Wesleyan University, Fall 2023, COMP 211

Lab 12: Binary Search Trees

---

1. Overview

The goal of this lab is to is to gain some practice working with binary search trees.

2. Create and organize your program files

For this lab, you should create a new directory lab12 in your labs directory to hold your code. Then download the following files from the lab website: `Makefile`, `driver.c`, `btree.c`, `btree.h`. During this lab you will modify `btree.c`.

3. Binary Search Tree

A binary search tree is a tree with the following properties:

- Every node has at most two children.
- Each node contains a unique key value.
- For each node, n, the key's in n's left subtree are smaller than n's key, and n's key is smaller than the keys in n's right subtree.

Recall from class that we can represent a binary tree using the following structs and create an empty binary tree using the empty function (this code has already been provided for you in `btree.c` and `btree.h`):

```
typedef struct bnode {
    int key;
    struct bnode *left;
    struct bnode *right;
} bnode;

typedef struct btree {
    bnode *root;
} btree;

btree* empty() {
    btree *t = malloc(sizeof(btree));
    t->root = NULL;
    return t;
}
```

Also recall from class that we can insert a new key (i.e., node) into the right location in a binary tree using the following function (this code has already been provided for you in `btree.c`):

```
void insert(btree *t, int key) {
    bnode *n = malloc(sizeof(bnode));
    n->key = key;
    n->left = NULL;
    n->right = NULL;

    // Tree is empty
    if (t->root == NULL) {
        t->root = n;
        return;
    }

    // Tree is not empty
    bnode *temp = t->root;
    while (temp != NULL) {
        bnode* prev = temp;
        if (key == temp->key) {
            return;
        } else if (key < temp->key) {
            temp = temp->left;
            if (temp == NULL) prev->left = n;
        } else {
            temp = temp->right;
            if (temp == NULL) prev->right = n;
        }
    }
}
```

Now, there are a number of other operations that are natural to perform on a binary search tree.

- `bool search(btree* t, int key)`: Perform an iterative search on the binary search tree `t`, return true if any node contains the key value of `key`, return false otherwise.

- `bool searchr(bnode* t, int key)`: Perform a recursive search on the binary search tree `t`, return true if any node contains the key value of `key`, return false otherwise.

- `void inorder(bnode* t)`: Perform a recursive inorder traversal of a binary search tree, printing out the key values at the appropriate points during the traversal. This function has already been written for you (also shown below), along with a wrapper function `print_inorder` which calls this function and prints some additional information.

```
void inorder(bnode* n) {
    if (n == NULL) return;
    inorder(n->left);
    printf("%d ", n->key);
    inorder(n->right);
}
```

- `void preorder(bnode* t)`: Perform a recursive preorder traversal of a binary search tree, printing out the key values at the appropriate points during the traversal. You will write this function, but a wrapper function `print_preorder` which calls this function has already been written for you.

- `void postorder(bnode* t)`: Perform a recursive postorder traversal of a binary search tree, printing out the key values at the appropriate points during the traversal. You will write this function, but a wrapper function `print_postorder` which calls this function has already been written for you.

The signatures and specifications of the functions you must implement are given in `btree.h`, which also specifies that there is a structure type `btree`. In `btree.c` you will implement all the functions specified in the header file (I have written stubs for those functions that you should replace).

4. Going further

We have not discussed how to delete a node from a binary search tree. There are different approaches that can be taken, however, deletion is generally more complicated than insertion, and simple, efficient deletion of a node from a binary search tree is an open problem.

However, one somewhat clunky approach is the following. If a key is not in the tree there is nothing to do. If a key is a leaf, deletion is straightforward: set the appropriate pointer in the parent to NULL then free the child node. If a key has 1 or 2 children, deletion becomes more complicated. In this case, one option is to replace the the key of the node to delete with the smallest key from the node's right subtree, then delete the node holding the smallest key from the right subtree.

Think about whether there are alternative ways to delete a node from a binary search tree, as well as how you would actually implement this deletion in code.