Name (Student ID): Anik Patel (40091908) Teacher: Dhrubajyoti Goswami

Course-section: COMP 352-AA Submission Date: June 22th, 2019

Assignment 4

Q1

A)

0	1	2	3	4	5	6	7	8	9	10	11	12
45		89	24	16	12	38	95	25		31	14	27

① (3(31) − 5) mod 13
= 10

② (3(45) − 5) mod 13
= 0

③ (3(14) − 5) mod 13
= 11

④ (3(89) − 5) mod 13
= 2

⑤ (3(24) − 5) mod 13
= 2 → collision +1 = 3

⑥ (3(95) − 5) mod 13
= 7

⑦ (3(12) − 5) mod 13
= 5

⑧ (3(38) − 5) mod 13
= 5 → collision +1 = 6

⑨ (3(27) − 5) mod 13
= 11 → collision +1 = 12

⑩ (3(16) − 5) mod 13
= 4

⑪ (3(25) − 5)
= 5 collision +1 = 6
collision +1 = 7
collision +1 = 8

h'(k): 7 − (k mod 7)

B)

0	1	2	3	4	5	6	7	8	9	10	11	12
45		89		16	12	24	95	25	38	31	14	27

k	h(k)	h'(k)
31	10 fine	
45	0 fine	
14	11 fine	
89	2 fine	
24	2 collision + (1)4 = 6 fine	

k	h(k)	h'(k)
95	7 fine	
12	5 fine	
38	5 collision + (1)4 = 9 fine	
27	11 collision + (1) 1 = 12 fine	
16	4 fine	
25	5 collision + (1)3 = 8 fine	

Q2

2

sorted part

pivot

to swap

22	72	38	48	13	14	93	69	45	58	13	81	79	
22	13	38	48	13	14	93	69	45	58	72	81	79	
22	13	38	45	13	14	93	69	48	58	72	81	79	
22	13	38	45	13	14	48	69	93	58	72	81	79	→ New pivots
14	13	38	45	13	22	48	69	72	58	93	81	79	
14	13	22	45	13	38	48	69	58	72	93	81	79	
14	13	13	45	22	38	48	69	58	72	93	81	79	
14	13	13	22	45	38	48	69	58	72	93	81	79	→ New pivots
13	13	14	22	38	45	48	58	69	72	79	81	93	

