(a) Algorithm F1 searches for a number in a square matrix made of N rows and N columns.

```
M: matrix of integer numbers
N: number of rows (columns) of M
x: integer number

function F1(M,x)
    for 0 <= i < N
        for 0 <= j < N
            if(M[i,j]==x)
                return true
    return false
end function
```

C·N

(C·N)N

C

C

$$\begin{bmatrix} [1, 2, 3], \\ [4, 5, 6,], \\ [7, 8, 9,] \end{bmatrix}$$

Select ALL statements that apply. [4]

i. The worst and best case time complexity of F1 are the same
ii. The worst-case time complexity of F1 is $\Theta(N^2)$
iii. The best-case time complexity of F1 is $\Theta(N)$
iv. The best-case time complexity of F1 is $\Theta(1)$
v. The worst-case time complexity of F1 is $\Theta(N)$.

(b) Consider the following recursive algorithm:

```
A: array of integer numbers
N: number of elements in A
x: integer number

function F2(A,N,x)
    if(N==0):
        return -1
    if(A[N-1]==x)
        return N-1
    return F2(A,N-1,x)
end function
```

returns index 2

Assume that the array A is equal to [5,4,3,2,1]. What is the value returned by F2(A,5,3)? [4]

(c) Consider the same recursive algorithm of part (b):

```
A: array of integer numbers
N: number of elements in A
x: integer number

function F2(A,N,x):
    if(N<0):
        return -1
    if(A[N-1]==x):
        return N-1
    return F2(A,N-1,x)
end function
```

T(N)

C

C

C

C

T(N-1)

$T(N) = T(N-1) + C$

What is the recurrence equation describing its worst-case time complexity?

Choose ONE option:

i. $T(N)= T(N-1) + N$
ii. $T(N)= T(N-1) + C$ (C is a constant)
iii. $T(N)= T(N/2) + C$ (C is a constant)
iv. $T(N)= T(N/2) + N$
v. None of the others

(d) What pseudocode fragment should replace Z in the following comparison-based sorting algorithm?

```
A: array of integer numbers
N: number of elements in A

function Sort(A,N):
    for 0 <= j < N
       pos=j
       for j+1 <= i < N
           if(A[i]<A[pos])
               Z
       aux=A[j]
       A[j]=A[pos]
       A[pos]=aux
end function
```

Swap

Choose ONE option:

i. pos=pos+1
ii. pos=j
iii. swap(A[i], A[pos])
iv. pos=i
v. None of the others

(e) What of the following statements are **true**?

Select ALL statements that apply. [4]

i. The worst-case time complexity of a comparison sort cannot be better than Θ(NlogN)
ii. Worst-case time complexity of sorting can be Θ(N) using non-comparison sorts
iii. Counting sort cannot be used to sort decimal numbers
iv. Mergesort is one of the comparison sorts with best worst-case performance
v. The best-case time complexity of non-comparison sorts is Θ(1)

(f) An 8-element hash table uses linear probing to deal with collisions. The hash function is h(k)=(2*k+1)%8. Assume the hash table starts empty. What is the content of it after inserting the following numbers (in this order): 4, 27, 10, 9, 12? [4]

Choose ONE option:

i. [4, 27, 10, 9, 12, -1, -1, -1]
ii. [27, 10, -1, -1, 4, -1, -1, -1]
iii. [12, 10, -1, -1, 9, -1, -1, -1]
iv. [4, 9, 10, 27, 12, -1, -1, -1]
v. None of the others

BEDMAS

$h(4) = ([2 \cdot 4] + 1)\ \%\ 8 = 1$

$h(27) = ([2 \cdot 27] + 1)\ \%\ 8 = 7$

$h(10) = \quad = 5$

$h(9) = \quad = 3$

$h(12) = \quad = 1$

hashTable = [-1, 4, 12, 9, -1, 10, -1, 27]

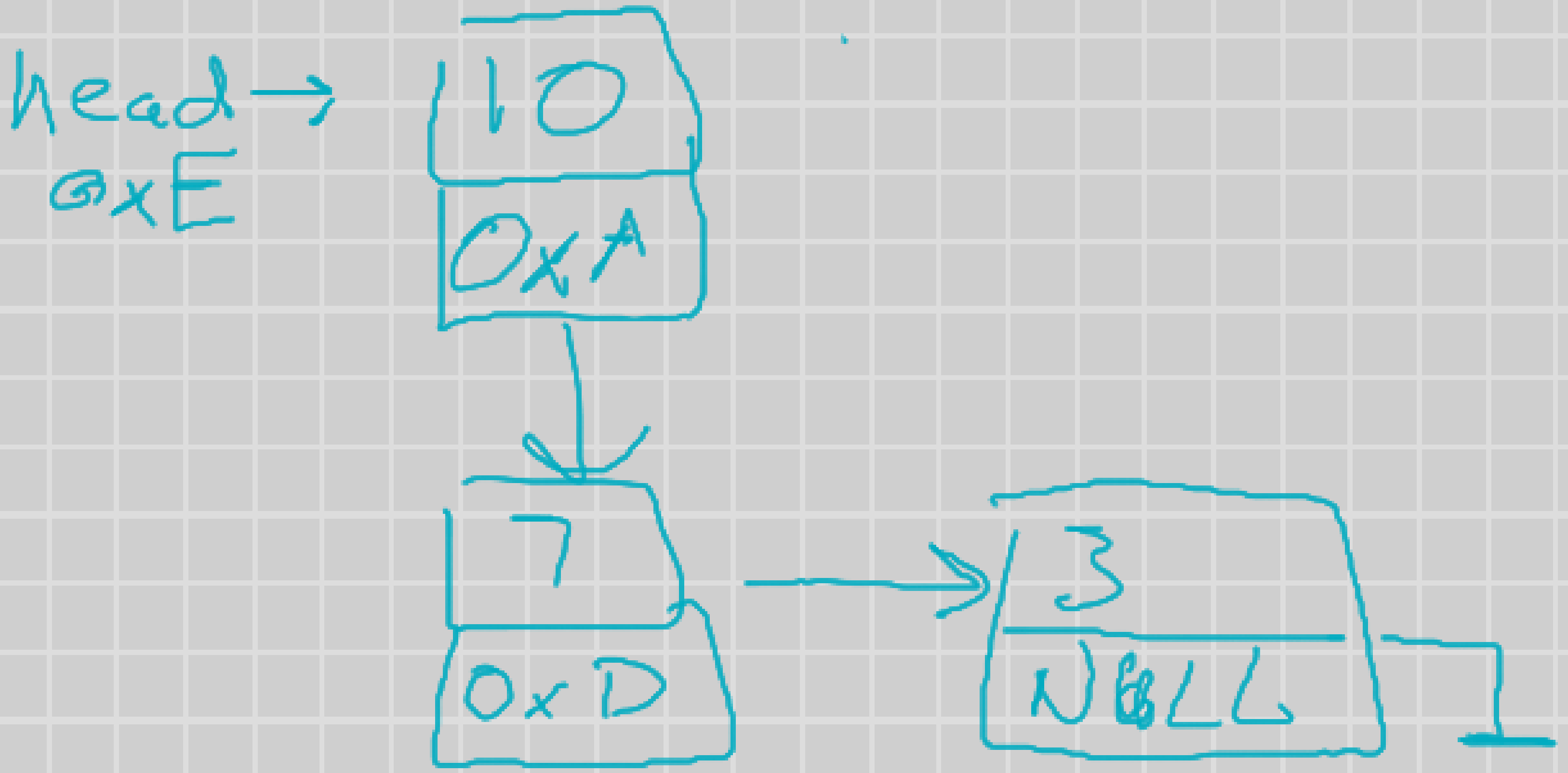
(g) Consider the following linked list:

