Final Assessment

Thi Van Anh DUONG Student ID: 90023112

Diploma of Information Technology, Curtin College

DSA1002: Data Structures and Algorithms

Coordinator: Khurram Hameed

23 September 2022

Student Declaration of Originality

☒	This assignment is my own original work, and no part has been copied from another student's work or source, except where it is clearly attributed.
☒	All facts/ideas/claims are from academic sources are paraphrased and cited/referenced correctly.
☒	I have not previously submitted this work in any form for THIS or for any other unit; or published it elsewhere before.
☒	No part of this work has been written for me by another person.
☒	I recognise that should this declaration be found to be false, disciplinary action could be taken and penalties imposed in accordance with Curtin College policy.

Electronic Submission:

☒	I accept that it is my responsibility to check that the submitted file is the correct file, is readable and has not been corrupted.
☒	I acknowledge that a submitted file that is unable to be read cannot be marked and will be treated as a non-submission.
☒	I hold a copy of this work if the original is damaged, and will retain a copy of the Turnitin receipt as evidence of submission.

Question 1: General Understanding (Total Marks: 10)

A. Consider a situation where you need to store the information of friends list of users on a social networking application. The list of friends is implemented as a dynamic data structure, where friends can be added or removed when required. The data structure should be able to track the mutual friends of multiple users, where connections between users are random and do not have any specific priority/order.

i. Describe your selection of the ADT to store individual user record. **(3 marks)**

I choose **Graph**, because:

Supported operands:

- The list of friends is created as a dynamic data structure, friends can be added or removed ➔ can be Tree, Graph, Heap or Linked List.
- Connections between users are random and do not have specific priority ➔ cannot be **Heap** and **Tree**.
- Be able to track mutual friends of multiple users ➔ mention to the connectivity and relationships between objects ➔ cannot be **Linked List** (because each node just knows about it next node and previous node, it does not show us the connections)

Computational Complexity:

Data Structure	Time Complexity					
	Storage	Add Vertex	Add Edge	Remove Vertex	Remove Edge	Query
Adjacency list	O(|V|+|E|)	O(1)	O(1)	O(|V| + |E|)	O(|E|)	O(|V|)
Incidence list	O(|V|+|E|)	O(1)	O(1)	O(|E|)	O(|E|)	O(|E|)
Adjacency matrix	O(|V|^2)	O(|V|^2)	O(1)	O(|V|^2)	O(1)	O(1)
Incidence matrix	O(|V| · |E|)	O(|V| · |E|)	O(|V| · |E|)	O(|V| · |E|)	O(|V| · |E|)	O(|E|)

Easy to add and remove!

Programing elegance:

The organization of graph is easy to implement and use in this case.

➔ Therefore, **Graph** is the best Abstract Data Type to implement not only this social networking application but any real-world system.

ii. Discuss the effectiveness of your selection for maintaining the friends list and mutual friends list determination. **(3 marks)**

- For maintaining the friends list:

  We can create a graph by considering each person as a vertex, pointing the edge between two nodes to indicate the two people are friends. By doing it, whenever user adds new friends, that person will be add to the network (graph), interact with others and expand their friend lists and vice versa when remove friends.

  + Add Vertex: $O(V^2)$ with V is vertex.

  + Add Edge: O(1)

  + Remove Vertex: $O(V^2)$

  + Remove Edge: O(1)

- For tracking mutual friends list:

  Means we find the path between two people. Firstly, we investigate with one person and do Bread-First-Search, or bidirectional BFS one from the source person and one from the one we want to check mutual friend. If we can find a path, means they have mutual friend.

  + Time complexity of is O(V + E) when Adjacency List is used and O(V^2) when Adjacency Matrix is used, where V stands for vertices and E stands for edges.

B. The student record management software needs to store the information of thousands of students at a time. The information of each student is sorted based on the student ID. Each student is tracked by his/her ID. This system also requires quick student record searching regardless of number of students (data size) for adding, removing, editing and updating the record.

   i. Support selection of a particular ADT based on the asymptotic time complexities (Big O notation) for record insertion, removal, traversal and record printing. **(4 marks)**

I choose **Hashtable**, because:

Supported operands:

- Store the information of thousand students at a time. ➔ Hastable is a complex data structures, can connect large amount of interconnected data at a time.

Computational Complexity:

- The system requires quick student records searching regarding of number of students with time complexity O(1) – best case and O(N) – worst case.
- Easily adding, removing, editing and updating the record with time complexity O(1) – best case and O(N) – worst case.

For more details, I did provide in the table below:

ACTIVITY	BEST CASE COMPLEXITY	AVERAGE CASE COMPLEXITY	WORST CASE COMPLEXITY
Searching	O(1)	O(1)	O(n)
Insertion	O(1)	O(1)	O(n)
Deletion	O(1)	O(1)	O(n)
Space Complexity	O(n)	O(n)	O(n)

Programing elegance:

The functional hastable is easy to implement and use in this case.

➔ Therefore, **Hastable** is an ADT completely suitable for this purpose.

Question 2: Recursion and Sorting (Total Marks: 16)

A. Consider the situation where you are supposed to sort data of all students (on student ID) in Curtin College/University (approximately 50,000 students or more). Support your selection of two best sorting algorithms describing stable vs unstable and in-place vs non-in-place property of the selected sorts. **(4 marks)**

For the data requirements:

- Data size: ~ 50.000 students or more ➔ choose sorting algorithms that works well on large data set
- Attribute: Sorting based on student ID (and the things is student ID never be duplicated) ➔ sorting algorithms cannot care about stable attribute (but if it is stable, it would be better).
- Space: Curtin College/ University will receive and update new students years by years, so I assume their space is available, but to avoid making the whole system crash I would prioritize space capacity to select a sorting algorithms.
- Time: is the **most important** criteria. When Curtin Staff working with student records when it comes to reports, student class lists, grades, etc… it can not take ages to peform tasks.

For afore-mentioned reasons, I choose:

Quick Sort is my first choice, because:

√ works well on large data sets.

√ Time complexity: running time is small compared to other sorting types.

Best case	Worst Case	Average Case
O(NlogN)	$O(N^2)$	O(NlogN)

√ In-place sort: don't require extra space for manipulating input (studentID).

X Unstable sort: because it does swapping of elements according to pivot's position (without considering their original positions). ➔ as I said above, this criteria is not important, we can ignore it.

√ Has tight code, easy to implement.

Merge Sort because:

√ works well on large data sets

√ Time complexity: running time is smallest compared to other sorting types including worst case scenario.

Best case	Worst Case	Average Case
O(NlogN)	O(NlogN)	O(NlogN)

X Non-in-place sort: it requires an extra array to merge the sorted subarrays.

√ Stable sort: does not change the order of element has same values ➔ 2 students cannot have the same student ID

√ Even though, in worst case, merge sort is slightly faster than quick sort, but it consumes extra spaces as my reason above. However, If stability is important and space is available, merge sort would be my first choice.

B. Consider the application of merge sort for low memory tablet-based computer for sorting. Will this be good choice of the described application, justify your answer by considering O(NlogN) time complexity and memory required to execute the sort. **(4 marks)**

In my perspective, I highly believe that it is not a good choice to perform sorting by using merge sort for low memory tablet-based computer.

It can be said that, memory allocated is the biggest drawback of merge sort. Looking at the table of time complexity and space complexity below:

Best case	Worst Case	Average Case	Space complexity (worst case)
O(NlogN)	O(NlogN)	O(NlogN)	O(N)

Even though, merge sort has ideal running time faster, it consumes extra memory up to O(N) when performs sorting. This is simply because, merge sort implemented by

using recursive function and requires an extra array to merge the sorted subarrays. If we use merge sort for a low-space computer, it can be affected to the performance of merge sort. The program will continue running and waiting for the memory allocated, which therefore time complexity will be extended due to short of memory. Finally, the whole program can be crashed.

Merge sort can be a good sort, but in this case it does not, space availability is the big problem.

C. Presence of redundant data elements (i.e., duplicate numbers in a list) could affect the execution of a sorting algorithm, if yes how it can affect the complexity of sorting. **(3 marks)**

In my opinion, duplicate elements can affect the complexity of sorting. In a list, with 2 identical values, its position is not important because they are the same. What I mean by that is, the list will not change when we change the two duplicate numbers, but we still have to swap them when perform sorting. Therefore, it increases the number of operations need to processing ➔ increase the complexity of algorithms.

In order to demonstrate it, I will test a list of duplicated number with 2 representative sorts: Bubble sort and Merge sort.

Unduplicated list : `A = [9,5,6,3,4,1,7,0,2,8]`

Duplicated list: `A = [9,9,6,8,8,4,3,5,4,9]`

With Bubble sort, ascending:

```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
ba 10 4.990399975213222e-05
```

(Unduplicated list)

```
[3, 4, 4, 5, 6, 8, 8, 9, 9, 9]
[3, 4, 4, 5, 6, 8, 8, 9, 9, 9]
[3, 4, 4, 5, 6, 8, 8, 9, 9, 9]
ba 10 5.167600102140568e-05
```

(Duplicated list)

➔ take more time to sort duplicate list.

With Merge sort, ascending:

```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
ma 10 0.00012886699914815836
```

(Unduplicated list)

```
[3, 4, 4, 5, 6, 8, 8, 9, 9, 9]
[3, 4, 4, 5, 6, 8, 8, 9, 9, 9]
[3, 4, 4, 5, 6, 8, 8, 9, 9, 9]
ma 10 0.00023292749938264024
```

(Duplicated list)

➔ take more time to sort duplicate list.

D. Convert the following iterative pseudo code (Fig. 1) to the recursive python code. Your code must include wrapper method and exception handling. **(5 marks) [clearly mention the base case in comments and provide basic test harness (simple main function in the same file)]**

```
Function revrse_str
Imports: Str1
Exports: Reverse Str1
for x =size (Str1)-1 to 0
      print (Str1(i))
End for
End
```

Fig. 1

Python file: **Q2PartD.py**

Question 3: Stacks, Queue and Lists (Total Marks: 15)

A. Implement a function "Transfer (S1,S2, S3)" where S1, S2 and S3are three stacks. The function transfers the data from S1 to S2 and S2 to S3, at the end print S1, S2 and S3 are printed to show change in the order of the elements **(5 marks) [You can use numpy arrays or linked list for this implementation]**

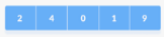
Python file: **Q3PartA.py**

B. Implement a circular queue (max size 10 elements) with the help of double ended single linked list, try to insert 12 random names of fruits in the queue, however the queue should be able to throw appropriate error messages/exceptions if the queue is full or empty. **(5 marks)**

Python file: **Q3PartB.py**

C. Discuss in detail the selection of double ended double linked list compared to double ended single linked list, single ended single linked list and array for the implementation of circular queue (Part B). Discuss in detail that which data structure can be used for this implementation, effective on time complexity and memory utilisation Consider the Big O notation (asymptotic) time complexity and memory utilisation to justify your answer. **(5 marks) [can be represented in tabular form]**

- **Discuss in detail the selection of 4 data structures in implementing circular queue**

Data structures	Double Ended Double Linked List: 	Double Ended Single Linked List: 	Single Ended Single Linked List: 	Array 
Advantages	The head and tail are not connected ➔ don't need extra effort to implement it in circular queue	The head and tail are not connected ➔ don't need extra effort to implement it in circular queue	-	Don't require extra space for connection
Disadvantages	Requires extra computer spaces to store the backwards connection (compared to DESLL)		The head and tail do not connected ➔ extra effort to fix it in circular queue Require extra computer spaces to store the forwards connection (compare to array)	Difficult to insert more items out of array range's size ➔ because: array has fixed size ➔ Require more logics to make array behave like Circular Queue.

- Choosing suitable data structure for implementing Circular Queue

  It also depends on which specific requirements of creating Circular Queue.

  On the one hand, I would go for **Array** if memory is pivotal important:
  - Memory Utilization: array has fixed memory, so user can only work on the exact amount of memory allocated previously.
  - Time complexity:
    Enqueue (insert): O(N)
    Dequeue (delete): O(N)
    ➔ memory is strength

  On the other hand, I will choose **Double Ended Single Linked List** if memory processing not a big deal:
  - It already behaves like a circular queue.
  - Memory Utilization: consumes extra space and CPU resources to store connection between nodes.
  - Time complexity:
    Enqueue (insert from head): O(1)
    Dequeue (delete from head): O(1)
    ➔ running time is strength

