Computing Laboratory

Binary Search Trees

Malay Bhattacharyya

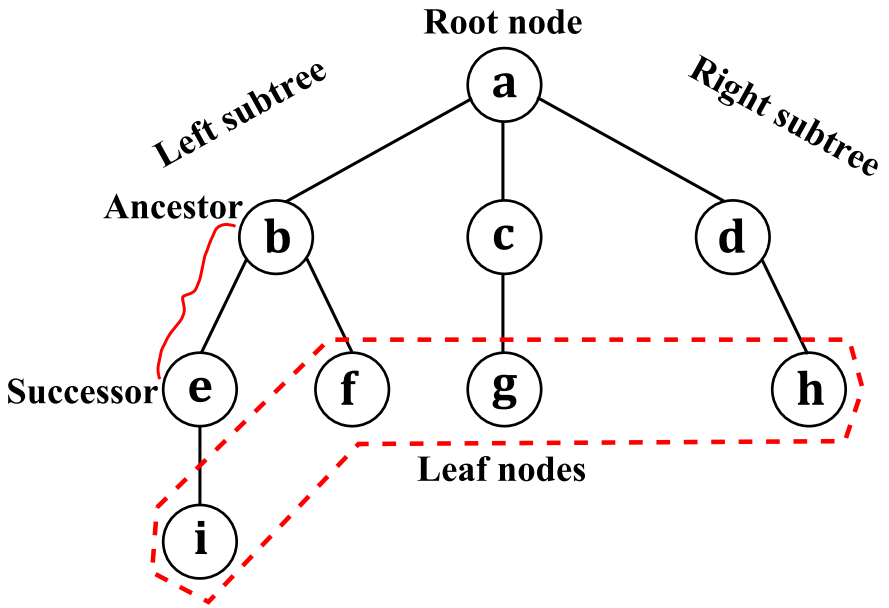
Associate Professor

MIU, CAIML, TIH
Indian Statistical Institute, Kolkata

November, 2023

Basics of a tree

A tree is recursively defined as a set of one or more nodes where a single node is designated as the root of the tree and all the remaining nodes can be partitioned into subtrees of the root.



Note: Tree is a non-linear data structure (a data item can be linked to more than two data items).

Types of trees

Trees can be of different types based on their use as data structures. Some of these are listed below.

- **Forests:** Disjoint union of trees.
- **Binary trees:** Trees having at most two subtrees per node.
- **Binary search trees:** Binary trees in which the nodes are arranged in an order (based on the data items that they keep).
- **Expression trees:** Trees that are used to store algebraic expressions.
- **Tournament trees:** Trees that are used to represent players using the leaf nodes and winners using the remaining nodes.

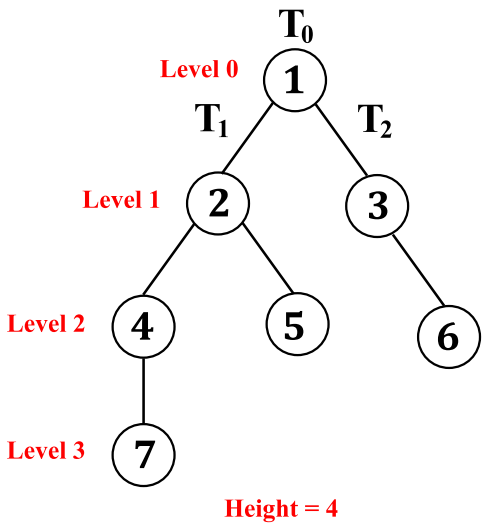
Note: Trees are data structures that are mainly used to store data items (labels or keys on the nodes) hierarchically.

Binary trees

Definition (Binary Tree)

A binary tree T over a domain D is either an empty set (empty binary tree) or a recursively-defined triplet $\langle T_0, T_1, T_2 \rangle$ such that

- T_0 is a node over D (root), and
- T_1 and T_2 are binary trees over D (left and right subtree, respectively).



Terminologies in binary trees

- **Left child:** Left successor node of a node.
- **Right child:** Right successor node of a node.
- **Parent:** Ancestor node of a set of children.
- **Sibling:** Nodes that are at the same level and share the same parent.
- **Degree of a node:** Number of children of a node.