head/0xA → 7/0xD → 3/NULL

A new node is inserted at the start of the list. Assume the new node stores number 10 and it is allocated memory address 0xE. What are the contents of the new list? [4]

Choose ONE option:

i. head/0xA → 7/0xD → 3/NULL → 10/0xE
ii. head/0xA → 10/0xE → 7/0xD → 3/NULL
iii. head/0xE → 10/0xA → 7/0xD → 3/NULL
iv. head/0xA → 7/0xD → 3/0xE → 10/NULL
v. None of the others



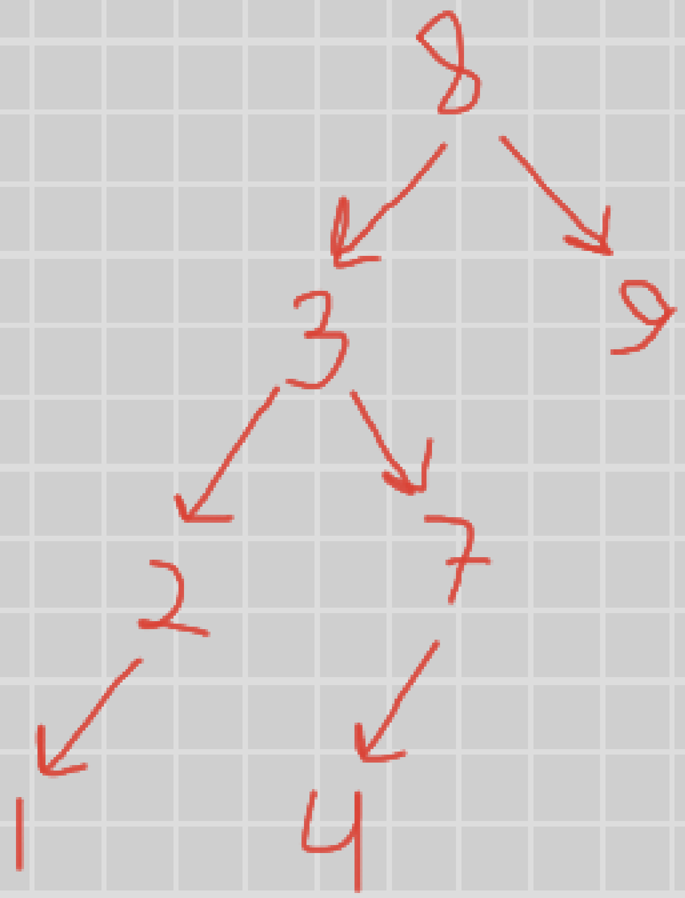
(h) The following numbers are inserted in a Binary Search Tree (in this order): 8, 3, 7, 2, 1, 9, 4

What information is printed on screen when traversing the tree with the algorithm shown below? [4]

```
function traverse(T)
    if(T.root!=NULL)
        traverse(T.right)
        traverse(T.left)
        print(T.root)
end function
```

Choose ONE option:

i. 9 4 7 1 2 3 8
ii. 1 2 4 7 3 9 8
iii. 1 2 3 4 7 8 9
iv. 9 8 7 4 3 2 1
v. None of the others



