Delft University of Technology
EEMCS Faculty

CSE1305 Algorithms & Data Structures

Final Written Exam

29 January 2019, 9:00–11:00

Examiners:

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

Parts of the examination and determination of the grade:

Exam part	Number of questions	Question specifics	Grade
Multiple-choice	22 questions (equal weights)	One correct answer per question	50%
Open questions	3 questions (different weights)	Multiple parts	50%

- The grade for the multiple-choice questions is computed as $10 \cdot \frac{\text{score}}{22}$.
- The grade for the open questions is computed as $10 \cdot \frac{\text{score}}{30}$.

Use of information sources and aids:

- A hand-written double-sided A4 cheat sheet can be used during the exam.
- No other materials may be used, including but not limited to books, lecture slides in any form, or devices such as laptops and phones.
- Scrap paper sheets are provided at the beginning of the exam. Additional scrap paper can be requested.

General instructions:

- Solve the exam on your own. Any form of collaboration is prohibited.
- You may not leave the examination room during the first 30 minutes.
- If you are eligible for extra time, put the "Verklaring Tentamentijd Verlenging" on your table.

Instructions for writing down your answers:

- You should answer the questions on the provided answer sheets.
- Write your name and student number on every sheet of paper.
- **For multiple-choice questions**, the order of the choices on the answer form **might not be A-B-C-D!**
- Tip: mark multiple-choice answers on this exam paper first, copy them to the answer form after revising.
- **For open questions**, provide all requested information and always give an explanation. Avoid irrelevant data, it could lead to deductions.
- **For proofs**, make sure your proof is properly structured and sufficiently explained. Statements or steps without justification could lead to point deductions.

This page is intentionally left blank.

Multiple-choice questions (50%, 22 points)

1. (1 point) Consider the Java implementation of method `operation` below.

```
public int operation(int[] a, int[] b) {
  int s = 0;
  for (i = 0; i < a.length; i++) {
    s = s + a[i];
  }
  for (j = 0; j < b.length; j++) {
    s = s + b[j];
  }
  return s;
}
```

Let n and m be the lengths of arrays `a` and `b`, respectively. What are the tightest time and space complexities of method `operation`?

A. $\mathcal{O}(nm)$ time, $\mathcal{O}(1)$ space.
B. $\mathcal{O}(nm)$ time, $\mathcal{O}(n+m)$ space.
C. $\mathcal{O}(n+m)$ time, $\mathcal{O}(1)$ space.
D. $\mathcal{O}(n+m)$ time, $\mathcal{O}(n+m)$ space.

Answer: The first loop is $\mathcal{O}(n)$ and the second loop is $\mathcal{O}(m)$. Since we don't know which is larger, we say this is $\mathcal{O}(n+m)$ or $\mathcal{O}(\max(n,m))$. Since there is only constant additional space being used to store variable `s`, which does not depend on the input size, the space complexity is constant $\mathcal{O}(1)$.

2. (1 point) Consider the time complexity of `insert` in a min-heap with n items, implemented using a dynamic array that grows when full from capacity C to capacity $C + \frac{C}{3}$. Which statement is **correct**?

A. The amortized time complexity of one `insert` operation is $\mathcal{O}(1)$.
B. The time complexity of one `insert` operation is always $\mathcal{O}(\log n)$.
C. Performing n `insert` operations takes $\mathcal{O}(n)$ time.
D. Performing n `insert` operations takes $\mathcal{O}(n \log n)$ time.

Answer: Note that the dynamic array grows by a factor of 1.33, leading to a geometric progression.

A. The complexity of a `removeMin` operation in a heap without needing to resize is $\mathcal{O}(\log n)$, thus the upper bound of the complexity with resizing can never be lower than $\mathcal{O}(\log n)$.
B. Since some operations involve resizing the underlying array, the upper bound is not necessarily $\mathcal{O}(\log n)$ for every single operation. It is $\mathcal{O}(\log n)$ on average, or **amortized**.
C. Since the amortized time complexity per operation is $\mathcal{O}(\log n)$, the time complexity of n operations is then $\mathcal{O}(n \log n)$.
D. Correct.

