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)`
  - …

insert(dbutler1, ….) …

find(adwin555)
Adwin Jahn, …

- dbutler1
  Dan Butler
  OH: Thurs 1:30-2:30
  …
- efgan
  Emilia Gan
  OH: Wed 11:00-12:00
  …
- adwin555
  Adwin Jahn
  OH: Fri 9:00-10:00
  …

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

A Modest Few Uses

key	attr1	attr2	attr3
k1	v11	v12	v13
k2	v21	v22	v23

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`	`find`	`delete`
• Unsorted linked-list	$O(1)$*	$O(n)$	$O(n)$
• Unsorted array	$O(1)$*	$O(n)$	$O(n)$
• Sorted linked list	$O(n)$	$O(n)$	$O(n)$
• Sorted array	$O(n)$	$O(\log n)$	$O(n)$

* 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

10	12	24	30	41	42	44	45	50
✓	✗	✓	✓	✓	✓	✗	✓	✓

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

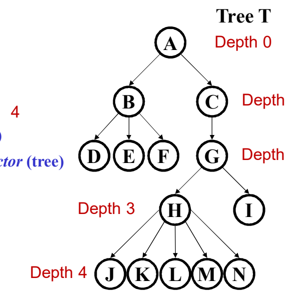
1. Binary trees
2. AVL trees
   - Binary search trees with *guaranteed balancing*
3. B-Trees
   - Also always balanced, but different and shallower
   - B-Trees are not the same as Binary Trees
     - B-Trees generally have large branching factor
4. Hash Tables
   - Not tree-like at all

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

Tree terms (review?)

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

Depth **(node)**
Height **(tree)** 4
Degree **(node)**
Branching factor **(tree)**



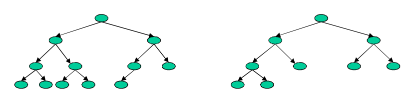
More tree terms

- 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

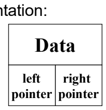
- Binary tree: Each node has at most 2 children (branching factor 2)
- *n*-ary tree: Each node has at most *n* children (branching factor *n*)
- Perfect tree: Each row completely full
- Complete tree: Each row completely full except maybe the bottom row, which is filled from left to right

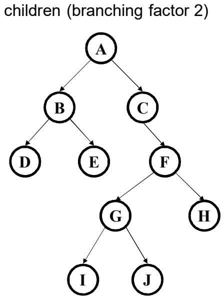


What is the height of a perfect binary tree with n nodes? $\lfloor \log_2 n \rfloor$
A complete binary tree?

Binary Trees

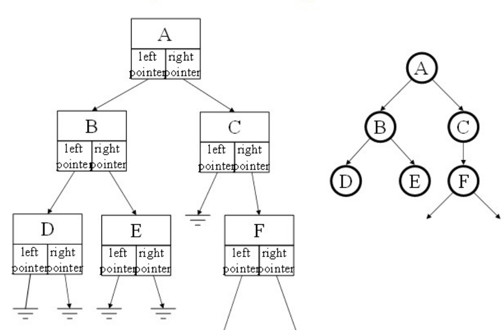
- Binary tree: Each node has at most 2 children (branching factor 2)
- Binary tree is
  - A root *(with data)*
  - A left subtree that's a binary tree
  - A right subtree that's a binary tree
- *These subtrees may be empty.*
- Representation:





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

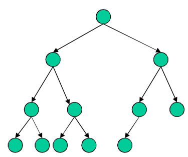
Binary Tree Representation

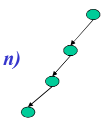


Binary Trees: Some Numbers

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: $h + 1$





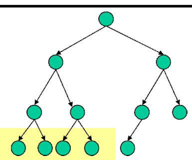
For n nodes, we cannot do better than O(`log` n) height and we want to avoid O(n) height

Calculating height

What is the height of a tree with root `root`?

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


Calculating height



What is the height of a tree with root `root`?

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

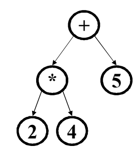
Running time for tree with *n* nodes: *O*(*n*) – single pass over tree

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

Tree Traversals

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

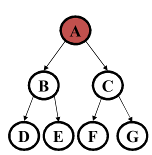
- *Pre-order*: root, left subtree, right subtree
  + * 2 4 5
- *In-order*: left subtree, root, right subtree
  2 * 4 + 5
- *Post-order*: left subtree, right subtree, root
  2 4 * 5 +



(an expression tree)

More on traversals

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

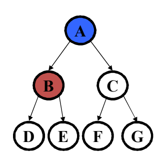


A = current node A = processing (on the call stack)

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);
  }
}
```

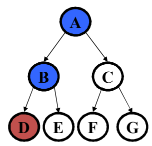


A = current node A = processing (on the call stack)

A = completed node ✓ = element has been processed

More on traversals

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

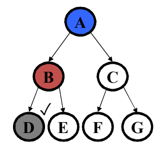


A = current node A = processing (on the call stack)

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);
  }
}
```

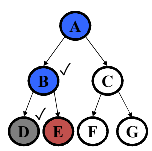


A = current node A = processing (on the call stack)

A = completed node √ = element has been processed

More on traversals

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

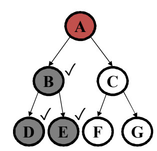


A = current node A = processing (on the call stack)

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);
  }
}
```

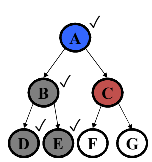


A = current node A = processing (on the call stack)

A = completed node √ = element has been processed

More on traversals

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

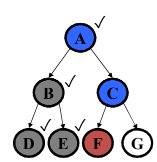


A = current node A = processing (on the call stack)

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);
  }
}
```

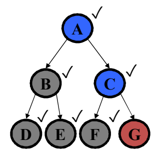


A = current node A = processing (on the call stack)

A = completed node √ = element has been processed

More on traversals

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

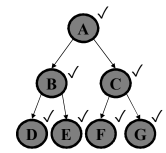


A = current node A = processing (on the call stack)

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);
  }
}
```

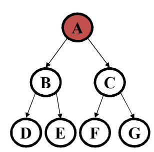


A = current node A = processing (on the call stack)

A = completed node ✓ = element has been processed

More on traversals

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



A = current node A = processing (on the call stack)

A = completed node ✓ = element has been processed

Preorder Exercise