CSE 373: Data Structures & Algorithms

Pseudocode; ADTs; Priority Queues; Heaps

Riley Porter
Winter 2017
Course Logistics

• HW 1 released Monday. Due a week from Tuesday.

• Java Review Session early next week, room and time TBA and posted on the course website

• Slides posted and updated from last time with links correctly working in PDF version
Pseudocode

Describe an algorithm in the steps necessary, write the shape of the code but ignore specific syntax.

**Algorithm:** Count all elements in a list greater than x

**Pseudocode:**

```plaintext
int counter // keeps track of number > x
while list has more elements {
    increment counter if current element is > than x
    move to next element of list
}
```
More Pseudocode

Algorithm: Given a list of names in the format "firstName lastName", make a Map of all first names as keys with sets of last names as their values

Pseudocode:

create the empty result map
while list has more names to process {
  firstName is name split up until space
  lastName is name split from space to the end
  if firstName not in the map yet {
    put firstName in map as a key with an empty set as the value
  }
  add lastName to the set for the first name
  move to the next name in the list
}
Pseudocode Practice

Come up with pseudocode for the following algorithm:

**Algorithm:** Given a list of integers, find the index of the maximum integer in the list.
Pseudocode Practice Solution

**Algorithm:** Given a list of integers, find the index of the maximum integer in the list.

if list is not empty:
    int maxIndex starts at 0 for first index
    for each index i in the list:
        if the element at index i is greater than the element at index maxIndex:
            reset maxIndex to i
        return maxIndex
else:
    error case: return -1? throw exception?
Terminology Review

• Abstract Data Type (ADT)
  – Mathematical description of a "thing" with set of operations

• Algorithm
  – A high level, language-independent description of a step-by-step process

• Data structure
  – A specific organization of data and family of algorithms for implementing an ADT

• Implementation of a data structure
  – A specific implementation in a specific language
Another ADT: Priority Queue

A priority queue holds comparable data

– Given $x$ and $y$, is $x$ less than, equal to, or greater than $y$

– Meaning of the ordering can depend on your data

– Many data structures require this: dictionaries, sorting

– Typically elements are comparable types, or have two fields: the *priority* and the *data*
**Priority Queue vs Queue**

**Queue**: follows First-In-First-Out ordering  
*Example*: serving customers at a pharmacy, based on who got there first.

**Priority Queue**: compares priority of elements to determine ordering  
*Example*: emergency room, serves patients with priority based on severity of wounds
Priorities

• Each item has a "priority"
  – The lesser item is the one with the greater priority
  – So "priority 1" is more important than "priority 4"
  – Can resolve ties arbitrarily

• Operations:
  – insert
  – deleteMin
  – is_empty

• deleteMin returns and deletes the item with greatest priority (lowest priority value)
• insert is like enqueue, deleteMin is like dequeue
  – But the whole point is to use priorities instead of FIFO
Priority Queue Example

Given the following, what values are \( a \), \( b \), \( c \) and \( d \)?

- \texttt{insert element1} with priority 5
- \texttt{insert element2} with priority 3
- \texttt{insert element3} with priority 4
- \( a = \texttt{deleteMin} \) // \( a = ? \)
- \( b = \texttt{deleteMin} \) // \( b = ? \)
- \texttt{insert element4} with priority 2
- \texttt{insert element5} with priority 6
- \( c = \texttt{deleteMin} \) // \( c = ? \)
- \( d = \texttt{deleteMin} \) // \( d = ? \)
Priority Queue Example Solutions

insert \textit{element1} with priority 5
insert \textit{element2} with priority 3
insert \textit{element3} with priority 4
\( a = \text{deleteMin} \quad // \quad a = \textit{element2} \)
\( b = \text{deleteMin} \quad // \quad b = \textit{element3} \)
insert \textit{element4} with priority 2
insert \textit{element5} with priority 6
\( c = \text{deleteMin} \quad // \quad c = \textit{element4} \)
\( d = \text{deleteMin} \quad // \quad d = \textit{element1} \)
Some Applications

- Run multiple programs in the operating system — "critical" before "interactive" before "compute-intensive", or let users set priority level
- Select print jobs in order of decreasing length
- "Greedy" algorithms (we’ll revisit this idea)
- Forward network packets in order of urgency
- Select most frequent symbols for data compression (Huffman CSE 143)
- Sorting (first \texttt{insert} all, then repeatedly \texttt{deleteMin})
Possible Implementations

