CSE 373: Data Structures and Algorithms

Lecture 11: Finish AVL Trees; Priority Queues; Binary Heaps

Instructor: Lilian de Greef
Quarter: Summer 2017
Today

• Announcements
• Finish AVL Trees
• Priority Queues
• Binary Heaps
Announcements

• Changes to Office Hours: from now on...
  • **No** more Wednesday morning & **shorter** Wednesday afternoon office hours
  • **New** Thursday office hours!
    • Kyle’s Wednesday hour is now 1:30-**2:30**pm
    • Ben’s office hours is now Thursday **1:00-3:00**pm

• AVL Tree Height
  • In section, computed that minimum # nodes in AVL tree of a certain height is
    \[ S(h) = 1 + S(h-1) + S(h-2) \]
  • where \( h \) = height of tree
  • Posted a proof next to these lecture slides online for your perusal
Announcements

• Midterm
  • Next Friday! (at usual class time & location)
  • Everything we cover in class until exam date is fair game (minus clearly-marked “bonus material”). That includes next week’s material!
  • Today’s hw3 due date designed to give you time to study.

• Course Feedback
  • Heads up: official UW-mediated course feedback session for part of Wednesday
  • Also want to better understand an anonymous concern on course pacing → Poll
The **AVL Tree** Data Structure

An **AVL tree** is a *self-balancing* binary search tree.

**Structural properties**

1. Binary tree property (same as BST)
2. Order property (same as for BST)
3. Balance condition:
   balance of every node is between -1 and 1
   
   where \( \text{balance}(node) = \text{height}(node.\text{left}) - \text{height}(node.\text{right}) \)

Result: **Worst-case** depth is \( O(\log n) \)
Single Rotations

(Figures by Melissa O’Neill, reprinted with her permission to Lilian)
Case #3: Right-Left Case

(Figures by Melissa O’Neill, reprinted with her permission to Lilian)
Case #3: Right-Left Case (after one rotation)

(Figures by Melissa O’Neill, reprinted with her permission to Lilian)
Case #3: Right-Left Case (after two rotations)

A way to remember it:
Move d to grandparent’s position. Put everything else in their only legal positions for a BST.

(Figures by Melissa O’Neill, reprinted with her permission to Lilian)
Practice time! Example of Case #4

Starting with this AVL tree:

```
  60
 /   \
30    80
|     |
10    50
```

Which of the following is the updated AVL tree after inserting 42?
(extra space for your scratch work and notes)
Practice time! Example of Case #4

Starting with this AVL tree:

Which of the following is the updated AVL tree after inserting 42?

What’s the name of this case?  What rotations did we do?
Insert, summarized

• Insert as in our generic 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 occurs because

• 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 Tree Efficiency

• Worst-case complexity of `find`:

• Worst-case complexity of `insert`:

• Worst-case complexity of `buildTree`:

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* logarithmic time is enough, use splay trees (also in the text, not covered in this class)
Lots of cool Self-Balancing BSTs out there!

Popular self-balancing BSTs include:

- **AVL tree**
- **Splay tree**
- **2-3 tree**
- **AA tree**
- **Red-black tree**
- **Scapegoat tree**
- **Treap**

(Not covered in this class, but several are in the textbook and all of them are online!)

(From https://en.wikipedia.org/wiki/Self-balancing_binary_search_tree#Implementations)
Wrapping Up: Hash Table vs BST Dictionary

Hash Table advantages:

BST advantages:
• Can get keys in sorted order without much extra effort
• Same for ordering statistics, finding closest elements, range queries
• Easier to implement if don’t have hash function (which are hard!).
• Can guarantee $O(\log n)$ \textit{worst-case} time, whereas hash tables are $O(1)$ \textit{average} time.
Priority Queue ADT & Binary Heap data structure

Like a Queue, but with priorities for each element.
An Introductory Example...

Gill Bates, the CEO of the software company Millisoft, built a robot secretary to manage her hundreds of emails. During the day, Bates only wants to look at a few emails every now and then so she can stay focused. The robot secretary sends her the most important emails each time. To do so, he assigns each email a priority when he gets them and only sends Bates the highest priority emails upon request.

All of your computer servers are on fire!  
Priority:  

Here’s a reminder for our meeting in 2 months.  
Priority:
Priority Queue ADT

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

In our introductory example:

• The data typically has two fields:
  • We’ll now use integers for examples, but can use other types /objects for priorities too!
Priority Queue ADT

Meaning:
  • A priority queue holds compare-able data
  • Key property:

Operations:
Priority Queue: Example

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

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 insert all, then repeatedly 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:**

• **Heap property:** The priority of every (non-root) node is less important than the priority of its parent

So:

• Where is the highest-priority item?
• Where is the lowest priority?
• What is the height of a heap with \( n \) items?
deleteMin: Step #1
deleteMin: Step #2 (Keep Structure Property)

Want to keep structure property
deleteMin: Step #3

Want to restore heap property
deleteMin: Step #3 (Restore Heap Property)

**Percolate down:**
- Compare priority of item with its children
- If item has lower priority, swap with the most important child
- Repeat until both children have lower priority or we’ve reached a leaf node

What is the run time?
deleteMin: Run Time Analysis

- Run time is
- A heap is a
- So its height with \( n \) nodes is
- So run time of deleteMin is
insert: Step #1
insert: Step #2
**insert**: Step #2 (Restore Heap Property)

**Percolate up:**
- Put new data in new location
- If higher priority than parent, swap with parent
- Repeat until parent is more important or reached root

What is the running time?
Summary: basic idea for operations

**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:**
1. *Preserve structure property*
2. *Restore heap property*
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
Summary: Priority Queue ADT

• **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
Binary Trees Implemented with an **Array**

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)
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

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

Pros outweigh cons: min-heaps almost always use array implementation
Practice time!
Starting with an empty array-based binary heap, which is the result after
1. insert (in this order): 16, 32, 4, 67, 105, 43, 2
2. deleteMin once

A)  

<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>15</td>
<td>32</td>
<td>43</td>
<td>67</td>
<td>105</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

B)  

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

C)  

<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>43</td>
<td>105</td>
<td>67</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

D)  

<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>
(extra space for your scratch work and notes)
Semi-Pseudocode: `insert` into binary heap

```java
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.
**Semi-Pseudocode:** \texttt{deleteMin} from binary heap

```c
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;
}
```
Example

2. *deleteMin* once

```
   0  1  2  3  4  5  6  7
```
Example

1. insert (in this order): 16, 32, 4, 67, 105, 43, 2
2. deleteMin once
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?