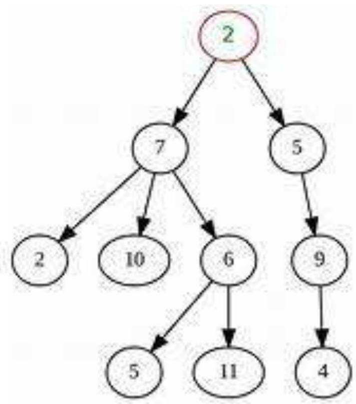
SYLLABUS

1.) Trees: Introduction to Trees,
2.) Binary Search Tree –
3.) Insertion, Deletion & Traversal
4.) Hashing: Brief introduction to hashing and hash functions,
5.) Collision resolution techniques:
6.) chaining and open addressing,
7.) Hash tables: basic implementation and operations,
8.) Applications of hashing in unique identifier generation, caching, etc.

1.) Introduction to Trees

- A tree data structure is a hierarchical structure that is used to represent and organize data in a way that is easy to navigate and search.
- It consists of a collection of nodes that are connected by edges and has a hierarchical relationship between the nodes.



- The topmost node of the tree is called the **root**, and the nodes below it are called **child nodes**.
- Each node can have multiple child nodes, and these child nodes can also have their own child nodes, forming a recursive structure.

Basic Terminologies in Tree Data Structure

1. **Parent Node**:
   - The node which is a predecessor of another node is called the parent node of that node.
   - Example: `{B}` is the parent node of `{D, E}`.
2. **Child Node**:
   - The node which is the immediate successor of another node is called the child node of that node.

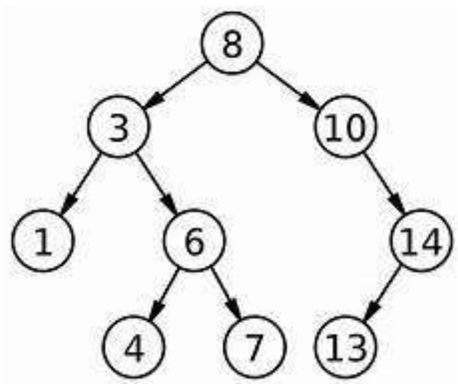
- Example: `{D, E}` are the child nodes of `{B}`.

3. **Root Node**:
    - The topmost node of a tree or the node which does not have any parent node is called the root node.
    - Example: `{A}` is the root node of the tree.
4. **Leaf Node or External Node**:
    - The nodes which do not have any child nodes are called leaf nodes.
    - Example: `{K, L, M, N, O, P, G}` are the leaf nodes of the tree.
5. **Ancestor of a Node**:
    - Any predecessor nodes on the path from the root to that node are called ancestors of that node.
    - Example: `{A, B}` are the ancestor nodes of the node `{E}`.
6. **Descendant**:
    - A node `x` is a descendant of another node `y` if and only if `y` is an ancestor of `x`.
7. **Sibling**:
    - Children of the same parent node are called siblings.
    - Example: `{D, E}` are called siblings.
8. **Level of a Node**:
    - The count of edges on the path from the root node to that node.
    - The root node has level 0.
9. **Internal Node**:
    - A node with at least one child is called an internal node.
10. **Subtree**:
    - Any node of the tree along with its descendants.

Remember, trees are used to represent hierarchical relationships and are essential for organizing data efficiently! □□

2.) Binary Search Tree (BST)

- A binary search tree is a binary tree with the following properties:
    - The left subtree of a node always contains keys less than the node's key.
    - The right subtree of a node always contains keys greater than the node's key.
    - Equal-valued keys are not allowed (no duplicate keys).



- Sometimes it is also referred to as an ordered binary tree or sorted binary tree.
- Searching in a BST is efficient, with best-case time complexity of Θ(log n). However, in the worst case (skewed tree), searching can take O(n).

3.) Operations on BST

1. Insertion

- To insert a key into a BST, compare the key with the root node:
    - If the key is smaller, move to the left subtree.
    - If the key is greater, move to the right subtree.
    - Repeat until a suitable position is found, and insert the key.
- Example: Inserting keys 50, 80, 30, 20, 100, and 40: !Insertion to BST

2. Deletion

- Deletion of a node can be performed as follows:

1. If the deleting node has no child, simply delete the node (make it point to NULL).
2. If the deleting node has 1 child, swap the key with the child and delete the child.
3. If the deleting node has 2 children, swap the key with the inorder successor (minimum key in the right subtree) and delete the successor.

