MIDTERM EXAM 2 (SAMPLE)

Note: The actual Midterm 2 will contain exactly 5 questions. In this sample exam the number of questions may differ. (*You must justify all your answers to recieve full credit. The questions that are written on paper should be photographed and uploaded as JPEG or PDF. The C++ programming question should be submitted as the C++ source file in a separate folder.*)

Question 1:

(*1.A. Given a list/stack/queue algorithm pseudocode, find its time complexity.*)

Consider the following List ADT (Abstract Data Type):

void L.add(E item)	// add *item* to the end of the list *L*.
void L.add(int pos, E item)	// add *item* at position *pos*; shift other items forward.
boolean L.contains(E item)	// return `true` iff the item is in the list *L*
int L.size()	// return the number of items in the list *L*
boolean L.isEmpty()	// return `true` iff the list is empty
E L.get(int pos)	// return the item at position pos in the List
E L.remove(int pos)	// remove and return the item at position pos in the List

Consider the following pseudocode – it erases all the odd elements and reverses the remaining list:

```
function ModifyList(L: List[int]):
pos := 0
while pos < L.size():
    if L.get(pos) % 2 == 1:
        L.remove(pos)
    else:
        pos := pos+1
n := L.size()
for i from $n - 1$ to 0 by -1:
    item := L.get(i)
    L.add(item)
repeat n times:
    L.remove(0)
```

What is the worst-case time complexity of the three loops (**while**, **for** and **repeat**) in the function *ModifyList(L)*. Express the complexity in terms of n (the size of the original list L)? Consider two cases:

(A) Assume that the List ADT is implemented as a doubly linked list with the pointers to the first and the last element and an additional number `size` telling the current number of items – the number of nodes in this linked list.

(B) Assume that the List ADT is implemented as an array (assume that the array has enough space for any possible input) and an additional number `size` telling the current number of items (namely, only the array elements from 0 to `size - 1` contain actual values, everything else in the array should be ignored).

Answer:

(A)

while takes $O(n^2)$, **for** takes $O(n^2)$, **repeat** takes $O(n)$ steps.

Indeed, if no item in the list is odd, then `pos` is incremented every time in the **while** loop. And then the calls to `L.get(pos)` become slower every time. (Unfortunately, the ADT methods `L.get()` cannot use iterator (or pointer to the linked list nodes – the implementation of list is encapsulated and hidden from the user). Using linked list explicitly, it can be traversed in $O(n)$ – in linear time (and it is possible to remove all the odd elements as well).

for loop also takes $O(n^2)$ as you need to travel :math`n-1`, then $n - 2$ steps, and so on. (Once again, you cannot benefit from the links in the linked list as the ADT methods do not exposse any internal pointers).

repeat loop (removing the first element n times) takes $O(n)$ steps. In this case no need to traverse the linked list.

(B)

while takes $O(n^2)$, **for** takes $O(n)$, **repeat** takes $O(n^2)$ steps.

Once again **while** loop can take quadratic time, but this time the worst case happens, if there are many odd elements that need to be removed (and all the array elements to the right need to be shifted to the left). For example, if every second element is odd, we would still get $O(n^2)$ steps.

for loop takes $O(n)$ as it is possible to access every item at position `pos` directly in a single step. It is also easy to add it to the end of the list (and increment $size$).

repeat loop takes $O(n^2)$ as one hs to shift the elements to the left many times.

Note: The algorithmic task (remove odd elements and reverse the list) can be done in $O(n)$ steps – all three loops can become efficient, if we keep using using the doubly linked list implementation.

In this case one needs additional ADT methods (for example, an iterator allowing to traverse the given list in both directions). Such operations are efficient in a linked list, if we do not require that the cursor (for reading or deletion) moves to `pos` from the beginning of the list every time.

□

Question 2:

(*2.A. Given some tree properties and element counts, calculate or estimate other counts.*)

Consider a tree T with the following properties:

- T has exactly 12 internal nodes,

- Every internal node of T has one or two children.
- The average number of children for an internal node is exactly 1.5.
- Tree T is built from `TNode` structures:

```
struct TNode { int info; TNode* left; TNode* right; };
```

(A) What is the number of leaves in tree T?

(B) What is the number of `NULL` values among all the `TNode.left` and `TNode.right` pointers used to build the tree T?

