Data structures and algorithms
Re-examination
Solution suggestions

Thursday, 2020-08-20, 8:30–12:30, in Canvas and Zoom

Examiner(s)	Peter Ljunglöf, Nick Smallbone, Pelle Evensen.
Allowed aids	Course literature, other books, the internet. ***You are not allowed to discuss the problems with anyone!***
Submitting	Submit your answers in ***one single*** PDF or Word or OpenOffice document. ***You can write your answers directly into the red boxes in this file.***
Pen & paper	Some questions are easier to answer using pen and paper, which is fine! In that case, take a photo of your answer and paste into your answer document. Make sure the photo is readable!
Assorted notes	You may answer in English or Swedish. Excessively complicated answers might be rejected. Write legibly – we need to be able to read and understand your answer!
Exam review	If you want to discuss the grading, please contact Peter via email.

There are 6 basic, and 3 advanced questions, and two points per question. So, the highest possible mark is 18 in total (12 from the basic and 6 from the advanced). Here is what you need for each grade:

- To **pass** the exam, you must get 8 out of 12 on the basic questions.
- To get a **four**, you must get 9 out of 12 on the basic questions, plus 2 out of 6 on the advanced.
- To get a **five**, you must get 10 out of 12 on the basic questions, plus 4 out of 6 on the advanced.
- To get **VG** (DIT181), you must get 9 out of 12 on the basic, plus 3 out of 6 on the advanced.

Grade	Total points	Basic points	Advanced
3 / G	≥ 8	≥ 8	—
4	≥ 11	≥ 9	≥ 2
5	≥ 14	≥ 10	≥ 4
VG	≥ 12	≥ 9	≥ 3

Question 1: Lagrange's four square theorem

In mathematics, *Lagrange's four-square theorem* states that every natural number n can be written as the sum of four squares. That is, $n = a^2 + b^2 + c^2 + d^2$ where a, b, c and d are integers ≥ 0. For example, $42 = 6^2 + 2^2 + 1^2 + 1^2$, and $89 = 7^2 + 6^2 + 2^2 + 0^2$.

https://en.wikipedia.org/wiki/Lagrange%27s_four-square_theorem

The following program takes as input a natural number n, and checks that Lagrange's four-square theorem holds for all numbers up to n. It first calculates which numbers can be written as the sum of *two* squares $a^2 + b^2$, and then checks that all numbers up to n can be written as the sum of two numbers of the form $a^2 + b^2$.

```
boolean checkLagrangeTheorem(int n):
    boolean[] isSumOfTwoSquares = new boolean[n]

    for i in 0..n-1:
        isSumOfTwoSquares[i] = false

    // which numbers can be written as the sum of two squares
    for i in 0..floor(sqrt(n)):
        // note: floor(x) rounds x down to the nearest integer
        // e.g. floor(3.7) = 3
        for j in 0..floor(sqrt(n)):
            if i*i + j*j < n:
                isSumOfTwoSquares[i*i + j*j] = true

    for i in 0..n-1:
        boolean ok = false
        // check if i can be written as j+k,
        // where j and k are the sum of two squares
        for j in 0..i:
            k = i-j
            if isSumOfTwoSquares[j] and isSumOfTwoSquares[k]:
                ok = true
        if not ok:
            // i cannot be written as a sum of four squares
            return false

    // all numbers in the range 0..n-1 can be written
    // as the sum of two squares
    return true
```

Part A

What is the worst-case time complexity of this program, as a function of n? Give your answer either using O-notation or using order of growth. You may assume that all arithmetic operations (including square root) take a constant amount of time.

The complexity is $O(n^2)$, i.e., quadratic.

There are three groups of loops. The first is of linear complexity because there is a single loop iterating over i = $0 \ldots n-1$ with a body of constant complexity. The second has two nested loops, but each one goes through $1 + floor(\sqrt{n}) = O(\sqrt{n})$ steps (and the inner loop body has constant complexity), so the nested complexity becomes $O(\sqrt{n}) \cdot O(\sqrt{n}) = O(n)$.

The third one is nested, where the outer loop iterates over $i = 0 \ldots n-1$, which is O(n) many iterations and the inner iterates over $0 \ldots i$, which is also $O(n)$ many iterations. The inner loop body is of constant complexity. So the total complexity is $O(n) \cdot O(n) = O(n^2)$.

Part B

When you run the program for n = 1,000,000, it takes one minute. If you plan to run the program for one day, roughly how big a value of n will you be able to check? Explain your answer, with reference to the complexity you calculated in part A.

One day is 1440 minutes, i.e., the program runs 1440 times longer than for when n = 1,000,000.