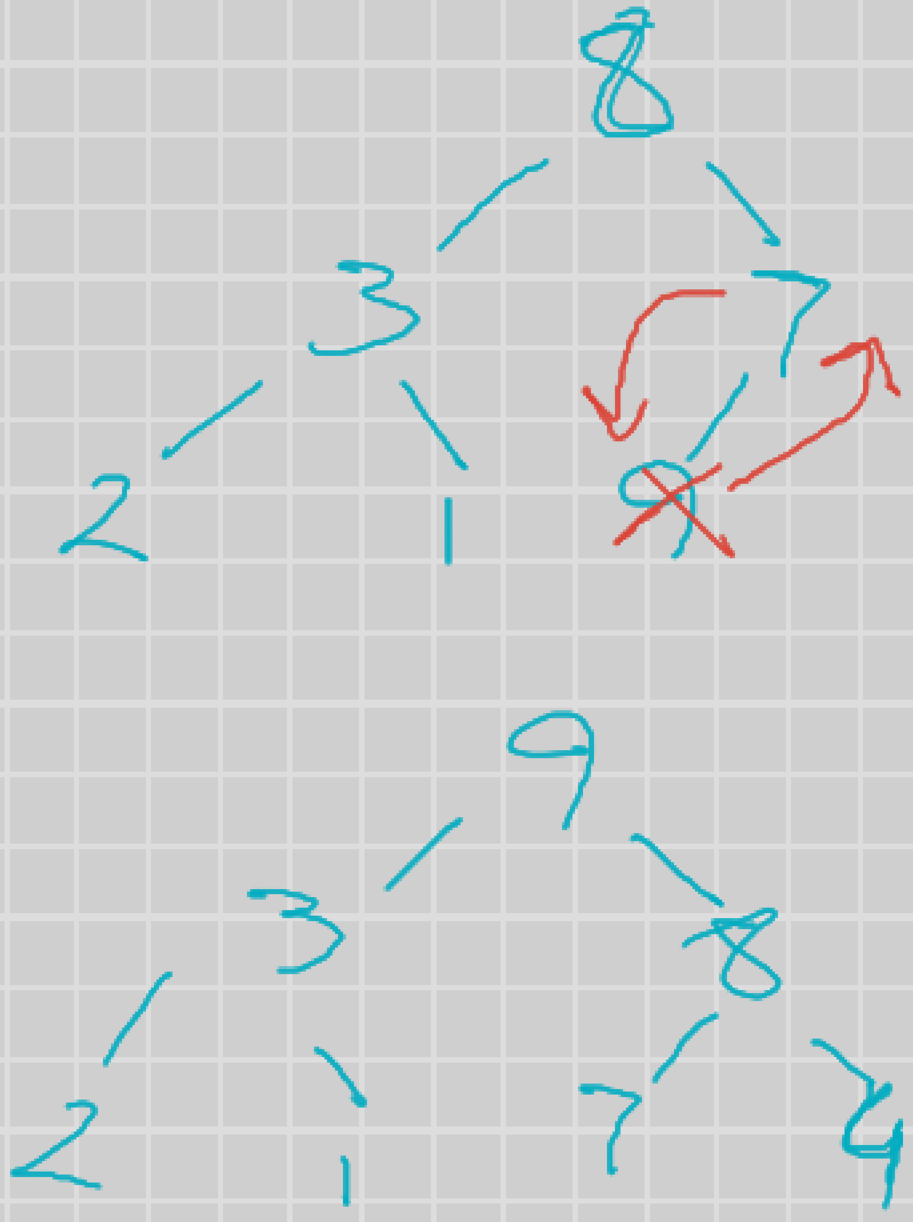
9 4 7 1 2 3 8

(i) The following numbers are inserted, one by one, in a max-heap : 8, 3, 7, 2, 1, 9, 4

What is the content of the array storing the heap? Position 0 in the array is the leftmost position [4]

Choose ONE option:

i. [8, 3, 7, 2, 1, 9, 4]
ii. [8, 3, 9, 2, 7, 1, 4]
iii. [1, 2, 4, 8, 3, 9, 7]
iv. [9, 3, 8, 2, 1, 7, 4]
v. None of the others



[9, 3, 8, 2, 1, 7, 4]

(j) Consider the graph represented by the following adjacency matrix:

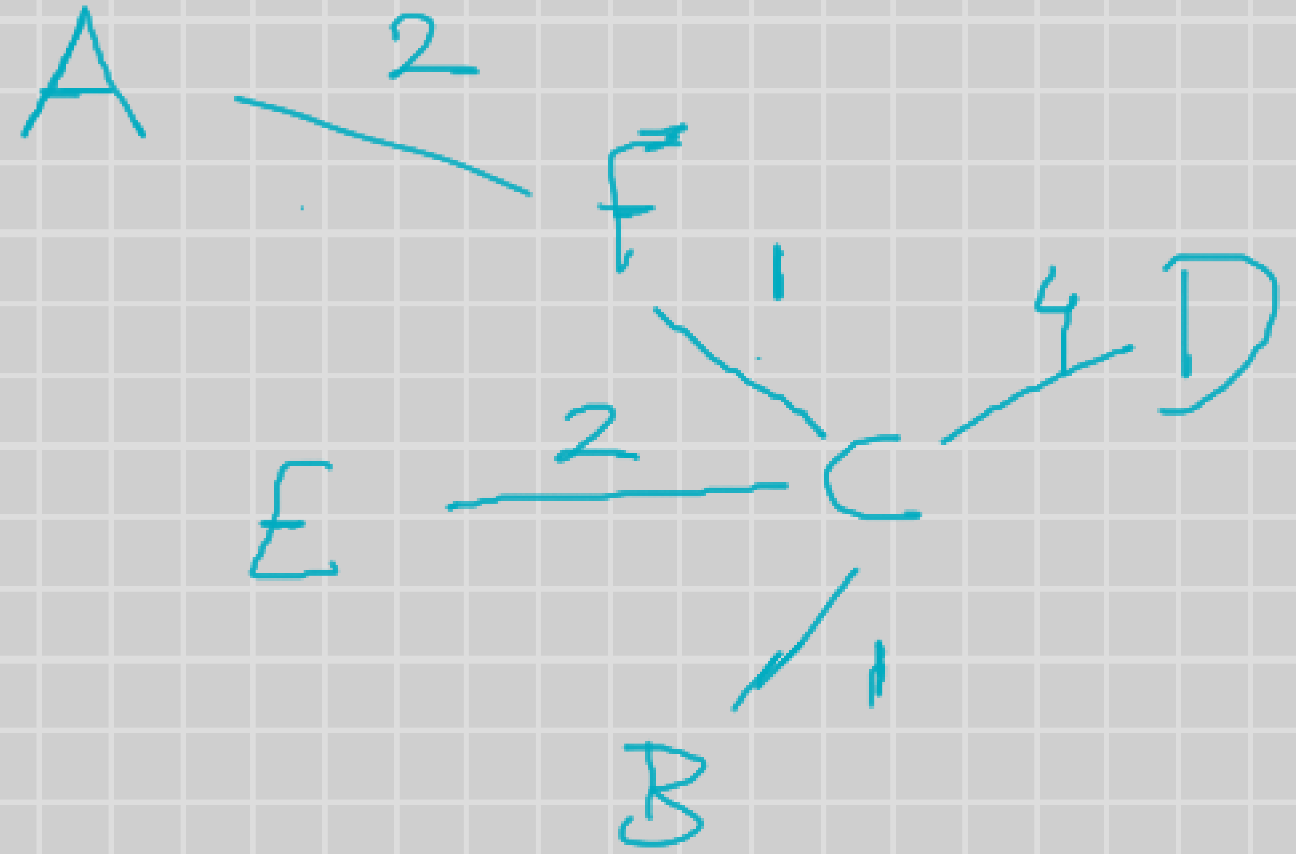
	A	B	C	D	E	F
A	~~1~~	4	10	5	8	2
B	4	~~3~~	~~1~~	7	9	3
C	10	1	~~3~~	4	2	~~1~~
D	5	7	4	~~12~~	6	10
E	8	9	2	6	~~11~~	5
F	2	3	1	10	5	~~7~~