only 1 elements between sorted parts, thus all can be conquered

→ 13	13	14	22	38	45	48	58	69	72	79	81	93

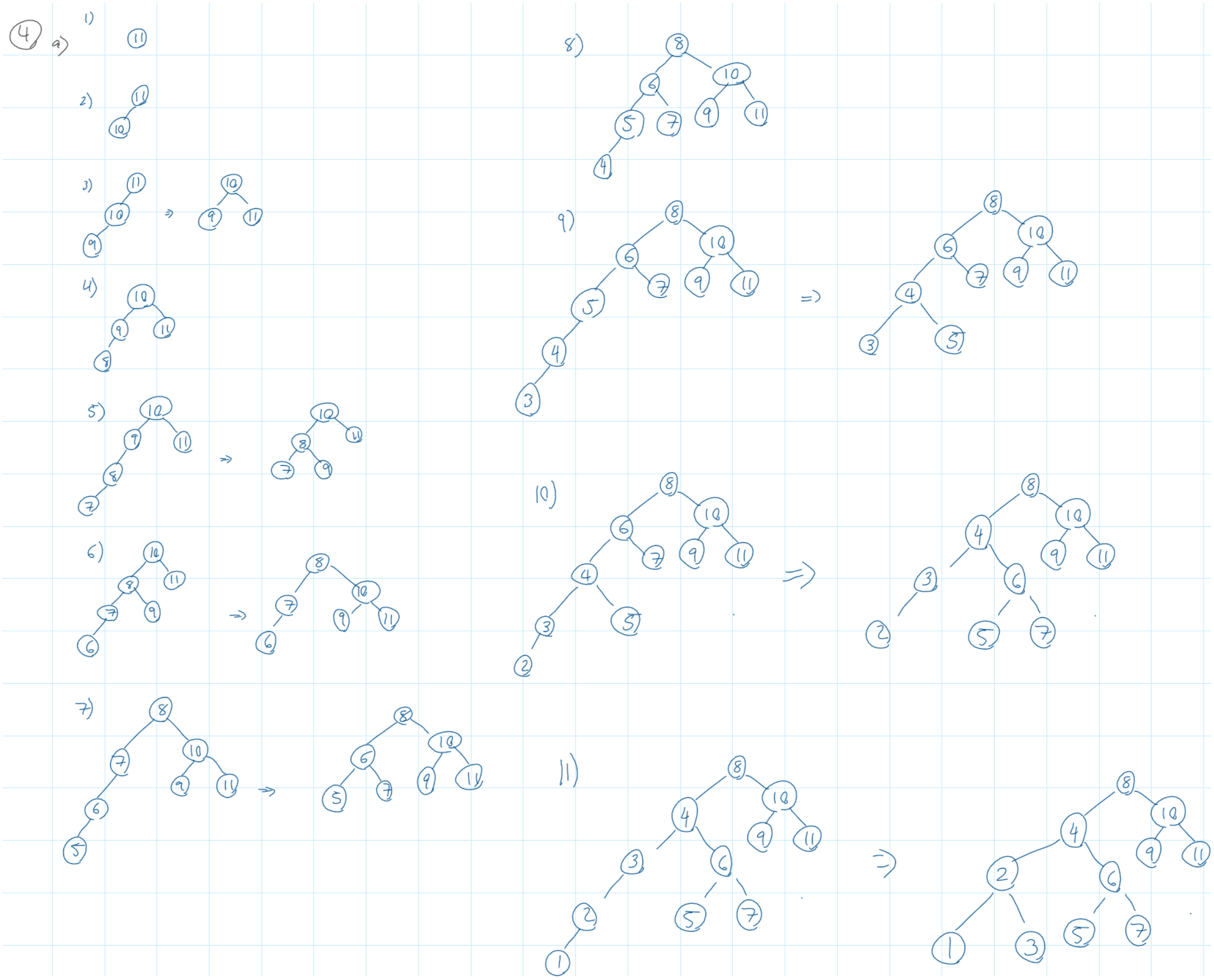
Q3

3

a) Merge sort allows us to split the data into more manageable "chunks" that can fit in the memory.
Merge sort is also a stable sorting algorithm while heap and quick are not.

b) Yes it would be possible to make an in-place merge sort but it would be difficult since we will not have the "optimizations" that the extra memory provide, where we may end up with $O(n^2)$

Q4

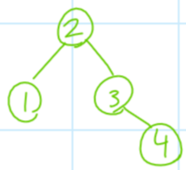


b)

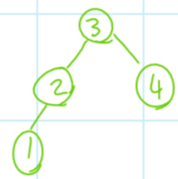
Each unique input will make the tree unique

ex:

Input: 1, 2, 3, 4 4, 3, 2, 1



vs

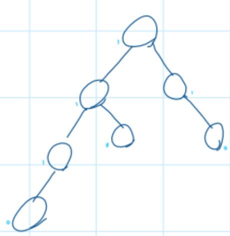


Depending on the order of the inputs, the root of the tree will change

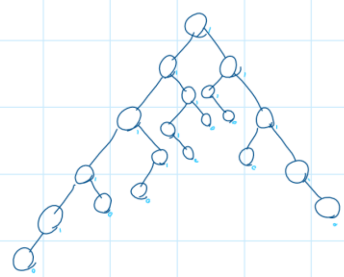
Q5

5)

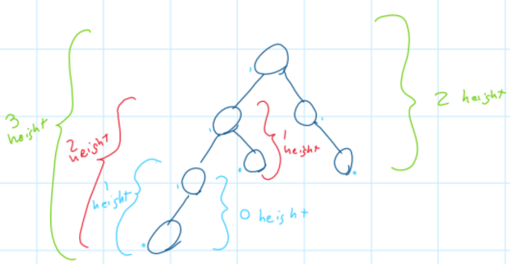
a)



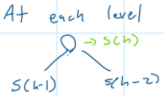
b)



c) Each step up from the bottom-most leaf, needs 1 less height going down.



