A new ADT: Priority Queue

- 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
  - Integers are comparable, so will use them in examples
    - But the priority queue ADT is much more general
    - Typically two fields, the priority and the data

Priorities

- Each item has a "priority"
  - In our examples, the lesser item is the one with the greater priority
  - So "priority 1" is more important than "priority 4"
  - (Just a convention, think "first is best")

- Operations:
  - insert
  - deleteMin
  - is_empty

- Key property: deleteMin returns and deletes the item with greatest priority (lowest priority value)
- Can resolve ties arbitrarily

Example

| insert x1 with priority 5 | (x1, 5) |
| insert x2 with priority 3 | (x1, 5) (x2, 3) |
| insert x3 with priority 4 | (x1, 5) (x3, 4) (x2, 3) |
| a = deleteMin // x2 | (x1, 5) (x3, 4) |
| b = deleteMin // x3 | (x1, 5) |
| insert x4 with priority 2 | (x1, 5) (x2, 3) (x4, 2) |
| insert x5 with priority 6 | (x1, 5) (x2, 3) (x5, 6) |
| c = deleteMin // x4 | (x1, 5) (x5, 6) |
| d = deleteMin // x1 | (x1, 5) |

- 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
- Sometimes blatant, sometimes less obvious

- Run multiple programs in the operating system
  - "critical" before "interactive" before "compute-intensive"
  - Maybe let users set priority level
- Treat hospital patients in order of severity (or triage)
- Select print jobs in order of decreasing length?
- Forward network packets in order of urgency
- Select most frequent symbols for data compression
- Sort (first insert all, then repeatedly deleteMin)

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
  - All times worst-case: assume arrays "have room"

<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(2 log n)</td>
</tr>
<tr>
<td>AVL tree</td>
<td>put in right place O(log n)</td>
<td>leftmost O(2 log n)</td>
</tr>
</tbody>
</table>
Our data structure: the Binary Heap

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

- Structure property: A complete binary tree
- Heap property: The priority of every (non-root) node is less than the priority of its parent
  - Not a binary search tree

Operations: basic idea

- **deleteMin:**
  1. Remove root node
  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:
- Preserve structure property
- Break and restore heap property

deleteMin

Delete (and later return) value at root node

deleteMin: Keep the Structure Property

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

- Keep structure property: When we are done, the tree will have one less node and must still be complete

- Pick the last node on the bottom row of the tree and move it to the “hole”

insert

- Add a value to the tree
- Afterwards, structure and heap properties must still be correct
**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.

**insert: Restore the heap property**

**Percolate up:**
- Put new data in new location.
- If parent is less important, swap with parent, and continue.
- Done if parent is more important than item or reached root.

Array Representation of Binary Trees

From node i:
- Left child: \(i \times 2\)
- Right child: \(i \times 2 + 1\)
- Parent: \(i / 2\)

Implicit (array) implementation:

<table>
<thead>
<tr>
<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>
</tr>
</tbody>
</table>

Judging the array implementation

**Plusses:**
- 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 size.

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

Pluses outweigh minuses: "this is how people do it!"

Pseudocode: insert into binary heap

```java
void insert(int val) {
  if (size == arr.length - 1) resize();
  size++;
  i = percolateUp(size, val);
  arr[i] = val;
}
```

```java
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.

Pseudocode: percolateUp(10, 30)

```java
void insert(int val) {
  if (size == arr.length - 1) resize();
  size++;
  i = percolateUp(size, val);
  arr[i] = val;
  return hole;
}
```

This pseudocode uses ints. In real use, you will have data nodes with priorities.

```
```
Pseudocode: `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;
}
```

```c
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. `insert`: 16, 32, 4, 67, 105, 43, 2
2. `deleteMin`

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

Example
1. `insert`: 16, 32, 4, 67, 105, 43, 2
2. `deleteMin`

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

Example
1. `insert`: 16, 32, 4, 67, 105, 43, 2
2. `deleteMin`

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

Example
1. `insert`: 16, 32, 4, 67, 105, 43, 2
2. `deleteMin`

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

1. insert: 16, 32, 4, 67, 105, 43, 2
2. deleteMin

Exercise

1. insert: 16, 32, 4, 67, 105, 43, 2
2. deleteMin

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 buildHeap
- \( n \) inserts works
  - Only choice if ADT doesn’t provide buildHeap explicitly
    - \( O(n \log n) \)
- Why would an ADT provide this unnecessary operation?
  - Convenience
  - Efficiency: an \( O(n) \) algorithm called Floyd’s Method
  - Common issue in ADT design: how many specialized operations
Floyd’s Method

1. Use \( n \) items to make any complete tree you want
   - That is, put them in array indices 1,…,\( n \)

2. Treat it as a heap and fix the heap-order property
   - Bottom-up: leaves are already in heap order, work up
     toward the root one level at a time

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

Example

- In tree form for readability
  - Purple for node not less than descendants
  - heap-order problem
  - Notice no leaves are purple
  - Check/fix each non-leaf

```
Step 1
```
- Happens to already be less than children (er, child)

```
Step 2
```
- Percolate down (notice that moves 1 up)

```
Step 3
```
- Another nothing-to-do step

```
Step 4
```
- Percolate down as necessary (steps 4a and 4b)
But is it right?

• “Seems to work”
  – Let’s prove it restores the heap property (correctness)
  – Then let’s prove its running time (efficiency)

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

Correctness

Loop invariant: For all j>i, arr[j] is less than its children

- True initially: If j > size/2, then j is a leaf
  - Otherwise its left child would be at position > size
- True after one more iteration: loop body and percolateDown make arr[i] less than children without breaking the property for any descendants

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

Efficiency

Easy argument: buildHeap is O(n log n) where n is size
- size/2 loop iterations
- Each iteration does one percolateDown, each is O(log n)

This is correct, but there is a more precise ("lighter") analysis of the algorithm...

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

Better argument: buildHeap is O(n) where n is size
- size/2 total loop iterations: O(n)
- 1/2 the loop iterations percolate at most 1 step
- 1/4 the loop iterations percolate at most 2 steps
- 1/8 the loop iterations percolate at most 3 steps
  - ... 
  - \((1/2) + (2/4) + (3/8) + (4/16) + (5/32) + ... < 2\) (page 4 of Weiss)
  - So at most \(2 \times (\text{size}/2)\) total percolate steps: O(n)
Lessons from buildHeap

- Without buildHeap, our ADT already let clients implement their own in $O(n \log n)$ worst case.
- By providing a specialized operation internal to the data structure (with access to the internal data), we can do $O(n)$ 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 analysis easily proved it was $O(n \log n)$.
    - Tighter analysis shows same algorithm is $O(n)$.

Exercise: Build the Heap using Floyd’s method

```
3 | 9 | 2 | 6 | 25 | 1 | 80 | 35
1   2   3   4   5   6   7   8
```

```
3
9   2
6   1
80
35
```

Autumn 2016  CSE373: Data Structures & Algorithms  43