CSE 373: Trees

book chapter 4

Pete Morcos
University of Washington
4/5/00

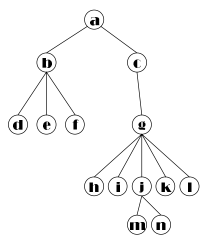
http://www.cs.washington.edu/education/courses/cse373/00sp

Why Trees?

- Lists (Queues, Stacks, arrays, etc.) represent a linear sequence
- Some data doesn't have a single linear ordering
  - Moves in a game
  - Organizational charts
  - Family trees
  - Classification hierarchies (e.g. genus/species)
  - File directories

Terms

– Root	a
– Leaf	d e f h i m n
– Parent	g -> c
– Children	g -> h i j k l
– Ancestors	g -> c a
– Descendants	g -> h i j k l m n
– Siblings	e -> d f
– "Cousins"	e -> g
– Path	c-m -> c g j m
– Depth	i -> 3
– Height	max of depths



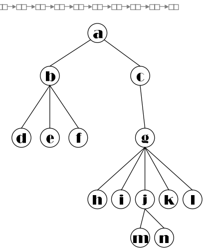
- Forest: one or more trees

More Tidbits

- Recursive definition: a tree is
  - an empty set (of nodes), or
  - a root with zero or more subtrees
- A tree with N nodes always has N-1 edges
- Edges are directed (parent -> child), but we often imply the direction by drawing parent higher up
- Two nodes have at most one path between them
- Sometimes we only put data in leaf nodes; interior nodes just there for organization
  - Leaf nodes probably a different type from interior nodes
  - Otherwise, leaf nodes are just nodes with no children (all NULL)

Tree Traversal

- Postorder: children, then root
  - d e f b h i m n j k l g c a
- Preorder: root, then children
  - a b d e f c g h i j m n k l
- Inorder: child, root, child
  - Only really makes sense for binary trees

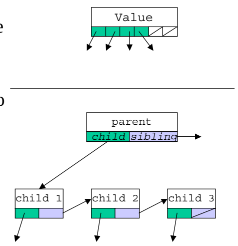


ADT Operations

- These are very generic. If we specify more details on the tree's behavior, we can come up with a more useful set.
- Tree as a whole:
  - GetRoot
  - Find
  - MakeEmpty
- Ops on a node, much like the ops on a list
  - AddChild/RemoveChild
  - NextChild/PrevChild
- Can make up more . . .

Implementing Trees

- If there is a (small) max no. of children in each node, we can use an array (or individual variables) to store child pointers
- If the number is unbounded, or so large an array would be wasteful, we need to implement some kind of growable list of children
  - Linked list in parent
  - Sibling pointers



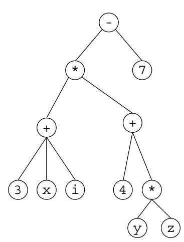
Parent Pointers

- As we discussed with doubly linked lists, if each node contains a pointer to its parent, some ops may get easier
  - Not always needed
- And, as with circularly linked lists, you might consider pointers to the root in each node
  - Unusual

Application: Expression Trees

`(3 + x + i)*(4 + y * z) - 7`

- Used in most compilers
- No parentheses needed; tree hierarchy shows structure
- Almost always strictly binary, unlike example here
- Packages data nicely for manipulation
  - e.g., if we know values of y and z, can simplify the "*" node in lower right to a constant

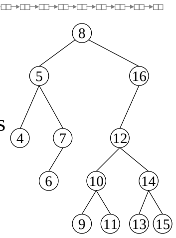


Binary Trees

- Max of two children per node; very common in computer science
- Minimum depth: approx. log N
- Maximum depth: N-1
  - Basically a linked list
- Our hope is to keep the depth well below O(N), so that we can make operations asymptotically more efficient than they would be with a list

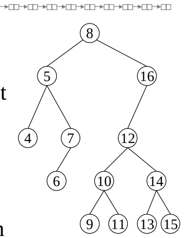
Binary Search Trees

- Value in every node is:
  - *greater* than all nodes in left subtree
  - *less* than all nodes in right subtree
- Duplicate values complicate things
- Operations:
  - Find, FindMin, FindMax
  - Insert, Remove
  - traversal (maybe)



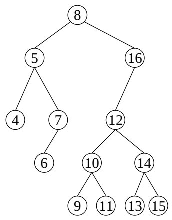
Insert

- (assuming no duplicates)
- Do same steps as a Find
- Will eventually stop when we hit a NULL pointer
- That's where it needs to go!
- Never tries to add a 3rd child—why?
- Consider inserting 7.5, 8.5, 20 in example



Remove

- More icky
- Easy if node has 0 or 1 children
- Removing interior node might leave 3 children (e.g. remove 5)
- Correct replacement usually not either child
  - Want largest in left subtree or smallest in right subtree
- Removing that one is always easy
  - Why?