- Example: Deleting node 30: !Deletion in BST

3. Searching

- The steps for searching are similar to insertion:
  - Compare the key with the root.
  - If not matched, repeat the steps until NULL is reached (key not found).
- Example: Searching for key 40: !Searching key in a BST

4. Traversal

- Inorder traversal of any BST outputs keys in non-decreasing order.
- Preorder and postorder traversals are also useful.
- Example code in C (pointer-based implementation):

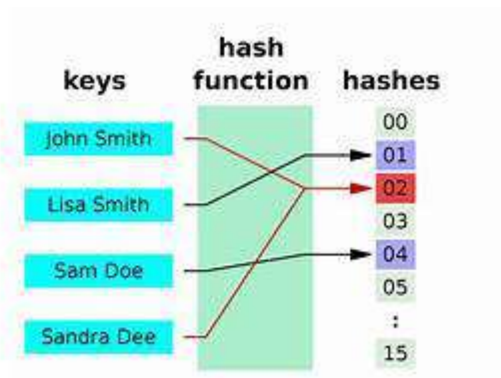
```c
#include <stdio.h>
#include <stdlib.h>

typedef struct node {
    int key;
    struct node* left;
    struct node* right;
} Node;

// Functions for creating new nodes, insertion, and traversal
(inorder, preorder, postorder)

int main() {
    // Create and use the BST
```

```
    return 0;
}
```

Remember, BSTs are powerful data structures for efficient searching and organizing data!

4.) Introduction to Hashing

- Hashing refers to the process of generating a fixed-size output from an input of variable size using mathematical formulas known as **hash functions**.
- This technique determines an index or location for the storage of an item in a data structure.



Components of Hashing

1. **Key**:
    - A key can be anything: a string or an integer.
    - It is fed as input to the hash function, which determines the index or location for storing the item.
2. **Hash Function**:
    - The hash function receives the input key and returns the index of an element in an array called a **hash table**.
    - The index is known as the **hash index**.

3. **Hash Table**:
   - A hash table is a data structure that maps keys to values using a special function called a hash function.
   - It stores data in an associative manner in an array, where each data value has its own unique index.

How Does Hashing Work?

- Suppose we have a set of strings: {“ab”, “cd”, “efg”}.
- Our objective is to store these strings in a table and search or update the values quickly in **constant time** (O(1)).
- Hashing allows us to achieve this efficiency by using hash functions to map keys to unique indices in the hash table.

Remember, hashing is a powerful technique for efficient data storage and retrieval! □□

5.) Collision Resolution Techniques

Certainly! In data structures, **collision resolution techniques** are essential for handling situations where multiple keys map to the same location (hash bucket) in a hash table. Let’s explore two common approaches:

1. **Separate Chaining (Open Hashing)**:
   - In this technique, each hash bucket contains a **linked list** of keys that collided.
   - When a collision occurs, the new key is simply **appended** to the linked list associated with that bucket.
   - The hash table is implemented as an **array of linked lists**.
   - **Advantages**:
     - Simple to implement.
     - Efficient for handling collisions.

- **Disadvantages**:
    - Requires additional memory for storing linked lists.
    - Linked list traversal can be slower than direct access.
- Example in C:

```
// Define a hash table with linked lists
struct Node {
    int key;
    struct Node* next;
};

struct Node* hashTable[SIZE]; // SIZE is the table size

// Initialize hash table
for (int i = 0; i < SIZE; ++i) {
    hashTable[i] = NULL;
}

// Insert a key into the hash table
void insert(int key) {
    int index = hashFunction(key);
    struct Node* newNode = createNode(key);
    newNode->next = hashTable[index];
    hashTable[index] = newNode;
}
```

2. **Open Addressing (Closed Hashing)**:
    - In this technique, when a collision occurs, the algorithm searches for the **next available slot** within the hash table.
    - Various methods (linear probing, quadratic probing, double hashing) determine the next slot to check.
    - The hash table is implemented as a **single array**.
    - **Advantages**:

- No additional memory overhead for linked lists.
- Better cache performance due to contiguous memory.

- **Disadvantages**:
  - Requires careful handling of deletion and resizing.
  - Clustering can occur, affecting performance.

