CSE 241 Algorithms and Data Structures 11/4/2015

Binary Heaps

A binary heap Q is an implementation of the priority queue data type. It supports the following operations:

- Q.insert(k) – insert a key k into the heap
- Q.min() – return the smallest key in the heap
- Q.extractMin(k) – remove the smallest key from the heap

The above operations (and the rest of this handout) are defined for a *min-first* heap. A *max-first* heap supports Q.max() and Q.extractMax(k). Binary heaps can support either min or max operations, but not both. Not surprisingly, the above ops can be generalized to work on records with key values, rather than just the keys alone.

Heap operations are designed to maintain the **heap property** (which is best described in the tree view rather than the array view)

> For any node x in a min-first heap, every node in the subtree rooted at x must have a key $\geq x$'s key.

A heap of size n supports **insert** and **extractMin** in time $O(\log n)$ and **min** in constant time. Heaps do *not* support efficient (sub-linear) search or fast successor/predecessor operations.

One additional important heap operation is **decreaseKey**(i, k). This operation replaces the ith key $Q[i]$ in a heap with the value k *if* $Q[i] > k$. You can think of the value i as being a pointer to a heap element. Lab 4 shows how to keep track of this pointer for heap applications.

The **decreaseKey** operation can run in time $O(\log n)$. Using this operation, we can implement deletion of any element of a heap in time $O(\log n)$.

The following pages give pseudocode for the major heap operations (except for **decreaseKey**), assuming that the heap is stored as an *array* $A[1 \ldots n]$. In such an array, the children of the element $A[i]$ are $A[2i]$ and $A[2i + 1]$, while the parent of $A[i]$ is $A[\lfloor i/2 \rfloor]$. $A[1]$ is the root of the heap, while $A[0]$ remains unused.

The following operations are all given for a *min-first* heap. Finding the minimum is trivial:

```
MIN(A)
    return A[1]
```

Insertion starts at the bottom of the heap and works its way up, trying to find the right place to insert the new key k. If the current key $A[i]$ is smaller than the new key k, the heap property demands that we move $A[i]$ below k.

```
INSERT(A, k)
    A.length++
    i ← A.length
    while i > 1 and k < A[⌊i/2⌋] do
        A[i] ← A[⌊i/2⌋]
        i ← ⌊i/2⌋
    A[i] ← k
```

(Notice that if we start the insertion loop in the middle of the heap rather than at the end, it's a very short step to **DecreaseKey**!)

Extracting the minimum key from a heap in the array representation requires that we put a new key in $A[1]$ without leaving any empty cells in the middle of A. We achieve this goal by moving the last element of the heap into $A[1]$, then fixing up the heap (using the **Heapify** procedure shown below) to make sure the heap property is maintained.

```
EXTRACTMIN(A)
    k ← A[1]
    A[1] ← A[length]
    A.length--
    HEAPIFY(A, 1)
    return k


HEAPIFY(A, i)
    if i ≤ ⌊A.length/2⌋                          ▷ i not a leaf
        j ← index of A[i]'s smallest child
        if A[j] < A[i]
            SWAP(A[i], A[j])
            HEAPIFY(A, j)
```