D'); *// making a new node*

*/\**

*\_\_\_\_*

*| D |*

*|\_\_\_\_|*

*\*/*

root->left\_child = new\_node('A'); *//left child of root*

*/\**

*\_\_\_\_*

*| D |*

*/|\_\_\_\_|*

*\_\_\_\_ /*

*| A |*

*|\_\_\_\_|*

*\*/*

root->right\_child = new\_node('F'); *//right child of root*

*/\**

*\_\_\_\_*

*| D |*

*/|\_\_\_\_|\*

*\_\_\_\_ / \_\\_\_\_*

*| A | | F |*

*|\_\_\_\_| |\_\_\_\_|*

*\*/*

root->left\_child->left\_child = new\_node('E'); *// new node*

*/\**

*\_\_\_\_*

*| D |*

*/|\_\_\_\_|\*

*\_\_\_\_ / \_\\_\_\_*

*| A | | F |*

*/|\_\_\_\_| |\_\_\_\_|*

*\_\_\_\_/*

*| E |*

*|\_\_\_\_|*

*\*/*

root->left\_child->right\_child = new\_node('B'); *// new node*

*/\**

*\_\_\_\_*

*| D |*

*/|\_\_\_\_|\*

*\_\_\_\_ / \_\\_\_\_*

*| A | | F |*

*/|\_\_\_\_|\ |\_\_\_\_|*

*\_\_\_\_/ \_\\_\_\_*

*| E | | B |*

*|\_\_\_\_| |\_\_\_\_\_|*

*\*/*

preorder(root);

printf("\n");

postorder(root);

printf("\n");

inorder(root);

printf("\n");

printf("%d\n", get\_height(root));

**return** 0;

}

**Output**:  
 D  A  E  B  F   
 E  B  A  F  D   
 E  A  B  D  F   
2

# Graph representation

A graph is a data structure that consist a sets of vertices (called nodes) and edges. There are two ways to store Graphs into the computer's memory:

* **Sequential representation** (or, Adjacency matrix representation)
* **Linked list representation** (or, Adjacency list representation)

In sequential representation, an adjacency matrix is used to store the graph. Whereas in linked list representation, there is a use of an adjacency list to store the graph.

In this tutorial, we will discuss each one of them in detail.

Now, let's start discussing the ways of representing a graph in the data structure.

Sequential representation

In sequential representation, there is a use of an adjacency matrix to represent the mapping between vertices and edges of the graph. We can use an adjacency matrix to represent the undirected graph, directed graph, weighted directed graph, and weighted undirected graph.

If adj[i][j] = w, it means that there is an edge exists from vertex i to vertex j with weight w.

An entry Aij in the adjacency matrix representation of an undirected graph G will be 1 if an edge exists between Vi and Vj. If an Undirected Graph G consists of n vertices, then the adjacency matrix for that graph is n x n, and the matrix A = [aij] can be defined as -

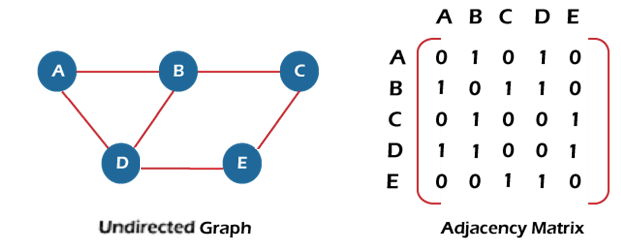
aij = 1 {if there is a path exists from Vi to Vj}

aij = 0 {Otherwise}

It means that, in an adjacency matrix, 0 represents that there is no association exists between the nodes, whereas 1 represents the existence of a path between two edges.

If there is no self-loop present in the graph, it means that the diagonal entries of the adjacency matrix will be 0.

Now, let's see the adjacency matrix representation of an undirected graph.



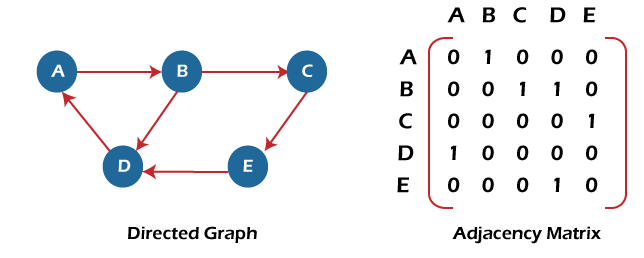
In the above figure, an image shows the mapping among the vertices (A, B, C, D, E), and this mapping is represented by using the adjacency matrix.

There exist different adjacency matrices for the directed and undirected graph. In a directed graph, an entry Aij will be 1 only when there is an edge directed from Vi to Vj.

Adjacency matrix for a directed graph

In a directed graph, edges represent a specific path from one vertex to another vertex. Suppose a path exists from vertex A to another vertex B; it means that node A is the initial node, while node B is the terminal node.

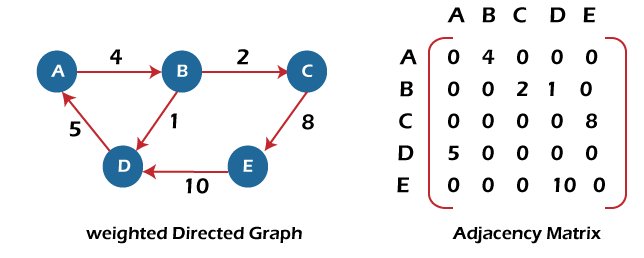
Consider the below-directed graph and try to construct the adjacency matrix of it.



In the above graph, we can see there is no self-loop, so the diagonal entries of the adjacent matrix are 0.

**Adjacency matrix for a weighted directed graph**

It is similar to an adjacency matrix representation of a directed graph except that instead of using the '1' for the existence of a path, here we have to use the weight associated with the edge. The weights on the graph edges will be represented as the entries of the adjacency matrix. We can understand it with the help of an example. Consider the below graph and its adjacency matrix representation. In the representation, we can see that the weight associated with the edges is represented as the entries in the adjacency matrix.



In the above image, we can see that the adjacency matrix representation of the weighted directed graph is different from other representations. It is because, in this representation, the non-zero values are replaced by the actual weight assigned to the edges.

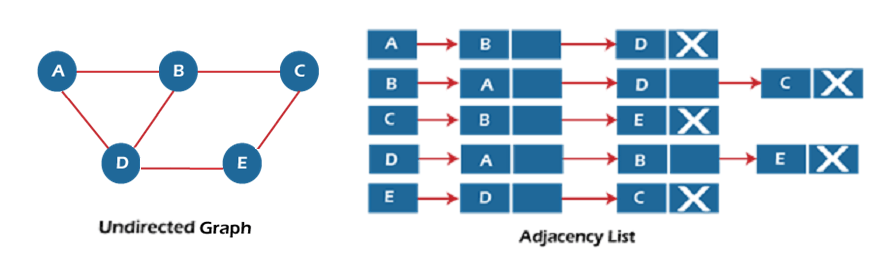
Adjacency matrix is easier to implement and follow. An adjacency matrix can be used when the graph is dense and a number of edges are large.

Though, it is advantageous to use an adjacency matrix, but it consumes more space. Even if the graph is sparse, the matrix still consumes the same space.

Linked list representation

An adjacency list is used in the linked representation to store the Graph in the computer's memory. It is efficient in terms of storage as we only have to store the values for edges.

Let's see the adjacency list representation of an undirected graph.

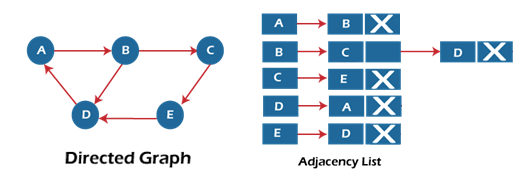


In the above figure, we can see that there is a linked list or adjacency list for every node of the graph. From vertex A, there are paths to vertex B and vertex D. These nodes are linked to nodes A in the given adjacency list.

An adjacency list is maintained for each node present in the graph, which stores the node value and a pointer to the next adjacent node to the respective node. If all the adjacent nodes are traversed, then store the NULL in the pointer field of the last node of the list.

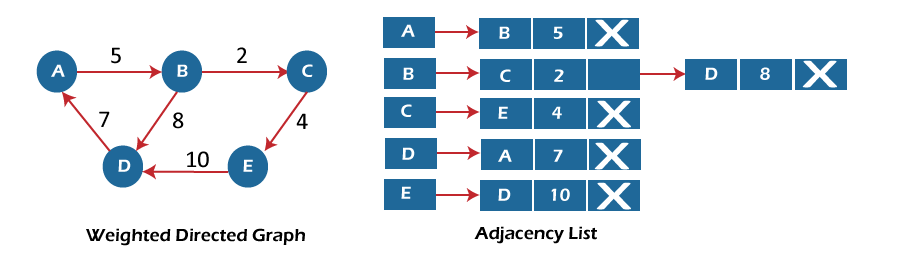
The sum of the lengths of adjacency lists is equal to twice the number of edges present in an undirected graph.

Now, consider the directed graph, and let's see the adjacency list representation of that graph.



For a directed graph, the sum of the lengths of adjacency lists is equal to the number of edges present in the graph.

Now, consider the weighted directed graph, and let's see the adjacency list representation of that graph.



In the case of a weighted directed graph, each node contains an extra field that is called the weight of the node.

In an adjacency list, it is easy to add a vertex. Because of using the linked list, it also saves space.

Implementation of adjacency matrix representation of Graph

Now, let's see the implementation of adjacency matrix representation of graph in C.

In this program, there is an adjacency matrix representation of an undirected graph. It means that if there is an edge exists from vertex A to vertex B, there will also an edge exists from vertex B to vertex A.

Here, there are four vertices and five edges in the graph that are non-directed.

1. /\* Adjacency Matrix representation of an undirected graph in C \*/
3. #include <stdio.h>
4. #define V 4 /\* number of vertices in the graph \*/
6. /\* function to initialize the matrix to zero \*/
7. **void** init(**int** arr[][V]) {
8. **int** i, j;
9. **for** (i = 0; i < V; i++)
10. **for** (j = 0; j < V; j++)
11. arr[i][j] = 0;
12. }
14. /\* function to add edges to the graph \*/
15. **void** insertEdge(**int** arr[][V], **int** i, **int** j) {
16. arr[i][j] = 1;
17. arr[j][i] = 1;
18. }
20. /\* function to print the matrix elements \*/
21. **void** printAdjMatrix(**int** arr[][V]) {
22. **int** i, j;
23. **for** (i = 0; i < V; i++) {
24. printf("%d: ", i);
25. **for** (j = 0; j < V; j++) {
26. printf("%d ", arr[i][j]);
27. }
28. printf("\n");
29. }
30. }
32. **int** main() {
33. **int** adjMatrix[V][V];
35. init(adjMatrix);
36. insertEdge(adjMatrix, 0, 1);
37. insertEdge(adjMatrix, 0, 2);
38. insertEdge(adjMatrix, 1, 2);
39. insertEdge(adjMatrix, 2, 0);
40. insertEdge(adjMatrix, 2, 3);
42. printAdjMatrix(adjMatrix);
44. **return** 0;
45. }

**Output:**

After the execution of the above code, the output will be -

# Graph RepresentationApplications, Advantages and Disadvantages of Graph

 [Read](javascript:void(0))

 [Discuss](javascript:void(0))

 [Courses](javascript:void(0))

 [Practice](javascript:void(0))

[**Graph**](https://www.geeksforgeeks.org/graph-data-structure-and-algorithms/) is a non-linear data structure that contains nodes (vertices) and edges. A graph is a collection of set of vertices and edges (formed by connecting two vertices). A graph is defined as **G = {V, E}** where **V** is the set of vertices and **E** is the set of edges.