3. (1 point) Consider a deque `d` containing elements $(1, 2, 3, 4, 5, 6)$, in this order, and an empty stack `s`. We first execute the following block of instructions **three times**:

```
s.push(d.last());
s.push(d.removeLast());
d.removeLast();
```

Then, we remove all elements from `s` using a series of `s.pop()` operations. What elements are returned and in what order?

A. (1,2,3,4,5,6)

B. (4,4,5,5,6,6)

C. (2,2,4,4,6,6)

D. (6,6,4,4,2,2)

Answer:

(0) first (1,2,3,4,5,6) last   top () bottom

(1) first (1,2,3,4,5,6) last   top (6) bottom
(2) first (1,2,3,4,5) last   top (6,6) bottom
(3) first (1,2,3,4) last   top (6,6) bottom
(4) first (1,2,3,4) last   top (4,6,6) bottom
(5) first (1,2,3) last   top (4,4,6,6) bottom
(6) first (1,2) last   top (4,4,6,6) bottom
(7) first (1,2) last   top (2,4,4,6,6) bottom
(8) first (1) last   top (2,2,4,4,6,6) bottom
(9) first () last   top (2,2,4,4,6,6) bottom

4. (1 point) Which of the following statements on sorting algorithms is **false**?

A. Insertion sort can run in linear time if the input sequence is nearly sorted.

B. Radix sort can be slower than $\mathcal{O}(n \log n)$ time sorting algorithms if the keys to be sorted are large (e.g. when the number d of elementary keys of each composite key is similar to n).

C. Merge sort can only be applied to sequences that fit into the main memory.

D. In the MSD radix sort variant that sorts from most to least significant keys, bucket sort needs to be applied recursively within each bucket defined in the previous iteration.

Answer:

A. When properly implemented, insertion sort has time complexity $\mathcal{O}(n + m)$, where n is number of elements in the sequence and m is the number of inversions. In a nearly sorted sequence, the number of inversions m is very small compared to the sequence size n.

B. Correct, radix sort is faster than $\mathcal{O}(n \log n)$ time algorithms if the radix (number of possible values for each elementary key N) and the number of elementary keys per composite key d are reasonably small, such that $d(n+N) << n \log n$. If $d \approx n$, then radix sort is $O(n^2)$.

C. Merge sort can be applied to very large data that do not fit into main memory.

D. Correct, otherwise the relative ordering of the keys across buckets can be broken.

5. (1 point) Which of the following recursive algorithms uses linear recursion?

A. Quick sort.

B. Merge sort.

C. Traversing a binary search tree.

D. Searching in a multi-way search tree.

Answer:

A. Quick sort makes two recursive calls: one for the sequence of elements smaller than the pivot, the other for the sequence of elements larger than the pivot.

B. Merge sort makes two recursive calls: one for each half of the input sequence.

C. Traversing a binary search tree makes at most two recursive calls (one per child), since it needs to visit all nodes in the tree.

D. Searching in a multi-way search tree makes at most one recursive call, for the one child whose subtree may contain the key that is being searched for.

6. (1 point) Consider the insertion sort algorithm. What is the state of sequence (7,1,3,6,2,5,4) after 3 complete executions of the algorithm's outer loop where the element being sorted is compared to at least one other element?

   A. (1,3,7,6,2,5,4)
   B. (1,2,3,6,7,5,4)
   C. (1,3,6,7,2,5,4)
   D. (1,2,3,7,6,5,4)

Answer: We start at index 1 (in insertion sort, the first element - at index 0 - is sorted by definition, since the order is established relative to previous elements and there are no previous elements when looking at the first element).

