Q1:

a)

Nameless returns the common parent node which is at the lowest position in the tree.

Stacks P_u and P_v respectively store the nodes from u and v to the root. We remove the top element from both stack until they are different, and return the reference to the last node that is still the same, therefore the node reference refers to the lowest common parent.

b)

worst case:

u and v are both child of one node and are both lowest leaves(maximum height).

Let h be the height of the tree.

Building stack P_u: $\theta(h)$

Building stack P_v: $\theta(h)$

Pop procedure while loop execute: $\theta(1)$

Pop procedure while loop times: h times

Total: $\theta(h) + \theta(h) + h\big(\theta(1)\big) = \theta(h)$.

best case:

u and v are both root node.

Building stack P_u: $\theta(1)$

Building stack P_v: $\theta(1)$

Pop procedure while loop execute: $\theta(1)$

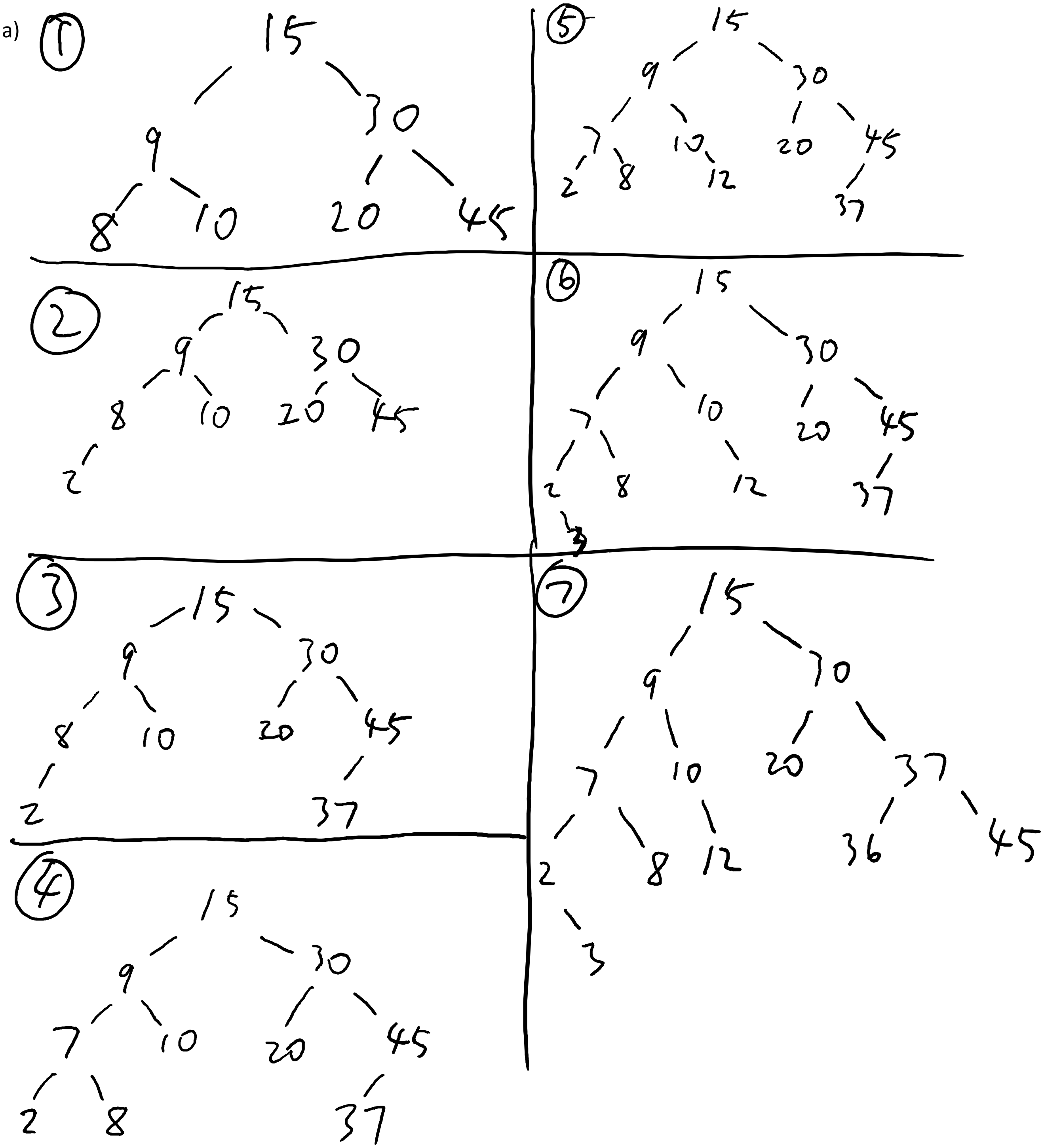
Pop procedure while loop times: 1 times

Total: $\theta(1) + \theta(1) + 1 \times \theta(1) = \theta(1)$.

Q2:

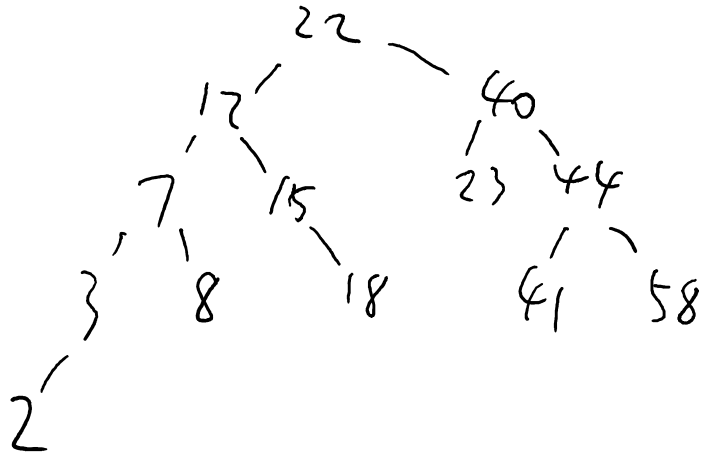
I:

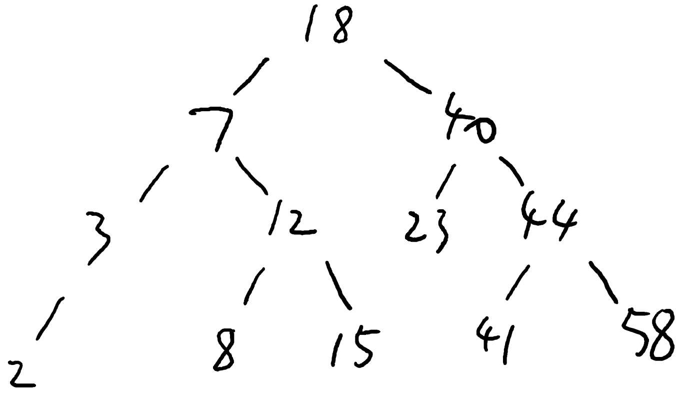
a)

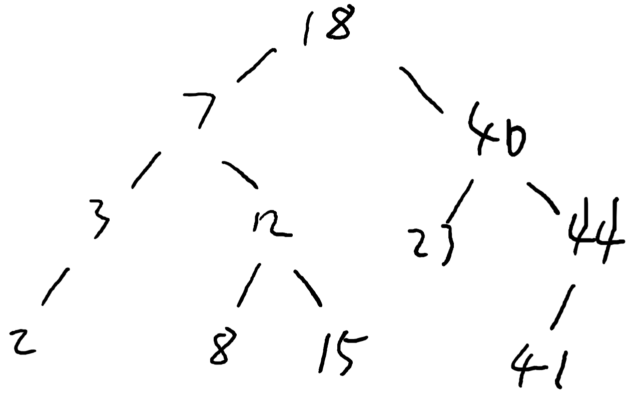


①
15
9
30
8
10
20
45
②
15
9
30
8
10
20
45
2
③
15
9
30
8
10
20
45
2
37
④
15
9
30
7
10
20
45
2
8
37
⑤
15
9
30
7
10
20
45
2
8
12
37
⑥
15
9
30
7
10
20
45
2
8
12
37
3
⑦
15
9
30
7
10
20
37
2
8
12
36
45
3

Q2b:


22
12
40
7
15
23
44
3
8
18
41
58
2


18
7
40
3
12
23
44
2
8
15
41
58


18
7
40
3
12
23
44
2
8
15
41

Q2:

II:

First delete the old node, then increment the value, and lastly add back the node.

```
IncKey(k,d){
      old <- Delete(k);
      Insert(d + old);
}
```

Deletion costs $\theta(height)$, increment costs $\theta(1)$, and insertion costs $\theta(height)$. Intotal the cost is in $\theta(height)$.

