Part 1: Multiple Choice

Enter your answers on the Scantron sheet.
We **will not** mark answers that have been entered on this sheet.
Each multiple choice question is worth **2.5 marks**.

In all the questions, $\log(x)$ means $\log_2(x)$.

1. Consider the following pairs of functions $f(n), g(n)$. For which pair the functions are such that $f(n)$ is $O(g(n))$ and $g(n)$ is $O(f(n))$?

   (A) $f(n) = n^2, g(n) = n\log(n^2)$
   (B) $f(n) = \log n, g(n) = 10^5$
   $\surd$(C) $f(n) = n, g(n) = 2\log n + 10n$
   (D) $f(n) = n^3, g(n) = 2/n^3$
   (E) $f(n) = 4, g(n) = \log n$

2. The following values are inserted, in the given order, in a hash table of size 7 that uses linear probing and hash function $h(k) = k \bmod 7$:

   4, 11, 5, 12, 6.

   In which entry of the table is the key 6 stored?

   (A) 0
   $\surd$(B) 1
   (C) 4
   (D) 5
   (E) 6

3. An ordered dictionary containing 10^{12} data items of the form `(key,data)` needs to be stored in disk. The disk hardware allows data blocks of size b to be transferred to main memory every time that the disk is accessed. Which of the following data structures would allow the `find(key)` operation to be performed as efficiently as possible in the worst case?

   (A) A B-tree of order b.
   (B) A B-tree in which every node stores $10^{12}/b$ data items.
   (C) A B-tree in which every node stores b data items.
   $\surd$(D) A B-tree in which every node has size b.
   (E) An AVL tree.

4. Let T be an AVL tree. Suppose that after adding a new data item to T, the tree becomes unbalanced. Let node z be the root of the smallest unbalance subtree in T. Let y be the child of z with larger height, and let x be the child of y with larger height. After re-balancing the tree (and, regardless of whether an LL, LR, RR, or RL rotation was performed), which node will be the root of the subtree formerly rooted at z? Assume that all keys stored in T are different.

   (A) x
   (B) y
   (C) The node among x, y, z storing the largest key.
   $\surd$(D) The node among x, y, z storing the middle key.
   (E) The node among x, y, z storing the smallest key.

5. Consider the following algorithm:

Algorithm $T(r)$
Input: Root r of a proper binary tree.

if r is a leaf **then return** 0
else {
    $p \leftarrow T$(left child of r)
    $q \leftarrow T$(right child of r)
    return $p + q + 1$
}

What does the algorithm compute?

(A) The number of nodes in the tree.
$\surd$(B) The number of internal nodes in the tree.
(C) The number of leaves in the tree.
(D) The height of the tree.
(E) The number of descendants of r.

6. The smallest AVL-tree of height 1 has 1 key ("smallest" here means "smallest number of keys"), the smallest AVL-tree of height 2 has 2 keys, and the smallest AVL tree of height 3 has 4 keys. How many keys does the smallest AVL-tree of height 5 have?

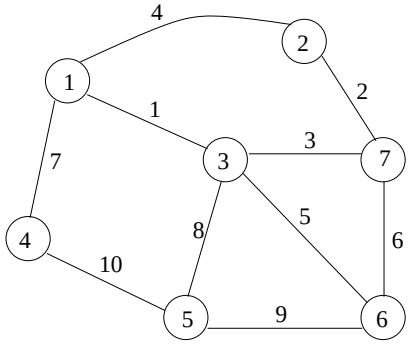
(A) 10
(B) 11
$\surd$(C) 12
(D) 13
(E) 14

7. How many different $(2, 4)$-trees containing the keys 1, 2, 3, 4, and 5 exist (each key must appears once in each one of these $(2, 4)$-trees)?

(A) 2
(B) 3
$\surd$(C) 4
(D) 5
(E) 6

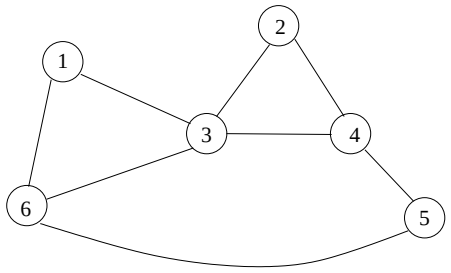
8. Consider the following graph. Which edges, and in which order, are selected by Prim's algorithm if it stars at vertex 1?

