BIRLA INSTITUTE OF TECHNOLOGY AND SCIENCE, PILANI

CS-F211: Data Structures and Algorithms

Lab 9: Binary Search Trees

Introduction

Welcome to week 9! This week we shall familiarise ourselves with an important data structure known as the *Binary Search Tree*, often abbreviated as *BST*. It is the first tree-based data structure that you will learn in this course. It is built on top of the underlying concept governing the binary search procedure, which you have learnt in your introductory computer programming course. There are several useful operations that can be performed on a binary search tree, and we shall learn them through examples and problems.

Topics to be covered in this labsheet

- Introduction to BSTs
- Traversals
- Operations on BSTs
    - Insertion
    - Querying (Search, Minimum/Maximum, Predecessor/Successor)
    - Deletion
- Time and Space Complexity of BST operations

<u>Note:</u> All functions implemented in this labsheet are provided to you in a file named "bst.c". There is no need to copy-paste code from this document to your code editor.

Introduction to Binary Search Trees

The Binary Search Tree data structure is a tree-like organisation or encapsulation of the binary search operation. It is a *binary tree* (meaning that each node can have at most two children) that supports an easy searching mechanism. A binary search tree obeys the following property: *"each internal node **y** stores an element **e** such that the elements stored in the left subtree of **y** are less than or equal to **e**, and the elements stored in the right subtree of **y** are greater than or equal to **e**"*. This property is fundamental in modelling its encapsulated binary search behaviour.

Evidently, a binary search tree can be stored as a linked data structure consisting of nodes and pointers. A binary search tree containing **integer keys** can be implemented as follows:

```c
typedef struct node {
    int value;
    struct node *left;
    struct node *right;
} Node;

typedef struct bst {
    Node *root;
} BST;
```

The following functions can be used to create (and return) new binary search trees and nodes:

```c
BST *new_bst()
{
    BST *bst = malloc(sizeof(BST));
    bst->root = NULL;
    return bst;
}

Node *new_node(int value)
{
    Node *node = malloc(sizeof(Node));
    node->value = value;
    node->left = NULL;
    node->right = NULL;
    return node;
}
```

Each node in our implementation contains a key and two pointers, one pointing to its left child and the other pointing to its right child. Our macro BST structure contains just a single pointer pointing to the root of the tree. This is analogous to the linked list structure that contains just the head pointer.

Figure 1 shows two example binary search trees.

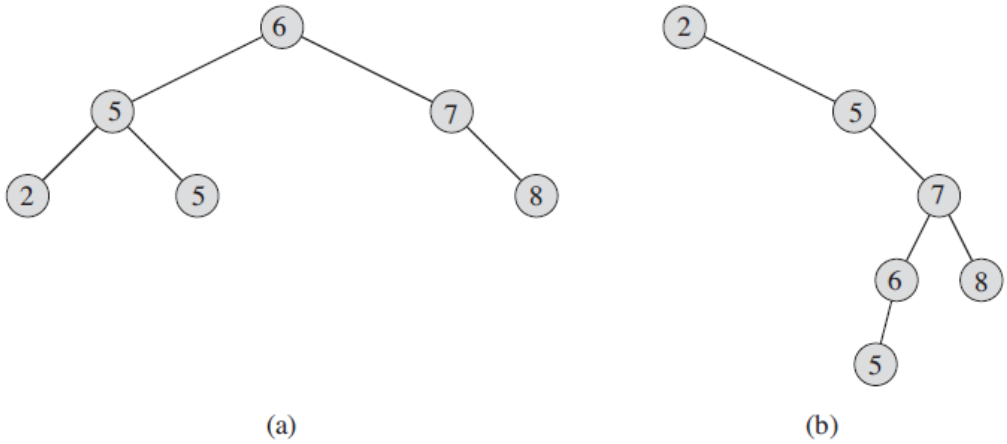

Figure 1: (a) is a BST with 6 nodes and height 2, (b) is a BST with the same 6 nodes but height 4

Observe that in Figure 1(a) the element at the root is 6, the elements 2, 5, and 5 in its left subtree are no larger than 6, and likewise the elements 7 and 8 in its right subtree are no smaller than 6. This property holds for every node in the tree. Similar observations can be made about the other tree as well.