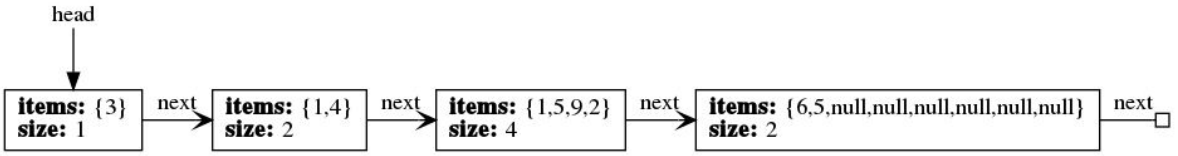
We want to find an m such that $m^2 = 1440n^2$. This gives us $m = \sqrt{1440n^2} = \sqrt{1440} \cdot n \approx 38n$.

So, we can check values of roughly 38,000,000.

Question 2: Linked dynamic arrays

A *linked dynamic array* is a data structure that stores a sequence of items, and supports getting and setting the *i*th item, and adding an item to the end of the sequence. Its design combines ideas from *linked lists* and *dynamic arrays*.

The data structure consists of a *linked list* of arrays. The last array in the list behaves specially: like a dynamic array, it can have empty space at the end. The other arrays in the list have no empty space. For example, here is a linked dynamic array containing the items 3, 1, 4, 1, 5, 9, 2, 6, 5:



Each node in the linked list has two data fields: *items*, an array of items, and *size*, which records how many items are stored in the array. In this example, the last array has six empty spaces since its *size* field is 2 but its array has a capacity of 8.

Notice that the arrays do not all have the same capacity. In a linked dynamic array, the first array always has a capacity of 1, and each array in the rest of the list is exactly twice as long as the previous array. In the example above, the array capacities go 1, 2, 4, 8.

Here is a partial implementation of linked dynamic arrays in a Java-like pseudocode. The class `LinkedDynamicArray` has just one field, `head`, which stores the first node in the linked list.

```
class LinkedDynamicArray<Elem>:
    // A class representing a linked list node
    class Node:
        Elem[] items // The array of items stored in this node
        int size     // The portion of the array that is used
        Node next    // The next node in the list
        // The constructor: returns a node containing an array of size 'capacity'.
        Node(int capacity):
            items = new Elem[capacity]
            next = null
            size = 0
    // The first node in the list
    Node hea
    // The constructor: returns a new empty linked dynamic array
    LinkedDynamicArray():
       head = new Node(1)
    // Return the number of items in the sequence
    int size():
        // To be implemented in part A of this question
    // Return the item stored at index i
    Elem get(int i):
        // To be implemented in question 7
    // Add an item to the end of the sequence
    void add(Elem item):
        // To be implemented in question 7
```

Part A

Starting from the code above, implement the method `size()` which returns the number of items in the sequence (e.g. for the example above of 3,1,4,1,5,9,2,6,5, it should return 9).

Write your answer either as pseudocode or code in a programming language of your choice. Pseudocode should be in **full detail**, like the pseudocode above. You may only use functionality that exists in the code above or that you implement yourself. In particular, you may not assume that an iterator has been defined for `Node`.

Just add up the 'size' field from each node:

```
int size():
    total = 0
    node = head
    while node != null:
        total = total + node.size
        node = node.next
    return total
```

Part B

What is the worst-case time complexity of `size` in terms of n, the number of items in the sequence?
Hint: Think about the relationship between n and the number of nodes in the list.

The complexity is logarithmic, $O(\log n)$.

The runtime is proportional to the number of nodes, but the number of nodes grows logarithmically in the number of items because the array size doubles with each node.

(Detailed explanation: Suppose there are k nodes. The first k–1 nodes are always full and so contain $1 + 2 + 4 + \ldots + 2^{k-2} = 2^{k-1} - 1$ items. In the worst case, the last node only has one item, and then the total number of items is therefore 2^{k-1}. Therefore $n \geq 2^{k-1}$, so $k \leq \log_2 n + 1$)

Question 3: An unknown sorting algorithm

Consider the following in-place sorting algorithm:

```
void sortInPlace(int[] A):
    for i in 1 ... A.length-1:
        int x = A[i]
        int j = i - 1
        while A[j] > x:
            A[j+1] = A[j]
            j = j - 1
            if j < 0:
                break
        A[j+1] = x
```

Part A

Which well-known sorting algorithm is this? Explain why.

Insertion sort.

Explanation: The outer loop inserts the elements A[1], A[2], … into the final sorted list, one at the time. The sorted list is always the sublist A[0]…A[i-1]. The inner loop inserts the element x=A[i] into the right position, by moving all larger elements one step to the right.

