Question 1

a) Given a tree T, where n is the number of nodes of T. Give an algorithm for computing the depths of all the ▪ Depth-First-Search with parameter (depth var) nodes of a tree T. What is the complexity of your algorithm in terms of Big-O?

▪ Initialize depth var to 1

▪ For every DFS: increment depth var

▪ Depth var returns depth of each node

Time complexity here would be O(n+e) where n represents the number of node and e represents the edges of the tree.

b) We say that a node in a binary search tree is full if it has both a left and a right child. Write an algorithm called Count-Full-Nodes(t) that takes a binary search tree rooted at node t, and returns the number of full nodes in the tree. What is the complexity of your solution?

Depth-First-Search with global variable (numOfCompleteNodes)
▪ Initialize numOfCompleteNodes to 0
▪ For every DFS: check if node has both a left and a right child
▪ If it has both child: increment the numOfCompleteNodes
▪ numOfCompleteNodes is returning the the count of complete node s in the tree at the end.

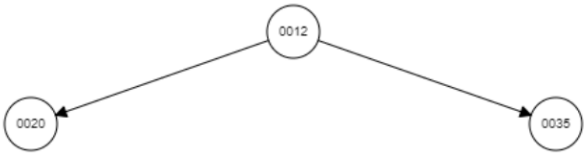
Time complexity here would be O(n+e) where n represents the number of node and e represents the edges of the tree.

Question 2

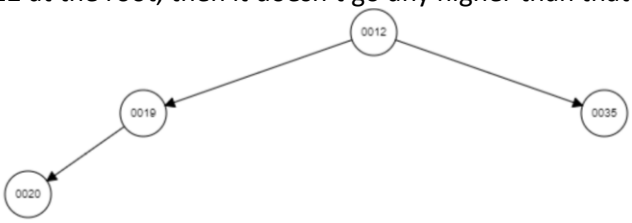
a) Draw the min-heap that results from the bottom-up heap construction algorithm on the following list of values: 20, 12, 35, 19, 7, 10, 15, 24, 16, 39, 5, 19, 11, 3, 27. Starting from the bottom layer, use the values from left to right as specified above. Show immediate steps and the final tree representing the min-heap. Afterwards perform the operation removeMin 6 times and show the resulting min-heap after each step.

List of values: 20, 12, 35, 19, 7, 10, 15, 24, 16, 39, 5, 19, 11, 3, 27.
Step 1: Insert keys 20, 12; Since 12 is smaller than 20, 12 moves up to the root and 20 becomes the child.

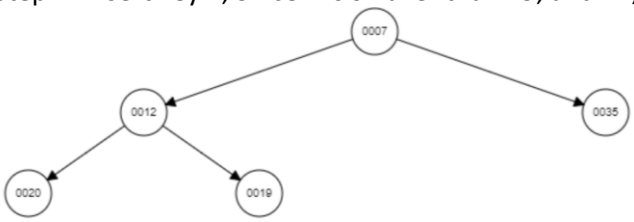
Step 2: Insert key 35; Since 35 is larger than 12 at the root, then it stays put:



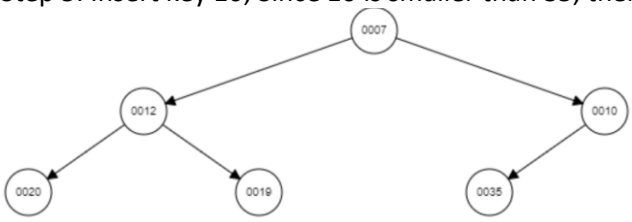
Step 3: Insert key 19; Since 19 is smaller than 20, it moves one level up, but since it is larger than 12 at the root, then it doesn't go any higher than that:



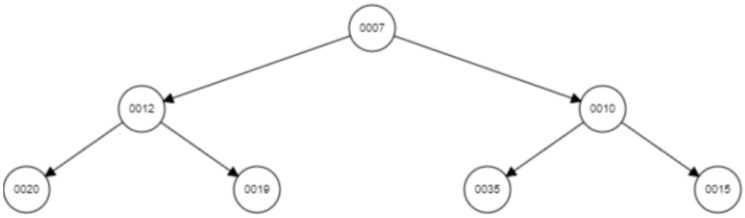
Step 4: Insert key 7; Since 7 is smaller than 19, and 12, then it moves two levels up:



