Outline

- Introduction to Trees
  - Binary Trees: Basic Definitions
  - Traversing Binary Trees
  - Node Representation of Binary Trees
  - Primitive Functions in Binary Trees

**Binary Trees: Basic Definitions**

- A binary tree is a finite set of elements that are either empty or is partitioned into three disjoint subsets. The first subset contains a single element called the root of the tree. The other two subsets are themselves binary trees called the left and right subtrees of the original tree. A left of the right subtree can be empty.

- Each element of a binary tree is called a node of the tree. The following figure shows a binary tree with 9 nodes where A is the root.

![Binary Tree Diagram](image)

- A node that has no sons is called the leaf.
- Node $n_1$ is the ancestor of node $n_2$ if $n_1$ is either the father of $n_2$ or the father of some ancestor of $n_2$. In such a case $n_2$ is a descendant of $n_1$.
- Two nodes are brothers if they are left and right sons of the same father.

![Binary Tree Diagram with Relationships](image)
If every nonleaf node in a binary tree has nonempty left and right subtrees, the tree is called a **strictly binary tree**.

A Strictly Binary Tree

The **level** of a node in a binary tree is defined as follows: The root of the tree has level 0, and the level of any other node in the tree is one more than the level of its father.

The **depth** of a binary tree is the maximum level of any leaf in the tree.

A **complete binary tree** of depth $d$ is the strictly binary all of whose leaves are at level $d$. A complete binary tree with depth $d$ has $2^d$ leaves and $2^d - 1$ nonleaf nodes.

A complete Binary Tree of depth 3

**Traversing Binary Trees**

One of the common operations of a binary tree is to **traverse** the tree. Traversing a tree is to pass through all of its nodes once. You may want to print the contents of each node or to process the contents of the nodes. In either case each node of the tree is visited.

There are three main traversal methods where traversing a binary tree involves visiting the root and traversing its left and right subtrees. The only difference among these three methods is the order in which these three operations are performed.
Traversing a binary tree in **preorder** (depth-first order)
1. Visit the **root**.
2. Traverse the **left subtree** in preorder.
3. Traverse the **right subtree** in preorder.

Traversing a binary tree in **inorder** symmetric order)
1. Traverse the **left subtree** in inorder.
2. Visit the **root**.
3. Traverse the **right subtree** in inorder.

Traversing a binary tree in **postorder** symmetric order)
1. Traverse the **left subtree** in postorder.
2. Traverse the **right subtree** in postorder.
3. Visit the **root**.

---

**Node Representation of Binary Trees**

Each node in a binary tree contains **info**, **left**, **right** and **father** fields. The left, right and father fields points the node’s left son, right son and the father respectively.

```c
struct node{
    int info; /* can be of different type*/
    struct node *left;
    struct node *right;
    struct node *father;
};
typedef struct node nodeptr;
```
**Primitive Functions in Binary Trees**

The **maketree** function allocates a node and sets it as the root of a single node binary tree.

```c
nodeptr *maketree(int x) {
    nodeptr *p;
    p = getnode();
    p->info = x;
    p->left = NULL;
    p->right = NULL;
    return p;
}
```

The **setleft** and **setright** functions sets a node with content \( x \) as the left son and right son of the node \( p \) respectively.

```c
void setleft(nodeptr *p, int x) {
    if(p == NULL) {
        printf("void insertion\n");
    } else if (p->left != NULL) {
        printf("invalid insertion\n");
    } else {
        p->left = maketree(x);
    }
}

void setright(nodeptr *p, int x) {
    if(p == NULL) {
        printf("void insertion\n");
    } else if (p->right != NULL) {
        printf("invalid insertion\n");
    } else {
        p->right = maketree(x);
    }
}
```

**Binary Tree Traversal Methods**

Recursive functions can be used to perform traversal on a given binary tree. Assume that dynamic node representation is used for a given binary tree.

In the following traversal methods, the tree is traversed always in downward directions. Therefore the father field is not needed.

The following recursive **preorder traversal** function displays the info part of the nodes in preorder. Note that the info part is integer number and tree is a pointer to the root of the tree.

```c
void pretrav(nodeptr *tree) {
    if(tree != NULL) {
        printf("%d\n", tree->info);
        pretrav(tree->left);
        pretrav(tree->right);
    }
}
```
The following recursive *inorder traversal* function displays the info part of the nodes in inorder. Note that the info part is integer number and tree is a pointer to the root of the tree.

```c
void intrav(nodeptr *tree)
{
    if(tree != NULL)
        intrav(tree->left);
    printf("%d\n", tree->info);
    intrav(tree->right);
}
```

The following recursive *postorder traversal* function displays the info part of the nodes in postorder. Note that the info part is integer number and tree is a pointer to the root of the tree.

```c
void posttrav(nodeptr *tree)
{
    if(tree != NULL)
        posttrav(tree->left);
    posttrav(tree->right);
    printf("%d\n", tree->info);
}
```

**Binary Search Tree: An Application of Binary Trees**

A binary tree, that has the property that all elements in the left subtree of a node \( n \) are less than the contents of \( n \), and all elements in the right subtree of \( n \) are greater than or equal to the contents of \( n \), is called a **Binary Search Tree** or Ordered Binary Tree.

Given the following sequence of numbers,

\[
14, 15, 4, 9, 7, 18, 3, 5, 16, 4, 20, 17, 9, 14, 5
\]

The following binary search tree can be constructed.

![A Binary Search Tree](image-url)
The **inorder (left-root-right)** traversal of the above Binary Search Tree and printing the info part of the nodes gives the **sorted sequence in ascending order**. Therefore, the Binary search tree approach can easily be used to sort a given array of numbers.

The inorder traversal on the above Binary Search Tree is:

3, 4, 4, 5, 5, 7, 9, 9, 14, 14, 15, 16, 17, 18, 20

**SEARCHING THROUGH THE BINARY SEARCH TREE**

- Searching operation of the binary search tree is always in downward direction. Therefore the following node structure can be used to represent the node of a given binary search tree.

  - Note that the father link is not required.

  ```c
  struct node{
    int info; /* can be of different type*/
    struct node *left;
    struct node *right;
  };

typedef struct node nodeptr;
  ```

- The following recursive function can be used to search for a given key element in a given array of integers. The array elements are stored in a binary search tree. Note that the function returns TRUE (1) if the searched key is a member of the array and FALSE (0) if the searched key is not a member of the array.

  ```c
  int BinSearch(nodeptr *p, int key)
  {
    if(p == NULL)  
      return FALSE;
    else {
      if (key == p->info)
        return TRUE;
      else{
        if(key < p->info)
          return BinSearch(p->left, key);
        else
          return BinSearch(p->right, key);
      }
    }
    
  }
  ```
The following recursive function can be used to insert a new node into a given binary search tree.

```c
nodeptr *insert(nodeptr *p, int x)
{
    if(p == NULL)
    {
        p = getnode();
        p->info = x;
        p->left = NULL;
        p->right = NULL;
        return p;
    }
    else
    {
        if(x < p->info)
            p->left = insert(p->left, x);
        else
            p->right = insert(p->right, x);
        return p;
    }
}
```