What is the cost of the minimum spanning tree? [4]

Choose ONE option:

i. 9
ii. 10
iii. 8
iv. 12
v. None of the others

Total = 10



$A \xrightarrow{2} F$

$F \xrightarrow{1} C$

$C \xrightarrow{1} B$

$C \xrightarrow{2} E$

$C \xrightarrow{4} D$

10

Question 2

The following algorithms, A1 and A2, solve the same problem. To do so, they receive as input arguments:

root: the root of a binary search tree storing integer numbers
x: an integer number

ALGORITHM 1

```
function A1(root, x)
   Q = new Queue()
   ENQUEUE(Q,root)
   while !ISEMPTY(Q)  do
        t = PEEK(Q)
        if (t==x)
             return TRUE
        else
        ENQUEUE(Q,left(t))
        ENQUEUE(Q,right(t))
        DEQUEUE(Q)
   end while
   return FALSE
end function
```

Breadth first

Note: the function ENQUEUE only inserts a new element in the queue if this element is different from NULL

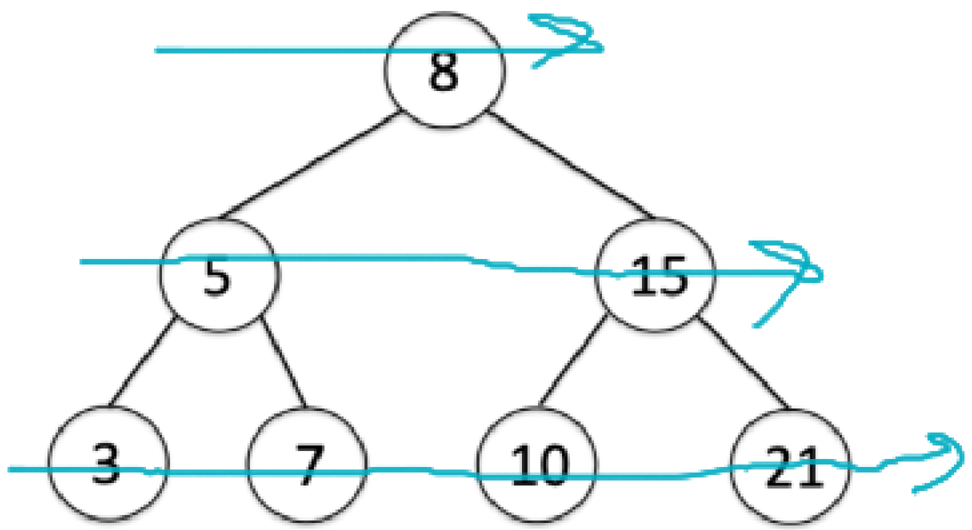
ALGORITHM 2

```
function A2(root,x)
   if (root==NULL)
      return FALSE
   else
      if(x == root->data)
         return TRUE
      else
         if (x< root->data)
           return A2(root->left,x)
         else
           return A2(root->right,x)
end function
```

Depth-first

Q=[~~15~~, ~~10~~, ~~21~~, ~~3~~, ~~12~~, ~~18~~, ~~25~~]

(a) For the following BST:



What is the content of the queue **immediately before returning** from the execution of A1(root,21)? [4]

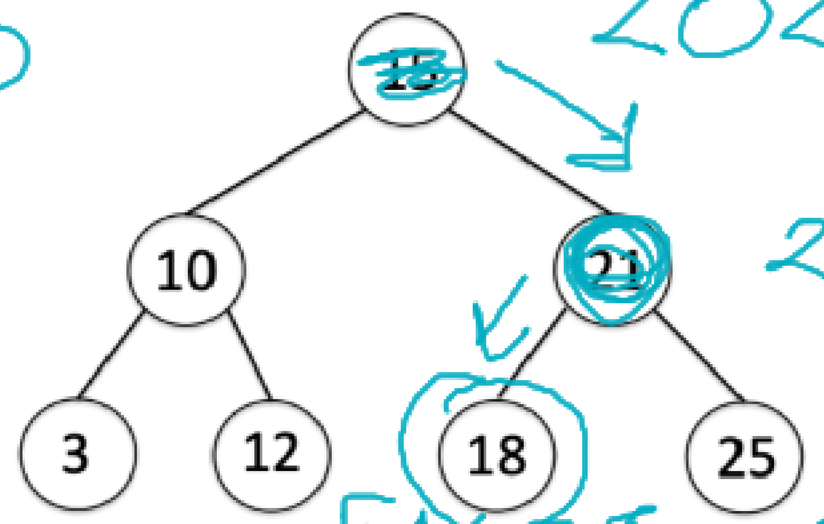
Q=[~~8~~, ~~5~~, ~~15~~, ~~3~~, ~~7~~, ~~10~~, 21]

returns TRUE

(b) For the following BST: [4]