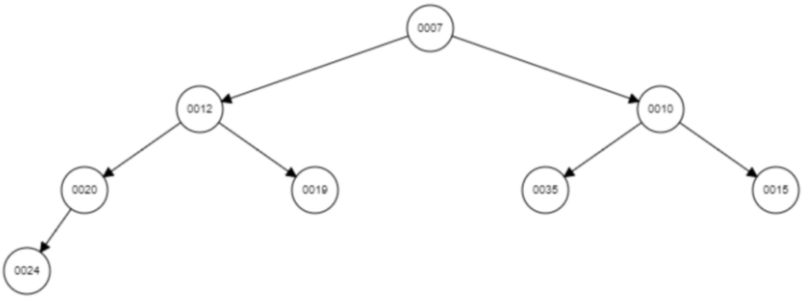
Step 5: Insert key 10; Since 10 is smaller than 35, then it moves one level up:



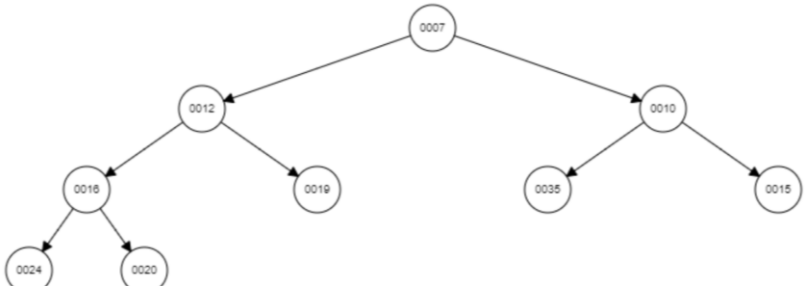
Step 6: Insert key 15; since 15 is not smaller than 10, then it stays put:



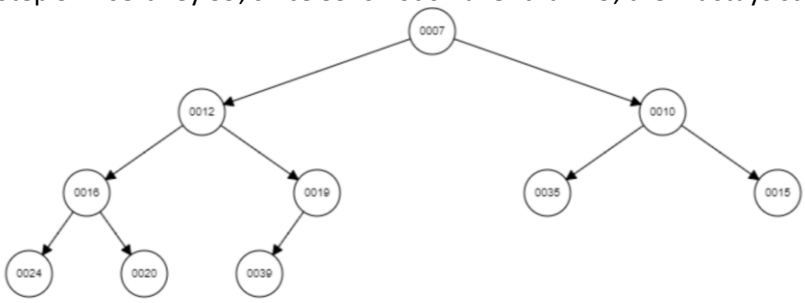
Step 7: Insert key 24; since 24 is not smaller than 20, then it stays put:



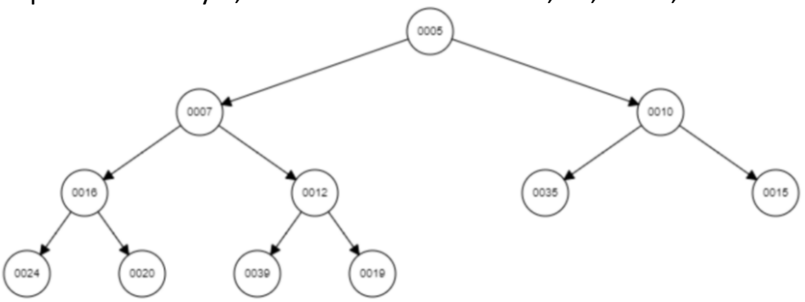
Step 8: Insert key 16; since 16 is smaller than 20, then it moves one level up:



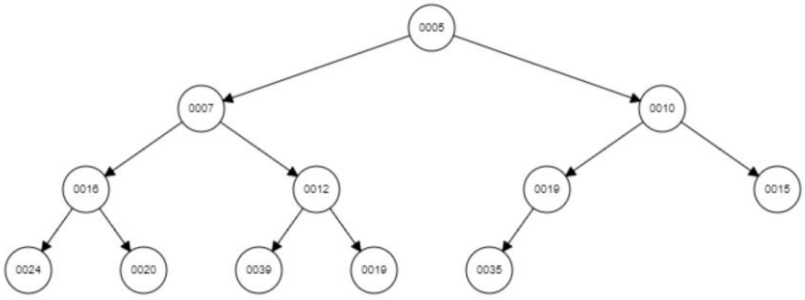
Step 9: Insert key 39; since 39 is not smaller than 19, then it stays still:



