ASSIGNMENT 4

Data Structures & Algorithms

COMP 352

Laila Alhalabi #40106558

April 19, 2021

Question 1.

(a) Construct the BST that results when the following integers are inserted in the order given:

50, 20, 40, 55, 45, 30 10, 15, 35, 60, 5, 25

(b) Perform preorder, inorder, and postorder traversals of the BST above.

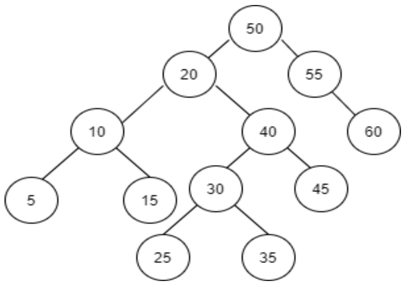
(c) Construct the BST that results when the following integers are deleted from the BST in the order given:

60, 30, 20

(d) Perform preorder, inorder, and postorder traversals of the BST in item (c) above.

ANSWER

(a)



(b)

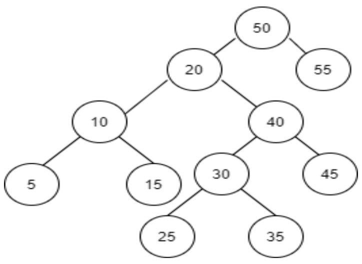
Preorder: 50, 20, 10, 5, 15, 40, 30, 25, 35, 45, 55, 60

Inoreder: 5, 10, 15, 20, 25, 30, 35, 40, 45, 50, 55, 60

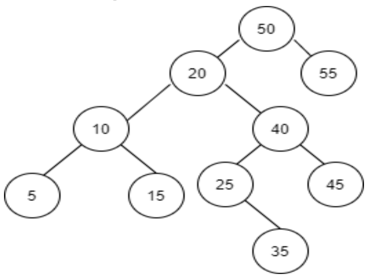
Postorder: 5, 15, 10, 25, 35, 30, 45, 40, 20, 60, 55, 50

(c)

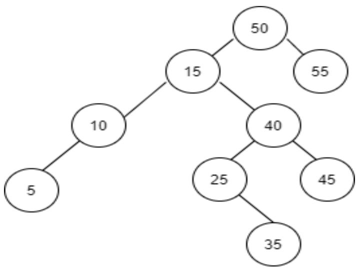
Deleting 60:



Deleting 30:



Deleting 20:



(d)

Preorder: 50, 15, 10, 5, 40, 25, 35, 45, 55

Inoreder: 5, 10, 15, 25, 35, 40, 45, 50, 55

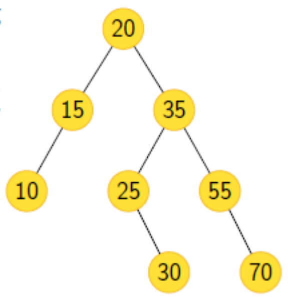
Postorder: 5, 10, 35, 25, 45, 40, 15, 55, 50

Question 2.

For each of the following questions start with the AVL tree shown below.

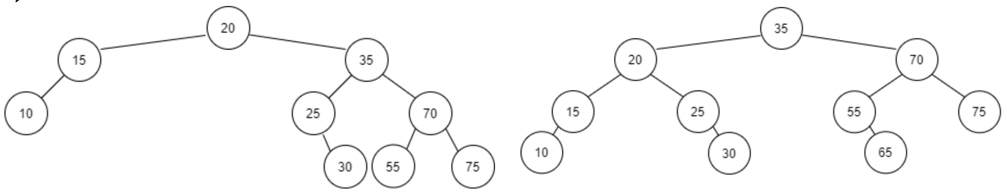
Show the AVL tree that results from performing the following operations.

To be clear, in addition to preserving the BST property, you will need to preserve the AVL property of the tree by rebalancing it after **each** insertion and **each** deletion operation.

