Binary Search Trees (BST)

- A BST is a ***binary tree*** that (if not empty) also follows ***two storage rules*** regarding its nodes' items:
  - For any node ***n*** of the tree, every item in ***n***'s ***left subtree*** (LST), if not empty, is ***less than or equal*** to the item in ***n***
  - For any node ***n*** of the tree, every item in ***n***'s ***right subtree*** (RST), if not empty, is ***greater than*** the item in ***n***

  NOTE: Enforcement of the rules requires that the nodes' items can be compared with the usual comparison operators **<**, **>**, **==** *etc*.
  - More rigorously, the operators must form a ***strict weak ordering*** as described in Fig. 6.4 on p. 302 (p. 293 for 2nd edition) of the textbook
  - (items can be arranged in a single line, proceeding from smaller to larger)

  CAVEAT: Slightly different storage rules are also used by others
  - Items in LST → *less than*, nodes in RST → *greater than or equal*
  - Items in LST → *less than*, nodes in RST → *greater than*

Binary Search Trees (BST) Searching

- Searching in a BST can be *drastically faster* than in a simple binary tree because of the order placed on the nodes (with respect to their items)
  - We know that all the items in the left subtree are less than or equal to the root's item and the items in the right subtree are greater than the root's item
  - We can compare the item we are seeking with the root's item and go the correct way based on the result of the comparison
    - This is possible because of the two rules mentioned in the preceding slide → without the rules, the item sought could be anywhere in the tree

Binary Search Trees

E.g.: *bag* using a BST

The *bag* ADT can be made more efficient by implementing it using a BST

```
template <class Item>
class bag
{
public:
   ... // typedef's, constructors, destructor
   size_type count(const Item& target) const;
   void insert(const Item& entry);
   bool erase_one(const Item& entry);
   size_type erase(const Item& entry);
   ... // prototypes of other public member functions
private:
   binary_tree_node<Item> *root_ptr;
   ... // other appropriate private members
};
```


Binary Search Trees

E.g.: *bag* using a BST
Counting Occurrences

- The public member function **`count`** determines how many times a **`target`** item occurs in a BST
  - **`size_type count(const Item& target) const;`**
- Because the items in a BST are ordered according to the ***two rules*** listed on the first slide, **`count`** doesn't have to examine every item in the BST to count the occurrences
  - Let **`result`** be a local variable (initialized to *zero*) used to keep track of the number of occurrences
  - 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 **`target`**
  - **`cursor`** will use the two rules to ***always move along the path where*** **`target`** ***might occur*** (the next slide elaborates this)

Binary Search Trees

E.g.: *bag* using a BST Counting Occurrences

- There are ***four*** possibilities at each search position:
  - `cursor` becomes ***0***, indicating that we've moved off the bottom of the tree
    - All occurrences counted
    - `return result;`
  - `target` is ***smaller*** than the item at the `cursor` node
    - `target` can only occur in the ***left*** subtree of the BST rooted at the `cursor` node
    - `cursor = cursor->left();`
  - `target` is ***larger*** than the item at the `cursor` node
    - `target` can only occur in the ***right*** subtree of the BST rooted at the `cursor` node
    - `cursor = cursor->right();`
  - `target` is ***equal*** to the item at the `cursor` node
    - An occurrence of `target` found and `target` can occur again only in the ***left*** subtree of the BST rooted at the `cursor` node
    - `++result;` then `cursor = cursor->left();`
- It is straightforward to implement `count` with a loop

Binary Search Trees

Inserting a Node

- The public member function `insert` inserts a new entry (received as a parameter) into a BST
  - `void insert(const Item& entry);`
- The `insert` function first deals with the special case in which the tree is ***empty***
  - `root_ptr = new binary_tree_node<Item>(entry);`
- If the tree is ***not empty***, the function search the tree to find the correct spot for inserting the entry as a leaf
  - Again, let a local pointer called `cursor` (initialized to the root pointer) be used to keep track of the current search position in the BST