$\surd$(A) (1,3), (3,7), (2,7), (3,6), (1,4), (3,5)
(B) (1,3), (3,7), (2,7), (7,6), (1,4), (3,5)
(C) (1,3), (2,7), (3,7), (3,5), (7,6), (1,4)
(D) (1,3), (2,7), (3,7), (1,4), (3,6), (3,5)
(E) (1,3), (3,7), (2,7), (3,5), (7,6), (1,4)



9. Consider the following graph. Which one of the following is a valid ordering of the vertices if the graph is traversed using depth first search (DFS)?

(A) 1, 2, 6, 3, 5, 4
(B) 4, 3, 2, 1, 5, 6
(C) 2, 3, 1, 4, 6, 5
(D) 5, 4, 6, 2, 3, 1
√(E) 3, 6, 1, 5, 4, 2



10. Let $G = (V, E)$ be an undirected, connected graph with n vertices and m edges. All vertices are initially un-marked. Consider the following algorithm:

Algorithm traverse(G, u)
Input: Undirected, connected graph G, and vertex u of G.

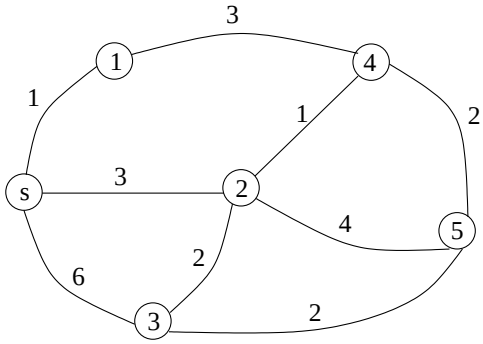
Mark u
$c \leftarrow 0$
For each edge (u, v) incident on u **do** {
    For each vertex w of G **do** $c \leftarrow c + 1$
    if v is not marked **then** $c \leftarrow c +$ traverse(G, v)
}
return c

Assume that G is stored in an adjacency list. What is the time complexity of the above algorithm in the worst case?

(A) $O(m)$
(B) $O(n + m)$
(C) $O(n^2)$
√(D) $O(nm)$
(E) $O(n^3)$

11. Consider the following graph. Assume that we use Dijkstra's algorithm to find shortest paths from vertex s to the other vertices in the graph. In which order are the final distance labels $u.d$ computed? (Or in other words, in which order are the shortest paths computed?)

√(A) s, 1, 2, 4, 3, 5
(B) s, 1, 2, 3, 4, 5
(C) s, 1, 2, 4, 5, 3
(D) s, 1, 4, 2, 3, 5
(E) s, 1, 4, 3, 2, 5



12. Let $G = (V, E)$ be an undirected graph. We are interested in selecting a data structure for representing G that allows us to implement the operations `areAdjacent(u,v)` and `incidentEdges(u)`. Let n be the number of vertices and m be the number of edges. Which of the following is true?
(A) An edge list (an array storing all edges of the graph) allows us to perform `incidentEdges(u)`

in $O(\text{degree}(\texttt{u}))$ time and `areAdjacent(u,v)` in $O(1)$ time.

(B) With an adjacency list `areAdjacent(u,v)` can be implemented in $O(1)$ time and `incidentEdges(u)` in $O(\text{degree}(\texttt{u}))$ time.

(C) An adjacency matrix allows us to implement both operations in $O(1)$ time.

(D) An adjacency list allows us to implement both operations in $O(1)$ time.

$\surd$(E) An adjacency matrix can implement `areAdjacent(u,v)` in $O(1)$ time and `incidentEdges(u,v)` in $O(n)$ time.

13. Consider the following algorithm.

Algorithm `Sort`(A, n)
Input: Array A containing n different integer values.
Out: Array A sorted in increasing order of value.

for $i \leftarrow n-1$ **downto** 1 **do** {
    $m \leftarrow i$
    (*)
    $t \leftarrow A[m]; A[m] \leftarrow A[i]; A[i] \leftarrow t$ // Swap $A[i]$ and $A[m]$
}
return A

Which of the following instructions must be inserted at the point marked (*) so that the algorithm correctly sorts the values stored in A in **increasing** order of value?