 Graphs can be used to model a wide variety of real-world problems, including social networks, transportation networks, and communication networks. They can be represented in various ways, such as by a set of vertices and a set of edges, or by a matrix or an adjacency list. The two most common types of graphs are directed and undirected graphs.

Terminologies of Graphs:

.An edge is one of the two primary units used to form graphs. Each edge has two ends, which are vertices to which it is attached.

.If two vertices are endpoints of the same edge, they are adjacent.

.A vertex’s outgoing edges are directed edges that point to the origin.

.A vertex’s incoming edges are directed edges that point to the vertex’s destination.

.The total number of edges occurring to a vertex in a graph is its degree.

.A vertex with an in-degree of zero is referred to as a source vertex, while one with an out-degree of zero is known as sink vertex.

.A path is a set of alternating vertices and edges, with each vertex connected by an edge.

.The path that starts and finishes at the same vertex is known as a cycle.

.A path with unique vertices is called a simple path.

.A spanning subgraph that is also a tree is known as a spanning tree.

.A connected component is the unconnected graph’s most connected subgraph.

.A bridge, which is an edge of removal, would sever the graph.

.Forest is a graph without a cycle.

**Graph Representation**:

Graph can be represented in the following ways:

1. **Set Representation:**Set representation of a graph involves two sets: Set of vertices **V = {V1, V2, V3, V4}** and set of edges**E = {{V1, V2}, {V2, V3}, {V3, V4}, {V4, V1}}**. This representation is efficient for memory but does not allow parallel edges.
2. **Sequential Representation:**This representation of a graph can be represented by means of matrices: Adjacency Matrix, Incidence matrix and Path matrix.
   * **Adjacency Matrix:** This matrix includes information about the adjacent nodes. Here, **aij = 1** if there is an edge from**Vi** to **Vj**otherwise **0**. It is a matrix of order **V×V**.
   * **Incidence Matrix:** This matrix includes information about the incidence of edges on the nodes. Here, **aij = 1** if the**jth** edge **Ej**is incident on **ith** vertex **Vi**otherwise**0**. It is a matrix of order**V×E.**
   * **Path Matrix:**This matrix includes information about the simple path between two vertices. Here, **Pij = 1** if there is a path from **Vi**to **Vj** otherwise **0**. It is also called as reachability matrix of graph **G**.
3. **Linked Representation:**This representation gives the information about the nodes to which a specific node is connected i.e. adjacency lists. This representation gives the adjacency lists of the vertices with the help of array and linked lists. In the adjacency lists, the vertices which are connected with the specific vertex are arranged in the form of lists which is connected to that vertex.

**Real-Time Applications of Graph:**

* **Social media analysis**: Social media platforms generate vast amounts of data in real-time, which can be analyzed using graphs to identify trends, sentiment, and key influencers. This can be useful for marketing, customer service, and reputation management.
* **Network monitoring:** Graphs can be used to monitor network traffic in real-time, allowing network administrators to identify potential bottlenecks, security threats, and other issues. This is critical for ensuring the smooth operation of complex networks.
* **Financial trading:** Graphs can be used to analyze real-time financial data, such as stock prices and market trends, to identify patterns and make trading decisions. This is particularly important for high-frequency trading, where even small delays can have a significant impact on profits.
* **Internet of Things (IoT) management:**IoT devices generate vast amounts of data in real-time, which can be analyzed using graphs to identify patterns, optimize performance, and detect anomalies. This is important for managing large-scale IoT deployments.
* **Autonomous vehicles:** Graphs can be used to model the real-time environment around autonomous vehicles, allowing them to navigate safely and efficiently. This requires real-time data from sensors and other sources, which can be processed using graph algorithms.
* **Disease surveillance**: Graphs can be used to model the spread of infectious diseases in real-time, allowing health officials to identify outbreaks and implement effective containment strategies. This is particularly important during pandemics or other public health emergencies.
* The best example of graphs in the real world is Facebook. Each person on Facebook is a node and is connected through edges. Thus, A is a friend of B. B is a friend of C, and so on.

**Advantages of Graph:**

* **Representing complex data:** Graphs are effective tools for representing complex data, especially when the relationships between the data points are not straightforward. They can help to uncover patterns, trends, and insights that may be difficult to see using other methods.
* **Efficient data processing:** Graphs can be processed efficiently using graph algorithms, which are specifically designed to work with graph data structures. This makes it possible to perform complex operations on large datasets quickly and effectively.
* **Network analysis:**Graphs are commonly used in network analysis to study relationships between individuals or organizations, as well as to identify important nodes and edges in a network. This is useful in a variety of fields, including social sciences, business, and marketing.
* **Pathfinding:**Graphs can be used to find the shortest path between two points, which is a common problem in computer science, logistics, and transportation planning.
* **Visualization**: Graphs are highly visual, making it easy to communicate complex data and relationships in a clear and concise way. This makes them useful for presentations, reports, and data analysis.
* **Machine** **learning**: Graphs can be used in machine learning to model complex relationships between variables, such as in recommendation systems or fraud detection.
* Graphs are used in computer science to depict the flow of computation.
* Users on Facebook are referred to as vertices, and if they are friends, there is an edge connecting them. The Friend Suggestion system on Facebook is based on graph theory.
* You come across the Resources Allocation Graph in the Operating System, where each process and resource are regarded vertically. Edges are drawn from resources to assigned functions or from the requesting process to the desired resources. A stalemate will develop if this results in the establishment of a cycle.
* Web pages are referred to as vertices on the World Wide Web. Suppose there is a link from page A to page B that can represent an edge. this application is an illustration of a directed graph.
* Graph transformation systems manipulate graphs in memory using rules, Graph databases store and query graph-structured data in a transaction-safe, perment manner.

**Disadvantages of Graph:**

* **Limited representation:**Graphs can only represent relationships between objects, and not their properties or attributes. This means that in order to fully understand the data, it may be necessary to supplement the graph with additional information.
* **Difficulty in interpretation:** Graphs can be difficult to interpret, especially if they are large or complex. This can make it challenging to extract meaningful insights from the data, and may require advanced analytical techniques or domain expertise.
* **Scalability issue**s: As the number of nodes and edges in a graph increases, the processing time and memory required to analyze it also increases. This can make it difficult to work with large or complex graphs.
* **Data quality issues**: Graphs are only as good as the data they are based on, and if the data is incomplete, inconsistent, or inaccurate, the graph may not accurately reflect the relationships between obje

A **stack** is a linear data structure that follows the **LIFO (Last In First Out)** principle. It is used to store and retrieve data in a specific order. [The stack data structure can be implemented using an array or a linked list 1](https://www.simplilearn.com/tutorials/data-structure-tutorial/stacks-in-data-structures)[2](https://www.programiz.com/dsa/stack).

In an array-based implementation, the stack is formed by using the array. All the operations regarding the stack are performed using arrays. Adding an element into the top of the stack is referred to as push operation. Push operation involves following two steps:

1. Incrementing the value of top.
2. [Placing the new element in the position pointed to by top](https://www.simplilearn.com/tutorials/data-structure-tutorial/stacks-in-data-structures) [3](https://www.javatpoint.com/ds-array-implementation-of-stack).

Removing an element from the top of the stack is referred to as pop operation. Pop operation involves following two steps:

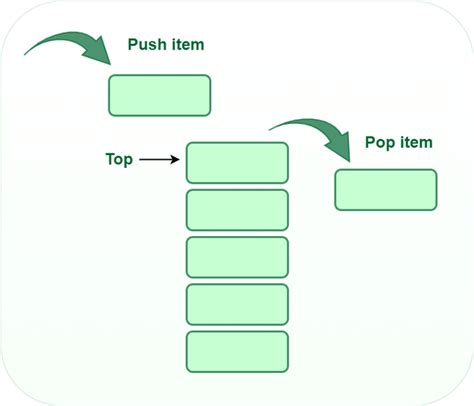
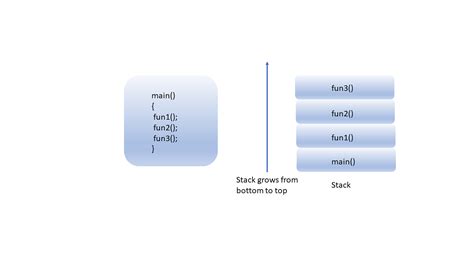
1. Returning the element pointed to by top.
2. [Decrementing the value of top](https://www.simplilearn.com/tutorials/data-structure-tutorial/stacks-in-data-structures) [3](https://www.javatpoint.com/ds-array-implementation-of-stack).

The basic operations that allow us to perform different actions on a stack are:

* Push: Add an element to the top of a stack.
* Pop: Remove an element from the top of a stack.
* IsEmpty: Check if the stack is empty.
* IsFull: Check if the stack is full.
* [Peek: Get the value of the top element without removing it](https://www.simplilearn.com/tutorials/data-structure-tutorial/stacks-in-data-structures) [2](https://www.programiz.com/dsa/stack)[4](https://www.geeksforgeeks.org/introduction-to-stack-data-structure-and-algorithm-tutorials/).

[Stacks are used in many applications such as reversing a word, checking for balanced parentheses, and evaluating postfix expressions](https://www.simplilearn.com/tutorials/data-structure-tutorial/stacks-in-data-structures) [1](https://www.simplilearn.com/tutorials/data-structure-tutorial/stacks-in-data-structures).

I hope this helps!



First, we need to define the maximum size of the stack and declare an array to hold the stack elements. The stack is represented by an array named stack with a fixed size, defined by the constant MAX. We also need an integer variable top to keep track of the index of the topmost element in the stack.

Example:

#define MAX 10

int stack[MAX];

int top = -1;

In the above code, the top variable is initialized to -1 to indicate that the stack is empty. When the top is set to -1, it means there are no elements in the stack, and the first element has not been added yet.

When you push the first element onto the stack, the top is incremented using the pre-increment operator (++top). Therefore, when the first element is added, the top will become 0, pointing to the first position in the array.

## Step 2: Implement the push Function

The push() function adds an element to the top of the stack. Before pushing the element, we need to check if the stack is full. If the stack is full (top == MAX - 1), we display an error message. Otherwise, we increment the top and add the element to the stack.

Example:

void push(int value) {

if (top == MAX - 1) {

printf("Error: Stack overflow!\n");

return;

}

stack[++top] = value;

}

The condition top == MAX - 1 checks if the stack is full. Since array indices start at 0, the maximum index in the array is MAX - 1. If top is equal to MAX - 1, it means that the stack is at its maximum capacity, and no more elements can be added (pushed) onto the stack. In this case, a stack overflow error message is displayed to inform the user that the stack is full.

The stack[++top] = value; increments the top index and stores the new element (the value variable) at the updated position in the stack array.

## Step 3: Implement the pop Function

The pop() function removes the top element from the stack and returns it. Before popping the element, we must check if the stack is empty. If the stack is empty (top == -1), we display an error message. Otherwise, we return the top element and decrement the top.

int pop() {

if (top == -1) {

printf("Error: Stack underflow!\n");

return -1;

}

return stack[top--];

}

## Step 4: Implement the main Function

The main() function provides a simple menu-driven interface for users to interact with the stack. The user can choose between pushing an element, popping an element, or exiting the program. Error messages are displayed in case of a stack overflow or stack underflow.