Since deletion doesn't break the AVL property, the tree is an AVL tree after "Delete(k)"; Since insertion also keeps the AVL property, the tree after increment is an AVL tree.

Q3:

I:

a)

-inf										+inf
-inf									53	+inf
-inf									53	+inf
-inf									53	+inf
-inf	9	12				50	51		53	+inf
-inf	9	12	26	37	44	50	51	52	53	+inf

b)

P contains: -inf, -inf, 23, 50, 50, 66, 66, 79

II:

a)

The algorithm decide to go down or go right by checking the number of keys skipped: the number of keys skipped > k means it is in range, go down; the number of keys skipped < k means it is in next few blocks, go right, and update k. If they are the same, the algorithm finds the key.

```
Select(L, k)
L: A skip list, k: index
p <- topmost left position of L
h <- L.depth , the height of the list
while (true){
      if (dist(p, h) > k){
            --h;
            p <- below(p);
      } else if (dist(p, h) < k){
            k -= dist(p, h);
            p <- after(p);
      } else {
            while (h > 0){
                  --h;
                  p <- below(p);
            }
            return p;
      }
}
```

b)

After insertion, if the coin toss gives a height which is higher than current height, we need to increase the height of the current list(make the tower for two special keys higher), so that the inserted key is not higher than the two ends.

After deletion, we need to lower the tower for the two special keys, so that it matches the height of the highest key(1 level higher).

Q4:

a)

ABDCEFGH to EHGDABCF

Requires search sequence : D,G,H,E

ABDCEFGH

DABCEFGH

GDABCEFH

HGDABCEF

EHGDABCF

b)

ABDCEFGH to ADBCEGHF

Requires search sequence : D,G,H

ABDCEFGH

ADBCEFGH

ADBCEGFH

ADBCEGHF

c)

for search sequence DHHGHEGH,

Compare + swap	D	H	H	G	H	E	G	H	Total
Before search	ABDCEFGH	DABCEFGH	HDABCEFG	HDABCEFG	GHDABCEF	HGDABCEF	EHGDABCF	GEHDABCF	HGEDABCF
MTF	3+2(1)	8+7(1)	1+0(0)	8+7(1)	2+1(1)	7+6(1)	3+2(1)	3+2(1)	35+34(7)
Before search	ABDCEFGH	ADBCEFGH	ADBCEFHG	ADBCEHFG	ADBCEHGF	ADBCHEGF	ADBCEHGF	ADBCEGHF	ADBCEHGF
Transpose heuristics	3+1	8+1	7+1	8+1	6+1	6+1	7+1	7+1	52+8

Note: If moving an element to the front and push back other elements requires only 1 swap, the number of swaps required is written in the bracket.

Q5:

a)

A = 1,5,9,15,21,30,35,46,47,50

Search 5:

Red is the part that has run.

```
InterpolationSearch(A[0, 9], 5)
if A[0] > 5 || A[9] < 5 then //false
    return false
if 0 = 9 then //false
    if A[l] = k then
        return true
    else
        return false
i ← l + $(r - l)\frac{k-A[l]}{A[r]-A[l]}$ // i is now 0+9*4/49=1
if A[1] == 5 then // 5 == 5, true
        return true
else if A[i] < k then
        return InterpolationSearch(A[i + 1, r], k)
else
        return InterpolationSearch(A[l, 5 - 1], 5)
```

Search 46:

Red is the part that has run.

