Administrivia

Midterm grades out later this week
HW 4 due Wednesday night
HW 5 out Wednesday (partner project)
Heaps
Priority Queue ADT

Imagine you have a collection of data from which you will always ask for the extreme value.

If a Queue is “First-In-First-Out” (FIFO) Priority Queues are “Most-Important-Out-First”
Example: Triage, patient in most danger is treated first

Items in Queue must be comparable, Queue manages internal sorting

Min Priority Queue ADT

state
- Set of comparable values
  - Ordered based on “priority”

behavior
- removeMin() – returns the element with the smallest priority, removes it from the collection
- peekMin() – find, but do not remove the element with the smallest priority
- insert(value) – add a new element to the collection

Max Priority Queue ADT

state
- Set of comparable values
  - Ordered based on “priority”

behavior
- removeMax() – returns the element with the largest priority, removes it from the collection
- peekMax() – find, but do not remove the element with the largest priority
- insert(value) – add a new element to the collection
Let’s start with an AVL tree

What is the worst case for `peekMin()`? \(O(\log n)\)

What is the best case for `peekMin()`? \(O(\log n)\)

Can we do something to guarantee best case for `removeMin()` and `peekMin()`?
Binary Heap

A type of tree with new set of invariants

1. **Binary Tree**: every node has at most 2 children

2. **Heap**: every node is smaller than its child

3. **Structure**: Each level is “complete” meaning it has no “gaps”
   - Heaps are filled up left to right
Self Check - Are these valid heaps?

Binary Heap Invariants:
1. Binary Tree
2. Heap
3. Complete

[Diagram showing three trees, with two marked as invalid and one as valid]
Implementing peekMin()

Runtime: O(1)
Implementing `removeMin()`

1.) Return min
2.) replace with last added

Structure maintained, heap broken
Implementing removeMin() - percolateDown

3.) percolateDown()

Recursively swap parent with smallest child
Practice: removeMin()
Implementing insert()

Algorithm:
- Insert a node to ensure no gaps
- Fix heap invariant
- percolate UP
**Practice:** Building a minHeap

Construct a Min Binary Heap by inserting the following values in this order:

5, 10, 15, 20, 7, 2

---

**Min Priority Queue ADT**

**state**
Set of comparable values
- Ordered based on “priority”

**behavior**
- `removeMin()` – returns the element with the smallest priority, removes it from the collection
- `peekMin()` – find, but do not remove the element with the smallest priority
- `insert(value)` – add a new element to the collection

---

**Min Binary Heap Invariants**

1. **Binary Tree** – each node has at most 2 children
2. **Min Heap** – each node’s children are larger than itself
3. **Level Complete** - new nodes are added from left to right completely filling each level before creating a new one

---

**Diagram:**

- **2**
- **5**
- **7**
- **10**
- **15**
- **20**

**Percolate Up:**

- **2**
- **7**
- **10**
- **15**
- **20**
minHeap runtimes

removeMin():
- Find and remove minimum node
- Find last node in tree and swap to top level
- Percolate down to fix heap invariant

insert():
- Insert new node into next available spot
- Percolate up to fix heap invariant
Implementing Heaps

How do we find the minimum node?
\[ \text{peekMin}() = \text{arr}[0] \]

How do we find the last node?
\[ \text{lastNode}() = \text{arr}[\text{size} - 1] \]

How do we find the next open space?
\[ \text{openSpace}() = \text{arr}[\text{size}] \]

How do we find a node’s left child?
\[ \text{leftChild}(i) = 2i + 1 \]

How do we find a node’s right child?
\[ \text{rightChild}(i) = 2i + 2 \]

How do we find a node’s parent?
\[ \text{parent}(i) = \frac{(i - 1)}{2} \]

Fill array in level-order from left to right

<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></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>A</td>
<td>B</td>
<td>C</td>
<td>D</td>
<td>E</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>F</td>
<td>G</td>
<td>H</td>
<td>I</td>
<td>J</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>K</td>
<td>L</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></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>
Heap Implementation Runtimes

```
char peekMin()  
timeToFindMin

<table>
<thead>
<tr>
<th></th>
<th>Tree</th>
<th>Array</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Θ(1)</td>
<td>Θ(1)</td>
</tr>
</tbody>
</table>

char removeMin()  
findLastNodeTime + removeRootTime + numSwaps * swapTime

<table>
<thead>
<tr>
<th></th>
<th>Tree</th>
<th>Array</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Θ(n)</td>
<td>Θ(log(n))</td>
</tr>
</tbody>
</table>

void insert(char)  
findNextSpace + addValue + numSwaps * swapTime

<table>
<thead>
<tr>
<th></th>
<th>Tree</th>
<th>Array</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Θ(n)</td>
<td>Θ(log(n))</td>
</tr>
</tbody>
</table>
```
Building a Heap

Insert has a runtime of $\Theta(\log(n))$

If we want to insert $n$ items...