Part B

Get an individualised list of numbers from here: http://www.cse.chalmers.se/~evensen/exam20200820.html

Arrange the numbers you got in an order that yields the greatest number of comparisons when running the above algorithm. That is, you want the condition `A[j] > x` to be executed as many times as possible.

The list of numbers was 95, 43, 70, 50, 31, 11, 85, 58.

To get the worst complexity, we should give the numbers in reverse sorted order, i.e.:

95, 85, 70, **58, 50**, 43, 31, 11

Motivate why the order you have given is the worst possible.

Each new element to be inserted into the sorted sublist will always be the smallest, which means that it will always be inserted first in the list. This means that the inner loop has to move all already-sorted elements one step to the right. So, the inner loop will go all the way to the first element.

Question 4: Hash tables

Get an individualised list of elements and hash values from here:
http://www.cse.chalmers.se/~evensen/exam20200820.html

Insert the elements you got in the order given and with the hash value given after the colon, into an initially empty hash table.

Example: [B:1, C:2, A:0] means that the following elements should be inserted: B (with hash value 1), C (with hash value 2), and A (with hash value 0), in that order.

Part A

Insert the elements in a *separate chaining* hash table ("open hashing" / "linear chaining").

Let the size of the hash table array be $m = 5$.

Assume the following list of elements and hash values: B:13, C:11, A:7, D:12, F:13, E:10

Since m=5, every hash value h should be taken modulo 5 to get the index, $i = h\%5$. This gives us the following separate chaining hash table:

```
0 → E
1 → C
2 → A → D
3 → B → F
4
```

Part B

Insert the elements in an *open addressing* hash table ("closed hashing" / "linear probing"). Let the size of the hash table array be $m = 10$, and assume linear probing with **probe interval 7**, i.e.: $h' = (h + 7) \% m$

The hash value should be taken modulo 10, and when there's a collision the new index should have probe interval 7:

```
0 → F   (E collision)          5 →
1 → C                          6 →
2 → D                          7 → A   (E collision)
3 → B   (F collision)          8 →
4 → E                          9 →
```

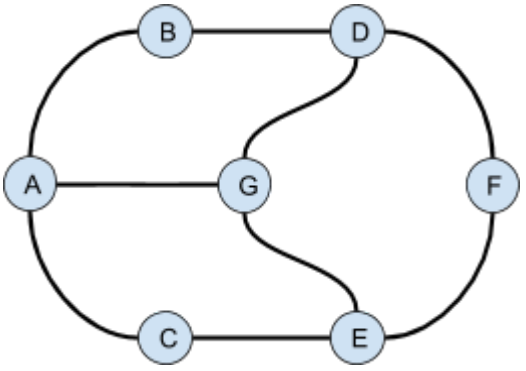
Explain what collisions you had and how each collision was resolved:

The first collision is F (h = 13%10 = 3). Then increase by the probe interval 7, giving new index 0.

Then E collides at index 0, and at the new index 7. So we continue probing and get the new index (7+7)%10 = 4, where there is a free spot.

Question 5: Make a graph directed

Assume the following undirected and unweighted graph:



Get a set of possible weights that you are allowed to use:
http://www.cse.chalmers.se/~evensen/exam20200820.html

Write down the set of possible weights that you got:

```
4, 5, 6, 9, 11, 12, 15, 16, 19, 20, 26, 27
```

Part A

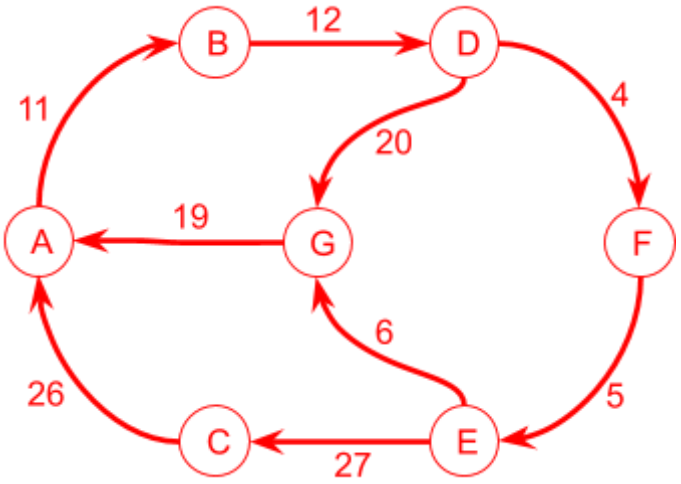
Add directions and assign weights from the possible weights (but no new edges) to the graph so that the following holds:

- all edges must be directed, and all weights must be unique (different from each other)
- there are an infinite number of possible paths from A to G
- the cheapest path from A to G (i.e., the path with the smallest total weight) is not the shortest path (i.e., the path passing through the fewest edges)
- if you remove the directions, the minimal spanning tree (MST) should have **total cost at least 50**

Important: all weights must come from the set of weights you got from the link above.
Each weight must only be used once, but you do not have to use all weights.

