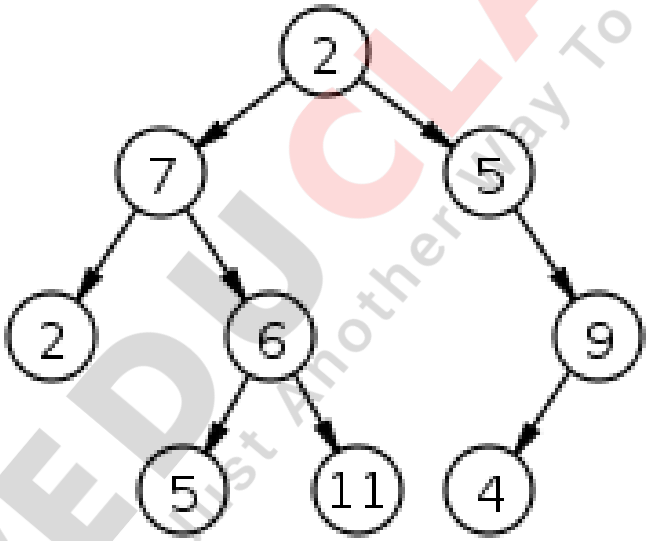
Binary Trees



Usage of binary trees

- Used for representing algebraic formulas.
- Used for searching large , dynamic lists.
- Iterative tree traversals

Terms used in trees

- **Nodes**:
  - Finite set of elements.
- **Branches**:
  - Finite set of directed lines.
- **Degree** :
  - Number of branches associated with the node.
- **Indegree**:
  - Branch directed towards the node.
- **Outdegree**:
  - Branch directed away from the node.

Terms used in trees

- **Degree of the node= indegree branches + outdegree branches**
- The **first node is** called as the **root.**
- The **indegree of the root is always zero.**
- **All nodes other than the root must have an indegree of exactly one. i.e they may have exactly one predecessor.**
- **All nodes** in the tree can have **zero , one or more branches. i.e** they may have **outdegree of zero, one or more.**

Terms used in trees

- **Leaf node/Terminal Nodes**:
  - Any node with an **outdegree zero. i.e a node** with **no successors.**
- **Internal node**:
  - A node that is **not a root or a leaf.**
- **Parent node**:
  - If a node has **successor nodes. i.e** if it has an **outdegree greater than zero**.
- **Child node**:
  - If a node has a **predecessor.** i.e a **child node has an indegree one.**

Terms used in trees

- **Siblings:**
  - **Two or more nodes** with the **same parent.**
- **Ancestor:**
  - any node in the path from the **root to the node**.
- **Descendant:**
  - All nodes in the path **from a given node to a leaf.**

Terms used in trees

- **Path**:
  - Sequence of nodes in which each node is adjacent to the next one.
- **Level**:
  - Distance from the root.
  - **Level of the root is zero.**
- **Height of the tree/depth of the tree:**
  - **Level of the leaf in the longest path from the root + 1**
  - Height of the empty tree is -1.

Terms used in trees

- **Size of the tree:**
  - **Number of nodes in the tree.**
- **Examples:**

Definition of a binary tree

- It is a tree in which nonode can have more than 2 subtrees.
- **The maximum outdegree for a node is two**.
- A node can have **zero , one or two subtrees.**

Other Definitions of Binary Tree

- A binary tree may also be defined as follows:-
  - A binary tree is a empty tree
  - A binary tree consists of a node called root, a left subtree and a right subtree both of which are binary trees once again.

- Examples

Properties of binary trees

- **Maximum height:**

  $H_{max} = N$ where N = nodes in binary tree.

- **Minimum height:**

  $H_{min} = [\log_2 N] + 1$

- **Minimum nodes:**
  - **Given the height of the binary tree, H**

    $N_{min} = H$

- **Maximum nodes:**
  - **Given the height of the binary tree, H**

    $N_{max} = 2^H - 1$

Properties of binary trees

- Balance factor of a binary tree:
  - Difference in height between its left and right subtrees.

    $$\mathbf{BF = H_{Left} - H_{Right}}$$
- In a Balanced binary tree, the height of its subtrees differs by no more than 1(i.e its balance factor is either -1,0,1)

General Trees

- A general tree(sometimes called as a tree) is defined as a non-empty finite set T of elements, called nodes such that:
  - The tree contains the root node.
  - The remaining nodes of the tree form an ordered collection of zero or more disjoint trees T1, T2….TM.
- Eg:

