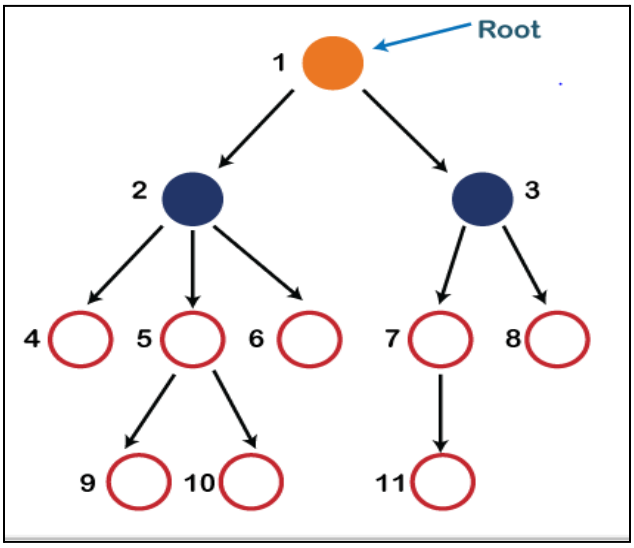
# TREES

## Introduction to trees:

A Tree can be defined as a collection of entities linked together to simulate a hierarchy.If there are n nodes, then there would be n-1 edges.

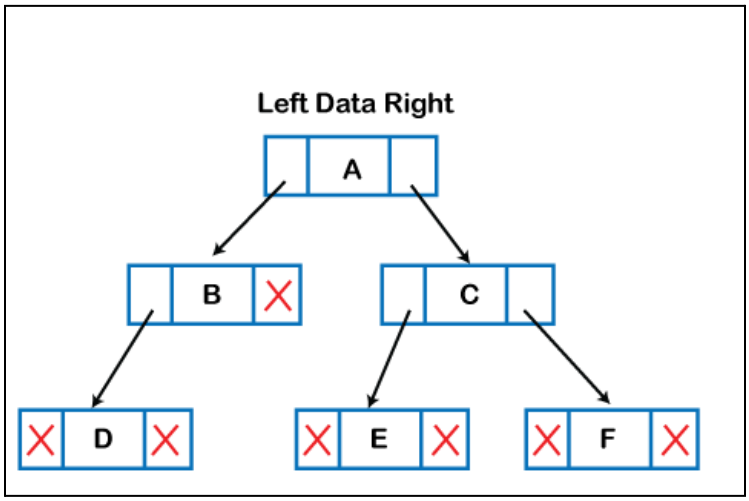



In the above structure, each node is labeled with some number. Each arrow shown in the above figure is known as a *link* between the two nodes.

- **Root:** The root node is the topmost node in the tree hierarchy. In other words, the root node is the one that doesn't have any parent. In the above structure, node numbered 1 is the root node of the tree.
- **Child node**: If the node is a descendant of any node, then the node is known as a child node.
- **Parent**: If the node contains any sub-node, then that node is said to be the parent of that sub-node.
- **Sibling**: The nodes that have the same parent are known as siblings.

- **Leaf Node**:- The node of the tree, which doesn't have any child node, is called a leaf node. A leaf node is the bottom-most node of the tree. There can be any number of leaf nodes present in a general tree. Leaf nodes can also be called external nodes.
- **Internal nodes**: A node has at least one child node known as an *internal*
- **Ancestor node**:- An ancestor of a node is any predecessor node on a path from the root to that node. The root node doesn't have any ancestors. In the tree shown in the above image, nodes 1, 2, and 5 are the ancestors of node 10.
- **Descendant:** The immediate successor of the given node is known as a descendant of a node. In the above figure, 10 is the descendant of node 5.

### Implementation of tree in memory:




## Applications of trees

The following are the applications of trees:

- Storing naturally hierarchical data: Trees are used to store the data in the hierarchical structure. For example, the file system. The file system stored on the disc drive, the file and folder are in the form of the naturally hierarchical data and stored in the form of trees.

