More on ADTs: Priority Queues & Dictionaries

Lecture 17
CS211 – Fall 2005

Announcements

• Assignment 5 is online (since Friday)
  • Due Wednesday, Nov 2
  • Multiple small tasks

Heaps

• A heap is a tree that
  • Has a particular shape (we’ll come back to this) and
  • Has the heap property
• Heap property
  • Each node’s value (its priority) is ≤ the value of its parent
  • This version is for a max-heap (max value at the root)
  • There is a similar heap property for a min-heap (min at the root)

Heap Implementation (the Big Trick)

• Can avoid using pointers!
• Use a complete binary tree stored in an array
  • Definition: Complete means that each level of the tree is filled except possibly the last, which is filled from left to right
• For A[i]
  • left child = 2 * i
  • right child = 2 * i + 1
  • parent = \[i / 2\]

Insert and GetMax Pseudocode

insert (item):
  Place item in a leaf (= next empty position in array);
  while (item > parent) {Swap item with parent;} // BubbleUp

getMax (i):
  max = root.value;
  Swap root and last item (call it v) in heap; // Ensures same shape for heap
  Decrease heap size by 1 (i.e., access less of the array);
  while (v < one of its children) // BubbleDown
    {Swap v with its largest child;}
  return max;

To Build a Heap

• How long to construct a heap, given the items?
• Worst-case time for insert() is O(log n)
• Total time to build heap using insert() is O(n log 1) + O(log 2) + ... + O(log n) or O(n log n)
  • We had two heap-fixing methods
    bubbleUp: move up the tree as long as we’re > our parent
    bubbleDown: move down the tree as long as we’re < one of our children
  • If we build the heap from the bottom-up using bubbleDown then we can build it in time O(n) (Wow!)
Efficient Heap Building

- Build from the bottom-up
- If there are n items in the heap...
  - There are about \( \frac{n}{2} \) mini-heaps of height 1
  - There are about \( \frac{n}{4} \) mini-heaps of height 2
  - There are about \( \frac{n}{8} \) mini-heaps of height 3 and so on
- The time to fix up a mini-heap is \( \Theta(\text{its height}) \)
- Total time spent fixing heaps is thus bounded by
  \[
  \frac{n}{2} + \frac{2n}{4} + \frac{3n}{8} + \ldots
  \]
- This can be rewritten as
  \[
  n \left( \frac{1}{2} + \frac{2}{4} + \ldots + \frac{i}{2^i} + \ldots \right) = n(2)
  \]
- Thus total heap-building time (using the bottom-up method) is \( \Theta(n) \)

HeapSort

- Given a Comparable[ ] array of length n,
  - Put all n elements into a heap: \( \Theta(n) \) or \( \Theta(n \log n) \)
  - Repeatedly get the min: \( \Theta(n \log n) \)

```java
public static void heapSort(Comparable[] a) {
    PriorityQueue<Comparable> pq = new PQ<Comparable>();
    for (Comparable x : a) { pq.put(x); }
    for (int i = 0; i < a.length; i++) { a[i] = pq.get(); }
}
```

PQ Application: Simulation

- Example: Given a probabilistic model of bank-customer arrival times and transaction times, how many tellers are needed
  - Assume we have a way to generate random inter-arrival times
  - Assume we have a way to generate transaction times
  - Can simulate the bank to get some idea of how long customers must wait
- Time-Driven Simulation
  - Check at each tick to see if any event occurs
- Event-Driven Simulation
  - Advance clock to next event, skipping intervening ticks
  - This uses a PQ!

Another PQ Implementation

- If there are only a few possible priorities then can use an array of queues
  - Each array position represents a priority (0..m-1 where m is the array size)
  - Each queue holds all items that have that priority
  - One text [Skiena] calls this a bounded height priority queue
  - Time for insert: \( \Theta(1) \)
  - Time for getMin:
    - \( \Theta(m) \) in the worst-case
    - Generally, faster
  - Example: airline check-in

Other PQ Operations

- `delete` a particular item
- `update` an item (change its priority)
- `join` two priority queues

- For delete and update, we need to be able to find the item
  - One way to do this: Use a Dictionary to keep track of the item’s position in the heap
- Efficient joining of 2 Priority Queues requires another data structure
  - Skew Heaps or Pairing Heaps (Chapter 23 in text)

Recall: Sets & Dictionaries

- ADT Set
  - Operations:
    - void insert (Object element);
    - boolean contains (Object element);
    - void remove (Object element);
    - boolean isEmpty () ;
    - void makeEmpty () ;
  - Note: no duplicates allowed
  - A “set” with duplicates is usually called a bag
- Where used:
  - Wide use within other algorithms

- ADT Dictionary
  - Operations:
    - void insert (Object key, Object value);
    - void update (Object key, Object value);
    - Object find (Object key);
    - void remove (Object key);
    - boolean isEmpty () ;
    - void makeEmpty () ;
  - Think of key = word; value = definition
- Where used:
  - Symbol tables
  - Wide use within other algorithms
Implementing Sets

