Topics for this week

- Binary Search Trees
  - a useful datastructure
  - how to build them from a list
  - how to search for an element
  - how to list/traverse all elements (in sorted order)
  - time complexity of search / traversal
Problem: Limits of binary search

What do we know about binary search?

- What is it used for?
  - Input: a sorted list and a number to search for
  - Output: YES/NO depending on whether the number is in the list

- How fast does it run (time complexity)?
  - $O(\log n)$ where $n$ is the size of the list
    - VERY FAST ← Good

Problem: what if elements are being added / deleted?

Then we need to resort = expensive

In such cases we might implement binary search trees
Problem: Binary Search Trees (BSTs)

- (a) solution to the limits of binary search

Definition: what do they look like

- list: 1 3 7 8 9 10 17
- tree:
  - root (the top node)
  - each node stores a single value
  - binary - means every node has ≤ 2 child nodes (i.e. branches downward)
  - search - means that every node has smaller values on the left and larger values on the right (in the left subtree and in the right subtree)

Examples of unbalanced trees:
Problem: Building BSTs

Given a list of elements, build a binary search tree.

Representation:

```
8
/   \
3    5
/     |
1    7
```

By a list (for now): the first elem represents the root then a list of elements smaller than the root follows then one list of elements larger in the same format

```
[(8,1,4), (3,2,3), (1,1,1), (7,-1,-1), (15,5,6), (9,-1,-1), (17,0,1)]
```

Pseudo code:

In addition to the value, we will store the index of the left child and the index of the right child

See lecture notes
Problem: Searching BSTs

Given a BST and a value x, does it appear in the BST?

Pseudo code:

```python
def BST_search ( tree, inode, x ):
    if inode < 0: return FALSE
    if value(tree, inode) > x:
        return BST_search (tree, leftchild(tree, inode), x)
    if value(tree, inode) < x:
        return BST_search (tree, rightchild(tree, inode), x)
    return TRUE
```

Testing:

See the implementation/lecture notes.

Time complexity:

If the tree is balanced: \( O(\log n) \)

where \( n \) is the # values stored in the tree

But in the worst case (see backtree in the implementation): \( O(n) \)
Problem: Traversing BSTs

Given a BST, list all the values that it stores (in increasing order).

Pseudo code:

```
def traversal(tree, node):
    if node < 0:
        return
    traversal(tree, leftchild(tree, node))
    print value at position node
    traversal(tree, rightchild(tree, node))
```

Testing:

see the lecture notes

Time complexity: \(O(n)\) since \(n\) is the number of values to print and the pseudo code does not perform other operations (except for recursive calls; there are about \(n\) of those).