Binary Search Trees

Inserting a Node
The `insert` function

- Suppose the new entry is ***less than or equal*** to the data at the **`cursor`** node
  - Check the **`left_field`** at **`cursor`** node
    - If it's ***0***, create a new node containing the entry and make the **`left_field`** of the **`cursor`** node point to it (and the task is then finished):

```
cursor->set_left(new binary_tree_node<Item>(entry));
```

    - 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

Inserting a Node
The `insert` function

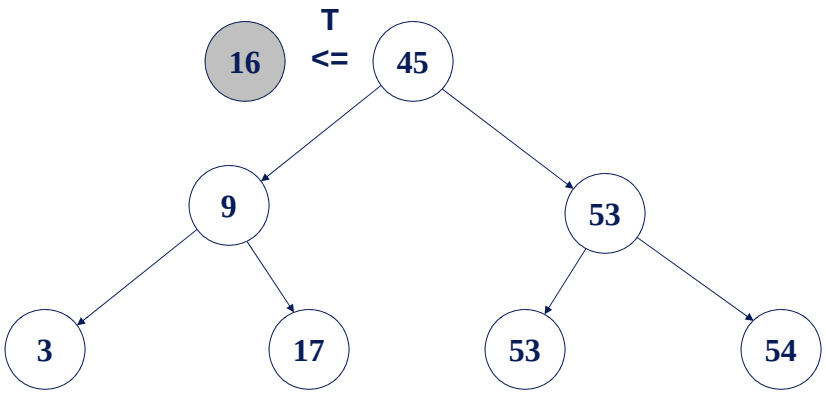
- Suppose the new entry is ***greater than*** the data at the **`cursor`** node
  - Check the **`right_field`** at **`cursor`** node
    - If it's ***0***, create a new node containing the entry and make the **`right_field`** of the **`cursor`** node point to it (and the task is then finished):

```
cursor->set_right(new binary_tree_node<Item>(entry));
```

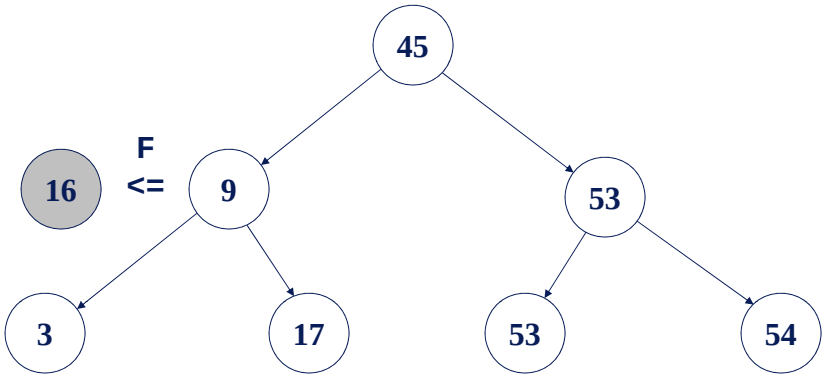
    - If it's ***not 0***, move the cursor to the ***right*** and continue the search for a correct spot to insert the entry:

```
cursor = cursor->right();
```