Bear in mind that a node in BST might contain an entire struct in itself, and need not just be integer keys. However, no matter the struct that a BST's nodes are made up of, the binary search tree property will always have to be checked, which means that the struct must contain a key in it that is a number (integer, float, or double) and that will be used for maintaining that property. For example, if it is a BST containing "Student" nodes, the key might be their CGPA (which would be a float attribute/member within the Student structure). Every function that we will be discussing in this labsheet will be designed in accordance with an integer BST. But always remember that this need not be the case, and in fact, for most problems, this will not be the case. All the functions might have to be modified appropriately if the node structure is modified to fit the needs of a different problem.

Traversals

The binary-search-tree property enables us to print out all the keys in a binary search tree in sorted order by the simple *"inorder tree walk"* algorithm (which follows the sequence "go left, print data, go right" at each node). Recursion is the cleanest way to implement this logic, though it can be done iteratively as well. This recursive algorithm can be programmed as follows:

```
void traverse_in_order(Node *node)
{
    if (node == NULL)
    {
        return;
    }
    traverse_in_order(node->left);
    printf("%d ", node->value);
    traverse_in_order(node->right);
}
```

As clear from the above function, we first recursively call our traversal function on the current node's left child, then print the current node's key value, and then call the traversal function on this node's right child. The base case of this function is simply the case when the current node is NULL, and in that case, we return. Through the process of recursion, this simple piece of code conducts a clean in-order traversal of the binary search tree.

An in-order traversal of a binary search tree gives us the elements of the BST in **sorted order**.

As you have seen in the lectures, one of the main indicators of performance for binary search trees is their *height* (or *depth*), which is defined as the maximum depth of a leaf node from the root of the tree (note that it is measured in terms of number of edges). The height of a search tree limits the worst-case time complexity of insert and search operations (which we shall discuss in the next section).

In-order traversal is not the only kind of traversal. There are also the appropriately named *pre-order traversal* (which follows the sequence "print data, go left, go right") and *post-order traversal* (which follows the sequence "go left, go right, print data").

Task 1: Implement the pre-order and post-order traversal functions in the *bst.c* source file. Now, write alternate forms of all three of the traversal functions that print a string "null" for the

children in the trees that are NULL pointers, rather than leave them blank. [You can also try converting them to iterative functions.]

Home Exercise 1: Write a function that performs a **level-order traversal** on a BST. A level-order traversal is defined as the left-to-right breadth-first search on the BST. In other words, we traverse all nodes at the first level first (left to right), followed by all nodes at the second level (again, left to right), and so on till we traverse all the nodes at level d+1 (where d is the depth of the BST). For instance, the level-order traversal of the tree in Figure 1(a) will be "6 5 7 2 5 8" and that of the tree in Figure 1(b) will be "2 5 7 6 8 5".

Home Exercise 2: Write a function to perform the **reverse level-order traversal** on a BST. A reverse level-order traversal is a level-order traversal in which we start from the $(d+1)^{th}$ level left to right, followed by the d^{th} level (again, left to right), and so on upto the 1^{st} level. For instance, the reverse level-order traversal of the tree in Figure 1(a) will be "2 5 8 5 7 6" and that of the tree in Figure 1(b) will be "5 6 8 7 5 2".

Operations on Binary Search Trees

Insertion

The insertion operation on a binary search tree is very straightforward. We start scanning for the correct position of the node to be inserted downward from the root node. This means that we keep comparing whether our node to be inserted has a key greater than or less than the node we are checking it with. If it is greater, then we move to our node's right child. If less, we move to the left child. We keep repeating this process till the child we are moving to becomes NULL. At that point, we have found the position where our node is to be inserted. Simple pointer manipulations at that location allow us to insert the node into its place.

This can be implemented in the form of a function as follows:

```
void insert(BST *bst, Node* node)
{
    if (bst->root == NULL)
    {
        bst->root = node;
        return;
    }
    Node *current = bst->root;
    while (current != NULL)
    {
        if (node->value < current->value)
        {
            if (current->left == NULL)
            {
                current->left = node;
                return;
            }
            current = current->left;
        }
        else
        {
            if (current->right == NULL)
            {
                current->right = node;
                return;
            }
```

```
            current = current->right;
        }
    }
}
```

If the BST's root is NULL, then the node we are trying to insert into the BST is going to become the root. In every other case, we must perform the traversal downward and add the node to the tree when we have found its correct position where it will satisfy the BST property.