- Organize data: It is used to organize data for efficient insertion, deletion and searching. For example, a binary tree has a logN time for searching an element.
- Tree: It is a special kind of tree that is used to store the dictionary. It is a fast and efficient way for dynamic spell checking.
- Heap: It is also a tree data structure implemented using arrays. It is used to implement priority queues.
- B-Tree and B+Tree: B-Tree and B+Tree are the tree data structures used to implement indexing in databases.
- Routing table: The tree data structure is also used to store the data in routing tables in the routers.

## Binary Search Tree

A binary search tree is a type of tree data structure that enables users to sort and store information. Because each node can only have two children, it is known as a binary tree. It is also known as a search tree because we can look up numbers in O(log(n)) time.

## Algorithm for a Binary Search Tree

Operations in  binary search tree, such as insertion and deletion of nodes and searching.

**Algorithm for search :**

1)check if tree is empty or not
If root == NULL
   return NULL;
2)check if current node contains data
If number == root->data
   return root->data;
3)Recursively search the left subtree if the number is smaller than the current node's data:
If number < root->data
   return search(root->left)
4)Recursively search the left subtree if the number is larger than the current node's data:
If number > root->data
   return search(root->right)

**Algorithm for insert search :**

**1)**Check if the node is NULL

If node == NULL
return createNode(data)

2)Recursively insert in the left subtree if the data is less than the current node's data:

```
if (data < node->data)
    node->left  = insert(node->left, data);
```

3)Recursively insert in the right subtree if the data is greater than the current node's data:

```
else if (data > node->data)
    node->right = insert(node->right, data);
```

4)Return the current node:

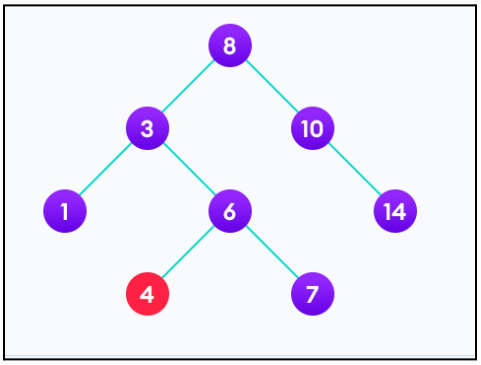
```
return node;
```

**Algorithm for delete :**

There are three cases for deleting a node from a binary search tree.

Case I

In the first case, the node to be deleted is the leaf node. In such a case, simply delete the node from the tree.

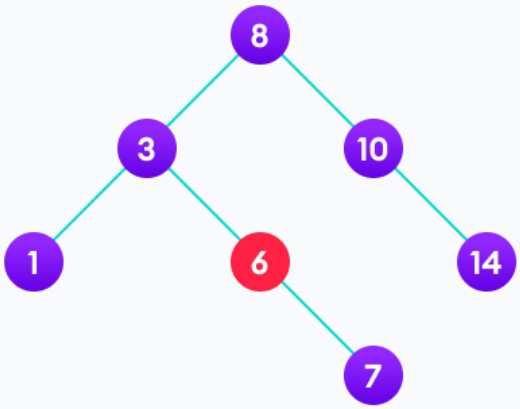



4 is to be deleted

### Case II

In the second case, the node to be deleted lies has a single child node. In such a case follow the steps below:

1. Replace that node with its child node.
2. Remove the child node from its original position.

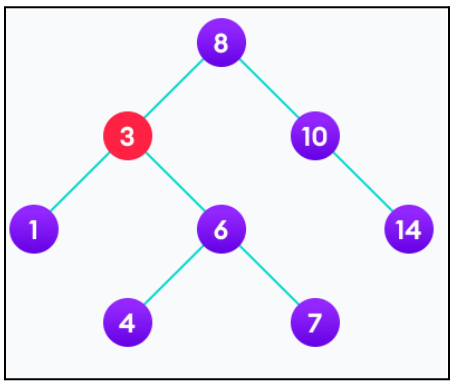



6 is to be deleted

**Case III**

In the third case, the node to be deleted has two children. In such a case follow the steps below:

1. Get the inorder successor of that node.
2. Replace the node with the inorder successor.
3. Remove the inorder successor from its original position.