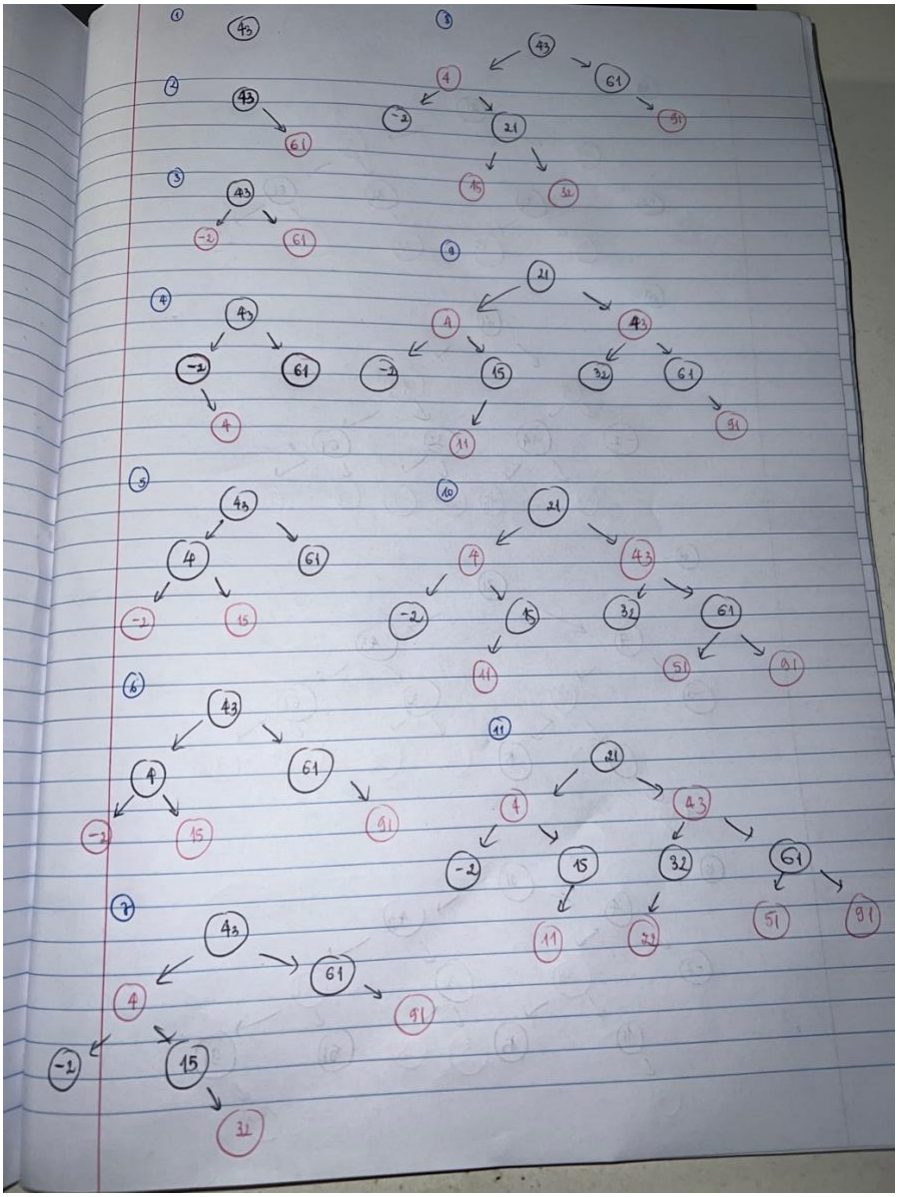
Question 4: Trees (Total Marks: 16)

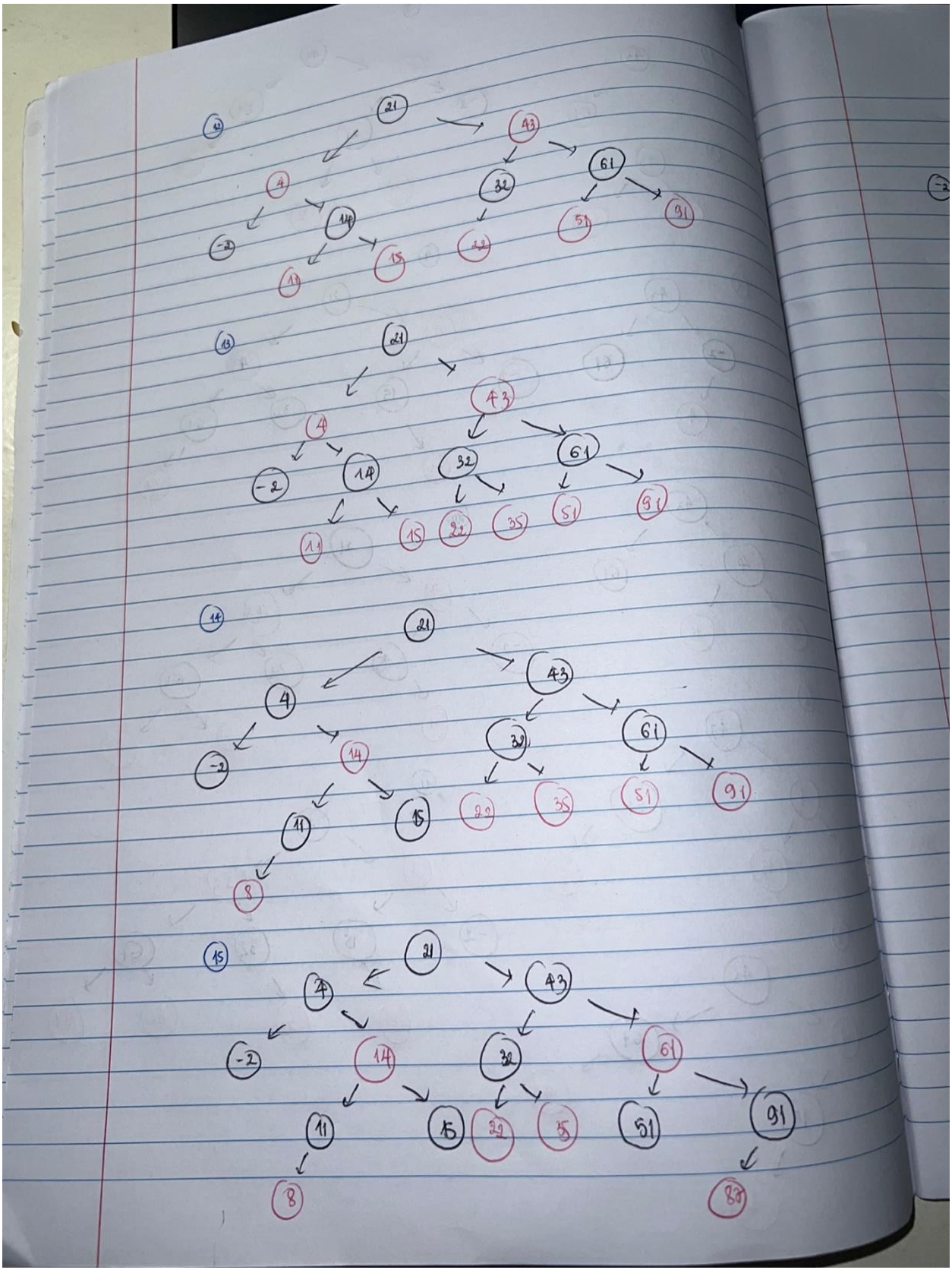
A. Considered the list/array shown in Fig. 2 and insert the elements in a Red Black Tree, show the developed tree. Also, draw the Red Black tree and BST on the reverse sorted data. Now compare the developed tree (all three) describe what is the effect of sorted or partially sorted data on the trees. Which is the best tree if compared on completeness. **(4 marks) [show steps for each element insertion]**

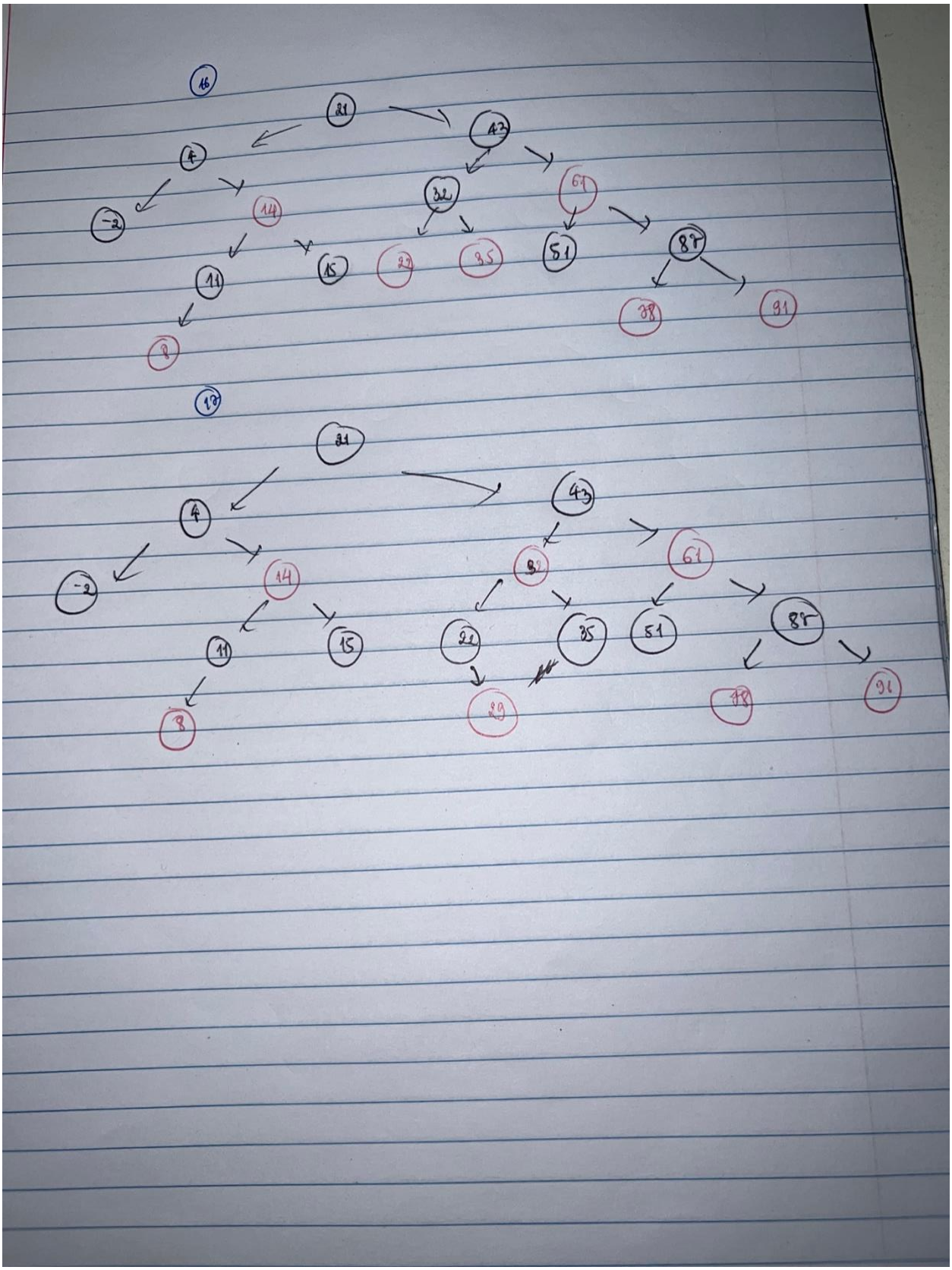
43	61	-2	4	15	91	32	21	11	51	22	14	35	8	87	78	29

Fig. 2

Red Black Tree with Fig.2



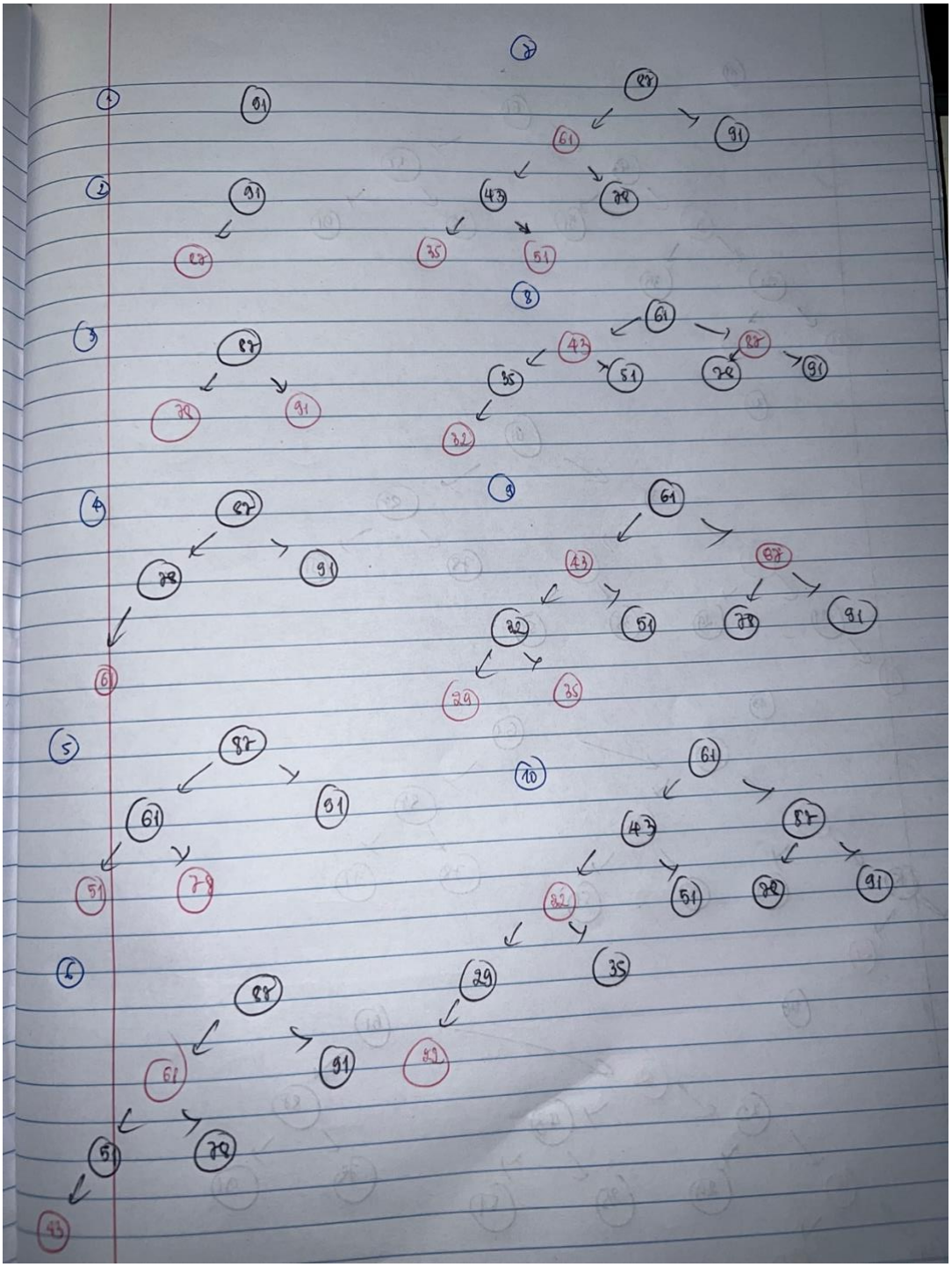

21
43
4
32
61
-2
14
51
91
11
15
21
43
4
-2
14
32
61
11
15
22
35
51
91
21
43
4
-2
14
32
61
11
15
22
35
51
91
8
21
43
4
-2
14
32
61
11
15
22
35
51
91
8
88