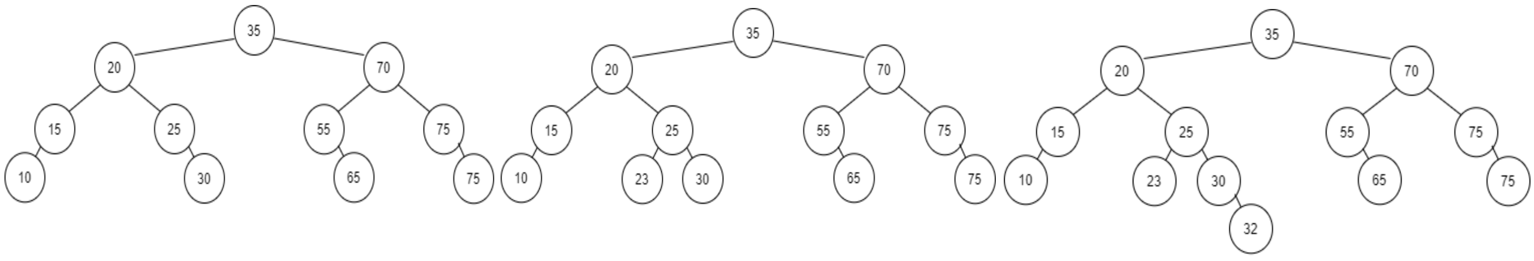


(a) Add 75 and then 65
(b) Add 75, 23, and then 32
(c) remove 15 and then 35
(d) remove 25 and then 30

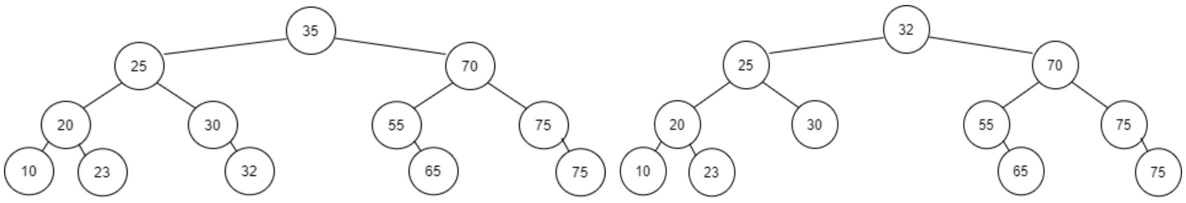
(a)



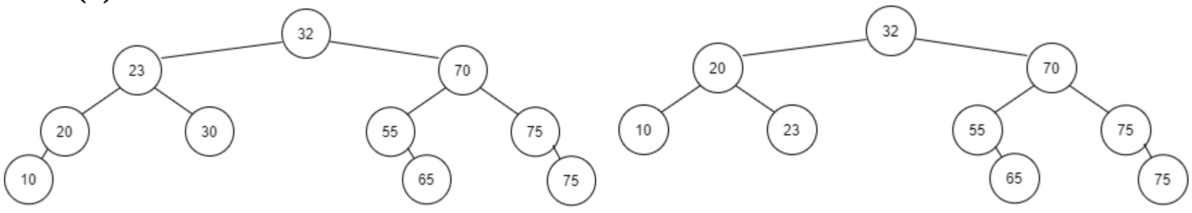
(b)