3 is to be deleted

| Operation | Best Case Complexity | Average Case Complexity | Worst Case Complexity |
|---|---|---|---|
| Search | O(log n) | O(log n) | O(n) |
| Insertion | O(log n) | O(log n) | O(n) |
| Deletion | O(log n) | O(log n) | O(n) |

## C Program for Binary search Tree

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

// Definition of the node structure
struct TreeNode {
    int data;
    struct TreeNode* left;
    struct TreeNode* right;
};

// Function to create a new node
struct TreeNode* createNode(int data) {
    struct TreeNode* newNode = (struct TreeNode*)malloc(sizeof(struct TreeNode));
    newNode->data = data;
    newNode->left = NULL;
    newNode->right = NULL;
    return newNode;
}

// Function to insert a node into the BST
struct TreeNode* insert(struct TreeNode* node, int data) {
    // If the tree is empty, return a new node
    if (node == NULL)
```

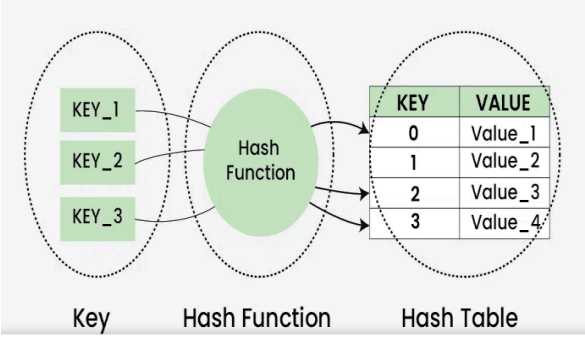
```
        return createNode(data);

    // Otherwise, recur down the tree
    if (data < node->data)
        node->left = insert(node->left, data);
    else if (data > node->data)
        node->right = insert(node->right, data);

    // Return the (unchanged) node pointer
    return node;
}

// Function to search a given key in a BST
struct TreeNode* search(struct TreeNode* root, int key) {
    // Base case: root is null or key is present at root
    if (root == NULL || root->data == key)
        return root;

    // Key is greater than root's key
    if (root->data < key)
        return search(root->right, key);

    // Key is smaller than root's key
    return search(root->left, key);
}

// In-order traversal to print the BST
void inorder(struct TreeNode* root) {
    if (root != NULL) {
        inorder(root->left);
        printf("%d ", root->data);
        inorder(root->right);
    }
}

int main() {
    struct TreeNode* root = NULL;
    root = insert(root, 50);
    insert(root, 30);
    insert(root, 20);
```

```
    insert(root, 40);
    insert(root, 70);
    insert(root, 60);
    insert(root, 80);

    // Print in-order traversal of the BST
    printf("In-order traversal: ");
    inorder(root);
    printf("\n");

