DELFT UNIVERSITY OF TECHNOLOGY
Faculty of Electrical Engineering, Mathematics and Computer Science

Examiner responsible: Joana Gonçalves and Robbert Krebbers
Examination reviewer: Stefan Hugtenburg

TI1316/TI1316TW Algorithms and Data Structures (Midterm Exam)

12 March 2018, 13.30 - 15.30

Parts of the examination:
[This paper] 22 multiple-choice questions (number of correct answers per question:1).
[This paper] 2 open questions (consisting of multiple parts).
[Weblab] 2 implementation questions on Weblab (consisting of multiple parts).

Determination of the grade:
35% multiple-choice questions (all questions are equally weighted).
35% open questions (see questions for weights).
30% implementation questions (see Weblab for weights).

Use of information sources and aids:
- Any version of the book Algorithms and Data Structures in Java by Tamassia, Goorich and Goldwasser is permitted.
- The use of of notes is not permitted.
- Scrap paper sheets are provided in the beginning of the exam. Additional scrap paper can be requested, please ask the surveillants.
- The use of any devices other than the assigned computer is not permitted.

Additional instructions:
- Solve the exam on your own. Any form of collaboration is fraud.
- You may not leave the examination room during the first 30 minutes.
- You will not be allowed to start the exam if you arrive after the first 30 minutes.
- If you are eligible for extra time, make sure to show your "Verklaring Tentamentijd Verlenging" to the surveillants.

Instructions on paper (analysis) part:
- Write your name and student number on every sheet of paper.
- Write the total number of sheets of paper you hand in.
- For multiple-choice questions, note that the order of the choices next to the boxes on the multiple-choice answer form might **not always be A-B-C-D**!
- Tip: mark answers to multiple-choice questions on this exam paper first. Copy them to the multiple-choice answer form after revising.
- For open questions, provide all requested information. Avoid irrelevant information, this could lead to point deductions.
- For proofs, make sure your proof is properly structured and sufficiently explained. Statements or steps without justification could lead to point deductions.

Instructions on Weblab (implementation) part:
- In case the Weblab server is too busy, you might not be able to compile your solution immediately. Please continue working on your solution and save it, so you can compile and run it later. You can also open another tab and work on other questions while waiting.
- Do not close the browser after you register for the exam. If you do so accidentally, ask for the help of the surveillants.
- Use the comment functionality of Weblab to ask for clarification about phrasing of the exam.

Multiple-choice questions (35%, 22 points)

Abstract data types

The following Java code shows a partial implementation of a data structure `DS`, which stores data elements of type `E` with keys of type `K`. Answer questions 1 and 2 below using this code.

```
public interface P<K,E> {
   public Entry<K,E> getElement() throws IllegalStateException;
}

public class DS<K,E> implements ADS<K,E> {
  /* ... */

  public Entry<K,E> method1() {
    if (data.isEmpty()) return null;
    return data.first().getElement();
  }

  public void method2(K k, E e) {
    P<K,E> p = data.last();
    while (p != null && compare(k, p.getElement().getKey()) < 0)
      p = data.before(p);
    if (p == null)
      data.addFirst(new Entry<>(k, e));
    else
      data.addAfter(p, new Entry<>(k, e));
  }
}
```

1. (1 point) The methods `method1` and `method2` in the Java code above provide specific implementations of operations defined by an ADT `ADS` for the class `DS`. What type of ADT is `ADS`?

   A. Positional list
   B. Doubly linked list
   C. Priority queue
   D. Map

2. (1 point) The above implementation of class `DS` uses an internal data structure to store its entries. Which data structure is this?

   A. Positional list
   B. Doubly linked list
   C. Singly linked list
   D. Heap

Arrays and lists

3. (1 point) Consider a dynamic array. What is the tightest amortized time complexity of n push operations (insertions at the end) if the capacity C is doubled whenever the array is full (new capacity is $2C$)? Consider that the initial capacity C_0 is much smaller than n.

   A. $\mathcal{O}(\log_2 n)$
   B. $\mathcal{O}(n)$
   C. $\mathcal{O}(n \log_2 n)$
   D. $\mathcal{O}(n^2)$

4. (1 point) What is the tightest amortized time complexity of n push operations (insertions at the end) if the capacity C of the dynamic array is instead increased by 5 when the array is full (new capacity $C+5$)? Consider that the initial capacity C_0 is much smaller than n.

