Intro to Scalable systems - Self-practice

Part 1

1) We can do this using two arrays.

   i) An array ~~to store~~ of size n to store the elements. Because we know duplicate's are not allowed, we can bound the size of the array to n. This array is not ordered. Let us call this is elements.

   ii) An array of size (n+1) to store the index of the element. In this array, we ~~we~~ shall call position, position[x] will contain the ~~the~~ index of the element x in the elements array. If an element is not added, we shall set its position to -1.

Apart from this, we shall also maintain a size variable, which shall contain the ~~total~~ number of elements in the data structure.

Size reduction : We first remove the last element added (~~somew~~ like a stack) and then update the particular position & decrement the size.

Adding an integer : Add the ~~an~~ integer to the end of the list, but check that the element does not exist using the position array.

The C++ code is :-

```
int size;
int elements[n];
int positions [n+1];
void init()  // resets the arrays
{
    size = 0;
    for(int pos : positions)
        pos = -1;
}

int add(int element)
{
    //check if element exists
    if (positions[element] != -1)
        return 1;        //indicate error

    size ++;
    elements [size] = element;
    positions [element] = size;
    return 0; // indicate success
}

int remove()
{   //check if element exists
    if(size == 0)
        return 1;  //error

    positions [elements [size]]
    = -1;
    size --;
    returns 0;
}
```

As can be seen, both operations happen in $O(1)$ (constant) time }.

Part 2

7) The recurrence relation is:

$$T(n) = \begin{cases} c & , n \le 1 \\ 2 * T(n-2) + d, & n > 1 \end{cases}$$

$$T(n-2) = 2T(n-4) + d$$

$$\therefore T(n) = 2(2T(n-4) + d) + d$$

$$= 4T(n-4) + 3d$$

$$T(n-4) = 2T(n-6) + d$$

$$\therefore T(n) = 4(2T(n-6) + d) + 3d$$

$$= 8T(n-6) + 7d$$

$$\therefore T(n) = 2^x T(n-2x) + (2^x - 1)d$$

$n - 2x = 1$

$2x = n$

$x = n/2$

$$= 2^{n/2} T(1) + (2^{n/2} - 1)d$$

$$= 2^{n/2} + 2^{n/2} d - 1$$

$$= 2^{n/2}(1 + d) - 1$$

Ignoring constant terms, $T(n) = O(2^{n/2})$

2) For n^k to be $O(n)$, $n^k \leq n$

$n^k \leq n$

$k \leq 1$ (We can $\log_n$ ~~a~~ both sides because we know n is +ve)

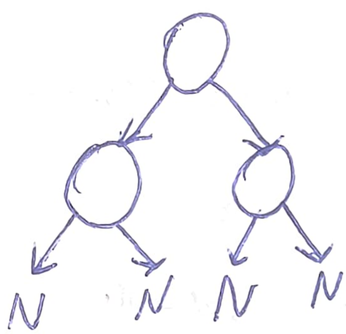
$\therefore 0 \leq k \leq 1$

Hence, for all $k \in [0, 1]$, $n^k = O(n)$

Part 3

1) Proof by induction:

Tree wit one node



number of ~~roots~~ pointers	no. of null pointers
2	2
6	4

(shown)

Proof by mathematics

We know that the ~~top~~ only leaf nodes of a complete binary tree will have NULL pointers.

The number of leaf nodes = $\left\lfloor \frac{n+1}{2} \right\rfloor$ ~~8~~, where n is the total no. of nodes.

~~If we consider to~~

Hence no. of null nodes = $2\left[\left\lfloor \frac{n+1}{2} \right\rfloor\right] = n+1$

Part 4

```
//Assuming Heap is stored in array H. Also, array is global.

modify-key (index, new-key)
{
    H[i] = new-key;                              // O(1)
    ~~if (H[i] - H[⌊(i-1)/2⌋])                   // check against parent~~
    ~~int temp = H[i];~~
    ~~H[i] = H[⌊(i-1)/2⌋]~~
    adjust (i);                                  // O(logn)
}
```

adjust (i) // Adjusts the tree

```
{ if (H[i] < H[⌊(i-2)/2⌋])   // Check against parent
    int temp = H[i];
    H[i] = H[⌊(i-2)/2⌋];
    H[⌊(i-2)/2⌋] = temp;
    adjust(⌊(i-2)/2⌋);
  else:
    return;
}
```

The worst case time complexity is $O(\log n)$ because of the adjust sub-module.