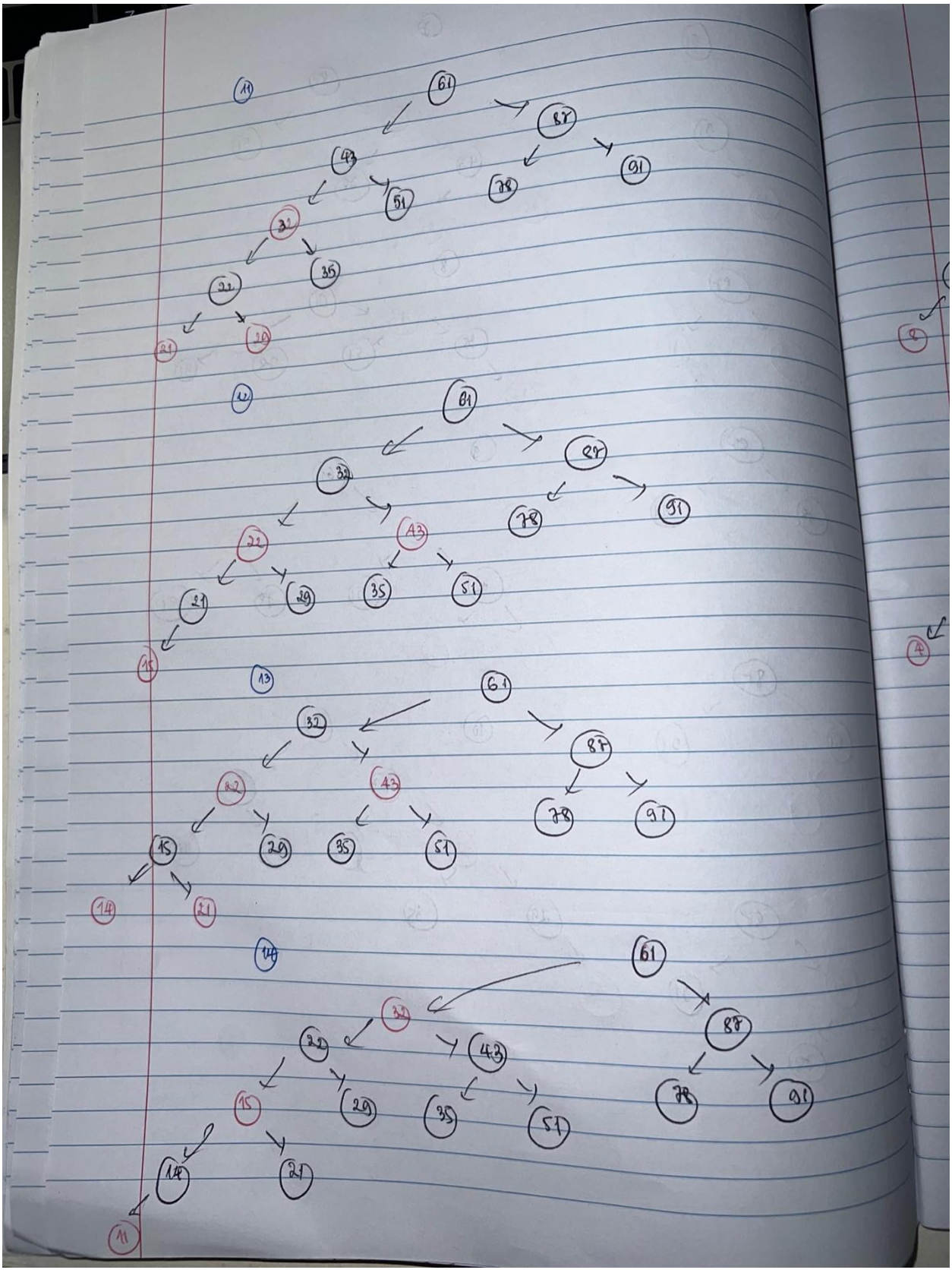


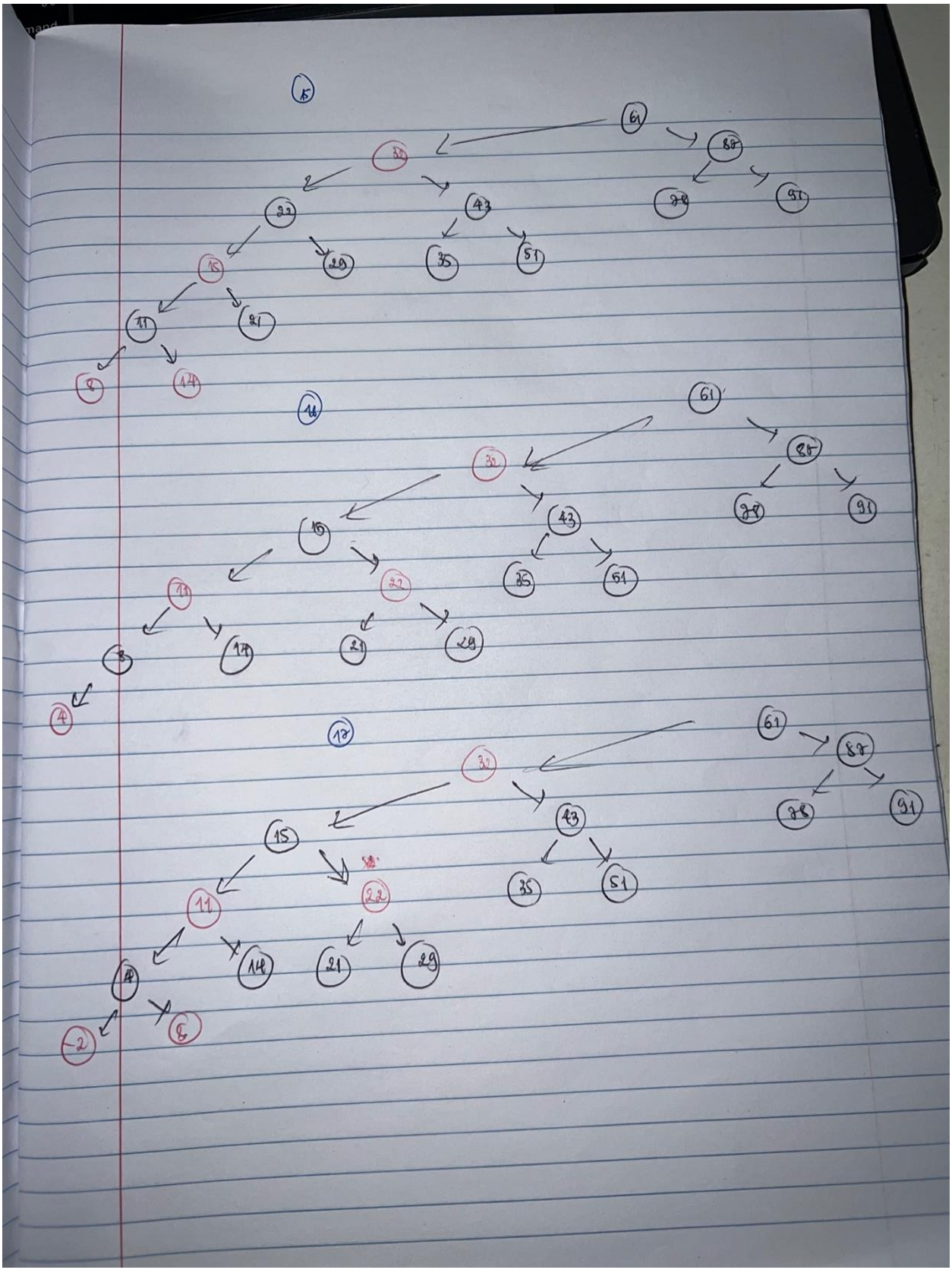
Reverse Sorted Data (sort by Bubble Sort – DSAsorts.py week 1):

```
ccadmin@CCUbuntu64bit:~/DSA1002/Prac09$ python3 DSAsorts.py
[91, 87, 78, 61, 51, 43, 35, 32, 29, 22, 21, 15, 14, 11, 8, 4, -2]
ccadmin@CCUbuntu64bit:~/DSA1002/Prac09$
```

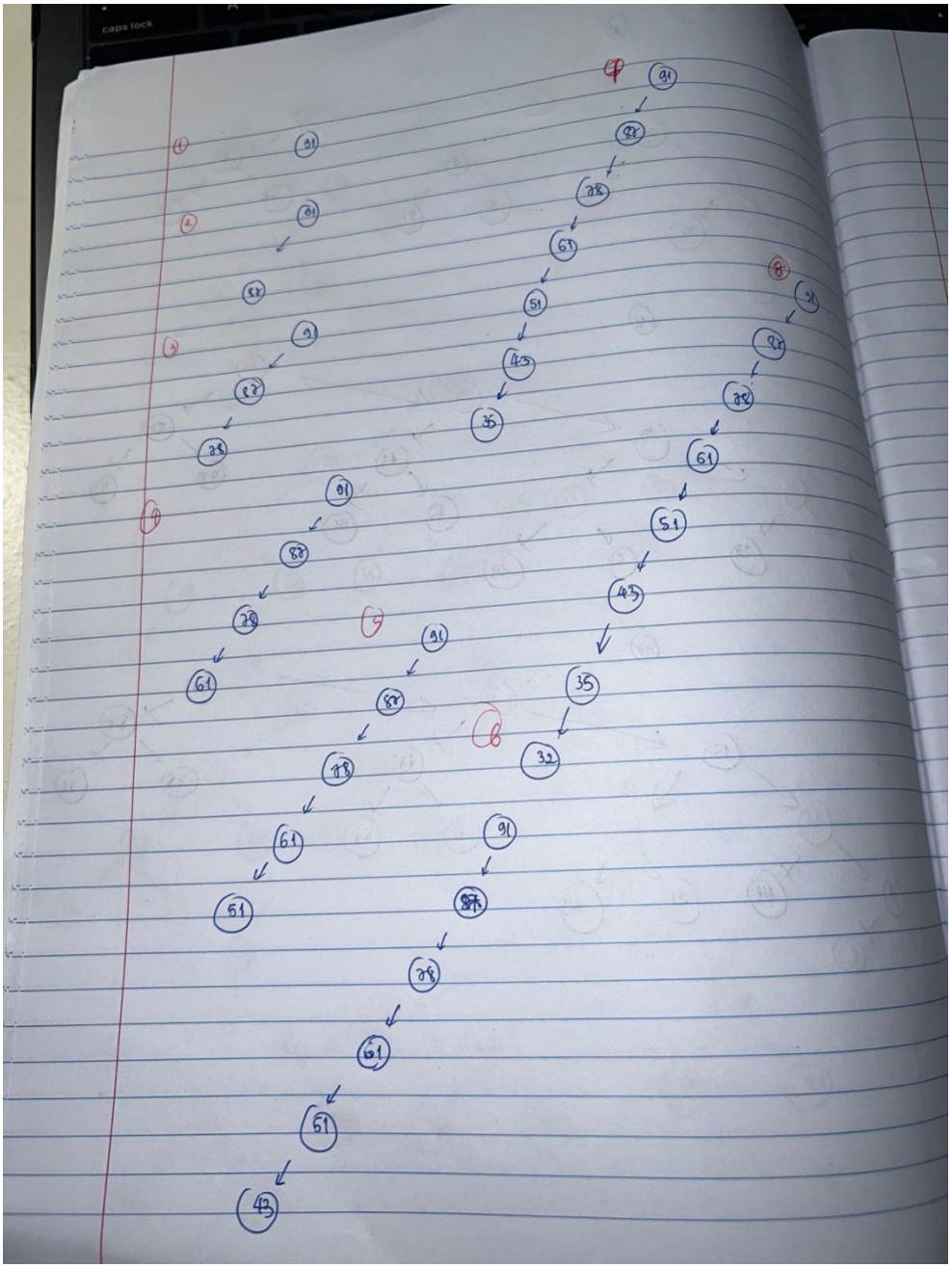
Red Black Tree with Fig.2 reverse sorted data

