Data Structures

R. K. Ghosh

IIT Bhilai

Binary Trees

Trees

Definition of a Node

A non-divisible unit of information (record) of a large data structure such as a linked list. A node may also contain links (pointers) to other nodes.

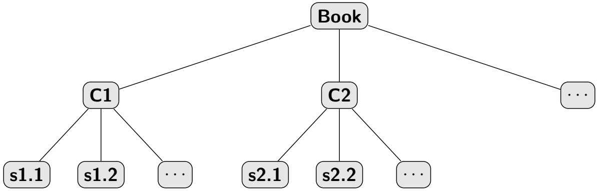
Recursive Definition of a Tree

A tree T can be empty, or may consist of

1. A special node r called the root.
2. A set of trees k trees $T_1, T_2, \ldots, T_k$ (possibly empty) with roots $r_1, r_2, \ldots, r_k$ respectively.

T is constructed by making $r_1, r_2, \ldots, r_k$ as children of r. Tree $T_1, T_2, \ldots, T_k$ are called subtrees of T.

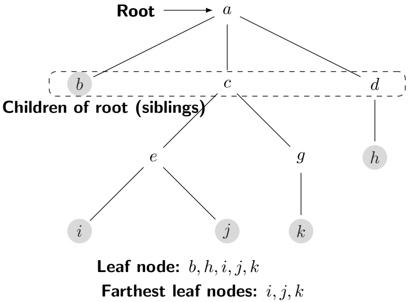
Tree



Terminology

- A tree is drawn by a figure of the type as shown.
- Parent child (relations shown by lines.
- A path: $n_1, n_2, \ldots, n_k$, such that $n_i = parent(n_{i-1})$.
- Length of a path is 1 less than number of nodes.

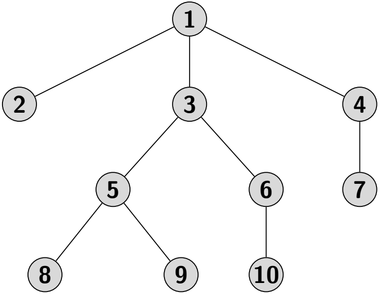
Tree



More Terminology

- a is an ancestor of all nodes including itself.
- All nodes including a are descendants of a.
- Ordered tree: siblings are ordered from left to right.
- k-ary tree: no node has more than k children.

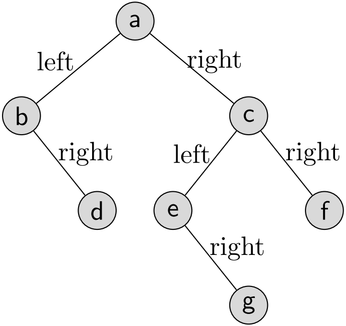
Ordered Trees



Positioning Nodes

- Ordered trees have special importance.
- Ordering can be used to identify position of a node.
- E.g., node 8 is to the left or 3, 4, 5, 6, 7, 9, and 10.
- But it is neither to left or right of ancestors 1, 3, 5.

Ordered Tree



Binary Trees

- If arity $k = 2$, then we have a binary tree.
- We distinguish between two children as left and right.
- Pictorial convention is to draw left child extended to the left and right child to right.

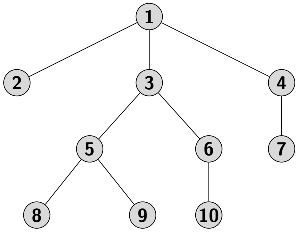
Tree Traversal

- We can systematically order nodes of a tree in many ways.
- Three most important ordering are: Preorder, Postorder and Inorder.
- Recursive definition of these orderings are as follows:
  - If a tree T is empty then empty list is the preorder, postorder and inorder listing of T.
  - If T consist of only one node, then the node by itself is the listing in all three orderings.
- Otherwise, let T be a tree with root r and k subtrees $T_1, T_2, \ldots, T_k$.

