Announcements

- Homework 3 is due
- Solutions 1 hour after class
- Course Evaluation
- Midterm Exam March 11th
Review

- Clarification about isomorphism
- buildHeap example
- HeapSort and HeapSelect
Math Background: Exponents

\[ X^A X^B = X^{A+B} \]
\[ \frac{X^A}{X^B} = X^{A-B} \]
\[ (X^A)^B = X^{AB} \]
\[ X^N + X^N = 2X^N \neq X^{2N} \]
\[ 2^N + 2^N = 2^{N+1} \]
Math Background: Logarithms

\[ X^A = B \text{ iff } \log_X B = A \]

\[ \log_A B = \frac{\log_C B}{\log_C A}; \quad A, B, C > 0, A \neq 1 \]

\[ \log AB = \log A + \log B; \quad A, B > 0 \]
Math Background: Series

\[
\sum_{i=0}^{N} 2^i = 2^{N+1} - 1
\]

\[
\sum_{i=0}^{N} A^i = \frac{A^{N+1} - 1}{A - 1}
\]

\[
\sum_{i=1}^{N} i = \frac{N(N + 1)}{2} \approx \frac{N^2}{2}
\]

\[
\sum_{i=1}^{N} i^2 = \frac{N(N + 1)(2N + 1)}{6} \approx \frac{N^3}{3}
\]
Big-Oh Notation

- We adopt special notation to define **upper bounds** and **lower bounds** on functions.
- In CS, usually the functions we are bounding are running times, memory requirements.
- We will refer to the running time as $T(N)$
Definitions

For $N$ greater than some constant, we have the following definitions:

\[ T(N) = O(f(N)) \quad \leftarrow \quad T(N) \leq cf(N) \]

\[ T(N) = \Omega(g(N)) \quad \leftarrow \quad T(N) \geq cf(N) \]

\[ T(N) = \Theta(h(N)) \quad \leftarrow \quad T(N) = O(h(N)), \quad T(N) = \Omega(h(N)) \]

There exists some constant $c$ such that $cf(N)$ bounds $T(N)$
Definitions

Alternately, \( O(f(N)) \) can be thought of as meaning

\[
T(N) = O(f(N)) \iff \lim_{N \to \infty} f(N) \geq \lim_{N \to \infty} T(N)
\]

Big-Oh notation is also referred to as asymptotic analysis, for this reason.
Comparing Growth Rates

\[ T_1(N) = O(f(N)) \text{ and } T_2(N) = O(g(N)) \]

then

(a) \[ T_1(N) + T_2(N) = O(f(N) + g(N)) \]
(b) \[ T_1(N)T_2(N) = O(f(N)g(N)) \]

※ If you have to, use l’Hôpital’s rule

\[ \lim_{N \to \infty} \frac{f(N)}{g(N)} = \lim_{N \to \infty} \frac{f'(N)}{g'(N)} \]
Abstract Data Types

- Defined by:
  - What information it stores
  - How the information is organized
  - How the information can be accessed

- Doesn’t specify implementation
## Tradeoffs

<table>
<thead>
<tr>
<th></th>
<th>insert</th>
<th>remove</th>
<th>lookup</th>
<th>index</th>
</tr>
</thead>
<tbody>
<tr>
<td>ArrayList</td>
<td>$O(N)$</td>
<td>$O(N)$</td>
<td>$O(N)$</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>LinkedList</td>
<td>$O(1)$</td>
<td>$O(1)$</td>
<td>$O(N)$</td>
<td>$O(N)$</td>
</tr>
<tr>
<td>Stack/Queue</td>
<td>$O(1)$</td>
<td>$O(1)$</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>BST</td>
<td>$O(d)=O(N)$</td>
<td>$O(d)=O(N)$</td>
<td>$O(d)=O(N)$</td>
<td>N/A</td>
</tr>
<tr>
<td>AVL</td>
<td>$O(\log N)$</td>
<td>$O(\log N)$</td>
<td>$O(\log N)$</td>
<td>N/A</td>
</tr>
</tbody>
</table>

