Binary Search Trees

CSE 2011

Fall 2009

10/21/2009 1:00 PM 1

Dictionary ADT

- The dictionary ADT models a searchable collection of key-element items
- The main operations of a dictionary are searching, inserting, and deleting items
- Multiple items with the same key are allowed
- Applications:
  – address book
  – credit card authorization
  – SIN database
  – student database

Dictionary ADT methods:

- find(k): if the dictionary has an item with key k, returns its element, else, returns NULL
- findAll(k): returns an iterator of entries with key k
- insert(k, o): inserts item (k, o) into the dictionary
- remove(k): if the dictionary has an item with key k, removes it from the dictionary and returns its element, else returns NULL
- removeAll(k): remove all entries with key k; return an iterator of these entries.
- size(), isEmpty()

2

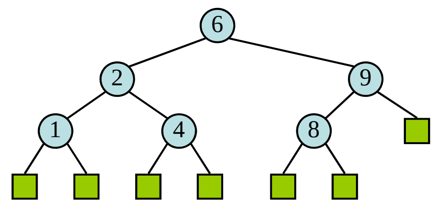
Binary Search Tree

- A binary search tree is a binary tree storing keys (or key-element pairs) at its internal nodes and satisfying the following property:

  Let ***u***, ***v***, and ***w*** be three nodes such that ***u*** is in the left subtree of ***v*** and ***w*** is in the right subtree of ***v***. We have
  $key(u) \leq key(v) \leq key(w)$

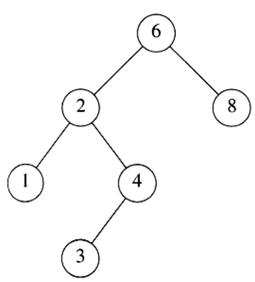
- External nodes (dummies) do not store items (non-empty proper binary trees, for coding simplicity)

- An inorder traversal of a binary search trees visits the keys in increasing order
- The left-most child has the smallest key
- The right-most child has the largest key

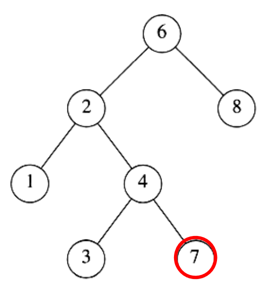


3

Binary Search Trees



A binary search tree

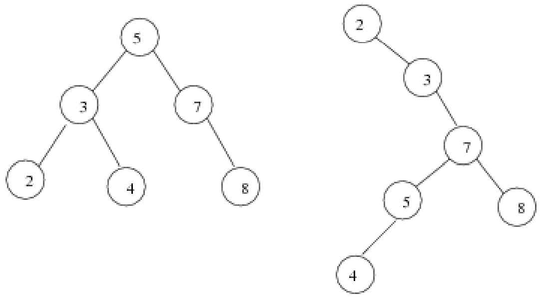


Not a binary search tree

4

Binary Search Trees

The same set of keys may have different BSTs.

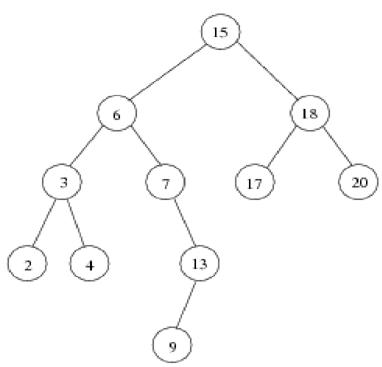


- Average depth of a node is O(logN).
- Maximum depth of a node is O(N).
- Smallest key? Largest key?

5

Inorder Traversal of BST

- Inorder traversal of BST prints out all the keys in sorted order.