The graph needs a loop, to get an infinite number of possibilities. One possibility is A→B→D→F→E→G→A. If we now let D→G be more expensive than D→F→E→G, the cheapest path will be A→B→D→F→E→G. The cheapest path is always part of the MST, so we only have to add two expensive edges E→C and C→A (the arrows are not important here), and let the cost of G→A be high to exclude it from the MST.

The MST now consists of the edges C→A→B→D→F→E→G and has total cost 26+11+12+4+5+6 = 64.



Part B

Run Dijkstra's algorithm on your final graph, starting at node A. In which order are the nodes visited, and what is the calculated cost to reach them? (We say that a node is visited when it is removed from the priority queue). Also, show the contents of the priority queue, after the visited node has been processed.

Fill this table with the missing information:

Node	Cost	Priority queue (after the node has been processed)
A	0	B:11
B	11	D:23
D	23	F:27, G:43
F	27	E:32, G:43
E	32	G:38, C:59
G	38	C:59
C	59	

Question 6: AVL trees or red-black trees

Important: There are two questions n:o 6 – one about AVL trees and one about red-black trees. You can choose which one you want to answer.

Grading of this question: If you answer both questions, your points on question 6 will be the *maximum*, not the sum! This means that if you get 1 point on the AVL question and 1 point on the red-black question, you will get 1 point in total.

In short: Select one of the two to answer. Skip the other one.

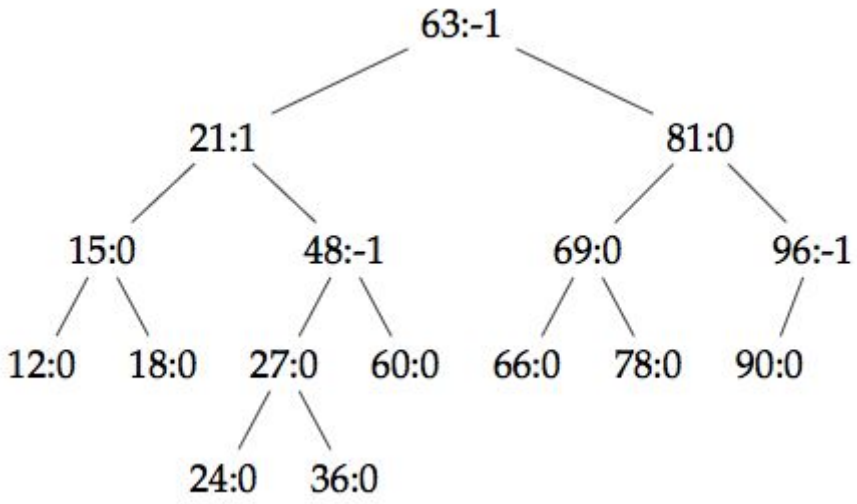
Question 6, alternative 1: AVL trees

Get an individualised binary search tree from here:
http://www.cse.chalmers.se/~evensen/exam20200820.html

Part A

Annotate the nodes in the tree with their AVL balance factor. The balance factor of a node is defined as the height of the right subtree minus the height of the left subtree.

Your tree may look different depending on the generated values, but here is one possibility:

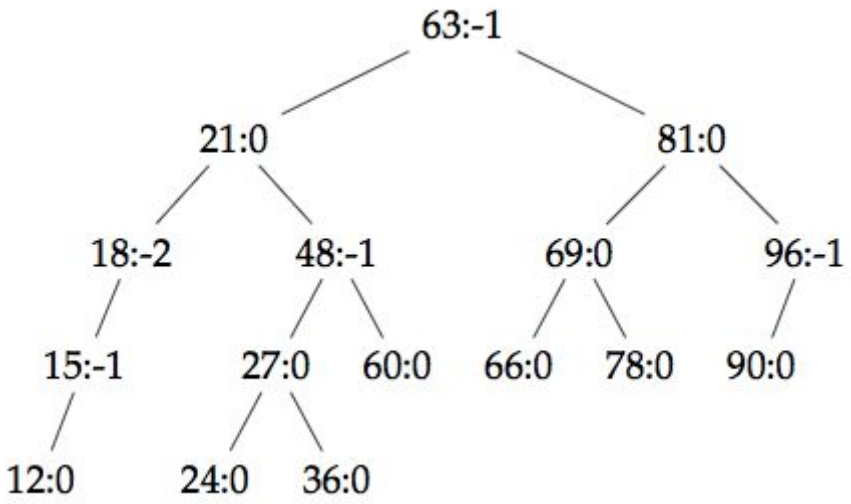


Part B

Take the AVL tree you created earlier and *transform* the tree using left and/or right *tree rotations* such that we get an unbalanced tree containing a so-called *left-left* case. You should show the resulting tree with updated balance factors, and explain which rotations you have used.

