What is Data Structure?

The data structure name indicates itself that organizing the data in memory. There are many ways of organizing the data in the memory as we have already seen one of the data structures, i.e., array in C language. Array is a collection of memory elements in which data is stored sequentially, i.e., one after another. In other words, we can say that array stores the elements in a continuous manner. This organization of data is done with the help of an array of data structures. There are also other ways to organize the data in memory. Let's see the different types of data structures.

The data structure is not any programming language like C, C++, java, etc. It is a set of algorithms that we can use in any programming language to structure the data in the memory.

To structure the data in memory, 'n' number of algorithms were proposed, and all these algorithms are known as Abstract data types.

Types of Data Structures: There are two types of data structures:

1. Primitive data structure
2. Non-primitive data structure

Primitive Data structure: The primitive data structures are primitive data types. The int, char, float, double, and pointer are the primitive data structures that can hold a single value.

Non-Primitive Data structure: The non-primitive data structure is divided into two types:

1. Linear data structure
2. Non-linear data structure

Linear Data Structure: The arrangement of data in a sequential manner is known as a linear data structure. The data structures used for this purpose are Arrays, Linked list, Stacks, and Queues. In these data structures, one element is connected to only one another element in a linear form.

When one element is connected to the 'n' number of elements known as a non-linear data structure. The best example is trees and graphs. In this case, the elements are arranged in a random manner.

We will discuss the above data structures in brief in the coming topics. Now, we will see the common operations that we can perform on these data structures.

Data structures can also be classified as:

Static data structure: It is a type of data structure where the size is allocated at the compile time. Therefore, the maximum size is fixed.

Dynamic data structure: It is a type of data structure where the size is allocated at the run time. Therefore, the maximum size is flexible.

Major Operations:

The major or the common operations that can be performed on the data structures are:
Searching: We can search for any element in a data structure.
Sorting: We can sort the elements of a data structure either in an ascending or descending order.
Insertion: We can also insert the new element in a data structure.
Updation: We can also update the element, i.e., we can replace the element with another element.
Deletion: We can also perform the delete operation to remove the element from the data structure.
Advantages of Data structures: The following are the advantages of a data structure:

Efficiency: If the choice of a data structure for implementing a particular ADT is proper, it makes the program very efficient in terms of time and space.

Reusability: The data structure provides reusability means that multiple client programs can use the data structure.

Abstraction: The data structure specified by an ADT also provides the level of abstraction. The client cannot see the internal working of the data structure, so it does not have to worry about the implementation part. The client can only see the interface.

Tree traversal (Inorder, Preorder an Postorder)

In this article, we will discuss the tree traversal in the data structure. The term 'tree traversal' means traversing or visiting each node of a tree. There is a single way to traverse the linear data structure such as linked list, queue, and stack. Whereas, there are multiple ways to traverse a tree that are listed as follows -

1. Preorder traversal
2. Inorder traversal
3. Postorder traversal

So, in this article, we will discuss the above-listed techniques of traversing a tree. Now, let's start discussing the ways of tree traversal.
Preorder traversal : Algorithm

Until all nodes of the tree are not visited
Step 1 - Visit the root node
Step 2 - Traverse the left subtree recursively.
Step 3 - Traverse the right subtree recursively.

Postorder traversal: Algorithm

Until all nodes of the tree are not visited
Step 1 - Traverse the left subtree recursively.
Step 2 - Traverse the right subtree recursively.
Step 3 - Visit the root node.

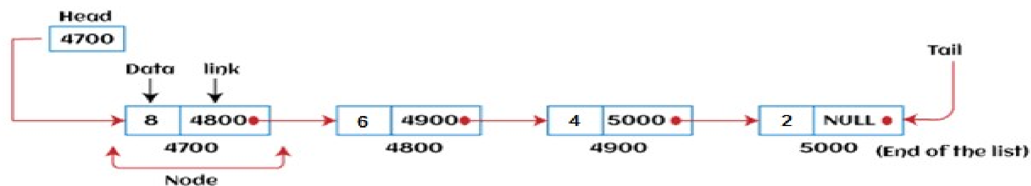
Inorder traversal: Algorithm
Until all nodes of the tree are not visited
Step 1 - Traverse the left subtree recursively.
Step 2 - Visit the root node.
Step 3 - Traverse the right subtree recursively.