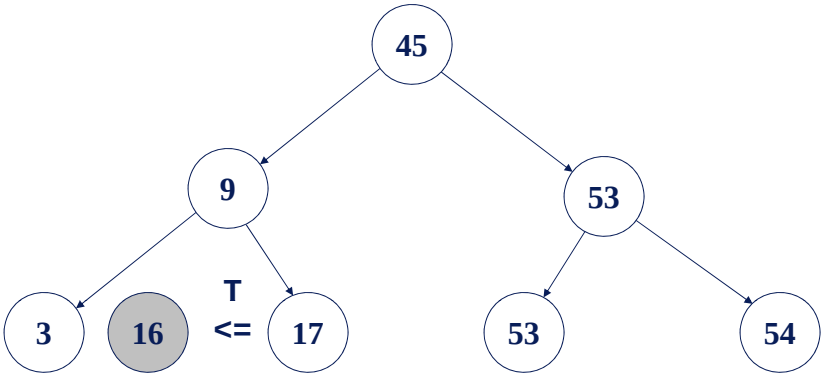
Binary Search Trees — Inserting a Node



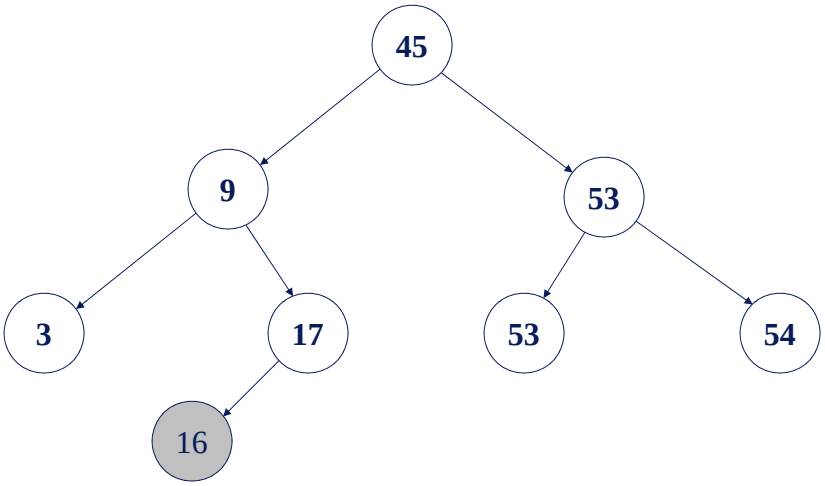
Binary Search Trees — Inserting a Node



Binary Search Trees — Inserting a Node



Binary Search Trees — Inserting a Node



Binary Search Trees

Removing a Node

- The public member function **`erase`** removes all copies of a specified item from a BST while **`erase_one`** removes one copy of a specified item from a BST
  - **`size_type erase(const Item& target);`**
  - **`bool erase_one(const Item& target);`**
  - Implementations are similar ➔ suffice to look at **`erase_one`**
- To implement **`erase_one`** ***directly*** requires that *many special cases* be dealt with and a *precursor* (similar to that used when removing an item from a singly linked list) be maintained
  - The textbook describes an ***indirect*** method that uses ***two auxiliary functions***

Binary Search Trees

Removing a Node
Auxiliary functions

- **`bst_remove`** BST may be ***empty*** or ***non-empty***
  - If the target was in the BST, then ***one copy*** of the target has been removed, **`root_ptr`** now points to the root of the new (smaller) BST, and the function returns true
  - If the target 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 **`root_ptr`** now points to the root of the new (smaller) BST
  - A copy of the removed item is returned via a reference parameter

- **bst_remove** has a recursive implementation to remove the target
  - The ***tree*** could be ***empty*** ➔ the function simply returns
  - The ***target*** could be ***less than*** the ***root*** ➔ make a ***recursive call to the left***
    - `bst_remove(root_ptr->left(), target);`
  - The ***target*** could be ***greater than*** the ***root*** ➔ make a ***recursive call to the right***
    - `bst_remove(root_ptr->right(), target);`
  - The ***target*** could be ***equal*** to the ***root*** ➔ ...

Target Equal to Root

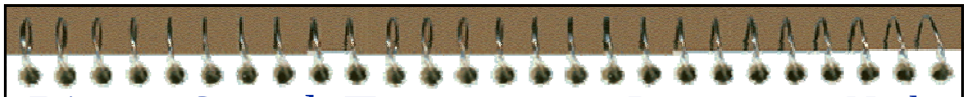
- If the root node's value is equal to the target's value, we have found a copy of the target
- We can't simply delete this node because it may have children ➔ need to deal with two cases:
  - ***Root has no left child***
  - ***Root has a left child***

Target Equal to Root
Root Has No Left Child

- In this case we can delete the root entry and make the right child the new root node
- This requires three steps:
  - `oldroot_ptr = root_ptr;`
  - `root_ptr = root_ptr->right();`
  - `delete oldroot_ptr;`