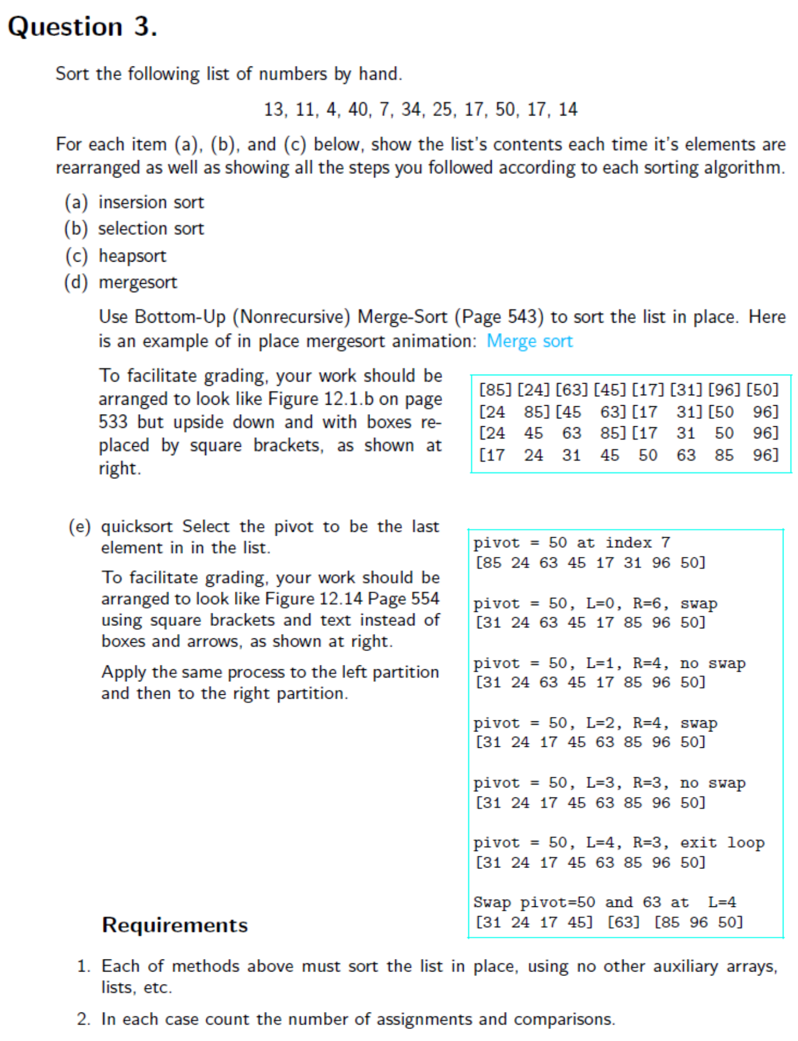


(c)



(d)





Question 3.

Sort the following list of numbers by hand.

13, 11, 4, 40, 7, 34, 25, 17, 50, 17, 14

For each item (a), (b), and (c) below, show the list's contents each time it's elements are rearranged as well as showing all the steps you followed according to each sorting algorithm.

(a) insersion sort
(b) selection sort
(c) heapsort
(d) mergesort

Use Bottom-Up (Nonrecursive) Merge-Sort (Page 543) to sort the list in place. Here is an example of in place mergesort animation: Merge sort

To facilitate grading, your work should be arranged to look like Figure 12.1.b on page 533 but upside down and with boxes replaced by square brackets, as shown at right.

```
[85] [24] [63] [45] [17] [31] [96] [50]
[24   85] [45   63] [17   31] [50   96]
[24   45   63   85] [17   31   50   96]
[17   24   31   45   50   63   85   96]
```

(e) quicksort Select the pivot to be the last element in in the list.

To facilitate grading, your work should be arranged to look like Figure 12.14 Page 554 using square brackets and text instead of boxes and arrows, as shown at right.

Apply the same process to the left partition and then to the right partition.

```
pivot = 50 at index 7
[85 24 63 45 17 31 96 50]

pivot = 50, L=0, R=6, swap
[31 24 63 45 17 85 96 50]

pivot = 50, L=1, R=4, no swap
[31 24 63 45 17 85 96 50]

pivot = 50, L=2, R=4, swap
[31 24 17 45 63 85 96 50]

pivot = 50, L=3, R=3, no swap
[31 24 17 45 63 85 96 50]

pivot = 50, L=4, R=3, exit loop
[31 24 17 45 63 85 96 50]

Swap pivot=50 and 63 at  L=4
[31 24 17 45]  [63]  [85 96 50]
```

Requirements

1. Each of methods above must sort the list in place, using no other auxiliary arrays, lists, etc.
2. In each case count the number of assignments and comparisons.

(a) insertionSort()

Insertion sort works by:
1. Peeking a key and comparing it with the keys ahead of it.
2. Repeating comparing till reaching the end.