Constructing a binary search tree involves performing multiple insert operations into it. But note that the BST can end up looking very skewed or very balanced, depending on the order in which we perform our insertions, and this affects the "height" of our BST, which in turn has a bearing on the time complexity of all our BST operations. Finding clever ways for maintaining height balance in a binary search tree is a big research topic in the field of data structures, and we shall learn some such techniques in next week's lab. For our purposes, we shall assume that, in the worst case, our BST can indeed be heavily skewed and have an O(n) height, where n is the number of nodes.

Task 2: Write a function *constructBST()* that takes as input an array of integers and creates a new BST and then iteratively performs the insert operation on a BST with those integers. It returns a pointer to the finally constructed BST (ie., BST*). [Bear in mind that this function will also change if the node structure is changed, because in that case you would be requiring multiple fields and the input to the function would be an array of structs.]

Querying

Searching

The searching operation in a binary search tree is logically very similar to the insertion operation and also leverages the binary search tree property. Our aim here is to find whether an input key exists in a given binary search tree. For this, we would perform the same traversal from the insert operation and try to find out where this key "should be", and then check whether it is actually present there.

This logic can be implemented in the following manner:

```
int search(BST *bst, int key)
{
    Node *current = bst->root;
```

```
    while (current != NULL)
    {
        if (key == current->value)
        {
            return 1;
        }
        else if (key < current->value)
        {
            current = current->left;
        }
        else
        {
            current = current->right;
        }
    }
    return 0;
}
```

The search routine can easily be modified to return, instead of zero/one, a pointer to the node where the key has been found.

Minimum and Maximum

Finding the minimum and maximum elements in a BST is very straightforward. For finding the minimum, we can simply keep following the “left child” pointers till we encounter a NULL, and likewise we can keep following the “right child” pointers till NULL for finding the maximum element.

These functions can be implemented as follows:

```
int find_min(BST *bst)
{
    Node *current = bst->root;
    while (current->left != NULL)
    {
        current = current->left;
    }
    return current->value;
}
```

```
int find_max(BST *bst)
{
    Node *current = bst->root;
    while (current->right != NULL)
    {
        current = current->right;
    }
    return current->value;
}
```

Predecessor and Successor

The problem of finding the in-order predecessor (or equivalently its in-order successor) is an intricate one and has many applications (such as in the deletion routine that we shall discuss soon afterwards). The in-order successor of a node **x** in a BST is defined as the node that has the smallest key that is greater than that of **x**. In other words, it is that node which appears immediately after node **x** when one performs an in-order traversal of the BST. Equivalently, the in-order predecessor of a node **x** is defined as the node that has the largest key smaller than that of **x**, or the one that appears immediately before node **x** in the in-order traversal.

These functions can be implemented as follows:

```
Node *predecessor(Node *node)
{
    if (node->left == NULL)
    {
        return NULL;
    }
    Node *current = node->left;
    while (current->right != NULL)
    {
        current = current->right;
    }
    return current;
}

Node *successor(Node *node)
{
```

```
    if (node->right == NULL)
    {
        return NULL;
    }
    Node *current = node->right;
    while (current->left != NULL)
    {
        current = current->left;
    }
    return current;
}
```

Task 3: Write a recursive function that takes a binary search tree as its input and finds out whether it satisfies the binary search tree property. Now, convert this function to an iterative one. You can test your functions by constructing an array of BSTs (which would be an array of structs) using the arrays of numbers present in the *n_integers.txt* file that was provided to you in lab 7 (it is provided along with this labsheet as well) and running your function on each element of the array. Make use of your *constructBST()* function for the BST creation part. [Recall that the *n_integers.txt* file contains arrays containing *n* integers (the arrays are on separate lines), each array is also preceded by a number indicating the length of that array.]

Task 4: Write a function to determine the height of a binary search tree, starting from its root. Test this function by constructing BSTs using the arrays in the *n_integers.txt* file.

Deletion

The deletion operation in a binary search tree is fairly more complex than the previously discussed operations and is not as straightforward. We need to be very careful that we do not end up violating the BST property while performing the operation. Deleting a node **x** from a BST involves three cases:

1. If **x** is a leaf node, ie., it has no children, then we simply remove the node by modifying its parent to point to NULL instead.
2. If **x** has one child, then we can elevate that child to take up **x**'s position in the BST by modifying **x**'s parent to point to **x**'s child followed by removing **x**.
3. If **x** has two children, we must find either x's in-order successor or predecessor y and then replace x's contents with **y**'s. Once this is done, we can remove the successor (or

predecessor) node used by recursively calling this same function on that successor (or predecessor).

