CSE 326 Data Structures
Midterm Review

Hal Perkins
Spring 2007

Dates
• Midterm Friday!
• Project 2 due next Wednesday
• Homework 4
  – Hmmm…..
  – We ought to talk about this….

Logistics
• Closed Notes
• Closed Book
• Open Mind
• You may bring a calculator, though don’t even think about loading it with notes or programs. And you probably won’t find it of much use anyway.

Material Covered
• Everything we’ve talked/read in class up to AVL trees
  – And for AVL trees, up to inserting and rotations, but not implementations in Java
**Material Not Covered**

- We won’t make you write syntactically correct Java code (pseudocode okay)
- We won’t make you do a super hard proof
- We won’t test you on the details of generics, interfaces, etc. in Java
  – But you should know the basic ideas since we spent a lecture on them and had to deal with them in project 2A

**Order Notation: Definition**

\[ O(f(n)) : \text{ a set or class of functions} \]

\[ g(n) \in O(f(n)) \iff \text{there exist constants } c \text{ and } n_0 \text{ such that:} \]

\[ g(n) \leq c \cdot f(n) \text{ for all } n \geq n_0 \]

**Example:** \( g(n) = 1000n \) vs. \( f(n) = n^2 \)

Is \( g(n) \in O(f(n)) \)?

Pick: \( n_0 = 1000 \), \( c = 1 \)

\[ 1000n \leq 1 \cdot n^2 \text{ for all } n \geq 1000 \]

So \( g(n) \in O(f(n)) \)

**Definition of Order Notation**

- **Upper bound:** \( T(n) = O(f(n)) \) or \( T(n) = O(f(n)) \)
  - Exist constants \( c \) and \( n' \) such that \( T(n) \leq c \cdot f(n) \) for all \( n \geq n' \)

- **Lower bound:** \( T(n) = \Omega(g(n)) \)
  - Exist constants \( c \) and \( n' \) such that \( T(n) \geq c \cdot g(n) \) for all \( n \geq n' \)

- **Tight bound:** \( T(n) = \Theta(f(n)) \)
  - When both hold: \( T(n) = O(f(n)) \) and \( T(n) = \Omega(f(n)) \)

**Log?**

\[ \log_k n \in O(\log_2 n)? \]

\[ \log_2 n^2 \in O(\log_2 n)? \]
Priority Queue ADT

- Checkout line at the supermarket ???
- Printer queues ???
- operations: insert, deleteMin

Implementations of Priority Queue ADT

<table>
<thead>
<tr>
<th></th>
<th>insert</th>
<th>deleteMin</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unsorted list (Array)</td>
<td>O(1)/O(N)</td>
<td>O(N)</td>
</tr>
<tr>
<td></td>
<td>worst-case fill, O(N) – to find value should say WHY, might reject on full instead</td>
<td></td>
</tr>
<tr>
<td>Unsorted list (Linked-List)</td>
<td>O(1)</td>
<td>O(N)</td>
</tr>
<tr>
<td></td>
<td>O(log N) to find loc w. Bin search, but O(N) to move vals, (or O(1) if in reverse order)</td>
<td></td>
</tr>
<tr>
<td>Sorted list (Array)</td>
<td>O(N)</td>
<td>O(1)</td>
</tr>
<tr>
<td></td>
<td>O(N) to find loc, O(1) to do the insert</td>
<td></td>
</tr>
<tr>
<td>Sorted list (Linked-List)</td>
<td>O(N)</td>
<td>O(N)</td>
</tr>
<tr>
<td></td>
<td>O(log N) to find val, leaf</td>
<td></td>
</tr>
<tr>
<td>Binary Search Tree (BST)</td>
<td>O(N)</td>
<td>O(N)</td>
</tr>
<tr>
<td></td>
<td>O(N) to find loc, O(1) to do the insert</td>
<td></td>
</tr>
<tr>
<td>Binary Heap</td>
<td>O(log N)</td>
<td>O(N)</td>
</tr>
<tr>
<td></td>
<td>close to O(1) 1.67 levels on average</td>
<td></td>
</tr>
</tbody>
</table>