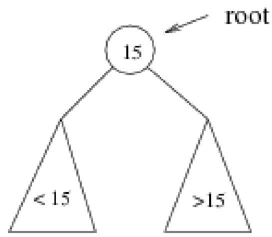


Inorder: 2, 3, 4, 6, 7, 9, 13, 15, 17, 18, 20

6

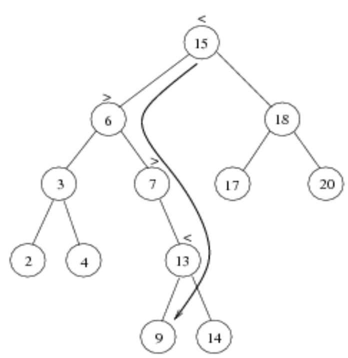
Searching BST

- If we are searching for 15, then we are done.
- If we are searching for a key < 15, then we should search in the left subtree.
- If we are searching for a key > 15, then we should search in the right subtree.



7

Example: Search for 9 ...



Search for 9:

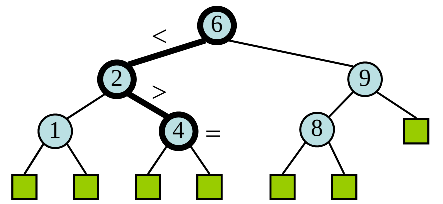
1. compare 9:15(the root), go to left subtree;
2. compare 9:6, go to right subtree;
3. compare 9:7, go to right subtree;
4. compare 9:13, go to left subtree;
5. compare 9:9, found it!

8

Search

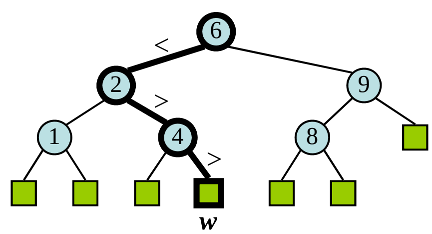
- To search for a key ***k***, we trace a downward path starting at the root
- The next node visited depends on the outcome of the comparison of ***k*** with the key of the current node
- If we reach a leaf, the key is not found and we return v (where the key should be if it will be inserted)
- Example: TreeSearch(4, T.root())
- Running time: ?

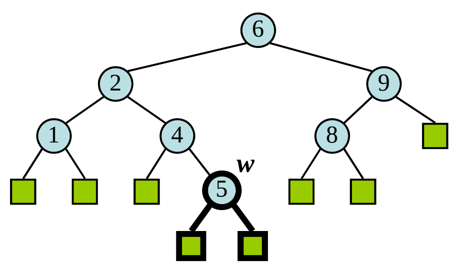
```
Algorithm TreeSearch(k, v)
    if T.isExternal (v)
        return (v);      // or return NO_SUCH_KEY
    if k < key(v)
        return TreeSearch(k, T.leftChild(v))
    else if k = key(v)
        return v
    else { k > key(v) }
        return TreeSearch(k, T.rightChild(v))
```



Insertion

- To perform operation insertItem(k, o), we search for key k
- Assume k is not already in the tree, and let w be the leaf reached by the search
- We insert k at node w and expand w into an internal node using *insertAtExternal*(w, (k,e))
- Example: *insertAtExternal*(w, (5,e)) with e having key 5





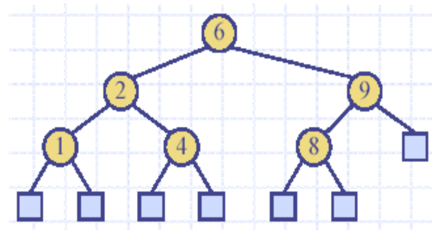
Insertion (2)

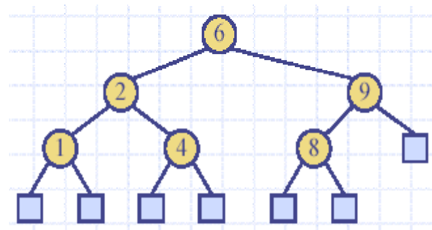
Insertion with duplicate keys

- Example: insert(2)
- Call *TreeSearch*(k, *leftChild*(w)) to find the leaf node for insertion
- Can insert to either the left subtree or the right subtree (call *TreeSearch*(k, *rightChild*(w))

Running time: ?