Insert your unbalanced tree here:

We can create a left-left case by rotating a node with two children to the left. For example, we have rotated the node 15 to the left:



Explain which rotations you used and in which order:

We rotated the node 15:0 to the left.

Question 6, alternative 2: Red-black trees

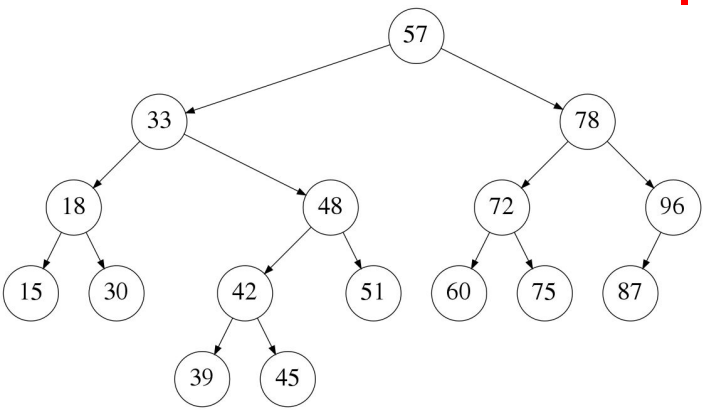
Get an individualised binary search tree from here:
http://www.cse.chalmers.se/~evensen/exam20200820.html

Part A

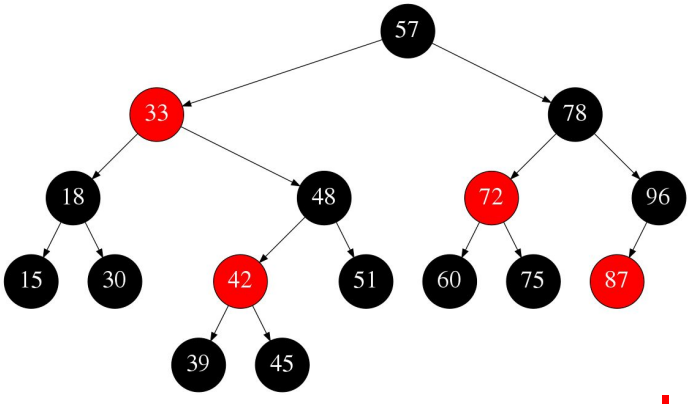
Colour the nodes in the tree red or black, so that the tree becomes a correctly balanced red-black tree, where no nodes should have two red children, and there is at least one red leaf node. *Do not transform the tree*, just colour the nodes! (Of course, this is not possible for all BSTs, but it is for the one you get)

Important: Don't forget the two additional restrictions: (1) no nodes should have two red children, and (2) at least one leaf node must be red. (This is to be able to do part B below.)

Suppose we start from the following BST:



The only possible way to colour it (given that a black node must have max 1 red child) is like so:

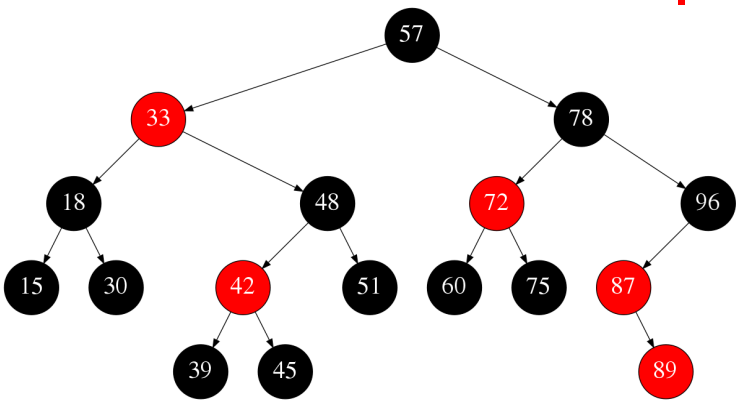


Part B

Insert a new red node into the tree, using the normal BST insertion algorithm, so that the final red-black tree is unbalanced, and the new red node is a **right child of another red node**.