Converting a general tree to a binary tree

- Identify the branch from the parent to the first child.The branches from parent become the left pointer in the binary tree.
- Connect the siblings with far-left child.
- Remove all unneeded branches from the parent to its children

Strictly binary tree

- A binary tree where each node is
  - either a leaf or
  - is an internal node with exactly two non-empty children.
  - i.e a node is allowed to have none or 2 children
- A strictly binary tree with n leaves always contains 2n-1 nodes.
- Eg:

Complete Binary Trees

- A **complete** binary tree has the maximum number of entries for its height.
- A ***complete binary tree of height h*** *is a binary* tree which contains exactly 2^l *nodes at level l,* $0 \leq d \leq h$.

  where l= level of the node N is the length of the node from the root to node N.
- A complete binary tree with **L levels** contains **a total of (**2^h*-1)nodes*
- **Eg:**

Nearly Complete Binary Trees

- A ***binary tree of level L is a almost complete/nearly complete*** binary tree if level 0 to L-2 are full and level L-1 is being filled from left to right
- An almost complete binary tree with *N* leaves that is not strictly binary has 2*N* nodes
- Eg:

Expression Trees

- An expression is a **sequence of tokens**
- **A token is either an operand or an operator**
- An expression tree is a binary tree with the following properties:
  - Each leaf is an operand
  - The root and the internal nodes are operators
  - Subtrees are subexpressions with the root being the operator
- Egs

Expression tree traversals

- 3 standard traversals are as follows:
  - Infix,
  - Postfix and
  - Prefix
- The inorder traversal produces infix expression , the postorder traversal produces postfix expression and the preorder traversal produces the prefix expression

Infix traversal algorithm

Algorithm infix(tree)

If (tree not empty)

a. if (tree token is an operand)
    a. Display operand
b. else
    a. Display open parenthesis
    b. Infix(left subtree)
    c. Display token
    d. Infix(right subtree)
    e. Display close parenthesis
c. endif

endif

Algorithm postfix(tree)

1. If (tree not empty)
    1. postfix(left subtree)
    2. postfix(right subtree)
    3. Display token
2. End if

Algorithm prefix(tree)

1. If (tree not empty)
    1. Display token
    2. prefix(left subtree)
    3. prefix(right subtree)
2. End if

Constructing an expression tree

- Converting a postfix expression to a expression tree:
  - Read the expression one symbol at a time
  - If the symbol is an operand, create a one-node tree and push a pointer to it onto a stack
  - If the symbol is an operator, pop pointers to trees T1 and T2 from the stack (T1 popped first) and form a new tree whose root is the operator and whose left and right children point to T1 and T2 respectively.
  - A pointer to this new tree is then pushed onto the stack
  - Eg:

Forest

- Set of several trees not linked to each other in any way
- Steps in converting the forest as binary trees:
  - Left most trees is represented as binary trees
  - Second tree is made the right child of the root node of the first tree
  - Third tree is made the right child of the root node of the second tree and so on….

Huffman Code

- The **ASCII is a fixed length code**.Each ASCII character is a 7 bit code.
- Every character uses **the maximum number of bits.**
- **Huffman code** makes character storage **more efficient.**
- **In Huffman coding ,shorter codes** are assigned to **characters that occur more frequently and longer codes** are assigned to **characters that occur less frequently.**

Huffman Code

- Example:
  - **E and T occur frequently**.Therefore assign one bit each.
  - **A,O ,R and N occur less frequently than E and T.** Therefore assign two bits each.
  - **U, I, D , M, C are next most frequent.** Therefore assign three bits each and so on…

Fixed Length Code

Fixed-Length Code	a	b	c	d	e	f	g
Frequency	37	18	29	13	30	17	6
Fixed-length	000	001	010	011	100	101	110

Total size is:
(37 + 18 + 29 + 13 + 30 + 17 + 6) x 3= 450 bits

Variable-Length Code

Fixed-Length Code	a	b	c	d	e	f	g
Frequency	37	18	29	13	30	17	6
Variable-lengthcode	10	011	111	1101	00	010	1100

Total size is:
37x2 + 18x3 + 29x3 + 13x4 + 30x2 + 17x3 + 6x4 = 402 bits

- A savings of approximately 11%

Huffman Code

- Usage:
  - Used in a **network transmission. The overall length of the transmission is shorter if Huffman encoded characters are transmitted rather than fixed-length encoding.**Huffman code is therefore **a popular data compression algorithm.**
  - **Saves transmission time.**