The same logic is implemented as follows:

```
void delete(BST *bst, Node *node)
{
    if (node == NULL)
        return;
    if (node->left == NULL && node->right == NULL)
    {
        // Node is a leaf
        Node* current = bst->root;
        while (current != NULL)
        {
            if (current->left == node)
            {
                current->left = NULL;
                break;
            }
            if (current->right == node)
            {
                current->right = NULL;
                break;
            }
            if (node->value < current->value)
            {
                current = current->left;
            }
            else
            {
                current = current->right;
            }
        }
        free(node);
        return;
    }

    if (node->left == NULL)
```

```
{
    // Node only has right child
    Node* current = bst->root;
    if (current == node)
    {
        bst->root = node->right;
        free(node);
        return;
    }
    while (current != NULL)
    {
        if (current->left == node)
        {
            current->left = node->right;
            break;
        }
        if (current->right == node)
        {
            current->right = node->right;
            break;
        }
        if (node->value < current->value)
        {
            current = current->left;
        }
        else
        {
            current = current->right;
        }
    }
    free(node);
    return;
}

if (node->right == NULL)
{
    // Node only has left child
    Node* current = bst->root;
    if (current == node)
```

```
        {
            bst->root = node->left;
            free(node);
            return;
        }
        while (current != NULL)
        {
            if (current->left == node)
            {
                current->left = node->left;
                break;
            }
            if (current->right == node)
            {
                current->right = node->left;
                break;
            }
            if (node->value < current->value)
            {
                current = current->left;
            }
            else
            {
                current = current->right;
            }
        }
        free(node);
        return;
    }

    // Node has both children
    Node *temp = successor(node);
    node->value = temp->value;
    delete(bst, temp);
    return;
}
```

Task 5: Write a function named *removeHalfNodes()* that removes all nodes that have only one child from an input BST. Do not invoke the *delete()* function implemented above to solve this. You can test this function by creating some BSTs using some of the arrays of numbers given to you in the *n_integers.txt* file, just like in Task 3.

Task 6: Create a new node structure for your BST, whereby a node contains `struct person` instead of `int value` as its attribute. The struct is defined as follows:

```
struct person
{
    int id;
    char *name;
    int age;
    int height;
    int weight;
};
```

Create a modified *constructBST()* function that creates a BST of these nodes by taking as input an array of structs. Use the "height" field as the key of the BST. You may need to modify certain other functions also to do this. Run this function with data from the *datX.csv* files (given).

Now, write a function *LCA()* that takes three inputs: a BST and the IDs of two nodes (ID is a field in the struct that is a part of the node). This function finds out the "least common ancestor" of the two corresponding nodes in the BST. The least common ancestor is defined between two nodes p and q as the lowest node (ie., the node at maximum depth) in the tree that has both p and q as its descendants (assume that a node can be a descendant of itself). Run the LCA() function on two random IDs in the size 10 BST and manually verify its correctness.

Home Exercise 3: The downside of the deletion approach that we have followed has to do with the case where the node has two children. In that case, with our approach (copying node's data into its parent), the node actually deleted might not be the node passed to the delete procedure. If other components of a program maintain pointers to nodes in the tree, they could mistakenly end up with "stale" pointers to nodes that have been deleted.

The approach that we have taken was followed in the first two editions of your reference book (Cormen, Leiserson, Rivest, Stein). But from their third edition onwards, they have updated it to a slightly more complicated variant which resolves the downside that we just discussed. Refer to pages 296-298 of the book's third edition and construct a new, more powerful variant of the *delete()* function on BSTs based on their approach.

Time and Space Complexity of BST operations

Clearly, the insert, search, and delete operations we just discussed will operate with a worst-case time complexity of O(h), which is the same as O(n), since the BST can be extremely skewed in the worst-case. However, the average-case time complexity of all three of these operations can be determined by assuming that a BST is rarely so skewed, especially if we insert the elements into the BST in an arbitrary or random sequence. Moreover, we don't always have to traverse upto the leaf node of the BST (except in the case of insertion). These are realistic assumptions, and given them, we can argue that the height of a BST is going to be O(log(n)) on average, and therefore, the insert, search, and delete operations work in O(log(n)) time in the average case.[1] It is also evident that any of the traversals on a BST that we have discussed will take O(n) time, as they would directly just traverse all the n nodes in the tree.