At each level



Q6

6) if balance is −1 ⇒ we know the right side is the "deepest" subtree

```
def getHeight(T):
    local ← T
    height ← 0
    while local.balance != 0 :          // exit condition
        height ← height + 1
        if local.balance == -1 :         // right is "deeper"
            local ← local.right          // traverse right
        else:
            local ← local.left           // left is "deeper"
                                         // traverse left
    return height
```

Q7

```
7) def hasDupe(R, visited):      // let visited be a list of already visited nodes

       current ← R
       if current is NULL:
           return False
       if current is last element of visited
           return True

       visited.append ← current    // saves current node
       return hasDupe(current.left) or hasDupe(current.right)
```

Since the tree is a BST and balanced, if a duplicate exists, it has to be connected to it's duplicate.

So if we are storing the parent at each step, we only need to look at the end. → checking is O(1) & traversing tree is O(n)

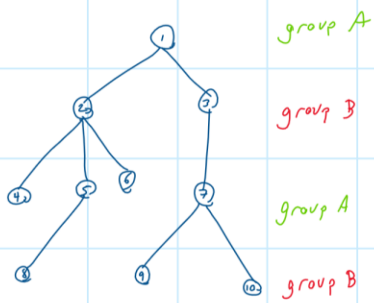
Making this $\boxed{O(n)}$

Q8

8) There are 2 kind of trees in this situation

Case 1: Even Nb of levels

consider groups A & B
every level alternates group



Tree ⇒ {1, 2, 3, 4, 5, 6, 7, 8, 9, 10}

A ⇒ {1, 4, 5, 6, 7}

B ⇒ {2, 3, 8, 9, 10}

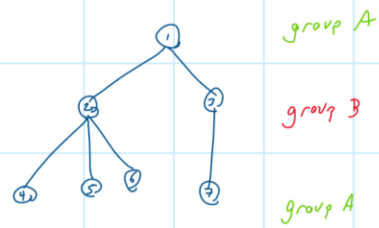
$A \cap B = \emptyset$

$A \cup B = \text{Tree}$

There are only edges between nodes of group A and nodes of group B

Case 2: Odd Nb of levels

consider groups A & B
every level alternates group



Tree ⇒ {1, 2, 3, 4, 5, 6, 7}

A ⇒ {1, 4, 5, 6, 7}

B ⇒ {2, 3,}

$A \cap B = \emptyset$

$A \cup B = \text{Tree}$

There are only edges between nodes of group A and nodes of group B

Q9

a) Proof by contradiction

Assume a bi-partite graph has at least 1 odd cycle

→ Cycle → $\{V_1, V_2, V_3 \ldots V_n\}$ where n is odd

→ For a bi-partite graph, each alternate vertex is a part of a different group.

V_1 is group A → odd ⇒ A
V_2 is group B → even ⇒ B
V_3 is group A → odd ⇒ A
V_4 is group B → even ⇒ B
...

→ n is odd so V_n is in group A

→ since this is a cycle
V_n is connected to V_1

→ group A can't connect to another node of group A, this contradicts our original statement

Therefore a bi-partite graph can not have an odd cycle.

Q10

10) Basically DFS though the whole graph where we indicate if a node is group A or B (alternating at each visite)

```
def hasBipartite(start):
    S ← new Stack
    A, B ← Empty list

    i ← 0
    S.push ← Start
    while (S is not Empty):
        current ← S.pop
        if current is not visited
            if i is even:
                A.append ← current
            else:
                B.append ← current
            current is now visited
            s.push all neighbours
    if A & B has a common element:
        return False

    return True
```

Q11

11) → Since we are talking about a complete graph, we can say that all the edges form a bijection

(every input has a unique output & every output is mapped to an input)

there are $n!$ bijections

→ The path has n vertices, or n starting points and every path can be considered in 2 directions

(ex: $1 \to 2 \to 3 \to 1$)

vs $1 \to 3 \to 2 \to 1$

$$\frac{n!}{2n} = \frac{\cancel{(n)}(n-1)!}{2\cancel{n}} = \boxed{\frac{(n-1)!}{2} \text{ Hamiltonian paths}}$$