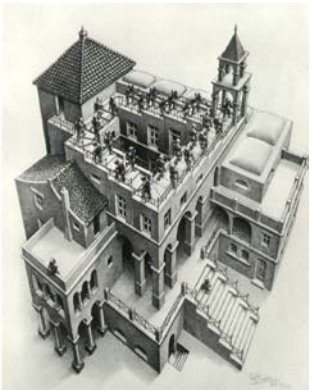
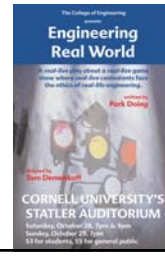


More ADTs

Lecture 17
CS211 – Fall 2006

Announcements

- A4 is online
  - Due Monday, Nov 6 (2 weeks minus 1 day)
- Ethics play:



- Cornell Mathematical Contest in Modeling
  - Teams of undergrads work over a weekend to solve real-world problems
    - Predator hunting strategies
    - Airline overbooking strategies
    - Policies to fight grade inflation
  - Contest dates: Oct 28-30
  - Information/Training
    - 10/17 at 6pm (251 Malott Hall) and
    - 10/25 at 6pm (253 Malott Hall)
  - $400+ in prizes

Recall

- We discussed several widely-used ADTs
  - Stacks & Queues
  - Dictionaries
  - Sets
  - Priority Queues
- For Stacks and Queues
  - Can implement so all operations take O(1) time
- For Dictionaries
  - Lists and arrays lead to slow implementations
  - Try Hash Table

Recall: A Hashing Example

- Suppose each word below has the following hashCode

jan	7
feb	0
mar	5
apr	2
may	4
jun	7
jul	3
aug	7
sep	2
oct	5

- How do we resolve collisions?
  - We'll use chaining: each table position is the head of a list
  - For any particular problem, this *might* work terribly
- In practice, using a good hash function, we can assume each position is equally likely

Recall: Analysis for Hashing with Chaining

- Analyzed in terms of *load factor* λ = n/m = (items in table)/(table size)
- We count the expected number of *probes* (key comparisons)
- Goal: Determine U = number of probes for an *unsuccessful* search
- Claim U is the same as the average number of items per table position = n/m = λ
- Claim S = number of probes for a *successful* search = 1 + λ/2

Table Doubling

- We know each operation takes time O(λ) where λ=n/m
- But isn't $\lambda = \Theta(n)$?
- What's the deal here? It's still linear time!

Table Doubling:

- Set a bound for λ (call it λ_0)
- Whenever λ reaches this bound we
  - Create a new table, twice as big and
  - Re-insert all the data
- Easy to see operations *usually* take time O(1)
  - But sometimes we copy the whole table

Analysis of Table Doubling

- Suppose we reach a state with n items in a table of size m and that we have just completed a table doubling

	Copying Work
Everything has just been copied	n inserts
Half were copied previously	n/2 inserts
Half of those were copied previously	n/4 inserts
…	…
Total work	n + n/2 + n/4 + … = 2n

Table Doubling, Cont'd

- Total number of insert operations needed to reach current table
  = copying work + initial insertions of items
  = 2n + n = 3n inserts
- Each insert takes expected time $O(\lambda_0)$ or O(1), so total expected time to build entire table is O(n)
  - Thus, expected time per operation is O(1)
- Disadvantages of table doubling:
  - Worst-case insertion time of O(n) is definitely achieved (but rarely)
  - Thus, not appropriate for time critical operations

Java Hash Functions

- Most Java classes implement their own hashCode() method
- hashCode() returns an int
- Java's HashMap class uses h(X) = X.hashCode() mod m
- h(X) in detail:

```
int hash = X.hashCode();
int index = (hash & 0x7FFFFFFF) % m;
```

- What hashCode() returns:
  - Integer:
    - uses the int value
  - Float:
    - converts to a bit representation and treats it as an int
  - Short Strings:
    - 37*previous + value of next character
  - Long Strings:
    - sample of 8 characters; 39*previous + next value

Hash Tables in Java

java.util.HashMap
java.util.HashSet
java.util.Hashtable (legacy)

- Uses chaining
- Initial (default) size = 101
- Load factor = λ_0 = 0.75
- Uses table doubling (2*previous+1)

- A node in each chain looks like this:

hashCode	key	value	next

original hashCode (before mod m)
Allows faster rehashing and (possibly) faster key comparison

Linear & Quadratic Probing

- These are techniques in which all data is stored directly within the hashtable array
- Linear Probing
  - Probe at h(X), then at
    - h(X) + 1
    - h(X) + 2
    - …
    - h(X) + i
  - Leads to *primary clustering*
    - Long sequences of filled cells
- Quadratic Probing
  - Similar to Linear Probing in that data is stored within the table
  - Probe at h(X), then at
    - h(X)+1
    - h(X)+4
    - h(X)+9
    - …
    - $h(X)+i^2$
  - Works well when
    - $\lambda < 0.5$
    - Table size is prime

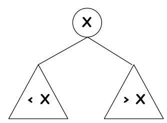
Hash Table Pitfalls

- Good hash function is required!
  - Whenever it is invoked on the same object, it *must* return the same result
  - Two objects that are equal *must* have the same hash code
  - Ideally: few collisions; even distribution of hash codes
- Watch the load factor (λ), especially for Linear & Quadratic Probing

Dictionary Implementations

- Ordered Array
  - Better than unordered array because Binary Search can be used for some operations
- Unordered Linked-List
  - Ordering doesn't help
- Direct Address Table
  - Small universe ⇒ limited usage
- Hashtables
  - O(1) expected time for Dictionary operations
  - Why look for anything better?

