CMPS 101, Spring 2016: HW 5

Merrick Swaffar and Siobhan O'Shea

11 May 2016

Q1: delete(A,i)

```
    n = A.length
    A[i] = A[n]
    A[n] = null
    return minheapify(A,i)

minheapify(A,i)
    left = 2i
    right 2i + 1
    if (A[i] < A[left] and A[i] < A[right])
        return A
    if (A[i] > A[left] and A[i] > A[right])
        swap (A[i], A[left])
        return heapify(A,left)
    swap (A[i], A[right])
    return heapify(A,right)
```

The time complexity of this algorithm is $\Theta(\log n)$ because delete does one call to heapify and then does linear time operations.

Q2: Mergearrays(A)

```
    if A.length == 1
        return A[1]
    L = mergearrays (A[1:n/2])
    R = mergearrays (A[n/2+1:n]
    return merge (L,R)
```

The time complexity of Mergearrays is O (n log k) because the array A has length k and it recursively splits the array so you have log k recursive calls and merge has a time complexity of O(n).

Q3: heapSort(A)

```
    H = buildminheap(A)
```

```
    for i = 1 to A.length
        A[i] = Extractmin(H)
    return A

Extractmin(A)
    min = A[1]
    A.size = A.size - 1
    minheapify(A,1)
    return min

Build minheap(A)
    for i = A.length//2 to 1
        min heapify (A,i)
```

If you have array [5 4 2 4' 1 3] when you heapify it you get [1 4' 2 5 4 3] and when you sort it you get [1 2 3 4' 4 5]. The 4' which is after 4 in the first array ends up before 4 in the sorted array which means that the sort is not stable.

Q4:

```
Insert(r,n)
    if r = null
        r = n
    else if r < n.key
        Insert (r.right, n)
    else
        Insert (r.left, n)

Delete(n)
    if n.right and n.left = null
        n = null
    else if n.right = null
        swap (n, n.left)
        delete (n.left)
    else
        swap (n, n.right)
        delete (n.right)

Block Delete(r,a)
    n = find(r,a)
    while (n.left != null)
        delete (n.left)
    delete(n)
```

```
find(r,a)
    if r = null or r.key = a
        return r
    if a < r.key
        return find(r.left, a)
    return find (r.right, a)
```

The time complexity of Insert is O(D) or Ω (log n) because you start at the top and work your way down the tree until you find where to insert. The time complexity of Delete is O(D) or Ω (log n) because you swap the element you have to delete until it gets to the bottom of the tree and then you delete it. The time complexity of Block Delete is O(Dn) or Θ (n log n) because it does a delete on all of the nodes that are either n or to the left of n.