Search Trees

Computing Lab

http://www.isical.ac.in/~dfslab

Binary search trees

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)

```c
typedef int DATA; // OR: typedef void * DATA;

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

/* Main operations: see bst.c for what each function returns */
extern NODE *insert(NODE *root, DATA d);
extern NODE *search(NODE *root, DATA d);
extern NODE *delete(NODE *root, DATA d);

/* Auxiliary operations (defined in bst-utils.c) */
extern int compare(NODE *n, DATA d);
extern NODE *detach_successor(NODE *node);

extern void print_tree(NODE *root, int indent);
extern void print_pstree(NODE *root); /* Ignore for now */
```

BST Insertion (bst1.c)

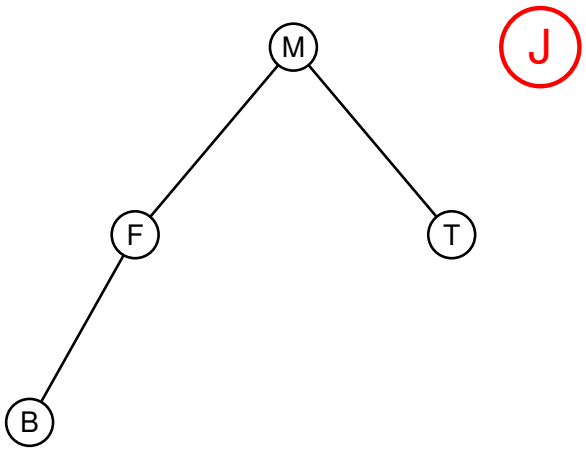
```
/**
 * Arguments: pointer to root, data
 * Returns: possibly modified pointer to root
 *
 * If "root" is NULL (empty tree), it will be changed to point to newly
 * inserted node. This (possibly changed) value of root is returned.
 *
 * Caller is responsible for updating to the new, returned value (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 (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;
```

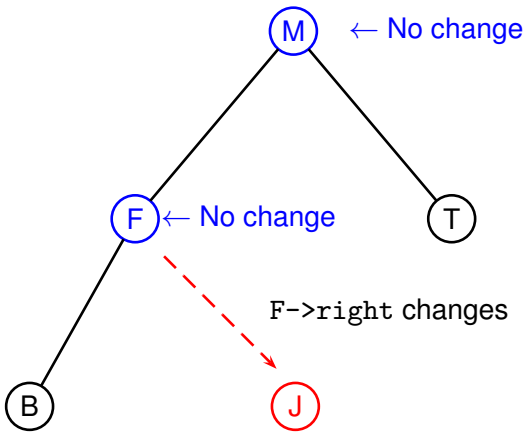
BST Insertion (contd.)

```
/* 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;
```



BST Insertion (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;
```



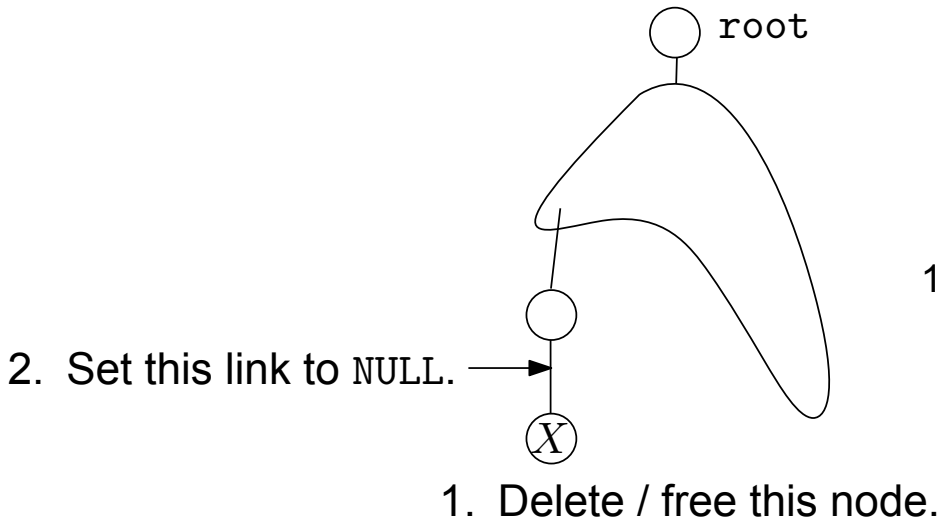
BST Searching

