

# COT 5405 - Spring 2009

## Homework #2 Solutions

February 6, 2009

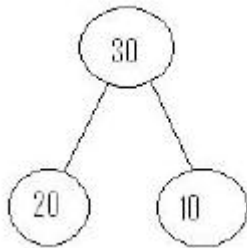
### Grading Policy:

- Please contact TA Chunchun Zhao by email (cot5405sp09@cise.ufl.edu) or in his office hours for any grading issues.
- Maximum score is 100 points.
- Each completed question worth 10 points.
- Partial credit is given if you dont answer a question completely.
- Problem 2 is graded for 50 points.
- Please notify the TA if you find anything wrong in this solution.

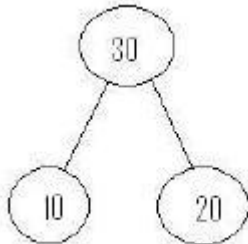
### Problem 1 (Problem 6-1 in page 142)

#### Part a:

These two examples do not always create same Heap when run on the same input array. We can show this with a counter example. Lets consider the array  $A = [10, 20, 30]$ . Now if we construct Heap using  $BUILD\_MAX\_HEAP(A)$ , we get following Heap.



On the other hand. if we construct the Heap using  $BUILD\_MAX\_HEAP'(A)$ , we get the following Heap.



This shows that, these two algorithms do not produce the same Heap, when used same input array.

## Part b:

In the worst case, the input array is given in increasing order. Each call to MAX\_HEAP\_INSERT will push the inserting node (i) from bottom to root. So, worst case running time is

$$\begin{aligned} &= \sum_{i=1}^n \lg i \\ &= \lg n! \\ &= \Theta(n \lg n) \end{aligned}$$

## Problem 2 page 192

### Part a: (Exercise 9.3-5)

Solution 1:

We assume that we have a subroutine MEDIAN(A,p,r) to find median value of subarray A[p..r] with worst case cost O(n). following algorithm finds the ith smallest element in A[p..r].

```
Selection(A,p,r,i)
  if p == r
    return A[p];
  value = MEDIAN(A,p,r);
  q = Partition(value): Rearrange the array A[p..r] into two subarrays A[p..q-1],A[q+1..r]
  such that A[q] = value and each element of A[p..q-1] is less than or equal to A[q],
  A[q] is less than or equal to each element of A[q+1..r]
  ( check algorithm in P146 , running time O(n))
  k = q - p + 1
  if i == k
    return A[q];
  else if i < k
    return Selection(A, p, q - 1, i)
  else
    return Selection(A, q + 1, r, i - k)
```

Every time we divide the array into two each size parts and do recursion with one of them, hence worst case running time is –

$$\begin{aligned} T(n) &\leq T\left(\frac{n}{2}\right) + O(n) \\ T(n) &= O(n) + O\left(\frac{n}{2}\right) + O\left(\frac{n}{4}\right) + \dots + O(1) \\ T(n) &= O(n) \end{aligned}$$

Solution 2:

We assume that we have a subroutine MEDIAN(A) to find median value of array A[0..n] with worst case cost O(n). following algorithm finds the kth smallest element in A.

```
Selection(A, k)
  min = MIN( A )
  max = MAX( A )
  if  $k < \frac{n}{2}$ 
    for i = n to (2n-2k-1)
      A[i] = min-1;
  end for
```

```

else
    for i = n to (2k-1)
        A[i] = max+1;
    end for
return MEDIAN(A)

```

Finding the minimum min and maximum max of the array cost  $O(n)$ . Hence worst case running time is

$$T(n) \leq O(n) + O(n) = O(n)$$

### Part b: (Exercise 9.3-7)

Following algorithm finds  $k$  closest elements from the median of an array.

- Find the median  $m$  of the array  $A$ .
- Compute absolute difference of each array element  $A[i]$  from the median  $m$  and create array  $D[i] = \text{abs}(A[i] - m)$ .
- Find  $k$  - th smallest element of array  $D$ . Assume that this value is  $dk$ .
- For each  $A[i], i = 1 \dots n$ 
  - if  $D[i] \leq dk$  report  $A[i]$

Each step of the above algorithm can be done in  $O(n)$  time. Hence, runtime of the algorithm is  $O(n)$ .

### Part c: (Exercise 9.3-8)

Here, arrays  $X$  and  $Y$  are sorted array of size  $n$ . So, median of both of the arrays will be the middle element of the array.

Now, if  $\text{median}(X) = \text{median}(Y)$ , then this element will be the median of all the elements of  $X$  and  $Y$ .

If  $\text{median}(X) < \text{median}(Y)$ , then median of  $2n$  elements will be found in subarray  $\text{median}(X)$  to  $X[n]$  and  $Y[1]$  to  $\text{median}(Y)$ .

If  $\text{median}(X) > \text{median}(Y)$ , then median of  $2n$  elements will be found in subarray  $X[1]$  to  $\text{median}(X)$  and  $\text{median}(Y)$  to  $Y[n]$ .