There may not be free lunch, but sometimes there’s a cheaper lunch.
Abstract Data Type: Lists

- An ordered series of objects
- Each object has a previous and next
  - Except first has no previous, last has no next
- We can insert an object to a list (at location $k$)
- We can remove an object from a list
- We can read an object from a list (location $k$)
Array Implementation of Lists

1st Hurdle: arrays have sizes
- Create bigger array when we run out of space, copy old array to big array

2nd Hurdle: Inserting object anywhere but the end
- Shift all entries forward one. O(N)

Get kth and insertion to end constant time O(1)
<table>
<thead>
<tr>
<th><strong>Linked Lists</strong></th>
<th><strong>Array Lists</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>No additional penalty on size</td>
<td>Need to estimate size/grow array</td>
</tr>
<tr>
<td>Insert/remove $O(1)$*</td>
<td>Insert/remove $O(N)$*</td>
</tr>
<tr>
<td>get kth costs $O(N)$*</td>
<td>get kth costs $O(1)$</td>
</tr>
<tr>
<td>Need some extra memory for links</td>
<td>Arrays are compact in memory</td>
</tr>
</tbody>
</table>
Stack Definition

- Essentially a very restricted List
- Two (main) operations:
  - Push(AnyType x)
  - Pop(AnyType x)
- Analogy – Cafeteria Trays, PEZ
Stack Implementations

- **Linked List:**
  - Push(x) <-> add(x,0)
  - Pop(x)  <->  remove(0)

- **Array:**
  - Push(x) <-> Array[k++] = x
  - Pop(x)  <->  return Array[--k]
Queues

- Stacks are Last In First Out
- Queues are First In First Out, first-come first-served
- Operations: enqueue and dequeue
Queue Implementation

- Linked List
  - add(x,0) to enqueue, remove(N-1) to dequeue

- Array List won’t work well!
  - add(x,0) is expensive
  - Solution: use a circular array
Circular Array Queue

- Don’t bother shifting after removing from array list
- Keep track of start and end of queue
- When run out of space, wrap around
  - modular arithmetic
- When array is full, increase size using list tactic
Trees

- Extension of Linked List structure:
  - Each node connects to multiple nodes
- Examples include file systems, Java class hierarchies
Tree Terminology

- Just like Lists, Trees are collections of nodes
- Conceptualize trees upside down (like family trees)
  - the top node is the root
  - nodes are connected by edges
  - edges define parent and child nodes
  - nodes with no children are called leaves
More Tree Terminology

- Nodes that share the same parent are **siblings**
- A **path** is a sequence of nodes such that the next node in the sequence is a child of the previous
- A node’s **depth** is the length of the path from root
- The **height** of a tree is the maximum depth
- If a path exists between two nodes, one is an **ancestor** and the other is a **descendant**
Tree Traversals

- Suppose we want to print all the nodes in a tree
- What order should we visit the nodes?
  - **Preorder** - read the parent before its children
  - **Postorder** - read the parent after its children
Preorder vs. Postorder

- preorder(node x)
  print(x)
  for child : Children
  preorder(child)

- postorder(node x)
  for child : Children
  postorder(child)
  print(x)
Binary Trees

- Nodes can only have two children:
  - left child and right child
- Simplifies implementation and logic
- Provides new inorder traversal
Inorder Traversal

- Read left child, then parent, then right child
- Essentially scans *whole* tree from left to right

```python
inorder(node x)
inorder(x.left)
print(x)
inorder(x.right)
```
Binary Tree Properties

- A binary tree is **full** if each node has 2 or 0 children.
- A binary tree is **perfect** if it is full and each leaf is at the same depth.
- That depth is $O(\log N)$.
Search (Tree) ADT

- ADT that allows insertion, removal, and searching by **key**
- A **key** is a value that can be compared
- In Java, we use the **Comparable** interface
- Comparison must obey transitive property
- Notice that the Search ADT doesn’t use any index
Inserting into a BST

- `insert(x)` calls `insert(x, root)`

- Recursive concept:

  - `insert(x, t)`
  
    - if (`x > t.key`)
      
