Assignment #4

Due: Friday Feb 18, 2022 (11:59 pm)

(Total 40 marks)

CMPUT 204
Department of Computing Science
University of Alberta

A number of questions in this assignment require you to study some problems and their solutions given in Problem Set #3. If a problem is related to some exercises of CLRS, possible solutions may also be found online. You may consult, adopt, or revise any of these solutions for your answers. When asked to show how an algorithm works, make sure you indicate clearly which algorithm you are applying. If you apply an algorithm different from what is given in Problem Set #3, please specify.

Problem 1. (16 marks)

a. (2 marks) Consider the following program:

```
Smallest (A, f, l)
  **Precondition: A[f..l] is an array of integers, f ≤ l and f, l ∈ ℕ.
  **Postcondition: Returns the smallest element of A[f..l].
  if f = l then
    return A[f]
  else
    m = ⌊(f+l)/2⌋
    return min(Smallest(A, f, m), Smallest(A, m + 1, l))
  end if
```

Write a recurrence relation that describes the time complexity of this program in terms of the size of array A and solve it.

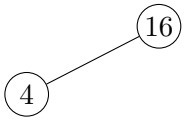
Solution Let's define $n = l - f + 1$. Then $T(n)$ is constant for sufficiently small n and $T(n) = T(\lfloor \frac{n}{2} \rfloor) + T(\lceil \frac{n}{2} \rceil) + c$ where c is a constant (for the time spent for computing m and taking the min and doing the recursive calls). We can use the master theorem (assuming n is a power of 2) with $a = 2$, $b = 2$ and $f(n) = n^0$, i.e. $c = 0$. This implies that $T(n) \in \Theta(n)$.

b. (2 marks) Show the result after each operation below, starting on an NIL BST: `Insert(16)`, `Insert(4)`, `Insert(3)`, `Insert(9)`, `Insert(1)`, `Insert(44)`, `Insert(29)`, `Delete(1)`, `Delete(4)`, `Insert(34)`.

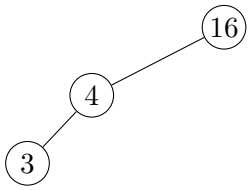
Solution

1.

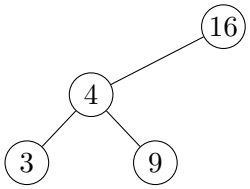
2.



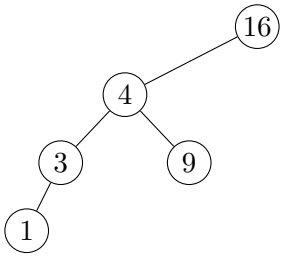
3.



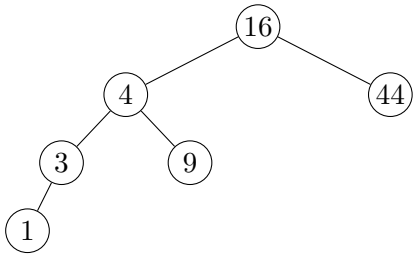
4.



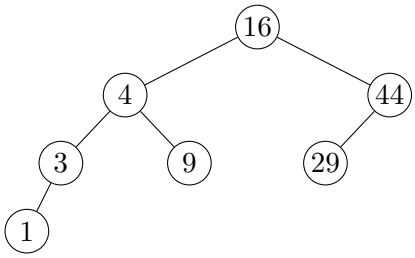
5.



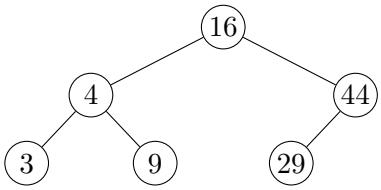
6.



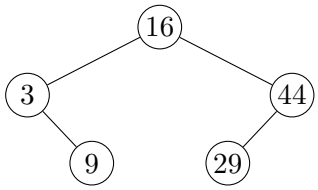
7.



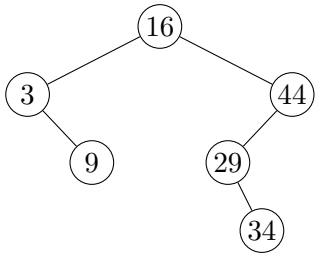
8.



9.

