Announcements

• Homework 2 due today
• Homework 3 out today (due July 22\textsuperscript{nd}) 😊
• Midterm next Friday

• Today
  – AVL Tree Review
  – Priority Queues
  – Min Heaps
The general right-left case
The general right-left case

Before we added the node, the tree was balanced. . .

If V (and U) were of height h, would the tree be balanced here?
The general right-left case

The actual value of \( h \) can be anything, we only care about the relative heights of the subtrees...

These two trees are equivalent, we just redefined \( h \)
The general right-left case
Insert, summarized

- Insert as in a BST
- Check back up path for imbalance, which will be 1 of 4 cases:
  - Node’s left-left grandchild is too tall
  - Node’s left-right grandchild is too tall
  - Node’s right-left grandchild is too tall
  - Node’s right-right grandchild is too tall
- Only one case occurs because tree was balanced before insert
- After the appropriate single or double rotation, the smallest-unbalanced subtree has the same height as before the insertion
  - So all ancestors are now balanced
AVL Trees efficiency

- Worst-case complexity of find: $O(\log n)$
  - Tree is balanced

- Worst-case complexity of insert: $O(\log n)$
  - Tree starts balanced
  - A rotation is $O(1)$ and there’s an $O(\log n)$ path to root
  - Tree ends balanced

- Worst-case complexity of buildTree: $O(n \log n)$

Takes some more rotation action to handle delete…
Pros and Cons of AVL Trees

Arguments for AVL trees:

1. All operations logarithmic worst-case because trees are *always* balanced
2. Height balancing adds no more than a constant factor to the speed of *insert* and *delete*

Arguments against AVL trees:

1. Difficult to program & debug [but done once in a library!]
2. More space for height field
3. Asymptotically faster but rebalancing takes a little time
4. If *amortized* (later, I promise) logarithmic time is enough, use splay trees (also in the text)
Done with AVL Trees (....phew!)

next up...

Priority Queues ADT
(Homework 3 😊)
A new ADT: Priority Queue

- A priority queue holds compare-able data
  - Like dictionaries, we need to compare items
    - Given $x$ and $y$, is $x$ less than, equal to, or greater than $y$
    - Meaning of the ordering can depend on your data
  - Integers are comparable, so will use them in examples
    - But the priority queue ADT is much more general
    - Typically two fields, the priority and the data
Priorities

• Each item has a “priority”
  – In our examples, the lesser item is the one with the greater priority
  – So “priority 1” is more important than “priority 4”
  – (Just a convention, think “first is best”)

• Operations:
  – insert
  – deleteMin
  – is_empty

• Key property: deleteMin returns and deletes the item with greatest priority (lowest priority value)
  – Can resolve ties arbitrarily
Example

insert \( x_1 \) with priority 5
insert \( x_2 \) with priority 3
\( a = \text{deleteMin} \ // \ x_2 \)
insert \( x_3 \) with priority 2
insert \( x_4 \) with priority 6
\( c = \text{deleteMin} \ // \ x_3 \)
\( d = \text{deleteMin} \ // \ x_1 \)

- Analogy: insert is like enqueue, deleteMin is like dequeue
  - But the whole point is to use priorities instead of FIFO
Applications

Like all good ADTs, the priority queue arises often

- Run multiple programs in the operating system
  - “critical” before “interactive” before “compute-intensive”
- Treat hospital patients in order of severity (or triage)
- Forward network packets in order of urgency
- Select most frequent symbols for data compression
- Sort (first \texttt{insert} all, then repeatedly \texttt{deleteMin})
  - Much like Homework 1 uses a stack to implement reverse
Finding a good data structure

- Will show an efficient, non-obvious data structure for this ADT
  - But first let’s analyze some “obvious” ideas for $n$ data items

<table>
<thead>
<tr>
<th>data</th>
<th>insert algorithm / time</th>
<th>deleteMin algorithm / time</th>
</tr>
</thead>
<tbody>
<tr>
<td>unsorted array</td>
<td>add at end</td>
<td>search</td>
</tr>
<tr>
<td>unsorted linked list</td>
<td>add at front</td>
<td>search</td>
</tr>
<tr>
<td>sorted circular array</td>
<td>search / shift</td>
<td>move front</td>
</tr>
<tr>
<td>sorted linked list</td>
<td>put in right place</td>
<td>remove at front</td>
</tr>
<tr>
<td>binary search tree</td>
<td>put in right place</td>
<td>leftmost</td>
</tr>
<tr>
<td>AVL tree</td>
<td>put in right place</td>
<td>leftmost</td>
</tr>
</tbody>
</table>
Our data structure

A binary min-heap (or just binary heap or just heap) has:

- **Structure property**: A complete binary tree
- **Heap property**: The priority of every (non-root) node is less important than the priority of its parent

  - *Not a binary search tree*

So:

- Where is the highest-priority item?
- What is the height of a heap with $n$ items?
Operations: basic idea