First call:

```
InterpolationSearch(A[0, 9], 46)
if A[0] > 5 || A[9] < 5 then //false
    return false
if 0 = 9 then //false
    if A[l] = k then
        return true
    else
        return false
i ← l + $(r - l)\frac{k-A[l]}{A[r]-A[l]}$ // i is now 0+9*45/49=8
if A[8] == 46 then // 47 != 46, false
        return true
else if A[i] < k then // false
        return InterpolationSearch(A[i + 1, r], k)
else
        return InterpolationSearch(A[0,  8 - 1], 46)
```

Second call:

```
InterpolationSearch(A[0, 7], 46)
1. if A[0] > 5 || A[7] < 5 then //false
…
3. if 0 = 7 then //false
…
8. i ← l + $(r - l)\frac{k-A[l]}{A[r]-A[l]}$ // i is now 0+7*45/45=7
9. if A[7] == 46 then // 46 == 46, true
10.        return true
…
```

b)

While searching 5, key comparison is done once.

While searching 46, key comparisons are done twice.

It is consistent with the expected complexity since$\log(\log(10)) \approx 1.732$.

c)

An example array could be

1, 2, 3, 4, 5, 46, 47, 48, 49, 50.

Searching 5:

First call:

```
InterpolationSearch(A[0, 9], 5)
3.    if 0 = 9 then //false
8.    i <- l + (r - l)(k-A[l])/(A[r]-A[l]) // i = 0 + 9 * 4/49 = 0
9.    if A[0] == 5 then // 1 != 5, false
12.         return InterpolationSearch(A[0 + 1, 9], 5)
```

Second call:

```
InterpolationSearch(A[1, 9], 5)
3.    if 1 = 9 then //false
8.    i<- l + (r - l)(k-A[l])/(A[r]-A[l]) // i = 0 + 8 * 3/48 = 0
9.    if A[1] == 5 then // 2 != 5, false
12.         return InterpolationSearch(A[1 + 1, 9], 5)
```

Third call:

```
InterpolationSearch(A[2, 9], 5)
3.    if 2 = 9 then //false
8.    i<- l + (r - l)(k-A[l])/(A[r]-A[l]) // i = 0 + 7 * 2/47 = 0
9.    if A[2] == 5 then // 3 != 5, false
12.         return InterpolationSearch(A[2 + 1, 9], 5)
```

Fourth call:

```
InterpolationSearch(A[3, 9], 5)
3.    if 3 = 9 then //false
8.    i<- l + (r - l)(k-A[l])/(A[r]-A[l]) // i = 0 + 6 * 1/46 = 0
9.    if A[3] == 5 then // 4 != 5, false
12.         return InterpolationSearch(A[3 + 1, 9], 5)
```

Fifth call:

```
InterpolationSearch(A[4, 9], 5)
3.    if 4 = 9 then //false
8.    i<- l + (r - l)(k-A[l])/(A[r]-A[l]) // i = 0 + 5 * 0/998883 = 0
9.    if A[4] == 5 then // 5 == 5, true
10.         return True
```

Searching 46:

First call:

```
InterpolationSearch(A[0, 9], 46)
3.    if 0 = 9 then //false
8.    i <- l + (r - l)(k-A[l])/(A[r]-A[l]) // i = 0 + 9 * 45/49 = 8
9.    if A[8] == 46 then // 49 != 46, false
14.         return InterpolationSearch(A[0, 8 - 1], 46)
```

Second call:

```
InterpolationSearch(A[0, 7], 46)
3.    if 0 = 7 then //false
8.    i<- l + (r - l)(k-A[l])/(A[r]-A[l]) // i = 0 + 7 * 45/47 = 6
9.    if A[6] == 46 then // 47 != 46, false
12.         return InterpolationSearch(A[0, 6 - 1], 46)
```

Third call:

```
InterpolationSearch(A[0, 5], 46)
3.    if 0 = 5 then //false
8.    i<- l + (r - l)(k-A[l])/(A[r]-A[l]) // i = 0 + 5 * 45/45 = 5
9.    if A[5] == 46 then // 46 == 5, true
10.         return True
```