Example in C:

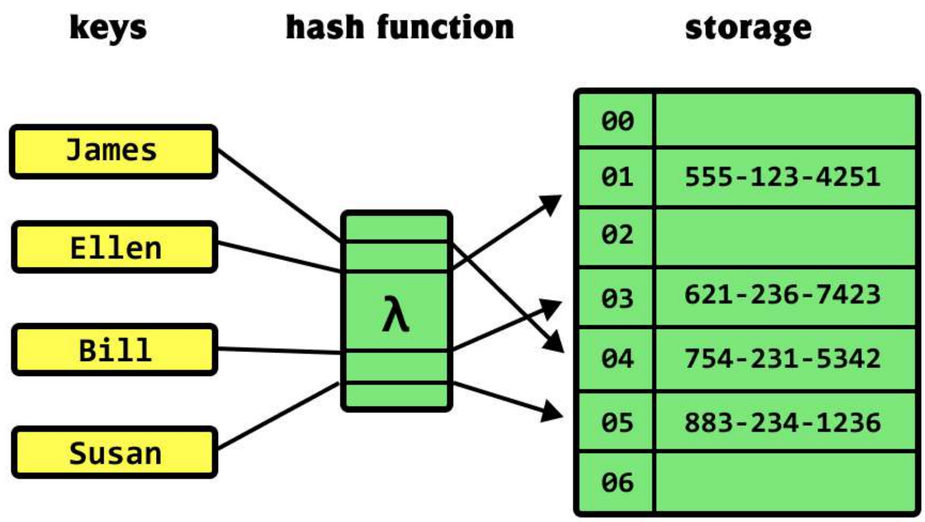
```
// Define a hash table using open addressing
int hashTable[SIZE]; // SIZE is the table size
int EMPTY = -1;

// Initialize hash table
for (int i = 0; i < SIZE; ++i) {
    hashTable[i] = EMPTY;
}

// Insert a key into the hash table
void insert(int key) {
    int index = hashFunction(key);
    while (hashTable[index] != EMPTY) {
        index = (index + 1) % SIZE; // Linear probing
    }
    hashTable[index] = key;
}
```

Remember that the choice of collision resolution technique depends on factors like memory usage, performance requirements, and ease of implementation.

7.) Hash Tables: An Overview

A **hash table** is a data structure that efficiently stores and retrieves **key-value pairs**. It operates based on the concept of **hashing**, where each key is transformed by a **hash**

function into a unique index within an array. This index serves as the storage location for the corresponding value. In simple terms, a hash table maps keys to their associated values.



Key Concepts:

- **Key**: A unique identifier used for indexing.
- **Value**: The data associated with a key.
- **Hash Function**: Transforms keys into array indices.
- **Collision**: When multiple keys map to the same index (conflict).

Collision Resolution Techniques:

1. **Chaining (Open Hashing)**:
    - Each hash bucket contains a **linked list** of keys that collided.
    - New keys are **appended** to the linked list associated with their bucket.
    - Implemented as an **array of linked lists**.

- **Advantages**:
  - Simple to implement.
  - Efficient for handling collisions.
- **Disadvantages**:
  - Requires additional memory for linked lists.
  - Linked list traversal can be slower.
- Example in C:

```
struct Node {
    int key;
    struct Node* next;
};
```

2. **Open Addressing (Closed Hashing)**:
   - Each slot stores either a single key or is left empty (NIL).
   - Techniques:
     - **Linear Probing**: Check the next slot linearly.
     - **Quadratic Probing**: Increased spacing between slots.
     - **Double Hashing**: Use another hash function for the next slot.
   - **Advantages**:
     - No extra memory for linked lists.
     - Better cache performance.
   - **Disadvantages**:
     - Careful handling of deletion and resizing.
     - Clustering can impact performance.

Good Hash Functions:

A good hash function minimizes collisions. Some methods include:

1. **Division Method**:
   - If `k` is the key and `m` is the table size:
     - `h(k) = k mod m`

- Example: For a table size of 10 and `k = 112`, `h(k) = 112 mod 10 = 2`.

Remember that choosing the right collision resolution technique and hash function depends on your specific use case and constraints[1234].

8.) Applications of hasing.