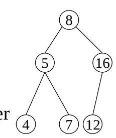


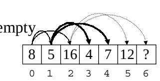
Lazy Deletion

- A "lazy" operation puts off the work as long as possible, usually in the hope that a future step will make the step unnecessary
- We can just mark removed nodes instead of actually reorganizing the tree
  - Skip them during insert/searches
  - Typically do the work when real nodes fall below a certain percentage
  - If tree is 50% deleted nodes, what is the extra cost of operations?
- Could also do this for lists
- To get the best benefit, modify Insert to reuse the marked nodes when possible

Array Implementation (?)

- We said at start that trees are for non-linear data
- There is a trick, often used with complete binary trees
  - A *complete* tree is one that has no gaps when you read the nodes left-to-right, top-to-bottom
- Use that left-to-right scan to impose a linear order on the nodes
- Simple formulas allow us to map between the two
  - Children of A[i] are A[2i + 1], A[2i + 2]
  - Obviously, need some way to tell when a cell is empty
- When applicable, a very efficient method
  - Very inefficient for non-complete trees. Why?





BST Analysis

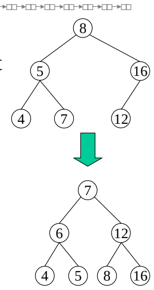
- Most ops are O(d), where d is tree depth
- Recall that log N <= d < N

operation	best	worst	avg
find			
insert			
remove			
build tree (N inserts)			

- Quite a spread...not as good as we'd hoped

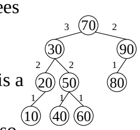
Balance

- The problem is that BSTs can get *unbalanced*, i.e. the depths of the left and right subtrees vary by a lot
- Many clever algorithms exist for maintaining balance
- Perfect balance too restrictive
  - Almost no flexibility in placement
  - Consider inserting 6 in example



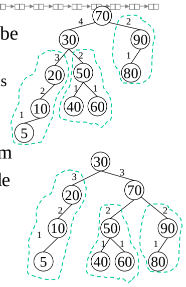
AVL Trees

- For every node, heights of left and right subtrees can differ by no more than 1
  - For efficiency, store current heights in each node
- Height will then be more or less log N (proof is a bit hairy, so we'll skip it)
- Some operations remain the same (e.g. Find), so now worst case is O(log N)
- Some ops must change, however, mainly Insert
  - Book glosses over Remove by assuming lazy deletion; we'll do same



Rotation

- An insert may cause the AVL property to be violated
  - After insert, walk back up tree updating heights
  - Stop if we hit a problem node (difference > 1)
- Since we added only 1 node, the heights will differ by exactly 2 if there is a problem
- Rotate around the deepest unbalanced node
  - Shift up the too-deep subtree
  - Shift down the too-shallow subtree
  - Fixup pointers to stay binary



Double Rotation

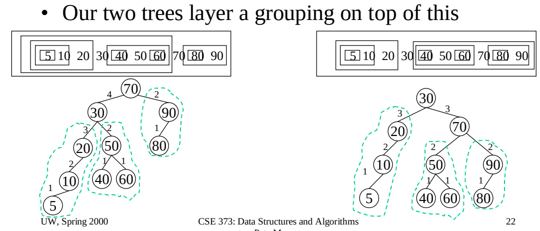
- A single rotation is enough to fix the tree when the too-deep subtree is the left-left or right-right grandchild of the unbalanced node
- If the left-right or right-left grandchild is the problem, this won't help (consider adding 65 in example)
- A double rotation splits up the too-deep subtree (great-grandchildren) and separates the halves
  - Book has good pictures, not repeated here

Why can we do all this?

- We're doing a lot of rearranging here. Why is it OK to mess with the data like this?
  - For example, a sorted linked list can't be rearranged...
- Need to distinguish between two types of structure
  - Inherent in data
    - numerical ordering, hierarchies, etc.
  - Extra imposed by choice of data structure
    - binary tree structure layered on top of linear ordered data
- We have freedom to change the latter as we please
  - This can be a useful insight when you design your own data structures

Inherent vs. Imposed Structure

- Our sample data only has ordering built in
  - 5 < 10 < 20 < 30 < 40 < 50 < 60 < 70 < 80 < 90
- Our two trees layer a grouping on top of this



AVL Analysis

- Ignoring deletion, insertion is the only operation that is different from a BST
- We've seen that rotation takes constant time (the case I didn't show is also constant time)
- Do we have to do more rotations?
  - No. Fixing the first problem node guarantees the others will be OK as we walk back up the tree
- Costs of AVL:
  - Extra depth data in each node (as much as +40% space)
  - 4 rotation cases to get right (L-L, L-R, R-L, R-R)

Next time

- Hashing
  - Read chapter 5 (skip section 5.6 in the C and C++ books)
- We may get to Heaps (chapter 6)
- Homework 1 due in class Friday!