20

20<15



20<21

20<18

FALSE

What is the return value of A2(root, 20)? FALSE

(c) What is the task performed by algorithms A1 and A2? Don´t forget to mention the return values for the different cases [4]

(d) What is the worst-case time complexity of A1? Use Theta notation [1] and explain your reasoning [3] [4]

(e) Assuming a fully balanced BST (a BST with all its levels fully populated) of N elements, what is the recurrence equation describing the running time of A2? [4]

(f) What is the worst-case time complexity of A2? Use Theta notation [1] and show your workings [3] [4]

(g) Which algorithm do you recommend to implement? Why? [6]

c) A1 is a breadth first search, returns true when the value searched for is present, false otherwise.

A2 is a depth-first search, returns true when the value searched for is present, false otherwise.

d) Algo A1, must traverse the entire queue in the worst case, so the worst case time complexity is THETA(N).

$$2.\ f(n) = n^{\log_b a} \Longrightarrow T(n) = \Theta(n^{\log_b a} \log n)$$

e) T(N) = T(N/2) + C

```
function A2(root,x)                    T(N)
   if (root==NULL)                     C1
      return FALSE
   else
      if(x == root->data)              C2
         return TRUE
      else
         if (x< root->data)            C3
           return A2(root->left,x)
         else
           return A2(root->right,x)    T(N/2) + C
end function
```

e) $T(N) = T(N/2) + C$

Use master theorem to for (f).

f) $a = 1$

$b = 2$

$f(n) = C$

$f(n) = n^{\log_b a}$

$= n^{\log_2 1}$

$= n^0$

$= 1$

Compare $f(n)$, to $\log_b a$

2. $f(n) = n^{\log_b a} \Longrightarrow T(n) = \Theta(n^{\log_b a} \log n)$

$C = = = 1 = C$

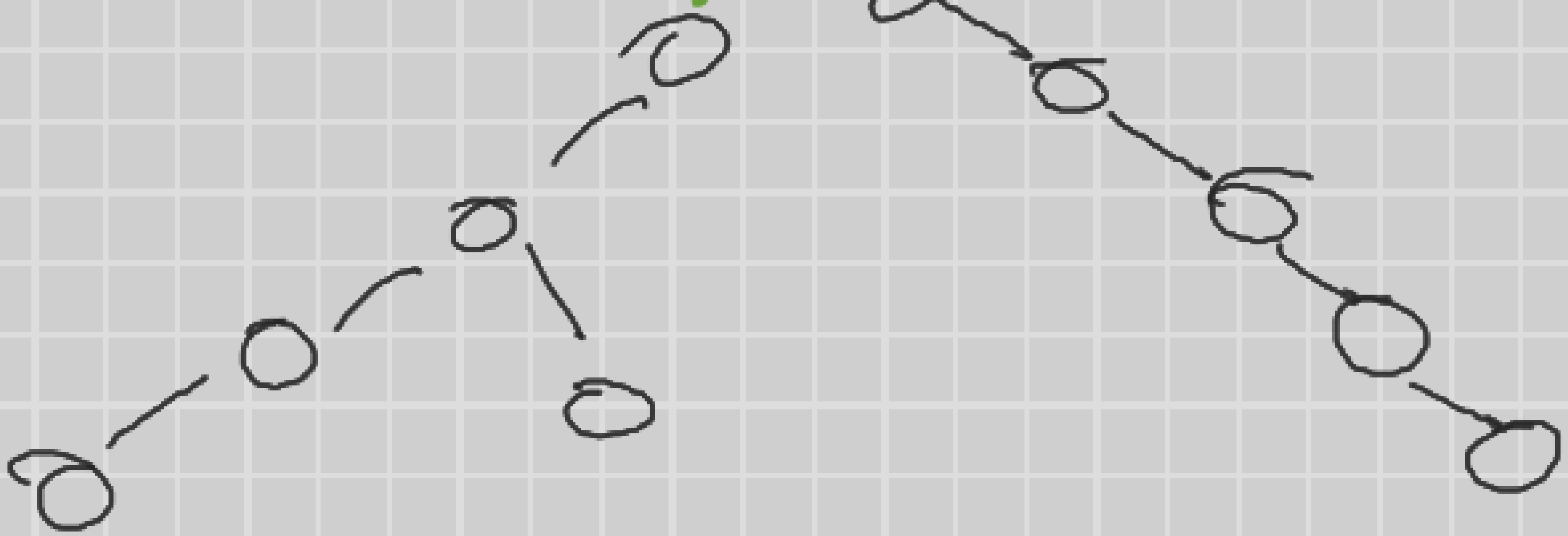
$T(N)$ is $\Theta(n^{\log_b a} \log n) = \Theta(n^0 \log n)$

$= \Theta(\log n)$

(g) Which algorithm do you recommend to implement? Why? [6]

Assuming that we're working with a full balanced BST, I would select algo A2 because it has a logarithmic time complexity.

In almost all cases algo A2 is preferrable to A1, because even in an unbalanced tree A2 will not perform worse than A1. A binary search tree will allow us to recursively search for a value.



Question 3

The data structure shown in Figure 1 can be thought as a list of two lists. In this case, the data structure is used to classify numbers in one of 2 groups: even numbers and odd numbers.