Certainly! **Hashing** plays a crucial role in various applications, including **unique identifier generation** and **caching**. Let's explore these use cases:

1. **Unique Identifiers (UIDs)**:
   - **Definition**: UIDs are string values or integers used to address unique resources in a domain. They allow consumers (systems or users) to refer to specific resources.
   - **Importance**: Without UIDs, distinguishing and accessing resources becomes nearly impossible.
   - **Generating UIDs with Hashing**:
     - Hashing algorithms, such as **MD5**, **SHA-1**, or **SHA-256**, can create metadata from general strings of information.
     - The hashed value represents the original data but is transformed in a way that makes it unique and unrecognizable.
     - Hash codes serve as both **integrity verification codes** and **identifiers**.
     - Example: When naming objects in an **Object Storage**, hashing ensures that each object version has a distinct identifier, even if the content changes[1].
2. **Caching**:
   - **Definition**: Caching involves storing frequently accessed data in a temporary storage (cache) to improve performance.
   - **Role of Hashing**:
     - Hash functions generate keys for storing data in a **hash table** (often used for caching).

        - The key represents the unique identifier of the data item.
        - When retrieving data, the hash function quickly locates the corresponding entry in the hash table, leading to efficient data retrieval.
    - **Example**: Web browsers use caching to store previously visited web pages. The hash-based keys allow quick access to cached content[2].

3. **Consistent Hashing**:
    - **Definition**: Consistent hashing is a technique used in distributed systems (e.g., load balancers, distributed caches).
    - **How It Works**:
        - Hash both keys and node identifiers (e.g., URLs or IP addresses).
        - Represent keys and nodes as hash values.
        - Ensures that when nodes are added or removed, only a fraction of keys need remapping.
    - **Benefits**: Scalability, fault tolerance, and efficient data distribution[3].

➔ Certainly! Here are some simple **C** example programs related to **hashing**:

1. **Hash Table with Separate Chaining (Linked Lists)**:
    - In this example, we create a hash table using separate chaining to handle collisions. Each bucket contains a linked list of key-value pairs.
    - We'll insert elements and demonstrate retrieval.

```
// Hash table using separate chaining (linked lists)
#include <stdio.h>
#include <stdlib.h>

struct Node {
    int key;
    int value;
    struct Node* next;
};
```

```
struct Node* hashTable[10]; // Assuming 10 buckets

int hashFunction(int key) {
    return key % 10; // Simple modulo-based hash function
}

void insert(int key, int value) {
    int index = hashFunction(key);
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    newNode->key = key;
    newNode->value = value;
    newNode->next = hashTable[index];
    hashTable[index] = newNode;
}

int search(int key) {
    int index = hashFunction(key);
    struct Node* current = hashTable[index];
    while (current != NULL) {
        if (current->key == key) {
            return current->value;
        }
        current = current->next;
    }
    return -1; // Key not found
}

int main() {
    // Initialize hash table
    for (int i = 0; i < 10; ++i) {
        hashTable[i] = NULL;
    }

    // Insert key-value pairs
    insert(42, 100);
```

```
    insert(17, 200);
    insert(31, 300);

    // Search for a value using a key
    int searchKey = 17;
    int result = search(searchKey);
    if (result != -1) {
        printf("Value associated with key %d: %d\n", searchKey,
result);
    } else {
        printf("Key %d not found.\n", searchKey);
    }

    // Clean up memory (free linked lists)
    for (int i = 0; i < 10; ++i) {
        struct Node* current = hashTable[i];
        while (current != NULL) {
            struct Node* temp = current;
            current = current->next;
            free(temp);
        }
    }

    return 0;
}
```

2. **Linear Probing (Open Addressing)**:
    - In this example, we use linear probing to handle collisions. If a slot is occupied, we check the next slot until an empty slot is found.
    - We'll insert elements and demonstrate retrieval.

```
// Hash table using linear probing (open addressing)
#include <stdio.h>
```

```
int hashTable[10]; // Assuming 10 slots
int EMPTY = -1;

int hashFunction(int key) {
return key % 10; // Simple modulo-based hash function
}

void insert(int key, int value) {
int index = hashFunction(key);
while (hashTable[index] != EMPTY) {
      index = (index + 1) % 10; // Linear probing
}
hashTable[index] = value;
}

int search(int key) {
int index = hashFunction(key);
while (hashTable[index] != EMPTY) {
      if (hashTable[index] == key) {
             return index;
      }
      index = (index + 1) % 10; // Linear probing
}
return -1; // Key not found
}

int main() {
// Initialize hash table
for (int i = 0; i < 10; ++i) {
      hashTable[i] = EMPTY;
}

// Insert key-value pairs
insert(42, 100);
insert(17, 200);
```

```
insert(31, 300);

// Search for a key
int searchKey = 17;
int result = search(searchKey);
if (result != -1) {
     printf("Key %d found at index %d.\n", searchKey, result);
} else {
     printf("Key %d not found.\n", searchKey);
}

return 0;
}
```