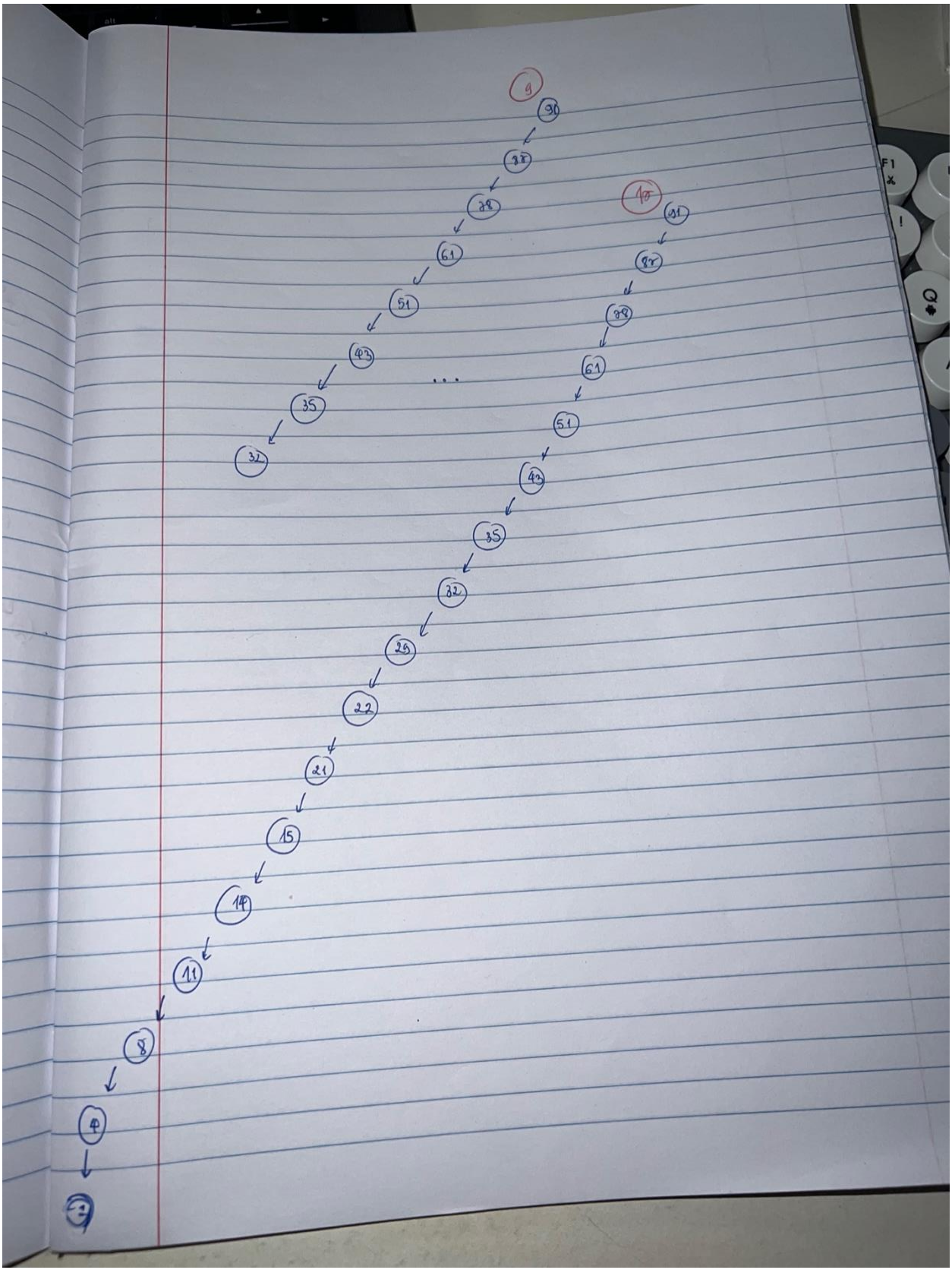






Binary Search Tree with Fig.2 reverse sorted data




9
30
38
28
61
51
43
35
32
...
91
88
61
51
43
35
32
25
22
21
15
14
11
8
4
2

Compare all 3 developed trees:

Character	Red Black Tree with Fig.2	Red Black Tree with Fig.2 reverse sorted data	Binary Search Tree with Fig.2 reverse sorted data
Tree shape:	More balanced in both sides	Skew to the left-hand side	Completely go to left side ➔ become a single linked list

How sorted and partially sorted data have effect on the trees?
➔ From my observation, the more data is sorted, the lower efficiency of tree search.
Specifically:

They all contain N items, but for:

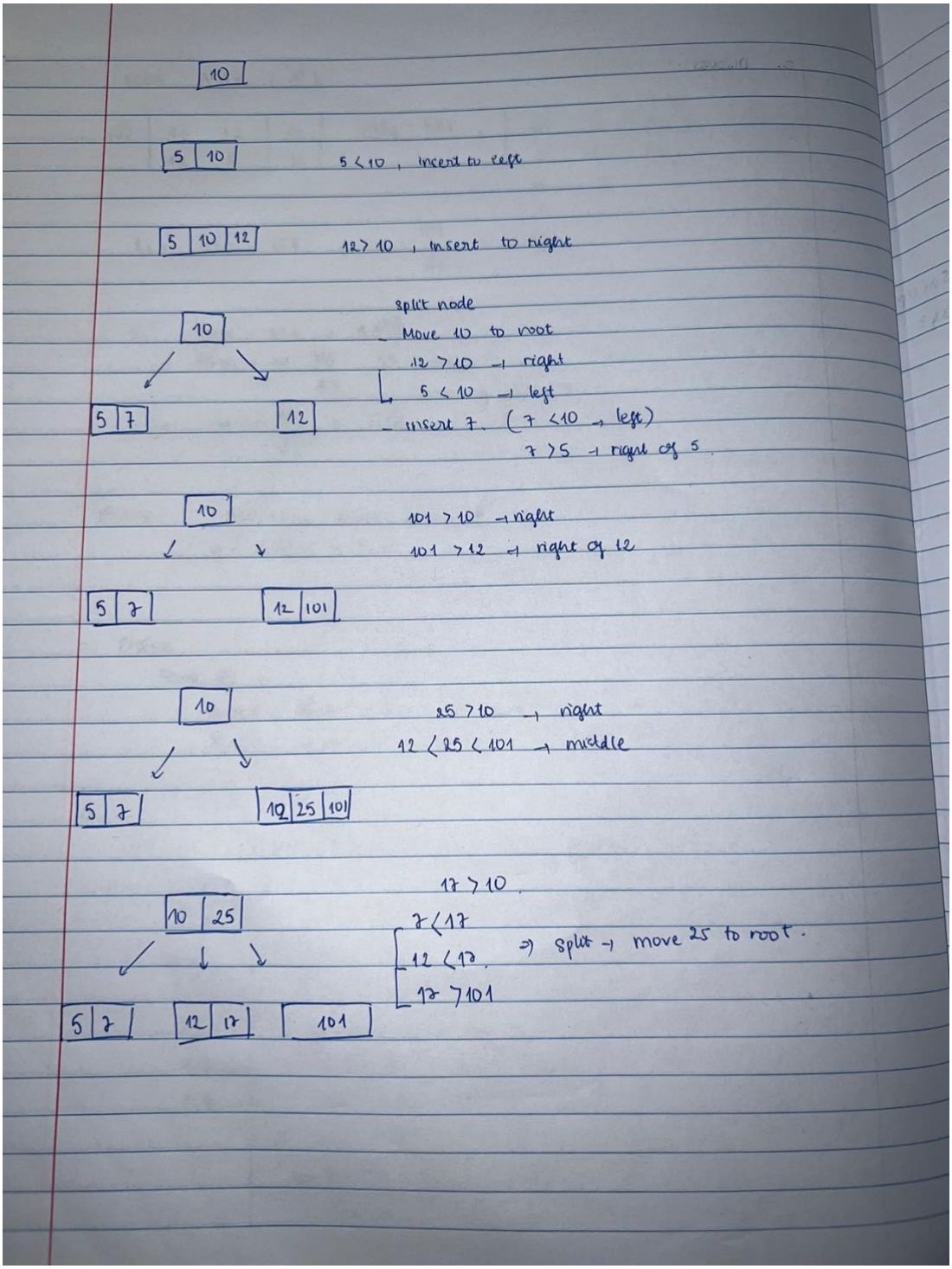
- Red Black Tree with Fig.2: has depth of log2N ➔ time complexity for searching is O(logN) → best case
- Red Black Tree with Fig.2 reverse sorted data: still has searching time is O(NlogN).
- Binary Search Tree with Fig.2 reverse sorted data: has depth of N ➔ searching time would be O(N) → worst case

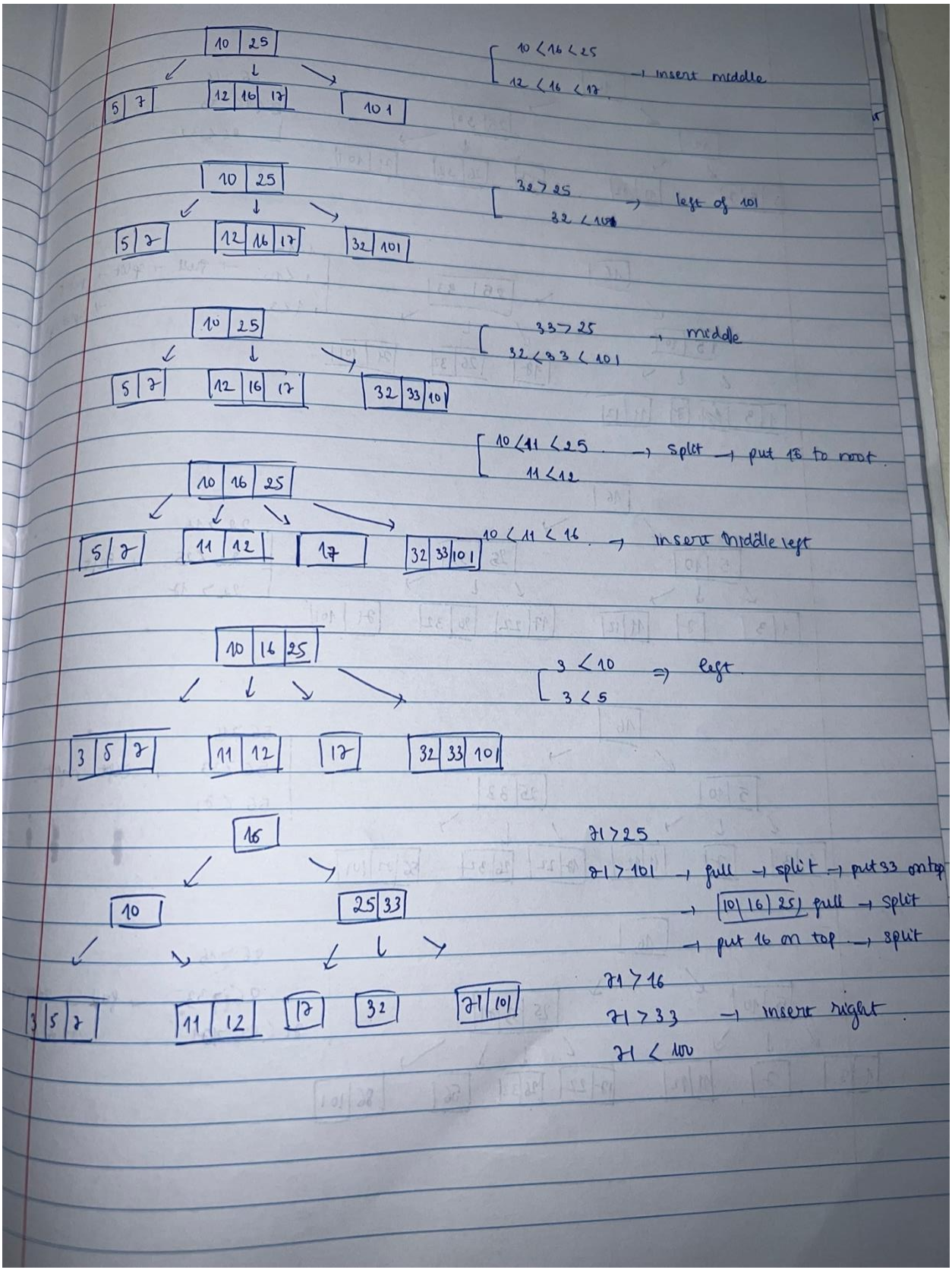
➔ Base on the afore-mentioned explanation, I think Red Black Tree is the best tree to perform tasks rather than Binary Search Tree. This is simply because, even though the data is either sorted or unsorted, it can deal with them all and ensure O(logN) searching time.

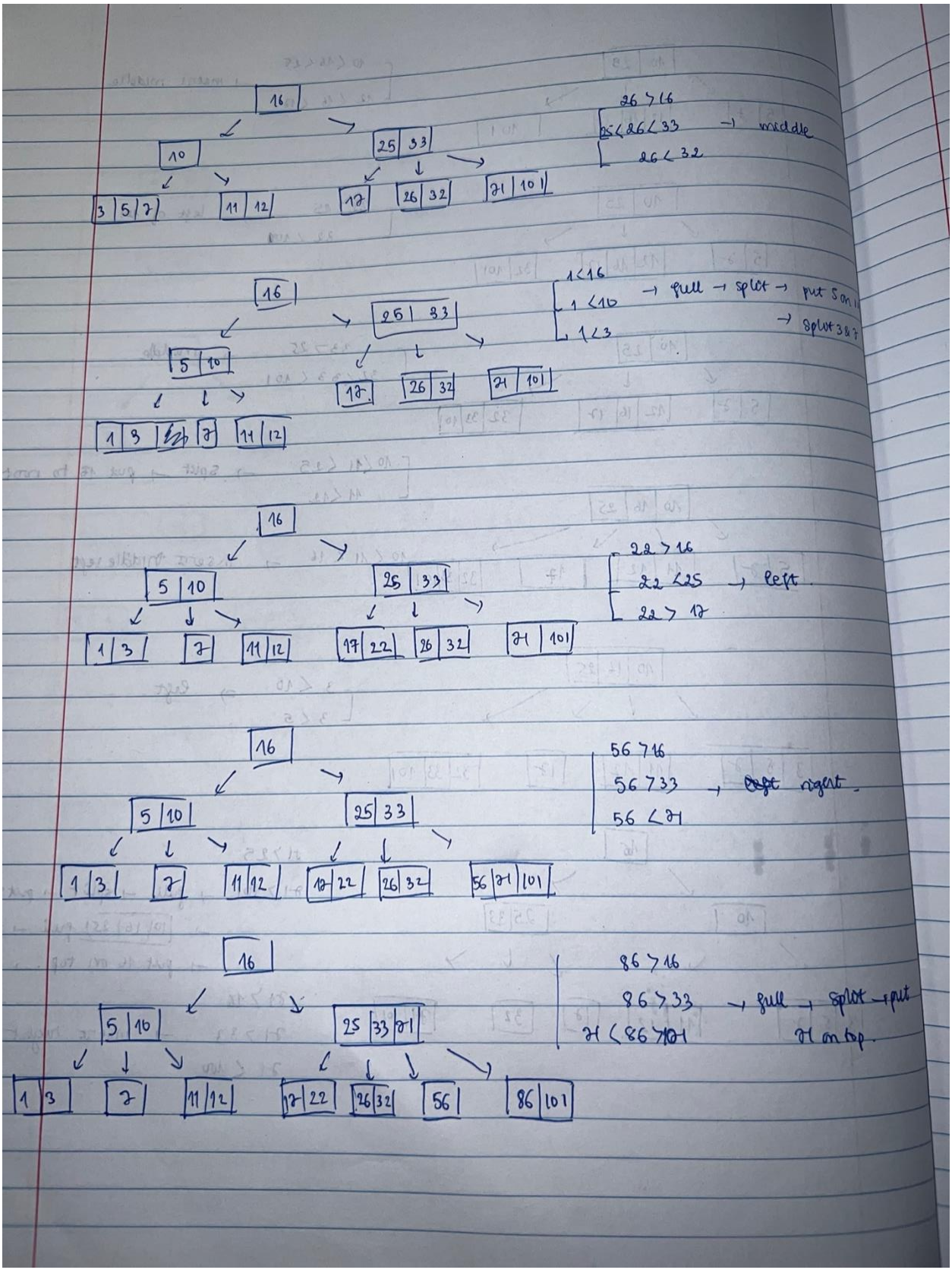
B. Consider the elements presented in Fig. 3 and draw a 2-3-4 Tree form initially empty root. Show and describe each step. **(3 marks)**