Pseudocode: (number of comparisions and assignments is 4)

```
Algorithm insertionSort (a[])
Input array a of n positive integers, Output the sorted array
for a ← 0 to array.length
        current ← array[a]; b ← a – 1;
        while (b > -1 & array[b] > current)
                array[b + 1] ← array[b]; b--;
         array[b + 1] ← current
```

Contents:
[11 13] 4 40 7 34 25 17 50 17 14
[4 11 13] 40 7 34 25 17 50 17 14
[4 11 13 40] 7 34 25 17 50 17 14
[4 7 11 13 40] 34 25 17 50 17 14
[4 7 11 13 34 40] 25 17 50 17 14
[4 7 11 13 25 34 40] 17 50 17 14
[4 7 11 13 17 25 34 40] 50 17 14
[4 7 11 13 17 25 34 40 50] 17 14
[4 7 11 13 17 17 25 34 40 50] 14
[4 7 11 13 14 17 17 25 34 40 50]

(b) selectionSort()

Selection sort works by:

1. Searching for the smallest value and inserting it first.
2. Discarding the smallest value and reapeating the same thing for the rest of the values and so on.

Pseudocode: (number of comparisions and assignments is 5)

```
Algorithm selectionSort (a[])
Input array a of n positive integers, Output the sorted array
for  a ← 0 to  array.length - 1
    int current ← a;
    for b ← a + 1 to  array.length
        if (array[b] < array[current])
            current ← b
        smaller ← array[current]
        array[current] ← array[a]
        array[a]  ← smaller
```

Contents:

```
[4] 11 13 40 7 34 25 17 50 17 14
4 [7] 13 40 11 34 25 17 50 17 14
4 7 [11] 40 13 34 25 17 50 17 14
4 7 11 [13] 40 34 25 17 50 17 14
4 7 11 13 [14] 34 25 17 50 17 40
4 7 11 13 14 [17] 25 34 50 17 40
4 7 11 13 14 17 [17] 34 50 25 40
4 7 11 13 14 17 17 [25] 50 34 40
4 7 11 13 14 17 17 25 [34] 50 40
4 7 11 13 14 17 17 25 34 [40] 50
```

(c) heapSort()

Heap sort works by:

1. Represent the array as a binary tree & build a heap.
2. The max element will be at the root (n-1).
3. The max element will be removed from the heap and the size will be reduced by 1.
4. A heapify() method was created to adjust the nodes of the heap.

Pseudocode: (number of comparisions and assignments is 3)

```
Algorithm heapSort (a[])
Input array a of n positive integers, Output the sorted array
for i ← array.length / 2 – 1 to i >= 0
    heapify(array, array.length, i)
for i ← array.length – 1 to  i >= 0
    int temp ← array[0]
    array[0] ← array[i]
    array[i] ← temp
    heapify(array, i, 0)
```

Contents:

```
50 40 34 17 17 4 25 13 11 7 14
40 17 34 14 17 4 25 13 11 7 [50]
34 17 25 14 17 4 7 13 11 [40] 50
25 17 11 14 17 4 7 13 [34] 40 50
17 17 11 14 13 4 7 [25] 34 40 50
17 14 11 7 13 4 [17] 25 34 40 50
14 13 11 7 4 [17] 17 25 34 40 50
13 7 11 4 [14] 17 17 25 34 40 50
11 7 4 [13] 14 17 17 25 34 40 50
7 4 [11] 13 14 17 17 25 34 40 50
4 [7] 11 13 14 17 17 25 34 40 50
```

(d) mergeSort()

Merge sort works by :
1. Dividing the array into two parts.
2. Sorting the two parts divided.
3. Merging the two parts.

<u>Pseudocode:</u> (number of comparisions and assignments is 26)

```
Algorithm mergeSort (a[])
Input array a of n positive integers, Output the sorted array
for currSize ← 1; currSize <= n-1; currSize ← 2*currSize
    for leftStart ← 0; leftStart < n-1; leftStart += 2*currSize
        mid ← min(leftStart + currSize - 1, n-1)
        rightEnd ← min(leftStart + 2*currSize - 1, n-1)
        merge(arr, leftStart, mid, rightEnd)
Algorithm merge (a[], left, mid, right)
Input array a of n positive integers, leftStart & rightEnd. Output merged array.
    n1 ← mid - left + 1
    int n2 ← right - mid
    L[] ← new int[n1]
    R[] ← new int[n2]
    for i ← 0; i < n1; i++
        L[i] ← arr[left + i]
    for j ← 0; j < n2; j++
        R[j] ← arr[mid + 1eft + j]
    i ← 0, j ← 0, k ← left
    while (i < n1 & j < n2)
        if (L[i] <= R[j])
            arr[k] ← L[i]
            i++
        else
            arr[k] ← R[j]
            j++
        k++
    while (i < n1)
        arr[k] ← L[i]
        i++, k++
    while (j < n2)
        arr[k] ← R[j]
        j++, k++
```