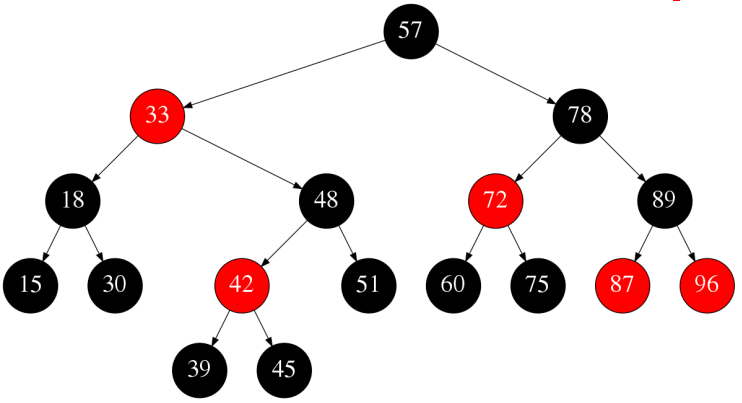
Important: The new red node must be a right child of another red node.

Let's insert the number 89:



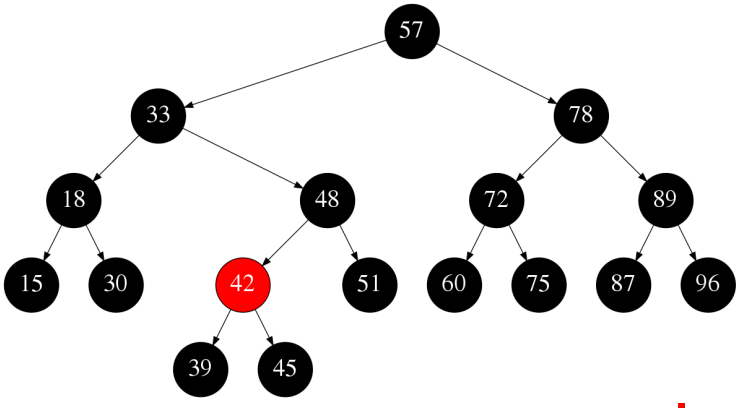
Then rebalance the tree, so that you have a correct red-black tree again.

Now we do some rotations to fix the 87–89 edge:



If you used the variant of red-black trees where a node can have two red children, you can stop here, because this is a correct red-black tree.
But see part C below.

Assuming that you used the variant of red-black trees from the lectures, where a node can only have one red child, there is one more step, doing colour flips to restore the invariant:

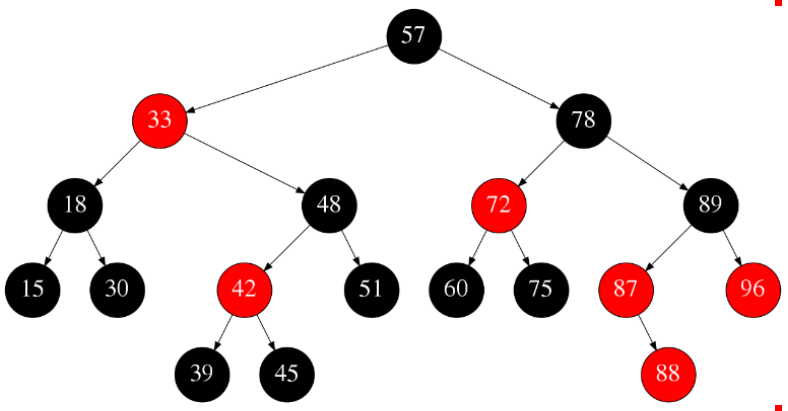


Part C

Look at the tree you got from part B. If the tree has no red leaf nodes, you should skip part C.

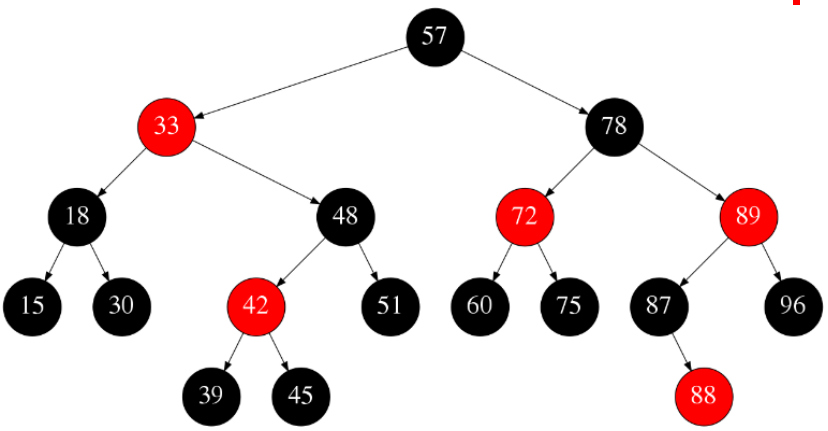
Otherwise, repeat the exact same process as in part B. Insert a new red node, using the normal BST insertion algorithm, so that the final red-black tree is unbalanced, and the new red node is a right child of another red node.