Linked list:

Linked list is a linear data structure that includes a series of connected nodes. Linked list can be defined as the nodes that are randomly stored in the memory. A node in the linked list contains two parts, i.e., first is the data part and second is the address part. The last node of the list contains a pointer to the null. After array, linked list is the second most used data structure. In a linked list, every link contains a connection to another link.

Representation of a Linked list

Linked list can be represented as the connection of nodes in which each node points to the next node of the list. The representation of the linked list is shown below -



How to declare a linked list?
It is simple to declare an array, as it is of single type, while the declaration of linked list is a bit more typical than array. Linked list contains two parts, and both are of different types, i.e., one is the simple variable, while another is the pointer variable. We can declare the linked list by using the user-defined data type structure. The declaration of linked list is given as follows -

```
struct node {
        int data;
        struct node *next;
        }
```

In the above declaration, we have defined a structure named as node that contains two variables, one is data that is of integer type, and another one is next that is a pointer which contains the address of next node.

Types of Linked list

Linked list is classified into the following types -

Singly-linked list - Singly linked list can be defined as the collection of an ordered set of elements. A node in the singly linked list consists 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.

Doubly linked list - Doubly linked list is a complex type of linked list in which a node contains a pointer to the previous as well as the next node in the sequence. Therefore, in a doubly-linked list, a node consists of three parts: node data, pointer to the next node in sequence (next pointer), and pointer to the previous node (previous pointer).

Circular singly linked list - In a circular singly linked list, the last node of the list contains a pointer to the first node of the list. We can have circular singly linked list as well as circular doubly linked list.

Circular doubly linked list - Circular doubly linked list is a more complex type of data structure in which a node contains pointers to its previous node as well as the next node. Circular doubly linked list doesn't contain NULL in any of the nodes. The last node of the list contains the address of the first node of the list. The first node of the list also contains the address of the last node in its previous pointer.

B Tree

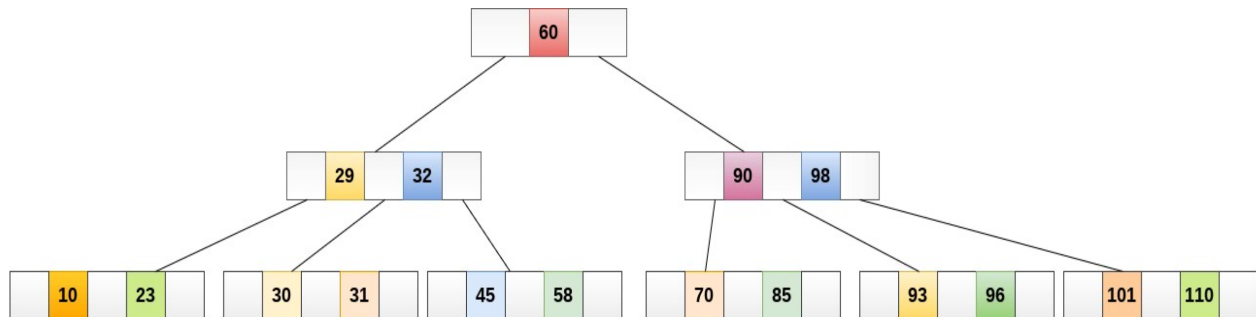
B Tree is a specialized m-way tree that can be widely used for disk access. A B-Tree of order m can have at most m-1 keys and m children. One of the main reason of using B tree is its capability to store large number of keys in a single node and large key values by keeping the height of the tree relatively small.

A B tree of order m contains all the properties of an M way tree. In addition, it contains the following properties.

1. Every node in a B-Tree contains at most m children.
2. Every node in a B-Tree except the root node and the leaf node contain at least m/2 children.
3. The root nodes must have at least 2 nodes.
4. All leaf nodes must be at the same level.

It is not necessary that, all the nodes contain the same number of children but, each node must have m/2 number of nodes.

A B tree of order 4 is shown in the following image.



While performing some operations on B Tree, any property of B Tree may violate such as number of minimum children a node can have. To maintain the properties of B Tree, the tree may split or join.

Operations

Searching : Searching in B Trees is similar to that in Binary search tree. For example, if we search for an item 49 in the following B Tree. The process will something like following :

1. Compare item 49 with root node 78. since 49 < 78 hence, move to its left sub-tree.
2. Since, 40<49<56, traverse right sub-tree of 40.
3. 49>45, move to right. Compare 49.
4. match found, return.