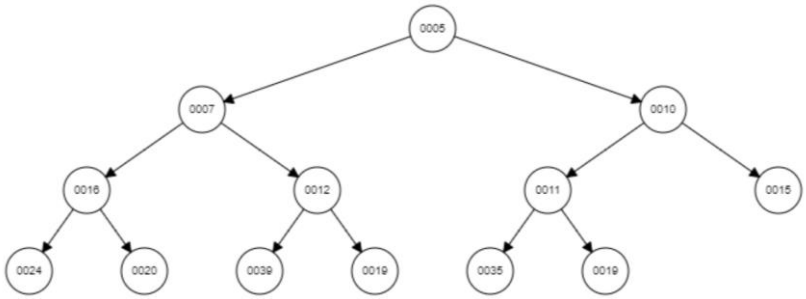
Step 10: Insert key 5; since 5 is smaller than 19, 12, and 7, then it moves three levels up:



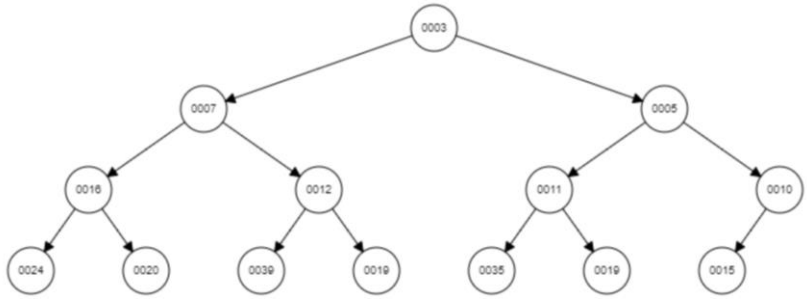
Step 11: Insert key 19; since 19 is smaller than 35, then it moves one level up:



Step 12: Insert key 11; since 11 is smaller than 19, then it moves one level up:

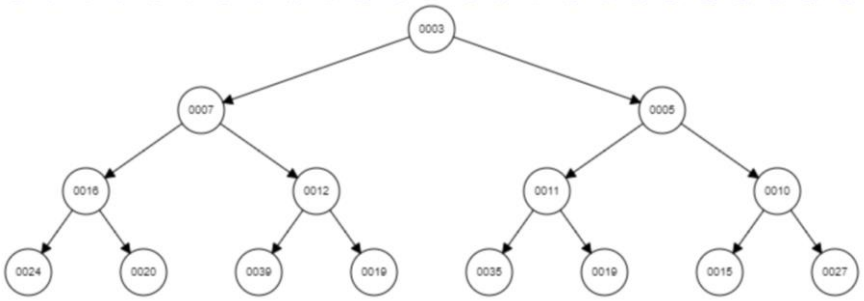


Step 13: Insert key 3; since 3 is smaller than 15, 10, and 5, then it moves three levels up:



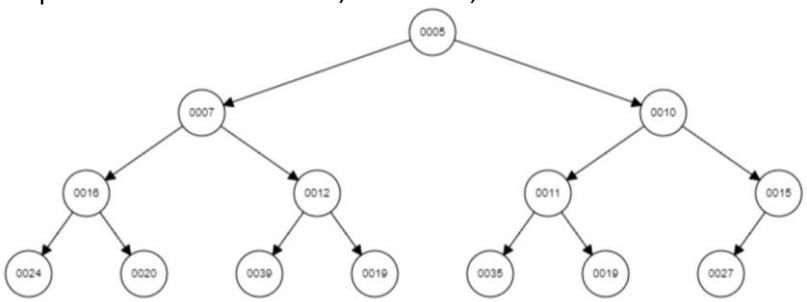
Step 14: Insert key 27; since 27 is not smaller than 10, then it stays still:

b) Create again a min-heap using the list of values from the above part (a) of this question but this time you have to insert these values step by step using the order from left to right (i.e. insert 20, then insert 12, then 35, etc.) as shown in the above question. Show the tree after each step and the final tree representing the min-heap.