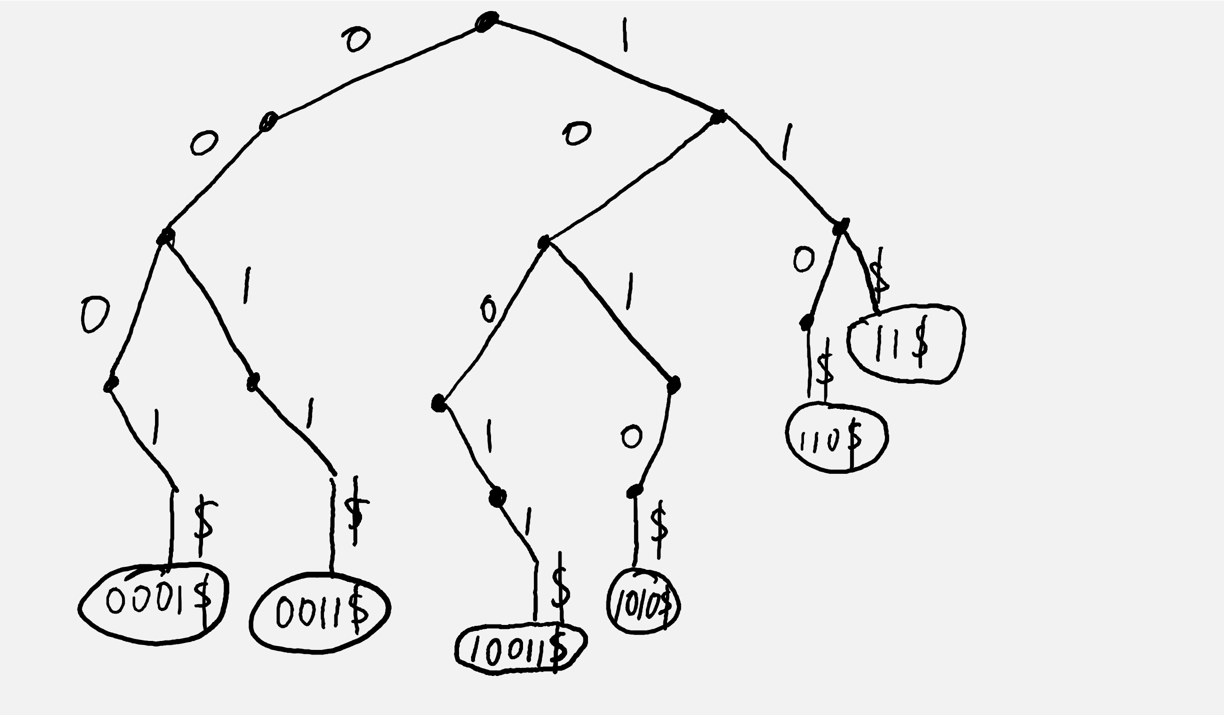
d)

given an number n, create an array $[2,3,4, \dots, n, n^3 + 100]$, then searching n in this array requires $n - 1$ calls to interpolation search.