(C) What is the largest and the smallest value of the height of the tree T. (Write your estimates for minimum height and maximum height, and explain why these estimates cannot be improved.)

Note: We define *height* of a tree as the number of edges to traverse on the path that connects its root with the deepmost leaf. (In particular, a tree with just the root node has height 0, but a tree with the root and a single child leaf has height 1.)

Answer: (A) 7, (B) 20, (C) 4 (min) and 12 (max).

(A) Out of 12 internal nodes there must be exactly 6 nodes with two children and 6 nodes with one child each (to achieve the average 1.5). A tree without two-child nodes would have just one "branch" that is terminated by a leaf. Each internal node having two children increases the number of downgoing "branches" by one, so it increases the number of leaves by one as well. Consequently, six two-child nodes cause seven leaves.

(B) The total number of nodes (both internal and leaves) is $12 + 7 = 19$. Consider another tree which is a full binary tree – it adds all the `NULL` pointers as new leaves; in this case all the former nodes (internal and leaves) become internal nodes.

One can prove by induction that the number of leaves exceeds the number of internal nodes exactly by 1 (Drozdek 2013, p.217). So there must be exactly $19 + 1 = 20$ leaves in the new tree (corresponding to the `NULL` pointers).

(C) The smallest height can be achieved locating all the six nodes with two children in the upper three levels (its height is 4). The largest heigth can be achieved by arranging the internal nodes of a binary tree one under another (its height is 12).

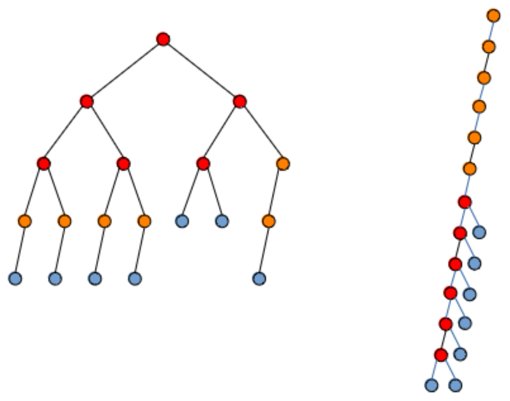


Fig. 1: Trees with the minimum height (4) and the maximum height (12).

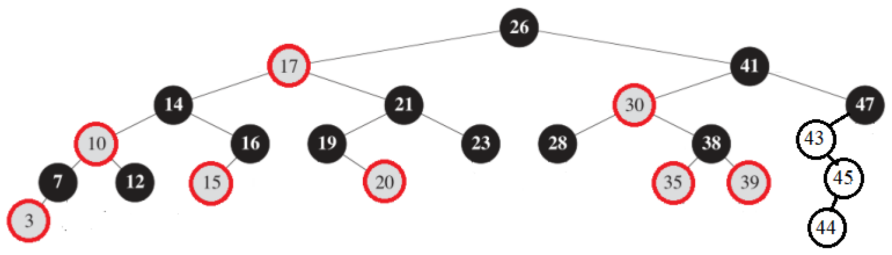
It is impossible to improve these estimates: Any tree with height three cannot have more nodes than the perfect tree of height three – it has just 7 internal nodes (not 12 as in our case). Also, no tree with 12 internal nodes can be taller than 12 (at most one level is added with every internal node).

Note: Both trees of "extremal" height (as well as all the other binary tree satisfying the conditions of this problem) have the same number of leaves and the same number of `NULL` pointers – they are *invariants* of any binary tree with the given parameters.

□

Question 3:

(*3.A. Perform insert and delete operations in an arbitrary binary search tree.*)



(A) Draw the Binary Search tree obtained when the tree shown in figure has its node 47 deleted.

(B) Draw the Binary Search tree obtained when the tree from (A) has its node 26 deleted.

note To avoid ambiguity, try to replace the node to be deleted with its inorder successor whenever it is possible. (You do not need to balance the resulting tree as an AVL or as a red-black tree – as long as it stays a binary search tree and preserves the tree ordering property.)

Answer:

(A) Since 47 is the very last node (in the given ordering), this is precisely the case when it does not have an inorder successor (so it has to be replaced by its inorder predecessor, namely the node 45. (In this case the node 44 becomes the right child of 43).

(B) The root 26 should be replaced by its inorder successor – the node 28.

Pictures are not shown, but you are encouraged to draw them.