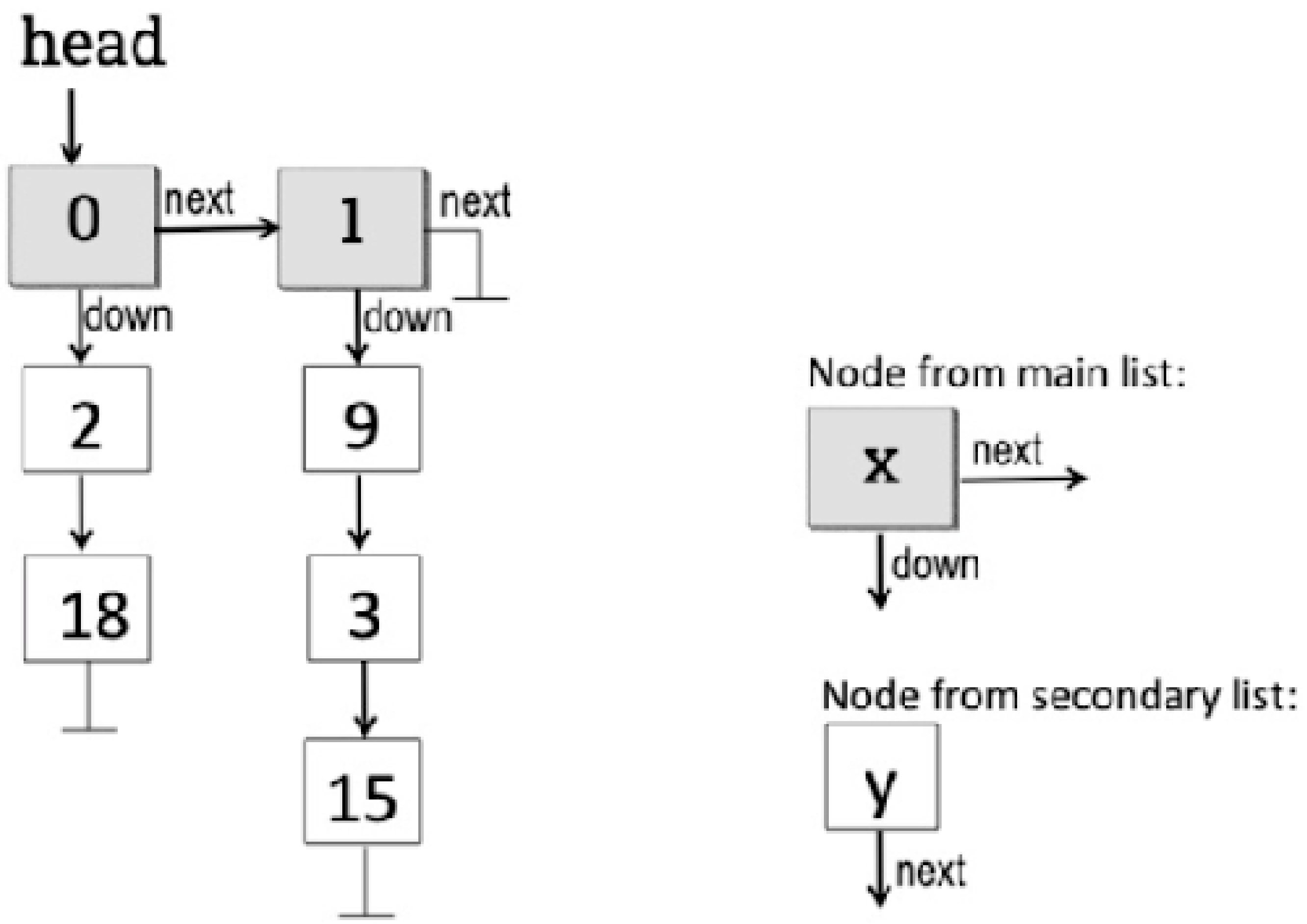


Figure 1. A list of two lists

The first node of the main list (the list drawn horizontally, made of shaded nodes) stores a number 0 to signal that the numbers stored in its secondary list (the list drawn vertically, 'hanging' from node storing number 0) are even (in the figure, the numbers 2 and 18). The second node of the main list stores a number 1, to signal that the numbers stored in its secondary list (in the figure, the numbers 9, 3 and 15) are odd.

(a) In a single linked list implemented in C++ this is the definition of a node: [6]

```
struct Node {
      int data;
      Node *next;
   };
```

This definition is useful for the nodes belonging to the secondary lists, but not for the nodes belonging to the main list. Main list nodes need two pointers (one for the next node in the main list and one for the first node in the secondary list). Assuming that the above definition is kept for the nodes of the secondary lists, propose a new definition of node for the nodes of the main list. Call this type of node Node_main and use the names of pointers given in Figure 1.

```
struct Node_main {
    int data;      // 0 or 1
    Node_main* next;
    Node* down;
};
```

(b) Write the pseudocode of the function INSERT(head,x) that inserts a new number in this data structure. If the number is even it must go to the first secondary list (the one 'hanging' from node 0 in the main list). Otherwise, it must go to the second secondary list. Assume the data structure already has the main list created and numbers are inserted at the start of the secondary list. [8]

(c) Write the pseudocode of the function SEARCH(head,x) that returns TRUE if the number x is in the data structure (in any of the secondary lists) and FALSE otherwise. [8]

```
b) function Insert(head, x)
       tmp <- head
       if (x mod 2 == 1)
           tmp <- head.next
       end if
       newNode <-- new Node(x)
       newNode.next <-- tmp.down
       tmp.down <-- newNode
   end function
```

```
if (x mod 2 = 1)
    tmp <- head.next

tmp <- tmp.down
```

```
c) function SEARCH(head, x)
       tmp <-- head // this pointer is used to traverse the list

       if ( x mod 2 = 0) then
          tmp <--head.down // point to the next node in the 0 list

       if ( x mod 2 = 1) then
          tmp <-- head.next.down // point to the next node in main list
                                 // points to 1st node in secondary lis

       while (tmp != NULL) do
           if (tmp.data == x) then
               return TRUE
           tmp <-- tmp.next

       return FALSE
   end function
```

