Data Structure and Algorithm, Spring 2017
Final Examination Solution

Problem 1. In each of the following question, please specify if the statement is **true** or **false**. If the statement is true, explain why it is true. If it is false, explain what the correct answer is and why. (18 points. 1 point for true/false and 2 points for the explanation for each question)

1. False. **Red** should be replaced with **black** to make it correct.
2. False. The reason is similar to above.
3. True. All leaves have the same depth.
4. False. **Fix** should be replaced with **open** to make it correct.
5. False. It's impractical to make accurate estimation.
6. True. Elements are placed in reverse order to the output array.

Problem 2. Fill the blanks / short answer questions. (26 points)

1. (3 points)

   (a) Steps to reproduce

   (b) What you expected to see

   (c) What you saw instead

2. (4 points)
   B[C[A[j]]] = A[j];
   C[A[j]]−−;

3. (4 points)
   Yes, it will still work correctly. Since $C[A[j]]$ determines where $A[j]$ should put, j from 1 to $A.lenght$ or $A.length$ downto 1 are both work. However, if it is changed to j from 1 to $A.length$, it will become unstable.

4. (3 points)

   (a) Backup and Restore

   (b) Synchronization

   (c) Short-term undo

   (d) Long-term undo

   (e) Track Changes

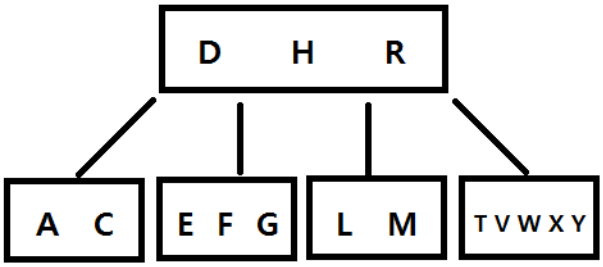
   (f) Track Ownership

   (g) Sandboxing

   (h) Branching and merging

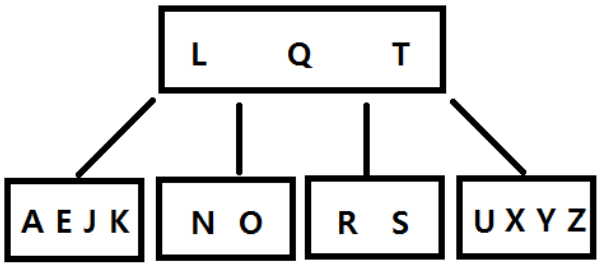
   You only have to answer three of them, or use your own words to briefly explain what these functions do.

5. (6 points)

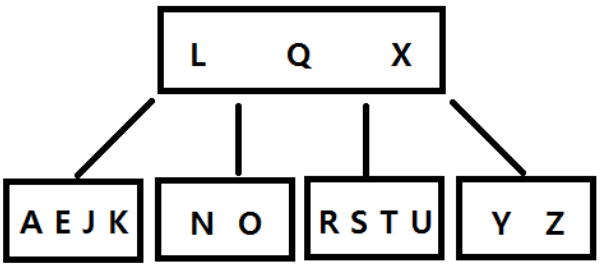


6. (6 points)

Solution 1 :



Solution 2 :



Problem 3. Graph and Depth-First Search (32 points)

1. Showed as follow:

	q	r	s	t	u	v	w	x	y	z
discovery time	1	17	2	8	18	3	4	9	13	10
finishing time	16	20	7	15	19	6	5	12	14	11
parent	NIL	NIL	q	q	r	s	v	t	t	x

2. (a) $(q, s), (s, v), (v, w), (q, t), (t, x), (x, z), (t, y), (r, u)$

   (b) $(w, s), (z, x), (y, q)$

   (c) (q, w)

   (d) $(r, y), (u, y)$

3. - **if**:
     Since $v.d \leq u.d$, we know that u must be either a descendant of v or in another subtree which will be traversed later when dfs. Since $u.f \leq v.f$, we know that u must be either a descendant of v or in another subtree which is already traversed when dfs. Combining above, we know that u must be a descendant of v. Therefore, by the definition, (u, v) is a back edge.
   - **only if**:
     If (u, v) is a back edge, it's clearly that in the depth-first-search tree, u is a descendant of v. Therefore, when dfs, we will first encounter v before encounter u. It's $v.d < u.d$. Obviously, we have $u.d < u.f$. Then, since u is a descendant of v, we should first finish u before finishing v. It's $u.f < v.f$. Combining all the inequation, we have $v.d < u.d < u.f < v.f$.