   A. $\mathcal{O}(\log_2 n)$
   B. $\mathcal{O}(n)$
   C. $\mathcal{O}(n \log_2 n)$
   D. $\mathcal{O}(n^2)$

5. (1 point) What is the tightest amortized time complexity of n push operations (insertions at the end) if the capacity C of the dynamic array is instead increased by $\lceil \frac{C}{4} \rceil$ whenever the array is full (new capacity $C + \lceil \frac{C}{4} \rceil$)? Consider that the initial capacity C_0 is much smaller than n.

   A. $\mathcal{O}(\log_2 n)$
   B. $O(n)$
   C. $\mathcal{O}(n \log_2 n)$
   D. $\mathcal{O}(n^2)$

6. (1 point) Consider the problem of finding the middle node in a list l of size n, given that n is odd. Count the number of accesses to positions of list l needed to find the middle node using the most time-efficient algorithm, and only $\mathcal{O}(1)$ space. If the same position is accessed several times, count each of those accesses separately. Let $\frac{x}{y}$ denote integer division. How many accesses to positions of list l does the most efficient algorithm perform to find the middle node by the most efficient algorithm when l is implemented by an array with a field that stores the size?

   A. 1
   B. $n+1$
   C. $\frac{n+2}{2}$
   D. $\frac{3n+2}{2}$

7. (1 point) Consider the problem described in question 6. How many accesses to positions of list l does the most efficient algorithm perform to find the middle node when l is implemented by a singly-linked list without a size field?

   A. 1
   B. $n+1$
   C. $\frac{n+2}{2}$
   D. $\frac{3n+2}{2}$

8. (1 point) Consider problem described in question 6. How many accesses to positions of list l does the most efficient algorithm perform to find the middle node when l is implemented by a doubly-linked list without a size field?

   A. 1
   B. $n+1$
   C. $\frac{n+2}{2}$
   D. $\frac{3n+2}{2}$

9. (1 point) Consider the following state of a circularly-linked list `cll`:

   $$\texttt{cll} = \{\text{“Birch”}, \text{“Cherry”}, \text{“Oak”}, \text{“Pine”}\}$$

   where the tail reference points to the node containing element “Cherry”. What is the state of `cll` after the operations `removeFirst` and `rotate` are performed, in this order?

   A. { “Cherry”, “Oak”, “Pine” }, with the implicit head pointing to “Pine”.
   B. { “Birch”, “Cherry”, “Pine” }, with the implicit head pointing to “Pine”.
   C. { “Birch”, “Cherry”, “Pine” }, with the implicit head pointing to “Birch”.
   D. { “Cherry”, “Oak”, “Pine” }, with the implicit head pointing to “Cherry”.

Stacks, queues and deques

10. (1 point) Consider a stack with n elements. What implementation is most space-efficient per element (for large n)?

    A. Circular array with capacity n.

    B. Dynamic array doubled successively when needed to accommodate for the n elements.

    C. Singly-linked list with n nodes.

    D. Circularly-linked list with n nodes.

11. (1 point) Consider the following sequence of operations, on an initially empty stack:

    $$\{\texttt{push}(5), \texttt{push}(3), \texttt{pop}(), \texttt{push}(2), \texttt{push}(8), \texttt{pop}(), \texttt{size}(), \texttt{push}(1), \texttt{pop}(), \texttt{pop}(), \texttt{pop}()\}.$$

    Which sequence of values is returned by this sequence of operations?

    A. { - , -, 3, - , -, 8, 3, -, 1, 2, 5}

    B. { - , -, 3, - , -, 8, 2, -, 1, 2, 5}

    C. { - , -, 3, - , -, 8, 2, -, 1, 3, 5}

    D. { - , -, 5, - , -, 3, 2, -, 2, 8, 1}

12. (1 point) Consider an implementation of a queue with n elements using a bounded, circular array with capacity C. Let f be the index indicating the front of the queue. What is the next available index for an `enqueue` operation?

