Binary Search Trees (BST)

- A BST is a ***binary tree*** that (if not empty) also follows the following ***storage rule*** regarding its nodes' items:
  - For *any node* ***n*** of the tree, every item in ***n***'s ***LST***, if not empty, is ***less than*** the item in ***n***
  - For *any node* ***n*** of the tree, every item in ***n***'s ***RST***, if not empty, is ***greater than*** the item in ***n***

NOTE: *any node* implies rule must hold not just for the overall tree but also for each and every subtree

NOTE: Enforcement of the rule requires that the nodes' items can be compared with the usual comparison operators **<**, **>**, **==** *etc.* (*i.e.*, can be arranged in a single line, from small to large)

Caveat Storage rule for defining BST can differ (*points valid under a certain definition may be invalid under other definitions*):

- LST items → *less than or equal*, RST items → *greater than* — *this is used by textbook but we'll use the above instead*
- LST items → *less than*, RST items → *greater than or equal*
- LST items → *less than or equal*, RST items → *greater than or equal*

Binary Search Trees (BST) Searching

- Due to the ordering of nodes (based on the values of items at the nodes), searching for a target value in a BST can be *greatly faster* than in a simple binary tree with no ordering
  - We know that all items in the LST are less than the root's item and all items in the RST are greater than the root's item
  - We compare the target value with that of root's item and make a *better informed decision* based on the result of the comparison
    - Such decision making works only if the specified storage rule is enforced → without the rule, a match could be found anywhere in the tree

Binary Search Trees *E.g.*: `IntSet` using a BST

We can implement the `IntSet` ADT we saw using a BST

```
class IntSet
{
public:
   struct btNode
   {
      int data;
      btNode* left;
      btNode* right;
   };
   ...
   bool contains(int anInt) const;
   bool add(int anInt);
   bool remove(int anInt);
   ...
private:
   btNode* root_ptr;
   ...
};
```

Items are stored in a tree; tree is to be maintained as a BST

Binary Search Trees *E.g.*: `IntSet` using a BST Querying Containment

- The public member function **`contains`** determines whether a BST contains a target item **`anInt`**
  - **`bool contains(int anInt) const;`**
- Because the items in a BST are ordered according to the ***rule*** listed on the first slide, **`contains`** doesn't have to inspect every item in the BST to ascertain containment
  - Let **`cursor`** be a local pointer (initialized to the *root pointer*) used to keep track of the current search position in the BST
  - **`cursor`** will move through the BST in search of **`anInt`**
  - **`cursor`** will use the rule to ***always move along the path where*** **`anInt`** ***might occur*** (the next slide elaborates this)

Binary Search Trees

E.g.: `IntSet` using a BST
Querying Containment

- There are ***four*** possibilities at each search position:
  - **`cursor`** becomes ***0***, indicating that we've moved off the bottom of the tree
    - Run out of nodes where a match can be found
    - **`return false;`**
  - **`anInt`** is ***smaller*** than the item at **`cursor`** node
    - **`anInt`** can only occur in the ***left*** subtree of the BST rooted at **`cursor`** node
    - **`cursor = cursor->left;`**
  - **`anInt`** is ***larger*** than the item at **`cursor`** node
    - **`anInt`** can only occur in the ***right*** subtree of the BST rooted at **`cursor`** node
    - **`cursor = cursor->right;`**
  - **`anInt`** is ***equal*** to the item at **`cursor`** node
    - Match of **`anInt`** found
    - **`return true;`**
- It is straightforward to implement **`contains`** with a loop

Binary Search Trees

E.g.: `IntSet` using a BST
Adding an Item

- The public member function **`add`** inserts an entry **`anInt`** (received as a parameter) into a BST
  - **`bool add(int anInt);`**
- The **`add`** function first deals with the special case in which the tree is ***empty***

```
root_ptr = new btNode;
root_ptr->data = anInt;
root_ptr->left = root_ptr->right = 0;
```

- If the tree is ***not empty***, the function searches the tree to see if a spot for adding **`anInt`** as a leaf can be found
  - Again, let local pointer **`cursor`** (initialized to the root pointer) be used to keep track of the current search position in the BST

Binary Search Trees

E.g.: `IntSet` using a BST
Adding an Item

- If **`anInt`** is ***equal to*** the data at **`cursor`** node
  - Entry already exists
    - **`return false;`**

