CSE 332: Data Abstractions
Lecture 6: Dictionaries; Binary Search Trees
Ruth Anderson
Winter 2013

Announcements
• Project 1 – phase B due Tues Jan 22nd 11pm via catalyst
• Homework 1 – due NOW!!
• Homework 2 – due Friday Jan 25th at beginning of class
• No class on Monday Jan 21st

Today
• Dictionaries
• Trees

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

Operations:
• insert(key, val):
  – places (key, val) in map
  (If key already used, overwrites existing entry)
• find(key):
  – returns val associated with key
• delete(key)

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

Comparison: Set ADT vs. Dictionary 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
  • Java HashSet implemented using a HashMap, for instance
Set ADT may have other important operations
– union, intersection, is_subset, etc.
– Notice these are binary operators on sets
– We will want different data structures to implement these operators

We will tend to emphasize the keys, but don’t forget about the stored values!
A Modest Few Uses for Dictionaries

Any time you want to store information according to some key and be able to retrieve it efficiently – a dictionary is the ADT to use!

– 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

\[ \begin{array}{ccc}
\text{insert} & \text{find} & \text{delete} \\
\text{Unsorted linked-list} & O(1) & O(n) & O(n) \\
\text{Unsorted array} & O(1) & O(n) & O(n) \\
\text{Sorted linked list} & O(n) & O(n) & O(n) \\
\text{Sorted array} & O(n) & O(\log n) & O(n) \\
\end{array} \]

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

*Note: If we do not allow duplicates values to be inserted, we would need to do \( O(n) \) work to check for a key’s existence before insertion

Lazy Deletion (e.g. in a sorted array)

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 \( \mathcal{O}(\log m) \) time where \( m \) is data-structure size (\( m \gg n \))
– May complicate other operations

Better Dictionary data structures

Will spend the next several lectures looking at 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 (red-black, splay)

Why Trees?

Trees offer speed ups because of their branching factors

• Binary Search Trees are structured forms of binary search
**Binary Search**

`find(4)`

```
  1 3 4 5 7 8 9 10
```

**Binary Search Tree**

Our goal is the performance of binary search in a tree representation

```
  1 3 4 5 7 8 9 10
```

**Why Trees?**

Trees offer speed ups because of their branching factors
- Binary Search Trees are structured forms of **binary search**

Even a basic BST is fairly good

<table>
<thead>
<tr>
<th></th>
<th>Insert</th>
<th>Find</th>
<th>Delete</th>
</tr>
</thead>
<tbody>
<tr>
<td>Worse-Case</td>
<td>O(n)</td>
<td>O(n)</td>
<td>O(n)</td>
</tr>
<tr>
<td>Average-Case</td>
<td>O(log n)</td>
<td>O(log n)</td>
<td>O(log n)</td>
</tr>
</tbody>
</table>

**Binary Trees**

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

- Representation:

  ![Binary Tree Diagram](https://via.placeholder.com/150)

  - Data
  - left pointer
  - right pointer

- For a dictionary, data will include a key and a value

**Binary Tree: Some Numbers**

Recall: height of a tree = longest path from root to leaf (count # of 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) {
  ???
}
```

Running time for tree with \( n \) nodes: \( O(n) \) – single pass over tree

Note: non-recursive is painful – need your own stack of pending nodes; much easier to use recursion’s call stack

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

\[ \text{(an expression tree)} \]

More on traversals

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

- **Structural property** ("binary")
  - 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
Are these BSTs?

3 5 4 1 8 4 5 4
1 7 11 18 10 6 11 5
8 20 15 21

Yes No

Find in BST, Recursive

\[
\text{Data \ find(\text{Key key, Node root})} =
\begin{cases}
\text{return null;}
& \text{if (root == null)}
\text{return find(key, root.left);} & \text{if (key < root.key)}
\text{return find(key, root.right);} & \text{if (key > root.key)}
\end{cases}
\]

Find in BST, Iterative

\[
\text{Data \ find(\text{Key key, Node root})} =
\begin{cases}
\text{while (root != null && root.key != key)} {
\text{return find(key, root.left);} & \text{if (key < root.key)}
\text{return find(key, root.right);} & \text{if (key > root.key)}
root = root.right;
\text{if (root == null)}
\text{return null;}
\text{return root.data;}
\end{cases}
\]

Other “finding operations”

- Find minimum node
- Find maximum node
- 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

\[
\text{insert(13)}
\text{insert(8)}
\text{insert(31)}
\]

(New) insertions happen only at leaves – easy!

1. Find
2. Create a new node
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,
  - Remove it
  - “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)

Deletion – The Two Child Case

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 from right subtree: findMin(node.right)
- predecessor from left subtree: findMax(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!
Delete Using Successor

```
findMin(right sub tree) → 7
delete(5)
```

Delete Using Predecessor

```
findMax(left sub tree) → 2
delete(5)
```

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?

```
1
2
3
```

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? \( O(n^2) \)
  - Is inserting in the reverse order any better?

```
8
4
2
7
3
5
9
6
1
```

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?

```
5
3
4
2
6
7
9
1
```

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? \( O(n \log n) \), definitely better
Give up on BuildTree for BST

The median trick will guarantee a $O(n \log n)$ build time, but it is not worth the effort.

Why?
- Subsequent inserts and deletes will eventually transform the carefully balanced tree into the dreaded list
- Then everything will have the $O(n)$ performance of a linked list

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 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

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

Potential Balance Conditions

3. Left and right subtrees of every node have equal number of nodes
4. Left and right subtrees of every node have equal height

Potential Balance Conditions

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 heights differing by at most 1.

**Definition:** \( \text{balance}(\text{node}) = \text{height}(\text{node}.\text{left}) - \text{height}(\text{node}.\text{right}) \)

**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 \( h \)

- Easy (well, efficient) to maintain
  - Using single and double rotations

1/18/2013