CSE 373

APRIL 10TH – DICTIONARY ADT
ASSORTED MINUTIAE

• HW2 due Wednesday at Midnight
TODAY'S SCHEDULE

• Floyd's Algorithm examples
• Correctness proof
• Dictionary ADT
void buildHeap() {
    for (i = size/2; i > 0; i--) {
        val = arr[i];
        percolateDown(i, val);
        arr[hole] = val;
    }
}
FLOYD’S METHOD

void buildHeap() {
    for(i = size/2; i>0; i--) {
        val = arr[i];
        percolateDown(i,val);
        arr[hole] = val;
    }
}

• Review: what does this do?
  • Size/2 – only nodes with children
  • Percolate down each of those nodes
  • How does this percolate down work?
EXAMPLE

Build a heap with the values: 12, 5, 11, 3, 10, 2, 9, 4, 8, 1, 7, 6

Stick them all in the tree to make a valid structure

In tree form for readability. Notice:

- **Purple** for node values to fix (heap-order problem)
- Notice no leaves are purple
- Check/fix each non-leaf bottom-up (6 steps here)
Purple shows the nodes that will need to be fixed.

We don’t know which ones they are yet, so we’ll traverse bottom up one level at a time and fix all the values.

Values to consider on each level circled in blue.
Step 1

Happens to already be less than it’s child
EXAMPLE

Percolate down (notice that moves 1 up)
EXAMPLE

Another nothing-to-do step
EXAMPLE

Percolate down as necessary (steps 4a and 4b)
EXAMPLE

Step 5
EXAMPLE

Step 6
CORRECTNESS

void buildHeap() {
    for (i = size/2; i > 0; i--) {
        val = arr[i];
        percolateDown(i, val);
        arr[hole] = val;
    }
}

• How do we prove this works?
  • Use inductive proof
    • Base case
      • The heap property is maintained for all elements after size/2 because they have no children
    • Step
      • When adding each element, the algorithm puts it into the right spot
CORRECTNESS

void buildHeap() {
    for (i = size/2; i > 0; i--) {
        val = arr[i];
        percolateDown(i, val);
        arr[hole] = val;
    }
}

- For all elements after i, the heap property should be preserved
  - This is why we can start at size/2
- percolateDown() ensures that each new element goes to the right place
- Once a loop has gotten to a node, the smallest elements are at the top of their subtrees.
LESSONS FROM BUILDHEAP

Without buildHeap, our ADT already let clients implement their own in $O(n \log n)$ worst case

- Worst case is inserting better priority values later

By providing a specialized operation internal to the data structure (with access to the internal data), we can do $O(n)$ worst case

- Intuition: Most data is near a leaf, so better to percolate down

Can analyze this algorithm for:

- Correctness and Efficiency:
  - First analysis easily proved it was $O(n \log n)$
  - Tighter analysis shows same algorithm is $O(n)$
LESSONS FROM BUILDHEAP

- Should all priority queues support buildHeap()?  
  - No downside to implementation  
  - Faster than $O(n \log n)$ naïve approach  
  - Not required for HW 2, but is commonly implemented
HEAPS

• What to know
  • How to implement all functions
  • How to analyze all functions
  • Understand the benefits of array implementation
  • Types of client problems
    • Hospitals, server scheduling, etc…
DICTIONARY ADT

- New abstract data type
  - Dictionary (aka Map)
  - Data – Key and Value pairs
    - Keys: must be comparable, used for lookup
    - Values: the actual data itself
  - Example (Store inventory):
    - Keys: IDs (barcodes)
    - Values: Product information
DICTIONARY ADT

• Operations

• `insert(key, value)`: inserts the key, value pair into the dictionary

• `find(key)`: returns the stored value for a particular key in the dictionary, returns null if not found.

• `delete(key)`: removes the key value pair specified by the given key from the dictionary. In this course you may assume unique keys.
SET ADT

• Slightly different from Dictionary
• No values, the set only cares if a key is present or not
• Find, insert and delete have few differences
• Possible to implement other functions from sets
  • Union, intersection, difference
APPLICATIONS

- Store information in key, value pairs
  - Very common usage pattern
    - Phone directories
    - Indexing
    - OS page tables
    - Databases
IMPLEMENTATIONS

• Important to allow fast operations over the keys
  • Dependent on what the client uses most
  • Could be many lookups and few inserts

• Keys and Values should be stored together in some way
  • Both objects in one node
  • Paired arrays (one stores keys and the other values)
## IMPLEMENTATIONS

- **Simple implementations**

<table>
<thead>
<tr>
<th>Data Structure</th>
<th>Insert</th>
<th>Find</th>
<th>Delete</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unsorted linked-list</td>
<td>$O(1)^*$</td>
<td>$O(n)$</td>
<td>$O(n)$</td>
</tr>
<tr>
<td>Unsorted array</td>
<td>$O(1)^*$</td>
<td>$O(n)$</td>
<td>$O(n)$</td>
</tr>
<tr>
<td>Sorted linked list</td>
<td>$O(n)$</td>
<td>$O(n)$</td>
<td>$O(n)$</td>
</tr>
<tr>
<td>Sorted array</td>
<td>$O(n)$</td>
<td>$O(\log n)$</td>
<td>$O(n)$</td>
</tr>
</tbody>
</table>

* Unless we need to check for duplicates
IMPLEMENTATIONS

• Other implementations
  • Binary Search Trees
  • Hashtables
NEXT CLASS

• Trees and traversals
• BST Dictionaries
• Analysis and tree balance