CSE 373: Data Structures and Algorithms

Lecture 12: Binary Heaps

Instructor: Lilian de Greef
Quarter: Summer 2017
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

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</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?
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

```
  4  15  32  43  67  105
0  1  2  3  4  5  6  7
```

A)  

```
16  32  4  67  105  43
0  1  2  3  4  5  6  7
```

B)  

```
  4  32  16  43  105  67
0  1  2  3  4  5  6  7
```

C)  

```
  4  32  16  43  105  67
0  1  2  3  4  5  6  7
```

D)  

```
  4  32  16  67  105  43
0  1  2  3  4  5  6  7
```
(extra space for your scratch work a notes)
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

```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;
}
```
Example

1. \textbf{insert (in this order)}: 16, 32, 4, 67, 105, 43, 2
2. \textbf{deleteMin once}

\begin{tabular}{cccccccc}
0 & 1 & 2 & 3 & 4 & 5 & 6 & 7
\end{tabular}
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?
Build Heap

• Suppose you have \( n \) items to put in a new (empty) priority queue
  • Call this operation \texttt{buildHeap}

• \( n \) inserts
  • Only choice if ADT doesn’t provide \texttt{buildHeap} explicitly

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

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

2. Fix the heap-order property
   • Bottom-up: percolate down starting at nodes one level up from leaves, work up toward the root
heapify (Floyd’s Method): Example

1. Use \( n \) items to make any complete tree you want
2. Fix the heap-order property from bottom-up

Which nodes break the heap-order property?
Why work from the bottom-up to fix them?

Why start at one level above the leaf nodes?
Where do we start here?
heapify (Floyd’s Method): Example
heapify (Floyd’s Method): Example
heapify (Floyd’s Method): Example
heapify (Floyd’s Method): Example
heapify (Floyd’s Method)

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

But is it right? ... it “seems to work”

- Let’s *prove* it restores the heap property
- Then let’s *prove* its running time
Correctness

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

*Loop Invariant:* For all $j > i$, $arr[j]$ is higher priority 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 `percolateDown` make $arr[i]$ higher priority than children without breaking the property for any descendants

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

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

Easier argument: `buildHeap` is `size` where `n` is `size`

- loop iterations
- Each iteration does one `percolateDown`, each is

This is correct, but there is a more precise (“tighter”) analysis of the algorithm...
Efficiency

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

Better argument: \( \text{buildHeap is } \quad \text{where } n \text{ is size} \)
- size/2 total loop iterations: \( O(n) \)
- 1/2 the loop iterations percolateDown at most
- 1/4 the loop iterations percolateDown at most
- 1/8 the loop iterations percolateDown at most
- ...
- \( \left( \frac{1}{2} + \frac{2}{4} + \frac{3}{8} + \frac{4}{16} + \ldots \right) < 2 \quad \text{(page 4 of Weiss)} \)

\[
\sum_{i=1}^{\infty} \frac{i}{2^i} = 2
\]
Lessons from buildHeap

• Without providing buildHeap, clients can implement their own that runs in worst case

• By providing a specialized operation (with access to the internal data), we can do 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 (easier) analysis proved it was $O(n \log n)$
    • Tighter analysis shows same algorithm is $O(n)$
Other branching factors for Heaps

\textit{d-heaps}: have \textit{d} children instead of 2
  \begin{itemize}
    \item Makes heaps shallower
  \end{itemize}

Example: 3-heap
  \begin{itemize}
    \item Only difference: three children instead of 2
    \item Still use an array with all positions from 1 \ldots \text{heapSize}
  \end{itemize}

\begin{tabular}{|c|c|}
  \hline
  \textbf{Index} & \textbf{Children Indices} \\
  \hline
  1 & \\
  \hline
  2 & \\
  \hline
  3 & \\
  \hline
  4 & \\
  \hline
  5 & \\
  \hline
  ... & ...
  \hline
\end{tabular}
Wrapping up Heaps

• What are heaps a data structure for?

• What is it usually implemented with? Why?

• What are some example uses?