CSE373: Data Structures & Algorithms
Lecture 5: Dictionaries; Binary Search Trees

Aaron Bauer
Winter 2014
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, ...

Next:

3. Dictionary (a.k.a. Map): associate keys with values
   – Extremely common
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*
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

There are many good data structures for (large) dictionaries

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!

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

For dictionary with $n$ key/value pairs

- Unsorted linked-list
- Unsorted array
- Sorted linked list
- Sorted array
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
– find $O(\log m)$ time where $m$ is data-structure size (okay)
– May complicate other operations
Tree terms (review?)

- root(tree)
- leaves(tree)
- children(node)
- parent(node)
- siblings(node)
- ancestors(node)
- descendents(node)
- subtree(node)

- depth(node)
- height(tree)
- degree(node)
- branching factor(tree)
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 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 is empty or
  - 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 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:
- 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: 
- 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:
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 \( \text{root} \)?

```c
int treeHeight(Node root) {
    ???
}
```
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));
}
```

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

(an expression tree)
Tree Traversals

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

- **Pre-order**: root, left subtree, right subtree
  
  + * 2 4 5

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

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

(an expression tree)
Tree Traversals

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

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

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

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

(an expression tree)
Tree Traversals

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

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

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

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

(an expression tree)
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
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);
    }
}
```

= 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
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

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

- A = current node
- = 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

```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