$\surd$(A) **for** $j \leftarrow 0$ **to** $i-1$ **do**
    if $A[j] > A[m]$ **then** $m \leftarrow j$

(B) **for** $j \leftarrow i+1$ **to** $n-1$ **do**
    if $A[j] > A[m]$ **then** $m \leftarrow j$

(C) **for** $j \leftarrow 0$ **to** $n-1$ **do**
    if $A[j] > A[m]$ **then** $m \leftarrow j$ }

(D) **for** $j \leftarrow 0$ **to** $i-1$ **do**
    if $A[j] < A[m]$ **then** $m \leftarrow j$ }

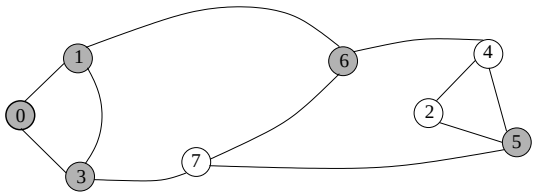
(E) **for** $j \leftarrow 0$ **to** $n-1$ **do**
    if $A[j] < A[m]$ **then** $m \leftarrow j$ }

Part 2: Written Answers

Write your answers **directly on these sheets**.

14. (12 marks) Let $G = (V, E)$ be an undirected graph in which every vertex is labelled either "free" or "toll". Write an algorithm that given two "free" vertices s and t it decides whether there is **at least one** path p from s to t that goes only through "free" vertices. If such a path exists, the algorithm must return the value `true`, otherwise it must return the value `false`.

    For example, for the following graph (the "free" vertices are shaded: 0, 1, 3, 5, 6) if $s = 0$ and $t = 6$ the algorithm must return the value `true` (as paths $\langle 0, 1, 6\rangle$ and $\langle 0, 3, 1, 6\rangle$ are as required); for $s = 1, t = 6$, the algorithm must also return `true` (as path $\langle 1, 6\rangle$ only goes through "free" vertices); however, for $s = 0, t = 5$, the algorithm must return `false` (as the only paths from vertex 0 to 5 must go through "toll" vertices).



    Algorithm free_path (s, t)
    In: "Free" vertices s and t
    Out: `True` if there is a path from s to t going only through "free" vertices; `false` otherwise.

```
Marx(s)
if s = t then return true;
else {
    For each edge (s, u) incident on s do {
        if u is labelled "free" and u is not marked then
            if free_path(u, t) then return true
    }
    return false
}
```

15. (4 marks) Compute the time complexity of your algorithm for the previous question in the worst case, assuming that the graph is stored in an adjacency list. **Explain** how you computed the time complexity and give the order of the time complexity.

    Let us first ignore the recursive calls. In each iteration of the for loop the algorithm performs a constant number c of operations. Since the graph is implemented using an adjacency list, the for loop is repeated degree(s) times, so the total number of operations performed by the loop is $c\times$degree(s). Outside the loop a constant number c_1 of operations are performed, so the total number of operations performed in one invocation of the algorithm is $c_1 + c\times$ degree(s).

    In the worst case all nodes of the graph are visited and all edges are processed, and the last edge processed is the one used to reach the destination vertex t. Therefore, the recursive calls make the algorithm call itself once per node, so the total number of operations performed by the algorithm is

$$\sum_{s\in G}(c_1 + c \times \text{degree}(s)) = c_1 n + 2cm \;\text{ is } O(m + n).$$

16. (12 marks) Consider two arrays A and B, each storing n different integer values. All values in A and B are different. In both arrays the values are sorted in increasing order. Write an algorithm that merges A and B and stores their values in an array C in such a way that the first n values in C are sorted in increasing order and the last n values are sorted in decreasing order. For example, if A and B are as follows:

A

3	6	8	13	19
0	1	2	3	4

B

1	4	9	11	17
0	1	2	3	4

Then array C must be this:

1	3	4	6	8	19	17	13	11	9
0	1	2	3	4	5	6	7	8	9

You cannot use any auxiliary data structures and you must write the complete algorithm (you cannot invoke algorithms for which you do not provide pseudocode).