Binary Search Trees

E.g.: `IntSet` using a BST
Adding an Item

- If **`anInt`** is ***less than*** the data at **`cursor`** node
  - Check the **`left`** field at **`cursor`** node
    - If it's ***0***, create a new node containing **`anInt`** and make the **`left`** field of **`cursor`** node point to it (and task is done):

```
cursor->left = new btNode;
cursor->left->data = anInt;
cursor->left->left = cursor->left->right = 0;
```

    - If it's ***not 0***, move the cursor to the ***left*** and continue the search for a correct spot to insert the entry:

```
cursor = cursor->left;
```

Binary Search Trees

E.g.: `IntSet` using a BST
Adding an Item

- If **`anInt`** is ***greater than*** the data at **`cursor`** node
  - Check the **`right`** field at **`cursor`** node
    - If it's ***0***, create a new node containing **`anInt`** and make the **`right`** field of **`cursor`** node point to it (and task is done):

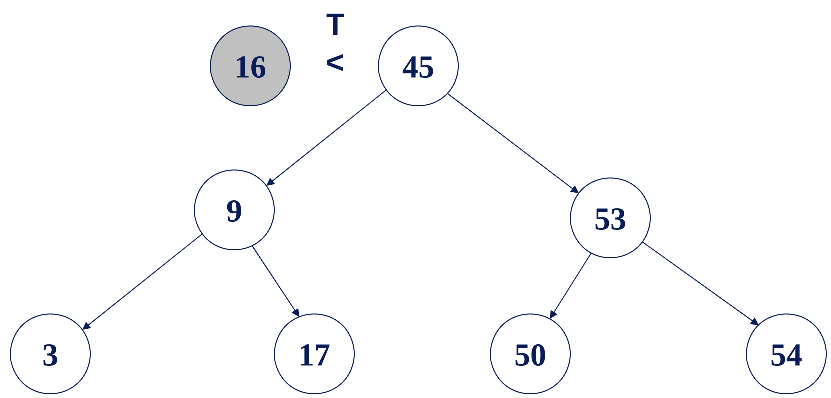
```
cursor->right = new btNode;
cursor->right->data = anInt;
cursor->right->left = cursor->right->right = 0;
```

    - If it's ***not 0***, move the cursor to the ***right*** and continue the search for a correct spot to insert the entry:

```
cursor = cursor->right;
```