□

Question 4:

(*4.D. Use and analyze Heapsort.*)

An array of 10 elements is used to initialize a minimum heap (as the first stage of the Heap sort algorithm):

$$\{5, 3, 7, 10, 1, 2, 9, 8, 6, 4\}$$

Assume that the minimum heap is initialized in the most efficient way (inserting elements level by level – starting from the bottom levels). All slots are filled in with the elements of the 10-element array in the order they arrive.

(A) How many levels will the heap tree have? (The root of the heap is considered L_0 – level zero. the last level is denoted by L_{k-1}. Just find the number k for this array.)

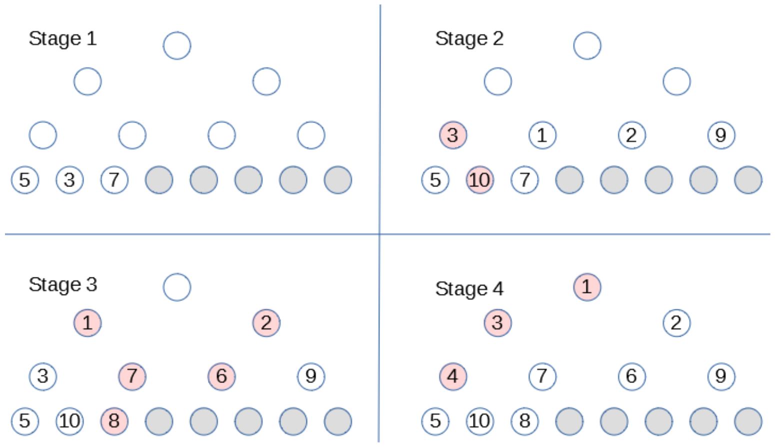
(B) Draw the intermediate states of the heap after each level is filled in. Represent the heap as a binary tree. (If some level L_k is only partially filled and contains less than 2^k nodes, please draw all the nodes as little circles, but leave the unused nodes empty.)

(C) What is the total count of comparisons ($a < b$) that is necessary to build the final minimum heap? (In this part you can assume the worst case time complexity – it is not necessarily achieved for the array given above.)

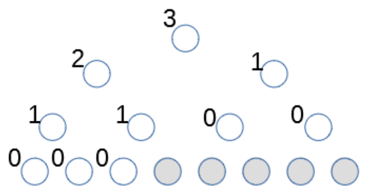
Answer:

(A) 10 elements need a tree with four levels (complete tree with 10 nodes). The last level L_3 will have just three nodes filled in.

(B) See the picture with all four stages of adding elements (unused slots are gray; the nodes that swap their places during the downheap operations are shown in pink).



(C) In a downheap operation (when you add a new node on top of two other nodes), you first need to compare the two siblings, then compare their parent with the smallest of the two siblings (and if it is larger than its child, then swap). So every time some node moves one level down, you need to spend at most two comparisons.

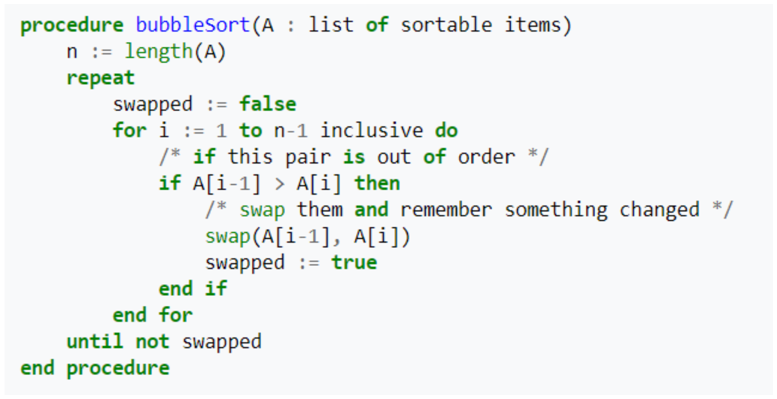


For our complete tree (with five grayed out slots in the last level), the worst case happens, if every node inserted at height h needs to spend $2h$ comparisons to travel to the very bottom (if we assume the worst case – that it is larger than everything that has been inserted so far). So the total number of comparisons is $2 \cdot (1 + 1 + 1 + 2 + 3) = 16$. In general, this time should grow as $O(n)$, where n is the number of items in the heap being built.

