SFU

Binary Search Trees

CMPT 125
Mo Chen
SFU Computing Science
18/3/2020

Lecture 27

Today:

- Building a tree using a stack
- Binary Search Trees

Trees and Recursion (Review)

Trees can be defined recursively.

- often, in terms of their subtrees

Recursive definitions benefit you in two ways:

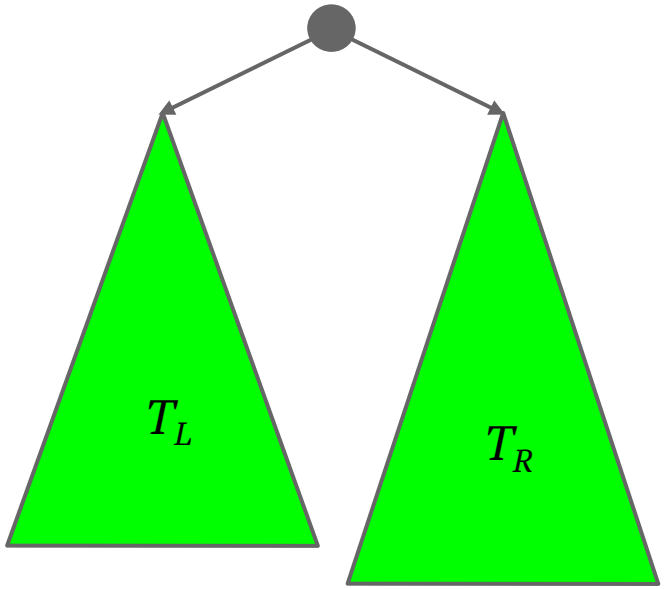
- allow you to write recursive algorithms
- allow you to reason about the structure by recursion (or induction)

Trees with special properties are usually also recursive

- E.g., full binary trees
- E.g., expression trees

T is a binary tree when either:

- T is an empty tree **(basis)**
- T has a root vertex whose left and right subtrees are binary trees **(recursive definition)**



Dynamic Set ADT / Lookup Table

Key
Galaxy S10
Pixel 3
Mate 20 Pro
iPhone XS

Define a Dynamic Set ADT as:

- a collection of keys
- each key may have some associated data
- insert(key, data) - adds (key, data) into the collection
- search(key) - returns the data associated with key if present, or NULL otherwise. (Alternate: true / false)

Implementations?

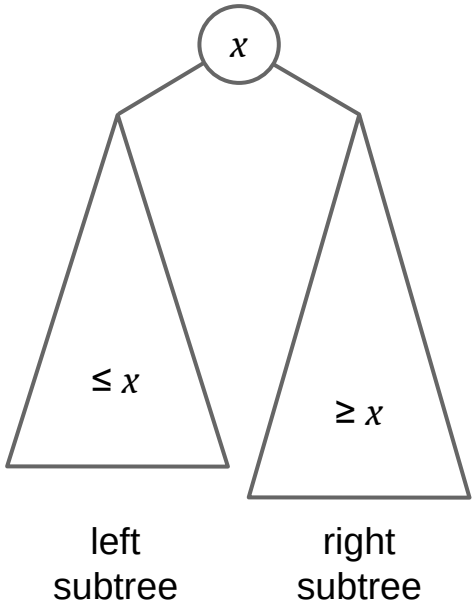
- Coded with an unordered array, search() will be $O(N)$.
- Coded with a sorted array, insert() will be $O(N)$.
- To do better, manage all the keys in a binary tree.

Binary Search Tree

To manage a collection of keys, use a binary tree, one key per node in the tree.

