CmpSci 187: Programming with Data Structures
Spring 2015

Lecture #19, Heaps and Priority Queues

John Ridgway

April 9, 2015

1 Heaps

Review: The Idea of a Heap

- A **heap** is a data structure that keeps some comparable elements in a **semi-sorted** state.
- The largest element is at the root of the tree, and larger elements will tend to be nearer the top of the tree.
- We'll use a heap to implement a **priority queue**, where the important operations are to insert a new element and remove the largest element.

The Heap Property

- Formally, a heap (in this case a **max-heap**) is a complete tree of comparable elements that satisfies the **heap property**: the element in every node is greater than (or equal to) the elements in its children.
- Hence that element must also be larger than the elements in any of its descendants. In particular, the largest element must be at the root. (In a **min-heap**, each node's element is less than (or equal to) the elements in its children.)

Introduction to Heaps

- Remember that a complete binary tree has its leaves on one level or on two adjacent levels; in the latter case the leaves on the upper level all exist and those on the lower level are left-justified.
- As we said, a heap is "somewhat sorted". We can find the maximum element quickly, but the farther down we go the less we know about the relative order of elements.

Heaps With Implicit Pointers

- As we mentioned last lecture, we can implement a tree structure with an array by using implicit pointers: the left child of node i is node $2i+1$, and the right child is node $2i+2$.
- With this convention, an array of length n corresponds exactly to an n-node complete binary tree.

```
public class Heap<T extends Comparable<T>>
    implements PriorityQueueInterface<T>
{
  private T[] elements;
  private int size;

  public Heap(int maxSize) {
    elements = (T[])(new Comparable[maxSize]);
    size = 0;
  }
  public boolean isEmpty() {
    return size == 0;
  }
  public boolean isFull() {
    return (size >= elements.length);
  }
```

Reheaping Up to Add Elements

- When we enqueue an element in the priority queue, our heap becomes larger by one element.
- We know exactly where the new element must go, in the next available array slot, so we can add it there, but this could destroy the heap property.
- We will fix the property by **reheaping up**.
- We do this by comparing the element with its parent.
- If the element is less than or equal to its parent then we're done.
- Otherwise we swap the element and its parent and repeat the reheaping up process with the parent.
- We continue this way we have satisfied the heap property, which might not happen until we reach the root. This means shifting up to $O(\log n)$ elements and takes $O(\log n)$ time in the worst case.

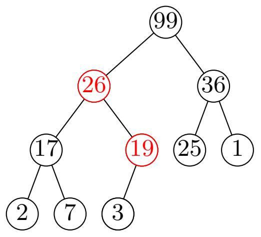
Recursive Code to Reheap Up

```
private void reheapUp(int i) {
  if (i <= 0) return;
  int p = (i -1) / 2;
  if (elements[i].compareTo(elements[p]) > 0) {
    swap(i, p);
    reheapUp(p);
  }
}

private void swap(int i1, int i2) {
  T temp = elements[i1];
  elements[i1] = elements[i2];
  elements[i2] = temp;
}
```

Clicker Question #1

Suppose we add a new element 26 to this heap. Not including the new node, how many nodes will have a different value from before after this operation?



A. 0

B. 1

C. **2**

D. 3

Reheaping Down to Dequeue

- We know that we want to return the element in the root, but we have to adjust the heap before we do that. We move the element from the former last location into the root.
- This may leave the root node violating the heap property. If so we must reheap down, swapping elements to move the element down until the heap property is restored.

Code for `dequeue`

```
public T dequeue()
    throws PriorityQueueUnderflowException {
  if (isEmpty()) {
    throw new PriorityQueueUnderflowException();
  }
  T hold = elements[0];
  size -= 1;
  elements[0] = elements[size];
```

```
    if (! isEmpty()) {
      reheapDown(0);
    }
    return hold;
  }
```

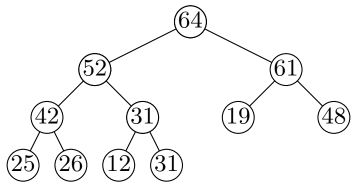
Reheaping Down

- We want to move the element downward until it no longer has children that are too big. The way we do this is to swap it with the element from the larger of the element's children; it may go there because it is greater than (or equal to) its new children.
- The natural way to do this is recursively.