When it comes to space complexity, if we implement recursive versions of the search, insert, and delete operations, there are at most n stack frames in memory at a time. Equivalently, if we implement the operations iteratively using some explicit stack mechanism, there could still be at most n elements in the stack. This implies that the space complexity of these operations is O(n).

It is left as an exercise for you to find out the time and space complexities of the minimum/maximum and predecessor/successor querying operations.

Task 7: Write a function that takes a binary search tree as an input and "flattens" it, ie., it constructs and returns a linked list equivalent to the binary search tree. The constructed linked list must be in the same order as that of the pre-order traversal of the BST. This process is illustrated in Figure 2.

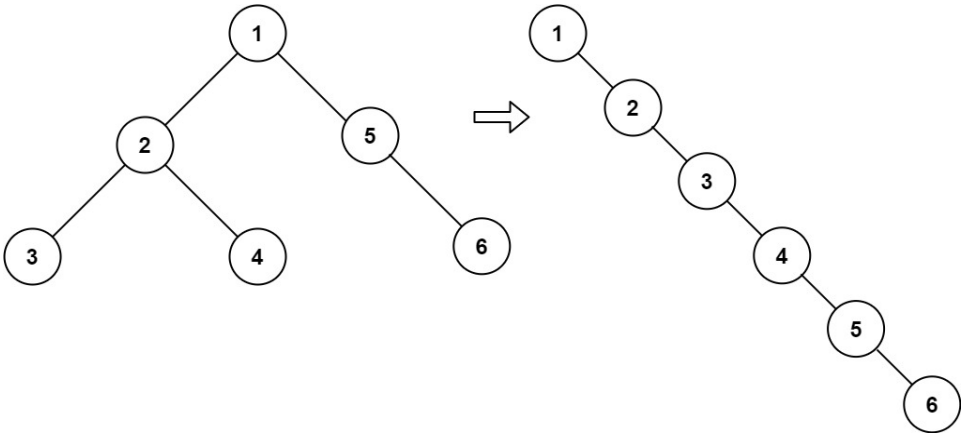

Figure 2: Illustrating the process of "flattening" a BST

[1] Interested students may refer to the proof of Theorem 12.4 on pages 300-303 of Cormen T.H., Leiserson, C.E., Rivest, R.L., and C. Stein. *Introduction to Algorithms*, MIT Press, 3rd Edition, 2009.

Now, try to perform the flattening "in-place" using O(1) auxiliary space only.

Home Exercise 4: Devise a function that takes a BST as an input and returns the **k^{th} smallest element** in the BST (where 'k' is also taken as an input).

Home Exercise 5: Recall that the worst-case search time in a BST is proportional to the depth of the tree. Thus, to minimize the worst-case search time, the height of the tree should be made as small as possible; by this metric, the ideal binary search tree is perfectly balanced.

However, in many applications of binary search trees, it is more important to minimise the total overall cost of multiple searches rather than the worst-case cost of a single search. If a particular node in a BST is searched more frequently than another, then it makes sense to put this node closer to the root of the tree compared to the other node (even if that means increasing the overall depth of the tree to something more than a perfectly balanced one). This way, we favour the node that is searched more often and disfavour the one that isn't. A perfectly balanced tree *need not* be the best choice if are aware that some items in the BST are significantly more frequently looked up than others. In fact, an imbalanced tree may be a better option in that case. In probabilistic terms, we are attempting to "optimise" (minimise) the "expected search cost" in our binary search tree.

Suppose that you are given as inputs a sorted array **key[0 … n-1]** of search keys and an array **freq[0 ... n-1]** of frequency counts, where **freq[i]** is the number of searches for **keys[i]**. Come up with a function *constructOBST()* that takes the above (along with an input array) as input and constructs a binary search tree that is ***optimal*** in the view of the above discussion.

[Note: Your approach should be to come up with a seemingly straightforward solution to this that apparently takes a lot of time, and then try to optimise your time complexity. A solution that runs in $O(n^3)$ time exists for this problem. In fact, certain optimisations to that solution can also bring down the time complexity to $O(n^2)$.]