<u>Part 5</u>

i) // Take array to be of size n, array name is A

```
int ~~ba~~ ternary-search (int key, int start, int end)
{
    int one_third = (end-start)/3 + start;
    int two_third = 2*(end-start)/3 + start;
    if (A[one_third] == key) return one_third;
```

```
else if (A[two_third] == key) return two_third;
else if (A[one_third] > key)
    return ternary_search(key, start, one_third)
  else if (A[two_third] > key)
      return ternary_search(key, one_third, two_third);
  else
     return ternary_search(key, two_third, end);

}
```

~~2) (Z+ Pseudo code:~~

~~{~~

~~while (curr_s != null && curr_t != null)~~

~~// hashmap.add~~

2) Algorithm

1) Create a hashmap to store element (key) & frequency

2) Create a while loop which runs until either one of the list has been traversed in full.

a) ~~Ele~~ ~~a~~ Add the element from S into the hashmap

i) If the element is already in the hash map, increment the frequency.

ii) If the element is not in the hashmap, add it ~~and~~ ~~is~~ with frequency = 1.

b) "Remove the element" from ~~S~~ T ~~into~~ from the hash map

i) If the element exists in the hash map, decrement its frequency.

ii) Otherwise, add the element to the hashmap with frequency = -1;

iii) If the frequency after decrement is 0, delete the element from the hash map

c) Increment the ~~pos~~ S_curr & T_curr pointers.

3) If ~~either~~ one of the lists has finished traversing, but the other has not, return 0 (not equal), because one has more elements than the other.

4) Check ~~of~~ the size (no of key-value pairs) on the hashmap

i) If no elements are remaining, S = T.

ii) Otherwise S ≠ T.

~~Part 6~~

~~1) a)~~

Part 7

~~1) Because A[1] < A[2] < ... < A[n], we know that A is a sorted one-dimensional array. For such an array, we can use binary search. The steps of the algorithms are:~~

~~1) Identify the start and end indexes of the array. Initially, they shall be 1 & n respectively.~~

~~2) Identify the midpoint : $\left\lfloor \frac{start+end}{2} \right\rfloor$~~

Part 7

1) Because the array is sorted in ascending orders we may use a modified version of binary search. The steps of the algorithms are as follows :

   1) Identify the start & end index of the search space. We shall be halving ~~the~~ the dimension of the search space in each iteration. Initially start = 1 and end = n.

   2) Identify the midpoint of start & end : $m = \left\lfloor \frac{end + start}{2} \right\rfloor$

   3) Check the element at index m
      a) If $A[m] = m$, we have found the answer.
      b) Else if $A[m] > m$, we need to search between start and m. Hence update s.t. end = m.
      c) Else, we know to search between m & end. Hence update ~~ea~~ start = m.

   4) Continue steps 2 & 3 until either the element is found or start = end.

Because we are halving the interval, the TC is $O(\log n)$

2) Algorithm:

1) Perform a first BFS from the root. Obtain the node that ~~that the~~ is the furthest from the root.

2) Perform a second BFS from the node found in step 1.

3) The distance between the node found in step 1 and the furthest node in the 2 BFS from step 2 will be the diameter.

Pseudocode:

Let n = no. of vertices & g = adjacent representation of tree.
& root = root of tree.

if (n ≤ 1): return 0;

u = BFS(root, 0)
diameter = BFS(u, 1)
print diameter

// code for BFS

~~function~~ BFS(start, return_dist)

```
queue q; dist[n] = {-1 * n};

q.enqueue(start)
dist[start] = 0   // dist is an array to contain distances from start
max_dist = 0
farthest = start

while queue is not empty:
    u = q.dequeue()

    for each neighbour v of u:
        if dist[v] == -1: // unvisited node
            q.enqueue(v);
            dist[v] = dist[u] + 1

            // Update farthest node
            if dist[v] > max_dist:
                max_dist = dist[v]
                farthest = v

if (return_dist) return max_dist;
else             return farthest;
```