• **findMin**: return root.data
• **deleteMin**:
  1. answer = root.data
  2. Move right-most node in last row to root to restore structure property
  3. “Percolate down” to restore heap property
• **insert**:
  1. Put new node in next position on bottom row to restore structure property
  2. “Percolate up” to restore heap property

**Overall strategy:**
• *Preserve structure property*
• *Break and restore heap property*
DeleteMin

Delete (and later return) value at root node
DeleteMin: Keep the Structure Property

- We now have a “hole” at the root
  - Replace it with another node
- Want to keep structure property
- Pick the last node on the bottom row of the tree and move it to the “hole”
DeleteMin: Restore the Heap Property

Percolate down:
• Compare priority of item with children
• If priority is less important, swap with the most important child and repeat
• Done if both children are less important than the item or we’ve reached a leaf node

What is the run time?
DeleteMin: Run Time Analysis

- Run time is $O(\text{height of heap})$
- A heap is a complete binary tree
- Height of a complete binary tree of $n$ nodes?
  - $\text{height} = \lceil \log_2(n) \rceil$
- Run time of `deleteMin` is $O(\log n)$
**Insert**

- Add a value to the tree
- Afterwards, structure and heap properties must still be correct
Insert: Maintain the Structure Property

• There is only one valid tree shape after we add one more node

• So put our new data there and then focus on restoring the heap property
**Insert: Restore the heap property**

**Percolate up:**
- Put new data in new location
- If parent is less important, swap with parent, and continue
- Done if parent is more important than item or reached root

What is the running time?
*Like deleteMin, worst-case time proportional to tree height: $O(\log n)$*
Binary Heap

- Operations
  - $O(\log n)$ insert
  - $O(\log n)$ deleteMin worst-case
  - Very good constant factors
  - If items arrive in random order, then insert is $O(1)$ on average
    - Because approx. 75% of nodes in bottom two rows
Summary

• **Priority Queue ADT:**
  - `insert` comparable object,
  - `deleteMin`

• **Binary heap data structure:**
  - Complete binary tree
  - Each node has less important priority value than its parent

• `insert` and `deleteMin` operations = $O(\text{height-of-tree}) = O(\log n)$
  - `insert`: put at new last position in tree and percolate-up
  - `deleteMin`: remove root, put last element at root and percolate-down
Array Representation of Binary Trees

From node $i$:
- Left child: $i \times 2$
- Right child: $i \times 2 + 1$
- Parent: $i / 2$

(wasting index 0 is convenient for the index arithmetic)

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>
<tr>
<td>7</td>
<td>13</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Judging the array implementation

Pros:
• Non-data space: just index 0 and unused space on right
  – In conventional tree representation, one edge per node (except for root), so \( n-1 \) wasted space (like linked lists)
  – Array would waste more space if tree were not complete
• Multiplying and dividing by 2 is very fast (shift operations in hardware)
• Last used position is just index \( \text{size} \)

Cons:
• Same might-be-empty or might-get-full problems we saw with stacks and queues (resize by doubling as necessary)

Pros outweigh cons: min-heaps almost always use array implementation
void insert(int val) {
    if (size == arr.length-1)
        resize();
    size++;
    i = percolateUp(size, val);
    arr[i] = val;
}

int percolateUp(int hole, int val) {
    while (hole > 1 &&
           val < arr[hole/2])
        arr[hole] = arr[hole/2];
    hole = hole / 2;
}

return hole;

This pseudocode uses ints. In real use, you will have data nodes with priorities.
### Pseudocode: deleteMin from binary heap

```cpp
int deleteMin() {
    if(isEmpty()) throw...
    ans = arr[1];
    hole = percolateDown
    (1,arr[size]);
    arr[hole] = arr[size];
    size--;
    return ans;
}

int percolateDown(int hole, int val) {
    while(2*hole <= size) {
        left  = 2*hole;
        right = left + 1;
        if(right > size ||
            arr[left] < arr[right])
            target = left;
        else
            target = right;
        if(arr[target] < val) {
            arr[hole] = arr[target];
            hole = target;
        } else
            break;
    }
    return hole;
}
```

![Binary heap diagram](image)
Example

1. insert: 16, 32, 4, 67, 105, 43, 2
2. deleteMin
Example

1. insert: 16, 32, 4, 67, 105, 43, 2
2. deleteMin
Example

1. insert: 16, 32, 4, 67, 105, 43, 2
2. deleteMin

```
   16
   / \
  32   0
  / \  / \
16 32 0
```

```
+---+---+---+---+---+---+---+
|   | 16 | 32 |   |   |   |   |
+---+---+---+---+---+---+---+
  0  1  2  3  4  5  6  7
```
Example

1. insert: 16, 32, 4, 67, 105, 43, 2
2. deleteMin
Example

1. insert: 16, 32, 4, 67, 105, 43, 2
2. deleteMin

```
4   32   16   67
0   1   2   3   4   5   6   7
```

```
4
/ \
/   /\n32  16
/   /\n67  43 2
```
Example