    A. $(n+1)\%C$

    B. $(f+n-1)\%C$

    C. $(f+n)\%C$

    D. $(f+n+1)\%C$

Heaps and priority queues

13. (1 point) Consider an implementation of a priority queue with a sorted list. Which operation establishes the ordering of the keys?

    A. `insert`

    B. `removeMin`

    C. `min`

    D. A priority queue needs to be explicitly sorted using the `sort` method.

14. (1 point) Consider that a heap of height h has k entries in the last level. How many nodes does the heap have in total?

    A. $2^h + k$

    B. $2^h - 1 + k$

    C. $2^{h-1} + k$

    D. $2^{h-1} - 1 + k$

15. (1 point) Consider the bottom-up construction of an array-based heap. What happens after the first phase in which all entries are added to the array?

    A. Down-heap bubbling from the root node.

    B. Down-heap bubbling from all nodes except the ones in the last level, starting from the root node and ending at the last node in the last but one level.

    C. Down-heap bubbling from all nodes except the ones in the last level, starting from the last node in the last but one level and ending up at the root node.

    D. Up-heap bubbling from the root node.

Sorting

Consider the following Java code that implements a sorting algorithm. Answer questions 16, 17, and 18 based on this code.

```
public static void sort(int[] data) {
  sort(data, 0, data.length - 1);
}

public static void sort(int[] data, int low, int high) {
  int i = low, j = high;
  int y = data[low + (high-low)/2];

  while (i <= j) {
    while (data[i] < y) i++;
    while (data[j] > y) j--;
    if (i <= j) {
      swap(data, i, j);
      i++;
      j--;
    }
  }

  if (low < j)  sort(data, low, j);
  if (i < high) sort(data, i, high);
}
```

16. (1 point) Which sorting algorithm is implemented by method `sort` in the Java code above?

    A. Selection sort

    B. Merge sort

    C. Quick sort

    D. Heap sort

17. (1 point) Is the implementation of method `sort` in the Java code above in-place or not in-place?

    A. Not in-place

    B. In-place

18. (1 point) Let n be the size of the array `data` received as input by the method `sort` in the Java code above. What is the tightest worst-case **space** complexity of method `sort`?

    A. $\mathcal{O}(n)$

    B. $\mathcal{O}(\log_2 n)$

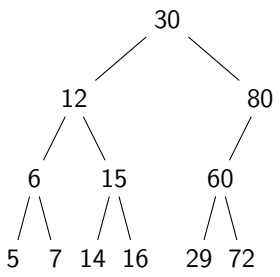
    C. $\mathcal{O}(1)$

    D. $\mathcal{O}(n^2)$

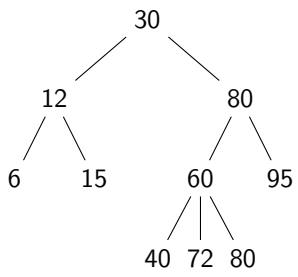
Trees

Consider the following trees **Tree 1** and **Tree 2**. Answer questions 19, 20, and 21 based on these trees.

Tree 1



Tree 2



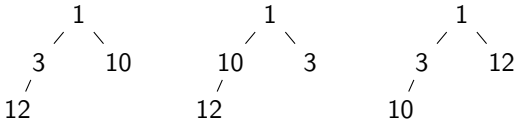
19. (1 point) Select the correct statement describing properties of **Tree 1**.
    - A. Tree 1 is both complete and a binary search tree.
    - B. Tree 1 is complete, but not a binary search tree.
    - C. Tree 1 is not complete, but it is a binary search tree.
    - **D. Tree 1 is neither complete nor a binary search tree.**

20. (1 point) Consider the result $\{5, 7, 6, 14, 16, 15, 12, 29, 72, 60, 80, 30\}$ of performing a traversal through **Tree 1**. Which kind of traversal was performed?
    - A. In-order traversal
    - B. Breadth-first traversal
    - C. Pre-order traversal
    - **D. Post-order traversal**

21. (1 point) Consider the result $\{30, 12, 80, 6, 15, 60, 95, 40, 72, 80\}$ of performing a traversal through **Tree 2**. Which kind of traversal was performed?
    - A. In-order traversal
    - **B. Breadth-first traversal**
    - C. Pre-order traversal
    - D. Post-order traversal

22. (1 point) How many different min-heaps exist with the set of keys $\{1, 3, 10, 12\}$?
    - A. 2
    - **B. 3**
    - C. 4
    - D. 5

Answer: The only choice at the first level is 1. At the second level, we can put either $3, 10$ or $10, 3$ or $3, 12$. Given the choice for the second level, there is a single choice for the third level.