Example:

#include <stdio.h>

#include <stdlib.h>

#define MAX 10

int stack[MAX];

int top = -1;

void push(int value) {

if (top == MAX - 1) {

printf("Error: Stack overflow!\n");

return;

}

stack[++top] = value;

}

int pop() {

if (top == -1) {

printf("Error: Stack underflow!\n");

return -1;

}

return stack[top--];

}

int main() {

int choice, value;

while (1) {

printf("1. Push\n2. Pop\n3. Exit\n");

printf("Enter your choice: ");

scanf("%d", &choice);

switch (choice) {

case 1:

printf("Enter value to push: ");

scanf("%d", &value);

push(value);

break;

case 2:

value = pop();

if (value != -1) {

printf("Popped value: %d\n", value);

}

break;

case 3:

default:

printf("Invalid choice!\n");

}

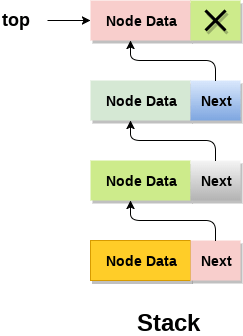
}

return 0;

# exit(0); Linked list implementation of stack

Instead of using array, we can also use linked list to implement stack. Linked list allocates the memory dynamically. However, time complexity in both the scenario is same for all the operations i.e. push, pop and peek.

In linked list implementation of stack, the nodes are maintained non-contiguously in the memory. Each node contains a pointer to its immediate successor node in the stack. Stack is said to be overflown if the space left in the memory heap is not enough to create a node.



The top most node in the stack always contains null in its address field. Lets discuss the way in which, each operation is performed in linked list implementation of stack.

## **Adding a node to the stack (Push operation)**

Adding a node to the stack is referred to as **push** operation. Pushing an element to a stack in linked list implementation is different from that of an array implementation. In order to push an element onto the stack, the following steps are involved.

1. Create a node first and allocate memory to it.
2. If the list is empty then the item is to be pushed as the start node of the list. This includes assigning value to the data part of the node and assign null to the address part of the node.
3. If there are some nodes in the list already, then we have to add the new element in the beginning of the list (to not violate the property of the stack). For this purpose, assign the address of the starting element to the address field of the new node and make the new node, the starting node of the list.

**Time Complexity : o(1)**

1. **void** push ()
2. {
3. **int** val;
4. struct node \*ptr =(struct node\*)malloc(sizeof(struct node));
5. **if**(ptr == NULL)
6. {
7. printf("not able to push the element");
8. }
9. **else**
10. {
11. printf("Enter the value");
12. scanf("%d",&val);
13. **if**(head==NULL)
14. {
15. ptr->val = val;
16. ptr -> next = NULL;
17. head=ptr;
18. }
19. **else**
20. {
21. ptr->val = val;
22. ptr->next = head;
23. head=ptr;
25. }
26. printf("Item pushed");
28. }
29. }

## **Deleting a node from the stack (POP operation)**

Deleting a node from the top of stack is referred to as **pop** operation. Deleting a node from the linked list implementation of stack is different from that in the array implementation. In order to pop an element from the stack, we need to follow the following steps :

* 1. **Check for the underflow condition:** The underflow condition occurs when we try to pop from an already empty stack. The stack will be empty if the head pointer of the list points to null.
  2. **Adjust the head pointer accordingly:** In stack, the elements are popped only from one end, therefore, the value stored in the head pointer must be deleted and the node must be freed. The next node of the head node now becomes the head node.

**Time Complexity : o(n)**

### **C implementation**

* 1. **void** pop()
  2. {
  3. **int** item;
  4. struct node \*ptr;
  5. **if** (head == NULL)
  6. {
  7. printf("Underflow");
  8. }
  9. **else**
  10. {
  11. item = head->val;
  12. ptr = head;
  13. head = head->next;
  14. free(ptr);
  15. printf("Item popped");
  17. }
  18. }

## **Display the nodes (Traversing)**

Displaying all the nodes of a stack needs traversing all the nodes of the linked list organized in the form of stack. For this purpose, we need to follow the following steps.

* 1. Copy the head pointer into a temporary pointer.
  2. Move the temporary pointer through all the nodes of the list and print the value field attached to every node.

**Time Complexity : o(n)**

### **C Implementation**

* 1. **void** display()
  2. {
  3. **int** i;
  4. struct node \*ptr;
  5. ptr=head;
  6. **if**(ptr == NULL)
  7. {
  8. printf("Stack is empty\n");
  9. }
  10. **else**
  11. {
  12. printf("Printing Stack elements \n");
  13. **while**(ptr!=NULL)
  14. {
  15. printf("%d\n",ptr->val);
  16. ptr = ptr->next;
  17. }
  18. }
  19. }

# Queue

1. A queue can be defined as an ordered list which enables insert operations to be performed at one end called **REAR** and delete operations to be performed at another end called **FRONT**.

2. Queue is referred to be as First In First Out list.

3. For example, people waiting in line for a rail ticket form a queue.



## **Applications of Queue**

Due to the fact that queue performs actions on first in first out basis which is quite fair for the ordering of actions. There are various applications of queues discussed as below.

1. Queues are widely used as waiting lists for a single shared resource like printer, disk, CPU.
2. Queues are used in asynchronous transfer of data (where data is not being transferred at the same rate between two processes) for eg. pipes, file IO, sockets.
3. Queues are used as buffers in most of the applications like MP3 media player, CD player, etc.
4. Queue are used to maintain the play list in media players in order to add and remove the songs from the play-list.
5. Queues are used in operating systems for handling interrupts.

## **Using Array:**

1. To implement a queue in C using an array, first define the queue's maximum size and declare an array of that size. The front and back integers were respectively set to 1. The front variable represents the front element of the queue, and the back variable represents the back element.
2. Before pushing the new element to the final position of the queue, we need to increase the back variable by 1. The queue is now full and no other additional elements can be added when the back position is reaching the queue's maximum capacity. We add an element to the front of the queue and increase the front variable by one only to remove an element from the queue. If the front and rear positions are equal and no more components can be deleted, hence we can say that the queue is empty.
3. Below is an instance of a queue written in C that makes use of an array:
4. #define MAX\_SIZE 100
6. **int** queue[MAX\_SIZE];
7. **int** front = -1;
8. **int** rear = -1;
10. **void** enqueue(**int** element) {
11. **if** (rear == MAX\_SIZE - 1) {
12. printf("Queue is full");
13. **return**;
14. }
15. **if** (front == -1) {
16. front = 0;
17. }
18. rear++;
19. queue[rear] = element;
20. }
22. **int** dequeue() {
23. **if** (front == -1 || front > rear) {
24. printf("Queue is empty");
25. **return** -1;
26. }
27. **int** element = queue[front];
28. front++;
29. **return** element;
30. }
32. **int** main() {
33. enqueue(10);
34. enqueue(20);
35. enqueue(30);
36. printf("%d ", dequeue());
37. printf("%d ", dequeue());
38. printf("%d ", dequeue());
39. printf("%d ", dequeue());
40. **return** 0;
41. }

The output of the code will be:

**Output:**

10 20 30 Queue is empty-1

**Explanation:**

1. First, we enqueue three elements 10, 20 and 30 into the queue.
2. Then, we dequeue and print the front element of the queue, which is 10.
3. Next, we dequeue and print the front element of the queue again, which is 20.
4. Then, we dequeue and print the front element of the queue again, which is 30.
5. Finally, we make a dequeue from an empty queue that outputs "Queue is empty" and returns -1.

Using Linked List:

Another alternate approach to constructing a queue in the programming language C is using a linked list. Each of the nodes in the queue is expressed using this method by a node which contains the element value and a pointer to the following node in the queue. In order to keep track of the first and last nodes in the queue, respectively, front and rear pointers are used.

We establish a new node with the element value and set its next pointer to NULL to add an element to the queue. To the new node, we set the front and rear pointers if the queue is empty. If not, we update the rear pointer and set the old rear node's next pointer to point to the new node.

When deleting a node from a queue, the preceding node is deleted first, then the front pointer is updated to the subsequent node in the queue, and finally the memory that the removed node was occupying is released. If the front pointer is NULL after removal, the queue is empty.

Here's an example of a queue implemented in C using a linked list:

**C Programming Language:**

1. #include<stdio.h>
2. #include <stdlib.h>
4. **struct** Node {
5. **int** data;
6. **struct** Node\* next;
7. };
9. **struct** Node\* front = NULL;
10. **struct** Node\* rear = NULL;
12. **void** enqueue(**int** element) {
13. **struct** Node\* new\_node = (**struct** Node\*)malloc(**sizeof**(**struct** Node));
14. new\_node->data = element;
15. new\_node->next = NULL;
16. **if** (front == NULL && rear == NULL) {
17. front = rear = new\_node;
18. **return**;
19. }
20. rear->next = new\_node;
21. rear = new\_node;
22. }
24. **int** dequeue() {
25. **if** (front == NULL) {
26. printf("Queue is empty");
27. **return** -1;
28. }
29. **struct** Node\* temp = front;
30. **int** element = temp->data;
31. **if** (front == rear) {
32. front = rear = NULL;
33. }
34. **else** {
35. front = front->next;
36. }
37. free(temp);
38. **return** element;
39. }
41. **int** main() {
42. enqueue(10);
43. enqueue(20);
44. enqueue(30);
45. printf("%d ", dequeue());
46. printf("%d ", dequeue());
47. printf("%d ", dequeue());
48. printf("%d ", dequeue());
49. **return** 0;
50. }

The output of the code will be:

**Output:**

10 20 30 Queue is empty-1

**Explanation:**

1. First, we enqueue three elements 10, 20 and 30 into the queue.
2. Then, we dequeue and print the front element of the queue, which is 10.
3. Next, we dequeue and print the front element of the queue again, which is 20.
4. Then, we dequeue and print the front element of the queue again, which is 30.
5. Finally, we try to dequeue from the empty queue, which prints the message "Queue is empty" and returns -1.

Advantages:

* Queues are particularly useful for implementing algorithms that require elements to be processed in a precise sequence, such as breadth-first search and task scheduling.
* Because queue operations have an O(1) time complexity, they can be executed fast even on enormous queues.
* Queues are particularly flexible since they may be simply implemented using an array or a linked list.