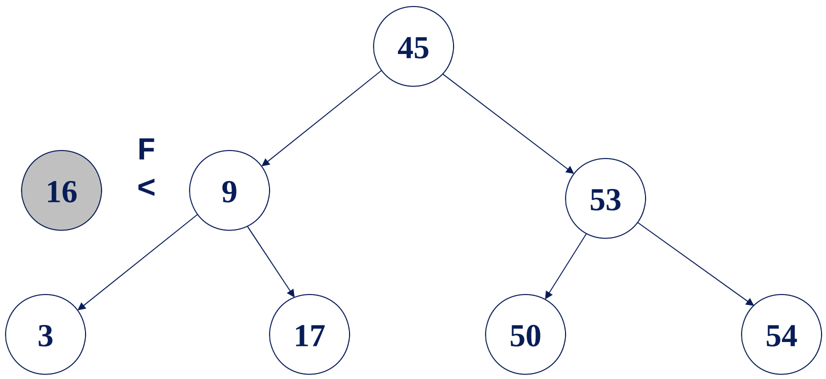

Binary Search Trees

Inserting a Node



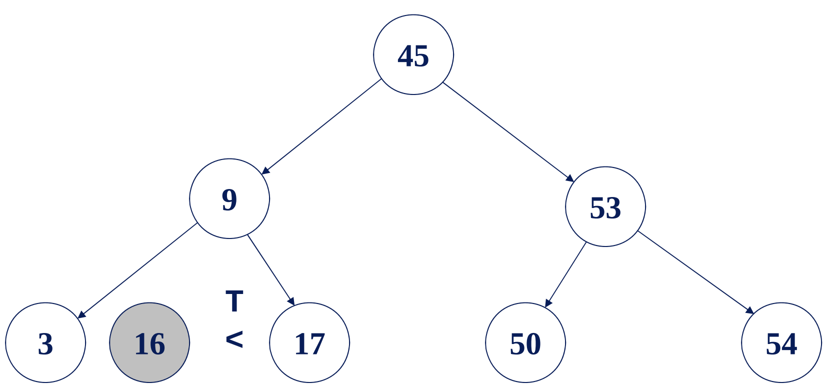
Binary Search Trees

Inserting a Node



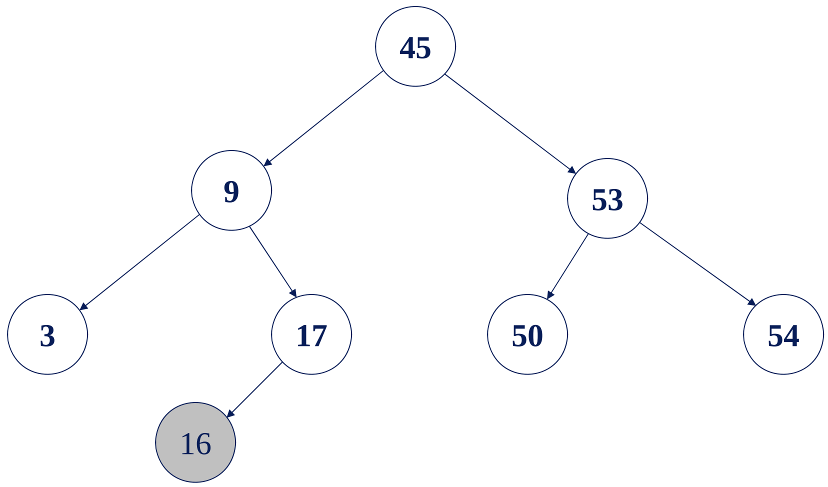
Binary Search Trees

Inserting a Node



Binary Search Trees

Inserting a Node



Binary Search Trees

E.g.: `IntSet` using a BST
Removing an Item

- The public member function **`remove`** removes a specified item **`anInt`** (if exists) from a BST
  - **`bool remove(int anInt);`**
- To implement **`remove`** ***directly*** entails *many special cases* to be dealt with and a *precursor* (similar to that used when removing an item from a singly linked list) be maintained
  - We will (next) consider a popular ***indirect*** way
    - (often referred to as the "usual standard way" or something similar)
  - It uses two helper functions (next)

Binary Search Trees

Removing a Node
Helper Functions

(Each receives, among others, root pointer of BST as *reference parameter* **bst_root**)

- **bst_remove** BST may be ***empty*** or ***non-empty***
  - If the target **anInt** was in the BST, then **anInt** has been removed, **bst_root** now points to the root of the new (smaller) BST, and the function returns true
  - If the target **anInt** was not in the BST, then the BST is unchanged, and the function returns false
- **bst_remove_max** For ***non-empty*** BST only
  - The ***largest item*** in the BST has been removed, and **bst_root** now points to the root of the new (smaller) BST
  - A copy of item removed is returned via a *reference parameter*

Binary Search Trees

Removing a Node
bst_remove Helper

- **bst_remove** has a recursive implementation to remove the target **anInt**
  - ***Tree*** could be ***empty*** → function simply returns false
  - Target **anInt** could be ***less than*** the ***root*** → make a ***recursive call to the left***
    - `bst_remove(bst_root->left, anInt);`
  - Target **anInt** could be ***greater than*** the ***root*** → make a ***recursive call to the right***
    - `bst_remove(bst_root->right, anInt);`
  - Target **anInt** could be ***equal*** to the ***root*** → **. . .** (next)

Target Equal to Root

- If root node's value is equal to target **anInt**, we have a match
- To remove this node, we have to consider two situations:
  - ***Root has no children or only 1 child***
    - (both children are empty or only 1 child is non-empty)
  - ***Root has two children***
    - (both children are non-empty)

Target Equal to Root
Root Has No Children or Only 1 Child

- In this case we can delete the root node and make 0 (if no children) or the root of the non-empty child (if only 1 child) the new root node
- This requires three steps:
  (1st and 3rd steps are identical for the different cases)

	Has No Children	*Has Only Left Child*	*Has Only Right Child*
❶	`old_bst_root = bst_root;`		
❷	`bst_root = 0;`	`bst_root = bst_root->left;`	`bst_root = bst_root->right;`
❸	`delete old_bst_root;`		

bst_remove Helper

Target Equal to Root and Root Has 2 Children

- In this case we have to delete the root node and make an ***appropriate child*** the new root node
- We'll *replace root w/ the node w/ the **largest** value in **LST***
  - Call **`bst_remove_max`** on the ***LST***
    - Have it remove the node with the largest value from the LST
    - Have it set **`bst_root->data`** (passed as *reference parameter*) equal to the largest value (that was in the node removed from the LST)
  - **`bst_remove_max(bst_root->left, bst_root->data);`**