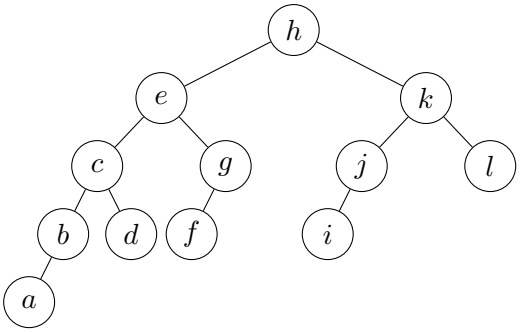


10.

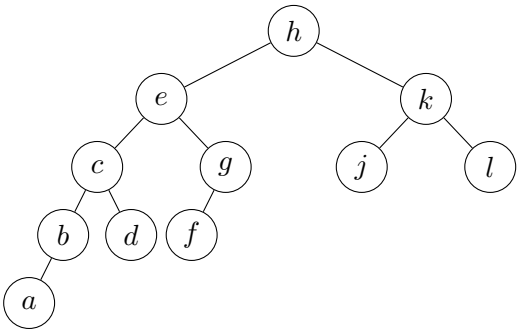


c. (2 marks) First, draw an AVL tree of height 4 that contains the minimum number of nodes. Then, delete a leaf node (any leaf node) from your AVL tree above so that the resulting tree violates the AVL-property. Then, apply tree-rotation to turn it to an AVL tree.

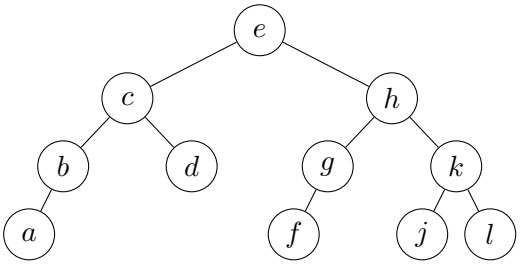
Solution



Deleting node i from the tree results in the following tree



Perform a right rotation around the root node h to fix it.



d. (2 marks) Demonstrate what happens when the following keys are inserted to a hash table with collisions resolved by chaining: 18, 22, 32, 8, 6, 10, 20, 16, 2, 29. Let the table have 9 slots and the hash function be $h(k) = k \mod 9$.

Solution

h(k)	keys
0	18
1	10
2	$29 \rightarrow 2 \rightarrow 20$
3	
4	22
5	32
6	6
7	16
8	8

Table 1: Caption

e. (2 marks) CLRS Exercise 6.5-9 (which can also be found in Problem Set #3) asks you to give an $O(n \lg k)$-time algorithm to merge k sorted lists into one sorted list, where n is the total number of elements in all the input lists. Use the following example to explain how your algorithm works for three sorted lists: L1 = [2,5], L2 = [4, 8], and L3 = [3, 7].

Solution We get the smallest element of each Li and construct them into a min-heap. For [2, 4, 3], we get a min-heap A = [2, 4, 3].

- Remove the smallest element, 2, and add it to the output X = [2]. The new array is A = [3, 4].
- Add element 5 from L1 to A. A = [3, 4, 5].
- Remove the smallest element 3, and add it to the output X = [2, 3]. The new array is A = [4, 5].
- Add 7 from L3. A = [4, 5, 7].

- Remove 4, add to X. X = [2, 3, 4]. A = [5, 7].
- Add 8 from L2. A = [5, 7, 8].
- Remove 5, add to X. X = [2, 3, 4, 5].
- There are no more elements in L1, L2, L3. And A = [7, 8].
- Remove 7, and add to X. X = [2, 3, 4, 5, 7].
- Remove 8, and add to X. X = [2, 3, 4, 5, 7, 8].

Output X is sorted. Extracting the minimum element and adding to the heap both take $O(\log k)$ time. Therefore, finding the whole sorted list takes $O(n \log k)$.

f. (3 marks) Consider Problem 7 of Problem Set #3

- Suppose the root of a tree T with 7 nodes is nearly balanced (note that the subtrees rooted at other nodes of the tree may not be nearly balanced). What is the maximum height of such a T?

  Solution: We know that if the root of a tree T with n nodes is nearly balanced, the maximum height of T is $\frac{2(n-1)}{3}$. Here, $n = 7$. So the max height is $\frac{2(7-1)}{3} = 4$.