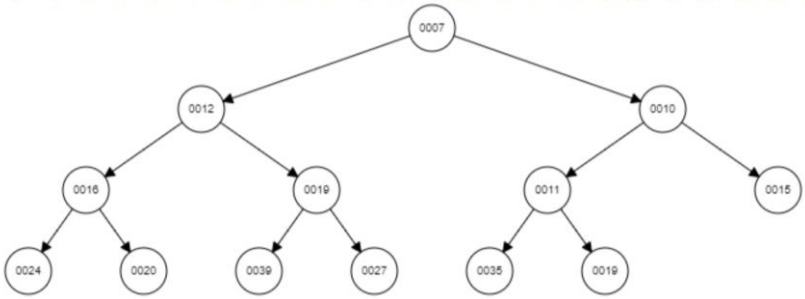


Now, we are going to perform the operation removeMin 6 times and show the resulting min-heap after each step

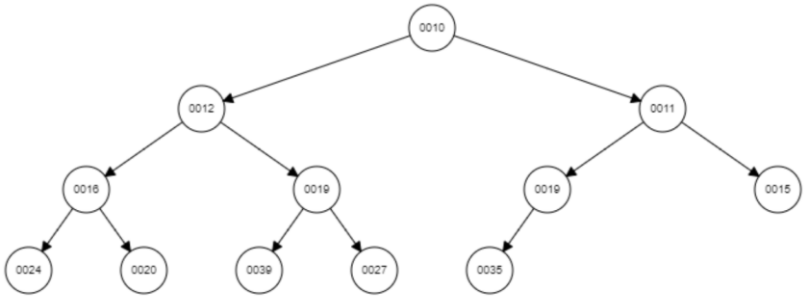
Step 1: Remove the smallest, which is 3, such that:



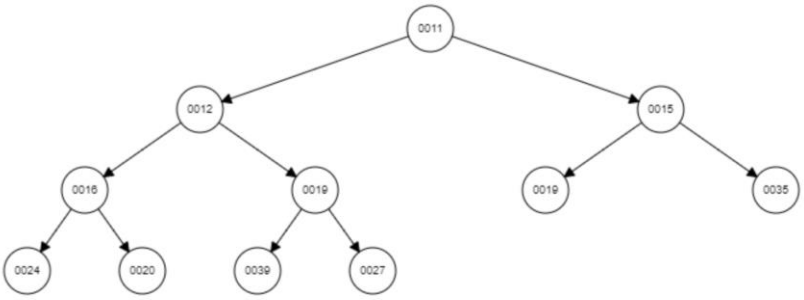
Step 2: Remove the new smallest value, which is 5, such that:



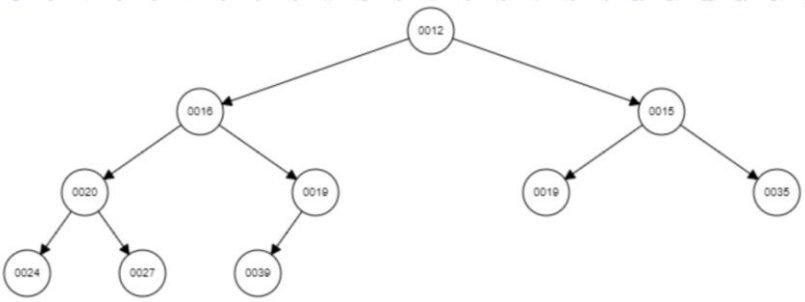
Step 3: Remove the new smallest value, which is 7, such that:



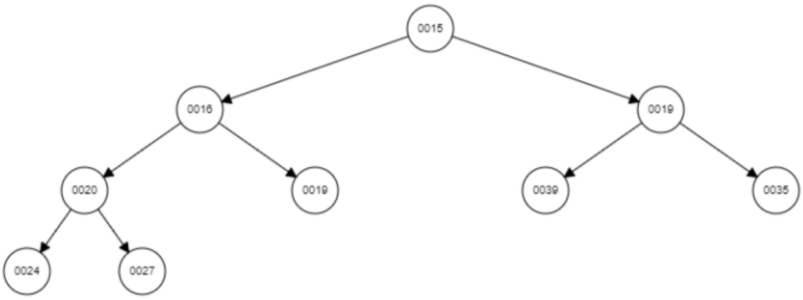
Step 4: Remove the new smallest value, which is 10, such that:



Step 5: Remove the new smallest value, which is 11, such that:

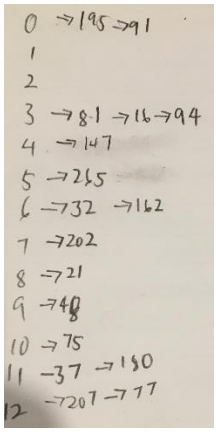


Step 6: Remove the new smallest value, which is 12, such that:



Question 3 Assume a hash table utilizes an array of 13 elements and that collisions are handled by separate chaining. Considering the hash function is defined as: h(k)=k mod 13.

i) Draw the contents of the table after inserting elements with the following keys:
32, 147, 265, 195, 207, 180, 21, 16, 189, 202, 91, 94, 162, 75, 37, 77, 81, 48.



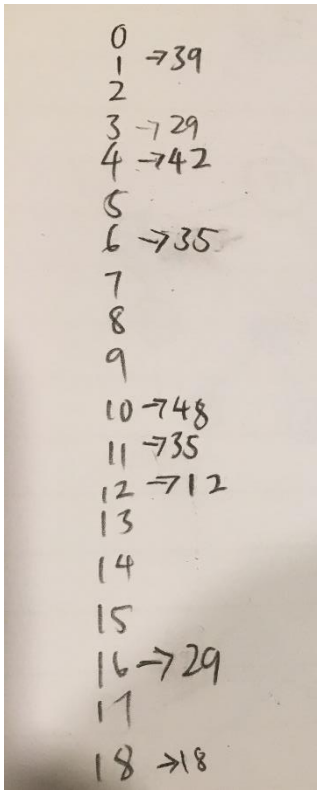
ii) What is the maximum number of collisions caused by the above insertions?

Question 4 To reduce the maximum number of collisions in the hash table described in Question 3 above, someone proposed the use of a larger array of 15 elements (that is roughly 15% bigger) and of course modifying the hash function to: h(k)=k mod 15. The idea is to reduce the load factor and hence the number of collisions. Does this proposal hold any validity to it? If yes, indicate why such modifications would actually reduce the number of collisions. If no, indicate clearly the reasons you believe/think that such proposal is senseless.

Yes, there will be total 15 different remainders if we divide by 15. Which means the key can go to 15 different arrays. The load factor will decrease as well since load factor equals to number of entries divided by number of arrays.

Question 5 Assume an open addressing hash table implementation, where the size of the array is N = 19, and that double hashing is performed for collision handling. The second hash function is defined as: d(k) = q - k mod q, where k is the key being inserted in the table and the prime number q is = 7. Use simple modular operation (k mod N) for the first hash function.

i) Show the content of the table after performing the following operations, in order: put(25), put(12), put(42), put(31), put(35), put(39), remove(31), put(48), remove(25), put(18), put(29), put(29), put(35).



ii) What is the size of the longest cluster caused by the above insertions?

4

iii) What is the number of occurred collisions as a result of the above operations?

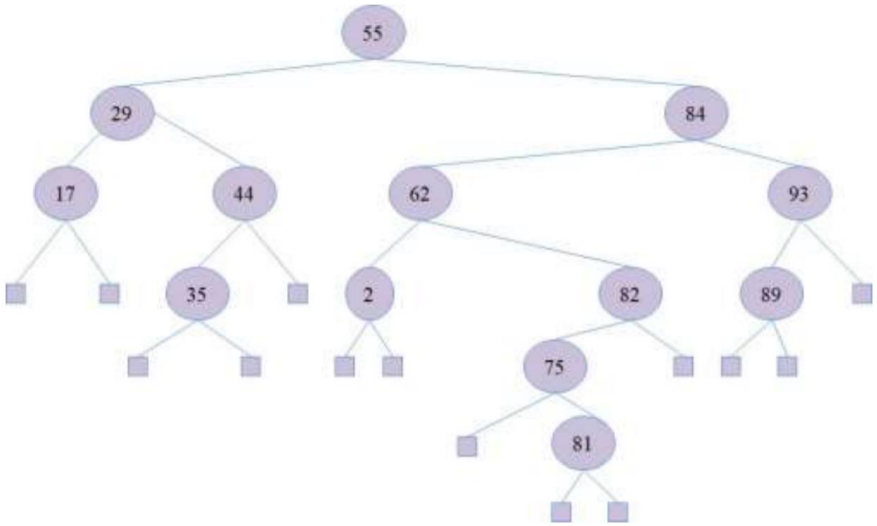
10

iv) What is the current value of the table's load factor?