<u>Contents:</u>

```
[13] [11] [4] [40] [7] [34] [25] [17] [50] [17] [14]
[ 11 13 ] [ 4 40 ] [7 34 ] [ 17 25 ] [ 17 50 ] [ 14 ]
[ 4 7 11 13 34 40 ] [ 14 17 17 25 50]
[ 4 7 11 13 14 17 17 25 34 40 50 ]
```

(e) quickSort()

Quick sort works by:
1. Picking the last element x as a pivot.
2. Dividing the array to L (elements > x), S (elements < x), R (elements = x).
3. Sorting the partitions and merging.

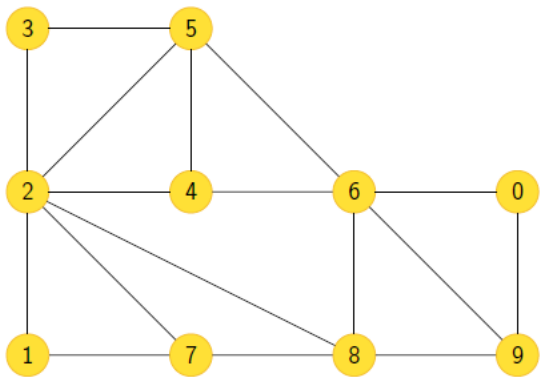
<u>Pseudocode:</u> (number of comparisions and assignments is 9)

```
Algorithm quickSort (a[], left, right)
Input array a of n positive integers, left & right indices. Output the sorted array
 if( left < right)
     int index ← partition(arr, left, right)
     quickSort(arr, left, index - 1)
     quickSort(arr, index+1, right)
Algorithm partition (a[], left, right)
Input array a of n positive integers, left & right indices. Output the sorted array
int pivot ← arr[right]
int high ← right
while (left < high)
     while(left < high & arr[left] < pivot)
          left++
     while(left < high & arr[high] >= pivot)
          high--
      swap(arr,left, high)
swap(arr, left, right)
```

<u>Contents:</u>
Pivot = 14 at index 10 swap
[13 11 4 7 40 34 25 17 50 17 14]
swap
[13 11 4 7 14 34 25 17 50 17 40]
Pivot = 7 at index 3
[13 11 4 7]
swap
[4 7 11 13]
Pivot = 40 at index 5
[34 25 17 50 17 40]
swap
[34 25 17 17 50 40]
Pivot = 40 at index 5
[34 25 17 17 40 50]
Pivot = 17 at index 4
[34 25 17 17]
swap
[17 17 34 25]
Pivot = 25 at index 3
[25 34]
MERGE
[4 7 11 13 14 17 17 25 34 40 50]

Question 4.

- Consider the graph shown below:



1. Give the adjacency matrix representing this graph.
2. Give the adjacency lists representing this graph.
3. Show the breadth-first search trees for the graph starting at node 0.
4. Show the depth-first search trees for the the graph starting at node 0.

(a)

	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
0	0	0	0	0	0	0	1	0	0	1
1	0	0	1	0	0	0	0	1	0	0
2	0	1	0	1	1	1	0	1	1	0
3	0	0	1	0	0	1	0	0	0	0
4	0	0	1	0	0	1	1	0	0	0
5	0	0	1	1	1	0	1	0	0	0
6	1	0	0	0	1	1	0	0	1	1
7	0	1	1	0	0	0	0	0	1	0
8	0	0	1	0	0	0	1	1	0	1
9	1	0	0	0	0	0	1	0	1	0

(b)