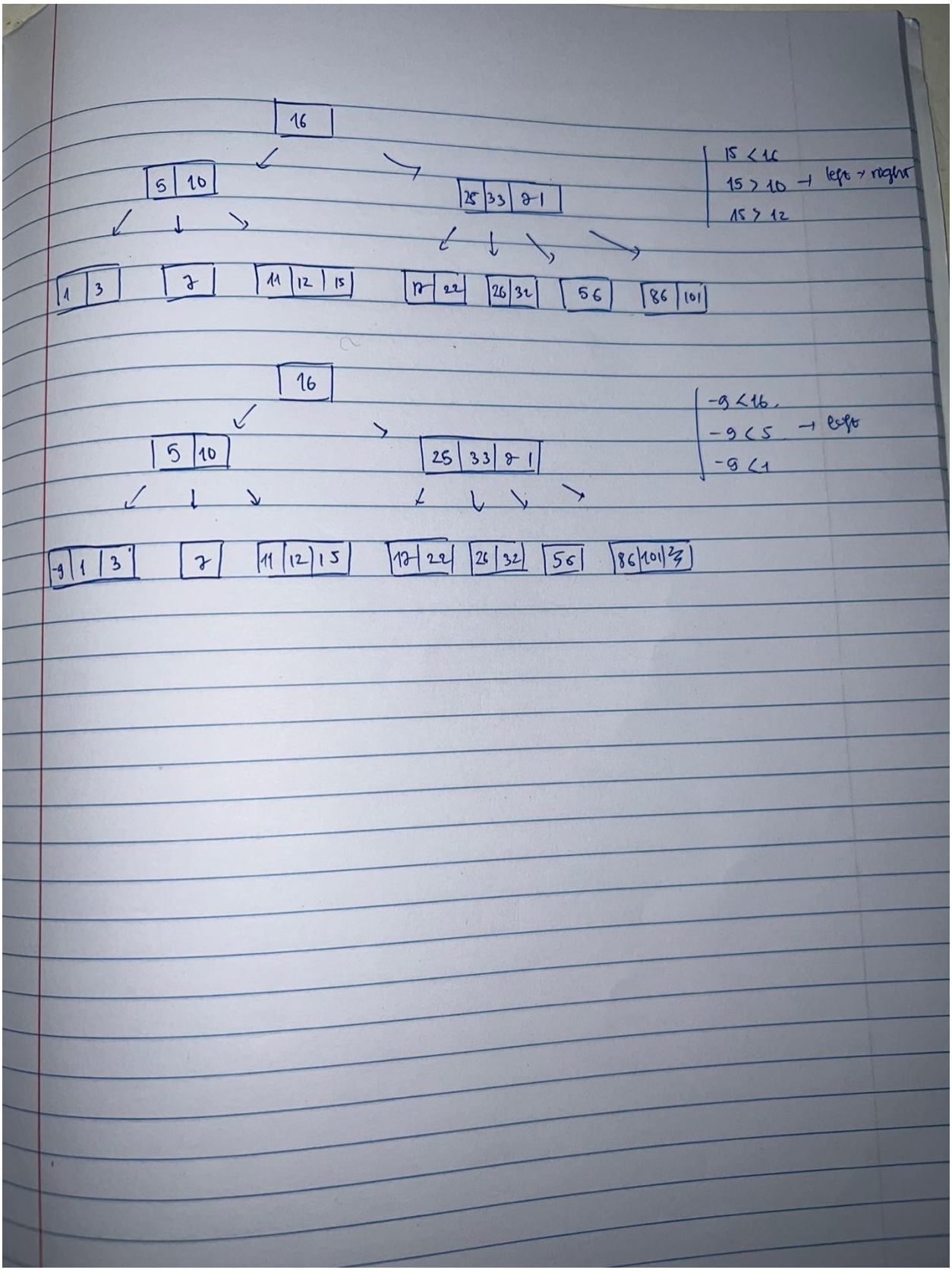
10	5	12	7	101	25	17	16	32	33	11	3	71	26	1	22	56	86	15	-9

Fig. 3

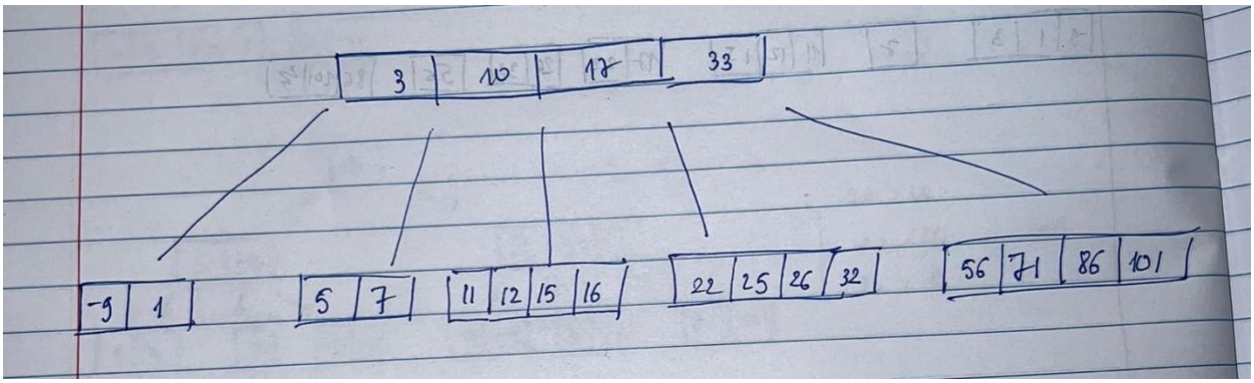



10 | 25
5 | 7
12 | 16 | 17
101
10 < 16 < 25
12 < 16 < 17
→ insert middle
10 | 25
5 | 7
12 | 16 | 17
32 | 101
32 > 25
32 < 101
→ left of 101
10 | 25
5 | 7
12 | 16 | 17
32 | 33 | 101
33 > 25
32 < 33 < 101
→ middle
10 | 16 | 25
5 | 7
11 | 12
17
32 | 33 | 101
10 < 11 < 25
11 < 12
→ split → put 16 to root
10 < 11 < 16
→ insert middle left
10 | 16 | 25
3 | 5 | 7
11 | 12
17
32 | 33 | 101
3 < 10
3 < 5
⇒ left
16
10
25 | 33
3 | 5 | 7
11 | 12
17
32
71 | 101
71 > 25
71 > 101 → full → split → put 33 on top
→ 10 | 16 | 25 full → split
→ put 16 on top → split
71 > 16
71 > 33 → insert right
71 < 100


16
10
25 33
3 5 7
11 12
17
26 32
71 101
26 > 16
25 < 26 < 33 → middle
26 < 32
16
5 10
25 33
1 3
7
11 12
17
26 32
71 101
1 < 16
1 < 10 → full → split → put 5 on
1 < 5 → split 3 & 7
16
5 10
25 33
1 3
7
11 12
17 22
26 32
71 101
22 > 16
22 < 25 → left
22 > 17
16
5 10
25 33
1 3
7
11 12
17 22
26 32
56 71 101
56 > 16
56 > 33 → right
56 < 71
16
5 10
25 33 71
1 3
7
11 12
17 22
26 32
56
86 101
86 > 16
86 > 33 → full → split → put
71 < 86 < 101
71 on top.


16
5 10
25 33 81
1 3
7
11 12 15
17 22
26 32
56
86 101
15 < 16
15 > 10 → left → right
15 > 12
16
5 10
25 33 81
-9 1 3
7
11 12 15
17 22
26 32
56
86 101
-9 < 16
-9 < 5 → left
-9 < 1

C. Convert the tree drawn in part B to a B Tree (degree 5), also discuss that which tree is more efficient for inserting and searching elements. Consider the Big O notation (asymptotic) time complexity to support you answer. **(4 marks)**



Time complexity	Binary Search Trees	Red Black Trees	2-3-4 Trees	B Trees (5 degrees)
Insert	O(N) – have to traverse all elements Easy to insert	O(logN) More complicated	O(logN) More complicated	O(logN) More complicated
Searching	O(logN) But O(N) – when have to traverse all elements in worst case with sorted data	O(logN)	O(logN)	O(logN)

Look at the time complexity table above, we can easily ignore BST because it takes more time in inserting and searching than the others.

For me, Red Black Trees, 2-4 Trees and B-Trees are the same in do inserting and searching. Because the advanced trees are self-balancing, the work required to insert, find and delete are significantly reduced. Furthermore, B-Tree also save me memory for the pointer and connectivity because it has multiple degree in a node to use.

However, I will consider to use B-Tree for the case if the data we work with is much more than what memory can handle searching as it saves on lag produced by disk rotation.

D. Consider the following UML diagrams (Fig. 4) for the BSTree class and BSTNode class. You are supposed to write a method to print all entries divisible by 4 and 5 in a sorted order without using any explicit sorting algorithm (i.e., any sorting algorithm). Write a test harness to check the functionality of the code. **(5 marks)**

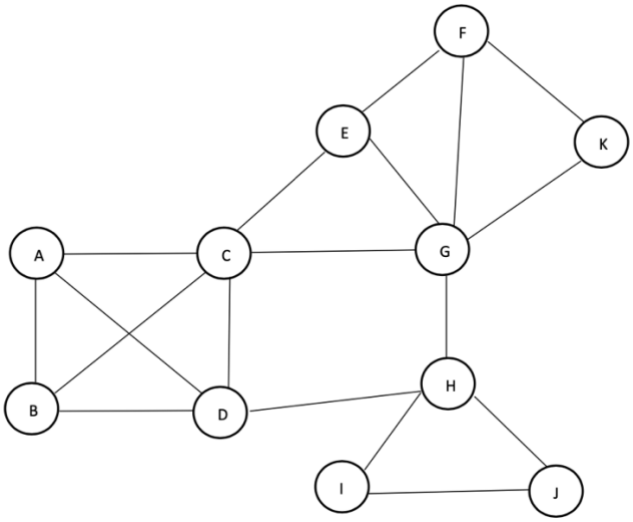
<<BSTNode>>
-Key: None -Value: None -Left_child: None -Right_child: None
+init(self, key, value)

<<BSTree>>
-Root: None
+init(self, root) +sorted_4_5_num(self)

Python file: **Q4PartD_Node.py, Q4PartD_Tree.py, Q4PartD_testHarness.py**

Question 5: Graphs (Total Marks: 12)

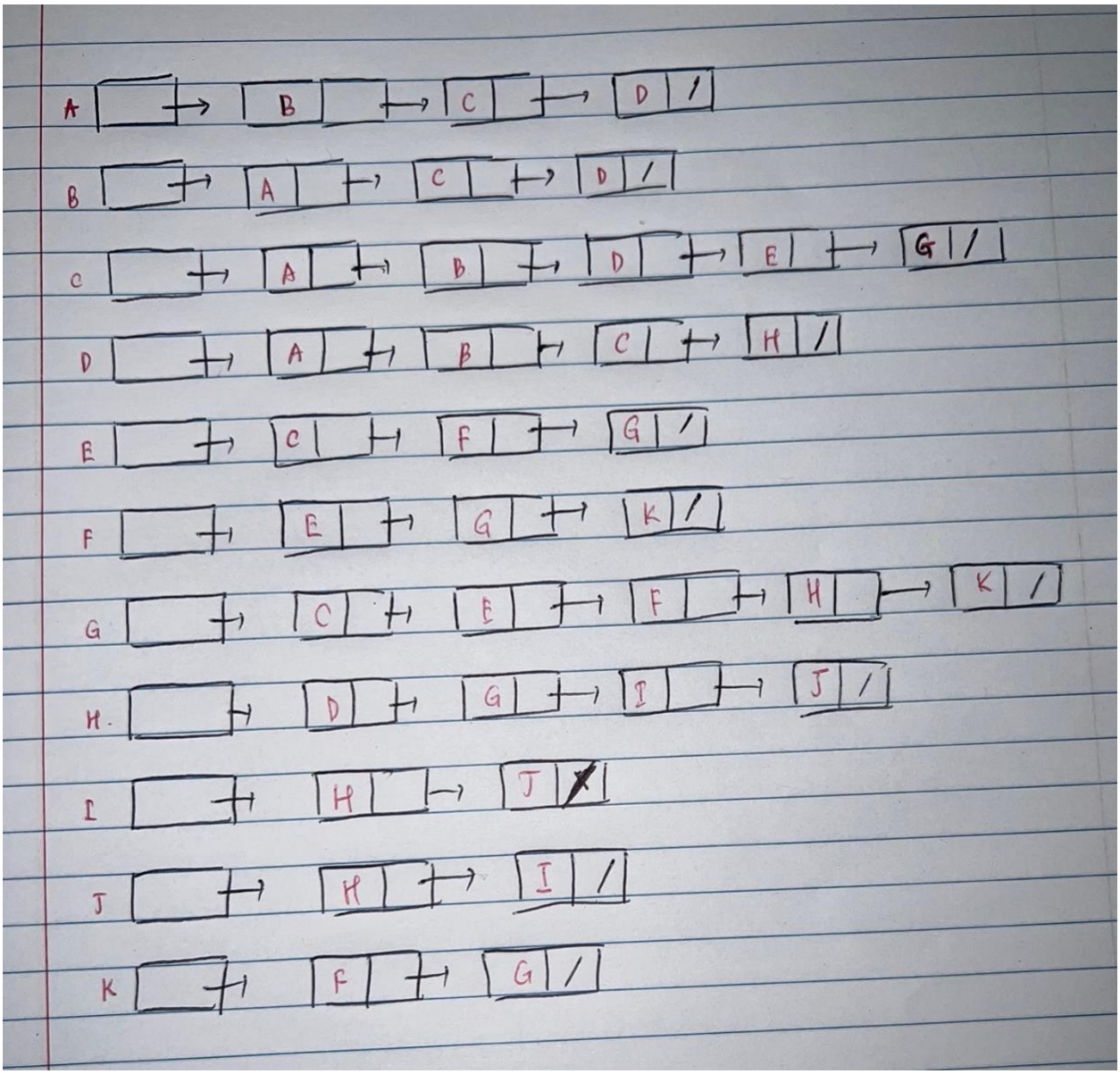
A. Represent the following graph in Adjacency Matrix and Adjacency List (for each vertex) representation. **(4 marks)**



