CS240 Assignment 2

Nissan Pow

June 13, 2006

Problem 1

(a)

```
DFS(Vertex v, int level, Vertex start)
        Mark v
        level = level + 1
        for each vertex, w, adjacent to v do
            if (w = start) and (level > 2)
                return false
            endif
            if w is unmarked then
                DFS(w)
            endif
        endfor

    Main()
        for each vertex v in graph G do
            DFS(v, 0, v)
        endfor
        return true
```

Assumption: the graph is undirected.
A tree is a connected graph with no cycles. In the algorithm, we perform a DFS on each vertex in the graph to check for cycles. We know when we've hit a cycle if we return to our starting node and the length of the path is greater than 2 (cannot have cycle between only 2 nodes, since we're assuming no multiple edges). So if we find a cycle, we return false; otherwise when we have finished checking each node, we return true.

(b)

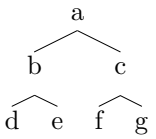
Yes, the data structure matters. Adjacency lists are more efficient as we can obtain all the neighbours of a node in O(1) time (assuming that all the nodes

are unique and we hash them), whereas with an adjacency matrix the same operation will take O(n) time. We need to obtain all the neighbours of a node to perform the algorithm above.

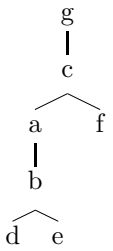
(c)

Assumption: the graph is undirected.
The problem would be pretty much the same. We simply run the same algorithm from part (a) for only node x instead of every vertex in G. This is so because a tree rooted at node x can be rearranged to be rooted at another node y, with y $\neq$ x. For example, tree (b) below is tree (a) with new root node g:

a.



b.



Problem 2

(a)

Yes the encoding is unambiguous.

(b)

```
printDFU(Node n) {
    output n
    if n.hasChildren() {
        foreach child of n { // sorted from left to right
            output n
        }
        foreach child of n { // sorted from left to right
            output n
        }
    }
}
```

Problem 3

(a)

1. | 4 | 2 | 7 | 5 | 1 | 8 | 9 | 6 | 3 |
   |---|---|---|---|---|---|---|---|---|

2. Compare 5 with min{6,3}. Since $3 < 5$, swap 3 and 5. 5 is now a leaf, so nothing more to check.

4	2	7	3	1	8	9	6	5

3. Compare 7 with min{8,9}. Since $7 < 8$, nothing to do.

4. Compare 2 with min{3,1}. Since $1 < 2$, swap 1 and 2. 2 is now a leaf, so nothing more to check.

4	1	7	3	2	8	9	6	5

5. Compare 4 with min{1,7}. Since $1 < 4$, swap 1 and 4. Next, compare 4 with min{3,2}. Since $2 < 4$, swap 2 and 4. 4 is now a leaf, so nothing more to check.

1	2	7	3	4	8	9	6	5

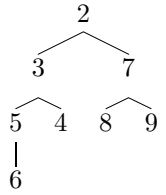
6. Done.

(b)

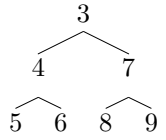
Complexity of heapify is O(n).

(c)

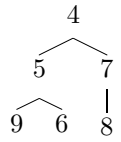
1. Remove 1. Move 5 to the top of the heap. Proceed to sift down 5. Compare 5 with min{2,7}; since $2 < 5$, swap 2 and 5. Compare 5 with min{3,4}; since $3 < 5$, swap 3 and 5. Compare 5 with 6; 5 is bigger, so we're done. The resulting heap:

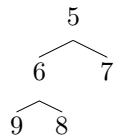
2. Remove 2. Move 6 to the top of the heap. Proceed to sift down 6. Compare 6 with min{3,7}; since $3 < 6$, swap 3 and 6. Compare 3 with min{5,4}; since $4 < 6$, swap 6 and 4. 6 is now a leaf, so we're done. The resulting heap is:

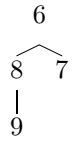
3. Remove 3. Move 9 to the top of the heap. Proceed to sift down 9. Compare 9 with min{4,7}; since 4 < 9, swap 4 and 9. Compare 9 with min{5,6}; since5 < 9, swap 5 and 9. 9 is now a leaf, so we're done. The resulting heap is:



4. Remove 4. Move 8 to the top of the heap. Proceed to sift down 8. Compare 8 with min{5,7}; since 5 < 8, swap 5 and 8. Compare 8 with min{9,6}; since 6 < 8, swap 6 and 8. 8 is now a leaf, so we're done. The resulting heap is:



5. Remove 5. Move 8 to the top of the heap. Proceed to sift down 8. Compare 8 with min{6,7}; since 6 < 8, swap 6 and 8. Compare 8 with 9; since 9 > 8, we're done. The resulting heap is:



(d)

The complexity of the algorithm used to remove a single key is $O(\log n)$. Whenever we remove a key, we will need to "sift down" the key as far as possible. This will require checking the node with max{node.left, node.right}. If the node is a leaf, there is nothing to check so we're done. Else, if node > max{node.left, node.right}, swap node with max{node.left, node.right} and repeat the procedure. Thus in the worst case, we will need to do 2×(height of tree) comparisons, and the height of the tree is $\log n$. Thus the complexity of the algorithm is $O(\log n)$.