(d) Write the pseudocode of the function DELETE(head, b) that receives as input arguments the head of the lost of two lists and a Boolean vale. The function DELETE(head,b) deletes one of the main nodes. If b equal 0, then the node storing number 0 is deleted. Otherwise, the main node storing number 1 is deleted. Consider the cases where: the main list is empty, it has only one node (node 0 or 1) or the two of them. [8]

The bool is either 0 or 1. If bool is 0 then it deletes the first node of the primary list, making the list inaccessible, ditto if the bool is 1. If both list are present and 0 is passed in, then head should point to the secondary list.

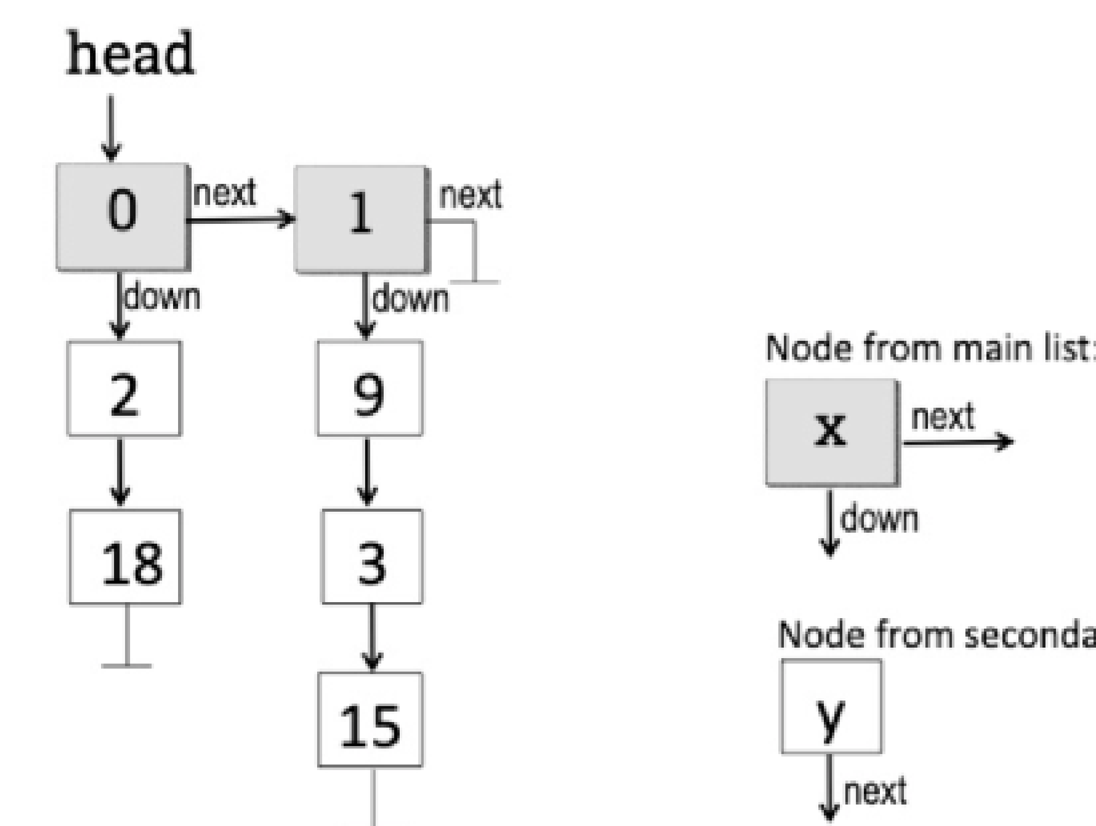
Cases:

1) List is empty (head is NULL)
2) If 0 and 1 lists are present
3) If only 1 list is present
4) If only 0 list is present

```
function DELETE(head, bool)
   if (head == NULL)
      return

   if (bool == 0 && head.data == 0) then                  // We make the 0 list inaccessible
     head <-- head.next                                     // points the 1 list or NULL. Covers case #

   if (bool == 1) then                                     // Make the 1 list inaccessible
     if (head.data == 0 && head.next != NULL) then  // if both lists are present
        head.next <-- NULL
     if (head.data == 1) then                              // if only list 1 is present
        head <-- NULL
```



Question 4

A software developer needs to solve the following problem: given the adjacency matrix of an **undirected weighted** graph, find the value of the k-th minimum cost edge. Assume that **all edge weights are different**, non-negative integer numbers, and not greater than 999. The number 1000 (one thousand) signals the absence of and edge.

For example, for the graph represented by the following adjacency matrix M:

	A	B	C	D
A	1000	3	1	5
B	3	1000	7	4
C	1	7	1000	9
D	5	4	9	1000

The 1^{st} minimum (that is, k=1) is 1, the second minimum (k=2) is 3, the third minimum (k=3) is **4**, and so on.

The algorithm to design must take as input arguments the adjacency matrix (M), its number of nodes (N) and the value of k. It must return the value of the k-th minimum cost edge.

The software developer came up with these two algorithms to solve the problem:

Algorithm 1:

1. Make a copy of the adjacency matrix. Call the copy M_copy.
2. Create a variable, called `min`, where the minimum value is recorded
3. Create a variable, called `count`. Initialise its value to 0
4. Visit every element of M_copy, from top to bottom, from left to right and record the minimum value in `min`
5. Once the minimum value is found, increase the variable `count` by one unit
6. If the condition `count==k` is true, return the value of `min`. Otherwise, delete the minimum value from M_copy (write number 1000 in the corresponding positions) and repeat steps 2-6

Algorithm 2:

- Create a min-heap storing the values of all edges
- Perform EXTRACT_MIN k times. The value last extracted is the k-th minimum.

