We’ll be doing two major things in recitation today:

- **Binary Search Trees** - which you have seen in lecture
- **A Practice Quiz on heaps and binary search trees** - which will help you prepare for the quiz that will be released later today.

**Binary Search Trees (BST’s)**

- This data structure helps you to find an element faster, though the worst case scenario may be $O(n)$
- They have an ordering invariant which states that every element in the left subtree must be less than the root and every element in the right subtree must be greater than the root.
- Note that the invariant mentions *strict* inequality. This means that you *cannot have duplicate elements* in a binary search tree.
- BSTs don’t have a shape invariant.
- Due to the absence of a shape invariant, the worst case is a linked list of elements. Try inserting (in the given order) 1, 2, 3 & 4. Also try 4, 3, 2 & 1.

- So lookup from or insertion into a BST is worst case $O(n)$ and is $O(\log n)$ only for a balanced tree. AVL trees, which you’ll learn about next week, are balanced and have $O(\log n)$ lookup and insert.
- We need **two data structures** to be able to represent a BST
  1. A **tree** which is defined as
     ```c
     struct tree_node {
         elem data;
         struct tree_node* left;
         struct tree_node* right;
     };
     typedef struct tree_node tree;
     ```
This is a **recursive** data structure.

2. A **BST** which is defined as

   ```
   struct bst_header {
       tree* root;
   };
   ```

   This is a **non-recursive** data structure. It has a pointer to the root of the BST. The root is a **tree**, so it can be recursed over.

**Why does it matter that the tree data structure is recursive?**

As we’ll see, the easiest way to perform operations in any kind of tree is to do them in a recursive manner. Mainly this involves thinking of each node in a tree as a root for the sub-tree consisting of its children. So, given a particular operation, we perform the it on smaller trees recursively and combine them in order to get the resulting operation on bigger trees.

Now let’s discuss some of the code from lecture

```c
/* is_ordered(T, lower, upper) checks if all elements in T
   * are strictly in the interval (elem_key(lower),elem_key(kupper)).
   * lower = NULL represents -infinity; upper = NULL represents +infinity
   */
bool is_ordered(tree* T, elem lower, elem upper) {
    if (T == NULL) return true;
    if (T->data == NULL) return false;
    key k = elem_key(T->data);
    if (!(lower == NULL || key_compare(elem_key(lower),k) < 0))
        return false;
    if (!(upper == NULL || key_compare(k,elem_key(upper)) < 0))
        return false;
    return is_ordered(T->left, lower, T->data)
        && is_ordered(T->right, T->data, upper);
}

elem tree_lookup(tree* T, key k)
//@requires is_ordtree(T);
//@ensures result == NULL || key_compare(elem_key(result), k) == 0;
{
    if (T == NULL) return NULL;
    int r = key_compare(k, elem_key(T->data));
    if (r == 0)
        return T->data;
    else if (r < 0)
        return tree_lookup(T->left, k);
    else
        return tree_lookup(T->right, k);
}
```
else // @assert r > 0;
    return tree_lookup(T->right, k);
}

elem bst_lookup(bst B, key k)
//@requires is_bst(B);
//@ensures \result == NULL || key_compare(elem_key(\result), k) == 0;
{
    return tree_lookup(B->root, k);
}

/* tree_insert(T, e) returns the modified tree
 * this avoids some complications in case T = NULL
 */
tree* tree_insert(tree* T, elem e)
//@requires is_ordtree(T);
//@requires e != NULL;
//@ensures is_ordtree(\result);
{
    if (T == NULL) {
        /* create new node and return it */
        T = alloc(struct tree_node);
        T->data = e;
        T->left = NULL; T->right = NULL;
        return T;
    }
    int r = key_compare(elem_key(e), elem_key(T->data));
    if (r == 0)
        T->data = e; /* modify in place */
    else if (r < 0)
        T->left = tree_insert(T->left, e);
    else // @assert r > 0;
        T->right = tree_insert(T->right, e);
    return T;
}

void bst_insert(bst B, elem e)
//@requires is_bst(B);
//@requires e != NULL;
//@ensures is_bst(B);
{
    B->root = tree_insert(B->root, e);
    return;
}
Practice Quiz

1 : Binary Search Trees

Draw a BST with the following elements inserted in the given orders

5 3 6 2 9 4 7

4 9 6 5 2 3 7

Can you make an observation regarding the position of 9 (the largest element) in the tree?
2 : Heaps

Draw a heap with the following elements inserted in the given orders (it would be a good idea to write down steps)

\[ 5 \ 3 \ 6 \ 2 \ 9 \ 4 \ 7 \]

\[ 4 \ 9 \ 6 \ 5 \ 2 \ 3 \ 7 \]

Can you make an observation regarding the position of 9 (the largest element) in the heap?