Where we are

Studying the absolutely essential ADTs of computer science and classic data structures for implementing them

ADTs so far:
1. Stack: push, pop, isEmpty, ...
2. Queue: enqueue, dequeue, isEmpty, ...
3. Priority queue: insert, deleteMin, ...

Next:
   – Probably the most common, way more than priority queue

The Dictionary (a.k.a. Map) ADT

• Data:
  − set of (key, value) pairs
  − keys must be comparable
  − insert(djg, Dan Grossman, …)

• Operations:
  − insert(key, value)
  − find(key)
  − delete(key)
  − …

find(trobison)
  − Tyler Robison, …

Will tend to emphasize the keys; don’t forget about the stored values

Comparison: The Set ADT

The Set ADT is like a Dictionary without any values
− A key is present or not (no repeats)

For find, insert, delete, there is little difference
− In dictionary, values are “just along for the ride”
− So same data-structure ideas work for dictionaries and sets

But if your Set ADT has other important operations this may not hold
− union, intersection, is_subset
− Notice these are binary operators on sets

Dictionary data structures

Will spend the next several lectures implementing dictionaries with three different data structures

1. AVL trees
   − Binary search trees with guaranteed balancing

2. B-Trees
   − Also always balanced, but different and shallower
   − B!=Binary; B-Trees generally have large branching factor

3. Hashtables
   − Not tree-like at all

Skipping: Other balanced trees (e.g., red-black, splay)

But first some applications and less efficient implementations…

A Modest Few Uses

Any time you want to store information according to some key and be able to retrieve it efficiently
− Lots of programs do that!

• Networks: router tables
• Operating systems: page tables
• Compilers: symbol tables
• Databases: dictionaries with other nice properties
• Search: inverted indexes, phone directories, …
• Biology: genome maps
• …
Simple implementations

For dictionary with $n$ key/value pairs

- Unsorted linked list
- Unsorted array
- Sorted linked list
- Sorted array

We’ll see a Binary Search Tree (BST) probably does better, but not in the worst case unless we keep it balanced

Lazy Deletion

A general technique for making delete as fast as find:
- Instead of actually removing the item just mark it deleted

Plusses:
- Simpler
- Can do removals later in batches
- If re-added soon thereafter, just unmark the deletion

Minuses:
- Extra space for the “is-it-deleted” flag
- Data structure full of deleted nodes wastes space
- find $O(\log m)$ time where $m$ is data-structure size (okay)
- May complicate other operations

Some tree terms (mostly review)

- There are many kinds of trees
  - Every binary tree is a tree
  - Every list is kind of a tree (think of “next” as the one child)
- There are many kinds of binary trees
  - Every binary min heap is a binary tree
  - Every binary search tree is a binary tree
- A tree can be balanced or not
  - A balanced tree with $n$ nodes has a height of $O(\log n)$
  - Different tree data structures have different “balance conditions” to achieve this

Binary Trees

- Binary tree is empty or
  - A root (with data)
  - A left subtree (may be empty)
  - A right subtree (may be empty)
- Representation:
- For a dictionary, data will include a key and a value

Binary Trees: Some Numbers

Recall: height of a tree = longest path from root to leaf (count edges)

For binary tree of height $h$:
- max # of leaves:
- max # of nodes:
- min # of leaves:
- min # of nodes:
**Binary Trees: Some Numbers**

Recall: height of a tree = longest path from root to leaf (count edges)

For binary tree of height \( h \):
- max # of leaves: \( 2^h \)
- max # of nodes: \( 2^{(h+1)} - 1 \)
- min # of leaves: 1
- min # of nodes: \( h + 1 \)

For \( n \) nodes, we cannot do better than \( O(\log n) \) height, and we want to avoid \( O(n) \) height

---

**Calculating height**

What is the height of a tree with root \( root \)?

```java
int treeHeight(Node root) {
  if(root == null)
    return -1;
  return 1 + max(treeHeight(root.left),
                  treeHeight(root.right));
}
```

---

**Tree Traversals**

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

- **Pre-order**: root, left subtree, right subtree
- **In-order**: left subtree, root, right subtree
- **Post-order**: left subtree, right subtree, root

(an expression tree)

---

**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: print tree with parent above indented children (pre-order)
- Example: evaluate an expression tree (post-order)
Binary Search Tree

- Structure property ("binary")
  - Each node has ≤ 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

Are these BSTs?

Find in BST, Recursive

```java
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;
}
```

Find in BST, Iterative

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

Other “Finding” Operations

- Find minimum node
  - “the liberal algorithm”
- Find maximum node
  - “the conservative algorithm”
- Find predecessor of a non-leaf
- Find successor of a non-leaf
- Find predecessor of a leaf
- Find successor of a leaf
**Insert in BST**

- Insert(13)
- Insert(8)
- Insert(31)

(New) insertions happen only at leaves – easy!

**Deletion in BST**

Why might deletion be harder than insertion?

---

**Deletion**

- Removing an item disrupts the tree structure
- Basic idea: find the node to be removed, then “fix” the tree so that it is still a binary search tree
- Three cases:
  - 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)

What can we replace 5 with?

**Deletion – The Two Child Case**

- delete(5)
**Deletion – The Two Child Case**

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

Options:
- **successor** from right subtree: \( \text{findMin}(\text{node.right}) \)
- **predecessor** from left subtree: \( \text{findMax}(\text{node.left}) \)
  - These are the easy cases of predecessor/successor

Now delete the original node containing successor or predecessor
- Leaf or one child case – easy cases of delete!

---

**BuildTree for BST**

- We had buildHeap, so let's consider buildTree
- 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?
  - Is inserting in the reverse order any better?

**Unbalanced BST**

- Balancing a tree at build time is insufficient, as sequences of operations can eventually transform that carefully balanced tree into the dreaded list
- At that point, everything is \( O(n) \) and nobody is happy
  - find
  - insert
  - delete

**Balanced BST**

**Observation**
- BST: the shallower the better!
- For a BST with \( n \) nodes inserted in arbitrary order
  - Average height is \( O(\log n) \) – see text for proof
  - Worst case height is \( O(n) \)
- Simple cases, such as inserting in key order, lead to the worst-case scenario

**Solution:** Require a **Balance Condition** that
1. Ensures depth is always \( O(\log n) \) – strong enough!
2. Is efficient to maintain – not too strong!

**Potential Balance Conditions**

1. Left and right subtrees of the root have equal number of nodes
   - Too weak!
   - Height mismatch example:
2. Left and right subtrees of the root have equal height
   - Too weak!
   - Double chain example:
Potential Balance Conditions

3. Left and right subtrees of every node have equal number of nodes
   - Too strong!
   - Only perfect trees \((2^n - 1)\) nodes

4. Left and right subtrees of every node have equal height
   - Too strong!
   - Only perfect trees \((2^n - 1)\) nodes

The AVL Balance Condition

Left and right subtrees of every node have heights differing by at most 1

**Definition:** balance\((node)\) = height\((node.left)\) – height\((node.right)\)

**AVL property:** for every node \(x\), \(-1 \leq \text{balance}(x) \leq 1\)

- Ensures small depth
  - Will prove this by showing that an AVL tree of height \(h\) must have a number of nodes exponential in \(h\)

- Efficient to maintain
  - Using single and double rotations