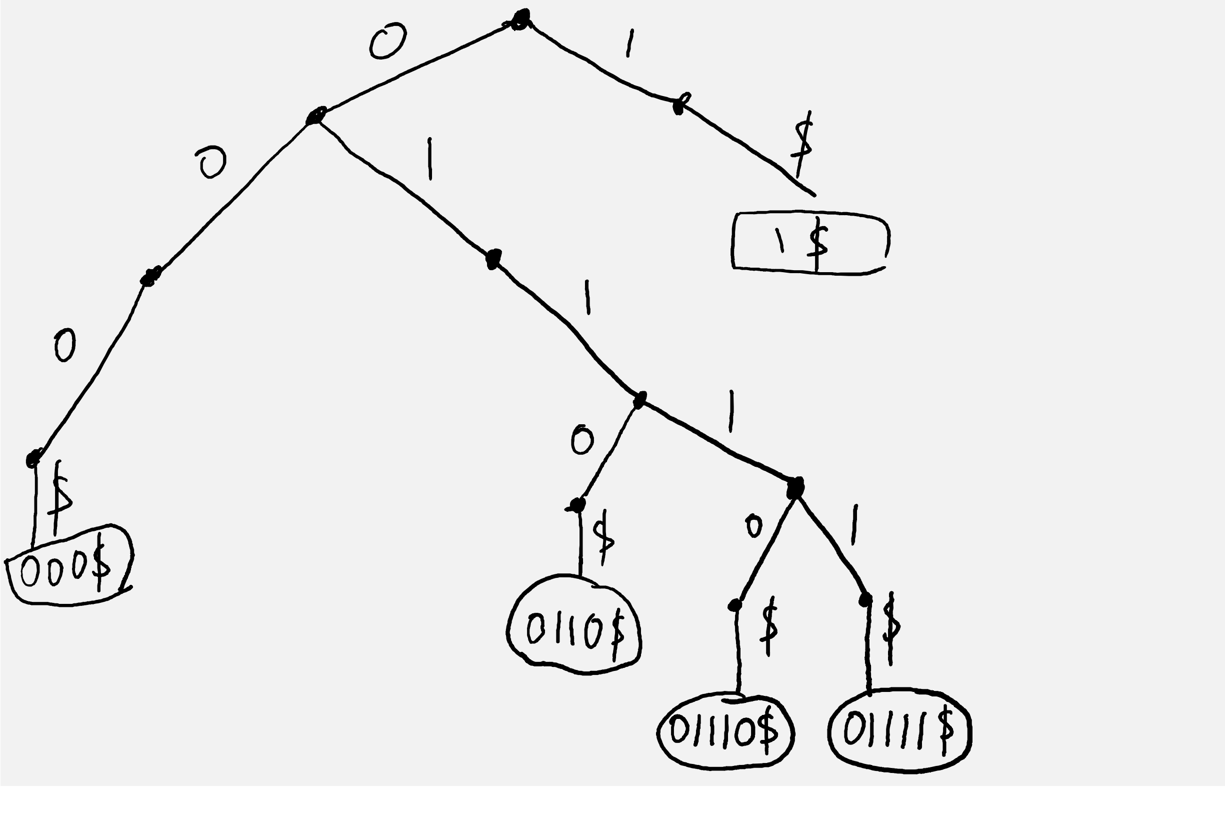
Since in the step "`i <-  l + (r - l)(k-A[l])/(A[r]-A[l])`", `(k-A[l])<=n` , `(A[r]-A[l])<= n^3 + 100 - n,` `(r - l) <= n,` so interpolation always returns the index 0, and it approaches the right answer by one step only. Finding the right answer travers the array except the last element, the time is $\theta(n - 1) = \theta(n)$.

Q6:

a)


0
1
0
0
1
0
1
0
1
0
$
1
1
1
0
$
$
$
1
$
11$
110$
0001$
0011$
10011$
1010$

b)


0
1
0
1
$
1$
0
1
1
0
$
0
1
$
$
$
000$
0110$
01110$
01111$

c)

The algorithm first search the position of x, then recursively add strings, which are (l – x.length) or less levels below that position, to a list. Lastly, return the list.

Searching x takes time $O(m)$ where m is the length of x;

Add string to lists takes time $O(2^{l-m+1})$ where m is the length of x, l is the given length.

```
//returns a pointer to the node containing x(without $)
Node* search(T, x){
      p : pointer to a node
      p = T.root;
      for (auto& one_char: x)
            if (one_char == 0){
                  p = p.left;
            } else if (one_char == 1){
                  p = p.right;
            }
      }
      return p;
}

// add strings starting from pointer p with length less than or equal
// to "length" to my_list
void add_strings(old_str, my_list, p, length){
old_str: string containing previous result
my_list: list of strings as result
p : pointer to a node
length: string length

      if (length >= 0){
            if (p.is_end()){
                  my_list.add(old_str + p.value);
            }
            add_strings(old_str + p.value, my_list, p.left, length - 1);
            add_strings(old_str + p.value, my_list, p.right, length - 1);
      }
}

List<string> Look(T, x, l){
T: Triee
x: given string
l: given maximum length

      p : pointer to a node
      p = search(T, x);
      length = l - x.length();
      my_list : list of strings, initially empty
      add_strings(x, my_list, p, length);
      return my_list;
}
```