Topic 07: BSTs (Overview)

- Structure
- Basic Operations
- Deletion
- Rotations
- Balanced Trees
Binary Search Trees: Basic Ops

- Remember: the BST invariant
  \[ \text{left} < \text{current node} < \text{right} \]
Binary Search Trees: Basic Ops

- Remember: the BST invariant
  
  \[ \text{left} < \text{current node} < \text{right} \]
Binary Search Trees: Basic Ops

- Remember: the BST invariant

\[ \text{left} < \text{current node} < \text{right} \]
Recursive

- Trees are a recursive data structure
  - Every subtree is a tree

- We will often define algorithms (or invariants) recursively
  - Perform the action by recursion
  - Apply the invariant to every node
How Large is a Tree?

• How large is a tree that stores $n$ nodes?
  - Best case: full, balanced
  - Worst case: linked list
How Large is a Tree?

• Why is a balanced tree better?
  – Recursion happens from the root
  – Many algos are $O(h)$
How Large is a Tree?

- What is the relationship between $n$ and $h$?

- Prove in class:
  
  A binary tree of height $h$ can store up to $2^{h+1} - 1$ nodes.
How Large is a Tree?

• What is the relationship between \( n \) and \( h \)?

• Prove in class:
  
  A binary tree of height \( h \) can store up to \( 2^{h+1} - 1 \) nodes.

• Therefore, in a perfect tree:
  
  \[ h = \Theta(lg n) \]
Topic 07: BSTs (Overview)

- Structure
- Basic Operations
- Deletion
- Rotations
- Balanced Trees
Binary Search Trees: Basic Ops

BSTNode search(BSTNode n, Key k) {
    if (n == null)
        return null;
    if (n.key == k)
        return n;
    if (k < n.key)
        return search(n.left, k);
    else
        return search(n.right, k);
}

Cost: \(O(h)\)
Binary Search Trees: Basic Ops

```java
BSTNode search(BSTNode n, Key k) {
    if (n == null)
        return null;
    if (n.key == k)
        return n;
    if (k < n.key)
        return search(n.left, k);
    else
        return search(n.right, k);
}
```

If we recurse into an empty subtree, then return null.
Binary Search Trees: Basic Ops

```java
BSTNode search(BSTNode n, Key k)
{
    if (n == null)
        return null;
    if (n.key == k)
        return n;
    if (k < n.key)
        return search(n.left, k);
    else
        return search(n.right, k);
}
```

If we find what we are looking for, then return it.
Binary Search Trees: Basic Ops

BSTNode search(BSTNode n, Key k)
{
    if (n == null)
        return null;
    if (n.key == k)
        return n;
    if (k < n.key)
        return search(n.left, k);
    else
        return search(n.right, k);
}

Otherwise, recurse.
Binary Search Trees: Basic Ops

```java
BSTNode search(BSTNode n, Key k) {
    if (n == null)
        return null;
    if (n.key == k)
        return n;
    if (k < n.key)
        return search(n.left, k);
    else
        return search(n.right, k);
}
```

We could transform this into a `while()` loop...but recursion is often easier to understand.
Binary Search Trees: Basic Ops

**BSTNode** `insert(BSTNode n, Key k)`

```java
{
    if (n == null)
        return new BSTNode(k);

    if (k < n.key)
        n.left = insert(n.left, k);
    else
        n.right = insert(n.right, k);
}
```

**Cost:** $O(h)$
Binary Search Trees: Basic Ops

```c
BSTNode min(BSTNode n)
{
    while (n.left != null)
    {
        n = n.left;
    }
    return n;
}
```

For something this simple, a loop works just as well as recursion.

Cost: $O(h)$
Binary Search Trees: Basic Ops

BSTNode insert(BSTNode n, Key k) {
    if (n == null)
        return new BSTNode(k);

    if (n.key < k)
        n.left = insert(n.left, k);
    else
        n.right = insert(n.right, k);
}

This code uses the “return the new subtree” design.

This simplifies the create-leaf case.