- Goal: Want ability to *report-in-order*, but can't afford inefficiency of ordered array
- Idea: Use a Binary Search Tree (BST)
- BST Property:



Deleting from a BST

Cases:

- Delete a leaf
  - Easy
- Delete a node with just one child
  - Delete and replace with child
- Delete a node with two children
  - Delete node's successor
  - Write successor's data into node

- How do we find the successor?
- The successor always has at most one child. Why?
- Would work just as well using predecessor instead of successor

BST Performance

- Time for insert(), find(), update(), remove() is O(h) where h is the height of the tree
- How bad can h be?
- Operations are fast if tree is *balanced*

- How balanced is a random tree?
  - If items are inserted in random order then the expected height of a BST is O(log n) where n is the number of items
- If deletion is allowed
  - Tree is no longer random
  - Tree is likely to become unbalanced

Analysis Sketch for Random BST

- Only the number of items and their order is important
  - Can restrict our attention to BSTs containing items {1,…, n}
- We assume that each item is equally likely to appear as the root
- Define H(n) ≡ *expected* height of BST of size n
- If item i is the root then expected height is
  1 + max { H(i-1), H(n-i) }
  We average this over all possible i
- Can solve the resulting recurrence (by induction) to show
  H(n) = O(log n)

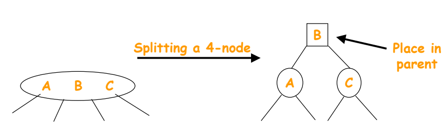
Why use a BST instead of a Hash Table?

- Balanced BST vs. Hash Table
  - Worst-case time O(log n) vs. expected time O(1)
- BSTs provide (additional) operations more efficiently
  - report-elements-in-order
  - getMin
  - getMax
  - select(k) // Find k^{th} element
    - (maintain size of each subtree by using an additional size field in each node)

- Criticism: Balanced BST schemes can be difficult to implement
  - But there are lots of reliable codes for these schemes available on the Web
  - Java includes a balanced BST scheme among its standard classes (java.util.TreeMap and java.util.TreeSet)

Example Balancing Scheme: 234-Trees

- Nodes have 2, 3, or 4 children (and contain 1, 2, or 3 keys, respectively)
- All leaves are at the same level
- Basic rule for insertion: We hate 4-nodes
  - Split a 4-node whenever you find one while coming down the tree
  - Note: this requires that parent is not a 4-node
- Delete is harder than insert
  - For delete, we hate 2-nodes
  - As in BSTs, cannot delete from a nonleaf so we use same BST trick: delete successor and recopy its data



234-Tree Analysis

- Time for insert or get is proportional to tree's height
- How big is tree's height h?
- Let n be the number of nodes in a tree of height h
  - n is large if all nodes are 4-nodes
  - n is small if all nodes are 2-nodes
- Can use this to show
  h = O(log n)

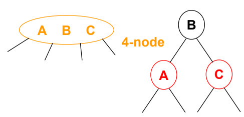
Analysis of tree height:

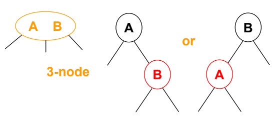
- Let N be the number of nodes, n be the number of items, and h be the height
- Define h so that a tree consisting of a single node is height 0
- It's easy to see $1+2+4+\ldots+2^h \leq N \leq 1+4+16+\ldots+4^h$
- It's also easy to see $N \leq n \leq 3N$
- Using the above, we have $n \geq 1+2+4+\ldots+2^h = 2^{h+1}-1$
- Rewriting, we have $h \leq \log(n+1) - 1$ or h = O(log n)
- Thus, Dictionary operations on 234-trees take time O(log n) in the worst case

234-Tree Implementation

- Can implement all nodes as 4-nodes
  - Wasted space
- Can allow various node sizes
  - Requires recopying of data whenever a node changes size
- Can use BST nodes to emulate 2-, 3-, or 4-nodes

Using BSTs to Emulate 234-Trees





- A 2-node can be represented with a standard BST node
- A 4-node can be represented with three BST nodes
- A 3-node can be represented with two BST nodes (in two different ways)

Red-Black Trees

- We need a way to tell when an emulated 234-node starts and ends
- We mark the nodes
  - Black: "root" of 234-node
  - Red: belongs to parent
  - Requires one bit per node
- 234-tree rules become rules for *rotations* and color changes in red-black trees
- Result:
  - One black node per 234-node
  - Number of black nodes on path from root to leaf is same as height of 234-tree
  - On any path: at most one red node per black node
  - Thus tree height for red-black tree is O(log n)

Balanced Tree Schemes

- AVL trees [1962]
  - named for initials of Russian creators
  - uses rotations to ensure heights of child trees differ by at most 1
- 23-Trees [Hopcroft 1970]
  - similar to 234-tree, but repairs have to move back up the tree
- B-Trees [Bayer & McCreight 1972]
- Red-Black Trees [Bayer 1972]
  - not the original name
- Red-black convention & relation to 234-trees [Guibas & Stolfi 1978]
- Splay Trees [Sleator & Tarjan 1983]
- Skip Lists [Pugh 1990]
  - developed at Cornell

Selecting a Dictionary Scheme

- Use an unordered array for small sets (< 20 or so)
- Use a Hash Table if possible
  - Cannot efficiently do some ops that are easy with BSTs
  - Running times are expected rather than worst-case
- Use an ordered array if few changes after initialization
- B-Trees are best for large data sets, external storage
  - Widely used within data base software
- Otherwise, Red-Black Trees are current scheme of choice
- Skip Lists are supposed to be easier to implement
  - But shouldn't have to implement—use existing code
- Splay trees are useful if some items are accessed more often than others
  - But if you know which items are most-commonly accessed, use a separate data structure