Problem 4

(a)(i)

```
extractMax(Matrix A[m][n])
    max = A[1][1]
    i = 1
    j = 1
```

```
while i ≤ m and j ≤ n and A[i][j] ≠ -∞ do
    if i ≠ m and j ≠ n then
        if A[i][j+1] > A[i+1][j] then
            A[i][j] = A[i][j+1]
            j = j+1
        else
            A[i][j] = A[i+1][j]
            i = i+1
        endif
    else if i=m then
        A[i][j] = A[i][j+1]
        j = j+1
    else if j = n then
        A[i][j] = A[i+1][j]
        i = i+1
    endif
endwhile
A[i][j] = -∞
return max
```

The maximum element is located at A[1][1], so we store this in some variable for returning later. When we extract the maximum element, we must then choose another element to replace it. By the properties of the 2WOM, we know that the candidate for the next biggest element is either at A[1][2] or A[2][1]. Without loss of generality, let us say the max is A[1][2]. So we move A[1][2] to A[1][1]. So now we must choose another element to fill in A[1][2]. And by the properties of 2WOM, we know that the candidates to move to A[1][2] are either A[1][3] or A[2][2], so we choose the max of those and move it to A[1][2]. This process is repeated until we reach an element that is -∞ or we have reached the last element in the matrix. Then we just fill in our current position with -∞, return the max and we're done.
At each step of the loop, we are either going down or across the matrix. Thus in the worst case, we will need to perform m+n iterations. Hence the complexity of the algorithm is $O(m + n)$.

(a)(ii)

```
insert(x)
    for j=n; j>0; j=j-1
        if A[m][j] ≠ -∞ then
            break
        endif
    endfor
    if A[m][j] ≠ -∞ then
        j=j+1
```

```
endif
for i=m; i>0; i=i-1
    if A[i][j] ≠ -∞ then
        break
    endif
endfor
if A[m][j] ≠ -∞ then
    i=i+1
    A[i][j] = x
    while A[i][j] > min{A[i-1][j], A[i][j-1]}
        swap (A[i][j], min{A[i-1][j], A[i][j-1]})
        if A[i-1][j] < A[i][j-1] then
            i = i-1
        else
            j = j-1
        endif
    endwhile
else // A[i][j] = -∞ // at top of column
    if j = 1 then
        A[1][1] = x
    else
        A[1][j] = x
        while j>1 and A[1][j] > A[1][j-1]
            swap (A[1][j], A[1][j-1])
            j = j-1
        endwhile
    endif
endif
```

First we search for the leftmost, bottommost element that is not -∞ if it exists. We do this by starting from the bottom row and going left, then up. If we arrive at a position where the element is -∞, then either the matrix is empty and we put x at A[1][1], or we have hit the top of a column and we place our element at this position and proceed to move it across to the left as far as possible. If we find an element that is not -∞, then we place our element below it and compare our element with the one immediately above it and to the left. If the element is greater than the min of those, we swap them and proceed in this fashion until no more swaps can be made. This indicates that our element is in the correct position. Choosing the min of the 2 elements guarantees that the 2WOM constraints are satisfied. This algorithm will run in O(m+n) time, since we only need to go across the length of the matrix once, and up the length of the matrix once.

(b)

Let $m = n = \lceil \sqrt{N} \rceil$.
Using the 2WOM, the asymptotic running time for HeapSort is:

$$
\begin{aligned}
& N(m+n) + N(m+n) \qquad \text{// (insert N elements) + (removeMax N times)} \\
= \;& 2N(2\sqrt{N}) \\
= \;& 4N^{\frac{3}{2}} \\
\in \;& O(N^{\frac{3}{2}})
\end{aligned}
$$

(c)

```
findMin()
    min = -∞
    for (j=n; j>0; j=j-1) // find leftmost element in last row that's not -∞
        if A[m][j] ≠ -∞ then
            min = A[m][j]
            break
        endif
    for (i=m; i>0; i=i-1) // find bottom-most element in column j that's not
-∞
        if A[i][j] ≠ -∞ then
            min = A[i][j]
            break
        endif
    // proceed right and then up to find all the "leaf" elements
    for (j=j+1; j≤n; j=j+1)
        for (i=i-1; i>0; i=i+1)
            if A[i][j] ≠ -∞ and A[i][j] < min // found a leaf
                min = A[i][j]
                break
            else if A[i][j] ≠ -∞ then
                break
            else if i=1 then // no more elements to the right; this element is -∞
                return min
            endif
        endfor
    endfor
    return min
```

Note that the minimum element must be a "leaf" element ie all the elements under it have value -∞. First we find the column, j, of the leftmost element in

the last row that's not -∞ if it exists. Then we find the row, i, of the bottom-most element in column j that's not -∞. We then proceed across and up, going up as much as possible until we find an element that's not -∞. If we reach row 1 without finding any element that's not -∞, we return the min since there are no more elements to the right of the one that we're currently at. Otherwise, we have found a "leaf" element so we compare it to the min. If the current element is smaller than the min, the min is now the value of this new element. This algorithm is correct since it finds the smallest elements in each row and then returns the min of those, which is essentially the minimum element of the entire matrix.

The complexity of this algorithm is:

(n+m) to find the leftmost,bottom-most element that's not -∞, and (n+m) for the other leaves.

$$\begin{aligned} & (n+m)+(n+m) \\ = \; & 2n+2m \\ = \; & 2(n+m) \\ \in \; & O(m+n) \end{aligned}$$

Problem 5

(b)

```
uint createNode(uint i) {
  char currChar_c, lastChar_c;
  lastChar_c = input_n.at(i);
  for (uint j=i+1; j<input_n.length(); ++j) {
    currChar_c = input_n.at(j);
    if (lastChar_c == '(' && currChar_c == ')') {
      return j;
    }
    else if (lastChar_c == '(' && isalnum(currChar_c)) {
      cout << lastChar_c << currChar_c;
      j++;
      j = createNode(j);
      cout << ')';
    }
    lastChar_c = input_n.at(j);
  }
}
```

This algorithm traverses the input string, character by character, using recursion to output the binary trees. The recursive subroutines return the index to the element that they have reached, so the main method is able to jump ahead to that location, ensuring that we only traverse the input string once.
Therefore the running time is O(n).