Disadvantages:

* A queue, unlike a stack, cannot be constructed with a single pointer, making queue implementation slightly more involved.
* If the queue is constructed as an array, it might soon fill up if too many elements are added, resulting in performance concerns or possibly a crash.
* When utilising a linked list to implement the queue, the memory overhead of each node can be significant, especially for small elements.
* Queues are a fundamental data structure used in computer science to manage and store data. They are used in various applications, from operating systems to networking systems, and are widely regarded as essential tools for software development. However, like any tool, there are some drawbacks and limitations of queues. In this article, we will explore the disadvantages of queues.
* [One of the limitations of queues is that they do not allow for random access to elements 1](https://www.prepbytes.com/blog/queues/what-are-the-drawbacks-of-simple-queues-in-data-structures/). [Another limitation is that queues have a limited capacity, and once that capacity is reached, new elements cannot be added until some elements are removed from the queue 1](https://www.prepbytes.com/blog/queues/what-are-the-drawbacks-of-simple-queues-in-data-structures/)[2](https://www.geeksforgeeks.org/applications-advantages-and-disadvantages-of-queue/). [Searching an element in a queue takes O(N) time](https://www.prepbytes.com/blog/queues/what-are-the-drawbacks-of-simple-queues-in-data-structures/) [2](https://www.geeksforgeeks.org/applications-advantages-and-disadvantages-of-queue/).
* In summary, simple queues have some limitations that should be considered when designing software systems. However, they remain an important tool for managing and storing data in many applications.
* [1](https://www.prepbytes.com/blog/queues/what-are-the-drawbacks-of-simple-queues-in-data-structures/): [Prepbytes](https://www.prepbytes.com/blog/queues/what-are-the-drawbacks-of-simple-queues-in-data-structures/)[2](https://www.geeksforgeeks.org/applications-advantages-and-disadvantages-of-queue/): [GeeksforGeeks](https://www.geeksforgeeks.org/applications-advantages-and-disadvantages-of-queue/)

### Disadvantages of Queues

Here are some disadvantages of queues that should be considered:

1. **Limited random access:** One of the limitations of queues is that they do not allow for random access to elements. This means that you cannot access or modify elements in the middle of the queue without first removing all the elements in front of it. This limitation can be problematic in certain scenarios where random access is needed.
2. **Limited capacity:** Queues have a limited capacity, and once that capacity is reached, new elements cannot be added until existing elements are removed. This limitation can lead to a situation where the queue is full, but new elements need to be added, resulting in overflow and data loss.
3. **Not suitable for search operations:** One of the drawbacks of Queues is that they are not suitable for search operations since they follow the FIFO (First In First Out) principle. If you need to search for a specific element in the queue, you will have to dequeue each element until you find the desired element.
4. **Overhead memory:** Queues require a significant amount of overhead memory to store the pointers that connect the elements. This overhead memory can be wasteful if you need to store a large number of small elements.
5. **Synchronization issues:** In a multithreaded environment, queues can have synchronization issues. If multiple threads try to access and modify the same queue simultaneously, it can result in race conditions and other synchronization problems.

# An Introduction to Data Structures

1. Since the invention of computers, people have been using the term "**Data**" to refer to Computer Information, either transmitted or stored. However, there is data that exists in order types as well. Data can be numbers or texts written on a piece of paper, in the form of bits and bytes stored inside the memory of electronic devices, or facts stored within a person's mind. As the world started modernizing, this data became a significant aspect of everyone's day-to-day life, and various implementations allowed them to store it differently.
2. **Data** is a collection of facts and figures or a set of values or values of a specific format that refers to a single set of item values. The data items are then classified into sub-items, which is the group of items that are not known as the simple primary form of the item.
3. Let us consider an example where an employee name can be broken down into three sub-items: First, Middle, and Last. However, an ID assigned to an employee will generally be considered a single item.

## **What is Data Structure?**

**Data Structure** is a branch of Computer Science. The study of data structure allows us to understand the organization of data and the management of the data flow in order to increase the efficiency of any process or program. Data Structure is a particular way of storing and organizing data in the memory of the computer so that these data can easily be retrieved and efficiently utilized in the future when required. The data can be managed in various ways, like the logical or mathematical model for a specific organization of data is known as a data structure.

The scope of a particular data model depends on two factors:

1. First, it must be loaded enough into the structure to reflect the definite correlation of the data with a real-world object.
2. Second, the formation should be so straightforward that one can adapt to process the data efficiently whenever necessary.

Some examples of Data Structures are Arrays, Linked Lists, Stack, Queue, Trees, etc. Data Structures are widely used in almost every aspect of Computer Science, i.e., Compiler Design, Operating Systems, Graphics, Artificial Intelligence, and many more.

Data Structures are the main part of many Computer Science Algorithms as they allow the programmers to manage the data in an effective way. It plays a crucial role in improving the performance of a program or software, as the main objective of the software is to store and retrieve the user's data as fast as possible.

## **Basic Terminologies related to Data Structures**

Data Structures are the building blocks of any software or program. Selecting the suitable data structure for a program is an extremely challenging task for a programme

Why should we learn Data Structures?

1. Data Structures and Algorithms are two of the key aspects of Computer Science.
2. Data Structures allow us to organize and store data, whereas Algorithms allow us to process that data meaningfully.
3. Learning Data Structures and Algorithms will help us become better Programmers.
4. We will be able to write code that is more effective and reliable.
5. We will also be able to solve problems more quickly and efficiently.

Understanding the Objectives of Data Structures

Data Structures satisfy two complementary objectives:

1. **Correctness:** Data Structures are designed to operate correctly for all kinds of inputs based on the domain of interest. In order words, correctness forms the primary objective of Data Structure, which always depends upon the problems that the Data Structure is meant to solve.
2. **Efficiency:** Data Structures also requires to be efficient. It should process the data quickly without utilizing many computer resources like memory space. In a real-time state, the efficiency of a data structure is a key factor in determining the success and failure of the process.

## **Understanding some Key Features of Data Structures**

Some of the Significant Features of Data Structures are:

1. **Robustness:** Generally, all computer programmers aim to produce software that yields correct output for every possible input, along with efficient execution on all hardware platforms. This type of robust software must manage both valid and invalid inputs.
2. **Adaptability:** Building software applications like Web Browsers, Word Processors, and Internet Search Engine include huge software systems that require correct and efficient working or execution for many years. Moreover, software evolves due to emerging technologies or ever-changing market conditions.
3. **Reusability:** The features like Reusability and Adaptability go hand in hand. It is known that the programmer needs many resources to build any software, making it a costly enterprise. However, if the software is developed in a reusable and adaptable way, then it can be applied in most future applications. Thus, by executing quality data structures, it is possible to build reusable software, which appears to be cost-effective and timesaving.

## **Classification of Data Structures**

A Data Structure delivers a structured set of variables related to each other in various ways. It forms the basis of a programming tool that signifies the relationship between the data elements and allows programmers to process the data efficiently.

We can classify Data Structures into two categories:

1. Primitive Data Structure
2. Non-Primitive Data Structure

The following figure shows the different classifications of Data Structures.

Primitive Data Structures

1. **Primitive Data Structures** are the data structures consisting of the numbers and the characters that come **in-built** into programs.
2. These data structures can be manipulated or operated directly by machine-level instructions.
3. Basic data types like **Integer, Float, Character**, and **Boolean** come under the Primitive Data Structures.
4. These data types are also called **Simple data types**, as they contain characters that can't be divided further

Non-Primitive Data Structures

1. **Non-Primitive Data Structures** are those data structures derived from Primitive Data Structures.
2. These data structures can't be manipulated or operated directly by machine-level instructions.
3. The focus of these data structures is on forming a set of data elements that is either **homogeneous** (same data type) or **heterogeneous** (different data types).
4. Based on the structure and arrangement of data, we can divide these data structures into two sub-categories -
   1. Linear Data Structures
   2. Non-Linear Data Structures

Linear Data Structures

A data structure that preserves a linear connection among its data elements is known as a Linear Data Structure. The arrangement of the data is done linearly, where each element consists of the successors and predecessors except the first and the last data element. However, it is not necessarily true in the case of memory, as the arrangement may not be sequential.

Based on memory allocation, the Linear Data Structures are further classified into two types:

1. **Static Data Structures:** The data structures having a fixed size are known as Static Data Structures. The memory for these data structures is allocated at the compiler time, and their size cannot be changed by the user after being compiled; however, the data stored in them can be altered.  
   The **Array** is the best example of the Static Data Structure as they have a fixed size, and its data can be modified later.
2. **Dynamic Data Structures:** The data structures having a dynamic size are known as Dynamic Data Structures. The memory of these data structures is allocated at the run time, and their size varies during the run time of the code. Moreover, the user can change the size as well as the data elements stored in these data structures at the run time of the code.  
   **Linked Lists, Stacks**, and **Queues** are common examples of dynamic data structures

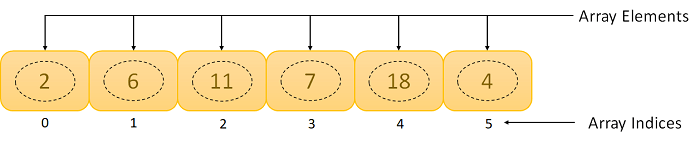
Types of Linear Data Structures

The following is the list of Linear Data Structures that we generally use:

**1. Arrays**

An **Array** is a data structure used to collect multiple data elements of the same data type into one variable. Instead of storing multiple values of the same data types in separate variable names, we could store all of them together into one variable. This statement doesn't imply that we will have to unite all the values of the same data type in any program into one array of that data type. But there will often be times when some specific variables of the same data types are all related to one another in a way appropriate for an array.

An Array is a list of elements where each element has a unique place in the list. The data elements of the array share the same variable name; however, each carries a different index number called a subscript. We can access any data element from the list with the help of its location in the list. Thus, the key feature of the arrays to understand is that the data is stored in contiguous memory locations, making it possible for the users to traverse through the data elements of the array using their respective indexes.



**Figure 3.** An Array

**Arrays can be classified into different types:**

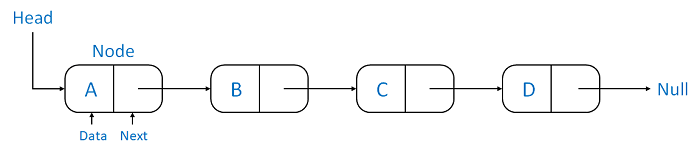
1. **One-Dimensional Array:** An Array with only one row of data elements is known as a One-Dimensional Array. It is stored in ascending storage location.
2. **Two-Dimensional Array:** An Array consisting of multiple rows and columns of data elements is called a Two-Dimensional Array. It is also known as a Matrix.
3. **Multidimensional Array:** We can define Multidimensional Array as an Array of Arrays. Multidimensional Arrays are not bounded to two indices or two dimensions as they can include as many indices are per the need.

**Some Applications of Array:**

1. We can store a list of data elements belonging to the same data type.
2. Array acts as an auxiliary storage for other data structures.
3. The array also helps store data elements of a binary tree of the fixed count.
4. Array also acts as a storage of matrices.

**2. Linked Lists**

A **Linked List** is another example of a linear data structure used to store a collection of data elements dynamically. Data elements in this data structure are represented by the Nodes, connected using links or pointers. Each node contains two fields, the information field consists of the actual data, and the pointer field consists of the address of the subsequent nodes in the list. The pointer of the last node of the linked list consists of a null pointer, as it points to nothing. Unlike the Arrays, the user can dynamically adjust the size of a Linked List as per the requirements.

 **Linked Lists can be classified into different types:**

1. **Singly Linked List:** A Singly Linked List is the most common type of Linked List. Each node has data and a pointer field containing an address to the next node.
2. **Doubly Linked List:** A Doubly Linked List consists of an information field and two pointer fields. The information field contains the data. The first pointer field contains an address of the previous node, whereas another pointer field contains a reference to the next node. Thus, we can go in both directions (backward as well as forward).
3. **Circular Linked List:** The Circular Linked List is similar to the Singly Linked List. The only key difference is that the last node contains the address of the first node, forming a circular loop in the Circular Linked List.

**Some Applications of Linked Lists:**

1. The Linked Lists help us implement stacks, queues, binary trees, and graphs of predefined size.
2. We can also implement Operating System's function for dynamic memory management.
3. Linked Lists also allow polynomial implementation for mathematical operations.
4. We can use Circular Linked List to implement Operating Systems or application functions that Round Robin execution of tasks.
5. Circular Linked List is also helpful in a Slide Show where a user requires to go back to the first slide after the last slide is presented.
6. Doubly Linked List is utilized to implement forward and backward buttons in a browser to move forward and backward in the opened pages of a website.