Adjacency Matrix:

	A	B	C	D	E	F	G	H	I	K	J
A	0	1	1	1	0	0	0	0	0	0	0
B	1	0	1	1	0	0	0	0	0	0	0
C	1	1	0	1	1	0	1	0	0	0	0
D	1	1	1	0	0	0	0	1	0	0	0
E	0	0	1	0	0	1	1	0	0	0	0
F	0	0	0	0	1	0	1	0	0	1	0
G	0	0	1	0	1	1	0	1	0	1	0
H	0	0	0	1	0	0	1	0	1	0	1
I	0	0	0	0	0	0	0	1	0	0	1
K	0	0	0	0	0	1	1	0	0	0	0
J	0	0	0	0	0	0	0	1	1	I0	0

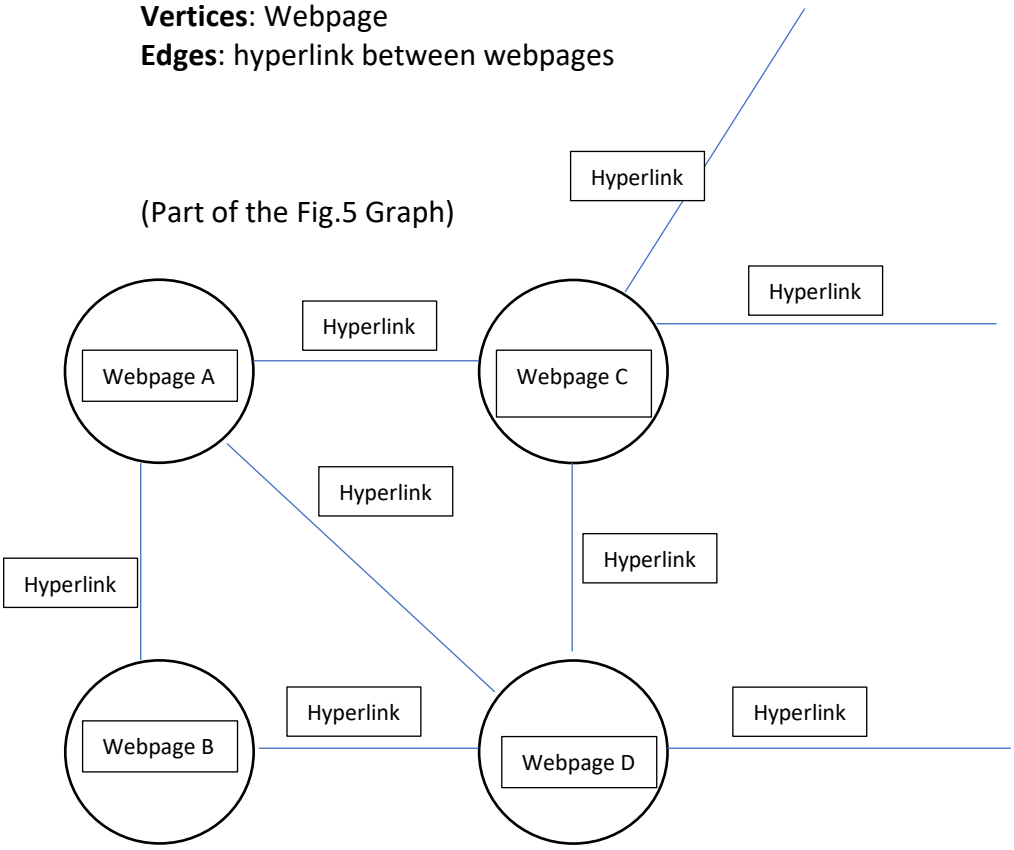
Adjacency List:



B. If this graph is used to represent World Wide Web (WWW), considering this application, describe that what data should be stored in vertices (smallest entity is each webpage) and edges respectively. **(2 marks)**

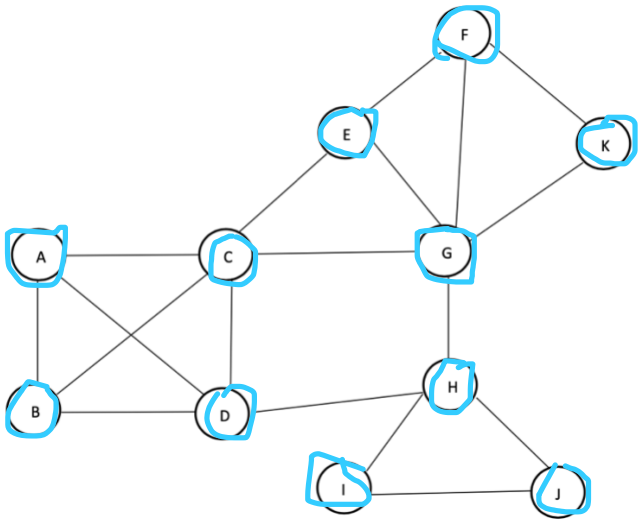
Vertices: Webpage
Edges: hyperlink between webpages

(Part of the Fig.5 Graph)



This is undirected graph. In this case, each node contains webpage and the edges will be the hyperlink (super link in the Internet: it can be URL as well) which points to the webpages. Between each webpage will be connect by its own link, therefore the user can easily move from webpages to webpages without the restriction of directions (like in directed graph) ➔ as everything is included and connected

C. Described the graphical steps for the depth first search along with value stored in stack/queue. Consider vertex "G" as the starting point. **(3 marks)**



We start visiting G first, add adjacent node of G to the stack

Visited

G	C	A	B	D	H	I	J	E	F	K

Stack

G										

➔ G has C, H, K, E, F is unvisited adjacent node. According to alphabetical order, we visit C first
→ add C to the stack

Stack

C	G									

C has A, B, D, E is unvisited adjacent node →visit A(alphabetical order) → add A to stack

Stack

A	C	G								

A has B, D is unvisited adjacent node →visit B(alphabetical order) → add B to stack
Stack

B	A	C	G							

B has only D is unvisited adjacent node →visit D → add D to stack
Stack

D	B	A	C	G						

D has only H is unvisited adjacent node →visit H → add H to stack
Stack

H	D	B	A	C	G					

H has I, J is unvisited adjacent node →visit I first → add I to stack
Stack

I	H	D	B	A	C	G				

I has only J is unvisited adjacent node →visit J → add J to stack
Stack

J	I	H	D	B	A	C	G			

Now, J is on top of stack does not have any unvisited adjacent node, remove J from Stack
Then, I is on top of stack does not have any unvisited adjacent node, remove I from Stack
Then, H is on top of stack does not have any unvisited adjacent node, remove H from Stack
Then, D is on top of stack does not have any unvisited adjacent node, remove D from Stack
Then, B is on top of stack does not have any unvisited adjacent node, remove B from Stack
Similarly, A is on top of stack does not have any unvisited adjacent node, remove A from Stack

Stack

C	G									

Now C is on top of Stack and has only E is unvisited adjacent node → visit E → add E to stack
Stack

E	C	G								

E is on top of Stack and has only F is unvisited adjacent node → visit F → add F to stack

Stack

F	E	C	G							

F is on top of Stack and has only K is unvisited adjacent node → visit K → add K to stack

Stack

K	F	E	C	G						

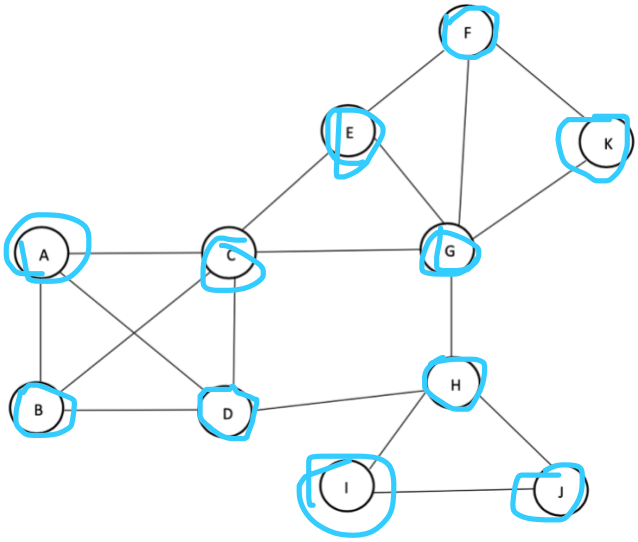
Now, K is on top of stack does not have any unvisited adjacent node, remove K from Stack
Now, F is on top of stack does not have any unvisited adjacent node, remove F from Stack
Now, E is on top of stack does not have any unvisited adjacent node, remove E from Stack
Now, C is on top of stack does not have any unvisited adjacent node, remove C from Stack
Now, G is on top of stack does not have any unvisited adjacent node, remove G from Stack
➔ Stack is Empty ➔ Finish

Stack

Output

G	C	A	B	D	H	I	J	E	F	K

D. Described the graphical steps for the breadth first search along with value stored in stack/queue. Consider vertex "D" as the starting point. **(3 marks)**



Visited D first → add unvisited adjacent node of D to Queue and mark them as visited

Visited

D	A	B	C	H	E	G	I	J	F	K

Queue

A	B	C	H							

A is in top of the queue, currently do not have any unvisited adjacent node → remove A from queue

