CSE 373: Data Structures and Algorithms

Lecture 12: Binary Heaps

Instructor: Lilian de Greef
Quarter: Summer 2017
Today

• Announcements

• Binary Heaps
  • insert
  • delete
  • Array representation of tree
  • Floyd’s Method of buildTree
  • d-heaps
Announcements

• Midterm on Friday
  • Practice midterms on course website
  • Note that some may cover slightly different material
  • Will start at 10:50, will end promptly at 11:50 (even if you’re late), so be early

• Will have homework 3 grades back before midterm

• Reminder: course feedback session on Wednesday
Priority Queue ADT

Like a Queue, but with priorities for each element.
Priority Queue ADT

Meaning:
• A priority queue holds compare-able data
• Key property:
  deleteMin returns and deletes the item with the highest priority
  (can resolve ties arbitrarily)

Operations:
• deleteMin
• insert
• isEmpty
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 $O(1)$</td>
<td>search $O(n)$</td>
</tr>
<tr>
<td>unsorted linked list</td>
<td>add at front $O(1)$</td>
<td>search $O(n)$</td>
</tr>
<tr>
<td>sorted circular array</td>
<td>search / shift $O(n)$</td>
<td>move front $O(1)$</td>
</tr>
<tr>
<td>sorted linked list</td>
<td>put in right place $O(n)$</td>
<td>remove at front $O(1)$</td>
</tr>
<tr>
<td>binary search tree</td>
<td>put in right place $O(n)$</td>
<td>leftmost $O(n)$</td>
</tr>
<tr>
<td>AVL tree</td>
<td>put in right place $O(\log n)$</td>
<td>leftmost $O(\log n)$</td>
</tr>
</tbody>
</table>
Binary Heaps

Data Structure for Priority Queue
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

1. Delete (and later return) the value at root

Now we have a "hole" at the root ➔ replace it with another node
**deleteMin**: Step #2 (Keep Structure Property)

(a complete binary tree)

Want to keep structure property

1. Pick last node on the bottom row and "move" it to the "hole"
deleteMin: Step #3

Want to restore heap property

3. "Percolate down"
If lower priority than child, swap the most important child
repeat
deleteMin: Step #3 (Restart 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? \( O(\log n) = O(\text{heap of heap}) \)
**deleteMin: Run Time Analysis**

- Run time is $O(\text{height heap})$

- A heap is a complete tree

- So its height with $n$ nodes is $\log n$

- So run time of deleteMin is $O(\log n)$
insert: Step #1

Put data into next available spot
(end of last row)

to maintain structure
(complete binary tree).

Maintain Structure Property
**insert: Step #2**

- Compare data with its parent.
- Swap with parent if it's more important than parent.
- Repeat.

**Restore Heap Property**
**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?

$O(\text{height}) = O(\log n)$
Summary: basic idea for operations

findMin: \textbf{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**

- 75% of values are in the bottom 2 rows
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></th>
<th>4</th>
<th>16</th>
<th>32</th>
<th>43</th>
<th>67</th>
<th>105</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, 7</td>
</tr>
</tbody>
</table>

B)

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

C)

<table>
<thead>
<tr>
<th></th>
<th>4</th>
<th>32</th>
<th>16</th>
<th>43</th>
<th>105</th>
<th>67</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, 7</td>
</tr>
</tbody>
</table>

D)
Semi-Pseudocode: `insert` into binary heap

```plaintext
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: `deleteMin` from binary heap

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

typedef struct Node {
    int value;
    struct Node *left, *right;
} Node;

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;
}
```

```
<table>
<thead>
<tr>
<th>10</th>
<th>20</th>
<th>80</th>
<th>40</th>
<th>60</th>
<th>85</th>
<th>99</th>
<th>700</th>
<th>50</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>
<td>8</td>
<td>9</td>
</tr>
</tbody>
</table>
```
Example

1. insert (in this order): 16, 32, 4, 67, 105, 43, 2
2. deleteMin once
Why AVL Trees cannot implement binary heaps via counter example:

Legit AVL tree

2

9

breaks Heap Property

breaks Structure Property