Tree Review

- \( \text{root}(T) \): A
- \( \text{leaves}(T) \): DEFJ..NI
- \( \text{children}(B) \): B, C
- \( \text{parents}(H) \): A
- \( \text{siblings}(E) \): D, F
- \( \text{ancestors}(F) \): A
- \( \text{descendants}(G) \): J, K, L, M, N
- \( \text{subtree}(C) \): itself plus all descendents

Heap Structure Property

- A binary heap is a complete binary tree.

Complete binary tree – binary tree that is completely filled, with the possible exception of the bottom level, which is filled left to right.

Examples:

Since they have this regular structure property, we can take advantage of that to store them in a compact manner.
**Heap Order Property**

**Heap order property:** For every non-root node X, the value in the parent of X is less than (or equal to) the value in X.

This is the order for a MIN heap – could do the same for a max heap.

![Heap order diagrams](image)

**Representing Complete Binary Trees in an Array**

From node i:
- left child: $2 \times i$
- right child: $(2 \times i) + 1$
- parent: $\lfloor i / 2 \rfloor$

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
<th>G</th>
<th>H</th>
<th>I</th>
<th>J</th>
<th>K</th>
<th>L</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
</tr>
</tbody>
</table>

**Heap Operations**

- findMin:
- insert(val): percolate up.
- deleteMin: percolate down.

![Heap operations diagrams](image)

**Insert: percolate up**

Now insert 90. (no swaps, even though 99 is larger!)

Optimization, bubble up an empty space to reduce # of swaps

Now insert 7.
DeleteMin: percolate down

Max # of exchanges? = O(\log N),
There is a good chance goes to bottom (started at bottom) vs. insert
- Could also use the percolate empty bubble down

BuildHeap: Floyd’s Method

Add elements arbitrarily to form a complete tree.
Pretend it’s a heap and fix the heap-order property!
Red nodes need to percolate down

A Solution: d-Heaps
• Each node has d children
• Still representible by array
• Good choices for d:
  – (choose a power of two for efficiency)
  – fit one set of children in a cache line
  – fit one set of children on a memory page/disk block

Operations on d-Heap
• Insert : runtime =
  depth of tree decreases,
  O(\log_d n) worst
• deleteMin: runtime =
  percolateDown requires comparison to find min,
  O(d \log_d n), worst/ave

Does this help insert or deleteMin more?
**Definition: Null Path Length**

Null path length (npl) of a node \( x \) is the number of nodes between \( x \) and a null in its subtree

OR

\[ \text{npl}(x) = \text{min} \text{ distance to a descendant with 0 or 1 children} \]

- \( \text{npl}(\text{null}) = -1 \)
- \( \text{npl}(\text{leaf}) = 0 \)
- \( \text{npl}(\text{single-child node}) = 0 \)

Equivalent definitions:
1. \( \text{npl}(x) \) is the height of largest complete subtree rooted at \( x \)
2. \( \text{npl}(x) = 1 + \text{min}\{\text{npl}(\text{left}(x)), \text{npl}(\text{right}(x))\} \)

**Leftist Heap Properties**

- **Heap-order property**
  - parent’s priority value \( \leq \) to children’s priority values
  - result: minimum element is at the root

- **Leftist property**
  - For every node \( x \), \( \text{npl}(\text{left}(x)) \geq \text{npl}(\text{right}(x)) \)
  - result: tree is at least as “heavy” on the left as the right

Are leftist trees…
- complete? No,
- balanced? no

**Merging Two Leftist Heaps**

- \( \text{merge}(T_1, T_2) \) returns one leftist heap containing all elements of the two (distinct) leftist heaps \( T_1 \) and \( T_2 \)