> We could just as well
> *replace root w/ the node w/ the **smallest** value in **RST***
> by calling a corresponding **`bst_remove_min`** on the ***RST***

bst_remove_max Helper

Outline of (Recursive) Algorithm

- If (tree has ***no right child***) // largest item @ root (base case)
  - Copy root node's data into the **`data`**-field reference parameter
  - Delete root node and make left child (may be 0) the new root
    - (using 3 steps similar to that given for **`bst_remove`**)
- Else // tree has ***right child*** (largest item not @ root)
  - Make recursive call to delete largest item from RST
    - Passing along the **`data`**-field *reference parameter*
      - (side effect to be delivered via the parameter is fulfilled in base case above)

Binary Search Trees — Balanced vs Unbalanced

- There is no requirement that a BST be full, complete, balanced, *etc*.
  - Only that the rule listed on the first slide be met
- That means it is possible to have a BST that's very unbalanced (for instance, only a single node in the LST but 1000 nodes in the RST)

Binary Search Trees — Balanced vs Unbalanced

- If the tree (and the subtrees) has drastically more nodes to one side over the other, the performance improvement in searching is diminished
  - In the worst case, it's no better than a linked list
- It'd be ideal if the node counts are equal (or within 1 node) for each and every corresponding subtree pair, . . .

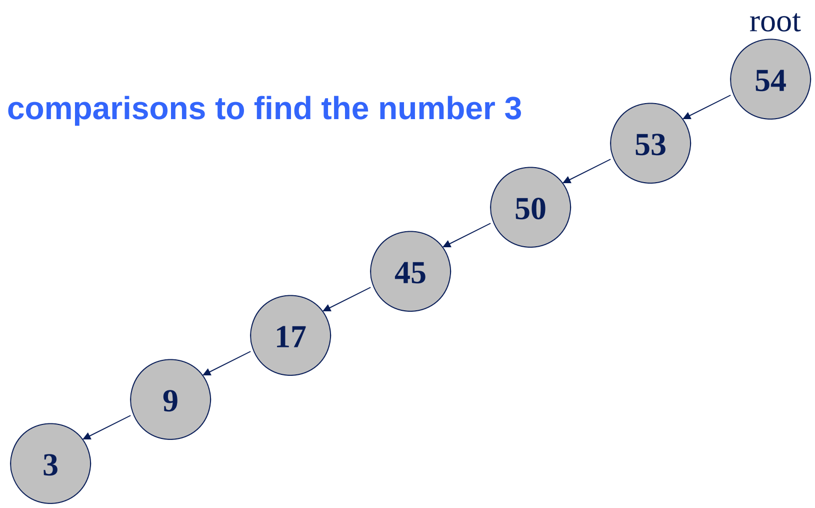
  (often described as ***weight-balanced***, more on this to come)

  . . . so that each decision roughly cuts in half the number of nodes that remain to be searched

Binary Search Trees

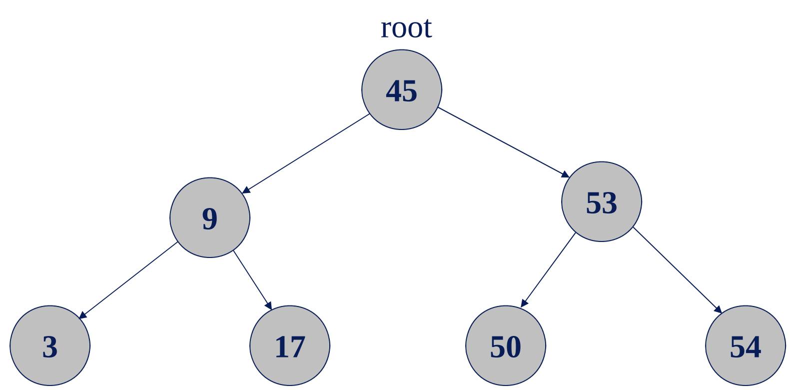
Balanced vs Unbalanced

7 comparisons to find the number 3



Binary Search Trees

Balanced vs Unbalanced



3 comparisons to find the number 3

Binary Search Trees — Balanced vs Unbalanced

- A ***weight-balanced*** BST is one in which each subtree has either the same number of nodes as the other or is off by only 1 node
  - This property holds for *any node* in the tree
- This type of tree offers the best performance improvement for searches
  - Each decision "prunes" as many nodes as possible
- However, it is typically too expensive to maintain a weight-balanced BST
  - (motivates ***height-balanced*** BSTs such as AVL trees)

Textbook Readings

- Chapter 10
  - Section 10.5