Input:
1) adjacency matrix
2) # of Nodes
3) value of k (the cost of an edge)

(a) Write the pseudocode of Algorithm 1 [8]

```
function A1(matrix, numNodes, k)
  mCopy <-- matrix
  min <-- 1000
  count <-- 0
  matSize <-- LENGTH[matrix]
  m, n <-- 0

  while TRUE do
    for (0 <= i < matrixSize)
      for (i+1 <= j < matrixSize)
        if (mCopy[i][j] < min)
          min <-- mCopy[i][j]
          m <-- i
          n <-- j

    count <-- count + 1

    if (count == k)
      return min

    mCopy[m][n] <--1000
    mCopy[n][m] <--1000
  end while
```

Input:

1) adjacency matrix

2) # of Nodes

3) value of k (the cost of an edge)

Algorithm 1:

1. Make a copy of the adjacency matrix. Call the copy M_copy.
2. Create a variable, called `min`, where the minimum value is recorded
3. Create a variable, called `count`. Initialise its value to 0
4. Visit every element of M_copy, from top to bottom, from left to right and record the minimum value in `min`
5. Once the minimum value is found, increase the variable `count` by one unit
6. If the condition `count==k` is true, return the value of `min`. Otherwise, delete the minimum value from M_copy (write number 1000 in the corresponding positions) and repeat steps 2-6

Question 4

A software developer needs to solve the following problem: given the adjacency matrix of an **undirected weighted** graph, find the value of the k-th minimum cost edge. Assume that **all edge weights are different**, non-negative integer numbers, and not greater than 999. The number 1000 (one thousand) signals the absence of and edge.

For example, for the graph represented by the following adjacency matrix M:

	A	B	C	D
A	1000	3	1	5
B	3	1000	7	4
C	1	7	1000	9
D	5	4	9	1000

The 1st minimum (that is, k=1) is 1, the second minimum (k=2) is 3, the third minimum (k=3) is 4, and so on.

The algorithm to design must take as input arguments the adjacency matrix (M), its number of nodes (N) and the value of k. It must return the value of the k-th minimum cost edge.

The software developer came up with these two algorithms to solve the problem:

(b) Write the pseudocode of Algorithm 2. Assume you already have implemented the min-heap functions:

- INSERT(heap,x):insert number x into the heap. Worst-case Theta(logN)
- BUILD_HEAP(A): build a min heap in place. Worst-case Theta(N)
- EXTRACT_MIN(heap): return the minimum value stored in the min-heap. Worst-case Theta(logN)

That is, you can use these functions with no need to write the pseudocode for them. [8]

```
function A2(matrix, numNodes, k)
    minHeap <-- new Min-Heap
    matrixSize <-- matrix.length
    min <-- 1000

    for (0 <= i < matrixSize)
        for (i+1 <= j < matrixSize)
            INSERT(heap, matrix[i][j])

    for ( 0 <= i < k)
        min <-- EXTRACT_MIN(heap)
        BUILD_HEAP(heap)

  return  min
```

Algorithm 2:

- Create a min-heap storing the values of all edges
- Perform EXTRACT_MIN k times. The value last extracted is the k-th minimum.

Question 4

A software developer needs to solve the following problem: given the adjacency matrix of an **undirected weighted** graph, find the value of the k-th minimum cost edge. Assume that **all edge weights are different**, non-negative integer numbers, and not greater than 999. The number 1000 (one thousand) signals the absence of and edge.

For example, for the graph represented by the following adjacency matrix M:

	A	B	C	D
A	1000	3	1	5
B	3	1000	7	4
C	1	7	1000	9
D	5	4	9	1000

The 1st minimum (that is, k=1) is 1, the second minimum (k=2) is 3, the third minimum (k=3) is 4, and so on.

The algorithm to design must take as input arguments the adjacency matrix (M), its number of nodes (N) and the value of k. It must return the value of the k-th minimum cost edge.

The software developer came up with these two algorithms to solve the problem:

k is the index in a sorted collection of edge costs, counted from k=1. We're searching for the kth lowest cost edge.

(c) What are the worst-case time complexities of A1 and A2? Use Theta-notation. Justify your findings. [8]

(d) What algorithm do you recommend for implementation? Justify in terms of worst-case time complexity. [6]

A2

```
function A1(matrix, numNodes, k)
    mCopy <-- matrix
    min <-- 1000
    count <-- 0
    matSize <-- LENGTH[matrix]
    m, n <-- 0

    while TRUE do
      for (0 <= i < matrixSize)
        for (i+1 <= j < matrixSize)
          if (mCopy[i][j] < min)
            min <-- mCopy[i][j]
            m <-- i
            n <-- j

      count <-- count + 1

      if (count == k)
        return min

      mCopy[m][n] <--1000
      mCopy[n][m] <--1000
    end while
```

N

$N \cdot N$

$N \cdot N \cdot N = N^3$

This is where the halving occurs (slightly less than half)

A1 has a time complexity of θ(N^3)

```
function A2(matrix, numNodes, k)
    minHeap <-- new Min-Heap
    matrixSize <-- matrix.length
    min <-- 1000

    for (0 <= i < matrixSize)
        for (i+1 <= j < matrixSize)
            INSERT(heap, matrix[i][j])

    for ( 0 <= i < k)
        min <-- EXTRACT_MIN(heap)
        BUILD_HEAP(heap)

    return  min
```

$N^2 \cdot \log N$

$k \cdot \log N$

$K \cdot N$

A2 has a time complexity of θ(N^2 x log N)

	A	B	C	D
A	1000	3	1	5
B	3	1000	7	4
C	1	7	1000	9
D	5	4	9	1000

This is what is traversed in A1 and A2