**Leftist Merge Continued**

- Swapping L and R if needed
- Work at each step = call to merge, swap (constant)
  - traverse the right path of both trees = length is at most \( \log N \)
- runtime: \( O(\log n) \)
**Leftist Merge Example**

![Diagram of leftist merge example]

**Sewing Up the Leftist Example**

![Diagram showing sewing up the leftist example]

**Finally...(Leftist)**

![Diagram showing leftist example]

**Skew Heaps**

Problems with leftist heaps:
- extra storage for npl
- extra complexity/logic to maintain and check npl
- right side is “often” heavy and requires a switch

Solution: skew heaps:
- “blindly” adjusting version of leftist heaps
- merge always switches children when fixing right path
- amortized time for: merge, insert, deleteMin = $O(\log n)$
- however, worst case time for all three = $O(n)$
Merging Two Skew Heaps

Only one step per iteration, with children always switched

Yet Another Data Structure: Binomial Queues

What’s a forest?
What’s a binomial tree?

• Structural property
  – Forest of binomial trees with at most one tree of any height

• Order property
  – Each binomial tree has the heap-order property

The Binomial Tree, $B_h$

• $B_h$ has height $h$ and exactly $2^h$ nodes
• $B_h$ is formed by making $B_{h-1}$ a child of another $B_{h-1}$
  • Root has exactly $h$ children
• Number of nodes at depth $d$ is binomial coeff. \( \binom{h}{d} \)
  – Hence the name; we will not use this last property

Binomial Queue with $n$ elements

Binomial Q with $n$ elements has a unique structural representation in terms of binomial trees!

Write $n$ in binary: $n = 1101_{\text{base 2}} = 13_{\text{base 10}}$
Merging Two Binomial Queues

Essentially like adding two binary numbers!

1. Combine the two forests
2. For \( k \) from 1 to maxheight {
   a. \( m \leftarrow \) total number of \( B_k \)'s in the two BQs
   b. if \( m=0 \): continue;
   c. if \( m=1 \): continue;
   d. if \( m=2 \): combine the two \( B_k \)'s to form a \( B_{k+1} \)
   e. if \( m=3 \): retain one \( B_k \) and combine the other two to form a \( B_{k+1} \)
}

Claim: When this process ends, the forest has at most one tree of any height

Example: Binomial Queue Merge

H1:  H2:

```
Example: Binomial Queue Merge

H1:  H2:

Example: Binomial Queue Merge

H1:  H2:
```
Example: Binomial Queue

Merge

H1: 1
  2 1 3
  5 1 5
  6

H2: 1
  3
  4
  5

Example: Binomial Queue

Merge

H1: 3
  2 1 3
  8 11 5
  6

H2: 1
  7
  2 1
  3 6
  9 6

More Recursive Tree Calculations:

Tree Traversals

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

Three types:
- **Pre-order**: Root, left subtree, right subtree
- **In-order**: Left subtree, root, right subtree
- **Post-order**: Left subtree, right subtree, root

The Dictionary ADT

• **Data**:
  - a set of (key, value) pairs

• **Operations**:
  - Insert (key, value)
  - Find (key)
  - Remove (key)

The Dictionary ADT is sometimes called the “Map ADT”.

- gerbil
  - small rodent
- Rat
  - larger rodent
- mouse
  - annoying rodent

insert(mouse, …)

find(rat)

The Dictionary ADT is sometimes called the “Map ADT”.
Binary Search Tree Data Structure

- **Structural property**
  - each node has \( \leq 2 \) children
  - result:
    - storage is small
    - operations are simple
    - average depth is small

- **Order property**
  - all keys in left subtree smaller than root's key
  - all keys in right subtree larger than root's key
  - result: easy to find any given key

- What must I know about what I store?
  - Comparison, equality testing

Find in BST, Recursive

```
Node Find(Object key,
          Node root) {
    if (root == NULL) {
        return NULL;
    }
    if (key < root.key) {
        return Find(key,
                    root.left);
    } else if (key > root.key) {
        return Find(key,
                    root.right);
    } else {
        return root;
    }
}
```

Runtime:

\( \Theta(\text{depth}) = \Theta(n) \) worst, \( \Theta(\log n) \) avg

Insert in BST

- Insert(13)
- Insert(8)
- Insert(31)

Runtime:

\( O(\text{depth}) = O(n) \) worst, \( O(\log n) \) avg

Insertions happen only at the leaves – easy!