Above algorithm will take at most  $\log n$  time. Because each time size of the problem is reducing by half.

## Problem 3 (Problem 11-3 in page 251)

### Part a:

From the problem statement, we can write the hash function as -

$$\begin{aligned}
 H(k, j) &= H(k) + \frac{j(j+1)}{2} \\
 &= H(k) + \frac{j^2}{2} + \frac{j}{2}
 \end{aligned}$$

So, we have  $c_1 = \frac{1}{2}$  and  $c_2 = \frac{1}{2}$ .

**Part b:**

We prove this by contradiction. Assume we have two probes  $i$  and  $j$  resulting in same position. where  $i \neq j$ . Therefore,

$$\begin{aligned}
 H(k, i) &= H(k, j) \\
 (H(k) + \frac{i^2}{2} + \frac{i}{2}) \bmod m &= (H(k) + \frac{j^2}{2} + \frac{j}{2}) \bmod m \\
 \frac{i^2}{2} + \frac{i}{2} - \frac{j^2}{2} - \frac{j}{2} &= m * c \\
 (i - j) * (i + j + 1) &= 2 * m * c
 \end{aligned}$$

Using the assumption that  $m$  is a power of 2,

Because exactly one of the factors  $j - i$  and  $j + i + 1$  is even,  $2 * m$  must divide one of the factors.

i) if  $j - i$  is even,  $j + i + 1$  is odd.  $j - i = 2 * m * k$  ( $k$  is positive integer) which is bigger than  $m$ , contradiction

ii) ( $j - i$ ) is odd.  $j + i + 1 = 2 * m * k$  suppose  $j > i$  then  $2 * j >= j + i + 1 = 2 * m * k$  ( $k$  is positive integer) then  $j > m$  which is impossible because  $j$  should always smaller than  $m$ .

**Problem 4**

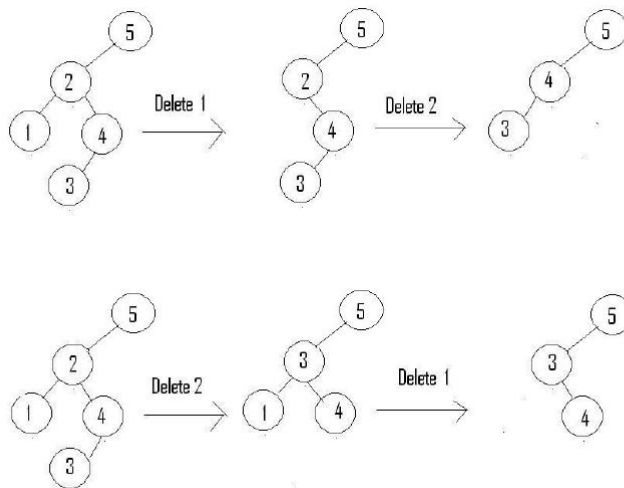
**Part a: (Exercise 12.2-5)**

Lets assume that a node,  $N$  in the binary search tree has two child. Its predecessor lies in its left subtree and is the rightmost node of this subtree. So, if predecessor of  $N$  has a right child,  $R$  which is larger than its parent and less than  $N$ . Value of  $R$  is between  $N$  and predecessor of  $N$ . This is a contradiction. So, predecessor of  $N$  cannot have a right child.

Again. its successor lies in its right subtree and is the leftmost node of this subtree. So, by similar argument we can show that, successor of  $N$  cannot have a left child.

**Part b: (Exercise 12.3-5)**

Deletion operation is not commutative. We can show this by counter example.



## Problem 5

### Part a: (Exercise 13.1-6)

The largest possible number of internal nodes in a red-black tree with black height is  $2^{2k}-1$ . The smallest possible number is  $2^k-1$ .

### Part b: (Exercise 13.2-4)

1) We can convert any binary search tree with  $n$  nodes into a right going chain of length  $n$  using at most  $O(n)$  right rotation operations.

As shown in figure 13.2 in page 278, every right rotation will increase one node on the right going chain while keeping all the nodes on the right going chain still remain on the path. There can be at most  $n-1$  nodes to be added to the right going chain by performing at most  $n-1$  right rotations.

2) We can convert any binary search tree  $T_1$  to any binary search tree  $T_2$ .

According to 1), we can perform at most  $n-1$  right rotations to convert  $T_1$  to right going chain of length  $n$ , same strategy for converting  $T_2$  to the same right going chain.

Since each right rotation has a corresponding left rotation, we can do the sequence of left rotations inversely from a right going chain of length  $n$ , converting into binary search tree  $T_2$  with  $n$  nodes using at most  $n-1$  left rotations.

Combining these two we observe that any arbitrary  $n$ -node binary search tree can be transformed into any other arbitrary  $n$ -node binary search tree using  $O(n)$  rotations.