Huffman Code

- Assign each character a weight based on its frequency of use.
- **Steps in building a tree based on Huffman coding.**

1. Organize the entire character set into a row, ordered according **to frequency from highest to lowest.Each character is now a node at the leaf level of a tree.**
2. **.**

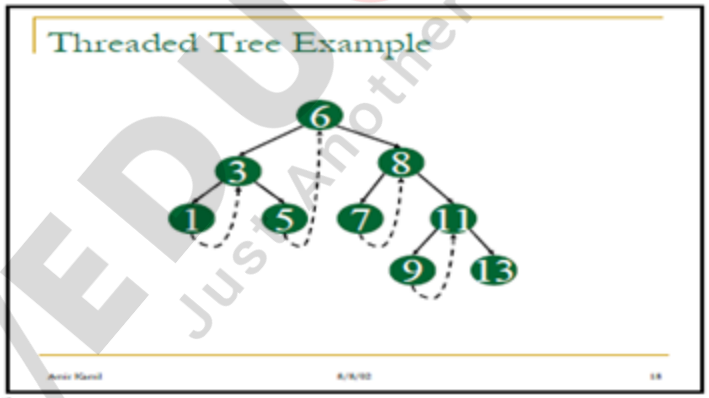
Huffman Code

2. Find the **2 nodes with the smallest combined frequency weights** and join them to form simple **two-le** **sum(v** **chose** **er possible choic** **ree.**

3. **Repe** **, on every level a**

Binary Threaded Tree

- It is a binary tree in which every node that does not have a right child has a thread(also known as link) to its inorder successor.
- Eg:



Binary Search Tree(BST)

- A **Binary Search Tree(BST)** is a binary tree with the following properties:-
  - All items in the left subtree are less than the root.
  - All items in the right subtree are greater than or equal to the root.
  - Each subtree is itself a binary search tree.
  - Eg:

Algorithms of Binary Search Tree

Insert node(Recursive) in a Binary Search Tree

```
struct bsttreenode * insert_bst(struct bsttreenode *t,int x)
1. if(t==NULL)
        1. t=dynamically allocate memory
        2. t->leftchild=NULL
        3. t->info=x
        4. t->rightchild=NULL
        5. return(t)
2. if(x<= t->info)
        t->leftchild=insert_bst(t->leftchild,x)
   else
        t->rightchild=insert_bst(t->rightchild,x)
   return(t)
```

Delete node from BST

void delete_bst(struct bsttreenode *t, int datatobedeleted)

1.[Declare]

found=0

struct bsttreenode *q,*parent, *x,*xsucc

2. if(t=NULL)

a. Display “TREE IS EMPTY"

b. return

3. [Initialize]

parent=x=NULL

q=t

4. Repeat a,b,c while(q!=NULL)

a.if(datatobedeleted=q->info)

1.found=1

2.x=q

break

b.parent=q

c. if(datatobedeleted<q->info)

1.q=q->leftchild

else

2.q=q->rightchild

5. if(found=0)

a. Display “DATA TO BE DELETED NOT FOUND"

b. return

Delete node from BST

6. //if data to be deleted has two children
   1. if(x->leftchild!=NULL && x->rightchild!=NULL)
      a. parent=x
      b.xsucc=x->rightchild
      c. Repeat while 1,2 (xsucc->leftchild !=NULL)
         1.parent=xsucc
         2.xsucc=xsucc->leftchild
      d. x->info=xsucc->info
      e. x=xsucc

7.//if the node to be deleted has no child
   1.if(x->leftchild==NULL && x->rightchild==NULL)
      a. if(parent->rightchild=x)
         1. parent->rightchild=NULL
      else
         2. parent->leftchild=NULL
      b. free(x)
      c. return

Delete node from BST

8. //if the node to be deleted has only right child
   1. if(x->leftchild==NULL && x->rightchild!=NULL)
      a. if(parent->leftchild=x)
         1.parent->leftchild= x->rightchild
         else
         2.parent->rightchild= x->rightchild
      b.free(x)
      c.return

9. //if the node to be deleted has only left child
   1. if(x->leftchild!=NULL && x->rightchild==NULL)
      a. if(parent->leftchild=x)
         1.parent->leftchild= x->leftchild
         else
         2.parent->rightchild= x->leftchild
      b. free(x)
      c. return

