CS 1501
bill-computer.science/1501

Trees
The Searching Problem

Given a collection of keys $C$, determine whether or not $C$ contains a specific key $k$
Can use solutions to the searching problem to create a symbol table or map

- Abstract structures that link keys to values
  - Key is used to search the data structure for a value
  - Described as a class in the text, but probably more accurate to think of the concept of a symbol table in general as an interface
  - Key functions:
    - put()
    - contains()
Review: How to solve the searching problem?

- Store collection in an array
  - Unsorted
  - Sorted
- Linked list
  - Unsorted
  - Sorted
- Store the collection in a hash table

- Differences?
- Runtimes?
What are our best performers?

- Best worst-case search time: binary search
- Dynamically allocated memory as-needed: linked list

- Can we achieve both of these?
- Can we come up with a way to implement binary search on a linked list?
def binary_search(collection, target):
    lo = 0
    hi = len(collection) - 1
    while hi >= lo:
        # Note this is a comment, and // is floor division
        mid = (lo + hi) // 2
        if target < collection[mid]:
            hi = mid - 1
        elif target > collection[mid]:
            lo = mid + 1
        else:
            return mid
    return -1
def binary_search(collection, target):
    """
    Efficient search through a sorted collection

    Parameters:
    collection: sorted, indexable collection
    target: key to search for

    Returns:
    int: index where target appears or -1 if not found
    """
    ...

Comments and docstrings
public static int indexOf(int[] a, int key) {
    ...
}

- An example of JavaDoc for Java code
- Also, look to adopt a programming style guide
Consider the following collection of numbers

- 5, 10, 16, 2, 8, 37, 45
- As an unsorted linked list:

```
Head → 5 → 10 → 16 → 2 → 8 → 37 → 45 →...
```

- As a sorted array:

```
Array → 2 → 5 → 8 → 10 → 16 → 37 → 45 →...
```
Binary search
What will our new data structure look like?

- To store 2, 5, 8, 10, 16, 37, 45:
What will our new data structure look like?

- To store 2, 5, 8, 10, 16, 37, 45:
What we’ve built is a tree

- Specifically, it’s a binary search tree
  - Binary because each node has a max of 2 children
    - The tree has a branching factor of 2
  - And it’s a tree used to solve the searching problem
    - Specific BST rules:
      - Each node’s left child leads to keys less than its key
      - Each node’s right child leads to keys greater than its key
class Node:
    def __init__(self, key):
        self.key = key
        self.left = None
        self.right = None

a_node = Node(5)
a_node.left = Node(1)
a_node.right = Node(10)
We should have a BST class in addition to the Node class:

```python
class BST:
    def __init__(self):
        self.root = None
```

We’ll add `put()` and `contains()` methods for this class.
Implementing `contains()`

```python
def contains(self, key):
    return self.contains_rec(self.root, key)

def contains_rec(self, cur, key):
    if cur is None:
        return False
    if key < cur.key:
        return self.contains_rec(cur.left, key)
    elif key > cur.key:
        return self.contains_rec(cur.right, key)
    else:
        return True
```
Implementing put()

def put(self, key):
    self.root = self.put_rec(self.root, key)

def put_rec(self, cur, key):
    if cur is None:
        return Node(key)
    if key < cur.key:
        cur.left = self.put_rec(cur.left, key)
    elif key > cur.key:
        cur.right = self.put_rec(cur.right, key)
    return cur
We did it!

- The following code

```python
t = BST()
t.put(10)
t.put(5)
t.put(37)
t.put(2)
t.put(8)
t.put(16)
t.put(45)
```

- Will create this tree
Is all memory is dynamically allocated as-needed like a Linked-List?

Is the worst case search runtime $O(\lg n)$?
Consider the following:

t = BST()
t.put(1)
t.put(2)
t.put(3)
t.put(4)
t.put(5)

● How many nodes will be accessed in running t.contains(5)?
● What are the insert and search runtimes?
It is still a tree, but it is not *balanced* like the previous tree.

Which of these should be considered a balanced tree?
The height of a tree is the number of levels in the tree.
- In this class, we will say the root of the tree appears in level 1.
- An empty tree has a height of 0.