Note: Every node other than the root node has a parent.

Pre-order traversal

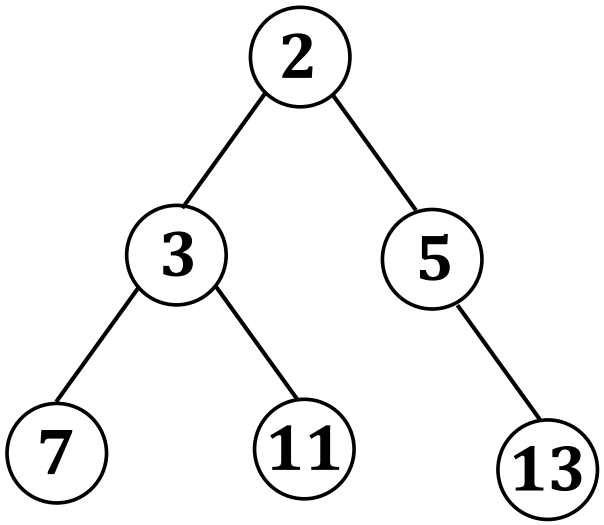
Perform the following operations recursively at each node:

1. Visit the root node.
2. Traverse the left subtree.
3. Traverse the right subtree.

Pre-order traversal

Perform the following operations recursively at each node:

1. Visit the root node.
2. Traverse the left subtree.
3. Traverse the right subtree.



Order of visited nodes: 2, 3, 7, 11, 5, 13

In-order traversal

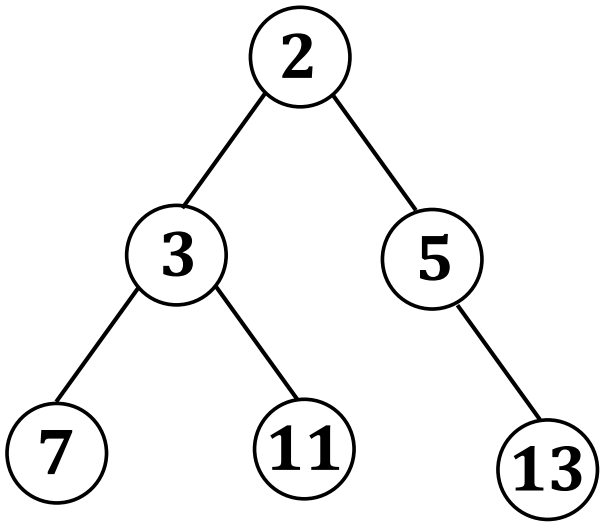
Perform the following operations recursively at each node:

1. Traverse the left subtree.
2. Visit the root node.
3. Traverse the right subtree.

In-order traversal

Perform the following operations recursively at each node:

1. Traverse the left subtree.
2. Visit the root node.
3. Traverse the right subtree.



Order of visited nodes: 7, 3, 11, 2, 5, 13

Post-order traversal

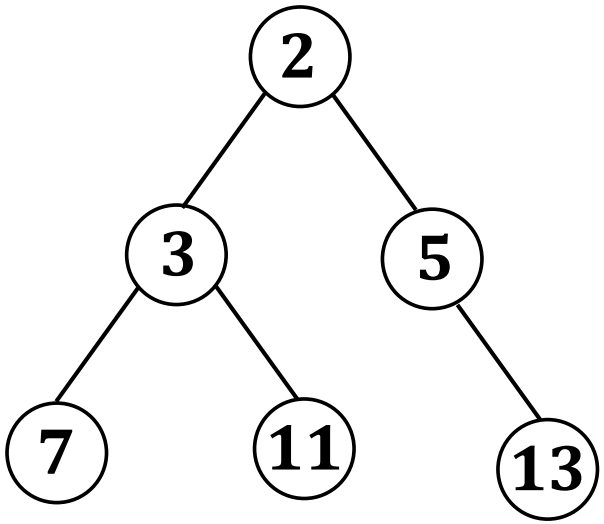
Perform the following operations recursively at each node:

1. Traverse the left subtree.
2. Traverse the right subtree.
3. Visit the root node.

Post-order traversal

Perform the following operations recursively at each node:

1. Traverse the left subtree.
2. Traverse the right subtree.
3. Visit the root node.



Order of visited nodes: 7, 11, 3, 13, 5, 2

Binary search trees

Definition

A binary search tree is principally a binary tree in which the following properties hold for all the nodes:

- key values in left subtree are less than the key value in the node
- key values in right subtree are greater than the key value in the node

Main operations

- **Insertion**
- **Search**
- **Deletion**

Auxiliary operations

- **Find successor**
- **Find predecessor**

Typedefs and helper functions (bst.h) – Traditional

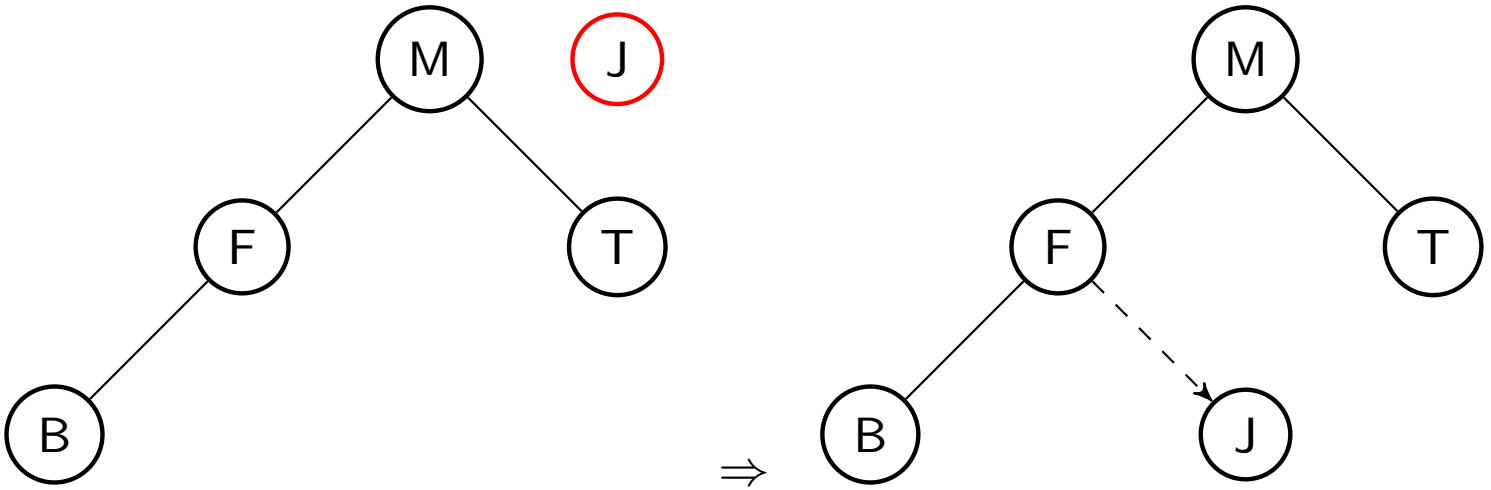
```
typedef int DATA; // Generic use: typedef void * DATA;

typedef struct node{
    DATA data;
    struct node *left, *right;
}BSTNODE;

extern int compare(BSTNODE *n, DATA d);
extern void inorder(BSTNODE *root);
extern void print_tree(BSTNODE *root, int indent);
extern void print_pstree(BSTNODE *root);
extern BSTNODE *search(BSTNODE *root, DATA d);
extern BSTNODE *detach_successor(BSTNODE *node);
```

BST insertion I

```
/**
* Arguments: pointer to root, data
* Returns: possibly modified pointer to root
* If "root" is NULL (empty tree), it will be changed to point to
* newly inserted node.
* This (possibly changed) value of root is returned.
* Caller is responsible for updating to the new, returned value (see
* recursive calls below, for example).
*/
BSTNODE *insert(BSTNODE *root, DATA d){
    /* Base case */
    if(root == NULL){
        root = Malloc(1, BSTNODE); /* should check return value */
        root->data = d;
        root->left = root->right = NULL;
        return root;
    }
```

BST insertion I

```
    /* Recurse */
    int cmp = compare(root, d);
    if(cmp < 0)
        root->left = insert(root->left, d);
    else
        if(cmp > 0)
            root->right = insert(root->right, d);
    return root;
}
```



BST insertion II

- `insert` and `delete` routines take pointer to the root, and change the root via this pointer as and when necessary.
- Since `root` is changed within these routines (if necessary), both routines return `void`.

