CSE373: Data Structures & Algorithms

Lecture 6: Binary Search Trees

Catie Baker
Spring 2015
Announcements

• HW2 due start of class Wednesday April 15th
Previously on CSE 373

– Dictionary ADT
  • stores (key, value) pairs
  • *find*, *insert*, *delete*

– Trees
  • Terminology
  • Binary Trees
Reminder: Tree terminology

- **Node / Vertex**
- **Left subtree**
- **Right subtree**
- **Root**
- **Edges**
- **Leaves**
Example Tree Calculations

Recall: **Height** of a tree is the **maximum** number of edges from the **root** to a **leaf**.

What is the **height** of this tree?
- Height of A = 0
- Height of B = 1

What is the **depth** of node G?
- Depth of G = 2

What is the **depth** of node L?
- Depth of L = 4
Binary Trees

- **Binary tree**: Each node has at most 2 children (branching factor 2)

- Binary tree is
  - A root *(with data)*
  - A left subtree *(may be empty)*
  - A right subtree *(may be empty)*

- Special Cases

---

**Complete Tree**

```
        A
       / \  
      B   C
     / \  /  
    D   E F  
```

**Perfect Tree**

```
        A
       / \  
      B   C
     / \  /  
    D   E F  G
```

**Full Tree**

```
        A
       / \  
      B   C
     /  H  I
    D   E F G
```
Tree Traversals

A traversal is an order for visiting all the nodes of a tree

- **Pre-order**: root, left subtree, right subtree
  
  \[ + \times 2 4 5 \]

- **In-order**: left subtree, root, right subtree
  
  \[ 2 \times 4 + 5 \]

- **Post-order**: left subtree, right subtree, root
  
  \[ 2 4 \times 5 + \]
void inOrderTraversal(Node t) {
    if(t != null) {
        inOrderTraversal(t.left);
        process(t.element);
        inOrderTraversal(t.right);
    }
}

A = current node
A = processing (on the call stack)
A = completed node
✓ = element has been processed
void inOrderTraversal(Node t) {
    if (t != null) {
        inOrderTraversal(t.left);
        process(t.element);
        inOrderTraversal(t.right);
    }
}

A = current node  \quad A = processing (on the call stack)
A = completed node  \quad ✓ = element has been processed
void inOrderTraversal(Node t) {
    if(t != null) {
        inOrderTraversal(t.left);
        process(t.element);
        inOrderTraversal(t.right);
    }
}

A = current node
A = processing (on the call stack)
A = completed node
✓ = element has been processed
void inOrderTraversal(Node t) {
    if (t != null) {
        inOrderTraversal(t.left);
        process(t.element);
        inOrderTraversal(t.right);
    }
}

A = current node
A = processing (on the call stack)
A = completed node ✓ = element has been processed
More on traversals

```java
void inOrderTraversal(Node t) {
    if (t != null) {
        inOrderTraversal(t.left);
        process(t.element);
        inOrderTraversal(t.right);
    }
}
```

- A = current node
- A = processing (on the call stack)
- A = completed node
- ✓ = element has been processed

D B
More on traversals

```java
void inOrderTraversal(Node t) {
    if (t != null) {
        inOrderTraversal(t.left);
        process(t.element);
        inOrderTraversal(t.right);
    }
}
```

A = current node  
A = processing (on the call stack)

A = completed node  
✓ = element has been processed

D B E
More on traversals

```java
void inOrderTraversal(Node t) {
    if (t != null) {
        inOrderTraversal(t.left);
        process(t.element);
        inOrderTraversal(t.right);
    }
}
```

A = current node  A = processing (on the call stack)
A = completed node ✓ = element has been processed

D B E A
void inOrderTraversal(Node t) {
    if (t != null) {
        inOrderTraversal(t.left);
        process(t.element);
        inOrderTraversal(t.right);
    }
}

A = current node  
A = processing (on the call stack) 
A = completed node  ✓ = element has been processed

D B E A
More on traversals

```java
void inOrderTraversal(Node t) {
    if (t != null) {
        inOrderTraversal(t.left);
        process(t.element);
        inOrderTraversal(t.right);
    }
}
```

A = current node  
B = processing (on the call stack)  
C = completed node  
✓ = element has been processed