Homework: implement method findAll(k)





11

Insertion Algorithm

No duplicate keys

```
Algorithm TreeInsert(k, e, v) {
    w = TreeSearch(k, v);
    T.insertAtExternal(w, (k,e));
    return w;
}
```

Example:
TreeInsert(5, e, T.root())

With duplicate keys

```
Algorithm TreeInsert(k, e, v) {
    w = TreeSearch(k, v);
    if k = key(w)   // key exists
      return TreeInsert(
          k, e, T.leftChild(w));
    T.insertAtExternal(w, (k,e));
    return w;
}
```

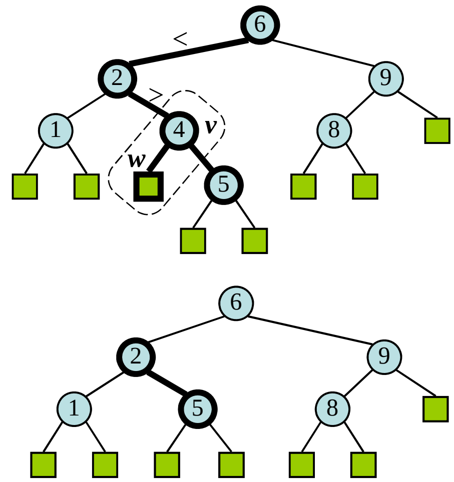
Example:
TreeInsert(2, e, T.root())

12

Deletion

- To perform operation removeElement(***k***), we search for key ***k***
- Assume key ***k*** is in the tree, and let let ***v*** be the node storing ***k***
- Case 1:

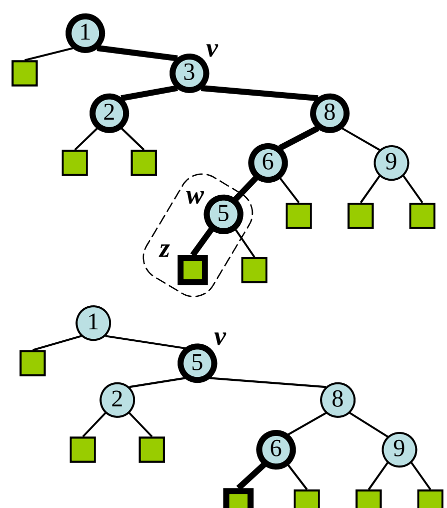
  If node ***v*** has a leaf child ***w***, we remove ***v*** and ***w*** from the tree with operation removeExternal(***w***)
- Example: remove 4
- Case 2: next slide



13

Deletion (2)

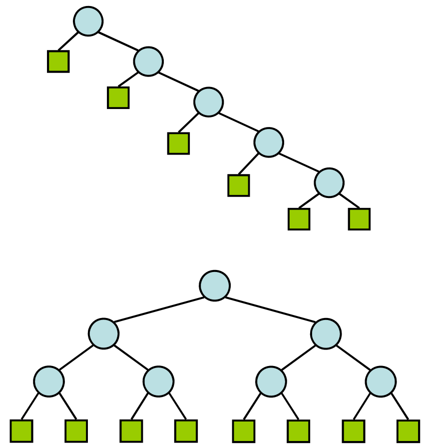
- We consider the case where the key ***k*** to be removed is stored at a node ***v*** whose children are both internal
  - we find the internal node ***w*** that follows ***v*** in an inorder traversal (who is ***w***?)
  - we copy ***key***(***w***) into node ***v***
  - we remove node ***w*** and its left child ***z*** (which must be a leaf) by means of operation removeExternal(***z***)
- Example: remove (3)
- Running time: ?
- Homework: implement removeAll(k)



14

Performance

- Consider a dictionary with n items implemented by means of a binary search tree of height h
  - the space used is $O(n)$
  - methods find(k) , insert() and remove(k) take $O(h)$ time
- The height h is $O(n)$ in the worst case and $O(\log n)$ in the best case



15

Next time …

- Midterm test, Oct. 27, 17:25-18:45.
- AVL trees

16