Iteration 1	(1,7,3,6,2,5,4)
Iteration 2	(1,3,7,6,2,5,4)
Iteration 3	(1,3,6,7,2,5,4)

7. (1 point) Consider the implementation of method `expertK` below.

```java
public static int expertK(int[] array, int k) {
  return expertK(array, 0, array.length-1, k);
}

private static int expertK(int[] array, int a, int b, int k) {
  if (a == b) return array[a];
  int left = a, right = b-1, choice = array[b];

  while (left <= right) {
    while (left <= right && array[left] < choice) left++;
    while (left <= right && array[right] >= choice) right--;
    if (left <= right) {
      swap(array, left, right);
      left++;
      right--;
    }
  }
  swap(array, left, b);

  if (k <= left-a) return expertK(array, a, left-1, k);
  else if (k <= left-a+1) return array[left];
  else return expertK(array, left+1, b, k-left+a-1);
}
```

Which algorithm is implemented by method `expertK` in the Java code above?

A. Selection sort.

B. Quick sort.

C. Median find.

D. Quick select.

8. (1 point) Consider the algorithm `expertK` from question 7. Which algorithmic design pattern does it follow?

   A. Decrease-and-conquer or prune-and-search.

   B. Divide-and-conquer.

   C. Brute force.

   D. Amortization.

9. (1 point) Consider an algorithm to move the second and the last but one elements of a sequence S with n elements to the middle of that sequence. Example: if S has elements $(1, 2, 3, 4, 5, 6, 7, 8)$, the algorithm should change S into $(1, 3, 4, 2, 7, 5, 6, 8)$. What is the complexity of the most time-efficient algorithm for this operation when S is implemented by an array or a singly-linked list?

   A. array: $\mathcal{O}(1)$ singly-linked list: $\mathcal{O}(1)$

   B. array: $\mathcal{O}(1)$ singly-linked list: $\mathcal{O}(n)$

   C. array: $\mathcal{O}(n)$ singly-linked list: $\mathcal{O}(1)$

   D. array: $\mathcal{O}(n)$ **singly-linked list:** $\mathcal{O}(n)$

Answer:

- Array: $\mathcal{O}(1)$ to access the second and last but one elements (by index), $\mathcal{O}(1)$ to find the middle (by arithmetic and index), $\mathcal{O}(n)$ to shift the elements between the second and the middle backward and the elements between the middle and the last but one forward in order to insert in the middle.
- SLL: $\mathcal{O}(1)$ to access the second element, $\mathcal{O}(n)$ to find the last but one element and calculate the size and middle index (if there's no size field), and $\mathcal{O}(n)$ to insert in the middle (since it requires traversal).

10. (1 point) What is the index of the left child of a node with index x in an array-based heap? Note: the first index of an array is 0 (zero).

    A. $2x - 1$

    B. $2x$

    C. $2x + 1$

    D. $2x + 2$

11. (1 point) Consider the `heapify` algorithm for building an array-based heap with n elements in $\mathcal{O}(n)$ time. What is the content of the array after a maximum-oriented heap is built using `heapify` for the input sequence (6,1,3,7,2,5,4)?

    A. (7,5,6,1,3,2,4)

    B. (7,6,5,1,3,2,4)

    C. (7,6,5,1,2,3,4)

    D. (7,6,3,1,2,5,4)

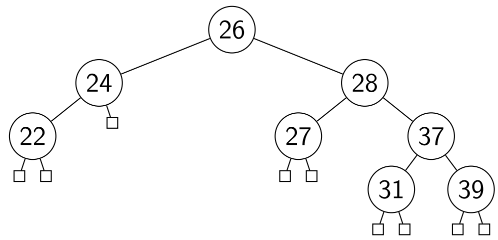
Answer: Input sequence: (6,1,3,7,2,5,4)

Heapify from 3: (6,1,5,7,2,3,4)
Heapify from 7: (6,7,5,1,2,3,4)
Heapify from 7: (7,6,5,1,2,3,4)