Load factor = 9/19

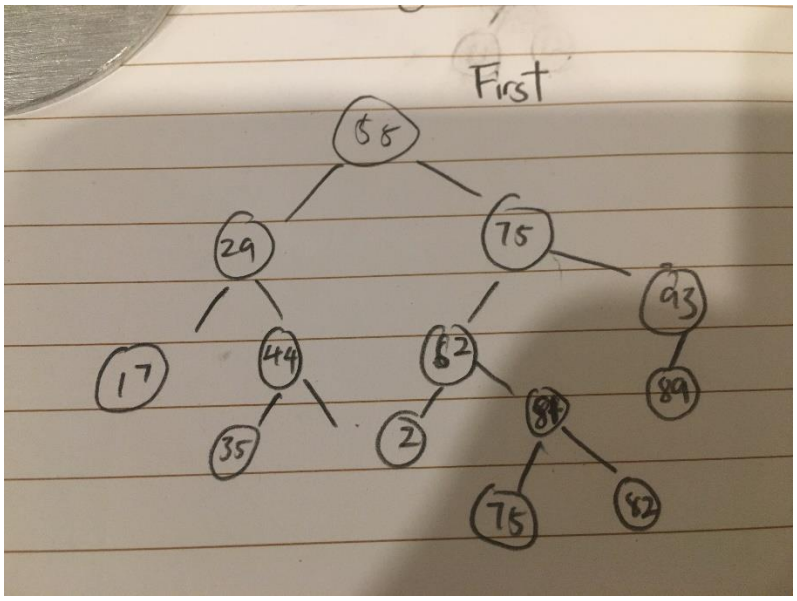
= 0.474

Question 6 Given the following tree, which is assumed to be an AVL tree:



i) Are there any errors with the tree as shown? If so, indicate what the error(s) are, correct these error(s) [you should attempt applying the smallest possible number of changes to correct the tree], show the corrected AVL tree, then proceed to the following questions (Questions ii to iv) and start with the tree that you have just corrected. If no errors are there in the above tree, indicate why the tree is correctly an AVL tree, then proceed to the following questions (Questions ii to iv) and continue working on the tree as shown above.

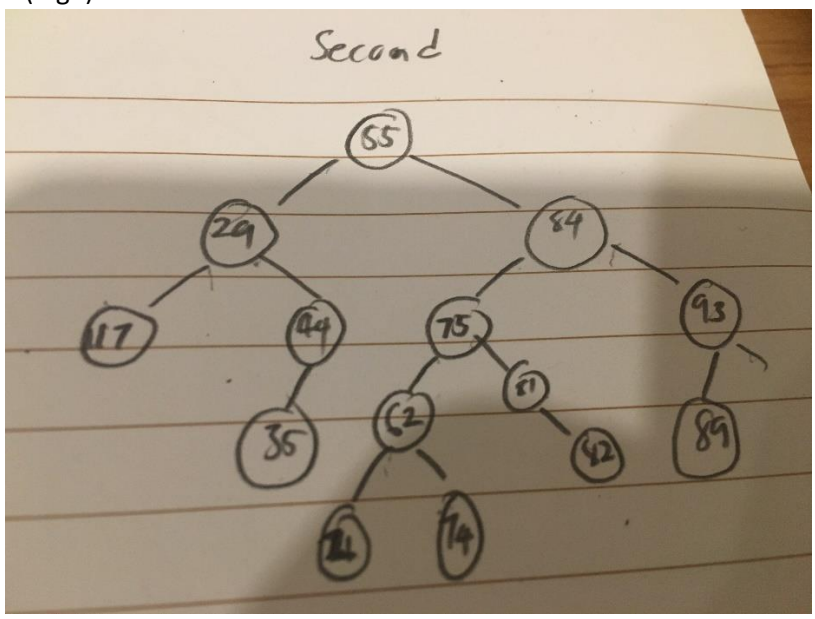
- ➔ The left subtree and right subtree are not balanced
- ➔ AVL trees require the height of subtrees of any node to differ by no more than one level