For any key x in the tree:

- its left subtree contains keys ≤ x
- its right subtree contains keys ≥ x



To implement insert() or search():

- compare value with root key
- recurse left if value < key
- recurse right if value > key

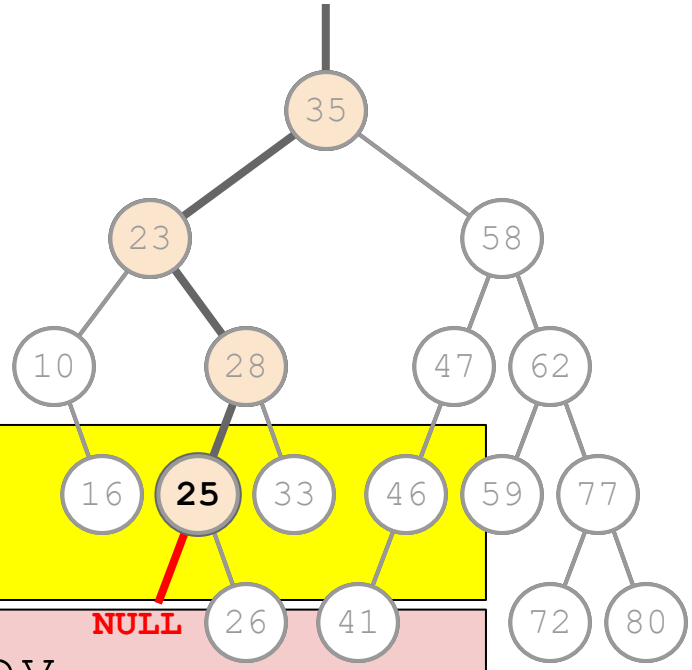
Binary Search Tree: `search()`

A recursive definition for *binary search tree (BST)*:

For root with key = x,

- left subtree is a BST with all keys ≤ x
- right subtree is a BST with all keys ≥ x

How would you search for `24`?




```
bool search(BTnode * root, int target) {
```

- Base Case
  - `return false` if tree empty

- Compare `target` to `root->key`
  - `return true` if found
  - recursively search left if `target < root->key`
  - recursively search right if `target > root->key`

```
}
```

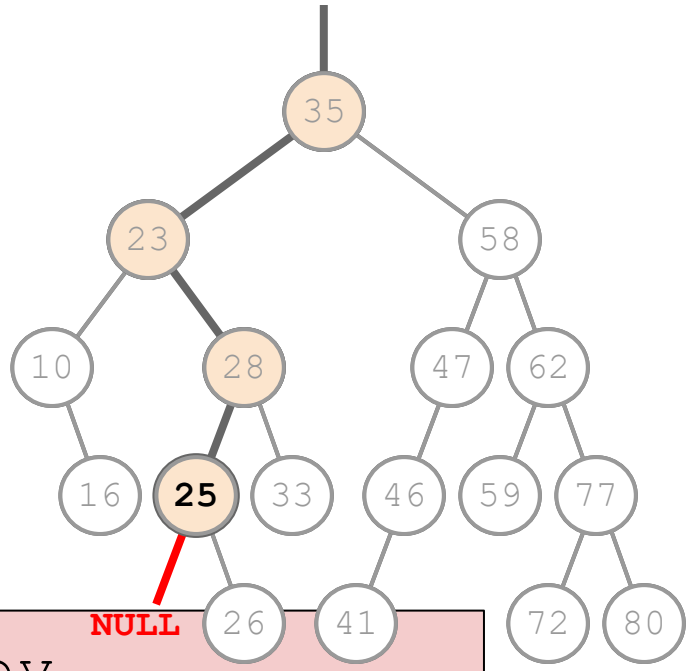
Binary Search Tree: `search()`

A recursive definition for *binary search tree (BST)*:

For root with key = x,

- left subtree is a BST with all keys ≤ x
- right subtree is a BST with all keys ≥ x

How would you search for `24`?




```
bool search(BTnode * root, int target) {
    //  Base case
    if (root == NULL) return false;
    if (root->key == target) return true;
```

- Compare `target` to `root->key`
  - `return true` if found
  - recursively search left if `target < root->key`
  - recursively search right if `target > root->key`

```
}
```

Binary Search Tree: `search()`

A recursive definition for *binary search tree (BST)*:

For root with key = x,

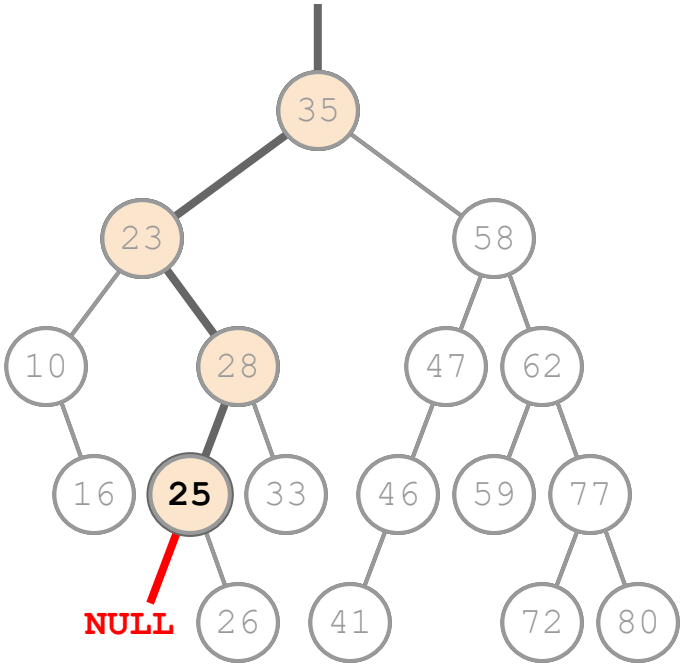
- left subtree is a BST with all keys ≤ x
- right subtree is a BST with all keys ≥ x

How would you search for `24`?

```
bool search(BTnode * root, int target) {
    //  Base case
    if (root == NULL) return false;
    if (root->key == target) return true;

    //  Search left subtree
    if (target < root->key)
        return search(root->left, target);

    //  Search right subtree
    else // root->key < target
        return search(root->right, target);
}
```



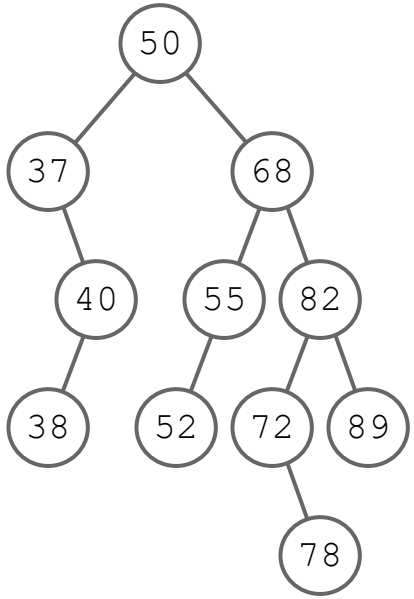
Q. What's the running time?

Binary Search Tree: `insert()`

Q. Where would you insert `82`?

Q. Where would you insert `40`?

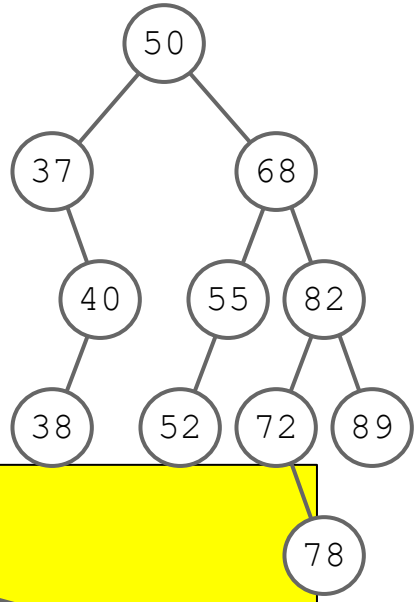
Q. How about `72`, `55`, `38`, `89`, `78`, `52`?



