Today’s Outline

- Announcements
  - Written HW #3 due next Friday, 1/29
  - Project 2A due next Monday, 2/1

- Today’s Topics:
  - Priority Queues
  - Dictionary ADT
  - Binary Search Trees

Today’s Outline

- Announcements
  - Written HW #3 due next Friday, 1/29
  - Project 2A due next Monday, 2/1

- Today’s Topics:
  - Priority Queues
  - Dictionary ADT
  - Binary Search Trees

Today’s Outline

- Announcements
  - Written HW #3 due next Friday, 1/29
  - Project 2A due next Monday, 2/1

- Today’s Topics:
  - Priority Queues
  - Dictionary ADT
  - Binary Search Trees

Today’s Outline

- Announcements
  - Written HW #3 due next Friday, 1/29
  - Project 2A due next Monday, 2/1

- Today’s Topics:
  - Priority Queues
  - Dictionary ADT
  - Binary Search Trees

Today’s Outline

- Announcements
  - Written HW #3 due next Friday, 1/29
  - Project 2A due next Monday, 2/1

- Today’s Topics:
  - Priority Queues
  - Dictionary ADT
  - Binary Search Trees

Today’s Outline

- Announcements
  - Written HW #3 due next Friday, 1/29
  - Project 2A due next Monday, 2/1

- Today’s Topics:
  - Priority Queues
  - Dictionary ADT
  - Binary Search Trees

Today’s Outline

- Announcements
  - Written HW #3 due next Friday, 1/29
  - Project 2A due next Monday, 2/1

- Today’s Topics:
  - Priority Queues
  - Dictionary ADT
  - Binary Search Trees

Today’s Outline

- Announcements
  - Written HW #3 due next Friday, 1/29
  - Project 2A due next Monday, 2/1

- Today’s Topics:
  - Priority Queues
  - Dictionary ADT
  - Binary Search Trees
Tree Calculations Example

How high is this tree?

More Recursive Tree Calculations: Tree Traversals

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

Three types:
- **Pre-order**: Root, left subtree, right subtree
- **In-order**: Left subtree, root, right subtree
- **Post-order**: Left subtree, right subtree, root

More Recursive Tree Calculations:

Tree Traversals

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

Three types:
- **Pre-order**: Root, left subtree, right subtree
- **In-order**: Left subtree, root, right subtree
- **Post-order**: Left subtree, right subtree, root

Traversals

```c
void traverse(BNode t){
    if (t != NULL)
        traverse (t.left);
    print t.element;
    traverse (t.right);
}
```

Which one is this?

Binary Trees

- Binary tree is
  - a root
  - left subtree (maybe empty)
  - right subtree (maybe empty)

- Representation:

```
  Data
  left  right
```

Binary Tree: Representation

Binary Tree: Special Cases

Complete Tree

Perfect Tree

Full Tree
Binary Tree: Some Numbers!

For binary tree of height $h$:
- max # of leaves:
- max # of nodes:
- min # of leaves:
- min # of nodes:

The Dictionary ADT

- Data:
  - a set of (key, value) pairs

- Operations:
  - Insert (key, value)
  - Find (key)
  - Remove (key)

The Dictionary ADT is sometimes called the "Map ADT"

The Dictionary ADT: A Modest Few Uses

Associates a key with a value
Main operations: Find, Insert, Delete

Examples:
- Networks: Router tables
- Operating systems: Page tables
- Compilers: Symbol tables

Probably the most widely used ADT!

Implementations

- Unsorted Linked-list
- Unsorted array
- Sorted array

Binary Search Tree Data Structure

- Structural property
  - each node has ≤ 2 children
  - result:
    - storage is small
    - operations are simple
    - average depth is small

- Order property
  - all keys in left subtree smaller than root’s key
  - all keys in right subtree larger than root’s key
  - result: easy to find any given key

Are these BSTs?
Find in BST, Recursive

```java
Node Find(Object key, Node root) {
    if (root == NULL) return NULL;
    if (key < root.key) return Find(key, root.left);
    else if (key > root.key) return Find(key, root.right);
    else return root;
}
```

Find in BST, Iterative

```java
Node Find(Object key, Node root) {
    while (root != NULL && root.key != key) {
        if (key < root.key) root = root.left;
        else root = root.right;
    }
    return root;
}
```

Insert in BST

```
Insert(13)
Insert(8)
Insert(31)
```

BuildTree for BST

- Suppose keys 1, 2, 3, 4, 5, 6, 7, 8, 9 are inserted into an initially empty BST.
  - Runtime depends on the order!
    - in given order
    - in reverse order
    - median first, then left median, right median, etc.

Bonus: FindMin/FindMax

- Find minimum
- Find maximum

Deletion in BST

Why might deletion be harder than insertion?
Lazy Deletion

Instead of physically deleting nodes, just mark them as deleted

- simpler
- physical deletions done in batches
- some adds just flip deleted flag

- extra memory for deleted flag
- many lazy deletions slow finds
- some operations may have to be modified (e.g., min and max)

Non-lazy Deletion

- Removing an item disrupts the tree structure.
- Basic idea: find the node that is to be removed. Then “fix” the tree so that it is still a binary search tree.
- Three cases:
  - node has no children (leaf node)
  - node has one child
  - node has two children

Non-lazy Deletion – The Leaf Case

Delete(17)

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:
- \textit{succ} from right subtree: \text{findMin}(t.\text{right})
- \textit{pred} from left subtree : \text{findMax}(t.\text{left})

Now delete the original node containing \textit{succ} or \textit{pred}
- Leaf or one child case – easy!
Finally…

Balanced BST

Observation
- BST: the shallower the better!
- For a BST with \( n \) nodes
  - Average height is \( O(\log n) \)
  - Worst case height is \( O(n) \)
- Simple cases such as insert(1, 2, 3, ..., \( n \)) lead to the worst case scenario

Solution: Require a Balance Condition that
1. ensures depth is \( O(\log n) \) — strong enough!
2. is easy to maintain — not too strong!

Potential Balance Conditions
1. Left and right subtrees of the root have equal number of nodes
2. Left and right subtrees of the root have equal height
3. Left and right subtrees of every node have equal number of nodes
4. Left and right subtrees of every node have equal height

The AVL Balance Condition
Left and right subtrees of every node have equal heights differing by at most 1

Define: \( \text{balance}(x) = \text{height}(x\text{.left}) - \text{height}(x\text{.right}) \)

AVL property: \( -1 \leq \text{balance}(x) \leq 1 \), for every node \( x \)
- Ensures small depth
  - Will prove this by showing that an AVL tree of height \( h \) must have a lot of (i.e. \( O(2^h) \)) nodes
- Easy to maintain
  - Using single and double rotations

The AVL Tree Data Structure

Structural properties
1. Binary tree property
2. Balance property: balance of every node is between -1 and 1
   Result: Worst case depth is \( O(\log n) \)

Ordering property
- Same as for BST
### Proving Shallowness Bound

Let $S(h)$ be the minimum number of nodes in an AVL tree of height $h$.

Claim: $S(h) = S(h-1) + S(h-2) + 1$

Solution of recurrence: $S(h) = O(2^h)$ (like Fibonacci numbers)

AVL tree of height $h=4$ with the minimum number of nodes.

### Testing the Balance Property

We need to be able to:
1. ...
2. ...
3. ...

NULLs have height $-1$.

### An AVL Tree

AVL tree with data, height, and children.