CSE373: Data Structures & Algorithms
Lecture 5: Dictionary ADTs; Binary Trees

Nicki Dell
Spring 2014
Today’s Outline

Announcements
- Homework 1 due TODAY at 11pm 😊
- Homework 2 out
  - Due in class Wednesday April 16th at the START of class
  - No late days!

Today’s Topics
• Finish Asymptotic Analysis
• Dictionary ADT (a.k.a. Map): associate keys with values
  – Extremely common
• Binary Trees
Summary of Asymptotic Analysis

Analysis can be about:

• The problem or the algorithm (usually algorithm)
• Time or space (usually time)
  – Or power or dollars or …
• Best-, worst-, or average-case (usually worst)
• Upper-, lower-, or tight-bound (usually upper)

• The most common thing we will do is give an $O$ upper bound to the worst-case running time of an algorithm.
Big-Oh Caveats

• Asymptotic complexity focuses on behavior for large $n$ and is independent of any computer / coding trick

• But you can “abuse” it to be misled about trade-offs

• Example: $n^{1/10}$ vs. $\log n$
  – Asymptotically $n^{1/10}$ grows more quickly
  – But the “cross-over” point is around $5 \times 10^{17}$
  – So if you have input size less than $2^{58}$, prefer $n^{1/10}$

• For small $n$, an algorithm with worse asymptotic complexity might be faster
  – If you care about performance for small $n$ then the constant factors can matter
Addendum: Timing vs. Big-Oh Summary

• Big-oh is an essential part of computer science’s mathematical foundation
  – Examine the algorithm itself, not the implementation
  – Reason about (even prove) performance as a function of $n$

• Timing also has its place
  – Compare implementations
  – Focus on data sets you care about (versus worst case)
  – Determine what the constant factors “really are”
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-Oh)

• 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

<table>
<thead>
<tr>
<th></th>
<th>insert</th>
<th>find</th>
<th>delete</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unsorted linked-list</td>
<td>$O(1)^*$</td>
<td>$O(n)$</td>
<td>$O(n)$</td>
</tr>
<tr>
<td>Unsorted array</td>
<td>$O(1)^*$</td>
<td>$O(n)$</td>
<td>$O(n)$</td>
</tr>
<tr>
<td>Sorted linked list</td>
<td>$O(n)$</td>
<td>$O(n)$</td>
<td>$O(n)$</td>
</tr>
<tr>
<td>Sorted array</td>
<td>$O(n)$</td>
<td>$O(\log n)$</td>
<td>$O(n)$</td>
</tr>
</tbody>
</table>

* 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
Better dictionary data structures

There are many good data structures for (large) dictionaries

1. Binary trees
2. AVL trees
   - Binary search trees with guaranteed balancing
3. B-Trees
   - Also always balanced, but different and shallower
   - B-Trees are not the same as Binary Trees
     • B-Trees generally have large branching factor
4. Hashtables
   - Not tree-like at all

Skipping: Other balanced trees (e.g., red-black, splay)
Tree terms (review?)

- **Root** (tree)
- **Leaves** (tree)
- **Children** (node)
- **Parent** (node)
- **Siblings** (node)
- **Ancestors** (node)
- **Descendants** (node)
- **Subtree** (node)

**Depth** (node)

**Height** (tree)

**Degree** (node)

**Branching factor** (tree)
More tree terms

• 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 search tree is a binary tree
  – Later: A binary heap is a different kind of 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
Kinds of trees

Certain terms define trees with specific structure

- **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 maybe the bottom row, which is filled from left to right

What is the height of a perfect binary tree with $n$ nodes? A complete binary tree?
**Binary Trees**

- **Binary tree**: Each node has at most 2 children (branching factor 2)

- Binary tree is
  - A root (*with data*)
  - A left subtree (*may be empty*)
  - A right subtree (*may be empty*)

- Representation:

<table>
<thead>
<tr>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>left pointer</td>
</tr>
</tbody>
</table>

- For a dictionary, data will include a key and a value
Binary Tree Representation
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) {
    ???
}
```
Calculating height

What is the height of a tree with root $\text{root}$?

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

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
  
  \[+ \ast 2 4 5\]

- **In-order**: left subtree, root, right subtree
  
  \[2 \ast 4 + 5\]

- **Post-order**: left subtree, right subtree, root
  
  \[2 4 \ast 5 +\]

(an expression tree)
More on traversals

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

- **A** = current node
- **A** = processing (on the call stack)
- **A** = completed node
- ✓ = element has been processed
void inOrderTraversal(Node t) {
  if (t != null) {
    inOrderTraversal(t.left);
    process(t.element);
    inOrderTraversal(t.right);
  }
}

= current node    = processing (on the call stack)
= completed node  ✓ = element has been processed
void \texttt{inOrderTraversal} (\texttt{Node t}) \{ 
    \textbf{if} (t \neq \texttt{null}) \{ 
        \texttt{inOrderTraversal}(t.\texttt{left}); 
        \texttt{process}(t.\texttt{element}); 
        \texttt{inOrderTraversal}(t.\texttt{right}); 
    \}
\}

\begin{itemize}
  \item[A] = current node
  \item[A] = processing (on the call stack)
  \item[A] = completed node
  \item[✓] = element has been processed
\end{itemize}
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)
- 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  A = processing (on the call stack)

A = 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
- **A** = processing (on the call stack)
- **A** = completed node
- ✓ = element has been processed
More on traversals

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

- A = current node
- A = processing (on the call stack)
- A = 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);
    }
}
```

= current node  = processing (on the call stack)
= 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
- **A** = processing (on the call stack)
- **A** = completed node
- ✓ = element has been processed
void inOrderTraversal(Node t)
{
    if(t != null) {
        inOrderTraversal(t.left);
        process(t.element);
        inOrderTraversal(t.right);
    }
}

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