    // Search for a key
    int key = 60;
    struct TreeNode* searchResult = search(root, key);
    if (searchResult != NULL)
        printf("Element %d found in the BST.\n", key);
       else
        printf("Element %d not found in the BST.\n", key);
        return 0;
}
```

# HASHING

## Introduction to hashing

Hashing in Data Structures refers to the process of transforming a given key to another value. It involves mapping data to a specific index in a hash table using a hash function.
The transformation of a key to the corresponding value is done using a Hash Function and the value obtained from the hash function is called Hash Code .




## Hash Function

A hash function is a type of mathematical operation that takes an input (or key) and outputs a fixed-size result known as a hash code or hash value. The hash function should produce a unique hash code for each input, which is known as the hash property.
There are different types of hash functions, including:

1.Division method : $h(k)=k \bmod m$
The table size(m) is 10 and the key is 23, the hash value would be 3 (23 % 10 = 3).

2.multiplication method : $h(k)=\lfloor m(kA \bmod 1) \rfloor$
The key is 23 and the constant is 0.618, the hash value would be 2 (floor(10*(0.61823 - floor(0.61823))) = floor(2.236) = 2).

3. Mid-Square Method
In the mid-square method, the key is squared, and the middle digits of the result are taken as the hash value.

4. Folding Method
The folding method involves dividing the key into equal parts, summing the parts, and then taking the modulo with respect to *mm*.

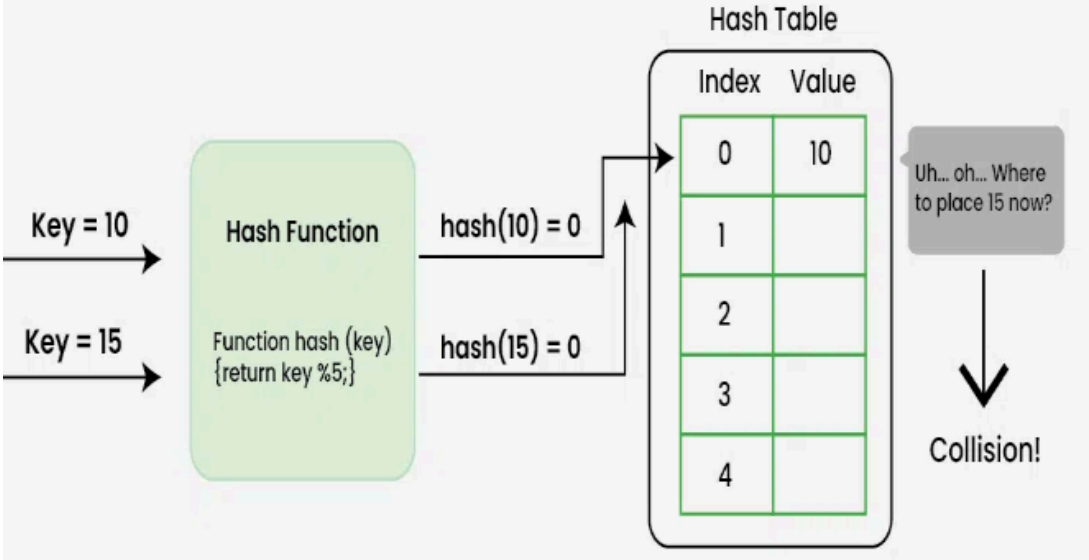
5. Cryptographic Hash Functions

Cryptographic hash functions are designed to be secure and are used in cryptography. Examples include MD5, SHA-1, and SHA-256.

6.universal hashing :
This method involves using a random hash function from a family of hash functions.

## What is Collision?

One of the main challenges in hashing is handling collisions, which occur when two or more input values produce the same hash value.




## Collision Resolution Techniques:

There are various techniques used to resolve collisions, including:

- Separate Chaining(Open Hashing): In this technique, each hash table slot contains a linked list of all the values that have the same hash value. This technique is simple and easy to implement, but it can lead to poor performance when the linked lists become too long.
- Open addressing(Closed Hashing): In this technique, when a collision occurs, the algorithm searches for an empty slot in the hash table by probing successive slots until an

empty slot is found. This technique can be more efficient than chaining when the load factor is low, but it can lead to clustering and poor performance when the load factor is high.

Different ways of Open Addressing

1. Linear Probing:This involves doing a linear probe for the following slot when a collision occurs and continuing to do so until an empty slot is discovered.

2. Quadratic Probing : When a collision happens in this, we probe for the i2-nd slot in the ith iteration, continuing to do so until an empty slot is discovered. In comparison to linear probing, quadratic probing has a worse cache performance. Additionally, clustering is less of a concern with quadratic probing.

3. Double hashing: you employ a different hashing algorithm, and in the ith iteration, you look for (i * hash 2(x)). The determination of two hash functions requires more time. Although there is no clustering issue, the performance of the cache is relatively poor when using double probing.

# Hash Table

**hash table** is a data structure that stores data in an array.Typically, a size for the array is selected that is greater than the number of elements that can fit in the hash table. A key is mapped to an index in the array using the hash function.

The hash function is used to locate the index where an element needs to be inserted in the hash table in order to add a new element. The element gets added to that index if there isn't a collision.If there is a collision, the collision resolution method is used to find the next available slot in the array.

**Implementation & Operations**

There are several operations that can be performed on a hash table, including:

- Insertion: Inserting a new key-value pair into the hash table.
- Deletion: Removing a key-value pair from the hash table.
- Search: Searching for a key-value pair in the hash table.

**Insertion**:To insert a key-value pair into a hash table, we first need to put an index into the array to store the key-value pair. If another key maps to the same index, we have a collision and need to handle it appropriately. One common method is to use chaining, where each bucket in the array contains a linked list of key-value pairs that have the same hash value.

