

# Search Trees

Data and File Structures Laboratory

<http://www.isical.ac.in/~dfs/2018/index.html>

## Definition

Binary tree in which following property holds for all nodes:

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

## Main operations

- Insertion
- Search
- Deletion

## Auxiliary operations

- Find successor
- Find predecessor

# Typedefs and helper function (bst.h)

```
typedef int DATA;
```

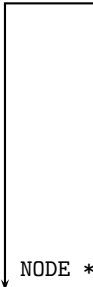
```
typedef struct node {  
    DATA data;  
    struct node *left, *right;  
} NODE;
```

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

# BST Insertion I (bst1.c)

```
/**
 * 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).
 */
NODE *insert(NODE *root, DATA d) {
    /* Base case */
    if (root == NULL) {
        root = Malloc(1, NODE); /* should check return value */
        root->data = d;
        root->left = root->right = NULL;
        return root;
    }
}
```

# BST Insertion I (bst1.c)



```
NODE *insert(NODE *root, DATA d) {  
    /* Base case */  
    if (root == NULL) {  
        root = Malloc(1, NODE); /* should check return value */  
        root->data = d;  
        root->left = root->right = NULL;  
        return root;  
    }  
}
```

# BST Insertion I (contd.)

```
/* 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;
}
```

- `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 (contd.)

```
void insert(NODE **rootptr, DATA d) {
    NODE *root = *rootptr;
    if (root == NULL) {
        root = Malloc(1, NODE); /* 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

```
NODE *search(NODE *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;
}
```

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 fashion.

## Helper function

```
NODE *detach_successor(NODE *node) {
    NODE *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 Deletion II I

```
void delete(NODE **nodeptr, DATA d) {
    NODE *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 II II

```
/* Case II: only one child */
if (node->left == NULL) {
    *nodeptr = node->right;
    free(node);
    return;
}
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 interface

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

```
/* INSERTION */
```

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

```
...
```

```
/* DELETION */
```

```
delete(&root, data);
```

# Viewing trees

```
$ ./bst1 10 2> bst1.tex  
$ latex tree.tex; dvips -o tree.ps tree; ps2pdf tree.ps  
$ <view tree.ps using installed document viewer>
```

# “Alternative” implementation

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

```
typedef struct {  
    unsigned int num_nodes, max_nodes;  
    int root, free_list;  
    NODE *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);
```

## Time for insertion / search

Data Structure	Worst case	Average case
Ordinary binary search trees	$O(N)$	$O(\lg N)$
Balanced binary search trees		$O(\lg N)$

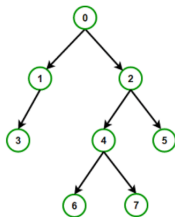
More on balanced BSTs next week + after the break.

# Problems I

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 program that computes the maximum width of a given binary tree. The maximum width of a tree is the maximum number of nodes at any level, where a level correspond to all nodes that are at the same distance from the root.

<http://www.techiedelight.com/find-maximum-width-given-binary-tree/>

3. You are given an array  $A$  which represents a binary tree in the following way: the parent of node  $i$  is given by  $A[i]$ . For the root node, the parent is denoted by  $-1$ . Construct the conventional representation of the binary tree from the above representation. For example, if  $A = \{-1, 0, 0, 1, 2, 2, 4, 4\}$ , then the tree is:



<http://www.techiedelight.com/build-binary-tree-given-parent-array/>

4. Write a recursive function `treeToList(NODE 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>

5. Write a function that takes a `NODE` as argument, and returns 1 if the argument is the root of a BST, 0 otherwise.

See <https://www.hackerrank.com/challenges/is-binary-search-tree> for more details.

Target complexity:  $O(n)$  time

Also, see SEDGEWICK AND WAYNE, problem 3.2.32.

# Problems IV

6. <https://www.hackerrank.com/challenges/tree-huffman-decoding>
7. Given a BST, and two numbers  $\min$ ,  $\max$  with  $\min \leq \max$ , trim the tree so that all its elements lie in  $[\min, \max]$ . Return the root of the new, trimmed tree. Note that the root of the tree may change.  
<http://leetcode.com/problems/trim-a-binary-search-tree/description/>
8. <http://www.spoj.com/problems/THREECOL/>
9. <https://www.hackerrank.com/challenges/balanced-forest>