Deletion in BST

Why might deletion be harder than insertion?

May be in middle, instead of at leaf
Non-lazy Deletion – The Leaf Case

Delete(17)

Deletion – The One Child Case

Delete(15)

Deletion – The Two Child Case

Delete(5)

Lazy Deletion

Instead of physically deleting nodes, just mark them as deleted

+ simpler
+ physical deletions done in batches
+ some adds just flip deleted flag

− extra memory for deleted flag
− many lazy deletions slow finds
− some operations may have to be modified (e.g., min and max)

How long do these operations take? (find, insert, delete)
Balanced BST

Observation
• BST: the shallower the better!
• For a BST with \( n \) nodes
  – Average height is \( O(\log n) \)
  – Worst case height is \( O(n) \)
• Simple cases such as insert(1, 2, 3, ..., n) lead to the worst case scenario

Solution: Require a **Balance Condition** that
1. ensures depth is \( O(\log n) \) – strong enough!
2. is easy to maintain – not too strong!

The AVL Balance Condition
Left and right subtrees of every node have equal heights differing by at most 1

Define: \( \text{balance}(x) = \text{height}(x.\text{left}) - \text{height}(x.\text{right}) \)

AVL property: \(-1 \leq \text{balance}(x) \leq 1\), for every node \( x \)

• Ensures small depth
  – Will prove this by showing that an AVL tree of height \( h \) must have a lot of (i.e. \( O(2^h) \)) nodes
• Easy to maintain
  – Using single and double rotations

The AVL Tree Data Structure

Structural properties
1. Binary tree property
2. Balance property: balance of every node is between -1 and 1

Result:
Worst case depth is \( O(\log n) \)

Ordering property
– Same as for BST
AVL tree insert

Let $x$ be the node where an imbalance occurs.

Four cases to consider. The insertion is in the
1. left subtree of the left child of $x$.
2. right subtree of the left child of $x$.
3. left subtree of the right child of $x$.
4. right subtree of the right child of $x$.

Idea: Cases 1 & 4 are solved by a single rotation. Cases 2 & 3 are solved by a double rotation.

Fix: Apply Single Rotation

AVL Property violated at this node (x)

Single Rotation:
1. Rotate between $x$ and child

Single rotation in general

Before red dot, $X = h$,
Q: height of tree is? $h+2$,
After red dot, $X = h+1$

$X < b < Y < a < Z$

Case 1, same for case 4

Height of tree before? Height of tree after? Effect on Ancestors?
Fix: Apply Double Rotation

AVL Property violated at this node (x)

Intuition: 3 must become root

Double Rotation
1. Rotate between x’s child and grandchild
2. Rotate between x and x’s new child

Balanced?

Double rotation in general

- Before red dot, X = h-1,
  - Q: height of tree is? h+2,
- After red dot, X = h
  - CROSSOUT

- Actually red dot could be at X or Y

W < b < X < c < Y < a < Z

Case 2, same for case 3

Height of tree before? Height of tree after? Effect on Ancestors?

Double rotation, step 1

Double rotation, step 2
Insertion into AVL tree

1. Find spot for new key
2. Hang new node there with this key
3. Search back up the path for imbalance
4. If there is an imbalance:
   - case #1: Perform single rotation and exit
   - case #2: Perform double rotation and exit

Both rotations keep the subtree height unchanged. Hence only one (single or double) rotation is sufficient!