**3. Stacks**

A **Stack** is a Linear Data Structure that follows the **LIFO** (Last In, First Out) principle that allows operations like insertion and deletion from one end of the Stack, i.e., Top. Stacks can be implemented with the help of contiguous memory, an Array, and non-contiguous memory, a Linked List. Real-life examples of Stacks are piles of books, a deck of cards, piles of money, and many more.

**The primary operations in the Stack are as follows:**

1. **Push:** Operation to insert a new element in the Stack is termed as Push Operation.
2. **Pop:** Operation to remove or delete elements from the Stack is termed as Pop Operation.

**Some Applications of Stacks:**

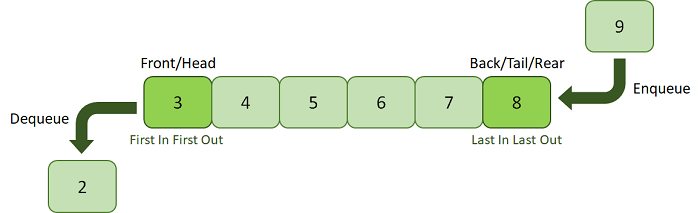
1. The Stack is used as a Temporary Storage Structure for recursive operations.
2. Stack is also utilized as Auxiliary Storage Structure for function calls, nested operations, and deferred/postponed functions.
3. We can manage function calls using Stacks.
4. Stacks are also utilized to evaluate the arithmetic expressions in different programming languages.
5. Stacks are also helpful in converting infix expressions to postfix expressions.
6. Stacks allow us to check the expression's syntax in the programming environment.
7. We can match parenthesis using Stacks.
8. Stacks can be used to reverse a String.
9. Stacks are helpful in solving problems based on backtracking.
10. We can use Stacks in depth-first search in graph and tree traversal.
11. Stacks are also used in Operating System functions.
12. Stacks are also used in UNDO and REDO functions in an edit.

**4. Queues**

A **Queue** is a linear data structure similar to a Stack with some limitations on the insertion and deletion of the elements. The insertion of an element in a Queue is done at one end, and the removal is done at another or opposite end. Thus, we can conclude that the Queue data structure follows FIFO (First In, First Out) principle to manipulate the data elements. Implementation of Queues can be done using Arrays, Linked Lists, or Stacks. Some real-life examples of Queues are a line at the ticket counter, an escalator, a car wash, and many more.

he following are the primary operations of the Queue:

1. **Enqueue:** The insertion or Addition of some data elements to the Queue is called Enqueue. The element insertion is always done with the help of the rear pointer.
2. **Dequeue:** Deleting or removing data elements from the Queue is termed Dequeue. The deletion of the element is always done with the help of the front pointer.



**Figure 8.** A Queue

**Some Applications of Queues:**

1. Queues are generally used in the breadth search operation in Graphs.
2. Queues are also used in Job Scheduler Operations of Operating Systems, like a keyboard buffer queue to store the keys pressed by users and a print buffer queue to store the documents printed by the printer.
3. Queues are responsible for CPU scheduling, Job scheduling, and Disk Scheduling.
4. Priority Queues are utilized in file-downloading operations in a browser.
5. Queues are also used to transfer data between peripheral devices and the CPU.
6. Queues are also responsible for handling interrupts generated by the User Applications for the CPU.

Non-Linear Data Structures

Non-Linear Data Structures are data structures where the data elements are not arranged in sequential order. Here, the insertion and removal of data are not feasible in a linear manner. There exists a hierarchical relationship between the individual data items.

Types of Non-Linear Data Structures

The following is the list of Non-Linear Data Structures that we generally use:

**1. Trees**

A Tree is a Non-Linear Data Structure and a hierarchy containing a collection of nodes such that each node of the tree stores a value and a list of references to other nodes (the "children").

The Tree data structure is a specialized method to arrange and collect data in the computer to be utilized more effectively. It contains a central node, structural nodes, and sub-nodes connected via edges. We can also say that the tree data structure consists of roots, branches, and leaves connected.

**Trees can be classified into different types:**

1. **Binary Tree:** A Tree data structure where each parent node can have at most two children is termed a Binary Tree.
2. **Binary Search Tree:** A Binary Search Tree is a Tree data structure where we can easily maintain a sorted list of numbers.
3. **AVL Tree:** An AVL Tree is a self-balancing Binary Search Tree where each node maintains extra information known as a Balance Factor whose value is either -1, 0, or +1.
4. **B-Tree:** A B-Tree is a special type of self-balancing Binary Search Tree where each node consists of multiple keys and can have more than two children.

**Some Applications of Trees:**

1. Trees implement hierarchical structures in computer systems like directories and file systems.
2. Trees are also used to implement the navigation structure of a website.
3. We can generate code like Huffman's code using Trees.
4. Trees are also helpful in decision-making in Gaming applications.
5. Trees are responsible for implementing priority queues for priority-based OS scheduling functions.
6. Trees are also responsible for parsing expressions and statements in the compilers of different programming languages.
7. We can use Trees to store data keys for indexing for Database Management System (DBMS).
8. Spanning Trees allows us to route decisions in Computer and Communications Networks.
9. Trees are also used in the path-finding algorithm implemented in Artificial Intelligence (AI), Robotics, and Video Games Applications.
10. **2. Graphs**
11. A Graph is another example of a Non-Linear Data Structure comprising a finite number of nodes or vertices and the edges connecting them. The Graphs are utilized to address problems of the real world in which it denotes the problem area as a network such as social networks, circuit networks, and telephone networks. For instance, the nodes or vertices of a Graph can represent a single user in a telephone network, while the edges represent the link between them via telephone.
12. The Graph data structure, G is considered a mathematical structure comprised of a set of vertices, V and a set of edges, E as shown below:
13. G = (V,E)

Basic Operations of Data Structures

In the following section, we will discuss the different types of operations that we can perform to manipulate data in every data structure:

1. **Traversal:** Traversing a data structure means accessing each data element exactly once so it can be administered. For example, traversing is required while printing the names of all the employees in a department.
2. **Search:** Search is another data structure operation which means to find the location of one or more data elements that meet certain constraints. Such a data element may or may not be present in the given set of data elements. For example, we can use the search operation to find the names of all the employees who have the experience of more than 5 years.
3. **Insertion:** Insertion means inserting or adding new data elements to the collection. For example, we can use the insertion operation to add the details of a new employee the company has recently hired.
4. **Deletion:** Deletion means to remove or delete a specific data element from the given list of data elements. For example, we can use the deleting operation to delete the name of an employee who has left the job.
5. **Sorting:** Sorting means to arrange the data elements in either Ascending or Descending order depending on the type of application. For example, we can use the sorting operation to arrange the names of employees in a department in alphabetical order or estimate the top three performers of the month by arranging the performance of the employees in descending order and extracting the details of the top three.
6. **Merge:** Merge means to combine data elements of two sorted lists in order to form a single list of sorted data elements.
7. **Create:** Create is an operation used to reserve memory for the data elements of the program. We can perform this operation using a declaration statement. The creation of data structure can take place either during the following:
   1. Compile-time
   2. Run-time  
      For example, the **malloc()** function is used in C Language to create data structure.
8. **Selection:** Selection means selecting a particular data from the available data. We can select any particular data by specifying conditions inside the loop.
9. **Update:** The Update operation allows us to update or modify the data in the data structure. We can also update any particular data by specifying some conditions inside the loop, like the Selection operation.
10. **Splitting:** The Splitting operation allows us to divide data into various subparts decreasing the overall process completion time.

Understanding the Abstract Data Type

As per the **National Institute of Standards and Technology (NIST)**, a data structure is an arrangement of information, generally in the memory, for better algorithm efficiency. Data Structures include linked lists, stacks, queues, trees, and dictionaries. They could also be a theoretical entity, like the name and address of a person.

From the definition mentioned above, we can conclude that the operations in data structure include:

1. A high level of abstractions like addition or deletion of an item from a list.
2. Searching and sorting an item in a list.
3. Accessing the highest priority item in a list.

Whenever the data structure does such operations, it is known as an **Abstract Data Type (ADT)**.

We can define it as a set of data elements along with the operations on the data. The term "abstract" refers to the fact that the data and the fundamental operations defined on it are being studied independently of their implementation. It includes what we can do with the data, not how we can do it.

An ADI implementation contains a storage structure in order to store the data elements and algorithms for fundamental operation. All the data structures, like an array, linked list, queue, stack, etc., are examples of ADT.

Array in Data Structure

In this article, we will discuss the array in data structure. Arrays are defined as the collection of similar types of data items stored at contiguous memory locations. It is one of the simplest data structures where each data element can be randomly accessed by using its index number.

In C programming, they are the derived data types that can store the primitive type of data such as int, char, double, float, etc. For example, if we want to store the marks of a student in 6 subjects, then we don't need to define a different variable for the marks in different subjects. Instead, we can define an array that can store the marks in each subject at the contiguous memory locations.

Properties of array

There are some of the properties of an array that are listed as follows -

* Each element in an array is of the same data type and carries the same size that is 4 bytes.
* Elements in the array are stored at contiguous memory locations from which the first element is stored at the smallest memory location.
* Elements of the array can be randomly accessed since we can calculate the address of each element of the array with the given base address and the size of the data element.

Representation of an array

We can represent an array in various ways in different programming languages. As an illustration, let's see the declaration of array in C language -

As per the above illustration, there are some of the following important points -

* Index starts with 0.
* The array's length is 10, which means we can store 10 elements.
* Each element in the array can be accessed via its index.

Why are arrays required?

Arrays are useful because -

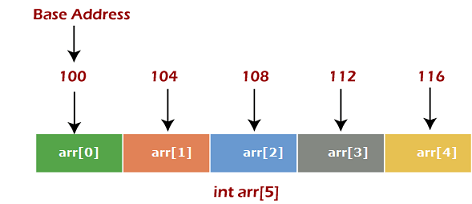
* Sorting and searching a value in an array is easier.
* Arrays are best to process multiple values quickly and easily.
* **Arrays are good for storing multiple values in a single variable -** In computer programming, most cases require storing a large number of data of a similar type. To store such an amount of data, we need to define a large number of variables. It would be very difficult to remember the names of all the variables while writing the programs. Instead of naming all the variables with a different name, it is better to define an array and store all the elements into it.

Memory allocation of an array

As stated above, all the data elements of an array are stored at contiguous locations in the main memory. The name of the array represents the base address or the address of the first element in the main memory. Each element of the array is represented by proper indexing.

We can define the indexing of an array in the below ways -

1. 0 (zero-based indexing): The first element of the array will be arr[0].
2. 1 (one-based indexing): The first element of the array will be arr[1].
3. n (n - based indexing): The first element of the array can reside at any random index number.



In the above image, we have shown the memory allocation of an array arr of size 5. The array follows a 0-based indexing approach. The base address of the array is 100 bytes. It is the address of arr[0]. Here, the size of the data type used is 4 bytes; therefore, each element will take 4 bytes in the memory.

How to access an element from the array?

We required the information given below to access any random element from the array -

* Base Address of the array.
* Size of an element in bytes.
* Type of indexing, array follows.

The formula to calculate the address to access an array element -

1. Byte address of element A[i]  = base address + size \* ( i - first index)

Here, size represents the memory taken by the primitive data types. As an instance, **int** takes 2 bytes, **float** takes 4 bytes of memory space in C programming.

We can understand it with the help of an example -

Suppose an array, A[-10 ..... +2 ] having Base address (BA) = 999 and size of an element = 2 bytes, find the location of A[-1].

