Data Structures in C

Prof. Georg Feil

Binary Search Trees

Winter 2018

Acknowledgement

- These lecture slides are based on slides by Professor Simon Hood
- Additional sources are cited separately

Reading Assignment (required)

- Read Data Structures (recommended textbook)
  - Chapter 9 sections 9.6 – 9.10

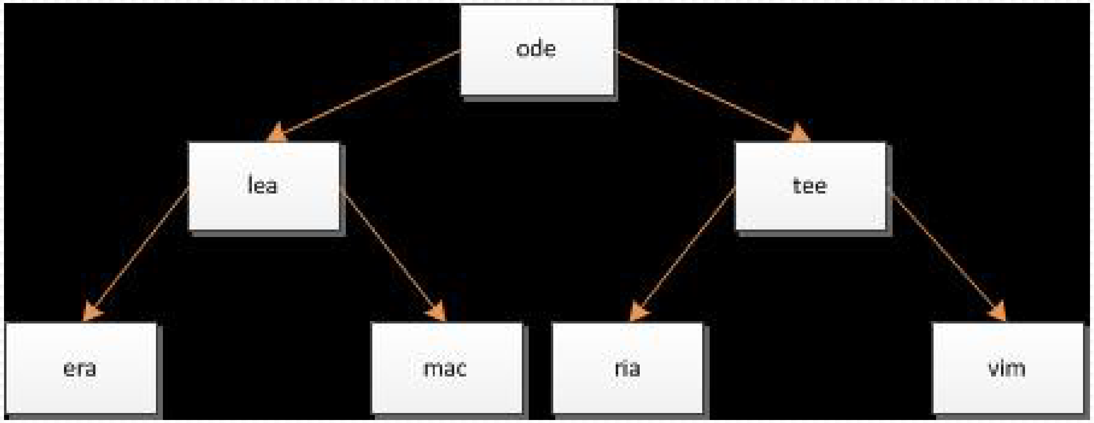
Note the textbook does a few things
we might consider poor style, for example
one-letter variable names and using int
for Boolean values.

Binary Search Tree definition

- A binary search tree is a tree with the additional property that, given any node
  - All nodes in the left subtree are smaller (or equal)
  - All nodes in the right subtree are larger (or equal)
- Such a tree makes searching a very simple process! It is roughly equivalent to a binary search in O(log(n)) time and is one of the most efficient known
  - But stay tuned for the O(1) hash search!
- If you can understand a Binary Search Tree, other types of trees should be easy to follow too

Binary Search Tree

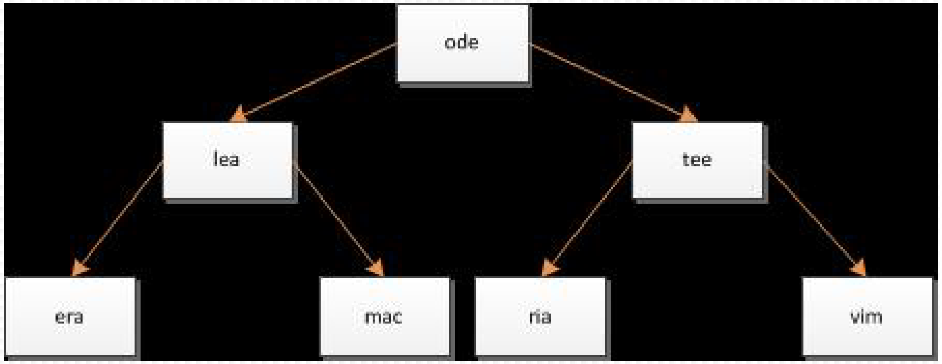
- Here's a binary search tree –we can search it in O(log(n)) time



- If we want to search for ria, we begin at the root (ode).
- ode is smaller than ria, so we move to the right
- tee is larger than ria so we move to the left (found it!)

Binary Search Tree

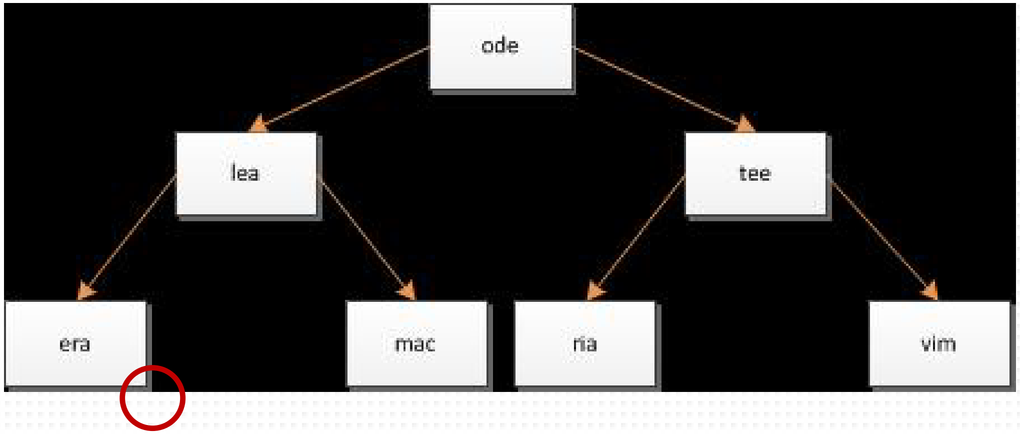
- What if we were searching for "fun"?