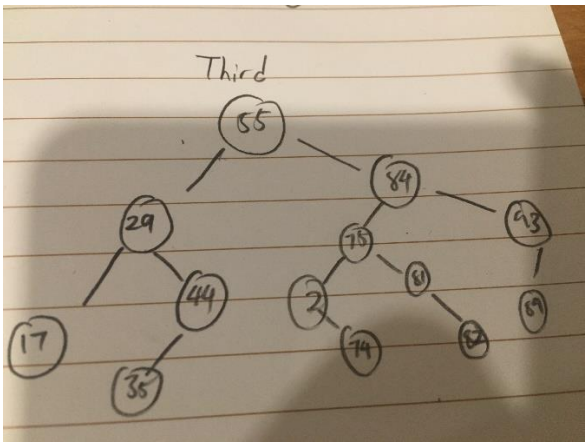
…2 can't even be there in the first place

ii) Show the AVL tree after put(74) operation is performed. Give the complexity of this operation in terms of Big-O notation.

O(logn)



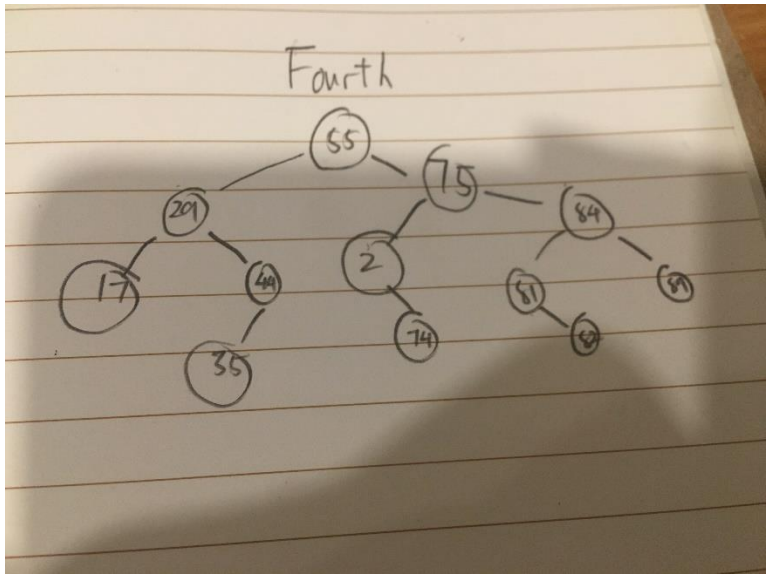
iii) Show the AVL tree after remove(62) is performed. Give the complexity of this operation in terms of Big-O notation.



-

O(logn)

iv) Show the AVL tree after remove(93) is performed. Show the progress of your work step-by-step. Give the complexity of this operation in terms of Big-O notation



O(log n)

Programming Part

1. A detailed report about your design decisions and specification of your CVR ADT including a rationale and comments about assumptions and semantics.

Ans: Array is for fast insertion and less size usage compare to ADT or arraylist. Initialize the array with a fixed size of threshold allows it to display data more quicker than other data structures in low data size. However, in large quantity of data, AVL tree is more suitable since every action on time complexity is O(logn) Which is ideal when you managing large quantity of sizes.

2.