Algorithm inc-dec(A, B, C, n)
$i_A \leftarrow 0$; $i_B \leftarrow 0$; $i_C \leftarrow 0$
// Sort first n values of C increasingly
while $i_C < n$ **do** {
    if $A[i_A] < B[i_B]$ **then** {
        $C[i_C] \leftarrow A[i_A]$; $++i_A$
    }
    else {
        $C[i_C] \leftarrow B[i_B]$; $++i_B$
    }
    $++i_C$
}
// Sort the last values in C decreasingly
$i_C \leftarrow 2n - 1$ // Start at the end of C
while $(i_A < n)$ **and** $(i_B < n)$ **do** {
    if $A[i_A] < B[i_B]$ **then** {
        $C[i_C] \leftarrow A[i_A]$; $++i_A$
    }
    else {
        $C[i_C] \leftarrow B[i_B]$; $++i_B$
    }
    $--i_C$
}
if $i_A < n$ **then**
    while $i_A < n$ **do** {
        $C[i_C] \leftarrow A[i_A]$; $++i_A$; $--i_C$
    }
else while $i_B < n$ **do** {
        $C[i_C] \leftarrow B[i_B]$; $++i_B$; $--i_C$
    }

17. (4 marks) Compute the time complexity of your algorithm for the previous question in the worst case. **Explain** how you computed the time complexity and give the order of the time complexity.

Every iteration of the first while loop performs a constant number c of operations and the loop is repeated n times, hence the total number of operations performed by this loop is cn. Note that the second while loop plus either the third or the fourth loops copy the last n values into array C, hence in total the second and third loop (or the second and fourth loop, depending on the result of the last if statement) perform n iterations. Each iteration of the second loop performs a constant number c_1 of operations, each iteration of the third loop also performs a constant number c_2 of operations and one iteration of the last loop performs a constant number c_3 of operations. Therefore, the second, third and fourth loops perform in total at most $n(c_1 + \max\{c_2, c_3\})$ operations.

Outside the loops there is an additional constant number c_4 of operations. Therefore, the total number of operations performed by the algorithm is $cn + n(c_1 + \max\{c_2, c_3\}) + c_4$ is $O(n)$.

18. (5.5 marks) Write a recurrence equation expressing the time complexity of the following algorithm. Explain your answer. Assume that n is a power of 2.

Algorithm `rec`(n)
Input: Integer value $n \geq 0$

```
if n = 0 then return 1
else {
    c ← 0
    For i ← 0 to n − 1 do c ← c + i
    c ← c + rec(n/2)
    return c
}
```

In the base case the algorithm performs a constant number c_0 of operations, hence

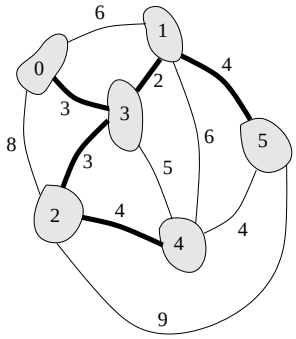
$f(0) = c_0$.

When $n > 0$, the **else** statement is executed. Note that each iteration of the **for** loop performs a constant number c_1 of operations and the loop is repeated n times, so the total number of operations performed by the loop is $c_1 n$. outside the loop, if we ignore recursive calls, the algorithm performs an additional constant number c_2 of operations. Since in the recursive call the value of the argument is $n/2$, then the equation for the time complexity of the algorithm when $n > 0$ is

$f(n) = c_1 n + c_2 + f(n/2)$

For the following 2 questions you can use any of the algorithms studied in class. You do not have to write full descriptions of the algorithms. Just indicate which algorithm you would use, and which changes you need to make to the algorithm to answer each question. Indicate also how to pre-process the input for the algorithm. For example, if the question is: given a road map with distances between cities, find a shortest way of driving from city A to city B; your answer might be: build a graph in which every node is a city and an edge represents a road. The length of an edge is the length of the corresponding road. Use Dijkstra's algorithm to find a shortest path from A to B. This is a shortest route to drive from A to B.