- Begin at the root (ode)
- ode is larger than fun so we go left
- lea is larger than fun so we go left
- era is smaller than fun, but there is no right to go to!
- We therefore conclude that fun is not in the list

Binary Search Tree – not found case

- Interestingly, by searching for a value, even though we did not actually find what we were looking for we found the exact place to put it



- Binary Search Trees make searching easy, and make inserting new values even easier! They combine the speed of binary search with the easy insertion of a linked list

Binary Search Tree – not found case

- Note that not all binary trees are equivalent –the order in which we insert our values determines the shape of the tree
  - In the best case our tree is nicely balanced
  - In the worst case we build a straight line and our search effectively becomes O(n)
- Fortunately, it can be proved that if the data comes in a random order, the average search for a given value is O(1.4log(n))

Binary Search Tree code

- Recall our binary tree structures

```
typedef struct{
    char word[MAX_WORD_SIZE];
} NodeData;

typedef struct treenode {
    NodeData data;  // struct inside a struct
    struct treenode* left;
    struct treenode* right;
} TreeNode;

typedef struct {
    TreeNode* root;
} BinaryTree;
```

Binary Search Tree code

- We'll need a function to create a new node containing some value in our binary search tree

```
TreeNode* newTreeNode(NodeData node) {
    TreeNode* p = (TreeNode*) malloc(sizeof(TreeNode));
    p->data = node;  // Copies the node data
    p->left = p->right = NULL;
    return p;
}
```

Building a Binary Search Tree

- Here's a function that searches for a specific string. If it finds it, it returns a pointer to the node containing the string. If not, it inserts a node containing the string and returns a pointer to the new node.

```
TreeNode* findOrInsert(BinaryTree* bt, char str[]) {
    NodeData node;
    strcpy(node.word, str); // Put 'str' in a node data structure
    if (bt->root == NULL)   // Is tree empty?
        return bt->root = newTreeNode(node);  // Returns value assigned

    TreeNode* curr = bt->root;  // Temporary pointer, current position
    int cmp;
    while ((cmp = strcmp(node.word, curr->data.word)) != 0) {
        if (cmp < 0) {  // Should we go left?
            if (curr->left == NULL)
                return curr->left = newTreeNode(node);
            curr = curr-> left;
        } else {  // Must be right
            if (curr->right == NULL)
                return curr->right = newTreeNode(node);
            curr = curr->right;
        }
    }
    printf("Item was found\n"); // If we reach here, item was found in the tree!
    return curr;
}
```

Building a Binary Search Tree

- We can use our recursive in-order traversal to print the sorted list

```
void inOrder(TreeNode* node) {
    if (node != NULL) {
        inOrder(node->left);
        printf("%s ", node->data.word);
        inOrder(node->right);
    }
}
```

Building a Binary Search Tree

- Finally, we might use the following main function to see it fly!

```
int main(int argc, char** argv) {
    char word[MAX_WORD_SIZE];
    BinaryTree bt;
    bt.root = NULL;
    printf("Please enter words. Enter X to finish.\n");
    while (true) {
        int cnt = scanf("%s", word);
        if (cnt != 1 || strcmp(word, "X") == 0)
            break;

        // Add word (string) to the BST
        findOrInsert(&bt, word);
    }
    inOrder(bt.root);   // Print out the tree
    return 0;
}
```

Best Binary Search Trees

- The best type of binary search tree (the one which will outperform all others for a given search) is a completely balanced tree
- There are existing algorithms which efficiently re-balance a tree (see this link) but they are not simple
- Instead, you'll create a balanced tree from a sorted list in your project
  - Why will your tree end up balanced?

Exercise 1

- Suppose the following words are added to a Binary Search Tree, in the order shown. Draw a picture of the tree after all the words have been added.

oval rectangle ellipse square octagon triangle circle

Exercise 2

- Put together all the code pieces to build a Binary Search Tree shown on the preceding slides. Get it running!
- Try entering letters of the alphabet for the node names
- Show that the tree is being built correctly
  - Hint: Draw the tree as you expect it should be built, then do an in-order traversal to see if it matches program output
- What's a good order to enter the letters?
  - What happens if you enter a, b, c, d, e, f?