12. (1 point) Which of the following statements about hash functions and hash tables **is correct**?
    - A. A hash function guarantees that no two keys have the same hashcode.
    - B. When defining a Java class to be used for the keys of a hash map, it is necessary to override the `compress` method to obtain a hash map with good performance.
    - C. In case of a collision, the method of separate chaining uses another free hash bucket.
    - **D. The Horner's rule can be used to efficiently compute polynomial hash codes.**

13. (1 point) We would like to implement pre-order traversal of a binary tree **without recursion**. What data structure should we use?
    - A. Set.
    - **B. Stack.**
    - C. Queue.
    - D. Priority queue.

14. (1 point) Consider the following AVL tree:



How does the insertion of **21** affect the above AVL tree?
    - A. The insertion does not affect the AVL property.
    - **B. The insertion violates the AVL property, which needs to be fixed with a single rotation.**
    - C. The insertion violates the AVL property, which needs to be fixed with a double rotation.
    - D. The insertion violates the AVL property, which needs to be fixed with a triple rotation.

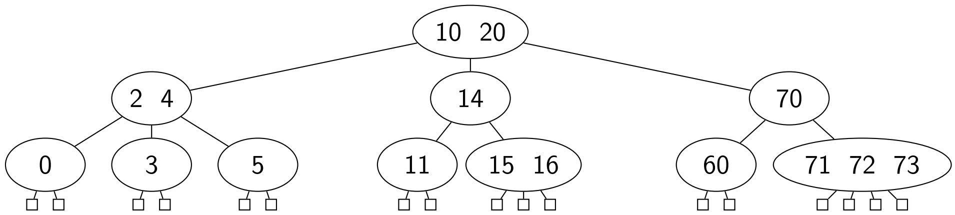
15. (1 point) What is the least number of nodes that can be stored in an AVL tree of depth 4?
    - A. 5
    - B. 6
    - **C. 7**
    - D. 8

Answer: The formula for computing the lower bound $o(h)$ on the number of nodes of an AVL tree of height h is as follows:

$$o(1) = 1 \qquad o(2) = 2 \qquad o(h) = 1 + o(h-1) + o(h-2) \text{ if } h > 2$$

For the given height 4, we have $o(4) = 7$.

16. (1 point) Consider the following (2,4) tree.



What is the number of red-black trees that correspond to the given (2,4) tree?

A. 2

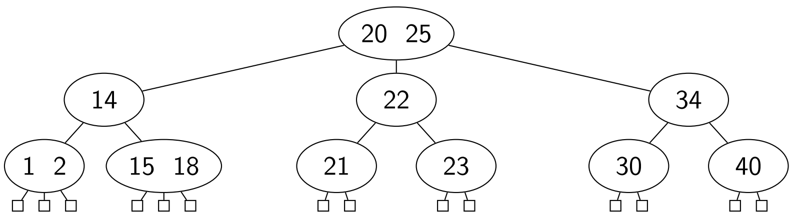
B. 3

C. 4

D. 8

Answer: Note that for each 3-node, there is a choice, but for 2 and 4-nodes there is no choice.

17. (1 point) Consider the following (2,4) tree:



When deleting **34** from the tree, how many underflows are caused in the tree?

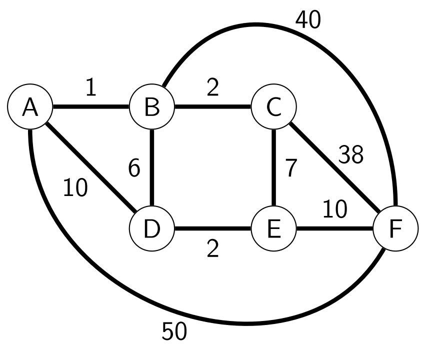
A. One underflow, which needs to be fixed by a fusion.

