BSTs and Balanced Trees

CS211
Fall 2000

Quadratic Probing + Hashing Pitfalls

- 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, ...
  - Works well when
    - λ < 0.5
    - Table size is prime

- Hash Table Pitfalls
  - Good hash function is required
  - Watch the load factor (λ), especially for Linear & Quadratic Probing

Dictionary Implementations

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

Deleting from a BST

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

```
      x
    /   \
   x     x
```

- 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 put(), get(), 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)
    - H(n) = O(log n)
Why use a BST instead of a Hashtable?

- If we use a balanced BST scheme then we achieve guaranteed worst-case time bounds.
- There are some operations that can be efficient on BSTs, but very inefficient on Hashtables.

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 the tree.
    - $n$ is large if all nodes are 4-nodes.
    - $n$ is small if all nodes are 2-nodes.
  - Can use this to show $h = O(\log n)$.

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.
- All paths from root to leaf have same number of black nodes.
- 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

Selecting a Priority Queue Scheme

- Use an unordered array for small sets (< 20 or so)
- Use a sorted array or sorted linked list if few insertions are expected
- Use an array of linked lists if there are few priorities
  - Each linked list is a queue of equal-priority items
  - Very easy to implement
- Otherwise, use a Heap if you can
- Heap + Hashtable
  - Allow change-priority operation to be done in O(log n) expected time
- Balanced tree schemes
  - Useful and practical
  - There are a number of alternate implementations that allow additional operations
    - Fibonacci heaps
    - Skew heaps
    - ...

Topics Covered Since Last Exam

- Data structure building blocks
  - Arrays, Lists, Trees, Graphs
  - The Java Collections Framework
    - Interfaces: Set, SortedSet, List, Map, SortedMap, Iterator
    - Classes: HashSet, TreeSet, ArrayList, LinkedList, HashMap, TreeMap
    - Utilities: java.util.Arrays, java.util.Collections
- GUIs
  - Layout
  - Event handling
- Priority Queues
  - Heaps
  - BSTs, Balanced Trees
  - Array of lists (index value is priority)
- Dictionaries
  - BSTs
  - Balanced Trees
  - Hash Tables