CS 4102, Algorithms: Heapsort

- Expectations:
  - Section 3.5 and CS216 slides
  - Next: Sections 4.2-4.5
    - Graph searching
- Problems to do are coming…

Review from CS216

- Review these slides!
  - Slides from 3-26-03 on *Priority Queues (Binary Min Heaps)*
    http://www.cs.virginia.edu/~cs216/notes/slides/heaps.pdf

Reminders, Terminology

- ADT priority queue
  - What's an ADT?
  - What's high priority?
  - Operations?
  - How is data stored?
- Heap data structure
  - The *heap structure*: an almost-complete binary tree
  - The *heap condition* or *heap order property:*
    - At any given node j, value[j] has higher priority than either of its child nodes' values
    - Heaps are weakly sorted
  - Higher priority: large or small?
    - Max-heap vs min-heap

Storing Heaps in Lists

- Heap structure allows us to store heaps in lists (arrays, vectors) effectively
- Indexing in our text: v[1] is the root. v[n] is the last item
- parent of node j is at j/2
- left-child of node j is at:
  - 2*j
- right-child of node j is at:
  - 2*j + 1
- "first leaf" is
  - n/2 + 1

Basic Heap Algorithms

- Let’s work with max-heaps for know
- Define a set of simple heap operations
  - Traverse tree, thus logarithmic complexity
- Highest item?
  - At the root. Just return it.
- Insert an item?
  - Add after the nth item (end of list)
  - Out of place? Swap with parent. Repeat, pushing it up the tree until in proper place
- Remove an item?
  - Hmm...

Insert, Algorithm 3.5.10, p. 140

This algorithm inserts the value *val* into a heap containing *n* elements. The array *v* represents the heap.

```
Input Parameters: val,v,n
Output Parameters: v,n

heap_insert(val,v,n) {
   i = n = n + 1
   // i is the child and i/2 is the parent.
   // If i > 1, i is not the root.
   while (i > 1 && val > v[i/2]) {
      v[i] = v[i/2]
      i = i/2
   }
   v[i] = val
}
```

Siftdown: Fix a Heap if Root Wrong

- Algorithm 3.5.7, p. 138
  - Also known as "Fixheap" and "heapify" (ugh)
  - Called at a given index (often root, but perhaps not)
- Assumption:
  - The left and right subtrees of node *i* are heaps.
  - The element at node *i* may violate heap condition
- Strategy:
  - Find larger of two children of current node
  - If current node is out-of-place, then swap with largest of its children
  - Keep pushing it down until in the right place or it's a leaf

```
// Input Parameter: v,i,n     Output Parameters: v
siftdown(v,i,n) {
   temp = v[i]
   // 2 * i ≤ n tests for a left child
   while (2 * i ≤ n) {
      child = 2 * i
      // if there is a right child and it is
      // bigger than the left child, move child
      if (child < n && v[child + 1] > v[child])
         child = child + 1
      // move child up?
      if (v[child] > temp)
         v[i] = v[child]
      else
         break // exit while loop
      i = child
   }
   // insert original v[i] in correct spot
   v[i] = temp
}
```

Delete, Algorithm 3.5.9, p. 139

This algorithm deletes the root (the item with largest value) from a heap containing *n* elements. The array *v* represents the heap.

```
Input Parameters: v,n
Output Parameters: v,n

heap_delete(v,n) {
   v[1] = v[n]
   n = n - 1
   siftdown(v,1,n)
}
```

How to Build a Heap

- Option 1:
  - Repeatedly Insert() a new item, start with a heap of 1 item
  - Cost: Θ(n lg n)   (Can you do the sum?)
- Option 2:
  - Take an unordered list, build the heap in place
  - Heapify() algorithm, page 141
  - Strategy:
    - Work bottom up, starting with lowest sub-heaps
    - Call Siftdown() on each

  - Note: This often called "BuildHeap" etc.
    - Cormen et. al. calls Siftdown() "heapify"

Heapify, Algorithm 3.5.12, p. 141

This algorithm rearranges the data in the array *v*, indexed from 1 to *n*, so that Heapsort it represents a heap.

```
Input Parameters: v,n
Output Parameters: v

heapify(v,n) {
   // n/2 is the index of the parent of
  // the last node
   for i = n/2 downto 1
      siftdown(v,i,n)
}
```

Complexity? Θ(n) See p. 142

Heapsort: the Strategy

- We can sort in-place by
  - Putting large items at the end of the list
  - Keeping a heap in the space in front of those
- So, to start off:
  - Put the largest item in the last position
  - Make sure items 1 through n-1 are a heap of size n-1
  - Repeat, moving the 2^{nd} largest into the n-1 position, etc.

Heapsort, Algorithm 3.5.16, p. 145

This algorithm sorts the array *v[*1], ... , *v*[*n*] in nondecreasing order. It uses the *siftdown* and *heapify* algorithms (Algorithms 3.5.7 and 3.5.12).

```
Input Parameters: v,n
Output Parameter: v
heapsort(v,n)  {
   // make v into a heap
   heapify(v,n)
   for i = n downto 2 {
      // v[1] is the largest among v[1], ... , v[i].
      // Put it in the correct cell.
      swap(v[1],v[i])
      // Heap is now at indexes 1 through i - 1.
      // Restore heap.
      siftdown(v,1,i - 1 )
   }
}
```

Heapsort's Complexity

- Constructing the heap: $\Theta(n)$
- Each call to Siftdown()
  - No greater than lg n, so O(lg n)
  - There are n-1 of these, so
  - Overall, O(n lg n)
  - We know it's Θ(n lg n) because of the lower-bound proof done earlier
  - Can prove directly it's Θ(n lg n) but our book doesn't (so let's not bother)
- In practice, slower then randomized Quicksort
- But truly in-place, and guaranteed log-linear