When all leaves appear on the same level and all non-leaf nodes of the tree have the maximum number of children, the tree is considered *full*.
- The first BST we built was a full tree.

When all levels of the tree except for the bottom have the maximum number of nodes and the last level is filled in from left to right, the tree is considered *complete*.
Keep in mind

- A binary search tree is a type of binary tree
  - With the extra rule applied that each node’s left subtree contains keys less than that node’s key, and the right subtree contains keys greater than that node’s key

- A binary tree is a type of tree
  - With a branching factor of two. There are other binary trees that are not binary search trees, and trees that are not binary trees (we’ll see examples of both this term)

- A tree is a type of...
  - A definition that we will get to much later in the term...
A completely balanced tree is one where all subtrees of each node are the same height.
Height balanced trees

- (Which is what I will mean when saying simply "balanced")
- A *height balanced binary tree* is one where the left and right subtrees of all nodes differ in height by no more than 1

![Diagram of height balanced trees](image-url)
BST runtimes

- Search and insert operations will take $O(h)$ time
  - Where $h$ is the height of the tree
  - But we need to describe runtimes in terms of input size!

■ How can we describe the height of a BST in terms of the number of keys it contains?

- Worst case: our linked list type tree
  - $O(n)$

- Best case: balanced
  - What is the height of a balanced BST in terms of the number of keys it contains?
Full BST height (completely balanced)

- Similar to counting number of bits required to store an int
  - 1 height stores 1 key
  - 2 height stores 3 keys
  - 3 height stores 7 keys
  - 4 height stores 15 keys...
- A height h tree stores $2^h - 1$ keys
  - $n = 2^h - 1$
  - $n + 1 = 2^h$
  - $h = \log(n + 1)$
- To store n keys, you need $O(\log n)$ height
Complete BST height (balanced)

- Everything but the last level is full
- So:
  - 2 height stores between 2 and 3 keys
  - 3 height stores between 4 and 7 keys
  - 4 height stores between 8 and 15 keys...
- A height h tree stores between \(2^{h-1}\) and \(2^h - 1\) keys
  - \(n = 2^{h-1}\)
  - \(\lg(n) = h - 1\)
  - \(h = \lg(n) + 1\)
- To store n keys, you once again need \(O(\lg n)\) height
Can be proven to be $O(lg \, n)$

- Start by proving that the minimum number of nodes in a height balanced binary tree (min_nodes(h)) is greater than $2^{(h/2 - 1)}$
  - Proof by induction
  - Can establish base cases for trees of heights 2 and 3
  - Note that the subtrees of a root of a height balanced binary tree of height h must have heights h-1 and h-2
    - $min\_nodes(h) = 1 + min\_nodes(h-1) + min\_nodes(h-2)$
- From there:
  - $min\_nodes(h) > 2^{(h/2 - 1)}$
  - $lg(min\_nodes(h)) > h/2 - 1$
  - $h < 2 \times lg(min\_nodes(h)) + 2 \leq 2 \times lg(n) + 2$
  - Hence, h is $O(lg \, n)$
But the BST rules don’t enforce that trees will be balanced, so we’d have to be lucky with the insertion order...

What about the average case?

- Propositions C and D in Section 3.2 of the text show it to be $O(lg n)$

  Similar approach to the average case analysis of Quicksort

- Proposition K from Section 2.3
BST performance:

- **Space requirements:**
  - $O(n)$, dynamically allocated

- **Search:**
  - Worst case: $O(n)$
  - Average case: $O(lg\ n)$

- **Insert:**
  - Worst case: $O(n)$
  - Average case: $O(lg\ n)$

- **What about removal?**
Tree traversal

- Often, in addition to put/contains/delete, you will want to visit each node in a tree and perform some action at each node.
  - This is known as a *traversal* of the tree.
- One thing you need to consider when performing a traversal is the order of performing the action vs visiting the children of that node.
In-order traversal
Pre-order traversal

- F
  - B
    - A
    - D
      - C
      - E
  - G
    - I
  - H
Post-order traversal