Pseudo Code

```
        public void setThreshold(int Threshold)
  {
    this.Threshold=Threshold;
    CarArray= new Car[Threshold];
  }

  public void setKeyLength(int Length) throws Exception{
    if(Length<10 || Length>17){
      throw new Exception("Please enter a number between 17 and 10");
    }

    KeyLength = Length;
  }

  public ArrayList prevAccids(String key)
  {
    ArrayList Result=null;
    if (Tree==null){
```

```
        for(int i=0;i<this.SizeOfArray;i++){
            if VIN of this CarArray equals(key){
                Result = CarArray[i].Accidents;
                Use collection to sort the array
                return Result;
            }
        }
    }else{
        Result = this.Tree.find(key).Accidents;
        Collections.sort(Result, Collections.reverseOrder());

    }

    return Result;
}



public void add(String key, Car newCar)
{
    if(Tree==null){
        if(newCar.VIN.length()!=KeyLength){
            println("Please enter VIN of same keylength");
            return;
        }
        for(int i=0;i<SizeOfArray;i++){
            if(CarArray[i].equals(newCar.VIN)){
```

```
            println("This VIN exist!");
            return;
          }
        }

        CarArray[SizeOfArray] = newCar;
        SizeOfArray++;
        if(SizeOfArray>=Threshold){
          System.out.println("Start Using AVLCar!");
          Tree = new AVLCar();
          for(Car car: CarArray){
            System.out.println("Added to AVL: "+ car);
            Tree.root=Tree.insert(Tree.root, car);
          }

        }
      }else{
        if(newCar.VIN.length()!=KeyLength){
          println("Please enter VIN of same keylength");
          return;
        }
        If Tree.findKey (Key) !=null
          return;
        Tree insert new Car
      }
    }
```

```
public void remove(String key){
   if (Tree==null){

      for(int i=0;i<this.SizeOfArray;i++){
            if(CarArray[i].VIN.equals(key)){
               while(CarArray[i+1]!=null){
                  CarArray[i]=CarArray[i+1];
                  i++;
               }
               Minus one to sizeOfArray
               return;
            }
         }
   }else{
      Tree delete node at string Key
   }
}

public String nextKey(String key)
{
   Initialize String Result to "";
   if (Tree==null){
      //Insertion Sort
      for(int i=0;i<SizeOfArray;i++){
         try{
            sort(CarArray, SizeOfArray);
            if(CarArray[i].VIN.equals(key)){
```

```
                return CarArray[i+1].VIN;
            }
        }catch null pointer exception to indicate when it reaches the end


    }
  }else{
    Result = Use Tree function to find successor
  }
  return Result;
}

public String prevKey(String key)
{
  String Result="";
  if (Tree==null){
    //Insertion Sort
    for(int i=0;i<SizeOfArray;i++){
      try{
        if(CarArray[i].VIN.equals(key)){
          sort(CarArray, SizeOfArray);
          return CarArray[i-1].VIN;
        }
      }catch(NullPointerException e){
        return null;
      }
    }
  }else{
```

```
        Result = Tree.inOrderPredecessor(key);
    }
    return Result;
}

public void allKeys(){
    if (Tree==null){
        //Insertion Sort
        sort(CarArray,SizeOfArray);
        for(int i=0;i<SizeOfArray;i++){
            print CarArray[i].VIN
        }
    }else{
        Tree perform inorder transversal

    }
}



public Car getValues(String Key)
{
    Car Result = null;
    if  CarArray not equals to null {
        for(int i=0;i<this.SizeOfArray;i++){
                if(CarArray[i].VIN.equals(Key)){
                    Result = CarArray[i];
                    return Result;
```

```
            }
          }
      }else{
         Result =  Tree.find(Key);

      }
      return Result;
   }

   public ArrayList generate(int n)
   {

      // chose a Character random from this String
      Initialize alphanumericString with bunch of different characters and numbers

      create StringBuilder size of keyLength
      initialize int j to 0;
      initialize ArrayList result;
      while j is less than n:
         for int I = 0, I less than KeyLength, i++

            // generate a random number between
            // 0 to AlphaNumericString variable length
            Generate a random index based on the length of AlphanumericString that we want to
choose

            add Character one by one in end of sb
```

```
    }
    If Tree isn't null
       If Tree found a node with existing VIN generated by the generator:
            Reset StringBuilder
         else
            add sb.toString to the result arrayList
            j++

    }else{
       If SizeOfArray equals to 0{
                add sb.toString to the result arrayList
                j++
       }else{
          Use for loop to iterate through array
                If array contains VIN built by stringbuilder:
                          Reset the stringBuilder
                Else If iterating reaches the end
                  Add sb.toString to the result arrayList
                  Reset stringbuilder
                  j++
               }
            }
          }
       }
    }
}
```

```
    Return Result
}
```