Searching in a B tree depends upon the height of the tree. The search algorithm takes O(log n) time to search any element in a B tree.

Inserting : Insertions are done at the leaf node level. The following algorithm needs to be followed in order to insert an item into B Tree.

1. Traverse the B Tree in order to find the appropriate leaf node at which the node can be inserted.
2. If the leaf node contain less than m-1 keys then insert the element in the increasing order.
3. Else, if the leaf node contains m-1 keys, then follow the following steps.
4. Insert the new element in the increasing order of elements.
5. Split the node into the two nodes at the median.
6. Push the median element upto its parent node.

If the parent node also contains m-1 number of keys, then split it too by following the same steps.

Deletion: Deletion is also performed at the leaf nodes. The node which is to be deleted can either be a leaf node or an internal node. Following algorithm needs to be followed in order to delete a node from a B tree.

1. Locate the leaf node.
2. If there are more than m/2 keys in the leaf node then delete the desired key from the node.
3. If the leaf node doesn't contain m/2 keys then complete the keys by taking the element from eight or left sibling.
4. If the left sibling contains more than m/2 elements then push its largest element up to its parent and move the intervening element down to the node where the key is deleted.
5. If the right sibling contains more than m/2 elements then push its smallest element up to the parent and move intervening element down to the node where the key is deleted.
6. If neither of the sibling contain more than m/2 elements then create a new leaf node by joining two leaf nodes and the intervening element of the parent node.
7. If parent is left with less than m/2 nodes then, apply the above process on the parent too.
8. If the the node which is to be deleted is an internal node, then replace the node with its in-order successor or predecessor. Since, successor or predecessor will always be on the leaf node hence, the process will be similar as the node is being deleted from the leaf node.

B+ Tree : B+ Tree is an extension of B Tree which allows efficient insertion, deletion and search operations.

In B Tree, Keys and records both can be stored in the internal as well as leaf nodes. Whereas, in B+ tree, records (data) can only be stored on the leaf nodes while internal nodes can only store the key values.

The leaf nodes of a B+ tree are linked together in the form of a singly linked lists to make the search queries more efficient.

Insertion in B+ Tree

Step 1: Insert the new node as a leaf node

Step 2: If the leaf doesn't have required space, split the node and copy the middle node to the next index node.

Step 3: If the index node doesn't have required space, split the node and copy the middle element to the next index page.

Deletion in B+ Tree

Step 1: Delete the key and data from the leaves.

Step 2: if the leaf node contains less than minimum number of elements, merge down the node with its sibling and delete the key in between them.

Step 3: if the index node contains less than minimum number of elements, merge the node with the sibling and move down the key in between them.

Convert Infix to Postfix notation

Before understanding the conversion from infix to postfix notation, we should know about the infix and postfix notations separately.

Infix expression: The expression of the form “a operator b” (a + b) i.e., when an operator is in-between every pair of operands.
Postfix expression: The expression of the form “a b operator” (ab+) i.e., When every pair of operands is followed by an operator.

Examples:

Input: A + B * C + D
Output: ABC*+D+

Input: ((A + B) – C * (D / E)) + F
Output: AB+CDE/*-F+

Illustration: Follow the below illustration for a better understanding

Consider the infix expression exp = “a+b*c+d”
and the infix expression is scanned using the iterator i, which is initialized as i = 0.

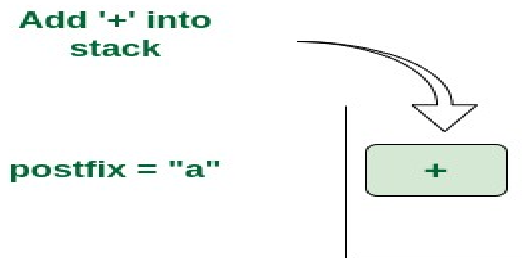
1st Step: Here i = 0 and exp[i] = ‘a’ i.e., an operand. So add this in the postfix expression. Therefore, postfix = “a”.

postfix = "a"

'a' is an operand. Add it in postfix expression

Add ‘a’ in the postfix

2nd Step: Here i = 1 and exp[i] = ‘+’ i.e., an operator. Push this into the stack. postfix = “a” and stack = {+}.

