CSE373: Data Structures and Algorithms
Dictionaries and Trees
Steve Tanimoto
Autumn 2016

This lecture material represents the work of multiple instructors at the University of Washington. Thank you to all who have contributed!

Let’s take a breath

- So far we’ve covered
  - Some simple ADTs: stacks, queues, lists
  - Some math (proof by induction)
  - How to analyze algorithms
  - Asymptotic notation (Big-O)

- Coming up…
  - Many more ADTs
    - Starting with dictionaries

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

- Data:
  - set of (key, value) pairs
  - keys must be comparable
- Operations:
  - insert(key, value)
  - find(key)
  - delete(key)

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

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!
- Search: inverted indexes, phone directories, ...
- Networks: router tables
- Operating systems: page tables
- Compilers: symbol tables
- Databases: dictionaries with other nice properties
- Biology: genome maps
- ...

Possibly the most widely used ADT

Simple implementations

For dictionary with n key/value pairs

- insert O(1) *
- find O(n)
- delete O(n)

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

* Unless we need to check for duplicates

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
- May complicate other operations
There are many good data structures for (large) dictionaries:

1. Binary trees
   - Binary search trees with guaranteed balancing

2. AVL trees
   - Also always balanced, but different and shallower
   - B-Trees are not the same as Binary Trees
     - B-Trees generally have large branching factor

3. B-Trees
   - Also always balanced, but different and shallower
   - B-Trees are not the same as Binary Trees

4. Hash Tables
   - Not tree-like at all

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

- Tree terms
  - Root (tree)
  - Leaf (tree)
  - Children (node)
  - Parent (node)
  - Siblings (node)
  - Ancestors (node)
  - Descendants (node)
  - Subtree (node)
  - Depth
  - Height
  - Degree
  - Branching factor

- Kinds of trees
  - Binary tree: Each node has at most 2 children (branching factor 2)
  - n-ary tree: Each node has at most n children (branching factor n)
  - Perfect tree: Each row completely full
  - Complete tree: Each row completely full except the bottom row, which is filled from left to right

What is the height of a perfect binary tree with n nodes? $\log_2 n$
**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 \( \text{root} \)?

```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 the system’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

* (an expression tree)

---

**More on traversals**

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

- \( \text{A} \) = current node
- \( \text{B} \) = processing (on the call stack)
- \( \text{C} \) = completed node
- \( \checkmark \) = element has been processed
More on traversals

```c
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
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)
A = completed node  ✓ = element has been processed

More on traversals

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

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

Preorder Exercise

```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)
A = completed node  ✓ = element has been processed