```
/**
 * Arguments: pointer to root, data
 * Returns: pointer to node containing search key if found, else NULL
 */
NODE *search(NODE *root, DATA d) {
    if (root == NULL) return NULL;
    int cmp = compare(root, d);
    if (cmp < 0) return search(root->left, d);
    if (cmp > 0) return search(root->right, d);
    return root;
}
```

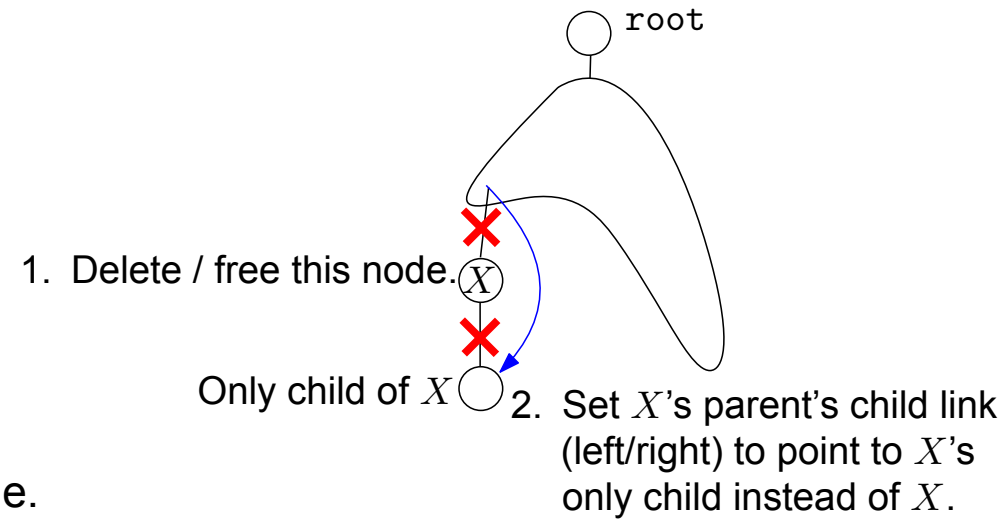
BST Deletion

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.



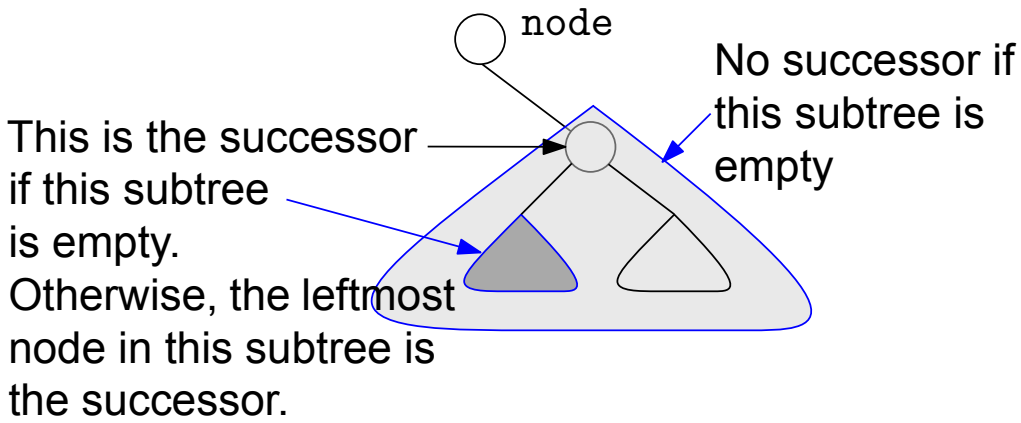
BST Deletion

(X is the node to be deleted.)

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).

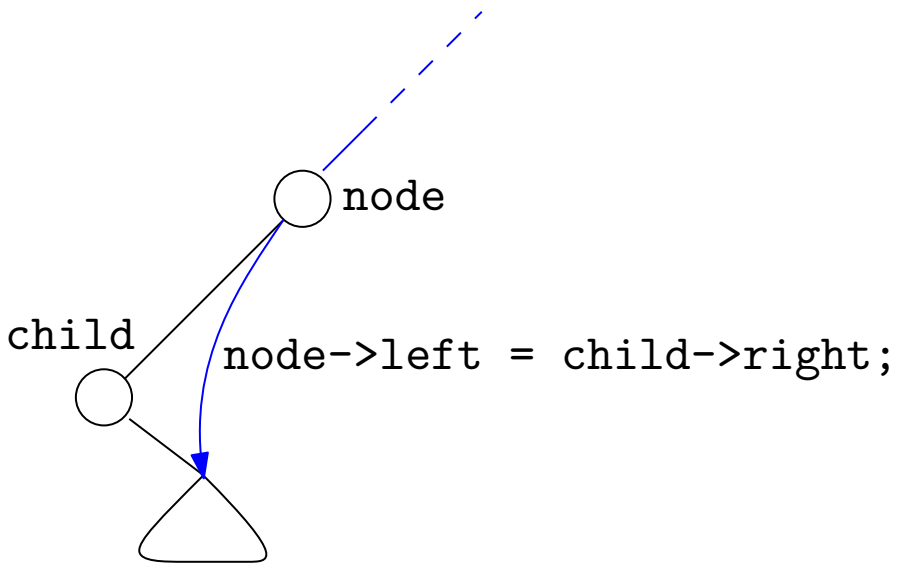
May also use X's predecessor, the largest key in left subtree of X in a similar fashion.



BST Deletion

Helper function

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



BST Deletion (bst.c) I