L(A[-1]) = 999 + 2 x [(-1) - (-10)]

= 999 + 18

## = 1 Basic operations

Now, let's discuss the basic operations supported in the array -

* Traversal - This operation is used to print the elements of the array.
* Insertion - It is used to add an element at a particular index.
* Deletion - It is used to delete an element from a particular index.
* Search - It is used to search an element using the given index or by the value.
* Update - It updates an element at a particular index.

Traversal operation

This operation is performed to traverse through the array elements. It prints all array elements one after another. We can understand it with the below program -

1. #include <stdio.h>
2. **void** main() {
3. **int** Arr[5] = {18, 30, 15, 70, 12};
4. **int** i;
5. printf("Elements of the array are:\n");
6. **for**(i = 0; i<5; i++) {
7. printf("Arr[%d] = %d,  ", i, Arr[i]);
8. }
9. }

**Output**

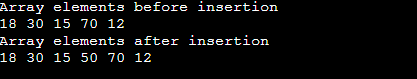
Array in DS

Insertion operation

This operation is performed to insert one or more elements into the array. As per the requirements, an element can be added at the beginning, end, or at any index of the array. Now, let's see the implementation of inserting an element into the array.

1. #include <stdio.h>
2. **int** main()
3. {
4. **int** arr[20] = { 18, 30, 15, 70, 12 };
5. **int** i, x, pos, n = 5;
6. printf("Array elements before insertion\n");
7. **for** (i = 0; i < n; i++)
8. printf("%d ", arr[i]);
9. printf("\n");
11. x = 50; // element to be inserted
12. pos = 4;
13. n++;
15. **for** (i = n-1; i >= pos; i--)
16. arr[i] = arr[i - 1];
17. arr[pos - 1] = x;
18. printf("Array elements after insertion\n");
19. **for** (i = 0; i < n; i++)
20. printf("%d ", arr[i]);
21. printf("\n");
22. **return** 0;
23. }

**Output**

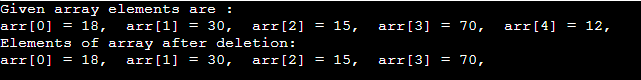


Deletion operation

As the name implies, this operation removes an element from the array and then reorganizes all of the array elements.

1. #include <stdio.h>
3. **void** main() {
4. **int** arr[] = {18, 30, 15, 70, 12};
5. **int** k = 30, n = 5;
6. **int** i, j;
8. printf("Given array elements are :\n");
10. **for**(i = 0; i<n; i++) {
11. printf("arr[%d] = %d,  ", i, arr[i]);
12. }
14. j = k;
16. **while**( j < n) {
17. arr[j-1] = arr[j];
18. j = j + 1;
19. }
21. n = n -1;
23. printf("\nElements of array after deletion:\n");
25. **for**(i = 0; i<n; i++) {
26. printf("arr[%d] = %d,  ", i, arr[i]);
27. }
28. }

**Output**



Search operation

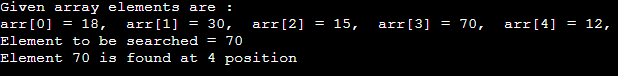
This operation is performed to search an element in the array based on the value or index.

### **Search operation**

This operation is performed to search an element in the array based on the value or index.

1. #include <stdio.h>
3. **void** main() {
4. **int** arr[5] = {18, 30, 15, 70, 12};
5. **int** item = 70, i, j=0 ;
7. printf("Given array elements are :\n");
9. **for**(i = 0; i<5; i++) {
10. printf("arr[%d] = %d,  ", i, arr[i]);
11. }
12. printf("\nElement to be searched = %d", item);
13. **while**( j < 5){
14. **if**( arr[j] == item ) {
15. **break**;
16. }
18. j = j + 1;
19. }
21. printf("\nElement %d is found at %d position", item, j+1);
22. }

**Output**

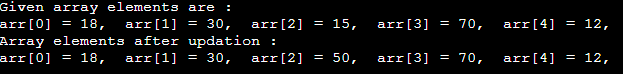


### **Update operation**

This operation is performed to update an existing array element located at the given index.

1. #include <stdio.h>
3. **void** main() {
4. **int** arr[5] = {18, 30, 15, 70, 12};
5. **int** item = 50, i, pos = 3;
7. printf("Given array elements are :\n");
9. **for**(i = 0; i<5; i++) {
10. printf("arr[%d] = %d,  ", i, arr[i]);
11. }
13. arr[pos-1] = item;
14. printf("\nArray elements after updation :\n");
16. **for**(i = 0; i<5; i++) {
17. printf("arr[%d] = %d,  ", i, arr[i]);
18. }
19. }

**Output**



## **Complexity of Array operations**

Time and space complexity of various array operations are described in the following table.

**Time Complexity**

|  |  |  |
| --- | --- | --- |
| **Operation** | **Average Case** | **Worst Case** |
| Access | O(1) | O(1) |
| Search | O(n) | O(n) |
| Insertion | O(n) | O(n) |
| Deletion | O(n) | O(n) |

**Space Complexity**

In array, space complexity for worst case is **O(n)**.

## **Advantages of Array**

* Array provides the single name for the group of variables of the same type. Therefore, it is easy to remember the name of all the elements of an array.
* Traversing an array is a very simple process; we just need to increment the base address of the array in order to visit each element one by one.
* Any element in the array can be directly accessed by using the index.

## **Disadvantages of Array**

* Array is homogenous. It means that the elements with similar data type can be stored in it.
* In array, there is static memory allocation that is size of an array cannot be altered.
* There will be wastage of memory if we store less number of elements than the declared size.

# Linked List

* Linked List can be defined as collection of objects called **nodes** that are randomly stored in the memory.
* A node contains two fields i.e. data stored at that particular address and the pointer which contains the address of the next node in the memory.
* The last node of the list contains pointer to the null.

DS Linked List

## **Uses of Linked List**

* The list is not required to be contiguously present in the memory. The node can reside any where in the memory and linked together to make a list. This achieves optimized utilization of space.
* list size is limited to the memory size and doesn't need to be declared in advance.
* Empty node can not be present in the linked list.
* We can store values of primitive types or objects in the singly linked list.

Why use linked list over array?

Till now, we were using array data structure to organize the group of elements that are to be stored individually in the memory. However, Array has several advantages and disadvantages which must be known in order to decide the data structure which will be used throughout the program.

Array contains following limitations:

1. The size of array must be known in advance before using it in the program.
2. Increasing size of the array is a time taking process. It is almost impossible to expand the size of the array at run time.
3. All the elements in the array need to be contiguously stored in the memory. Inserting any element in the array needs shifting of all its predecessors.

Linked list is the data structure which can overcome all the limitations of an array. Using linked list is useful because,

1. It allocates the memory dynamically. All the nodes of linked list are non-contiguously stored in the memory and linked together with the help of pointers.
2. Sizing is no longer a problem since we do not need to define its size at the time of declaration. List grows as per the program's demand and limited to the available memory space.

Singly linked list or One way chain

Singly linked list can be defined as the collection of ordered set of elements. The number of elements may vary according to need of the program. A node in the singly linked list consist of two parts: data part and link part. Data part of the node stores actual information that is to be represented by the node while the link part of the node stores the address of its immediate successor.

One way chain or singly linked list can be traversed only in one direction. In other words, we can say that each node contains only next pointer, therefore we can not traverse the list in the reverse direction.

Consider an example where the marks obtained by the student in three subjects are stored in a linked list as shown in the figure.

DS Singly Linked List

In the above figure, the arrow represents the links. The data part of every node contains the marks obtained by the student in the different subject. The last node in the list is identified by the null pointer which is present in the address part of the last node. We can have as many elements we require, in the data part of the list.

Operations on Singly Linked List

There are various operations which can be performed on singly linked list. A list of all such operations is given below.

Node Creation

1. struct node
2. {
3. **int** data;
4. struct node \*next;
5. };
6. struct node \*head, \*ptr;
7. ptr = (struct node \*)malloc(sizeof(struct node \*));

Insertion

The insertion into a singly linked list can be performed at different positions. Based on the position of the new node being inserted, the insertion is categorized into the following categories.

|  |  |  |
| --- | --- | --- |
| **SN** | **Operation** | **Description** |
| 1 | [Insertion at beginning](https://www.javatpoint.com/insertion-in-singly-linked-list-at-beginning) | It involves inserting any element at the front of the list. We just need to a few link adjustments to make the new node as the head of the list. |
| 2 | [Insertion at end of the list](https://www.javatpoint.com/insertion-in-singly-linked-list-at-end) | It involves insertion at the last of the linked list. The new node can be inserted as the only node in the list or it can be inserted as the last one. Different logics are implemented in each scenario. |
| 3 | [Insertion after specified node](https://www.javatpoint.com/insertion-in-singly-linked-list-after-specified-node) | It involves insertion after the specified node of the linked list. We need to skip the desired number of nodes in order to reach the node after which the new node will be inserted. . |

Deletion and Traversing

The Deletion of a node from a singly linked list can be performed at different positions. Based on the position of the node being deleted, the operation is categorized into the following categories.

|  |  |  |
| --- | --- | --- |
| **SN** | **Operation** | **Description** |
| 1 | [Deletion at beginning](https://www.javatpoint.com/deletion-in-singly-linked-list-at-beginning) | It involves deletion of a node from the beginning of the list. This is the simplest operation among all. It just need a few adjustments in the node pointers. |
| 2 | [Deletion at the end of the list](https://www.javatpoint.com/deletion-in-singly-linked-list-at-end) | It involves deleting the last node of the list. The list can either be empty or full. Different logic is implemented for the different scenarios. |
| 3 | [Deletion after specified node](https://www.javatpoint.com/deletion-in-singly-linked-list-after-specified-node) | It involves deleting the node after the specified node in the list. we need to skip the desired number of nodes to reach the node after which the node will be deleted. This requires traversing through the list. |
| 4 | [Traversing](https://www.javatpoint.com/traversing-in-singly-linked-list) | In traversing, we simply visit each node of the list at least once in order to perform some specific operation on it, for example, printing data part of each node present in the list. |
| 5 | [Searching](https://www.javatpoint.com/searching-in-singly-linked-list) | In searching, we match each element of the list with the given element. If the element is found on any of the location then location of that element is returned otherwise null is returned. . |

Linked List in C: Menu Driven Program

1. #include<stdio.h>
2. #include<stdlib.h>
3. struct node
4. {
5. **int** data;
6. struct node \*next;
7. };
8. struct node \*head;
10. **void** beginsert ();
11. **void** lastinsert ();
12. **void** randominsert();
13. **void** begin\_delete();
14. **void** last\_delete();
15. **void** random\_delete();
16. **void** display();
17. **void** search();
18. **void** main ()
19. {
20. **int** choice =0;
21. **while**(choice != 9)
22. {
23. printf("\n\n\*\*\*\*\*\*\*\*\*Main Menu\*\*\*\*\*\*\*\*\*\n");
24. printf("\nChoose one option from the following list ...\n");
25. printf("\n===============================================\n");
26. printf("\n1.Insert in begining\n2.Insert at last\n3.Insert at any random location\n4.Delete from Beginning\n
27. 5.Delete from last\n6.Delete node after specified location\n7.Search **for** an element\n8.Show\n9.Exit\n");
28. printf("\nEnter your choice?\n");
29. scanf("\n%d",&choice);
30. **switch**(choice)
31. {
32. **case** 1:
33. beginsert();
34. **break**;
35. **case** 2:
36. lastinsert();
37. **break**;
38. **case** 3:
39. randominsert();
40. **break**;
41. **case** 4:
42. begin\_delete();
43. **break**;
44. **case** 5:
45. last\_delete();
46. **break**;
47. **case** 6:
48. random\_delete();
49. **break**;
50. **case** 7:
51. search();
52. **break**;
53. **case** 8:
54. display();
55. **break**;
56. **case** 9:
57. exit(0);
58. **break**;
59. **default**:
60. printf("Please enter valid choice..");
61. }
62. }
63. }