1. insert: 16, 32, 4, 67, 105, 43, 2
2. deleteMin

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>32</td>
<td>16</td>
<td>67</td>
<td>105</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

```
       4
      / \
     32 16
    /   / \
   67 105
```

```
Example

1. insert: 16, 32, 4, 67, 105, 43, 2
2. deleteMin

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>32</td>
<td>16</td>
<td>67</td>
<td>105</td>
<td>43</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Diagram of a binary search tree with values from 16 to 105.
Example

1. insert: 16, 32, 4, 67, 105, 43, 2
2. deleteMin
Example

1. insert: 16, 32, 4, 67, 105, 43, 2
2. deleteMin
Example

1. insert: 16, 32, 4, 67, 105, 43, 2
2. deleteMin
Example

1. insert: 16, 32, 4, 67, 105, 43, 2
2. deleteMin

<table>
<thead>
<tr>
<th></th>
<th>4</th>
<th>32</th>
<th>16</th>
<th>67</th>
<th>105</th>
<th>43</th>
<th></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>
</tr>
</tbody>
</table>

```
4
32
32
67
105
43
16
```

```
4
32
32
67
105
43
16
```
Other operations

- **decreaseKey**: given pointer to object in priority queue (e.g., its array index), lower its priority value by $p$
  - Change priority and percolate up

- **increaseKey**: given pointer to object in priority queue (e.g., its array index), raise its priority value by $p$
  - Change priority and percolate down

- **remove**: given pointer to object in priority queue (e.g., its array index), remove it from the queue
  - **decreaseKey** with $p = \infty$, then deleteMin

Running time for all these operations?
**Build Heap**

- Suppose you have \( n \) items to put in a new (empty) priority queue
  - Call this operation `buildHeap`

- \( n \) inserts
  - Only choice if ADT doesn’t provide `buildHeap` explicitly
  - \( O(n \log n) \)

- Why would an ADT provide this unnecessary operation?
  - Convenience
  - Efficiency: an \( O(n) \) algorithm called Floyd’s Method
  - Common issue in ADT design: how many specialized operations
Floyd’s Method

1. Use $n$ items to make any complete tree you want
   - That is, put them in array indices 1,…,n

2. Treat it as a heap and fix the heap-order property
   - Bottom-up: percolate down starting at nodes one level up from leaves, work up toward the root

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

• In tree form for readability
  – Purple for node not less than descendants
    • heap-order problem
  – Notice no leaves are purple
  – Check/fix each non-leaf bottom-up (6 steps here)
Example

- Happens to already be less than children (er, 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
But is it right?

• “Seems to work”
  – Let’s *prove* it restores the heap property (correctness)
  – Then let’s *prove* its running time (efficiency)

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

Loop Invariant: For all \( j > i \), \( \text{arr}[j] \) is less than its children

- True initially: If \( j > \text{size}/2 \), then \( j \) is a leaf
  - Otherwise its left child would be at position \( > \text{size} \)
- True after one more iteration: loop body and \text{percolateDown} make \( \text{arr}[i] \) less than children without breaking the property for any descendants

So after the loop finishes, all nodes are less than their children
Efficiency

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

Easy argument: buildHeap is $O(n \log n)$ where $n$ is size

- $\frac{\text{size}}{2}$ loop iterations
- Each iteration does one percolateDown, each is $O(\log n)$

This is correct, but there is a more precise ("tighter") analysis of the algorithm…
Better argument: `buildHeap` is $O(n)$ where $n$ is `size`

- `size/2` total loop iterations: $O(n)$
- 1/2 the loop iterations percolate at most 1 step
- 1/4 the loop iterations percolate at most 2 steps
- 1/8 the loop iterations percolate at most 3 steps
- ...
- $\left(\frac{1}{2} + \frac{2}{4} + \frac{3}{8} + \frac{4}{16} + \ldots\right) < 2$ (page 4 of Weiss)
  - So at most $2 \times \left(\frac{\text{size}}{2}\right)$ total percolate steps: $O(n)$
Lessons from `buildHeap`

- Without `buildHeap`, clients can implement their own in $O(n \log n)$ worst case

- By providing a specialized operation (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:
    - Non-trivial inductive proof using loop invariant
  - Efficiency:
    - First analysis easily proved it was $O(n \log n)$
    - Tighter analysis shows same algorithm is $O(n)$
Other branching factors

- $d$-heaps: have $d$ children instead of 2
  - Makes heaps shallower

- Homework: Implement a 3-heap
  - Just have three children instead of 2
  - Still use an array with all positions from 1...heap-size used

<table>
<thead>
<tr>
<th>Index</th>
<th>Children Indices</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2,3,4</td>
</tr>
<tr>
<td>2</td>
<td>5,6,7</td>
</tr>
<tr>
<td>3</td>
<td>8,9,10</td>
</tr>
<tr>
<td>4</td>
<td>11,12,13</td>
</tr>
<tr>
<td>5</td>
<td>14,15,16</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>