4. For an edge (u, v) in an undirected graph, either u is an ancestor of v or v is an ancestor of u.
   Reason: Assume (u, v) is an edge in an undirected graph. Since the graph is undirected, this edge leads both from u to v and from v to u. Without loss of generality, assume vectex u is traversed before vertex v is. As DFS can go from u to v through this edge, vertex v must be an descendant of u.
   Because an edge (u, v) leads to an ancestor-descendant relationship between vertex u and v, every edge must be classified as the first three types. So there is no cross edge on an undirected graph.

5.
   - **adjacency lists**: $\Theta(|V| + |E|)$
   - **adjacency matrix**: $\Theta(|V|^2)$

6. There are many answers to this question, here are two of them:

```
DFS-Visit(G, u)
 1  stack.init()
 2  stack.push(u)
 3  while stack.IsEmpty() ==FALSE
 4        u = stack.top()
 5        if u.color ==WHITE
 6              time = time + 1
 7              u.d = time
 8              u.color = GRAY
 9              stack_tmp.init()
10              for each v ∈ G.Adj[u]
11                    if v.color ==WHITE
12                          v.π = u
13                          stack_tmp.push(v)
14              while stack_tmp.IsEmpty() ==FALSE
15                    stack.push(stack_tmp.top())
16                    stack_tmp.pop()
17        else if u.color ==GRAY
18              stack.pop()
19              u.color = BLACK
20              time = time + 1
21              u.f = time
22        else if u.color ==BLACK
23              stack.pop()
```

```
DFS-VISIT(G, u)
 1  stack.init()
 2  stack.push((u, 0))
 3  while stack.IsEmpty() ==FALSE
 4        u, id = stack.top()
 5        stack.pop()
 6        if id == 0
 7              time = time + 1
 8              u.d = time
 9              u.color = GRAY
10        finish =true
11        for i ∈ range(id, G.Adj[u].size())
12              v = G.Adj[u][i]
13              if v.color ==WHITE
14                    finish =false
15                    v.π = u
16                    stack.push((u, i + 1))
17                    stack.push((v, 0))
18                    Break
19        if finish ==true
20              u.color = BLACK
21              time = time + 1
22              u.f = time
```

Problem 4. Heap (20 points)

The procedure BUILD-MAX-HEAP makes use of MAX-HEAPIFY in a bottom-up manner to convert the array $A[1\,..\,n]$, where $n = A.\mathit{length}$, into a max-heap.

1. Please show that BUILD-MAX-HEAP takes $O(n)$ time to complete. (6 points)

   For a heap with depth=H, it takes 0 action to modify the leaves; 1 action to modify nodes at depth (H-1); 2 actions for nodes at depth (H-2) .... and at most H actions to modify the root at depth 0.

   Let $S = H + 2 \times (H-1) + 2^2 \times (H-2) + ... + 2^{H-1} \times 1$

   $2S = 2^1 \times H + 2^2 \times (H-1) + 2^3 \times (H-2) + ... + 2^H \times 1$

   $2S - S = 2^{H+1} - 2 - H$

   Since $H = \log n$

   $S = 2^{\log n+1} - 2 - \log n = O(n)$ Therefore, the Build-Max-Heap function takes $O(n)$ to complete.

2. TA Hsun came up with a different method to build a max heap - inserting the elements one by one into an initially empty max heap. The method is implemented as BIULD-MAX-HEAP-I. Please show that this method can in fact take up to $\Omega(n \log n)$ to complete in the worst case. (6 points)

   For i-th insertion, the worst case takes $O(\log i)$ to complete, where i is the current size of the heap. That is, it takes 0 exchange to insert the element on the 0th layer, at most 1 exchange to insert the 2^1 elements on the 1st layer, at most 2 exchanges to insert 2^2 elements on the 2nd layer, and so on.