19. (5 marks) There are n small islands in a lake and it is desired to build bridges to connect them so that each island can be reached from any other island via one or more bridges. The islands are numbered from 0 to $n-1$. The cost of constructing a bridge between islands i and j is $c(i,j)$. Describe an algorithm that determines which bridges must be built so that the total construction cost is minimum. For example, for the set of islands shown below, an optimum solution is the set of bridges in bold and it has total cost 16.

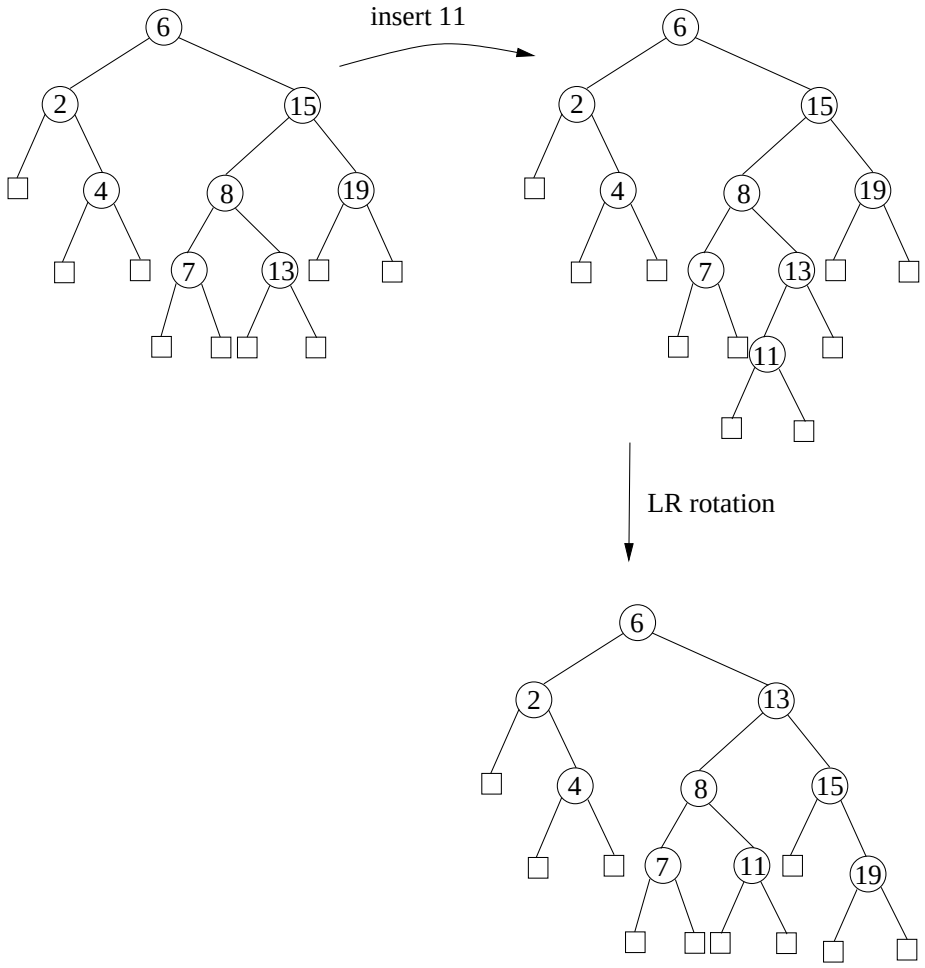


Build a graph in which every vertex represents an island and every edge corresponds to a potential bridge. The cost of each edge is given by the cost of the corresponding bridge. Use Prim's algorithm to compute a minimum spanning tree. This yields the minimum cost way of connecting the islands.