```
NODE *delete(NODE *root, DATA d) {
    NODE *s;

    if (root == NULL) return NULL;

    int cmp = compare(root, d);
    if (cmp < 0)
        root->left = delete(root->left, d);
    else if (cmp > 0)
        root->right = delete(root->right, d);
    else {
        if (root->left == NULL &&
            root->right == NULL) {
            /* Case I: leaf, just delete */
            free(root);
            return NULL;
        }
        /* Case II: only one child */
```

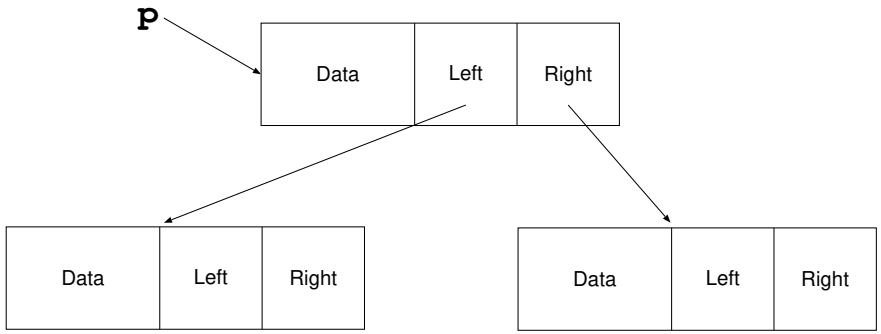
BST Deletion (bst.c) II

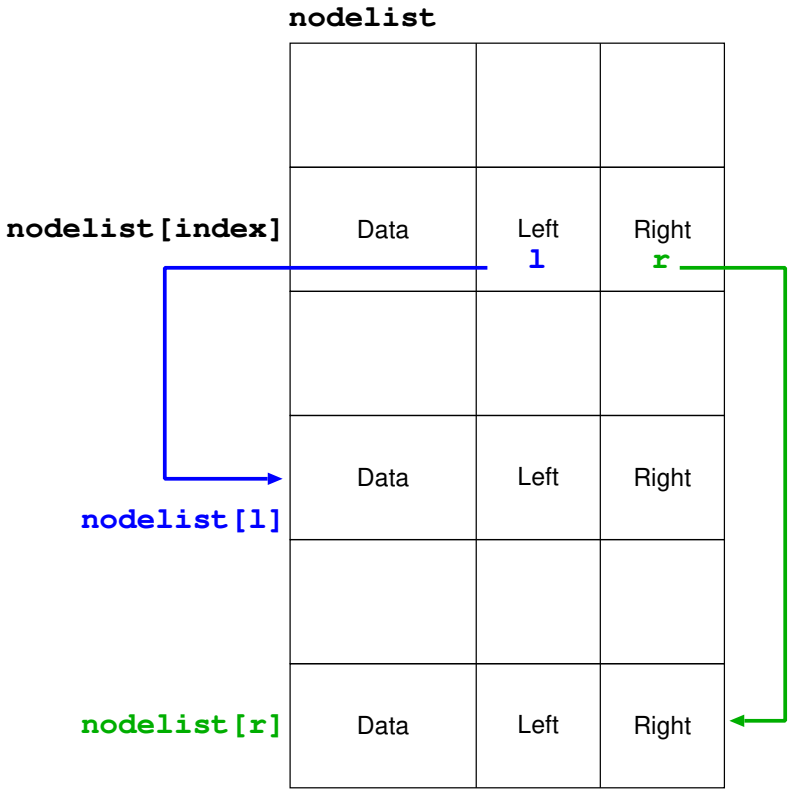
```
    if (root->left == NULL) {
        s = root->right;
        free(root);
        return s;
    }
    if (root->right == NULL) {
        s = root->left;
        free(root);
        return s;
    }
    /* Case III: both sub-trees present */
    s = detach_successor(root);
    root->data = s->data;
    free(s);
    return root;
}

return root;
```

BST interface (bst-testing.c)

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

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

...

/* DELETION */
root = delete(root, data[j]);
```

Recap: traditional vs. alternative implementations






```
NODE *p;
p->data

root   (type: NODE *p)
```

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

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

“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;

/* Main operations: see bst-alt.c for what each function returns */
extern int insert(TREE *, int root, DATA d);
extern int search(TREE *, int root, DATA d);
extern int delete(TREE *, int root, DATA d);

/* Auxiliary operations (defined in bst-alt-utils.c) */
extern int detach_successor(TREE *, int node);

extern void print_tree(TREE*, int root, int indent);
```

Balanced BSTs

Time for insertion / search

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

Problems I

1. Complete the **incomplete** "alternative" implementation provided on the course page.
2. Modify the BST operations so that they work for any **generic** version of binary search trees.

   As before, you will need to store the element-size and a comparator function inside `TREE`.
3. Implement *non-recursive* versions of pre-order, in-order and post-order traversals for the array-based representation of binary trees.
4. Write a program that, given a BST T and an integer n, finds the node of T whose value is nearest to n.
5. Write a program which, given a pair of BSTs, print the data elements common to the two trees. For this program, assume that the data elements are integers.

Problems II

6. 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.
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/`