CS 101, Assignment 5

Christopher Huynh, Oliver Rene

March 2016

1 Question 1

DeleteFromHeap(heap A,node k)
  swap(A[k],A[end])
  A[end] = null;
  Heapify(A,k)

Heapify(heap A, node k)
  if(a[k] == null)
    return
  if(A[2k] $\leq$ A[2k+1])
    swap(A[k],A[2k])
    Heapify(A,A[2k])
  else
    swap(A[k],A[2k+1])
    Heapify(A[k],A[2k+1])

Because recursion of Heapify is run on either the left children or right children and swap should run in O(1) time, the reccurence relation is $T(n) \leq T(\frac{n}{2}) + \theta(1)$
By Master Theorem where A=1, B=2, and C=0, $C = log_b a \implies 0 = log_2 1$. Since Case 2 of Master Theorem is satisfied, $T(n) = \theta(n^c log(n)) = \theta(log(n))$

2 Question 2

KthLargestArray(array A, number k)
  heap = buildMaxHeap(A)
  #initialize returnArray
  for i to k
    returnArray[i]=extractMax(heap)
  return returnArray

extractMax(Heap A)
  tmp=A[1]
  swap(A[1],A[end])
  A[end] = null;
  Heapify(A,1)
  return tmp

buildHeap(array A)
  for(int i=A(length);i $\geq$ 0;i–)
    Heapify(A,i)

```
Heapify(heap A, index k)
  if(A[k]==null)
    return
  if(A[2k]≥A[2k+1])
    swap(A[k],A[2k+1])
    Heapify(A,2k+1)
  else
    swap(A[k],A[2k])
    Heapify(A,2k)
```

In running buildHeap, Heapify has a different cost at every level. At the bottom most level of the heap, there are 2^h nodes running heapify 0 times because they are already at the bottom and there is nothing to do. At the level before there are 2^{h-1} nodes running heapify 1 time because there is only one level below that it can recursivvely run. The level before that has 2^{h-2} nodes running heapify 2 times and so on. The total cost would be similar to 0 + 1 + 2 + ... + n. Where the cost of the root is n. Informally under Master Theorem, the cost of the root dominates and BuildHeap runs in O(n).
Because a recursion only happens on one child and all other operations are in O(1), the recursion relation is T(n) ≤ T(n/2) + O(1). By Master Theorem where A=1, B=2, and C=0, $C = log_b a \implies 0 = log_2 1$. Since Case 2 of Master Theorem is satisfied, $T(n) = \theta(n^c log(n)) = \theta(log(n))$. Therefore, extractMax runs in O(logn). Because we run extractMax k times, the total time complexity of KthLargestArray is O(n+klog(n))

3 Question 3

```
Heapsort(array A)
  buildminheap(A)
  n = length(A)
  for i to n
    returnArrray[i]=extractmin(A)
  return returnArray
```

As an example we have the input array $[1, 2, 3_1, 3_2, 4]$. After extracting the first minimum, we are left with the array $[2, 3_2, 3_1, 4]$. Extracting another minimum, we have $[3_2, 3_1, 4]$. The next extraction extracts 3_2 therefore is not stable, because the end result is $[1, 2, 3_2, 3_1, 4]$.

4 Question 4

```
BlockDelete(tree tree, key a)
  if(tree.root==null)
    return
  if(tree.root.key ≤ a)
    tree.root.leftchild = null;
    tree.root = tree.root.rightchild
    BlockDelete(tree.root, a)
  else
    BlockDelete(tree.root.leftchild,a)
```

Because a recursion only happens on one child and all other operations are in O(1), the recursion relation is T(n) ≤ T(n/2) + O(1). By Master Theorem where A=1, B=2, and C=0, $C = log_b a \implies 0 = log_2 1$. Since Case 2 of Master Theorem is satisfied, $T(n) = \theta(n^c log(n)) = \theta(log(n))$. Therefore, BlockDelete runs in O(logn)

```
Insert(tree tree, int key)
  if(tree.root==null)
    tree.root.key=key
  else
    if(key < tree.root.key)
      Insert(tree.root.leftchild,key)
    else
      Insert(tree.root.rightchild,key)
```

Because a recursion only happens on one child and all other operations are in O(1), the recursion relation is T(n) $\leq$ T(n/2) + O(1). By Master Theorem where A=1, B=2, and C=0, $C = log_b a \implies 0 = log_2 1$. Since Case 2 of Master Theorem is satisfied, $T(n) = \theta(n^c log(n)) = \theta(log(n))$. Therefore, Insert runs in O(logn)

```
Delete(tree tree, int key)
  if(tree.root.key == key)
    if(tree.root.leftchild == null && tree.root.rightchild == null)
      tree.root = null;
    else if(tree.root.leftchild == null && tree.root.rightchild != null)
      tree.root = tree.root.rightchild
    else if(tree.root.leftchild != null && tree.root.rightchild == null)
      tree.root = tree.root.leftchild
    else
      pre = findPredecessor(tree.root)
      swap(pre,tree.root)
      pre = null
  else if(key < tree.root.key)
    Delete(tree.leftchild,key)
  else
    Delete(tree.rightchild,key)
```

Because a recursion only happens on one child and all other operations are in O(1), the recursion relation is T(n) $\leq$ T(n/2) + O(1). By Master Theorem where A=1, B=2, and C=0, $C = log_b a \implies 0 = log_2 1$. Since Case 2 of Master Theorem is satisfied, $T(n) = \theta(n^c log(n)) = \theta(log(n))$. Therefore, Delete runs in O(logn)