```
typedef struct node {
    char* key;
    int value;
    struct node* next;
} node;
node* hash_table[100];
void insert(char* key, int value) {
    unsigned long hash_value = hash(key) % 100;
    node* new_node = (node*) malloc(sizeof(node));
    new_node->key = key;
    new_node->value = value;
    new_node->next = NULL;
    if (hash_table[hash_value] == NULL) {
        hash_table[hash_value] = new_node;
    } else {
        node* curr_node = hash_table[hash_value];
        while (curr_node->next != NULL) {
            curr_node = curr_node->next;
        }
        curr_node->next = new_node;
    }
}
```

**Explanation:**

- First, a struct called node is defined, which represents a single node in the hash table.
- Each node has three members: a char* key to store the key, an int value to store the associated value, and a pointer to another node called next to handle collisions in the hash table using a linked list.
- An array of node pointers called hash_table is declared with a size of 100. This array will be used to store the elements of the hash table.
- The insert function takes a char* key and an int value as parameters.
- It starts by computing the hash value for the given key using the hash function, which is assumed to be defined elsewhere in the program.
- The hash value is then reduced to fit within the size of the hash_table array using the modulus operator % 100.
- A new node is created using dynamic memory allocation (malloc(sizeof(node))), and its members (key, value, and next) are assigned with the provided key, value, and NULL, respectively.
- If the corresponding slot in the hash_table array is empty (NULL), indicating no collision has occurred, the new node is assigned to that slot directly (hash_table[hash_value] = new_node).

## Applications of Hashing

Hashing has many applications in computer science, including:

- Databases: Hashing is used to index and search large databases efficiently.
- Cryptography: Hash functions are used to generate message digests, which are used to verify the integrity of data and protect against tampering.

- Caching: Hash tables are used in caching systems to store frequently accessed data and improve performance.
- Spell checking: Hashing is used in spell checkers to quickly search for words in a dictionary.
- Network routing: Hashing is used in load balancing and routing algorithms to distribute network traffic across multiple servers.

## Advantages of Hashing:

- Fast Access: Hashing provides constant time access to data, making it faster than other data structures like linked lists and arrays.
- Efficient Search: Hashing allows for quick search operations, making it an ideal data structure for applications that require frequent search operations.
- Space-Efficient: Hashing can be more space-efficient than other data structures, as it only requires a fixed amount of memory to store the hash table.

## Limitations of Hashing:

- Hash Collisions: Hashing can produce the same hash value for different keys, leading to hash collisions. To handle collisions, we need to use collision resolution techniques like chaining or open addressing.
- Hash Function Quality: The quality of the hash function determines the efficiency of the hashing algorithm. A poor-quality hash function can lead to more collisions, reducing the performance of the hashing algorithm.