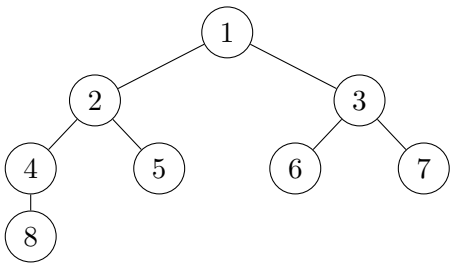
- To calculate the maximum possible height of a nearly balanced tree with n nodes, we start from

$$H_{max}(n) = 1 + H_{max}(\tfrac{2}{3}(n-1))$$

  Use $n = 8$ as an example to explain how this equation helps derive the maximum height of a nearly balanced tree. To support your answer, draw a nearly balanced tree that consists of 8 nodes. (Hint: The above equation can be viewed as a recursive definition, with some base cases added.)

  Solution:

$$\begin{aligned} H_{max}(8) &= 1 + H_{max}(4) \\ &= 1 + 1 + H_{max}(2) \\ &= 1 + 1 + 1 + H_{max}(0) \\ &= 3 \end{aligned}$$



- If a BST is nearly balanced, what would be the running time for searching an element in it? Briefly justify your answer.

Solution: Searching in a BST takes $O(h)$ time, where h is the height of the tree. Since $h \in O(\log n)$ for a nearly balanced tree, searching takes $O(\log n)$ in a nearly balanced BST.

g. (3 marks) Consider Problem 9 of Problem Set #3. Explain, using n = 10, how your solution works. In particular, explain how you identify which bottle is poisoned.

Solution We begin by first ordering the $n = 10$ bottles arbitrarily in a line, and labeling them $b_1, \ldots, b_n$. Have the king select $k = \lfloor \log n \rfloor + 1 = \lfloor \log 10 \rfloor + 1 = 3 + 1 = 4$ tasters and label them t_1, t_2, t_3, t_4. The index, i, of each bottle, b_i, can be represented in binary by at least 4 bits. Each taster's index corresponds to a bit position in the 4-bit representation of the bottles index. The tasters are made to test all bottles of wine whose corresponding bit 1. After a full month, we know that tasters who have died represent bit 1 and tasters who have survived represent bit 0. The ordering of these bits make up the binary representation of the poisoned bottle. Table 2 shows bottles and tasters for $n = 10$ and $k = 4$.

	t_1	t_2	t_3	t_4
b_1	0	0	0	1
b_2	0	0	1	0
b_3	0	0	1	1
b_4	0	1	0	0
b_5	0	1	0	1
b_6	0	1	1	0
b_7	0	1	1	1
b_8	1	0	0	0
b_9	1	0	0	1
b_{10}	1	0	1	0

Table 2: 10 bottles and 4 tasters

Problem 2. (8 marks)

A d-ary heap is like a binary heap except that non-leaf nodes have d children instead of 2 children (with one possible exception for the parent of the last leaf). Also see CLRS Problem 6.1, page 167. In this question, we consider 3-ary max-heaps.

Here, we give general solutions for any $d \geq 2$.

a. Given an array A and index i, how do you compute its left child, middle child, and right child, and how do you compute the parent of $A[i]$?

Solution: We can use an array to represent a d-ary heap. The root is $A[1]$ and for every node $A[i]$ its d children are in $A[d(i-1)+2], \ldots, A[di+1]$. The parent and jth child of node i can be computed by:

Parent (i)
  return $\lfloor \frac{i-2+d}{d} \rfloor$

Child (i, j)
  return $d(i-1) + j + 1$

b. What is the height of a 3-ary heap of n elements? Justify your answer briefly.

Solution: Since every node (other than leaves) in a d-ary tree has d children, the number of nodes in a d-ary tree of height h is at most $1+d+d^2+\ldots+d^h = \frac{d^{h+1}-1}{d-1}$ and is at least $1+d+d^2+\ldots+d^{h-1}+1 = \frac{d^h-1}{d-1}+1$. That is,

$$\frac{d^h - 1}{d-1} + 1 \leq n \leq \frac{d^{h+1} - 1}{d-1}$$

Working out the two inequalities, we get

$$\log_d[(n-1)(d-1)+1] \geq h \geq \log_d(1 + n(d-1)) - 1$$

Therefore, $h \in \Theta(\log_d(dn)) = \Theta(\log_d(d) + \log_d(n)) = \Theta(1 + \log_d(n)) = \Theta(\log_d(n))$.

c. Write a version of Max-Heapify for a 3-ary max-heap and analyze its running time.