# Types of Linked List

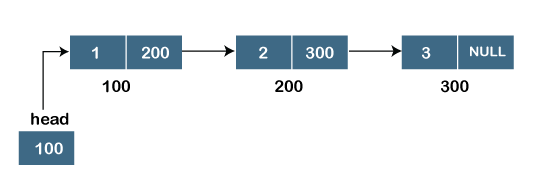
1. Before knowing about the types of a linked list, we should know what is **linked list**. So, to know about the linked list, click on the link given below:

## **Types of Linked list**

1. **The following are the types of linked list:**

### **Singly Linked list**

1. It is the commonly used linked list in programs. If we are talking about the linked list, it means it is a singly linked list. The singly linked list is a data structure that contains two parts, i.e., one is the data part, and the other one is the address part, which contains the address of the next or the successor node. The address part in a node is also known as a **pointer**.
2. Suppose we have three nodes, and the addresses of these three nodes are 100, 200 and 300 respectively. The representation of three nodes as a linked list is shown in the below figure:
3. We can observe in the above figure that there are three different nodes having address 100, 200 and 300 respectively. The first node contains the address of the next node, i.e., 200, the second node contains the address of the last node, i.e., 300, and the third node contains the NULL value in its address part as it does not point to any node. The pointer that holds the address of the initial node is known as a **head pointer**.
4. The linked list, which is shown in the above diagram, is known as a singly linked list as it contains only a single link. In this list, only forward traversal is possible; we cannot traverse in the backward direction as it has only one link in the list.

 **Representation of the node in a singly linked list**

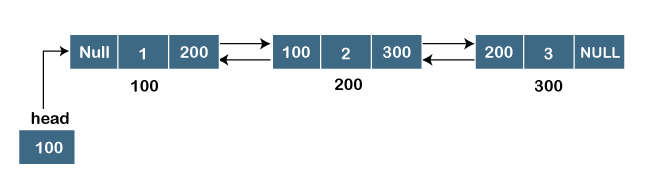
1. struct node
2. {
3. **int** data;
4. struct node \*next;
5. }

In the above representation, we have defined a user-defined structure named a **node** containing two members, the first one is data of integer type, and the other one is the pointer (next) of the node type.

Doubly linked list

As the name suggests, the doubly linked list contains two pointers. We can define the doubly linked list as a linear data structure with three parts: the data part and the other two address part. In other words, a doubly linked list is a list that has three parts in a single node, includes one data part, a pointer to its previous node, and a pointer to the next node.

Suppose we have three nodes, and the address of these nodes are 100, 200 and 300, respectively. The representation of these nodes in a doubly-linked list is shown below:



As we can observe in the above figure, the node in a doubly-linked list has two address parts; one part stores the ***address of the next*** while the other part of the node stores the ***previous node's address***. The initial node in the doubly linked list has the **NULL** value in the address part, which provides the address of the previous node.

**Representation of the node in a doubly linked list**

1. struct node
2. {
3. **int** data;
4. struct node \*next;
5. struct node \*prev;
6. }

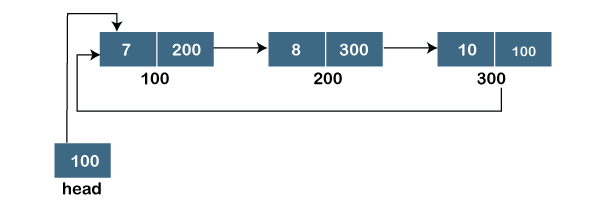
In the above representation, we have defined a user-defined structure named ***a node*** with three members, one is **data** of integer type, and the other two are the pointers, i.e., **next and prev** of the node type. The **next pointer** variable holds the address of the next node, and the **prev pointer** holds the address of the previous node. The type of both the pointers, i.e., **next and prev** is **struct node** as both the pointers are storing the address of the node of the ***struct node*** type.

Circular linked list

A circular linked list is a variation of a singly linked list. The only difference between the ***singly linked list*** and a ***circular linked*** list is that the last node does not point to any node in a singly linked list, so its link part contains a NULL value. On the other hand, the circular linked list is a list in which the last node connects to the first node, so the link part of the last node holds the first node's address. The circular linked list has no starting and ending node. We can traverse in any direction, i.e., either backward or forward. The diagrammatic representation of the circular linked list is shown below:

1. struct node
2. {
3. **int** data;
4. struct node \*next;
5. }

A circular linked list is a sequence of elements in which each node has a link to the next node, and the last node is having a link to the first node. The representation of the circular linked list will be similar to the singly linked list, as shown below:

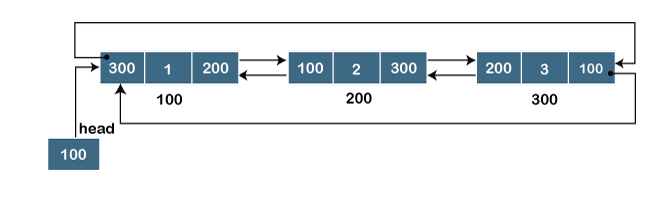


To know more about the circular linked list, click on the link given below:

<https://www.javatpoint.com/circular-singly-linked-list>

Doubly Circular linked list

The doubly circular linked list has the features of both the ***circular linked list*** and ***doubly linked list***.



The above figure shows the representation of the doubly circular linked list in which the last node is attached to the first node and thus creates a circle. It is a doubly linked list also because each node holds the address of the previous node also. The main difference between the doubly linked list and doubly circular linked list is that the doubly circular linked list does not contain the NULL value in the previous field of the node. As the doubly circular linked contains three parts, i.e., two address parts and one data part so its representation is similar to the doubly linked list.

1. struct node
2. {
3. **int** data;
4. struct node \*next;
5. struct node \*prev;
6. }

# Applications of Queue Data Structure

1. Imagine a line forming at the desk of a movie theatre. A new individual join at the tail end of the line as the person in line who is first receives the ticket and departs.

### **Queue Data Structure: What Is It?**

1. An ordered succession of elements is contained in a queue, which is a linear data structure. It is an abstract data type that resembles a stack partially.
2. We can conduct operations on a queue's ends, unlike stacks, though.
3. At one end of the queue, we add data, and at the other, we remove it.

## Some of the Application of Queue in Data Structure

Here are some of the common application of queue in data structure:

* Queues can be used to schedule jobs and ensure that they are executed in the correct order.
* Queues can be used to manage the order in which print jobs are sent to the printer.
* Queues are used to implement the breadth-first search algorithm.
* Queues can be used to manage incoming calls and ensure that they are handled in the correct order.
* Queues can be used to manage the order in which processes are executed on the CPU.
* Queues can be used for buffering data while it is being transferred between two systems. When data is received, it is added to the back of the queue, and when data is sent, it is removed from the front of the queue.

## Other Application of Queue in Data Structure

Some other application of queue in data structure:

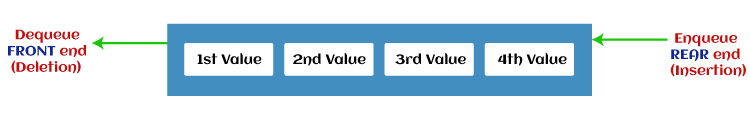
* Applied as buffers on playing music in the mp3 players or CD players.
* Applied on handling the interruption in the operating system.
* Applied to add a song at the end of the playlist.
* Applied as the waiting queue for using the single shared resources like CPU, Printers, or Disk.
* Applied to the chat application when someone sends messages to us and we don’t have an internet connection then the messages are stored in the server of the chat application

### **ypes of Queue**

There are four different types of queue that are listed as follows -

Simple Queue or Linear Queue

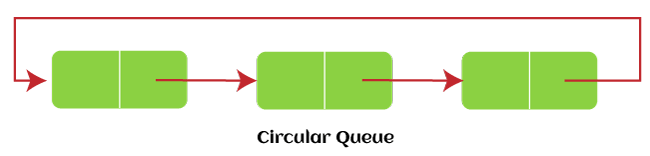
In Linear Queue, an insertion takes place from one end while the deletion occurs from another end. The end at which the insertion takes place is known as the rear end, and the end at which the deletion takes place is known as front end. It strictly follows the FIFO rule.



The major drawback of using a linear Queue is that insertion is done only from the rear end. If the first three elements are deleted from the Queue, we cannot insert more elements even though the space is available in a Linear Queue. In this case, the linear Queue shows the overflow condition as the rear is pointing to the last element of the Queue.

Circular Queue

In Circular Queue, all the nodes are represented as circular. It is similar to the linear Queue except that the last element of the queue is connected to the first element. It is also known as Ring Buffer, as all the ends are connected to another end. The representation of circular queue is shown in the below image -



The drawback that occurs in a linear queue is overcome by using the circular queue. If the empty space is available in a circular queue, the new element can be added in an empty space by simply incrementing the value of rear. The main advantage of using the circular queue is better memory utilization.

Priority Queue

It is a special type of queue in which the elements are arranged based on the priority. It is a special type of queue data structure in which every element has a priority associated with it. Suppose some elements occur with the same priority, they will be arranged according to the FIFO principle. The representation of priority queue is shown in the below image -

### Types of Queue Deque (or, Double Ended Queue)

In Deque or Double Ended Queue, insertion and deletion can be done from both ends of the queue either from the front or rear. It means that we can insert and delete elements from both front and rear ends of the queue. Deque can be used as a palindrome checker means that if we read the string from both ends, then the string would be the same.

Deque can be used both as stack and queue as it allows the insertion and deletion operations on both ends. Deque can be considered as stack because stack follows the LIFO (Last In First Out) principle in which insertion and deletion both can be performed only from one end. And in deque, it is possible to perform both insertion and deletion from one end, and Deque does not follow the FIFO principle.

The representation of the deque is shown in the below image -

# Types of Queue Hash Function in Data Structure

There are almost 150 Zettabytes of data getting generated each day, which is equivalent to 150 trillion Gigabytes of data. With such an enormous speed of data growth, there emerges a need to store this data in an effective and efficient manner. By an effective and efficient manner of storage, we mean a way that will provide us the flexibility of retrieving the data in a minimal amount of time, because the more time required for an operation will directly increase the cost associated with that particular operation. So, in order to reduce the cost of an operation and do that task in an efficient manner, we need to reduce the retrieval time of the data for that particular task. And the solution for reducing the retrieval time is the Hash function or hash table. The hash function is used to map or bind the data to a particular hash value and then that hash value will be used as an index or a key to store that value in the hash table. The main benefit of storing data in the hash tables is that the retrieval time of the data stored in the hash tables is of unit time. That means the data that is stored in the hash table can be read with time complexity of O(1). So, in this way, the hash tables play an important role in drastically reducing the time required for reading the data from the hash tables. And for the working of the hash tables, it requires a hash function. Now let us see what is a hash function and how it works.

Hash Function can be defined as an algorithm or a function that is used to map or convert data of bigger size or length to a fixed or small index or hash value. In other words, a hash function can be defined as an algorithm that will be used to convert the data of higher length or size to data that is within a fixed range or size.