Building a tree takes $O(n\log(n))$
- Add a node, fix the heap, add a node, fix the heap

Can we do better?
- Add all nodes, fix heap all at once!
Clever building a heap – Floyd’s Method

Facts of binary trees
- Increasing the height by one level doubles the number of possible nodes
- A complete binary tree has half of its nodes in the leaves
- A new piece of data is much more likely to have to percolate down to the bottom than be the smallest element in heap

1. Dump all the new values into the bottom of the tree
   - Back of the array

2. Traverse the tree from bottom to top
   - Reverse order in the array

3. Percolate Down each level moving towards overall root
Floyd’s buildHeap algorithm

Build a tree with the values:
12, 5, 11, 3, 10, 2, 9, 4, 8, 15, 7, 6

1. Add all values to back of array
2. percolateDown(parent) starting at last index
Floyd’s buildHeap algorithm

Build a tree with the values:
12, 5, 11, 3, 10, 2, 9, 4, 8, 15, 7, 6

1. Add all values to back of array
2. percolateDown(parent) starting at last index
   1. percolateDown level 4
   2. percolateDown level 3
Floyd’s buildHeap algorithm

Build a tree with the values:
12, 5, 11, 3, 10, 2, 9, 4, 8, 15, 7, 6

1. Add all values to back of array
2. percolateDown(parent) starting at last index
   1. percolateDown level 4
   2. percolateDown level 3
   3. percolateDown level 2

keep percolating down like normal here and swap 5 and 4
Floyd’s buildHeap algorithm

Build a tree with the values:
12, 5, 11, 3, 10, 2, 9, 4, 8, 15, 7, 6

1. Add all values to back of array
2. percolateDown(parent) starting at last index
   1. percolateDown level 4
   2. percolateDown level 3
   3. percolateDown level 2
   4. percolateDown level 1
Floyd’s buildHeap algorithm

Build a tree with the values:
12, 5, 11, 3, 10, 2, 9, 4, 8, 15, 7, 6

1. Add all values to back of array
2. percolateDown(parent) starting at last index
   1. percolateDown level 4
   2. percolateDown level 3
   3. percolateDown level 2
   4. percolateDown level 1
Floyd’s Heap Runtime

We step through each node – n
We call percolateDown() on each n – log n
thus it’s O(nlogn)
... let’s look closer...

Are we sure percolateDown() runs log n each time?
- Half the nodes of the tree are leaves
  - Leaves run percolate down in constant time
- ¼ the nodes have at most 1 level to travel
- 1/8 the nodes have at most 2 levels to travel
  - etc...

work(n) \approx n/2 \times 1 + n/4 \times 2 + n/8 \times 3 + ...
Closed form Floyd’s buildHeap

\[ \text{work}(n) \approx \frac{n}{2} \cdot 1 + \frac{n}{4} \cdot 2 + \frac{n}{8} \cdot 3 + ... \]

factor out \( n \)

\[ \text{work}(n) \approx n(\frac{1}{2} + \frac{2}{4} + \frac{3}{8} + ...) \]

find a pattern \( \rightarrow \) powers of 2

\[ \text{work}(n) \approx n\left(\sum_{i=1}^{\log_2 n} \frac{i}{2^i}\right) \]

\( ? = \) how many levels = height of tree = \( \log(n) \)

Infinite geometric series

\[ \text{work}(n) \approx n\sum_{i=1}^{\log_2 n} \frac{i}{2^i} \]

\( \text{if } -1 < x < 1 \text{ then } \sum_{i=0}^{\infty} x^i = \frac{1}{1-x} = x \)

\[ \text{work}(n) \approx n\sum_{i=1}^{\log_2 n} \frac{i}{2^i} \leq n\sum_{i=0}^{\infty} \frac{i}{2^i} = n \cdot 2 \]

Floyd’s buildHeap runs in \( O(n) \) time!