Similarly to B → remove B

Queue

C	H									

C have E & G is unvisited adjacent node → mark them as visited and add to queue

Queue

C	H	E	G							

C now does not have any unvisited adjacent node → remove C

Queue

H	E	G								

H have I & J is unvisited adjacent node → mark them as visited and add to queue

Queue

H	E	G	I	J						

H now does not have any unvisited adjacent node → remove H

Queue

E	G	I	J							

E just has F is unvisited adjacent node → mark F as visited and add to queue

Queue

E	G	I	J	F						

E now does not have any unvisited adjacent node → remove E

Queue

G	I	J	F							

G just has K is unvisited adjacent node → mark K as visited and add to queue

Queue

G	I	J	F	K						

G, I, J, F, K now does not have any unvisited adjacent node → remove them all

➔ Queue is empty ➔ Finish

Queue

Output

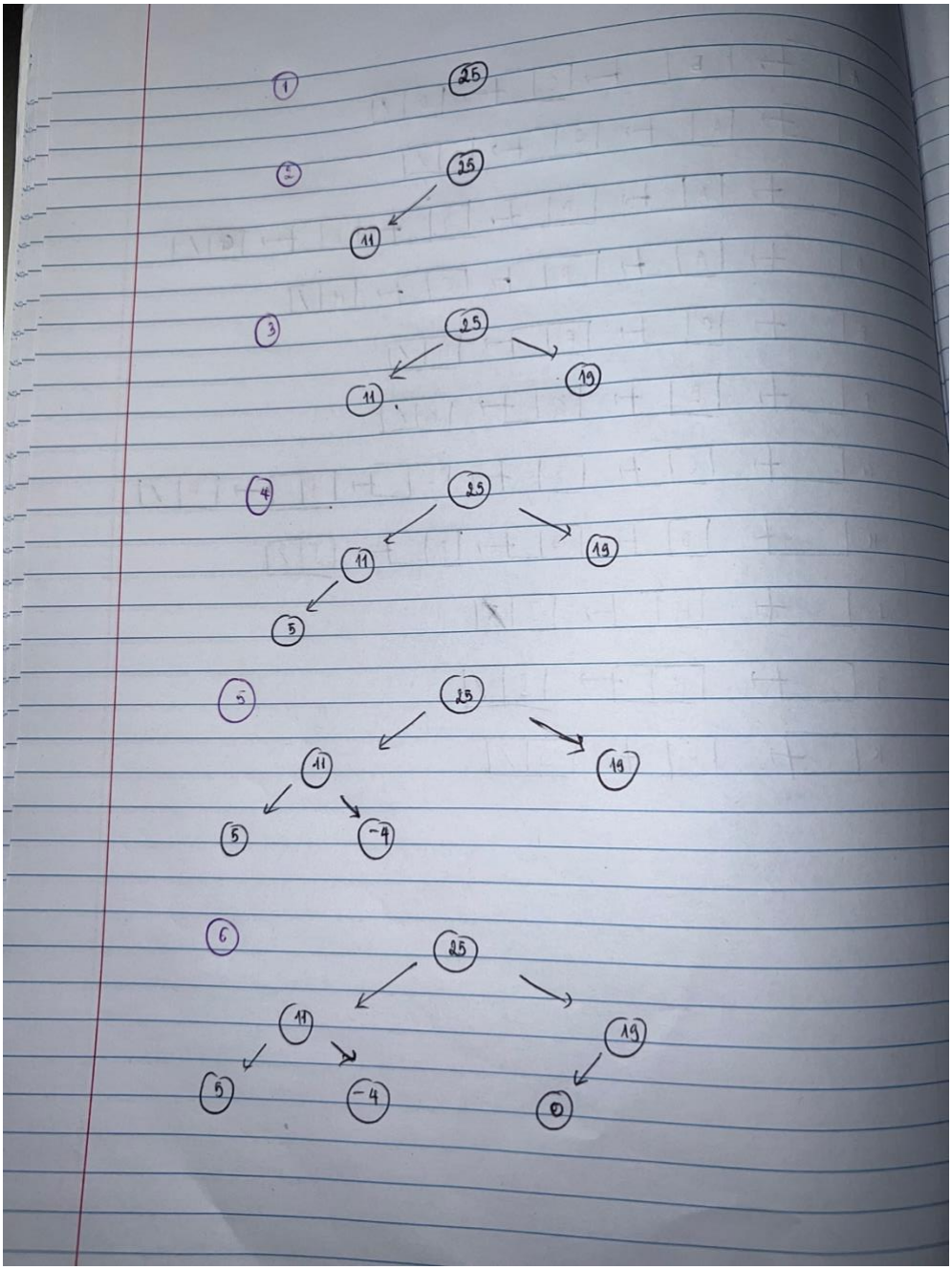
D	A	B	C	H	E	G	I	J	F	K

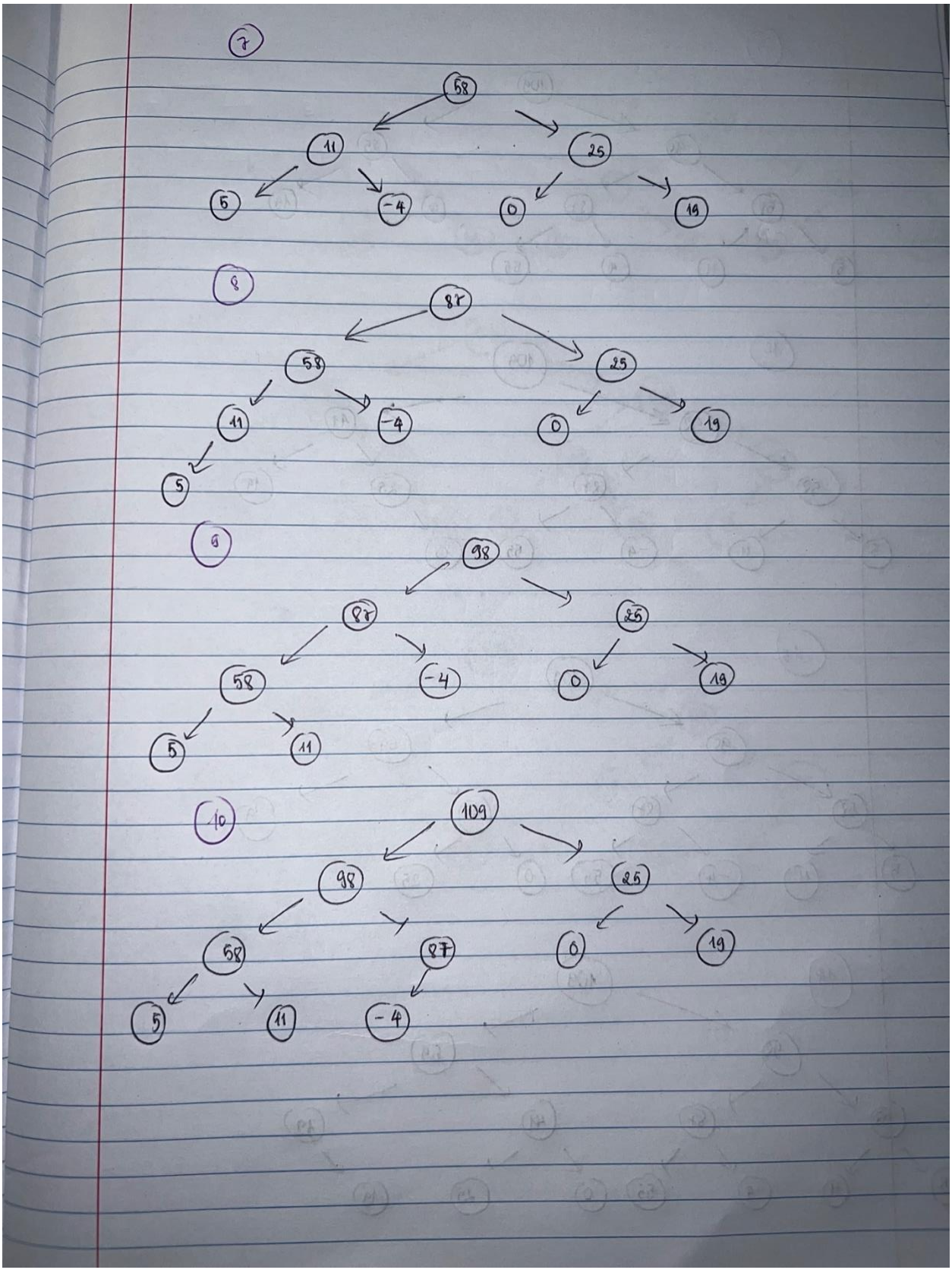
Question 6: Heaps (Total Marls: 13)

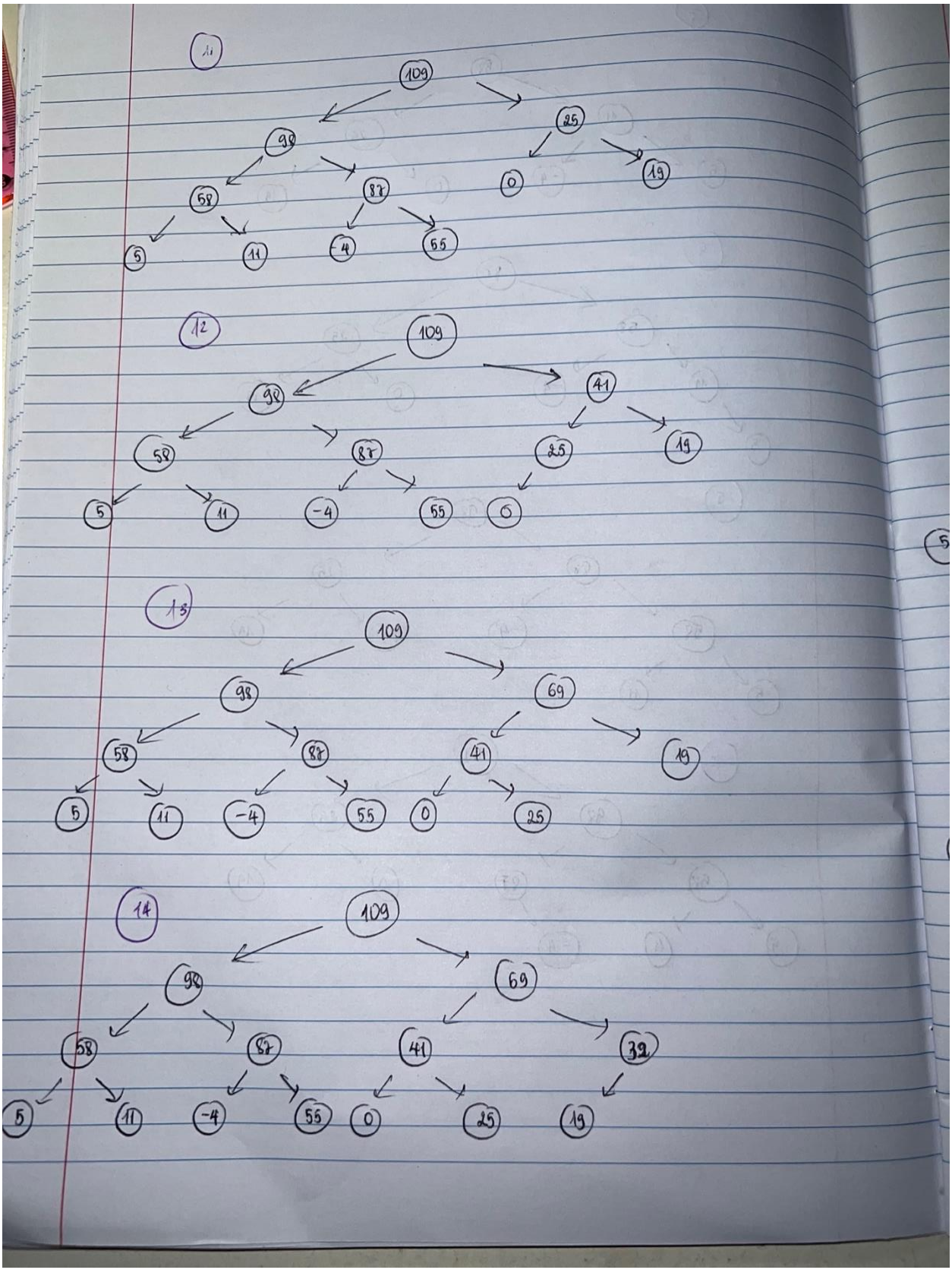
A. Draw a max heap using the number list, show you working as graphical steps for each insert, representing the step by step building of the heap. **(4 marks)**

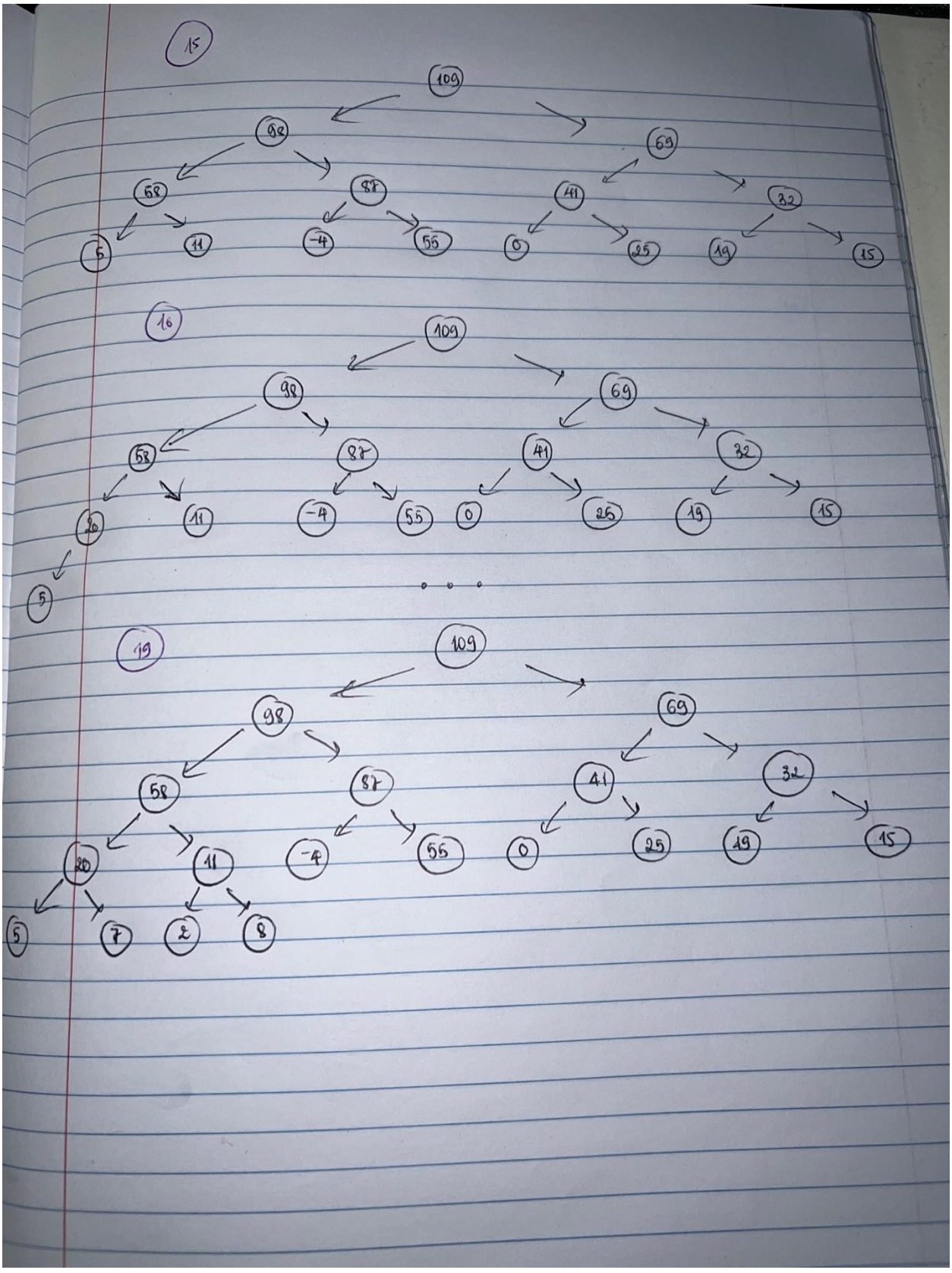
25	11	19	5	-4	0	58	87	98	109	55	41	69	32	15	20	7	2	8

Fig. 6




7
58
11
25
5
-4
0
19
8
87
58
25
11
-4
0
19
5
9
98
87
25
58
-4
0
19
5
11
10
109
98
25
58
87
0
19
5
11
-4


11
109
98
25
58
87
0
19
5
11
-4
55
12
109
98
41
58
87
25
19
5
11
-4
55
0
13
109
98
69
58
87
41
19
5
11
-4
55
0
25
14
109
98
69
58
87
41
32
5
11
-4
55
0
25
19


15
109
98
69
58
87
41
32
5
11
-4
55
0
25
19
15
16
109
98
69
58
87
41
32
20
11
-4
55
0
25
19
15
5
19
109
98
69
58
87
41
32
20
11
-4
55
0
25
19
15
5
7
2
8

B. Show an array-based representation of the number list as heap, satisfying the related arithmetic operations for traversing. **(2 marks)**

Array – based representation:

(Read the array from top to bottom from left to right)

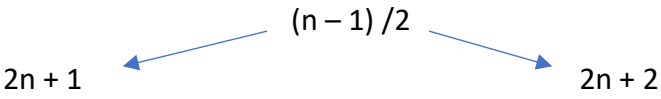
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
109	98	69	58	87	41	32	20	11	-4	55	0	25	19	15	5	7	2	8

Demonstrating the arithmetic operations for traversing:

In the lecture slide, we have:

```
leftChildIdx = (currIdx * 2) + 1
rightChildIdx = (currIdx * 2) + 2
parentIdx = (currIdx – 1) / 2
```

I can write like this:



With:

- n: current index
- (n -1)/2 : parent node
- 2n + 1: left child
- 2n + 2: right child

For example:

- Find the children of element 87 (index = 4):

➔ Left child = 2n + 1 = 2 x 4 + 1 = 9 → element -4 (index = 9 compared with array – based representation and heap visualization) √ True

➔ Right child = 2n + 2 = 2 x 4 + 2 = 10 → element 55 (index = 10 compared with array – based representation and heap visualization) √ True

- Find the parent of element 19 (index = 13):

➔ Parent node = (n-1)/2 = (13-1)/2 = 6 → element 32 (index = 10 compared with array – based representation and heap visualization) √ True

Therefore, the arithmetic operations for traversing is correct !