You only had to do this part if you used the variant of red-black trees where a node can have two red children (because your part b was simpler). Here we insert 88:



Then rebalance the tree, so that you have a correct red-black tree again.

Do a colour flip to repair the invariant:



Question 7 (advanced): Linked dynamic arrays, part II

This question builds on the linked dynamic array data structure defined in question 2. Please read that question before you attempt this one.

In this question you will implement two operations on linked dynamic arrays: getting the item at index *i*, and adding an item to the end of the sequence.

Write your answer either as pseudocode or code in a programming language of your choice. Pseudocode should be in **full detail**. You may only use functionality that exists in the `LinkedDynamicArray` class or that you implement yourself.

Part A

Implement a method `get()` which returns the *i*th item in the sequence, counting from 0.
For example, if `x` is the sequence 3,1,4,1,5,9,2,6,5 from question 2, then `x.get(2)` should return 4.

Idea: go through the list, keeping track of how many items we have skipped past.

```
Elem get(int i):
    node = head
    while i >= node.size:
        i = i - node.size
        node = node.next
    return node.items[i]
```

What is the worst-case complexity of `get()` in terms of *n*, the total number of items in the sequence?

The complexity is $O(\log n)$, similar to question 2b.

Part B

To add an item to the end of a linked dynamic array, we first check if there is free space in the last array of the list. If there is, we store the new item in that space. Otherwise, we add a new array of *twice* the capacity to the linked list, and store the item there.

Implement a method `add()` which adds a new item to the end of the sequence.

Your method should take (where *n* is the length of the sequence):

- *O(log n)* time when the last array of the list is not full
- *O(n)* time when the last array is full
  (assume that creating an array of size *n* takes *O(n)* time).

Recall that you can use `arr.length` to get the capacity of an array `arr` in Java.

```
void add(Elem item):
    // Search for the last node in the list
    node = head
    while node.next != null:
        node = node.next
    // Create a new node if the last node is full
    if node.items.length == node.size:
        node.next = new Node(node.size * 2)
        node = node.next
    // Add the item to the last node (which is not full now)
    node.items[node.size] = item
    node.size = node.size + 1
```

Explain why your method has this complexity.
(For the interested reader: this implies that the method takes amortised logarithmic time overall.)

When the last array is not full, it takes $O(\log n)$ time to find the last node (similar to before) and does a constant amount of extra work to add the item. When the last array is full, it also spends $O(n)$ time to create a new node so the total complexity is $O(n)$. (Note that when the last node is full, `node.size`×2 is $O(n)$, in fact it turns out to be exactly $n+1$.)

Part C

Is there a way to augment the class to get constant time complexity (amortised) for the `add()` method? You don't have to write any code, just explain what changes/additions you would do.

One way is to add an extra field to the LinkedDynamicArray class:

```
Node last
```

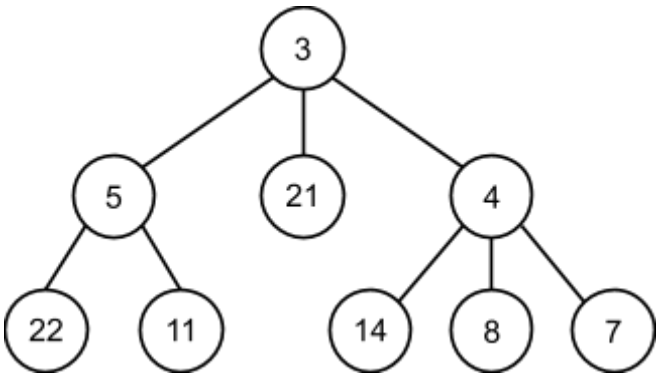
which stores the last node in the list. (We would have to change `add` so that it updates this field when creating a new node.) Then we can skip the search at the beginning of the add method.

Another way would be to use an array of nodes, instead of a linked list. (This is doable because we only ever need a logarithmic number of nodes, so e.g. an array of size 64 would be able to store 2^{64-1} items.) This would also have the advantage that we could make `get()` take constant time.

Question 8 (advanced): Rose heaps

A “rose tree” is a tree where the nodes can have any number of children. Since we don’t know how many children a given node has, we cannot use a fixed number of children pointers (such as “left” and “right” for binary trees). One way to implement the children of a node is as a linked list. (Another possibility is as an array, but that’s not what we’re going to do here).