B. One underflow, which needs to be fixed by a transfer.

C. Two underflows, which need to be fixed by a fusion and transfer.

D. Two underflows, which need to be fixed by two fusions.

18. (1 point) Consider the following weighted graph:



During the execution of Dijkstra's algorithm starting in vertex A, how many different non-infinite values will the shortest path to vertex F take (i.e. labels of vertex F)?

A. 2

B. 3

C. 4

D. 5

19. (1 point) Consider below two different implementations of a method that finds and returns the edge of a graph with origin vertex u and target vertex v, when it exists. Note: to simplify the code, these methods use vertices and edges directly, not positions as seen in the book.

```
public Edge<E> getEdge1(Vertex<V> u, Vertex<V> v) {
  // note that variable 'edges' is a field
  for(Edge<E> e : edges) {
    if (e.getOrigin().equals(u) && e.getTarget().equals(v))
      return e;
  }
  return null;
}

public Edge<E> getEdge2(Vertex<V> u, Vertex<V> v) {
  List<Edge<E>> outEdges = u.getOutgoing();
  for(Edge<E> e : outEdges) {
    if (e.getTarget().equals(v))
      return e;
  }
  return null;
}
```

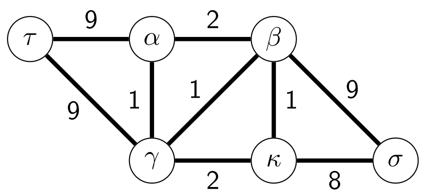
Each of these two implementations relies on a different data structure to represent the graph on which it operates. What are the two data structures?

A. getEdge1: edge list getEdge2: adjacency map
B. getEdge1**: edge list** getEdge2**: adjacency list**
C. getEdge1: adjacency list getEdge2: edge list
D. getEdge1: adjacency list getEdge2: adjacency map

20. (1 point) Consider again methods getEdge1 and getEdge2 from question 19. What are their tightest time complexities in Big-$\mathcal{O}$ notation if n and m are the numbers of vertices and edges in the graph and $deg(x)$ is the outdegree of vertex x, respectively?

A. getEdge1: $\mathcal{O}(n^2)$ getEdge2: $\mathcal{O}(n+m)$
B. getEdge1: $\mathcal{O}(n^2)$ getEdge2: $\mathcal{O}(deg(u))$
C. getEdge1: $\mathcal{O}(m)$ getEdge2: $\mathcal{O}(n+m)$
D. getEdge1**:** $\mathcal{O}(m)$ getEdge2**:** $\mathcal{O}(deg(u))$

21. (1 point) Consider the following weighted graph:



In general, a graph can have multiple minimum spanning trees. How many different minimum spanning trees does the above graph have?

A. 1
B. 2
C. 3
D. 4

22. (1 point) Given a vertex v in an undirected graph, what would be the most efficient algorithm to find a cycle starting and ending in v that contains the least number of edges?

    A. Kruskal's algorithm.

    B. Depth-first traversal.

    C. Breadth-first traversal.

    D. Dijkstra's algorithm.

Open questions (50%, 30 points)

23. Consider the following Java implementation of an iterative algorithm.

```
1 public static boolean methodX(int[] a) {
2   for (int i = 0; i < a.length; i++) {
3     for (int j = i+1; j < a.length; j++) {
4       if(a[i] == a[j])
5         return false;
6     }
7   }
8   return true;
9 }
```

(a) (5 points) Write the polynomial expressing the worst-case **time** complexity of method `methodX` as a function of n, where n is the number of elements in the array `a`. Define all variables and constants. Explain each term of the polynomial by referring to the lines of code.

Answer:

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

where:

- n is the number of elements in the input array `a`;
- c_0 accounts for the primitive instructions associated with calling the method `methodX` (line 1), and returning from the method `methodX` (lines 8); This also includes the initialisation of i.
- c_1 accounts for operations within the first `for` loop, including conditional test and increment of variable `i` (line 2); this also includes the initialisation of j.
- c_2 accounts for operations within the second `for` loop, including conditional test and increment of variable `j` (line 3), as well as the if statement (line 4) which always evaluates to false in the worst-case.