   Let S be the total time. $S = 1 \times 0 + 2^1 \times 1 + 2^2 \times 2 + 2^3 \times 3 + ... + 2^{h-1} \times (h-1)$

   $2S = 2^2 \times 1 + 2^3 \times 2 + 2^4 \times 3 + ... + 2^{h-1} \times (h-2) + 2^h \times (h-1)$

   $S = -2^1 - 2^2 - 2^3 - ... - 2^{h-1} + 2^h \times (h-1)$

   $= -2^h + 2 + 2^h \times (h-1) = 2^h \times (h-2) + 2$

   Since $h = O(\log n)$, $S = n(\log n - 2) + 2 = \Omega(n \log n)$

3. Give an example to show that BUILD-MAX-HEAP and BUILD-MAX-HEAP-I can generate different heaps. (4 points)

   Let the number sequence be [1, 2, 3]. With Build-Max-Heap function, the original root, 1, will exchange with the larger right child, 3, resulting in a heap [3, 2, 1]. On

the other hand, with Build-Max-Heap-I, by inserting the numbers one-by-one, it takes two actions to complete the heap. First, we insert 1 to an empty heap. Then, to insert 2, we exchange the root, 1, with its left child, 2. Then, when inserting 3, we exchange the root, 2, with its right child, 3, to complete the max-heap, resulting in a heap [3, 1, 2].

4. Is heapsort a stable algorithm? If yes, please prove that it is stable. If not, please give a counter-example. (4 points)

   An algorithm is stable if it preserves the original order of the equal keys. Heapsort is not a stable algorithm. For example, for a number sequence [1a, 2, 1b], the max-heap is [2, 1a, 1b]. After 2 is removed, 1b is placed at the root, and thus, will be removed before 1a. The sorted sequence will be [2, 1b, 1a].

```
PARENT(i)
1  return ⌊i/2⌋

LEFT(i)
1  return 2i

RIGHT(i)
1  return 2i + 1
```

```
MAX-HEAPIFY(A, i)
 1  l = LEFT(i)
 2  r = RIGHT(i)
 3  if l ≤ A.heap-size and A[l] > A[i]
 4       largest = l
 5  else
 6       largest = i
 7  if r ≤ A.heap-size and A[r] > A[largest]
 8       largest = r
 9  if largest ≠ i
10       exchange A[i] with A[largest]
11       MAX-HEAPIFY(A, largest)
```

```
BUILD-MAX-HEAP(A)
1  A.heap-size = A.length
2  for i = ⌊A.length/2⌋ downto 1
3      MAX-HEAPIFY(A,i)
```

```
MAX-HEAP-INSERT(A,key)
1  A.heap-size = A.heap-size + 1
2  i = A.heap-size
3  A[i] = key
4  while i > 1 and A[PARENT(i)] < A[i]
5      exchanage A[i] with A[PARENT(i)]
6      i = PARENT(i)
```

```
BUILD-MAX-HEAP-I(A)
1  A.heap-size = 1
2  for i = 2 to A.length
3      MAX-HEAP-INSERT(A, A[i])
```

Problem 5. Hashing (12 points)

Consider inserting the keys $10, 22, 31, 4, 15, 28, 17, 88, 59$ into a hash table of length $m = 11$ using the following methods. Show the results after the insertion of each number (i.e., for each method, there will be 9 drawings).