Code for `reheapDown`

```
private void reheapDown(int i) {
  int l = 2 * i + 1;
  int r = 2 * i + 2;
  if (l >= size) { return; }
  int big = l;
  if (r < size &&
      elements[r].compareTo(elements[l]) > 0)
    {
      big = r;
    }
  if (elements[i].compareTo(elements[big]) < 0) {
    swap(big, i);
    reheapDown(big);
  }
}
```

Clicker Question #2



If we dequeue the top element from this heap, what will be the values of the first seven nodes?

A. 73, 64, 61, 42, 52, 19, 48

B. 31, 64, 61, 42, 52, 19, 48

C. 64, 52, 61, 42, 31, 19, 48

D. 64, 52, 61, 42, 31, 19, 48

E. 64, 61, 52, 48, 42, 31, 19

2 Priority Queues

Priority Queues and Their Uses

- A **priority queue** is a collection where the elements come from an ordered type, and where the removal operation gives us not the newest element (as in a stack) or the oldest element (as in a queue) but the **largest** element according to the order.
- We might call this BIFO for "best-in-first-out", as opposed to FIFO for a queue and LIFO for a stack.
- There are many situations where the next item in a list to tackle is the most important according to some measure. Operating systems have a relative priority on processes and favor the ones with highest priority in timesharing.
- We often make a priority queue of items that each have a number attached for their priority. The `compareTo` operation on these pairs calls an item "larger" if it has higher priority.

The `PriorityQueueInterface`

```
public interface PriorityQueueInterface<T
                     extends Comparable<T>> {
  boolean isEmpty();
  boolean isFull();
  void enqueue(T element);
  T dequeue();
}
```

Implementing a Priority Queue

- If we used an unsorted list to implement a priority queue, we could enqueue in $O(1)$ time but we would always need $O(n)$ time to dequeue from a queue of size n.
- With an array-based sorted list, we could dequeue in $O(1)$ time but would need $O(n)$ to enqueue, because in the worst case we must move $O(n)$ elements.
- With a reference-based sorted list, dequeuing is again $O(1)$ time but now enqueuing requires $O(n)$ time in the worst case to find the place to insert.
- In a binary search tree, both enqueuing and dequeuing require a trip from the root of the tree down to a leaf in the worst case. This is $O(\log n)$ time if the tree is balanced, but could be as bad as $O(n)$ if it is not.
- Using a heap we can guarantee $O(\log n)$ time for both insertion and removal, because a heap is always balanced after each operation.

Clicker Question #3

Suppose we have implemented a priority queue with an array-based heap. What is the worst-case running time of the fastest possible `contains` method, if the heap has n elements?

A. $O(n \log n)$

B. $O(n)$

C. $O(\log n)$

D. $O(1)$

The Java `PriorityQueue`

- Like `Stack`, the `PriorityQueue` in the Java Collections is a class rather than an interface. It is a specific implementation of a priority queue using a heap.
- Elements are compared using their natural order if they are `Comparable`, or using a `Comparator` object if they are not.
- The essential methods of the class are `add`, `contains`, `offer`, `peek`, `poll`, `remove`, and `iterator`, just as for an ordinary queue. (It implements the `Queue` interface.)