(b) (4 points) Simplify your polynomial expression as much as possible. State the tightest worst-case Big-$\mathcal{O}$ **time** complexity of `methodX` as a function of n. You **do not** have to give a proof, but you should clearly justify your answer.

Answer:

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

The constants can be disregarded, since $\{c_0, c_1, c_2/2\} \ll n$. 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) (3 points) Explain what the algorithm `methodX` calculates, i.e. for which arrays `a` does `methodX` return `true`. Describe a faster solution to perform the same calculation and state its tightest worst-case **time** complexity.

Answer: The algorithm `methodX` returns true iff all elements in the array `a` are unique (i.e. if `a` does not contain duplicate elements). A faster solution is to sort the array `a`, and then check if no consecutive elements in the sorted array are the same. The tightest worst-case time complexity of this solution is $O(n \log n)$.

24. Consider the following Java implementation of an algorithm on trees with integer values at nodes.

```
1 public static int methodY(Node node) {
2   if (node == null)
3     return 0;
4
5   return node.getElement() +
6     methodY(node.getLeft()) +
7     methodY(node.getRight());
8 }
```

(a) (4 points) State the base and recurrence equation for the **time** complexity of method `methodY` in terms of the **height** h of the tree rooted at `node`. Refer to the relevant parts of the code to justify your answer.

Answer:

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

where:

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

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

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

(b) (6 points) Derive the closed form of the given recurrence equation. You should either:

- derive the closed form solution by repeatedly unfolding the recurrence equation, or
- guess the closed form and prove correctness of your solution by induction.

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.

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

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.

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

(c) (2 points) State the tightest worst-case Big-$\mathcal{O}$ **time** complexity of `methodY` as a function of h. You **do not** have to give a proof, but you should clearly justify your answer.

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 `methodY` in Big-Oh notation is $\mathcal{O}(2^h)$.

(d) (2 points) State the tightest worst-case Big-$\mathcal{O}$ **space** complexity as a function of n, the **number of nodes** n in the tree rooted at `node`. You **do not** have to give a proof, but you should clearly justify your answer.

Answer: In the worst-case, when the tree is of the shape of a list, we have that the height h is $\mathcal{O}(n)$. Since the worst-case space complexity is proportional to the height, we obtain that the worst-case space complexity of method `methodY` in Big-Oh notation is $\mathcal{O}(n)$.

25. Prove that n^3 is $\Theta(2 + 4n + n^3)$.

(a) (2 points) State in detail the mathematical conditions that should be proved.

Answer: In order to prove that n^3 is $\Theta(2 + 4n + n^3)$, we have to prove that n^3 is both $\mathcal{O}(2 + 4n + n^3)$ and $\Omega(2 + 4n + n^3)$.
Therefore, we have to show that there exist positive constants c_1, c_2, and n_0, such that:

$$c_1 \cdot (2 + 4n + n^3) \leq n^3 \leq c_2 \cdot (2 + 4n + n^3) \quad \text{for all } n \geq n_0.$$

(b) (2 points) Prove that these conditions hold. Explain all the steps in your proof.

Answer: Let $c_1 = {}^1\!/_7$ and $c_2 = 1$ and $n_0 = 1$. When filling out these constants in the formula and performing some simplification, we obtain:

$$2/7 + 4/7n + 1/7n^3 \leq n^3 \leq 2 + 4n + n^3$$

This formula holds for any $n \geq n_0$ where $n_0 = 1$, as

$$\begin{aligned} 2/7 + 4/7n + 1/7n^3 &\leq 2/7n^3 + 4/7n^3 + 1/7n^3 \\ &= n^3 \\ &\leq 2 + 4n + n^3 \end{aligned}$$