Binary Search Tree: `insert()`

Strategy for `insert()`:

- Similar to `search()`, recurse down the tree until you see a `NULL`, then place the new node there




```
//  Returns the new root of the binary search tree
BTnode * insert(BTnode * root, int key) {
```

- Base Case
  - construct and return new node if tree empty

Q. What's the purpose of this?

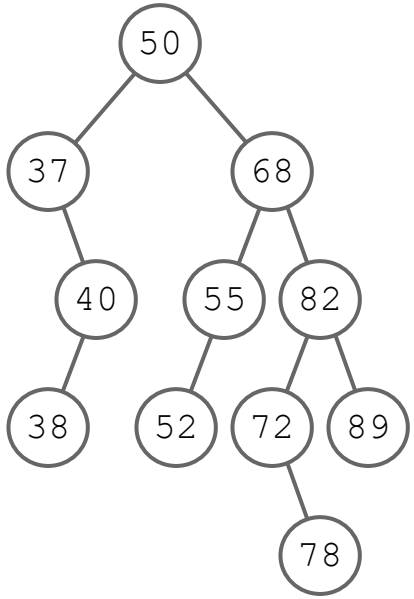
- Compare `key` to `root->key`
  - recursively insert to the left if `key <= root->key`
  - recursively insert to the right if `key > root->key`
- `return root`

```
}
```

Binary Search Tree: `insert()`

Strategy for `insert()`:

- Similar to `search()`, recurse down the tree until you see a `NULL`, then place the new node there




```
//  Returns the new root of the binary search tree
BTnode * insert(BTnode * root, int key) {
    //  Base case
    if (root == NULL)
        return new BTnode(key, NULL, NULL);
```

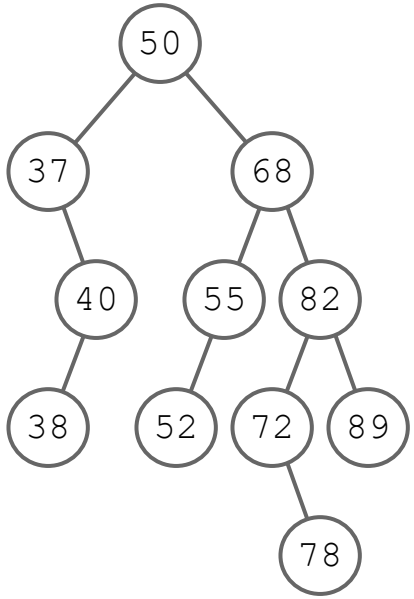
- Compare `key` to `root->key`
  - recursively insert to the left if `key <= root->key`
  - recursively insert to the right if `key > root->key`
- `return root`

```
}
```

Binary Search Tree: `insert()`

Strategy for `insert()`:

- Similar to `search()`, recurse down the tree until you see a `NULL`, then place the new node there




```
//  Returns the new root of the binary search tree
BTnode * insert(BTnode * root, int key) {
    //  Base case
    if (root == NULL)
        return new BTnode(key, NULL, NULL);

    //  Insert to left subtree
    if (key <= root->key)
        root->left = insert(root->left, key);

    //  Insert into right subtree
    else // root->key < key
        root->right = insert(root->right, key);

    return root;
}
```

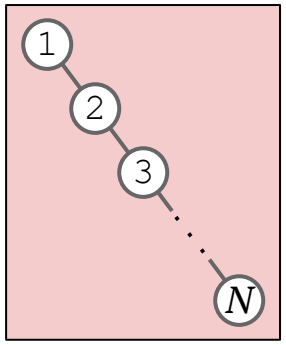
Q. What's the running time?

Running Time Analysis

What's the worst case running time of `search()`?

What about `insert()`?

- both are based on maximum recursion depth
- *depth* of a node = distance from the root
- tree *height* = maximum of all depths



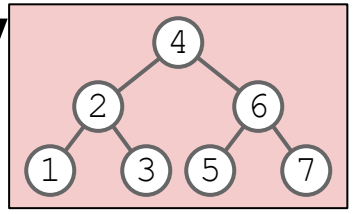
worst-case

How does height (h) relate to the number of nodes (N)?

- worst case? a linear, anaemic tree. $h = O(N)$
- best case? a *balanced* tree. $h = O(\log N)$
- average case? randomly inserted keys. average height = $O(\log N)$

Rule of Thumb: Balanced is best for efficiency

- There are algorithms to re-balance search trees, so that operations are always $O(\log N)$.
- E.g., AVL trees, red-black trees, B-Trees



best case