• Unsorted Array
  – **insert** by inserting at the end
  – **deleteMin** by linear search

• Sorted Circular Array
  – **insert** by binary search, shift elements over
  – **deleteMin** by moving “front”
More Possible Implementations

• Unsorted Linked List
  – \textbf{insert} by inserting at the front
  – \textbf{deleteMin} by linear search

• Sorted Linked List
  – \textbf{insert} by linear search
  – \textbf{deleteMin} remove at front

• Binary Search Tree
  – \textbf{insert} by search traversal
  – \textbf{deleteMin} by find min traversal
One Implementation: Heap

Heaps are implemented with Trees

A \textit{binary min-heap} (or just \textit{binary heap} or \textit{heap}) is a \textbf{data structure} with the properties:

- **Structure property:** A \textit{complete} binary tree
- **Heap property:** The priority of every (non-root) node is greater than the priority of its parent
  - \textbf{Not} a binary search tree
Tree Review

• root of tree:
• leaves of tree:
• children of B:
• parent of C:
• subtree C:
• height of tree:
• height of E:
• depth of E:
• degree of B:

• perfect tree:
• complete tree:
Tree Review

- root of tree: A
- leaves of tree: H, E, F, G
- children of B: D, E, F
- parent of C: A
- subtree C: in blue
- height of tree: 3
- height of E: 0
- depth of E: 2
- degree of B: 3

- perfect tree: every level is completely full
- complete tree: all levels full, with a possible exception being the bottom level, which is filled left to right
Structure Property: Completeness

• A Binary Heap is a complete binary tree:
  – A binary tree with all levels full, with a possible exception being the bottom level, which is filled left to right

Examples:

![Binary Heap Trees](image)

are these trees complete?
Structure Property: Completeness

• A Binary Heap is a **complete** binary tree:
  – A binary tree with all levels full, with a possible exception being the bottom level, which is filled left to right

**Examples:**

- Complete: 10
  - 20
  - 80
  - 40
  - 60
  - 85
  - 99
- Incomplete: 10
  - 20
  - 80
  - 30
  - 40
  - 400
Heap Order Property

• The priority of every (non-root) node is greater than (or equal to) that of its parent. AKA the children are always greater than the parents.

which of these follow the heap order property?
Heap Order Property

• The priority of every (non-root) node is greater than (or equal to) that of it's parent. AKA the children are always greater than the parents.

Heaps have a property where each node is greater than its children. The left diagram shows a valid heap (also known as heap property), while the right diagram shows a tree that does not have the heap property (not the heap property).
Heaps

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

- **Structure property:** A complete binary tree
- **Heap property:** The priority of every (non-root) node is greater than (or equal to) the priority of its parent. AKA the children are always greater than the parents.
  - *Not* a binary search tree

Which of these are heaps?
Heaps

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

- **Structure property:** A complete binary tree
- **Heap property:** The priority of every (non-root) node is greater than (or equal to) the priority of its parent. AKA the children are always greater than the parents.
  - *Not* a binary search tree

![Heaps diagram](image)
Heaps

• Where is the highest-priority item?

• What is the height of a heap with $n$ items?

• How do we use heaps to implement the operations in a Priority Queue ADT?
Heaps

• Where is the highest-priority item?
  At the root (at the top)

• What is the height of a heap with \( n \) items
  \( \log_2 n \) (We’ll look at computing this next week)

• How do we use heaps to implement the operations in a Priority Queue ADT?
  See following slides
Operations: basic idea

• **deleteMin:**
  1. \texttt{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

\textbf{Overall strategy:}

- **Preserve structure property**
- **Break and restore heap property**
1. Delete (and later return) value at root node
2. Restore the Structure Property

- We now have a "hole" at the root
  - Need to fill the hole with another value

- When we are done, the tree will have one less node and must still be complete
3. Restore the Heap Property

Percolate down:
- Keep comparing with both children
- Swap with lesser child and go down one level
  - What happens if we swap with the larger child?
- Done if both children are $\geq$ item or reached a leaf node
Insert

• Add a value to the tree

• Afterwards, structure and heap properties must still be correct

• Where do we insert the new value?
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
Maintain the heap property

Percolate up:
- Put new data in new location
- If parent larger, swap with parent, and continue
- Done if parent ≤ item or reached root
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>
</tbody>
</table>
Judging the array implementation

Plusses:
• Less "wasted" 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 size

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

Plusses outweigh minuses: "this is how people do it"