Trees
in C

Algorithms and data structures ID1021

Johan Montelius

Spring 2026

Introduction

Linked lists might be useful but the true value of linked data structures comes when we have more complex structures; the next step up from a linked list is a tree structure. It's called a tree since the structure originates in a *root* that then divides into branches that are further divided into *branches*. A branch that does not divide further is terminated by a *leaf*. Although the structure is called a tree it is most often drawn with the root at the top and branches going down, don't get confused.

The trees that we will work on now are so called *binary trees* i.e. a branch always divides into two branches, if it does not end in a leaf. The operations that we will look at are: construction, adding and searching for an item. We will later look at more general tree structures but the principles are the same.

A binary tree

Let's construct a binary tree where each node in the tree has: a value, a left branch and a right branch. The values could be anything but to be able to talk about sorted trees we require them to be *comparable*. In this example we will simply use integers as values.

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

typedef struct tree {
  node *root;
} tree;
```

You will need procedures to construct and free trees; a tree that is freed should of course free all nodes before it can free the data structure of the tree itself.

```
tree *construct_tree {
  tree *tr = (tree*)malloc(...);

  tree->root = ...
  return tr;
}

void free_tree(tree *tr) {
  :
  :
}
```

Freeing the nodes could be done recursively.

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

void free_node(node *nd) {
  if (nd != NULL) {
    :
    :
  }
}
```

Now assume that that the tree is sorted so all nodes with values smaller than the root key are found in the left branch and the nodes with larger values in the right branch. The ordering is of course recursive so if we go down the left branch we will find smaller values to the left etc.

Now implement two procedures:

- `void add(tree *tr, int value)` : add a new node (leaf) to the tree that holds the `value`. Note that the tree should still be sorted. If the value exists, do nothing.

- `bool lookup(tree *tr, int value)` : return true or false depending on if the value is found.

When implementing `add()` one could chose to implement it recursively. The algorithm would look like follows, start in the root node:

- If the value of the node is equal to the value, do nothing.
- If the value of the node is greater than the value and
  - we have left branch - recursively add the key value to the left branch and return,
  - if not - create a new node and set it as the left branch and return.
- Same thing for right branch.

The lookup-procedure becomes very similar in its structure i.e. recursive traversal of the tree in order to find the value that we are looking for. Set up a benchmark and compare the execution time for a growing data set. Note that when you construct a binary tree you should not construct it using an ordered sequence of values - what would happen if you did? How does the lookup algorithm compares to the binary search algorithm that you used in one of the previous assignments?

As an experiment, implement the `add()` operation but now without using a recursive strategy i.e. keep track of where you are in the tree as you go down a branch. Which approach is simpler to understand?

Depth first traversal

Very often you want to go through all items that you have in a tree and the question then arise in what order you should traverse the tree. If the tree like in our example is ordered with smaller values to the left, one natural order would be to traverse the items starting with the leftmost and then work your way towards the rightmost. This strategy is an example of a *depth first* strategy i.e. you go down as deep as possible before considering the alternatives.

The order could be called *in-order* since we present all items in the left branch before presenting the item of the node itself. The item of the node is thus *in-between* the items of the left and the right branch. We could also present the items in a *pre-order* or *post-order*; the name describes where in the order we place the item of the node.

A simple example of this could be to add a print procedure that prints all values in in-order.

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

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

An explicit stack

When we implement the print method we make use of the implicit stack of the programming language. The programming stack was, as you probably realized, quite nice to have since it saved us from keeping track of what to do next. We could of course use an explicit stack and do the push and pop operations ourselves; but as you will see this is quite tricky.

Use your dynamic stack implementation from one of the first assignments and adapt it to be a stack of nodes (in C: pointers to nodes). We now define an `invariant` that should always be true; an invariant is a property of a data structure or the property of the state of the computation at a particular point in the code. The invariant will help us understand what needs to be done in each step and goes as follows:

> The left sub-tree of a node that is pop:ed from the stack has been printed, the value of the node itself has not, nor the values of the right sub-tree.

Before you start coding, take a white paper and make some drawings of what the stack and the tree might look like. Now think think about a scenario where you have pop:ed a reference to a node from the stack - what should you do? Since the left branch has been handled you should print the value of the node itself and then proceed down the right branch. Only when the right branch had been handled should you pop the next node and continue.

When you look at the right branch it could of course be that it is empty (a null reference). Then your done, but if you have a node there you should move down the left branch and push the nodes on the way down. When you find the left-most node you're in a position where the left branch had been handled i.e. it is as if you just pop:ed the node from the stack.

```
void print(tree *tr) {
    stack *stk = create_stack();
    node *cur = tr->root;

    // move to the leftmost node

    while(cur != NULL) {
      // print value of node

      if( cur->right != NULL) {
          // move to the leftmost node, push nodes as you go
        } else {
          // pop a node from the stack
        }
      }
    }
}
```

Implement the print procedure and test it to see that it works. You might wonder if there is any reason to use an explicit stack instead of the stack of the programming language but it turns out that it could come in handy.In the print example there is no point in using an explicit stack but we could have a scenario where we want to save a state that describes the sequence of elements after a specific element; more on this on a lecture.