Open questions (35%, 12 points)

23. Consider the following implementation of `methodX`. Let the time complexities of methods `insert` and `removeMin` be $\mathcal{O}(1)$ and $\mathcal{O}(k)$, respectively, where k denotes the number of entries in priority queue `P`. `ArrayList` methods used in this code have the following time complexities: `size` and `add` are $\mathcal{O}(1)$, `remove` as used in the code is $\mathcal{O}(1)$ since it always removes at the end. Assume that the priority queue `P` is empty when `methodX` is called.

```
public static <E> void methodX(ArrayList<E> S, PQ<E, Object> P) {          1
  int n = S.size();                                                        2
  for (int j = n-1; j >= 0; j--) {                                         3
    E element = S.remove(j);                                               4
    P.insert(element, null);                                               5
  }                                                                        6
  for (int j = 0; j < n; j++) {                                            7
    E element = P.removeMin().getKey();                                    8
    S.add(element);                                                        9
  }                                                                        10
}                                                                          11
```

(a) (3 points) Write down the polynomial expressing the **time** complexity of `methodX`. Define all variables and constants, and explain which lines of the code each term of the polynomial refers to.

Answer:

$$\begin{aligned} T(n) &= c_0 + c_1 n + c_2(n + (n-1) + \ldots + 2 + 1) \\ &= c_0 + c_1 n + c_2 \sum_{i=1}^{n} i \\ &= c_0 + c_1 n + c_2 \frac{n(n+1)}{2} \end{aligned}$$

where:

- n is the number of elements in the input list S;
- c_0 accounts for the primitive instructions associated with calling the method `methodX` (line 1), declaring and initializing variables `n` (line 2) and `j` (lines 3,7), and returning from the method `methodX` (line 11);
- c_1 accounts for operations within the first `for` loop, including conditional test and decrement on `j` (line 3), operation `remove` and assignment to variable `element` (line 4), and operation `insert` (line 5).
- c_2 accounts for operations within the second `for` loop, including conditional test and increment of `j` (line 7), operation `add` (line 9) and assignment of the result of `removeMin` to variable `element` (line 8).

(b) (1 point) Simplify your polynomial expression as much as possible. Derive the tightest worst-case time complexity in Big-Oh notation.

Answer:

$$\begin{aligned} T(n) &= c_0 + c_1 n + c_2 \frac{n(n+1)}{2} \\ &= c_0 + c_1 n + c_2 \frac{n^2 + n}{2} \\ &= c_0 + \left(c_1 + \frac{c_2}{2}\right) n + \frac{c_2}{2} n^2 \end{aligned}$$

The constants can be disregarded, since $\{c_0, c_1, c_2\} \ll n$. The term n^2 grows faster than any other term in the polynomial when $n \to \infty$, therefore the time complexity of method `methodX` in Big-Oh notation is $\mathcal{O}(n^2)$.

(c) (1½ points) Describe a more efficient solution for the problem addressed by the method `methodX`.

Answer: The algorithm `methodX` sorts an input sequence `S` with n elements using a priority queue `P` implemented by an unsorted list, with time complexity $\mathcal{O}(n^2)$. Both heap sort and merge sort would provide faster solutions, with worst-case time complexity $\mathcal{O}(n \log_2 n)$. Quicksort would also run faster on average, with an expected runtime of $\mathcal{O}(n \log_2 n)$, worst-case $\mathcal{O}(n^2)$.

24. Consider the following method `size`, which calculates the number of nodes in a tree:

```
public static int size(Node node) {                                    1
  if (node == null)                                                    2
    return 0;                                                          3
                                                                       4
  return 1 + size(node.getLeft()) + size(node.getRight());             5
}                                                                      6
```

(a) (2 points) State the base and recurrence equation for the **time** complexity of method `size` in terms of the height h of the tree rooted at node `node`. Refer to the relevant parts of the code to justify why your equations are correct.