□

Question 5:

(*5.A. Use and analyze Selection sort, Insertion sort, Bubble sort algorithms.*)



```
procedure bubbleSort(A : list of sortable items)
    n := length(A)
    repeat
        swapped := false
        for i := 1 to n-1 inclusive do
            /* if this pair is out of order */
            if A[i-1] > A[i] then
                /* swap them and remember something changed */
                swap(A[i-1], A[i])
                swapped := true
            end if
        end for
    until not swapped
end procedure
```

The image shows Bubble sort pseudocode for a 0-based array $A[0]\dots A[n-1]$ of n elements.

(A) How many comparisons (`A[i-1] > A[i]`) in this algorithm are used to sort the given array. Show the state of the array after each `for` loop in the pseudocode is finished.

$$A[0] = 9,\ 0,\ 1,\ 2,\ 3,\ 4,\ 5,\ 6,\ 7,\ A[9] = 8.$$

(B) How many comparisons (`A[i-1] > A[i]`) in this algorithm are used to sort the following array:

$$A[0] = 1,\ 2,\ 3,\ 4,\ 5,\ 6,\ 7,\ 8,\ 9,\ A[9] = 0.$$

Answer:

(A) 18 comparisons, 2 executions of the **for** loop:

After the first **for** loop the array is sorted:

$$A[0] = 0,\ 1,\ 2,\ 3,\ 4,\ 5,\ 6,\ 7,\ 8,\ A[9] = 9.$$

After the second **for** loop and 9 more comparisons no further swaps occur and the algorithm stops. The array is still the same:

$$A[0] = 0,\ 1,\ 2,\ 3,\ 4,\ 5,\ 6,\ 7,\ 8,\ A[9] = 9.$$

(B)

90 comparisons, 10 executions of the **for** loop:

After the first **for** loop:

$$A[0] = 1,\ 2,\ 3,\ 4,\ 5,\ 6,\ 7,\ 8,\ 0,\ A[9] = 9.$$

After the second **for** loop:

$$A[0] = 1,\ 2,\ 3,\ 4,\ 5,\ 6,\ 7,\ 0,\ 8,\ A[9] = 9.$$

After the ninth **for** loop:

$$A[0] = 0,\ 1,\ 2,\ 3,\ 4,\ 5,\ 6,\ 7,\ \ 8,\ A[9] = 9.$$

After the tenth **for** loop the array stays the same and the algorithm stops:

$$A[0] = 0,\ 1,\ 2,\ 3,\ 4,\ 5,\ 6,\ 7,\ \ 8,\ A[9] = 9.$$

□

Note: Small values near the end of the list will slow down the Bubble sort considerably. The authors of an accelerated Bubble-sort variant (Comb sort) call such values *turtles*. See https://bit.ly/3mmS6C4.

Question 6:

(*6.C. (C++ code) Use STL classes for priority queue operations.*)

Create a C++ program that reads from the standard input a positive integer n, and then exactly n space-separated integers. It should output the median of these integers to the standard output as a real number. Use `std::priority_queue` to find the median. (You can assume that the program will receive data that matches this description and you do not need to handle erroneous input.)

Note: Recall that the *median* of a collection of n numbers $a_1, a_2, \ldots, a_n$ is the middle number in the sorted order of a_i (if n is odd), or the arithmetic mean of the two middle numbers in the sorted order of a_i (if n is even).

Answer: The solution is given below. We build a maximum heap using default comparator (since it is median, the sorting direction does not matter; it could also be a minimum heap). Operation `pop()` removes the maximum element, but `top()` returns the maximum without modifying the queue.

```cpp
#include <iostream>
#include <queue>

using namespace std;
int main() {
    int n;
    cin >> n;
    // priority queue (by default it is maximum queue)
    priority_queue<int> queue;
    // use the standard input to populate the priority queue
    for (int i = 0; i < n; i++) {
        int a;
        cin >> a;
        queue.push(a);
    }
    // remove everything until the middle element
    for (int i = 0; i < (n-1)/2; i++) {
        cout << "Ignoring " << queue.top() << endl;
        queue.pop();
    }
    // convert the middle element into a real number
    double result = 1.0*queue.top();
    // if n is even, find average with the other middle element
    if (n % 2 == 0) {
        queue.pop();
        result = (result + queue.top())/2;
    }
    cout << result << endl;
}
```

□