CSE 326 Data Structures

Dave Bacon

Midterm Review
Dates

- Midterm Friday!
- Project 2 due next Wednesday
- Homework 4 due next Friday
The Temporal Setup

- Friday, in class (i.e. 12:30-1:20) MGH 231
- Must take in lecture you are registered for
- We will start PROMPTLY at 12:30!

- You’ve got 50 minutes, don’t get hung up on a problem!
Logistics

- Closed Notes
- Closed Book
- Open Mind
- You may bring a calculator, though don’t even think about loading it with notes or programs.
Material Covered

- Everything we’ve talked/read in class up to AVL trees
- No splay trees
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 like in problems 1b and 2b from HW 3
- We won’t test you on the details of generics, interfaces, etc. in Java
Study Guide

• Sample midterm on website with solutions
• Midterm study guide

• Homework 1-3 solutions on web
Order Notation: Definition

\( O(f(n)) : \) a set or class of functions

\( g(n) \in O(f(n)) \) if there exist constants \( c \) and \( n_0 \) such that:

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

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

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

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

$\log_k n \in O(\log_2 n)$?

$\log \log n$

$\log_2 n^2 \in O(\log_2 n)$?
Definition of **Order Notation**

- **Upper bound:** \( T(n) = O(f(n)) \)  
  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:
  
  \[ T(n) = O(f(n)) \]
  \[ T(n) = \Omega(f(n)) \]
Priority Queue ADT

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

[Diagram showing operations and elements: insert and deleteMin with values 6, 2, 15, 23, 12, 18, 45, 3, 7]
# 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></td>
<td></td>
</tr>
<tr>
<td>Unsorted list (Linked-List)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Sorted list (Array)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Sorted list (Linked-List)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Binary Search Tree (BST)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Binary Heap</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Tree Review

\textbf{root}(T):
\textbf{leaves}(T):
\textbf{children}(B):
\textbf{parent}(H):
\textbf{siblings}(E):
\textbf{ancestors}(F):
\textbf{descendants}(G):
\textbf{subtree}(C):
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:
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.

```
not a heap
```

```
10
  20
   30
   15

10
  20
   40
   60
   80
   85
   99
```

```
10
  80
   50
   700
```
Representing Complete Binary Trees in an Array

From node $i$:
- left child:
- right child:
- parent:

Implicit (array) implementation:

<table>
<thead>
<tr>
<th></th>
<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>
<td>12</td>
</tr>
</tbody>
</table>
Heap Operations

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

```
  10
 /   \
20   80
|    /\|
40  60 85 99
|  |  |  |
50 700 65 15
```

```
  10
 /   \
15   80
|    /\|
40  20 85 99
|  |  |  |
50 700 65 60
```
DeleteMin: percolate down
BuildHeap: Floyd’s Method

Add elements arbitrarily to form a complete tree. Pretend it’s a heap and fix the heap-order property!
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 =

- deleteMin: runtime =

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

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

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

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

Equivalent definitions:

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

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

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

Are leftist trees... complete? balanced?
Merging Two Leftist Heaps

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

If $npl(R') > npl(L_1)$

$R' = \text{Merge}(R_1, T_2)$

runtime:
Leftist Merge Example

(special case)
Sewing Up the Leftist Example

Done?
Finally… (Leftist)
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

- 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: $n = 1101_{\text{(base 2)}} = 13_{\text{(base 10)}}$

![Binomial Tree Diagram]
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:  
1
 /  
7 3
 /   
21  

-1
 /  
2 1
 /  
8 11 5
    /  
6    

H2:  
5
 /  
9 6
 /  
7
Example: Binomial Queue Merge

H1: 

H2:
Example: Binomial Queue Merge

H1:

H2:
Example: Binomial Queue Merge

H1:

H2:
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

(an expression tree)
The Dictionary ADT

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

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

```
insert(rea, ....)
```

```find(dabacon)
```

```
• dabacon
  Dave Bacon
  OH: Tu 12:30-1:30
  CSE 460

• ethanpg
  Ethan Phelps-Goodman,
  OH: Th 10:30-11:30
  CSE 218

• rea
  Ruth Anderson
  OH: M 3:30-4:30
  CSE 460
```

*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?
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;
}
Insert in BST

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

Insertions happen only at the leaves – easy!

Runtime:
Deletion in BST

Why might deletion be harder than insertion?
Non-lazy Deletion – The Leaf Case

Delete(17)
Deletion – The One Child Case

Delete(15)
Deletion – The Two Child Case

Delete(5)

What can we replace 5 with?
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)
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

\[ X < b < Y < a < Z \]

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

AVL Property violated at this node (x)

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

\[
\begin{align*}
\text{Height of tree before?} & \quad \text{Height of tree after?} \quad \text{Effect on Ancestors?}
\end{align*}
\]
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!