- Recall: ADT Set
  - Operations:
    - void insert (Object element);
    - boolean contains (Object element);
    - void remove (Object element);
    - boolean isEmpty ( );
  - Can use a Dictionary
    - Values in (key, value) pairs are ignored
    - All operations are expected time O(1) using hash table (see next several slides)
  - If the universe is not too large
    - Can use a table of bits (i.e., a bit-vector)
      - We need n bits for a universe of size n
    - This implementation also allows for fast union, intersection, and complement

Goal: Design a Dictionary

- Operations
  - void insert (key, value)
  - void update (key, value)
  - Object find (key)
  - void remove (key)
- Array implementation:
  - Uses an array of (key, value) pairs
  - Unsorted Sorted
  - insert O(1) O(n)
  - update O(n) O(log n)
  - find O(n) O(log n)
  - remove O(n) O(n)

Direct Address Table

- Assumes the key set is from a small Universe
- Example: Addresses on my street
  - Start at 1, go to 40
  - A few lots don’t have houses
- For a Direct Address Table, we make an array as large as the Universe
- To find an entry, we just index to that entry of the array
- Dictionary operations all take O(1) time

What if the Universe is large?

- Idea is to re-use table entries via a hash function
  - h : U \rightarrow [0, \ldots, m-1]
    - U = all legal identifiers
    - m = table size
  - h must
    - Be easy to compute
    - Cause few collisions
    - Have equal probability for each table position

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

Analysis for Hashing with Chaining

- Analyzed in terms of load factor \( \lambda = \frac{n}{m} \), where \( n \) is the number of items currently held in the array
- Claim U is the same as the average number of items per table position = \( n/m = \lambda \)
- Claim S = number of probes for a successful search = \( 1 + \lambda/2 \)
Table Doubling

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

Table Doubling:
- Set a bound for $\lambda$ (call it $\lambda_0$)
- Whenever $\lambda$ 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

<table>
<thead>
<tr>
<th>Copying Work</th>
</tr>
</thead>
<tbody>
<tr>
<td>everything has just been copied</td>
</tr>
<tr>
<td>Half were copied previously</td>
</tr>
<tr>
<td>Half of those were copied previously</td>
</tr>
<tr>
<td>Total work</td>
</tr>
</tbody>
</table>

Analysis of 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 the `hashCode()` method
- `hashCode()` returns an `int`
- Java’s `HashMap` class uses $h(X) = X$..`hashCode()` mod $m$
- $h(X)$ in detail:
  ```java
  int hash = X.hashCode();
  int index = (hash & 0x7FFFFFFF) % m;
  ```

hashCode( ) Requirements

- Contract for `hashCode()` method:
  - Whenever it is invoked in the same object, it must return the same result
  - Two objects that are equal must have the same hash code
  - Two objects that are not equal should return different hash codes, but are not required to do so

Hash Tables in Java

- Java’s `HashMap` class
- `java.util.HashMap`
- `java.util.HashSet`
- `java.util.Hashtable` (legacy)
- Use chaining
- Initial (default) size = 101
- Load factor = $\lambda_0 = 0.75$
- Uses table doubling ($2^\text{previous}+1$)

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]
Hashing Application: Spell Checking

- We want to create a “spelling dictionary” containing 10,000 words
  - A spelling query should be fast
  - Should return true iff word is contained in dictionary

**Basic idea:**
- Use a Hashtable consisting only of bits (say 100K bytes or about 800,000 bits)
- Compute a hash value for each word and turn on the corresponding bit in the table
- What’s the probability of a false positive? (It’s too high!)
- Fix: Use more hash functions

---

Linear & Quadratic Probing

- These are techniques in which all data is stored directly within the hash table 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, ...
  - Works well when \( \lambda < 0.5 \)
    - Table size is prime

---

Hash Table Pitfalls

- Good hash function is required
- Watch the load factor (\( \lambda \)), 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 \( \Rightarrow \) limited usage
- **Hashables**
  - \( O(1) \) expected time for Dictionary operations

- **Goal**: Want ability to report-in-order, but can’t afford inefficiency of ordered array
- **Idea**: Use a Binary Search Tree (BST)
- **BST Property:**
  - \( x < x' \Rightarrow \) x is a left child of x'

---

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 balanced 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, \ldots, n\}
- We assume that each item is equally likely to appear as the root
- Define \( H(n) = \text{expected height of BST of size } n \)
- 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 Hashtable?

- If we use a balanced BST scheme then we achieve guaranteed worst-case time bound of \( O(\log n) \) for typical Dictionary ops
- There are some operations that can be efficient on BSTs, but very inefficient on Hashables
  - report-elements-in-order
  - getMin
  - getMax
  - select(k) // find the \( k \)-th element
    (maintain size of each subtree by using an additional size field in each node)
- Note that 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 packages (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 \( N \leq 3N \)
  - 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
  - Requires recopying of data whenever a node changes size
- 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 BTs
  - 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 database 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 + Hashable
  - 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
  - Skew heaps
  - Pairing heaps
  - Fibonacci heaps
  - …