Trees in C

Sam Serbouti serbouti@kth.se

October 11, 2024

Introduction

In this assignment, I worked with binary trees in C. The goal is to be able to add and look for an element and the tree must always remain sorted. Additionally, I implemented an in-order, depth-first traversal using both the recursive approach and an explicit stack to simulate recursion manually.

In a tree, each element (called a node) has a value and two[1] pointers leading to lower nodes which we can compare to branches. If a node equals the null pointer, it means it is a dead-end (or a leaf).

1 Implementing the data structure

Basic procedures

```
tree *construct_tree(){
    tree *tr = (tree*)malloc(..
        ..sizeof(tree));
    tr->root = NULL;
    return tr;
}

void free_tree(tree *tr){
    if (tr !=NULL){
        free_node(tr->root);
        free(tr);
    }
}
```

```
node *construct_node(int val){
    node *nd= (node*)malloc(sizeof(node));
    nd->value = val;
    nd->left=NULL;
    nd->right = NULL;
    return nd;
}

void free_node(node *nd){
    if (nd != NULL){
        free_node(nd->left);
        free_node(nd->right);
        free(nd);
    }
}
```

Adding a node and searching for one recursively

Both operations work together under the assumption that the tree is sorted: we find smaller values on the left.

[1] for a binary tree

```
node* add_recursive(node *nd,int val){
    if (nd==NULL){
        return construct_node(val);
    }
    if (val<nd->value) {
        nd->left=add_recursive(..
                ..nd->left, val);
    } else if(val>nd->value){
        nd->right=add_recursive(..
            ..nd->right, val);
    }
    return nd;
}
```

```
bool lookkup_recursive(node *nd, int val){
    if (nd ==NULL){
        return false;
    }
    if (val == nd->value){
        return true;
    } else if (val < nd->value) {
        return lookkup_recursive(nd->left, val);
    } else {
        return lookkup_recursive(nd->right, val)
    }
}
```

2 Runtime benchmarks

For add and lookup operations in a randomly generated tree

In a tree that isn't falling down(meaning, that each node has about as much on its left side as it does on its right side), the total height of the tree is proportional to $log_2(n)$ since each node "doubles" at every level.

For both add and lookup, we don't stop going through the tree until with find a $NULL$ (at the end of the branch) which would take $log(n)$ times a constant.

For my benchmarks, I start by warming up with a tree with 99000 nodes. Then, for each number of nodes (`n`), I generate a tree with random integers (using `add(tr, rand()%100)`) and measure the time it takes to either add a node of value 100 to the tree, or lookup for `rand()%100`. I do so for `n` going from 100 to 10000000 on a log scale and at each value of `n` I make 10 measurements and extract the minimum, average and maximum of each.

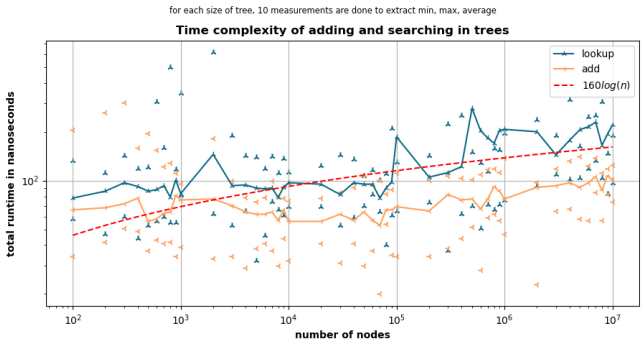

Figure 1: In a balanced tree, add and lookup $\hookrightarrow \mathcal{O}(lg(n))$ which is similar to an average case of a binary search algorithm

For add and lookup operations in an ordered tree

This time, when adding nodes to the tree, each has a value bigger than the previous one like so : `nxt += rand()%10 + 1;` Now, the tree is unbalanced tree and acts much like a linked list (its height is n and each add or search operation goes through n levels) so the operations$\hookrightarrow \mathcal{O}(n)$.

number of nodes (n)	balanced tree	unbalanced tree
100	86	340
500	466	1692
1000	599	856
5000	643	909
10000	781	1007
50000	1099	1290

Table 1: Runtime for building an entire tree of size n using recusrive add in nanoseconds

3 Non-recursive approach

```
void add_non_recursive(tree *tr, int val){
    node *nd = construct_node(val)
    if (tr->root == NULL){tr->root = nd;}
    node *cursor= tr->root;
    node *above =NULL;
    while (cursor!=NULL) {
        above=cursor;
        if (val<cursor->value) {cursor = cursor->left;}
        else if(val>cursor->value) {cursor = cursor->right;}
        else {return};
    }
    if(val<above->value) {
        above->left = nd;
    } else{
        above->right = nd;
    }
}
```

I can try to implement an add operation that doesn't use recursivity. In that case, I must keep track of my cursor (`node *cursor`) 's position and climb on the tree step by step: starting from the root, if the tree is empty I can just add the new node ; I climb up the tree and keep comparing the cursor's value to the one I want to add (if smaller I go left, if larger I go right, if equal we don't do anything). I need to keep track of the node above because I need to know where to insert once I find the end (NULL). The recursive approach is much easier to understand and implement. However,

a non-recursive method can avoid stack overflows if we use add_recursive in a very large tree.

4 In-order depth first printing

In an in-order depth first traversal of a tree, I start by going down the left branch until I reach its end, then the current branch (ie node) then the branch the most at the right. Here are procedures that use this chosen path to print the content of a tree:

```
static void print(struct node *nd) {
    if (nd!=NULL){
        print(nd->left);
        printf("%d ;", nd->value);
        print(nd->right);
    }
}
```

```
void print_tree(struct tree *tr) {
    if (tr->root != NULL) {
        print(tr->root);
        printf("\n");
    }
}
```

5 Stack

Stack data structure

So far, invisibly, the recursion use an in-built[2] stack structure (that has a `int top`, a `int size` and `node **array` and array of node pointers)to perform the recursion of the add and look operations. I will now try to build a stack explicitly and this will allow me to avoid the recursivity.

We first go down to the leftmost node, print it, then process the right branch. This approach avoids recursion, making the stack simulate the recursive call stack that would otherwise be used in recursive in-order traversal.

Tree traversal and print

```
start the cursor at root
while cursor not at the end AND stack is not empty
    while there is still a node on the branch:
        push(stk, cur) (push the current node in the stack)
        cur = cur->left (move the cursor to the left subbranch)
    cur = pop(stk)
    print cur->value
    cur = cur->right (if there is a right subbranch, repeat and push the left nodes)
free_stack(stk)
```

Functionality tests

I can now test a few features of the explicit stack-based depth-first tree structure:

[2] built in the programming language as a way to track function calls

```
print tree:   20 30 40 50 60 70 80
search 40 in tree (should return true): Found
search 100 in tree (should return false): Not Found
 add 40 to tree (shouodn't change)
print tree:   20 30 40 50 60 70 80
```

Advantages of using an explicit stack

- it can be dynamically allocated (more flexible for handling large inputs as memory is allocated from the heap, which has a much larger capacity, whereas we could get a stack overflow from a deep tree)
- it gives more control on what is stored and when. This can be useful in scenarios where I want to pause and resume execution, or revisit certain nodes (for example, if we try to solve a maze, we might need to revisit some nodes).

Conclusion

While building a tree structure, I compared recursive and non-recursive methods, while the recursive approach is often simpler to implement, the use of an explicit stack can offer more control in some scenarios.

In the end, I built a dynamic stack-based depth-first in-order traversal that manually recreated the recursive process using an explicit stack to store nodes. Though it was particularly tricky to build the stack and get the pointers to nodes to work.