General Search Algorithms

- The natural setting for general search is **graphs**, which we'll start looking at next week. But we can see a lot of the general principles of searching by looking at searches on a **grid**.
- Recall the rectangular grid of squares on which we counted the continents. We used a recursive **depth-first search**, using the method stack to find all the squares on the same continent as our initial square.
- We could also place the squares being considered on a queue, for a **breadth-first search**. This has the advantage that when we find a path, that path is as short as possible, counting each move north, east, south, or west as being length 1.
- What if all moves to adjacent squares are not equally costly? For example, we might have lowland squares that cost 1 to enter, hills that cost 2, and mountains that cost 3.

Shortest Paths on a Grid

- This map has squares of those four kinds, and three cities called `X`, `Y`, and `Z`, on lowland squares.
- How would we best find the shortest paths from `X` to `Y`, `X` to `Z`, and `Y` to `Z`?
- Actually it's easiest to find all the shortest paths from `X` to every other node.

```
MMMMHMXLL
MMMMMHHHL
MLLLMLLHL
HLHHMHLLL
HLHHOHHLH
HLHHOHLLL
LLOOOOOOL
YLOOOOOOZ
```

Entry costs: `L`: 1, `H`: 2, `M`: 3

- `X` is in row 0 and column 6, and is at distance 0 from itself, so we'll start with a data item (0, 6, 0).
- `X`'s three neighbors give us items (0, 5, 3), (0, 7, 1), and (1, 6, 2). Let's throw those three items into a min-PQ, with distance from `X` as the thing we compare.

```
MMMMH3X1L
MMMMMH2HL
MLLLMLLHL
HLHHMHLLL
HLHHOHHLH
HLHHOHLLL
LLOOOOOOL
YLOOOOOOZ
```

Entry costs: `L`: 1, `H`: 2, `M`: 3

- The item in the PQ of minimum distance is (0, 7, 1). We dequeue it and enqueue its two neighbors, omitting (0, 6) because it has already been on and off the queue.
- The new items are (0, 8, 2) and (1, 7, 3). The new distances are the 1 to get to (0, 7) plus the distance from (0, 7) to the new node.

```
MMMMH3X12
MMMMMH23L
MLLLMLLHL
HLHHMHLLL
HLHHOHHLH
HLHHOHLLL
LLOOOOOOL
YLOOOOOOZ
```

Entry costs: `L`: 1, `H`: 2, `M`: 3

- Now the queue has three items on it: (0, 5, 3), (1, 6, 2), and (0, 8, 2). We have a tie for the minimum and can dequeue either – let's take (1, 6, 2).
- Its new neighbors are (1, 5, 4), (2, 6, 3), and (1, 7, 4).
- We then dequeue (0, 8, 2), producing (1, 8, 3).

```
MMMMH3X12
MMMMM4233
MLLLML3HL
HLHHMHLLL
HLHHOHHLH
HLHHOHLLL
LLOOOOOOL
YLOOOOOOZ
```

Entry costs: `L`: 1, `H`: 2, `M`: 3

- In this way we get an expanding area of nodes for which we know the distance from `X`. When this area includes our targets `Y` and `Z`, we will be done.
- If there are multiple paths from `X` to a node, the item for the shortest path will come off the PQ first.

```
MMMMHMXLL
MMMMMHHHL
MLLLMLLHL
HLHHMHLLL
HLHHOHHLH
HLHHOHLLL
LLOOOOOOL
YLOOOOOOZ
```

Entry costs: `L`: 1, `H`: 2, `M`: 3

Clicker Question #4

There are two paths of length 5 from `X` to (2, 7). Which one will the priority queue find first?

A. `X` $\to$ (1, 6) $\to$ (2, 6) $\to$ (2, 7)

B. `X` $\to$ (0, 7) $\to$ (1, 7) $\to$ (2, 7)

C. some other path

D. **we can't tell from what we've been told**

```
MMMMHMXLL
MMMMMHHHL
MLLLMLLHL
HLHHMHLLL
HLHHOHHLH
HLHHOHLLL
LLOOOOOOL
YLOOOOOOZ
```

Entry costs: `L`: 1, `H`: 2, `M`: 3