Computer Science 331

Binary Search Trees — Insertion and Deletion

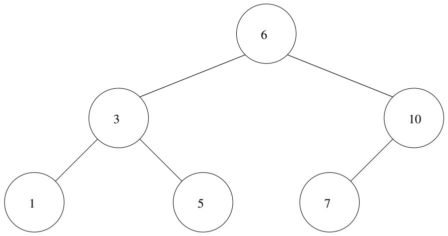
Mike Jacobson

Department of Computer Science
University of Calgary

Lecture #13

Outline

Insertion: An Example



Idea:

Nodes Visited (inserting 9):

- Start at 6 :
- Next node
- Next node
- Next node

A Recursive Insertion Algorithm

```
// Non-recursive public function calls recursive worker function
public void insert(E key, V value)
  { root = insert(root, key, Value); }

protected
bstNode<E,V> insert(bstNode<E,V> T, E newKey, V newValue) {
  if (T == null)

  else if (newKey.compareTo(T.key) < 0)

  else if (newKey.compareTo(T.key) > 0)

  else

  return T;
}
```

Analysis: Partial Correctness

Prove that `insert` is partially correct for all trees `T` of height h.

Base cases are correct (by inspection):

- empty tree replaced by new node containing `newKey` and `newValue`
- if `T.key == newKey`, a `KeyFoundExcpetion` is thrown

Assume that the algorithm is correct for all trees of height $\leq h-1$:

- if `newKey < T.key`, key/value inserted in left subtree
- if `newKey > T.key`, key/value inserted in right subtree
- in either case, algorithm is called recursively on a subtree of height at most $h-1$ and new subtree is correct by assumption
- the new `T` is still a BST, because both children are BSTs and the new element was added to the correct subtree

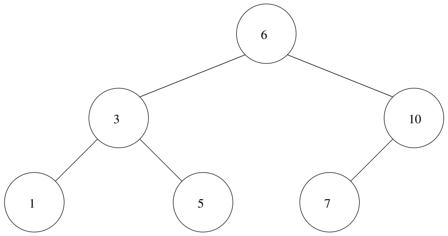
---

Termination and Bound on Running Time

Let h_i denote the height of the subtree with root x at level i of the recursion. Consider the function $f(i) = h_i + 1$:

Worst case running time is $\Theta(h)$:

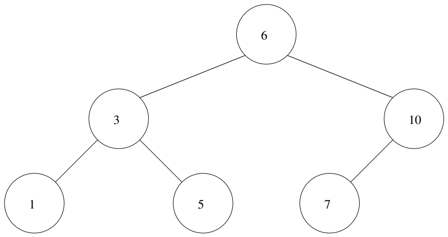
Deletion: Four Important Cases



Key is/has . . .

1. Not Found (Eg: Delete 8)
2. At a Leaf (Eg: Delete 7)
3. One Child (Eg: Delete 10)
4. Two Children (Eg: Delete 6)

First Case: Key Not Found



Idea:

Nodes Visited (delete 8):

- Start at 6 :
- Next node
- Next node
- Next node

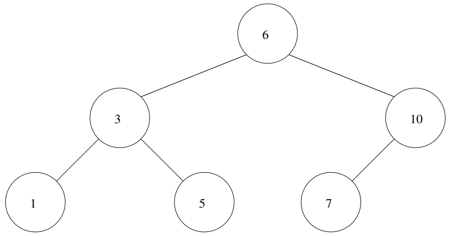
Algorithm and Analysis

```
protected bstNode<E,V> delete(bstNode<E,V> T, E key) {
  if (T != null) {
    if (key.compareTo(T.key) < 0)
      T.left =  delete(T.left, key);
    else if (key.compareTo(T..key) > 0)
      T.right = delete(T.right,key);
    else if ...
      // found node with given key
  }
  else
    throw new KeyNotFoundException();
  return T;
}
```

Correctness and Efficiency For This Case:

- tree is not modified if key is not found (base case will be reached)
- worst-case cost $\Theta(h)$ (same as `search`)

Second Case: Key is at a Leaf



Idea:

Nodes Visited (delete 7):

- Start at 6 :
- Next node
- Next node

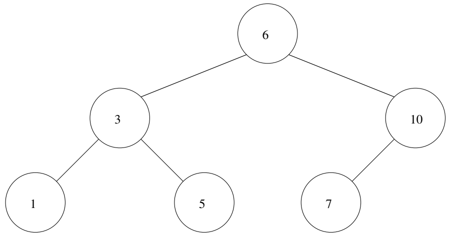
Algorithm and Analysis

Extension of Algorithm:

```
else if ()
```

Correctness and Efficiency For This Case:

Third Case: Key is at a Node with One Child



Idea:

Nodes Visited (delete 10):

- Start at 6 :
- Next node

Algorithm and Analysis

Extension of Algorithm:

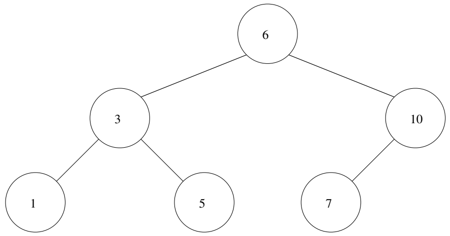
```
else if (T.left == null)

else if (T.right == null)
```

Correctness and Efficiency For This Case:

-
    -
    -
-

Fourth Case: Key is at a Node with Two Children



Idea:

Nodes Visited (delete 6):

- Start at 6 :
-
-

Algorithm and Analysis

Extension of Algorithm:

```
else {



}
```

Correctness and Efficiency For This Case:

-
-
    -
    -

More on Worst Case

All primitive operations (`search`, `insert`, `delete`) have worst-case complexity $\Theta(n)$

- all nodes have exactly one child (i.e., tree only has one leaf)
- Eg. will occur if elements are inserted into the tree in ascending (or descending) order

On average, the complexity is $\Theta(\log n)$

- Eg. if the tree is full, the height of the tree is $h = \log_2(n+1) - 1$

Need techniques to ensure that all trees are close to full

- want $h \in \Theta(\log n)$ in the worst case
- one possibility: red-black trees (next three lectures)

References

Binary Trees:

- Text, Sections 8.1-8.3 Discussed in more detail, including algorithms for tree traversals

Binary Search Trees:

- Text, Section 8.4

Note: Deletion Case 4 (deleting a node with two children) is handled slightly differently in the text — the node is replaced by its “in-order predecessor” as opposed to the “in-order successor” as done in the notes. Both methods are equally acceptable.