The input parameter that is passed to a hash function is the input data that needs to map to some hash data. And the output or result provided by a hash function depicts the hash value or the hashes that are associated with that input parameter value. The hash functions are associated with the hash tables that will actually store the data in the memory and the hash function is used only to map those values to the hash tables. The hash value returned by the hash function for the data item passed as an input parameter is then further used as an index to map or store that input data into the hash table. Or, we can say that the hash value returned by the hash function for the data item passed as an input parameter is used as a key for storing that data which will help in the easy and efficient retrieval of the stored data.

For an ideal hash function to work, it should satisfy two basic properties or conditions so that it can deliver optimal results within a specified span of computation period. These two basic properties or conditions for an efficient hash function to store data in the hash table are:

* Firstly, the hash function should be very fast in calculating and delivering the results. The speed is one of the main and crucial parameters that will affect the overall efficiency of a hash function. The relation between the speed of computation of the results and efficiency of a hash function is inversely proportional, which means more time required for computation and delivering the results lesser the efficiency of the hash function and vice-versa. So, ideally, it is required that the time span that is required for the calculations and delivering the results of a hash function should be as least as possible in order to maintain the efficiency of the hash function.
* Once the speed of creating the result is achieved, the next step is to deliver the results correctly and accurately. So, the second most important parameter is the accuracy of the results generated by the hash function. So, it is required that the result generated by the hash function should be unique and accurate because the hash values that are generated by the hash function acts as keys while sorting the data in the hash tables. So, the uniqueness of the hash value generated ensures that no two data should be mapped to the same key or hash value. That is why the accuracy and uniqueness of the result generated affects the efficiency of the whole hash function which in return affects the efficiency of storing data in the hash tables.

# What is Performance Analysis of an algorithm?

If we want to go from city "A" to city "B", there can be many ways of doing this. We can go by flight, by bus, by train and also by bicycle. Depending on the availability and convenience, we choose the one which suits us. Similarly, in computer science, there are multiple algorithms to solve a problem. When we have more than one algorithm to solve a problem, we need to select the best one. Performance analysis helps us to select the best algorithm from multiple algorithms to solve a problem.  
When there are multiple alternative algorithms to solve a problem, we analyze them and pick the one which is best suitable for our requirements. The formal definition is as follows...

**Performance of an algorithm is a process of making evaluative judgement about algorithms.** That means when we have multiple algorithms to solve a problem, we need to select a suitable algorithm to solve that problem.  
We compare algorithms with each other which are solving the same problem, to select the best algorithm. To compare algorithms, we use a set of parameters or set of elements like memory required by that algorithm, the execution speed of that algorithm, easy to understand, easy to implement, etc., Generally, the performance of an algorithm depends on the following elements...

1. Whether that algorithm is providing the exact solution for the problem?
2. Whether it is easy to understand?
3. Whether it is easy to implement?
4. How much space (memory) it requires to solve the problem?
5. How much time it takes to solve the problem? Etc.,

When we want to analyse an algorithm, we consider only the space and time required by that particular algorithm and we ignore all the remaining elements.  
Based on this information, performance analysis of an algorithm can also be defined as follows...

**Performance analysis of an algorithm is the process of calculating space and time required by that algorithm.**

Performance analysis of an algorithm is performed by using the following measures...

1. Space required to complete the task of that algorithm (**Space Complexity**). It includes program space and data space
2. Time required to complete the task of that algorithm (**Time Complexity**)

# Substitution Method in Data Structure

Here we will see how to use substitution method to solve recurrence relations. We will take two examples to understand it in better way.

Suppose we are using the binary search technique. In this technique, we check whether the element is present at the end or not. If that is present at middle, then the algorithm terminates, otherwise we take either the left and right subarray from the actual array again and again. So in each step the size of the array decreases by n / 2. Suppose the binary search algorithm takes T(n) amount of time to execute. The base condition takes O(1) amount of time. So the recurrence equation will be like below −

T(n)={T(1)forn≤1\2T(n2)+cforn>1�(�)={�(1)����≤1\2�(�2)+�����>1

Solve − We will substitute the formula in each step to get the result −

T(n)=T(n2)+c�(�)=�(�2)+�

By substituting T(N/2) we can write,

T(n)=(T(n4)+c)+c�(�)=(�(�4)+�)+�

T(n)=T(n4)+2c�(�)=�(�4)+2�

T(n)=T(n8)+3c�(�)=�(�8)+3�

T(n)=T(n2k)+kc�(�)=�(�2�)+��

Now if(n2k)(�2�)reaches to 1, it indicates that 2k is n. So the value of k is log2𝑛

The complexity of T(n) = ϴ(log n)

Similarly, if we choose another example like merge sort, then in that case we divide the list into two parts. This division is taking place until the list size is only 1. After that we merge them in sorted order. The merging algorithm takes O(n) amount of time. So if the merge sort algorithm is taking T(n) amount of time, then by dividing it into two halves, and do the same task for each of them, they will take T(n/2) and so on. So the recurrence relation will be like below −

T(n)={T(1)forn=1\2T(n2)+cnforn>1�(�)={�(1)����=1\2�(�2)+������>1

Solve − We will substitute the formula in each step to get the result −

T(n)=2T(n2)+cn�(�)=2�(�2)+��

By substituting T(N/2) we can write,

T(n)=2(2T(n4)+cn2)+cn�(�)=2(2�(�4)+��2)+��

T(n)=4T(n4)+2cn�(�)=4�(�4)+2��

T(n)=8T(n8)+3cn�(�)=8�(�8)+3��

T(n)=2kT(n2k)+kcn�(�)=2��(�2�)+���

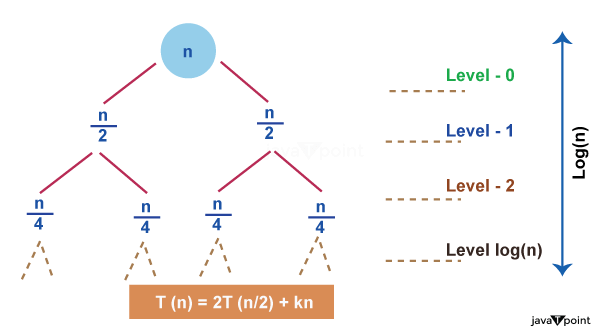
Now if(n2k)(�2�)reaches to 1, it indicates that 2k is n. So the value of k is log2𝑛. And T(n) will be −

𝑇(𝑛) = 𝑛𝑇(1) + 𝑐𝑛 log2𝑛

The complexity is θ(n log n)

### ############# **What Is Recursion Tree Method?**

* Recurrence relations like T(N) = T(N/2) + N or the two we covered earlier in the kinds of recursion section are solved using the recursion tree approach. These recurrence relations often use a divide and conquer strategy to address problems.
* It takes time to integrate the answers to the smaller sub problems that are created when a larger problem is broken down into smaller sub problems.
* The recurrence relation, for instance, is T(N) = 2 \* T(N/2) + O(N) for the Merge sort. The time needed to combine the answers to two sub problems with a combined size of T(N/2) is O(N), which is true at the implementation level as well.
* For instance, since the recurrence relation for binary search is T(N) = T(N/2) + 1, we know that each iteration of binary search results in a search space that is cut in half. Once the outcome is determined, we exit the function. The recurrence relation has +1 added because this is a constant time operation.
* The recurrence relation T(n) = 2T(n/2) + Kn is one to consider. Kn denotes the amount of time required to combine the answers to n/2-dimensional sub problems.
* Let's depict the recursion tree for the aforementioned recurrence relation.



We may draw a few conclusions from studying the recursion tree above, including

1. The magnitude of the problem at each level is all that matters for determining the value of a node. The issue size is n at level 0, n/2 at level 1, n/2 at level 2, and so on.

In general, we define the height of the tree as equal to log (n), where n is the size of the issue, and the height of this recursion tree is equal to the number of levels in the tree. This is true because, as we just established, the divide-and-conquer strategy is used by recurrence relations to solve problems, and getting from issue size n to problem size 1 simply requires taking log (n) steps.

* Consider the value of N = 16, for instance. If we are permitted to divide N by 2 at each step, how many steps are required to get N = 1? Considering that we are dividing by two at each step, the correct answer is 4, which is the value of log(16) base 2.

log(16) base 2

log(2^4) base 2

4 \* log(2) base 2, since log(a) base a = 1

so, 4 \* log(2) base 2 = 4

3. At each level, the second term in the recurrence is regarded as the root.

Although the word "tree" appears in the name of this strategy, you don't need to be an expert on trees to comprehend it.The **recursion tree method** is a technique used to solve recurrence relations that follow the divide and conquer approach. [It involves drawing a tree where each node represents the cost of a sub-problem and then summing up the costs at each level to find the asymptotic notation 1](https://www.scaler.com/topics/data-structures/recursion-tree-method/).

[Here are the steps to solve a recurrence relation using the recursion tree method 2](https://www.codesdope.com/course/algorithms-now-the-recursion/):

1. Convert the recurrence relation into a tree.
2. Calculate the cost of each level of the tree.
3. Sum up the costs of all levels to obtain an expression in terms of asymptotic notation.

The recursion tree method is useful for analyzing the time complexity of recursive algorithms. [By examining the structure of the tree, we can determine the number of recursive calls made and the work done at each level 3](https://www.javatpoint.com/daa-recursion-tree-method)[4](https://www.cs.cornell.edu/courses/cs3110/2012sp/lectures/lec20-master/lec20.html).

I hope this helps!

# Master Method

The Master Method is used for solving the following types of recurrence

T (n) = a TDAA Master Method+ f (n) with a≥1 and b≥1 be constant & f(n) be a function and DAA Master Methodcan be interpreted as

Let T (n) is defined on non-negative integers by the recurrence.

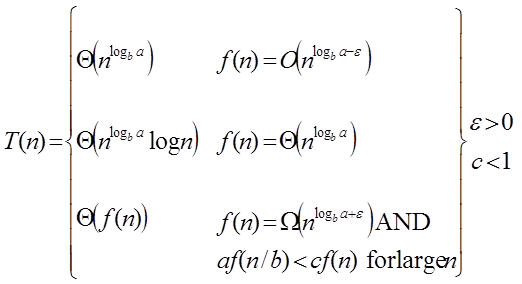
T (n) = a TDAA Master Method+ f (n)

In the function to the analysis of a recursive algorithm, the constants and function take on the following significance:

* n is the size of the problem.
* a is the number of subproblems in the recursion.
* n/b is the size of each subproblem. (Here it is assumed that all subproblems are essentially the same size.)
* f (n) is the sum of the work done outside the recursive calls, which includes the sum of dividing the problem and the sum of combining the solutions to the subproblems.
* It is not possible always bound the function according to the requirement, so we make three cases which will tell us what kind of bound we can apply on the function.

## **Master Theorem:**

It is possible to complete an asymptotic tight bound in these three cases:



**Case1:** If f (n) = DAA Master Method for some constant ε >0, then it follows that:

T (n) = Θ DAA Master Method

**Example:**

T (n) = 8 T DAA Master Method apply master theorem on it.

**Solution:**

Compare T (n) = 8 T DAA Master Method with

T (n) = a T DAA Master Method

a = 8, b=2, f (n) = 1000 n2, logba = log28 = 3

Put all the values in: f (n) = DAA Master Method

1000 n2 = O (n3-ε )

If we choose ε=1, we get: 1000 n2 = O (n3-1) = O (n2)

Since this equation holds, the first case of the master theorem applies to the given recurrence relation, thus resulting in the conclusion:

T (n) = Θ DAA Master Method

Therefore: T (n) = Θ (n3)

**Case 2:** If it is true, for some constant k ≥ 0 that:

F (n) = Θ DAA Master Method then it follows that: T (n) = Θ DAA Master Method