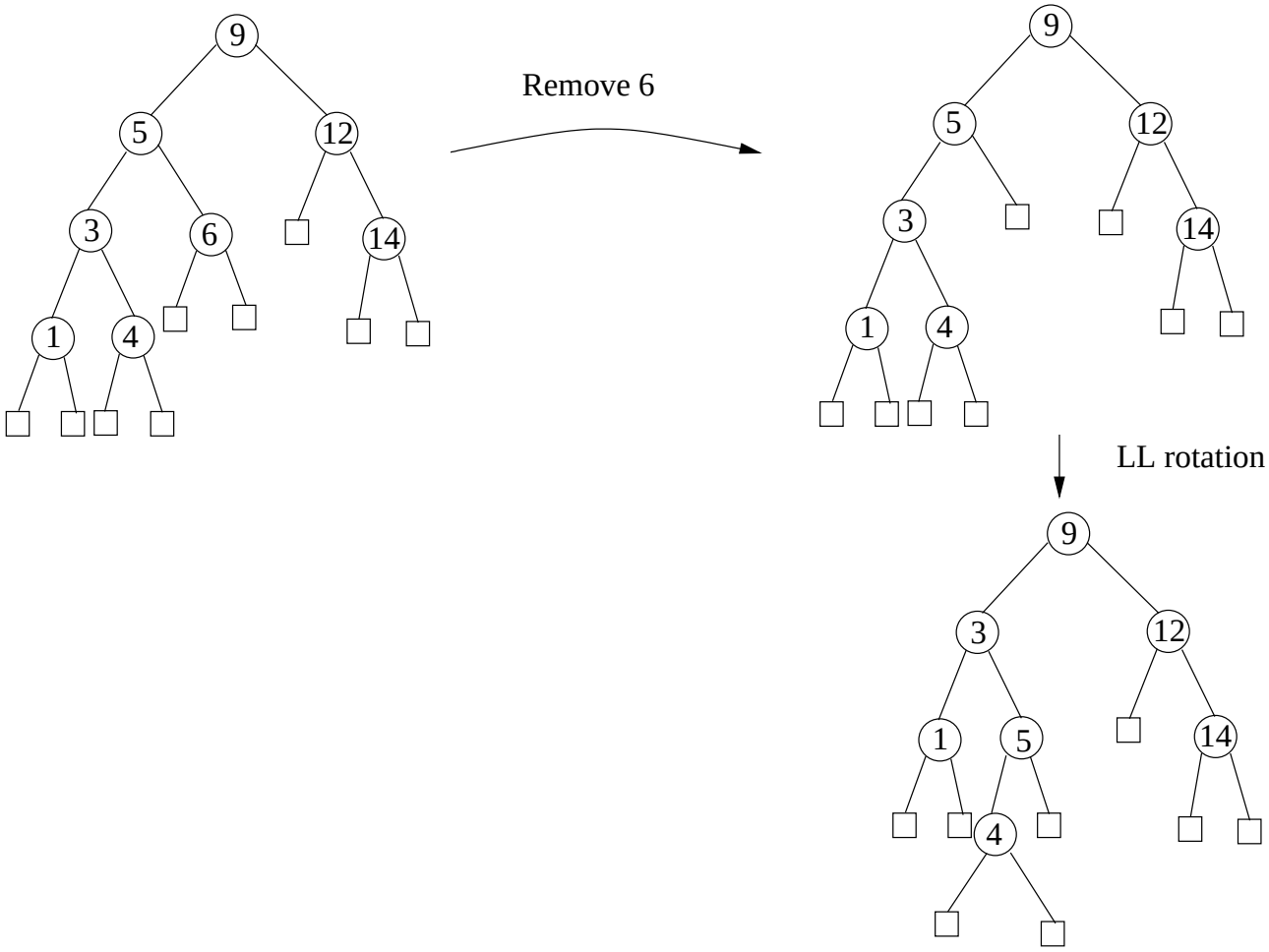
20. (5 marks) Let $G = (V, E)$ be a graph representing a communication network in which the edges are labelled either *slow* or *fast*. Describe an algorithm for finding a path from vertex s to vertex t with the smallest number of *slow* edges (it does not matter how many *fast* edges there are in the path as long as the number of *slow* edges is as small as possible).

    Assign to each *slow* edge of G a length of n and to every *fast* edge a length of 1. The use Dijkstra's algorithm to compute a shortest path from s to t. Since *slow* edges have cost larger than any simple path formed by *fast* edges, this shortest path will minimize the number of *slow* edges.