C. Show the process for deleting/removing 2 values in the tree built in part A. Show steps (including each step of the trickle-down) on the array-based representation built in part B. **(4 marks)**

Since the heap is a way of implementing a priority queue and the heap is organized such that the highest priority item is always the root node
➔ we will always remove the root node

REMOVE 1:

Step 1: remove root node 109

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
	98	69	58	87	41	32	20	11	-4	55	0	25	19	15	5	7	2	8

Step 2: take the node has smallest index to replace the root

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
8	98	69	58	87	41	32	20	11	-4	55	0	25	19	15	5	7	2	

Step 3: trickle-down, swap value of index 0 with the highest value (index 1)

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
98	8	69	58	87	41	32	20	11	-4	55	0	25	19	15	5	7	2	

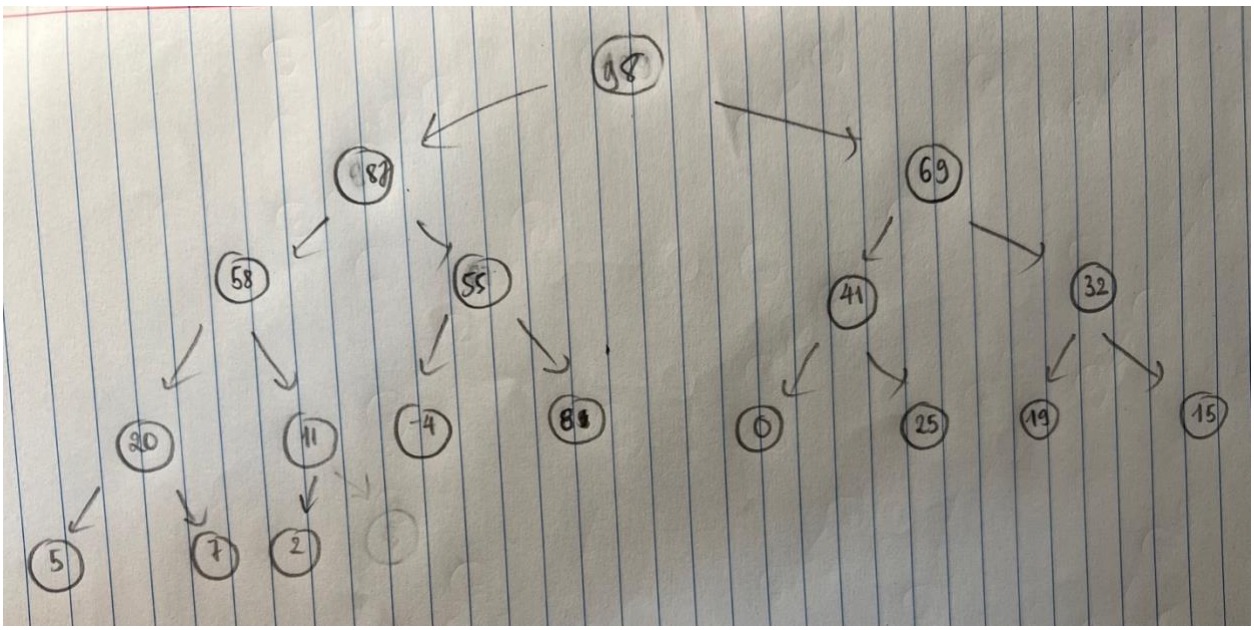
Step 4: trickle-down, swap value of index 1 with the highest value below it (index 4)

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
98	87	69	58	8	41	32	20	11	-4	55	0	25	19	15	5	7	2	

Step 5: trickle-down, swap value of index 4 with the highest value below it (index 10)

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
98	87	69	58	55	41	32	20	11	-4	8	0	25	19	15	5	7	2	

Output:



REMOVE 2:

Step 1: remove root node 98

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
	87	69	58	55	41	32	20	11	-4	8	0	25	19	15	5	7	2

Step 2: take the node has smallest index to replace the root

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
2	87	69	58	55	41	32	20	11	-4	8	0	25	19	15	5	7	

Step 3: trickle-down, swap value of index 0 with the highest value (index 1)

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
87	2	69	58	55	41	32	20	11	-4	8	0	25	19	15	5	7	

Step 4: trickle-down, swap value of index 1 with the highest value below it (index 3)

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
87	58	69	2	55	41	32	20	11	-4	8	0	25	19	15	5	7	

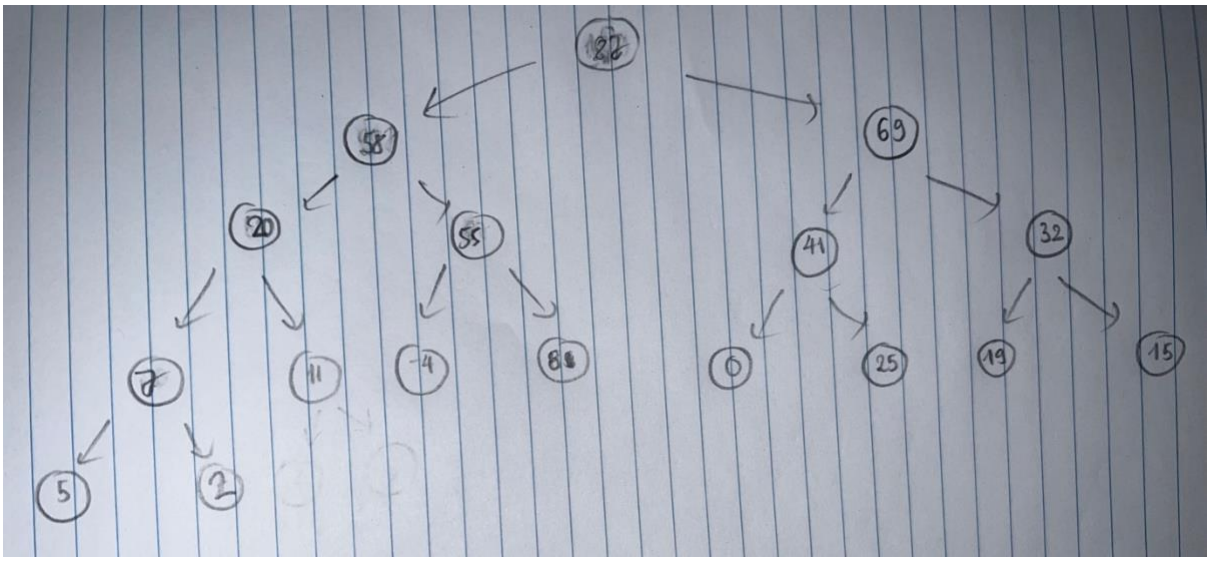
Step 5: trickle-down, swap value of index 3 with the highest value below it (index 7)

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
87	58	69	20	55	41	32	2	11	-4	8	0	25	19	15	5	7	

Step 6: trickle-down, swap value of index 7 with the highest value below it (index 16)

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
87	58	69	20	55	41	32	7	11	-4	8	0	25	19	15	5	2	

Output:



D. Give an example of real-life applications of Heap data structure, clearly describe how the data will be stored (i.e., identify key). **(3 marks)**

Heap plays an important role in many algorithms, among all **selection algorithms** is what I want to talk about.

Selection algorithms : with heap is built from an unsorted data

- find min
  - in max heap: most bottom node → O(logN)
  - in min heap: root node → O(1)
- find max
  - in min heap: most bottom node → O(logN)
  - in max heap: root node → O(1)
- Find median: we need two heaps: min heap and max heap. Because our heap size remains unchanged, we can consider the median as the average of the root elements of both heaps, if the sizes of both heaps = n / 2. Otherwise, the root element of the min-heap is the median.

Time complexity explain:

- 1 element: time complexity of the step is Log 1 (a single element being in a single heap)
- 2 elements, time complexity of the step is Log 1 (as we have one element in each heap)

  ….
- 4 elements, the complexity of the step is Log 2 (as we have two elements in each heap)

➔ So, with n elements, the complexity is Log n (as we have n/2 elements in each heap)
➔ Inserting n/2 values
➔ Time complexity: O(nlogn)

It would be a faster way than finding the median by other methods.

Question 7: Hash Table(s) (Total Marks: 18)

A. Consider the following (Fig.7) hash functions for implementation of quadratic probing with a table length of 27 items. Use the hash function to insert the following value (Fig.8) in the hash table (add missing rows). Show your steps including any collusions. **(6 marks)**

I did write a small python file to demonstrate the quadratic probing algorithm

Python file: **Q7PartA.py**

Here is the step by step when I insert values to the hash table by hand:

- Insert key “24”:

ASCII(2) = 50 and ASCII(4) = 52

Checksum = 3445 + 50 + 52 = 3547

hashIdx = 3547 % 27 = 10

- Insert key “31”:

ASCII(3) = 51 and ASCII(1) = 49

Checksum = 3445 + 51 + 49 = 3545

hashIdx = 3545 % 27 = 8

- Insert key “1234”:

ASCII(1) = 49 ; ASCII(2) = 50; ASCII(3) = 51; ASCII(4) = 52

Checksum = 3445 + 49 + 50 + 51 + 52 = 3647

hashIdx = 3647 % 27 = 2

- Insert key “456”:

ASCII(4) = 52 ; ASCII(5) = 53; ASCII(6) = 54

Checksum = 3445 + 52 + 53 + 54 = 3604

hashIdx = 3604 % 27 = 13

- Insert key “789”:

ASCII(7) = 55 ; ASCII(8) = 56; ASCII(9) = 57

Checksum = 3445 + 55 + 56 + 57 = 3613

hashIdx = 3613 % 27 = 22

- Insert key “890”:

ASCII(8) = 56 ; ASCII(9) = 57; ASCII(0) = 48

Checksum = 3445 + 55 + 56 + 57 = 3606

hashIdx = 3606 % 27 = 15

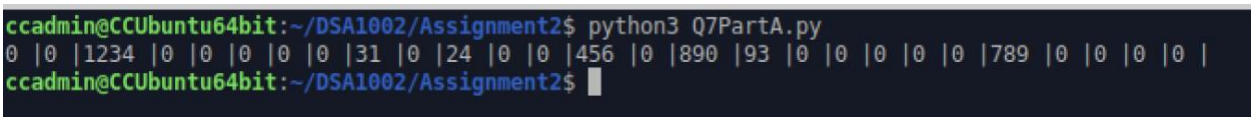
- Insert key “93”:

ASCII(9) = 57 ; ASCII(3) = 51

Checksum = 3445 + 57 + 51 = 3553

hashIdx = 3553 % 27 = 16

When you run the python file **Q7PartA.py**, you will receive the exact same index if insert value !



I did not see any duplicate index, so there is no collusion here. Just simply put the insert key and value into hash table with the corresponding index.

	Hash Table	
Index	Key	Value
0		
1		
2	“1234”	“Student 3”
3		
4		
5		
6		
7		
8	“31”	“Student 2”
9		
10	“24”	“Student 1”
11		
12		
13	“456”	“Student 4”
14		
15	“890”	“Student 6”
16	“93”	“Student 7”
17		
18		
19		
20		
21		
22	“789”	“Student 5”
23		
24		
25		
26		

B. How you will keep the load factor below 0.75 for the hash table, what exactly is the meaning of using this load factor in the case of the hash table developed in Part A. Describe your answer by stating effectiveness w.r.t. memory utilisation and overhead of hash table resizing for this load factor selection. **(6 marks)**

Meaning of load factor: help us measure how full the hash table is allowed to get before its capacity is automatically increased.

In order to keep the load factor below 0.75, I will:

+ write a shouldResize() function: this function will check the number of value in hash table with its size. If count / size >= 0.75 ➔ resize the hashtable

+ create another function call resize():

- Set count = 0
- Initialize oldkey and oldvalue
- Set size = size * 2
- Copy the value from old hashtable to newhashtable

```
for i in range(len(oldValues)):
        if oldValues[i] is not None:
                self.put(oldKeys[i], oldValues[i])
```

+ then in the insert function, when I finish insert I will check whether or not it over 0.75 load factor by calling shouldResize() function.

➔ by doing that, the load factor will be always less than 0.75

From my understanding:

- The lower load factor → more empty spaces → less chances of collision → high performance → high space requirement (need more memory)
- The higher load factor → fewer empty spaces → higher chance of collision → lower performance → lower space requirement (don't waste too much memory and computer resources).

C. Describe the limitations of separate chaining for the implementation of a hash table, describe how the searching and element insertion time is affected by the separate chain-based implementation of hash table. Show and explain the time complexity for separate chaining. **(6 marks)**

Disadvantages of separate chaining:

- Waste of memory which uses for storing empty spaces (as some parts of the space in hash table never be touched)
- The longer the chain is the bigger time complexity for searching: 0(n) in the worst scenario
- Waste resources for the links in linked list (pointer)

Legend:

- m: number of empty spaces left in hash table
- n: number of key inserted in the hash table
- a: load factor

  a = m/n

+ Searching: average case: O(1 + a); worst case: O(n)
+ Inserting: average case: O(1 + a); worst case: O(n)
+ Deleting: In order to delete a key, we need to search the key first and delete it. In average case the time complexity of deletion is O(1+ a). But in worst case the time complexity of the search operation is O(n). So, the time complexity of deleting a particular key in worst case is also O(n).

If the length of hash table is bigger and bigger, it leads to the worst case in searching, inserting and deleting of hashtable → O(n)