Solution: The algorithm is very similar to Max-Heapify for the binary case, except that in this case we compare the key with each of its d children.

```
Max-Heapify (A, i)
  largest ⟵ i
  for j ⟵ 1 to d do
    j ⟵ Child(A, i, j)
    if j ≤ heapsize[A] and A[j] > A[largest] then
      largest ⟵ j
    end if
  end for
  if largest ≠ i then
    exchange A[i] and A[largest]
    Max-Heapify (A, largest)
  end if
```

The number of comparisons in each iteration is $\Theta(d)$ and the number of recursive calls is at most $O(\log_d n)$ (proportional to the height of the tree). Thus the running time of this version of Max-Heapify is $O(d \log_d n)$.

d. Write a version of Build-Heap for a 3-ary max-heap and analyze its running time.

Solution: Similar to the binary case, we can call Max-Heapify to turn A into a heap. The leaves are already a heap. The parent of the last element at index i is $\lfloor \frac{i-2+d}{d} \rfloor$.

```
Build-Max-Heap (A)
  heapsize[A] ⟵ length[A]
  for i ⟵ parent(heapsize(A)) downto 1 do
    Max-Heapify (A, i)
  end for
```

The proof of correctness of Build-Max-Heap in CLRS for binary heaps can be easily modified to work for this case (we skip the details). A naive analysis of running time yields $O(nd \lg_d n)$ as each Max-Heapify takes $O(d \log_d n)$. But here is a more careful analysis. The number of nodes at the root level (level 0) of a d-ary heap is $d^0 = 1$, at level 2 is d, at level 3 is d^2, and in general at level i which has height $h - i$ is d^i. By part (c) the time required for Max-Heapify at height i is $O(di)$. Thus the running time of Build-Max-Heap is:

$$\begin{aligned} T(n) &\leq \sum_{i=0}^{i=h} d^{h-i} O(di) \\ &\leq O(d^{h+1}) \sum_{i=0}^{i=h} \frac{i}{d^i} \\ &\leq O(nd) \sum_{i=0}^{\infty} \frac{i}{d^i}. \end{aligned}$$

From A.8 in CLRS, the summation above is at most $\frac{1/d}{(1-1/d)^2} = \frac{d}{(d-1)^2}$. Thus $T(n) \in O(n \frac{d^2}{(d-1)^2})$ and since $d \geq 2$: $\frac{d^2}{(d-1)^2} \leq 2$. Therefore $T(n) \in O(n)$.

e. Let array $A = [2, 8, 5, 4, 7, 10, 12, 6]$. Invoke your Build-Heap(A) to build a 3-ary max-heap. Just show the resulting array.

Solution $A = [12, 10, 6, 4, 7, 2, 8, 5]$

Problem 3. (8 marks) Given a max-heap A containing n keys and two element x and y such that $x < y$. Propose an efficient algorithm for reporting (say printing) all the keys z in A satisfying $x \leq z \leq y$ (note that x and y may not necessarily be in A). The running time of your algorithm should be $O(k)$ where k is the number of elements that are greater than or equal to x.

Give your algorithm in pseudocode and analyse its running time. In particular, comment on why your algorithm is more efficient than a linear search over A: check each element a in A to see whether it is the case that $x \leq a \leq y$.

Solution: The algorithm will be a recursive traversal of the tree (heap). At each node since we know that the largest value of that subtree is at the root, by comparing the root with x and y we can determine whether we must explore that subtree or not. If the root of that subtree is smaller than x then all the elements of that subtree are smaller than x. Otherwise, we will check each of the subtrees rooted at the children of the current node. We only print when the current node is smaller than y. If it's bigger, then we keep searching without printing.

The running time is $O(k)$ since we can call the function at most k times before we hit the base condition. The worst case happens when y is the largest element. The algorithm is more efficient than a linear search over A since linear search would take $O(n)$ time where n is the number of elements in A.