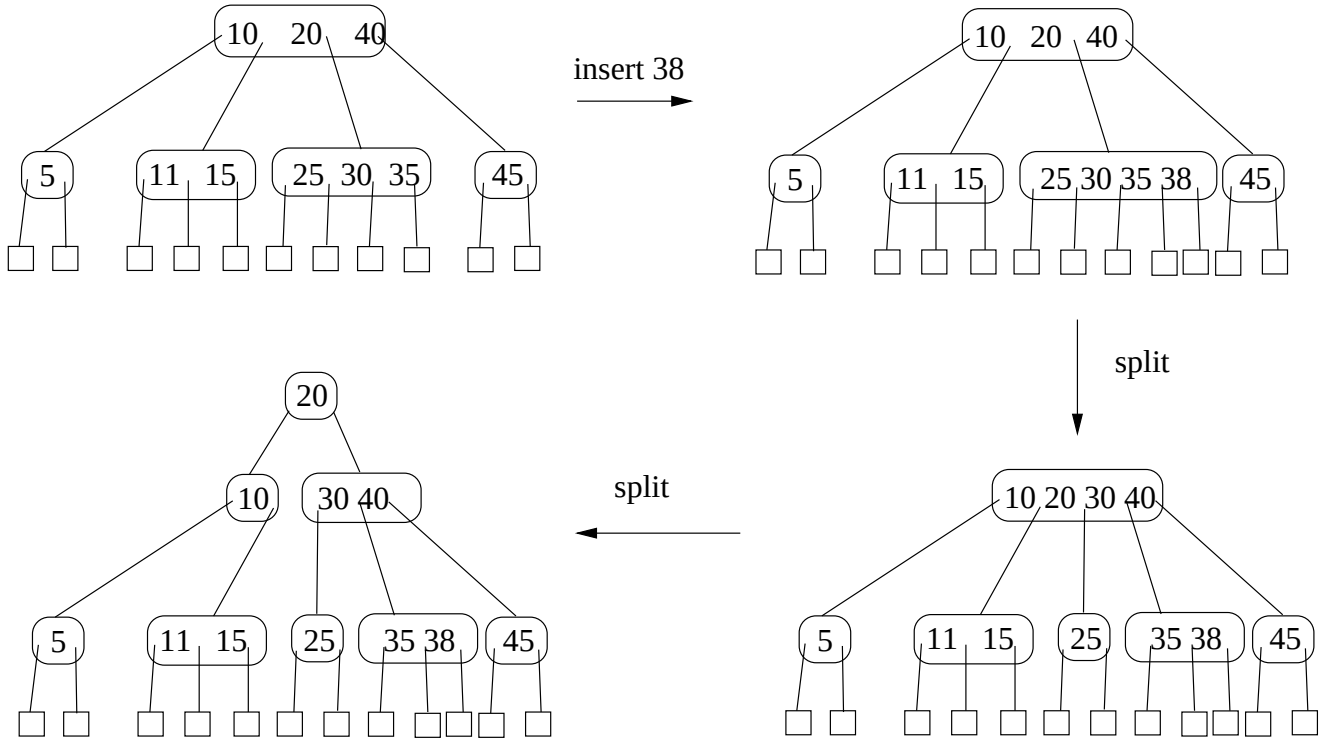
21. (5 marks) Consider the following AVL tree. Insert the key 11 into this tree and re-balance if needed. You **must** use the algorithms discussed in class, and if re-balancing is needed you **must** indicate the kind of rotation performed. Show **all** intermediate trees.



22. (5 marks) Consider the following AVL tree. Remove the key 6 and re-balance the tree if needed. You **must** use the algorithms discussed in class, and if re-balancing is needed you **must** indicate the kind of rotation performed. Show **all** intermediate trees.



23. (5 marks) Insert the key 38 in the following (2,4)-tree. You **must** use the algorithm discussed in class for inserting information in a $(2, 4)$-tree. Show **all** intermediate trees and **give** the name of the operation performed in each intermediate tree.



24. (5 marks) Delete the value 19 from the following (2,4)-tree. You **must** use the algorithm discussed in class for removing information from a $(2, 4)$-tree. Show **all** intermediate trees and **give** the name of the operation performed on each intermediate tree.