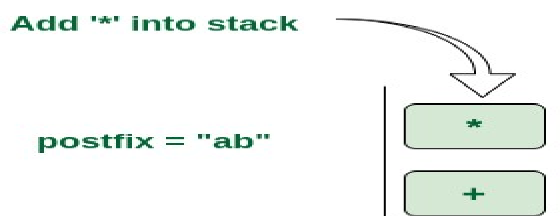


Push ‘+’ in the stack

3rd Step: Now i = 2 and exp[i] = ‘b’ i.e., an operand. So add this in the postfix expression. postfix = “ab” and stack = {+}.

Add ‘b’ in the postfix

4th Step: Now i = 3 and exp[i] = ‘*’ i.e., an operator. Push this into the stack. postfix = “ab” and stack = {+, *}.

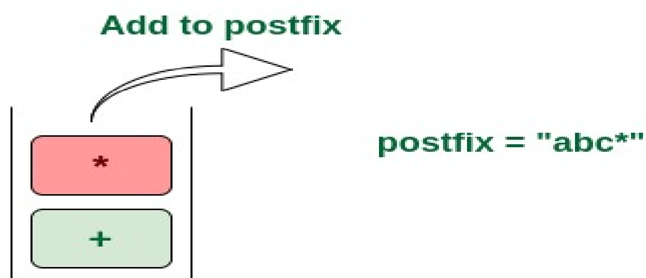


Push ‘*’ in the stack

5th Step: Now i = 4 and exp[i] = ‘c’ i.e., an operand. Add this in the postfix expression. postfix = “abc” and stack = {+, *}.

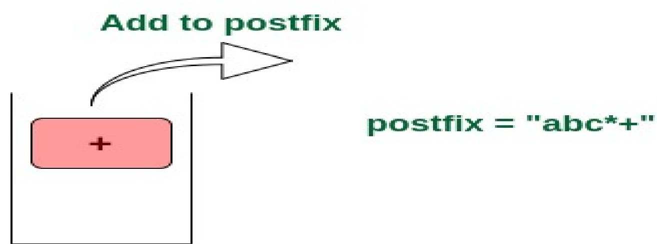
Add ‘c’ in the postfix

6th Step: Now i = 5 and exp[i] = ‘+’ i.e., an operator. The topmost element of the stack has higher precedence. So pop until the stack becomes empty or the top element has less precedence. ‘*’ is popped and added in postfix. So postfix = “abc*” and stack = {+}.



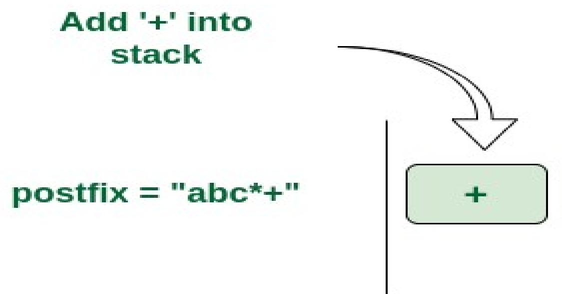
Pop ‘*’ and add in postfix

Now top element is ‘+‘ that also doesn’t have less precedence. Pop it. postfix = “abc*+”.



Pop ‘+’ and add it in postfix

Now stack is empty. So push ‘+’ in the stack. stack = {+}.

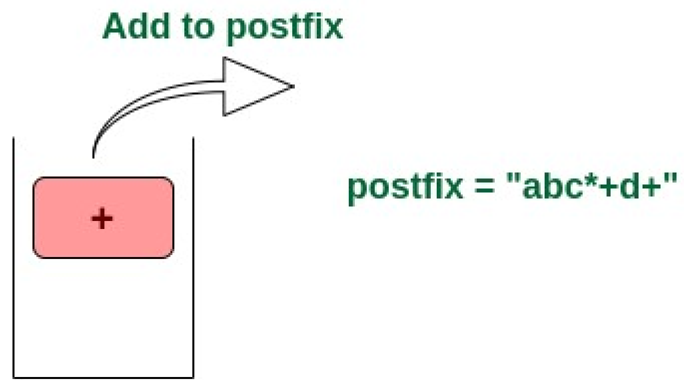


Push ‘+’ in the stack

7th Step: Now i = 6 and exp[i] = ‘d’ i.e., an operand. Add this in the postfix expression. postfix = “abc*+d”.

Add ‘d’ in the postfix

Final Step: Now no element is left. So empty the stack and add it in the postfix expression. postfix = “abc*+d+”.



Pop ‘+’ and add it in postfix