DBECAF
More on traversals

void inOrderTraversal(Node t) {
  if (t != null) {
    inOrderTraversal(t.left);
    process(t.element);
    inOrderTraversal(t.right);
  }
}

A = current node  A = processing (on the call stack)
A = completed node ✓ = element has been processed

D B E A F C G
More on traversals

```java
void inOrderTraversal(Node t) {
    if (t != null) {
        inOrderTraversal(t.left);
        process(t.element);
        inOrderTraversal(t.right);
    }
}
```

Sometimes order doesn’t matter
  • Example: sum all elements
Sometimes order matters
  • Example: evaluate an expression tree
Binary Search Tree (BST) Data Structure

• Structure property (binary tree)
  – Each node has \( \leq 2 \) children
  – Result: keeps operations simple

• Order property
  – All keys in left subtree smaller than node’s key
  – All keys in right subtree larger than node’s key
  – Result: easy to find any given key

A binary search tree is a type of binary tree (but not all binary trees are binary search trees!)
Are these BSTs?

Activity!
Find in BST, Recursive

```
Data find(Key key, Node root){
    if(root == null)
        return null;
    if(key < root.key)
        return find(key,root.left);
    if(key > root.key)
        return find(key,root.right);
    return root.data;
}
```

What is the running time?

**Worst case** running time is $O(n)$.
- Happens if the tree is very lopsided (e.g. list)

![BST Diagram]
Find in BST, Iterative

```
Data find(Key key, Node root) {
    while (root != null && root.key != key) {
        if (key < root.key) {
            root = root.left;
        } else { // key > root.key
            root = root.right;
        }
    }
    if (root == null) {
        return null;
    }
    return root.data;
}
```

Worst case running time is $O(n)$.
- Happens if the tree is very lopsided (e.g. list)
**Bonus: Other BST “Finding” Operations**

- **FindMin**: Find *minimum* node
  - Left-most node

- **FindMax**: Find *maximum* node
  - Right-most node
Insert in BST

Again… worst case running time is $O(n)$, which may happen if the tree is not balanced.
Deletion in BST

Why might deletion be harder than insertion?
Removing an item may disrupt the tree structure!
Deletion in BST

• Basic idea: find the node to be removed, then “fix” the tree so that it is still a binary search tree

• Three potential cases to fix:
  – Node has no children (leaf)
  – Node has one child
  – Node has two children
Deletion – The Leaf Case

delete(17)
Deletion – The One Child Case

delete(15)
Deletion – The One Child Case

delete(15)
Deletion – The Two Child Case

delete(5)

What can we replace 5 with?
Deletion – The Two Child Case

Idea: Replace the deleted node with a value guaranteed to be between the two child subtrees

Options:
• successor minimum node from right subtree
  \texttt{findMin(node.right)}

• predecessor maximum node from left subtree
  \texttt{findMax(node.left)}

Now delete the original node containing successor or predecessor
Deletion: The Two Child Case (example)

delete(23)
Deletion: The Two Child Case (example)

delete(23)
Deletion: The Two Child Case (example)

delete(23)
Deletion: The Two Child Case (example)

delete(23)

Success! 😊
Lazy Deletion

• Lazy deletion can work well for a BST
  – Simpler
  – Can do “real deletions” later as a batch
  – Some inserts can just “undelete” a tree node

• But
  – Can waste space and slow down find operations
  – Make some operations more complicated:
    • e.g., findMin and findMax?
BuildTree for BST

• Let’s consider buildTree
  – Insert all, starting from an empty tree

• Insert keys 1, 2, 3, 4, 5, 6, 7, 8, 9 into an empty BST
  – If inserted in given order, what is the tree?
  – What big-O runtime for this kind of sorted input? $O(n^2)$
  – Not a happy place
  – Is inserting in the reverse order any better?
**BuildTree for BST**

- Insert keys 1, 2, 3, 4, 5, 6, 7, 8, 9 into an empty BST

- What we if could somehow re-arrange them
  - median first, then left median, right median, etc.
  - 5, 3, 7, 2, 1, 4, 8, 6, 9

  - What tree does that give us?

  - What big-O runtime?

  \[ O(n \log n), \text{ definitely better} \]

  - So the order the values come in is important!