        `insert(x, t.right)`
      
    - elseif (`x < t.key`)
      
        `insert(x, t.left)`

- Actual code needs to manage links/null etc
Searching a BST

- **findMin(t)**
  
  ```java
  if (t.left == null) return t.key
  else return findMin(t.left)
  ```

- **contains(x,t)**

  ```java
  if (t == null) return false
  if (x == t.key) return true
  if (x > t.key), then return contains(x, t.right)
  if (x < t.key), then return contains(x, t.left)
  ```
Deleting from a BST

- Removing a leaf is easy, removing a node with one child is also easy.
- Nodes with no grandchildren are easy.
- Nodes with both children and grandchildren need more thought.
- Why can’t we replace the removed node with either of its children?
A Removal Strategy

- First, find node to be removed, t
- Replace with the smallest node from the right subtree
  
  - \( a = \text{findMin}(t.\text{right}); \)
  - \( t.\text{key} = a.\text{key}; \)
- Then delete original smallest node in right subtree
  
  \( \text{remove}(a.\text{key}, t.\text{right}) \)
AVL Trees

- Motivation: want height of tree to be close to log N
- AVL Tree Property:
  For each node, all keys in its left subtree are less than the node’s and all keys in its right subtree are greater. Furthermore, the height of the left and right subtrees differ by at most 1
AVL Tree Visual
Tree Rotations

- To balance the tree after an insertion violates the AVL property,
- rearrange the tree; make a new node the root.
- This rearrangement is called a rotation.
- There are 2 types of rotations.
AVL Tree Visual: Before insert
AVL Tree Visual:
After insert
AVL Tree Visual: Single Rotation
AVL Tree
Single Rotation

- Works when new node is added to outer subtree (left-left or right-right)
- What about inner subtrees? (left-right or right-left)
AVL Tree Visual:
Before Insert 2
AVL Tree Visual: After Insert 2
AVL Tree Visual: Single Rotation Fails
AVL Tree Visual: Double Rotation
AVL Tree Visual: Double Rotation
Splay Trees

- Like AVL trees, use the standard binary search tree property
- After any operation on a node, make that node the new root of the tree
- Make the node the root by repeating one of two moves that make the tree more spread out
Easy cases

- If node is root, do nothing
- If node is child of root, do single AVL rotation
- Otherwise, node has a grandparent, and there are two cases
Case 1: zig-zag

- Use when the node is the right child of a left child (or left-right)
- Double rotate, just like AVL tree
Case 2: zig-zig

- Use when node is the right-right child (or left-left)
- Reverse the order of grandparent->parent->node
- Make it node->parent->grandparent
Priority Queues

- New abstract data type Priority Queue:
  - Insert: add node with key
  - deleteMin: delete the node with smallest key
  - (increase/decrease priority)
Heap Implementation

- Priority queues are most commonly implemented using Binary Heaps
- Binary tree with special properties
- Heap Structure Property: all nodes are full, (except possibly one at the bottom level)
- Heap Order Property: any node is smaller than its children
Array Implementation

- A full tree is regular: we can easily store in an array
  - Root at A[1]
  - Node i has children at 2i and (2i+1)
  - Parent at floor(i/2)
- No links necessary, so faster (in most languages)
Insert

- To insert key $X$, create a hole in bottom level
- Percolate up
  - Is hole’s parent is less than $X$
    - If so, put $X$ in hole, heap order satisfied
    - If not, swap hole and parent and repeat
DeleteMin

- Save root node, and delete, creating a hole
- Take the last element in the heap X
- **Percolate down:**
  - Check if X is less than hole’s children
    - if so, we’re done
    - if not, swap hole and smallest child and repeat
Building a Heap from an Array

How do we construct a binary heap from an array?

Simple solution: insert each entry one at a time

Each insert is worst case $O(\log N)$, so creating a heap in this way is $O(N \log N)$

Instead, we can jam the entries into a full binary tree and run percolateDown intelligently
buildHeap

- Start at deepest non-leaf node
  - in array, this is node N/2
- percolateDown on all nodes in reverse level-order
  - for i = N/2 to 1
    percolateDown(i)