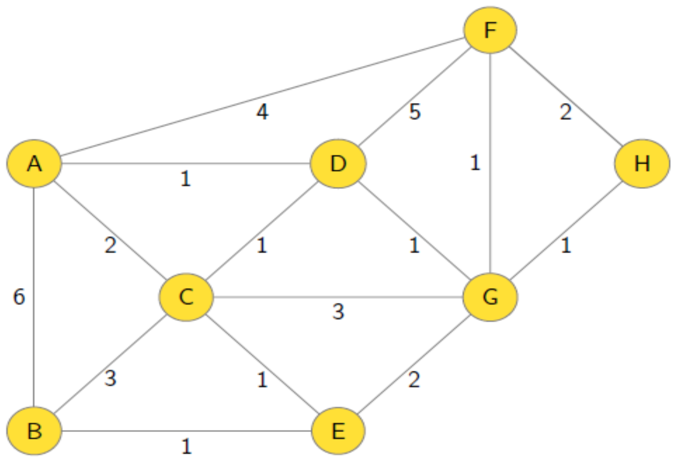
[**0** [6, 9]
1 [2, 7]
2 [1, 3, 4, 5, 7, 8]
3 [2, 5]
4 [2, 5, 6]
5 [2, 3, 4, 6]
6 [0, 4, 5, 8, 9]
7 [1, 2, 8]
8 [2, 6, 7, 9]
9 [0, 6, 8] **]**

(c)
BFS: 0 6 9 4 5 8 2 3 7 1

(d)
DFS: 0 6 4 5 3 2 1 7 8 9

Question 5.

- Consider the graph shown below:



1. Applying Kruskal's algorithm, find a minimal spanning tree for the graph.

   Use a table that includes column headers shown below (and possibly other columns of your choice) to show the progress of the algorithm at each step.

Step	Edge Considered	Cost	Accepted/Rejected

2. Applying Prim's algorithm, find a minimal spanning tree for the graph.

   Use a table that includes the following columns (and possibly other columns of your choice) to show the progress of the algorithm at each step.

Vertex	Visited?	Cost	Predecessor

(1) Minimal Spanning Tree = 8

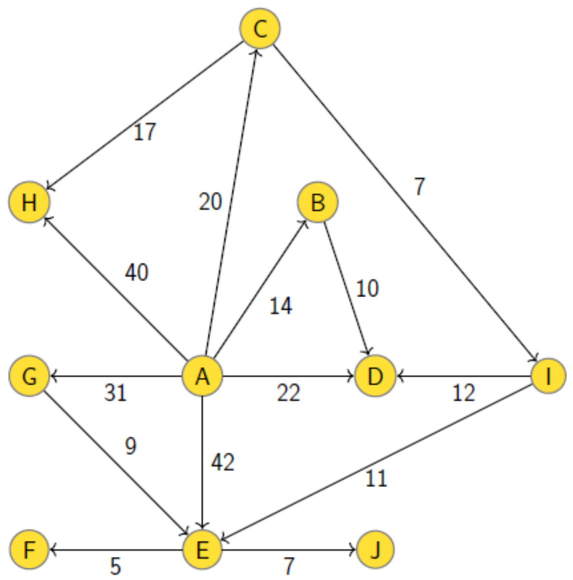
Step	**Edge Considered**	**Cost**	**A/R**
1	AD	1	A
2	DC	1	A
3	CE	1	A
4	EB	1	A
5	DG	1	A
6	GF	1	A
7	FH	2	A
8	EG	2	R
9	CA	2	R
10	HG	1	R

(2) Minimal Spanning Tree = 8

Vertex	**Visit**	**Cost**	**Predecessor**
A	Y	0	-
B	Y	1	E
C	Y	1	D
D	Y	1	A
E	Y	1	C
F	Y	1	G
G	Y	1	D
H	Y	2	F

Question 6.

Consider the graph shown below.



(a) Find depth-first and breadth-first topological orderings for the graph.

(b) Applying Dijkstra's algorithm, find the shortest distance from node A to every other node in the graph.

Use a table that includes the following columns (and possibly other columns of your choice) to show the progress of the algorithm at each step.

Vertex	Visited?	Cost	Predecessor

(a)
DFS: ABDCIEJFHG
BFS: ABCHDGIEJF

(b)

Vertex	Visit	Cost	Predecessor
A	Y	0	
B	Y	14	A
C	Y	20	A
D	Y	22	A
E	Y	38	I (path: A, C, I)
F	Y	43	E (path: A, C, I, E)
G	Y	31	A
H	Y	37	C (path: A, C)
I	Y	27	C (path: A, C)
J	Y	45	E (path: A, C, I, E)