1. Linear hashing (3 points)

initial	0	1	2	3	4	5	6	7	8	9	10
10											10
22	22										10
31	22									31	10
4	22				4					31	10
15	22				4	15				31	10
28	22				4	15	28			31	10
17	22				4	15	28	17		31	10
88	22	88			4	15	28	17		31	10
59	22	88			4	15	28	17	59	31	10

2. Quadratic probing with $c_1 = 1$ and $c_2 = 3$ (3 points)

initial	0	1	2	3	4	5	6	7	8	9	10
10											10
22	22										10
31	22									31	10
4	22				4					31	10
15	22				4				15	31	10
28	22				4		28		15	31	10
17	22			17	4		28		15	31	10
88	22		88	17	4		28		15	31	10
59	22		88	17	4		28	59	15	31	10

3. Double hashing with $h_1(k) = k$ and $h_2(k) = 1 + (k \mod (m-1))$ (3 points)

initial	0	1	2	3	4	5	6	7	8	9	10
10											10
22	22										10
31	22									31	10
4	22				4					31	10
15	22				4	15				31	10
28	22				4	15	28			31	10
17	22			17	4	15	28			31	10
88	22			17	4	15	28	88		31	10
59	22		59	17	4	15	28	88		31	10

4. Chaining (3 points)

initial	0	1	2	3	4	5	6	7	8	9	10
10											10
22	22										10
31	22									31	10
4	22				4					31	10
15	22				4 15					31	10
28	22				4 15		28			31	10
17	22				4 15		28 17			31	10
88	22 88				4 15		28 17			31	10
59	22 88				4 15 59		28 17			31	10

Problem 6. N-way Merge (30%)

Let's recall the typical merge sort for this problem. In the merging phase, we have two sorted arrays of the same size. With two pointers at the start of each array, we iteratively do: (1) check which pointer points to the smaller number, (2) push that number into the output array, and (3) move the pointer one step further until the two arrays have been traversed thoroughly.

Naive Merging

In a similar fashion, we can solve the N-way merge problem with N pointers at the start of the N arrays respectively. Therefore, the 3 steps mentioned in the paragraph above would become:

1. Find the smallest of the N numbers being currently pointed to.
2. Output that number to STDOUT.
3. Move the corresponding pointer 1 step further.

This algorithm should be pretty straightforward and can be implemented easily. We give a pseudocode snippet below.

```
function merge(arrays, N, M):
    pointers[N] = {0}
    while True:
        # find the smallest
        smallest = 0
        smallest_at = -1
        for n = 0 to N-1:
            if pointers[n] < M:
                if smallest_at == -1 or
                    arrays[n][pointers[n]] < smallest:
                    smallest = arrays[n][pointers[n]]
                    smallest_at = n
        # output
        print "%d " % (smallest)
        # move the pointer and check if complete
```

```
        pointers[smallest_at] += 1
        if all pointers == M:
            break
```

This algorithm could get you a score of 60 out of 100 for this problem. For the test cases 7 to 10, it is very likely that you receive a TLE, so let's dive into time complexity analysis then. First of all, the while loop will run exactly $N \times M$ times because we increase one of the pointers by 1 during each iteration. In the first section inside the while loop, "find the smallest", it costs $O(N)$. The second section is outputting 1 number to standard output, so it's $O(1)$. For the last section, "move the pointer and check if complete", it is going to take $O(N)$ time, *i.e.*, the number of pointers we have. Consequently, the total time complexity of this algorithm is $O(N^2M)$.

Improving

To reduce the running time, we can utilize ***min-heap***. With the help of a min-heap, the "find the smallest" process will be $O(1)$. We give an example pseudocode below using min-heap.

```
function merge(arrays, N, M):
    # initialize
    pointers[N] = {0}
    heap = create_min_heap()
    for i = 0 to N-1:
        heap.insert(arrays[i][pointers[i]])
    # main loop begins
    while heap is not empty:
        smallest = heap.pop()
        print "%d " % (smallest.value)
        # move the pointer and insert if needed
        pointers[smallest.at] += 1
        if pointers[smallest.at] < M:
            heap.insert(arrays[smallest.at][pointers[smallest.at]])
```

Now we can have a little time complexity analysis. First of all, the number of nodes in our min-heap is going to be limited by N because there are exactly N nodes after the *initialize* phase and all insert operation will always follow a pop operation inside the

while loop. That results in a $O(log(N))$ time complexity for every insert operation of the min-heap. Therefore, considering inserting and popping every number in those arrays, the total time complexity is: $O(N \times M \times log(N))$. Generally speaking, this algorithm will result in an AC status.

Note

The min-heap is not the only solution to this problem.