Search for a node in BST

int search_bst(
struct bsttreenode *t,
int targetkey)

1. [Declare] & Initialize]
   struct bsttreenode *q
   found=0
   q=t
2. if(q=NULL)
   a. found=0
   b. return found
3. while(q!=NULL)
   1. if(q->info =targetkey)
      a. found=1
      b return found
   2.if(targetkey< q->info)
      a. q=q->leftchild
   3.if(targetkey> q->info)
      a. q=q->rightchild

return found

To find the smallest node in the BST

```
struct bsttreenode *
        smallest_node_in_bst(struct bsttreenode *t)
1. [Declare and Initialize]
          struct bsttreenode *q
           q=t
2.  if(q->leftchild=NULL)
          1. Display " NO LEFT SUBTREE"
           2. return t
3.  return smallest_node_in_bst(q->leftchild)
```

To find the largest node in the BST

```
struct bsttreenode *
        largest_node_in_bst(struct bsttreenode *t)
1. [Declare and Initialize]
          struct bsttreenode *q
           q=t
2.  if(q->rightchild=NULL)
          1. Display “ NO LEFT SUBTREE"
           2. return t
3.  return largest_node_in_bst(q->rightchild)
```

Recursive Preorder traversal in a binary search tree

void preorder_display_bst(struct bsttreenode *t)

1. if(t!=NULL)
    1. Display the info contained in the root
    2. preorder_display_bst(t->leftchild)
    3. preorder_display_bst(t->rightchild)

Recursive Inorder traversal in a binary search tree

void inorder_display_bst(struct bsttreenode *t)

1. if(t!=NULL)
    1. inorder_display_bst(t->leftchild)
    2. Display the info contained in the root
    3. inorder_display_bst(t->rightchild)

Recursive postorder traversal in a binary search tree

void postorder_display_bst(struct bsttreenode *t)

1. if(t!=NULL)
    1. postorder_display_bst(t->leftchild)
    2. postorder_display_bst(t->rightchild)
    3. Display the info contained in the root

Breadth First Traversal in a Binary Search Tree

void Breadth_first_Traversal_bst(struct bsttreenode *t)

1.[Declare and Initialize]

struct bsttreenode *queue[100] = Initialize to 0

size = 0

queue_pointer = 0

2. Repeat 1 to 4 while(t)

1. Display t->info
2. if(t->left)

a. queue[size++] = t->leftchild

3. if(t->right)

a. queue[size++] = root->rightchild

4. t = queue[queue_pointer++]

Algorithms to implement non-recursive insertion, preorder, inorder and postorder traversal in a BST

Inserting in a BST non recursively

Algorithm
insert_into_non_rec_bst(
struct bsttreenode *t,int v)

1.[Declare]
struct bsttreenode *temp,*pr

2. temp=Dynamically allocate memory
temp->leftchild=NULL
temp->info=v
temp->rightchild=NULL

3. if(non_rec_bst==NULL)
non_rec_bst=temp
else

1. Repeat a and b
while(t!=NULL)
a. pr=t
b. if(v<t->info)
t=t->leftchild
else
t=t->rightchild
2. t=temp
3. if(v<pr->info)
pr->leftchild=temp
else
pr->rightchild=temp

Non-recursive preorder traversal

1. Do 1,2
    1. Repeat 1 to 4 while(t!=NULL)
        1. Display t->info
        2. increment top
        3. stk[top]=t;
        4. t=t->leftchild
    2. if(top!=-1)
        1. t=stk[top]
        2. Decrement top
        3. t=t->rightchild

while(t!=NULL || top!=-1)

Non-recursive Inorder traversal

1. Do 1,2
    1. Repeat 1,2,3 while(t!=NULL)
        1. Increment top
        2.stk[top]=t
        3. t=t->leftchild
    2. if(top!=-1)
        1.t=stk[top]
        2. Decrement top
        3. Display t->info
        4. t=t->rightchild

while(t!=NULL || top!=-1)

Non-recursive Postorder traversal

Do 1,2

1. Repeat 1,2,3 while(t!=NULL)
    1. Increment top
    2. stk[top]=t
    3. t=t->leftchild
2. if(top!=-1)
    1.t=stk[top]
    2.if(t->rightchild !=NULL && t->rightchild != non_rec_bst)
        1. t=t->rightchild
    else
        1. non_rec_bst=stk[top]
        2. Decrement top
        3. Display non_rec_bst->info
        4. t=NULL

while(t!=NULL || top!=-1)