It will also make it easier to implement some of our tree-balancing algorithms.
Binary Search Trees: Basic Ops

```java
BSTNode successor(BSTNode n) {
    if (n.right != null)
        return min(n.right);

    BSTNode p = n.parent;
    while (p != null) {
        if (p.key > n.key)
            return p;
        else {
            p = p.parent;
        }
    }
    return null;
}
```

**Cost:**
\(O(h)\)

**In Class:**
Can you explain why this algorithm works?

And can you prove that the cost is correct?
Topic 07: BSTs (Overview)

- Structure
- Basic Operations
- Deletion
- Rotations
- Balanced Trees
Delete is Hard

• Can you write an algorithm that can delete any node from a BST?
Deletion Strategy

• To solve hard data structure problems, we often divide the problem into cases.
  – Like proof by cases

• What is the simplest deletion scenario?
Deletion: Case 1

- Case 1: The node to delete is a leaf
- Solution: Simply delete it
Deletion: Case 2

- Case 2: The node has a single child (subtree)
- Solution: Move the subtree up
Deletion: Case 3

- Case 3: The node has two children
- Solution: Swap with successor, then delete successor
Deletion: Case 3

Consider an in-order traversal of the nodes near the node to be deleted.

In-Order Traversal:
Deletion: Case 3

Swap node-to-delete and successor.
(This momentarily puts the tree out of order.)

In-Order Traversal:
Deletion: Case 3

Delete successor (which contains the value-to-be-deleted).

The order of the tree is restored.
Deletion: Case 3

- What if the successor is not a leaf???
- Do the same...but treat this (recursively) as Case 2.
Deletion: Case 3

Swap node-to-delete and successor.  
(This momentarily puts the tree out of order.)

In-Order Traversal:
Deletion: Case 3

Delete successor (which contains the value-to-be-deleted).

The order of the tree is restored.

In-Order Traversal:
Case 3 Recursion

• In Class:

Prove that the recursive deletion (in Case 3) is never another Case 3 deletion.

Prove that the recursive deletion (in Case 3) always happens in a descendant of the original node-to-delete.
Binary Search Trees: Basic Ops

Cost of Delete

- $O(h)$ to search
- $O(h)$ to find successor
- $O(1)$ to move successor
- $O(1)$ to delete successor
- $O(h)$ total
Topic 07: BSTs (Overview)

- Structure
- Basic Operations
- Deletion
- Rotations
- Balanced Trees
Rotations

• A “rotation” rewrites the structure of the tree without changing its nodes

• Very useful for rebalancing a tree
Rotations

These Trees are Equivalent

All we did was change which node was the root.
Rotations

Right Rotation in General

Assume that A, C are nodes.

B, D, E are node pointers. They might be null, leaves, or the root of large trees.
Rotations

An In-Order Traversal of the Tree:

E C D A B
Rotations

An In-Order Traversal of the Tree:

E   C   D   A   B
Remember!
This rotation probably happens deep within a larger tree.
We can convert an imbalanced tree into an optimal one.
Rotations

Using Rotations
We can convert an imbalanced tree into an optimal one.
Rotations

Using Rotations

We can convert an imbalanced tree into an optimal one.
Rotations

Using Rotations
We can convert an imbalanced tree into an optimal one.
Rotations

Using Rotations

We can convert an imbalanced tree into an optimal one.
Rotations

Using Rotations

We can convert an imbalanced tree into an optimal one.
Topic 07: BSTs (Overview)

- Structure
- Basic Operations
- Deletion
- Rotations
- Balanced Trees
Balanced Trees

- All of the major operations in binary trees are $O(h)$, where $h$ is the height.

- Ideally, a tree with $n$ nodes should have height $\lg n$ – thus all of the ops would be $O(\lg n)$.

- But worst case height is $O(n)$. 
Balanced Trees

• A variety of systems try to keep the tree balanced
  – Typically, minimize the height; prove $h = O(\lg n)$
  – A few more complex designs
Balanced Trees

Typical Concept

Start with a simple, small tree.
Balanced Trees

Typical Concept
Allow insertions to grow the tree.
Balanced Trees

Typical Concept
If one subtree gets much larger than the other, then rotate to rebalance.
Balanced Trees

Typical Concept
If one subtree gets much larger than the other, then rotate to rebalance.
Balanced Trees

Typical Concept

Usually, the balancing happens throughout the tree, at every node.

This subtree needs rebalancing.
Topic 07: BSTs (Overview)

- Structure
- Basic Operations
- Deletion
- Rotations
- Balanced Trees

Summary