Tree Traversal

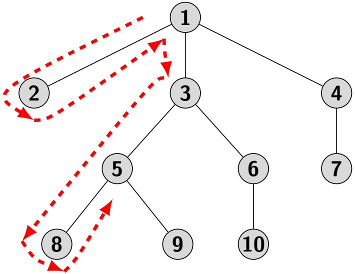
- Preorder listing of T: list root r of T, preorder list of all subtree $T_1, T_2, \ldots, T_k$ in left to right order.
- Postorder listing of T: postorder list of all subtree $T_1, T_2, \ldots, T_k$ in left to right order followed by root r of T.
- Inorder listing of T: inorder listing of T_1 followed by root r and then inorder listing of each group of nodes $T_2, T_3, \ldots, T_k$ in inorder.

Tree Traversal



- Preorder listing: 1, 2, 3, 5, 8, 9, 6, 10, 4,7.
- Postorder listing: 2, 8, 9, 5, 10, 6, 3, 7, 4, 1.
- Inorder listing: 2, 1, 8, 5, 9, 3, 10, 6, 7, 4.

Tree Traversal



Walk Around the Tree

- A walk around the tree can be used to produce listings:
- Walk outside the tree, starting at the root,
- Stay as close to the tree as possible.
- Move counter clockwise until reaching back to the starting point.

Tree Traversal

- For preorder traversal: list the node first time it is encountered in walk around the tree.
  - Listing will be: 1, 2, 3, 5, 8, 9, 6, 10, 4, 7.
- For inorder traversal: list the node second time it is encountered.
  - Listing will be: 2, 1, 8, 5, 9, 3, 10, 6, 7, 4.
- For postorder traversal: list the node last time it is encountered in the walk.
  - Listing will be: 2, 8, 9, 5, 10, 6, 3, 7, 4, 1.

Tree Structure

```
typedef struct treenode {
    int info;
    struct treenode * left;
    struct treenode * right;
} TREENODE;

void preOrder(TREENODE *t) {
    if (t != NULL) {
        printf("%d\t", t->info);
        preOrder(t->left);
        preOrder(t->right);
    }
}
```

Create Tree

```
TREENODE * create() {
    int x;
    TREENODE *p;
    printf("Enter data (-1 for no data): ") ;
    scanf("%d", &x);
    if (x == -1)
        return NULL;
    p = (TREENODE *) malloc(sizeof(TREENODE)) ;
    if (p == NULL) {
        printf("Error in malloc\n");
        exit(1);
    }
    p->info = x;
    printf("Enter leftchild of %d: \n", x) ;
    p->left = create() ;
    printf("Enter rightchild of %d: \n", x ) ;
    p->right = create() ;
    return p;
}
```

Postorder and Inorder

```
void postOrder(TREENODE *t) {
    if (t != NULL) {
        postOrder(t->left);
        postOrder(t->right);
        printf("%d\t", t->info);
    }
}

void inOrder(TREENODE *t) {
    if (t != NULL) {
        inOrder(t->left);
        printf("%d\t", t->info);
        inOrder(t->right);
    }
}
```

Membership Search

- Proceed like preorder.
  - Look for x in the current node.
  - If not found, search x in left subtree.
  - If not found in left subtree, search right subtree

Code Snippet for Membership Search

```
TREENODE * search(TREENODE * t, int x) {
    TREENODE *p ;
    if ((t == NULL) || ( t->info == x))
        return t ;
    p = search(t->left , x);
    if (p == NULL)
         p = search(t->right , x) ;
    return p;
}
```

Tree Height

- Proceed like postorder.
  - Recursively find height of left subtree.
  - Recursively find height of right subtree.
  - Find maximum of two heights.
  - Add 1 to the computed maximum.

Code Snippet for Tree Height

```
int treeHeight(TREENODE *t) {
   int rHeight, lHeight;
   int maxHeight;
   int i;
   if(t == NULL) {
     return 0;
   } else {
      maxHeight = 0;
      lHeight = treeHeight(t->left);
      rHeight = treeHeight(t->right);
      if (rHeight > lHeight) {
          if (maxHeight < rHeight)
              maxHeight = rHeight;
      } else {
          if (maxHeight < lHeight)
              maxHeight = lHeight;
     }
     return maxHeight + 1;
    }
}
```

Tree Size

```
int treeSize(TREENODE *t) {
    int size;
    if (t == NULL)
        return 0;
    else {
        size = 1 + treeSize(t->left) + treeSize(t->right);
        return size;
    }
}
```

- Also proceed like postorder.
- Recursively compute sizes of both right and left subtrees.
- Add both sizes and add 1 for the root.