Target Equal to Root
Root Has a Left Child

- In this case we cannot delete the root entry and make the right child the new root node
- We need to *replace root with the largest entry in the left subtree* ➔ where the second auxiliary function comes in
  - Call **`bst_remove_max`** on the ***left subtree***
    - Have the function remove the largest entry from the left subtree
    - Have the function set **`root_ptr->data()`** equal to the data value of the entry removed (from the left subtree)
  - `bst_remove_max( root_ptr->left(), root_ptr->data() );`

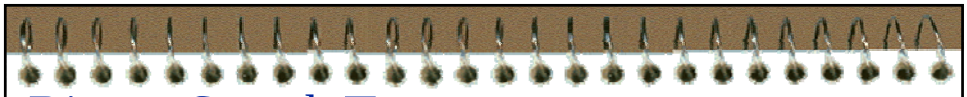


Binary Search Trees

Removing a Node `bst_remove_max`

Outline of (Recursive) Algorithm

- If (tree has ***no right child***) // largest item @ root (base case)
  - Copy root's data field into data field reference parameter
  - Delete root entry and make left child the new root node
    - (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



Binary Search Trees

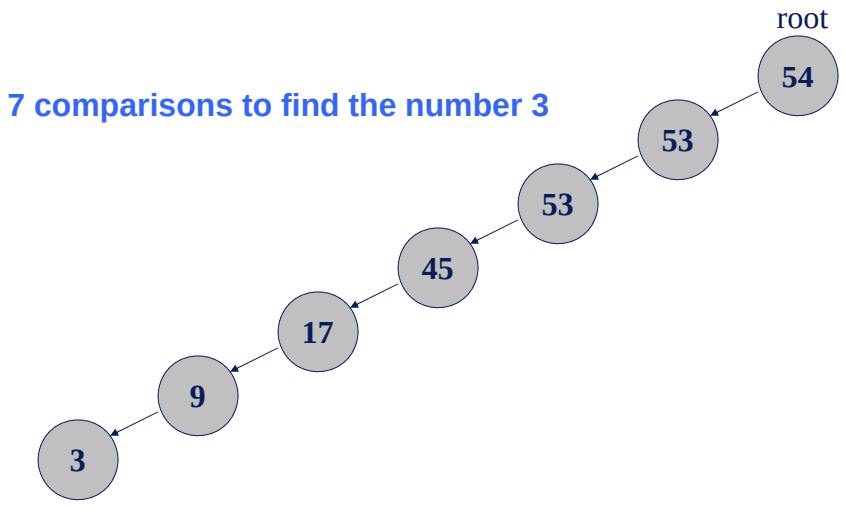
Balanced vs Unbalanced

- There is no requirement that a BST be full, complete, balanced, *etc.*
  - Only that the two rules listed on the first slide be met
- That means it is possible to have a BST in which there is only one node in the left subtree of root and perhaps 1000 nodes in the right subtree

Binary Search Trees — Balanced vs Unbalanced

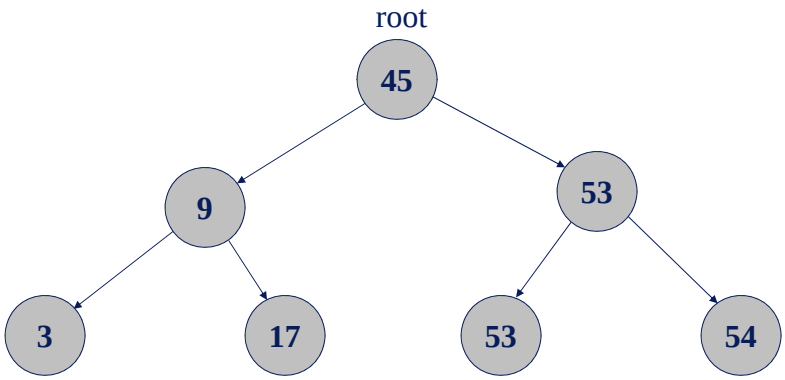
- If the tree (and the subtrees) has drastically more nodes to one side over the other, the performance improvement in searching is diminished
  - In the worst case, it's no better than a linked list
- The ideal is to have the same number of nodes in each subtree (or within 1 node), so that each decision 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 balanced binary search tree 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

Textbook Readings

- Chapter 10
  - Section 10.5