BST insertion II

```
void insert(BSTNODE **rootptr, DATA d){
    BSTNODE *root = *rootptr;
    if(root == NULL){
        root = Malloc(1, BSTNODE); /* check return value */
        root->data = d;
        root->left = root->right = NULL;
        *rootptr = root;
    }
    int cmp = compare(root, d);
    if(cmp < 0)
        insert(&(root->left), d);
    else
        if(cmp > 0)
            insert(&(root->right), d);
    return;
}
```

BST searching

```
BSTNODE *search(BSTNODE *root, DATA d){
    if(root == NULL)
        return NULL;
    int cmp = compare(root, d);
    if(cmp < 0)
        return search(root->left, d);
    else
        if(cmp > 0)
            return search(root->right, d);
        else
            return root;
}
```

BST deletion

Let X be the node to be deleted.

Case I X is a leaf node.
Simply delete X.

Case II X has one child.
Replace the link to X with a link to its only child.

Case III X has 2 children.

1. Find S, the successor of X (node with smallest key in right subtree of X).
2. Replace the value in X by the value in S.
3. Delete node S from the tree (see Cases I and II above).

May also use X's predecessor, the largest key in left subtree of X in a similar fasion.

BST deletion

```
void delete(BSTNODE **nodeptr, DATA d){
    BSTNODE *node, *s;
    assert(nodeptr != NULL);
    node = *nodeptr;
    if(node == NULL)
        return;
    int cmp = compare(node, d);
    if(cmp < 0)
        delete(&(node->left), d);
    else
        if(cmp > 0)
            delete(&(node->right), d);
        else{
            if(node->left == NULL && node->right == NULL){
                /* Case I: leaf, just delete */
                *nodeptr = NULL;
                free(node);
                return;
            }
```

BST deletion (Contd.)

```
                /* Case II: only one child (no left child) */
                if(node->left == NULL){
                    *nodeptr = node->right;
                    free(node);
                    return;
                }
                /* Case II: only one child (no right child) */
                if(node->right == NULL){
                    *nodeptr = node->left;
                    free(node);
                    return;
                }
                /* Case III: both sub-trees present */
                s = detach_successor(node);
                node->data = s->data;
                free(s);
            }
    return;
}
```

BST deletion (Contd.)

Auxiliary function:

```
BSTNODE *detach_successor(BSTNODE *node){
    BSTNODE *nptr;
    assert(node != NULL);
    /* Go to right child, then as far left as possible */
    nptr = node->right;
    if(nptr == NULL) /* no successors */
        return NULL;
    if(nptr->left == NULL){
        node->right = nptr->right;
        return nptr;
    }
    while(nptr->left != NULL){
        node = nptr;
        nptr = nptr->left;
    }
    node->left = nptr->right;
    return nptr;
}
```

BST interface

```
BSTNODE *root = NULL; // root of the tree

/* INSERTION */
for(i = 0; i < num; i++){
    data = rand() % 100;
    insert(&root, data);
}

...

/* DELETION */
delete(&root, data);
```

Pre-order traversal

```
int preorder(BSTNODE *node){
    if(NULL == node) // Reached a leaf node
        return 0;
    // Visit the node
    PRINT(node->d);
    preorder(node->left);
    preorder(node->right);
}
```