Answer:

$$T(0) = c_0 \qquad T(h) = 2 \cdot T(h-1) + c_1$$

where:

c_0 accounts for the constant time operations in the base case, i.e. calling the method `size` (line 1), the conditional (line 2), and returning from method `size` (line 3).

c_1 accounts for the constant time operations in the recursive case, i.e. calling the method `size` (line 1), the conditional (line 2), the arithmetic operations (line 5), and returning from method `size` (line 5).

$2 \cdot T(h-1)$ accounts for the two recursive calls (line 5).

(b) (3 points) Derive the closed form of your recurrence equation. You should either prove the correctness of your solution by induction, or you should provide a detailed derivation of how your solution can be obtained by repeatedly unfolding the recurrence equation.

Answer:

Option 1. By repeated unfolding:

$$\begin{aligned}
T(h) &= 2 \cdot T(h-1) + c_1 && \text{(by unfolding } T(h)\text{)} \\
&= 2 \cdot (2 \cdot T(h-2) + c_1) + c_1 && \text{(by unfolding } T(h-1)\text{)} \\
&= 4 \cdot T(h-2) + 3c_1 && \text{(by arithmetic)} \\
&= 2^k \cdot T(h-k) + (2^k - 1)c_1 && \text{(by repeating } k \text{ times)} \\
&= 2^h \cdot T(0) + (2^h - 1)c_1 && \text{(by letting } k = h\text{)} \\
&= 2^h c_0 + (2^h - 1)c_1 \\
&= 2^h(c_0 + c_1) - c_1
\end{aligned}$$

Option 2. By induction:

Closed form solution. The closed form solution of the above recurrence is $T(h) = 2^h(c_0 + c_1) - c_1$. This can be obtained by repeatedly unfolding $T(h)$ using $T(h) = 2T(h-1) + c_1$ and replacing $T(0)$ using $T(0) = c_0$.

Induction proof.
Base case: For $h = 0$, prove $T(0) = c_0$.

Proof.

$$\begin{aligned} T(h) &= 2^0(c_0 + c_1) - c_1 \\ &= 1(c_0 + c_1) - c_1 \\ &= c_0 + c_1 - c_1 \\ &= c_0 \end{aligned}$$

□

Induction step: For $h > 0$, prove $T(h) = 2^h(c_0+c_1)-c_1$ assuming $T(h-1) = 2^{h-1}(c_0+c_1)-c_1$.

Proof.

$$\begin{aligned} T(h) &= 2 \cdot T(h-1) + c_1 && \text{(by } T(h) = 2T(h-1) + c_1\text{)} \\ &= 2 \cdot \left(2^{h-1}(c_0 + c_1) - c_1\right) + c_1 && \text{(by IH } T(h-1) = 2^{h-1}(c_0 + c_1) - c_1\text{)} \\ &= 2^{h-1+1}(c_0 + c_1) - 2c_1 + c_1 && \text{(by arithmetic)} \\ &= 2^h(c_0 + c_1) - c_1 \end{aligned}$$

□

(c) (½ point) State the tightest worst-case time complexity in terms of the height h of the tree in Big-Oh notation.

Answer: We previously established that the closed form is $T(h) = 2^h(c_0 + c_1) - c_1$. The constants in the closed form can be disregarded, therefore the time complexity of method `size` in Big-Oh notation is $\mathcal{O}(2^h)$.

(d) (1 point) State the base and recurrence equation for the **space** complexity of method `size` in terms of the height h of the tree. Refer to the relevant parts of the code to justify why your equations are correct.

Answer:

$$S(0) = c_0 \qquad S(h) = S(h-1) + c_1$$

where:

c_0 accounts for the stack frame associated with calling the method `size` in the base case.

c_1 accounts for the stack frame associated with calling the method `size` in the recursive case.

$S(h-1)$ accounts for the recursive calls (line 5). Note that unlike the case of time complexity, we have $S(h-1)$ instead of $2 \cdot S(h-1)$ since the space of the first recursive call can be reused by the second.

Implementation questions (30%)

There are two implementation questions on Weblab.

End of the exam