This question is about defining a new kind of heap that is based on a rose tree. The idea behind this heap is to have multiple children instead of just two (as is the case in a binary heap). A rose heap must satisfy the heap property: the value of each node is less than (or equal to) the value of its children. The following is an example of a rose heap:



An implementation for such a heap might look like this (we use a linked list to implement the children):

```
class RoseHeap<Elem>:
    private class Node:
        Elem value
        LinkedList<Node> children
        Node(Elem val):  // The constructor for Node
            value = val
            children = new LinkedList<Node>()
    private Node root

    public Elem findMin():
        return root.value

    public void insert(Elem value):
        insertNode(new Node(value))

    public void insertNode(Node node):
        // To be implemented in part A
        if root == null:
            // What to do if the root is null?
        else if node.value < root.value:
            // What to do if the node is smaller than the root?
        else:
            // What to do if the root is smaller (or equal)?

    public Elem deleteMin():
        // To be implemented in part C
```

Part A

Complete the implementation of `insertNode()`:

```
void insertNode(Node node):
    if root == null:
        root = node
    else if node.value < root.value:
        node.children.add(root)
        root = node
    else:
        root.children.add(node)
```

Part B

Get an individualised list of numbers from here: http://www.cse.chalmers.se/~evensen/exam20200820.html

```
    69, 34, 51, 22, 52, 72, 63, 86
```

Build a new rose heap by hand, from the list of numbers you got.
Add all elements to an initially empty heap, in the same order as they are given.

If `add()` adds elements at the end of the list:

```
    22
    ↓
    34 → 52 → 72 → 63 → 86
    ↓
    69 → 51
```

If `add()` adds elements at the begining of the list:

```
    22
    ↓
    86 → 63 → 72 → 52 → 34
    ↓
    51 → 69
```

Part C

Complete the implementation of `deleteMin()`:

```
Elem deleteMin():
    Elem value = root.value
    LinkedList<Node> children = root.children
    root = null
    for each Node child in children:
        insertNode(child)
    return value
```

Part D

What is the worst-case complexity of your implementation of `deleteMin()`?

It's linear in the length of the children list.

How does a rose heap compare to other heap implementations, such a binary heap or a skew heap? Are there any advantages or disadvantages?

After adding the minimum element to a rose heap, all later additions will only increase the length of the children list. So, if the minimum element comes first, the heap will be in essence a long linked list. This does not affect insertion which is always constant, but deletion will in the worst case be linear. Which is bad compared to other heaps that have logarithmic deletion complexity.

Question 9 (advanced): Paths of a given length

The following class can be used to implement binary search trees:

```
class BST<Elem>:
    private class Node:
        Node left, right
        Elem value
    Node root
```

Your task is to write a method that returns *all paths from the root to any leaf, of a given length*.
A path is here a list of values, i.e., the values that you encounter when you move along the path.

```
class BST<Elem>:
    ...
    List<Elem[]> pathsOfLength(int len):
        // Returns all paths from the root to a leaf,
        // of the given length 'len',
        // where a path is an array of values
```

Your solution should only visit a node at most once. Also, you should not visit more nodes than necessary (i.e., don't visit nodes that cannot possibly lead to a solution).

```
List<Elem[]> pathsOfLength(int len):
    List<Elem[]> foundPaths = new List()
    if root == null:
        return paths
    List<Elem> path = new List()
    path.addLast(root)
    recursivePathsOfLength(len-1, root, path, foundPaths)
    return foundPaths

recursivePathsOfLength(int len, Node node, List<Elem> path, List<Elem[]> foundPaths):
    if len == 0 and node.left == node.right == null:
        foundPaths.add(path.toArray())
    else if len > 0:
        if node.left != null:
            path.addLast(node.left.value)
            recursivePathsOfLength(len-1, node.left, path, foundPaths)
            path.removeLast()
        if node.right != null:
            path.addLast(node.right.value)
            recursivePathsOfLength(len-1, node.right, path, foundPaths)
            path.removeLast()
```

Explain why your solution doesn't visit any node more than once:

The recursive function performs a depth-first search, visiting the left and the right child of a node at most once.

What unnecessary nodes are not visited by `pathsOfLength()`?

The nodes that are on paths that are longer than the given length.

You can assume that the class `List` has the following methods, and that all of them are efficient (i.e., constant time complexity, except `toArray()` which is linear).

```
class List<Elem>:
    List()  // The constructor

    Elem[] toArray()  // Teturns a copy of the list as an array
    boolean isEmpty()
    int size()

    void addFirst(Elem e)
    Elem getFirst()
    Elem deleteFirst()

    void addLast(Elem e)
    Elem getLast()
    Elem deleteLast()
```

(The alert reader notices that this is more like Java's `Deque` interface than the `List` interface, but that's not really important for the task at hand).