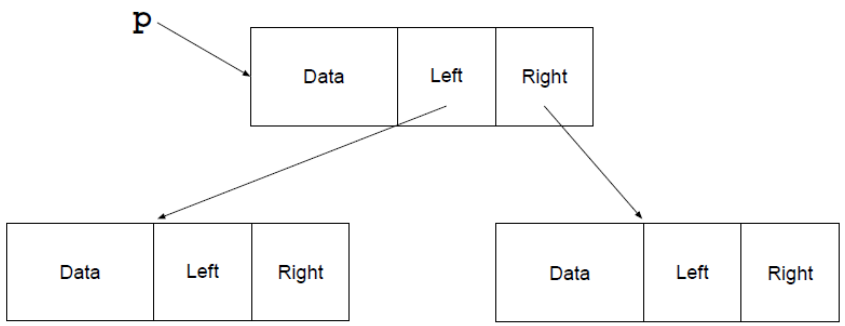
In-order traversal

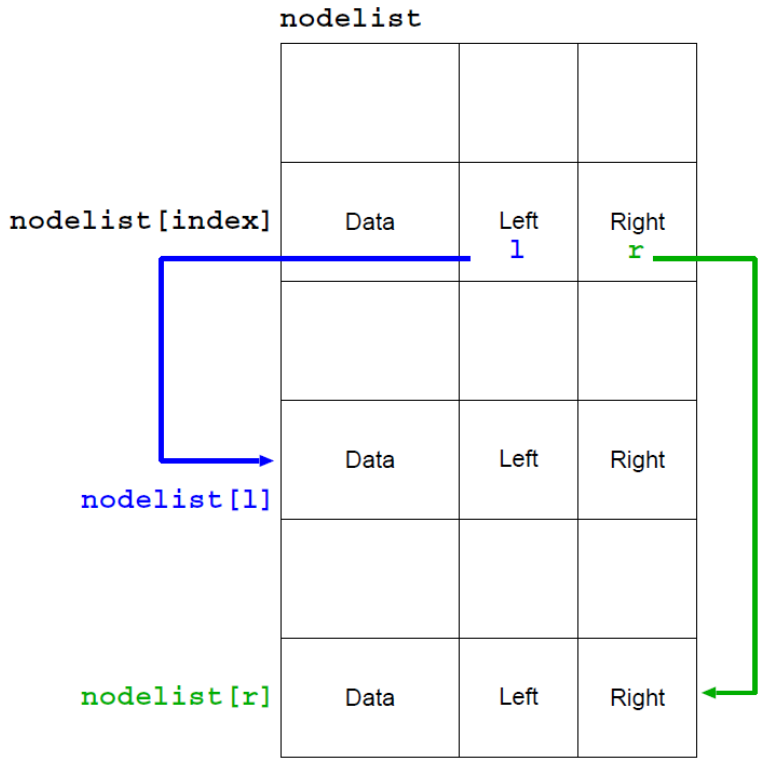
```
int inorder(BSTNODE *node){
    if(NULL == node) // Reached a leaf node
        return 0;
    // Visit the node
    inorder(node->left);
    PRINT(node->d);
    inorder(node->right);
}
```

Post-order traversal

```
int postorder(BSTNODE *node){
    if(NULL == node) // Reached a leaf node
        return 0;
    // Visit the node
    postorder(node->left);
    postorder(node->right);
    PRINT(node->d);
}
```

Traditional vs. Alternative implementations






```
BSTNODE *p;
p->data

root (type: BSTNODE *p)
```

```
int index;
tree->nodelist[index].data

tree->root (type: int)
```

Typedefs and helper functions (bst-alt.h) – Alternative

```
typedef struct node{
    DATA data;
    int left, right;
}BSTNODE;

typedef struct{
    unsigned int num_nodes, max_nodes;
    int root, free_list;
    BSTNODE *nodelist;
}TREE;

extern void inorder(TREE *, int root);
extern void print_pstree(TREE*, int root);
extern int search(TREE *, int root, DATA d);
extern int detach_successor(TREE *, int node);
```

Problems – Day 17

1. Complete the "alternative" implementation of binary search trees, a skeleton of which is provided in `bst-alt.c`. You may choose either option I or II discussed above.
2. Write a recursive function `treeToList(BSTNODE root)` that takes a BST and *only rearranges the internal pointers* to make a circular doubly linked list out of the tree nodes. The `previous` pointers should be stored in the `left` field and the `next` pointers should be stored in the `right` field. The list should be arranged so that the nodes are in increasing order. Return the head pointer to the new list.
   Target complexity: $O(n)$ time. Your program should *reuse* the tree nodes, *without* creating a separate node.
   `http://cslibrary.stanford.edu/109/TreeListRecursion.html`

Problems – Day 17

3. Write a program which, given a pair of BSTs, merge them into a single BST and print the data elements within it following an in-order traversal. For this program, assume that the data elements are integers.
4. Given a BST and two data elements within it, find the distance between them. Let the distance be calculated as the number of edges you need to traverse to reach to one elemenet from the other. You may assume that both the given data elements exist in BST.