Here is the psuedocode (we call it with $i = 1$):

```
Report-All (A, i, x, y)
  ** returns all z (x ≤ z ≤ y) in the heap rooted at A[i].
  if A[i] < x then
    return
  end if
  if A[i] < y then
    Print A[i]
  end if
  if 2i ≤ heapsize(A) then
    Report-All (A, 2i, x, y)
  end if
  if 2i + 1 ≤ heapsize(A) then
    Report-All (A, 2i + 1, x, y)
  end if
```

Problem 4. (8 marks) We have a set J of n jars and a set L of n lids such that each lid in L has a unique matching jar in J. Unfortunately all the jars look the same (and all the lids look the same). We can only try fitting a lid ℓ to a jar j for any choice of $\ell \in L$ and $j \in J$; the outcome of such an experiment is either a) the lid is smaller, b) the lid is larger, or c) the lid fits the jar. We cannot compare two lids or two jars directly. Describe an algorithm to match all the lids and jars. The expected number of tries (experiments) of fitting a lid to a jar performed by your algorithm must be $O(n \log n)$.

We describe an algorithm and then you will be asked to answer some questions.

The idea of the algorithm is very similar to that of Quicksort.

1. If $n = 1$ (i.e. only one jar and one lid) then they must be matching; return this as the solution.
2. If $n > 1$ we do the following:
   (a) Take anyone of the lids, say $L[\ell]$ as the pivot (for example take one of them randomly)
   (b) Compare this lid against all the jars (using n comparison) to partition the jars into three sets: 1) J_s are all those jars that are smaller than $L[\ell]$, 2) J_l are all those jars that are larger than $L[\ell]$, 3) the 3rd set contains the unique jar, say j, that fits into $L[\ell]$.
   (c) Use jar j (that is matching lid ℓ) and compare against all the lids in L (except ℓ) to partition L into a set of smallers, denoted by L_s, and those that are largers, denoted by L_l; this takes $n-1$ comparisons.
   (d) Note J_s are all the jars smaller than ℓ and L_s are all the lids smaller than j, and therefore the lids of jars in J_s are exactly those in L_s; similarly the lids of jars in J_l are exactly those in J_l. Solve the following two subproblems recursively: jars and lids in J_s and L_s as one subproblem, and jars and lids in J_l and L_l as another subproblem.

Here are the questions that you are asked to answer.

- Provide a running time analysis.

  Solution So using $2n-1$ comparisons in steps 2(b) and 2(c) above, we break the problem of size n into two subproblems of size n_s (where $n_s = |L_s| = |J_s|$) and n_l (where $n_l = |L_l| = |J_l|$), where $n_s + n_l = n - 1$. Therefore, the recurrence relation for this is $T(n) = T(n_s) + T(n - n_s - 1) + O(n)$. This is the same recurrence as for the quicksort. Therefore, using the same analysis as in quicksort, if each pivot lid is selected randomly, then the expected number of comparisons of this algorithm is $O(n \log n)$.

- Suppose it is required that your algorithm always select the last lid as the pivot in the given line up. Consider a problem with 7 lids $l_1, \ldots, l_7$ and 7 jars $j_1, \ldots, j_7$, in the increasing order of their sizes, where all jars have distinct sizes and l_i uniquely fits j_i for all $1 \le i \le 7$. Given the sequence of lids, $\langle l_5, l_1, l_6, l_7, l_2, l_3, l_4 \rangle$, and an arbitrary sequence of jars, explain how your algorithm works. In your answer you can assume that any subproblem of the given problem is solved correctly. That is, you don't need to show how subproblems are solved, just give the result.

  Solution In the first recursive step, the pivot is l_4. The state of the algorithm J_s, J_l, L_s, L_l is given in Table 3. The subproblems are then recursively solved by choosing l_3 and l_7 as pivots.

J_s	$\langle j_1, j_2, j_3 \rangle$	L_s	$\langle l_1, l_2, l_3 \rangle$
J_l	$\langle j_5, j_6, j_7 \rangle$	L_l	$\langle l_5, l_6, l_7 \rangle$

Table 3: State of the algorithm (jars and lids) after the first recursive step

- Suppose your algorithm must take the last lid as the pivot in the given line up. What would be the worst-case running time of your algorithm and when it is guaranteed to happen? What is the best-case running time? You do not need to present your algorithm, but it must be an efficient one under the requirement.

  Solution Since the algorithm is similar to quicksort, the worst-case running time is $O(n^2)$, which occurs if we choose the smallest or the largest lid as pivot in each step.
  The best-case running time is $O(n \log n)$.