Key 1

INSTITUTE OF AERONAUTICAL ENGINEERING

(**AUTONOMOUS**)

Code No: BCSB02

M. Tech I Semester Regular Examinations
ADVANCED DATA STRUCTURES
(Computer Science and Engineering)

Time: 3 hours Max. Marks: 70

****

1 a) Recursion and iteration both repeatedly executes the set of instructions. Recursion is when a statement in a function calls itself repeatedly. The iteration is when a loop repeatedly executes till the controlling condition becomes false. The primary difference between recursion and iteration is that is a **recursion** is a process, always applied to a function. The **iteration** is applied to the set of instructions which we want to get repeatedly executed.

b)

```
while(currNode!=null){
    nextNode = currNode.next;
    currNode.next = prevNode;
    prevNode = currNode;
    currNode = nextNode;
}
```

2 a)

Asymptotic notations are mathematical tools to represent time complexity of algorithms for asymptotic analysis. The following 3 asymptotic notations are mostly used to represent time complexity of algorithms.

1) **Θ Notation:** The theta notation bounds a functions from above and below, so it defines exact asymptotic behavior.
2) **Big O Notation:** The Big O notation defines an upper bound of an algorithm, it bounds a function only from above.

Ω Notation: Just as Big O notation provides an asymptotic upper bound on a function, Ω notation provides an asymptotic lower bound.

b)

i) PREFIX + - + A * B C / % D E F / G H

ii) PREFIX + - + % A B * / C D E F * G H

3 a)

Suppose we want to design a system for storing employee records keyed using phone numbers. And we want following queries to be performed efficiently:

1. Insert a phone number and corresponding information.
2. Search a phone number and fetch the information.
3. Delete a phone number and related information.

We can think of using the following data structures to maintain information about different phone numbers.

1. Array of phone numbers and records.
2. Linked List of phone numbers and records.
3. Balanced binary search tree with phone numbers as keys.
4. Direct Access Table.
5.

Hashing is an improvement over Direct Access Table. The idea is to use hash function that converts a given phone number or any other key to a smaller number and uses the small number as index in a table called hash table.

b)

In **linear probing**, we linearly probe for next slot. For example, typical gap between two probes is 1 as taken in below example also.

Quadratic probing is an open addressing scheme in computer programming for resolving collisions in hash tables—when an incoming data's hash value indicates it should be stored in an already-occupied slot or bucket. Quadratic probing operates by taking the original hash index and adding successive values of an arbitrary quadratic polynomial until an open slot is found.

Double hashing is a computer programming technique used in **hash** tables to resolve **hash**collisions, in cases when two different values to be searched for produce the same **hash** key. It is a popular collision-resolution technique in open-addressed **hash** tables.

4

Since a hash function gets us a small number for a key which is a big integer or string, there is possibility that two keys result in same value. The situation where a newly inserted key maps to an already occupied slot in hash table is called collision and must be handled using some collision handling technique.

Solution:

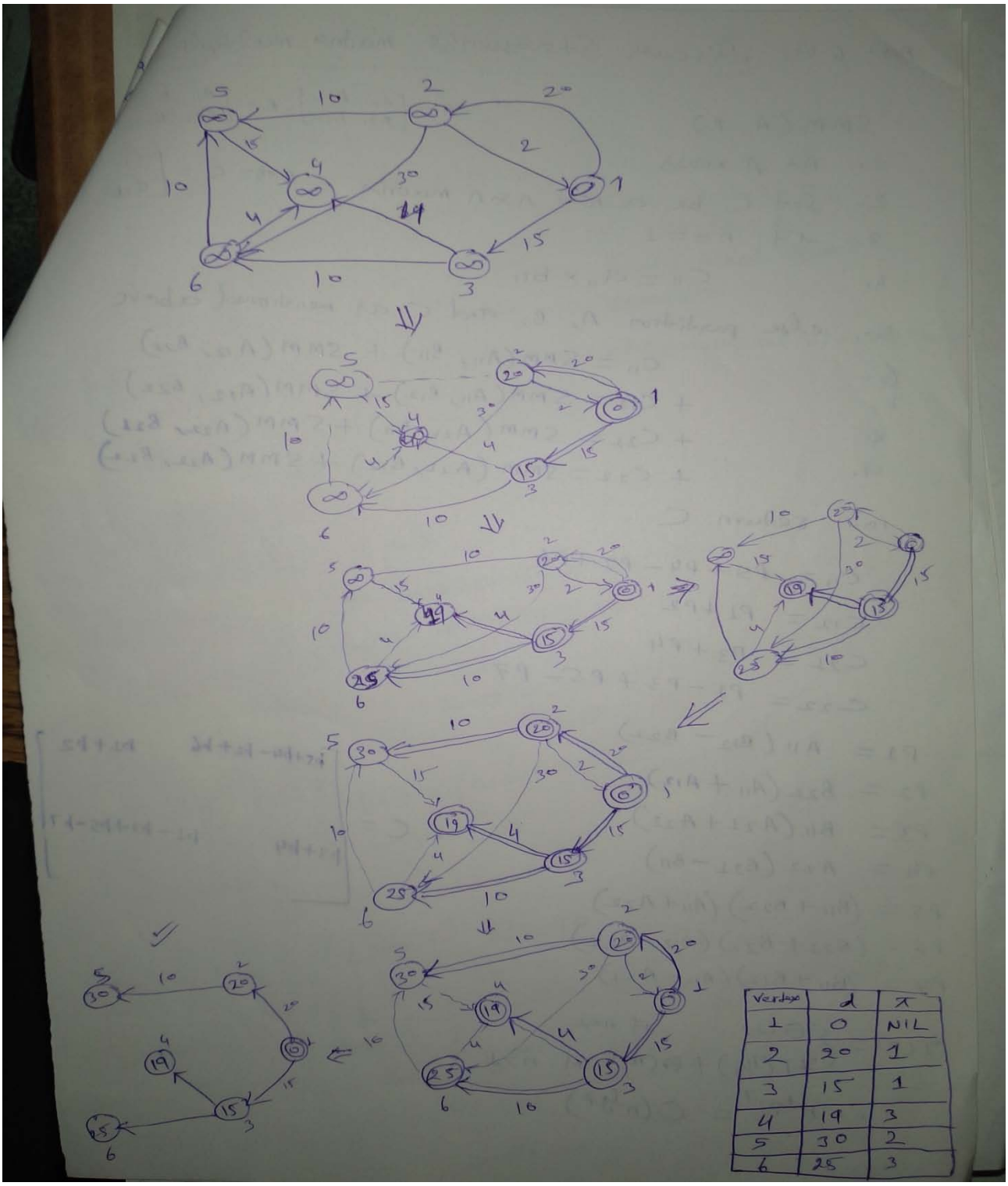
Keys 12, 18, 13, 2, 3, 23, 5 and 15 are inserted in hash table as:

For key 12, h(12) is 12%10 = 2. Therefore, 12 is placed at 2nd index in the hash table.
For key 18, h(18) is 18%10 = 8. Therefore, 18 is placed at 8th index in the hash table.
For key 13, h(13) is 13%10 = 3. Therefore, 13 is placed at 3rd index in the hash table.
For key 2, h(2) is 2%10 = 2. However, index 2 is already occupied with 12. Therefore, using linear probing, 2 will be placed at index 4 as index 2 and 3 are already occupied.
For key 3, h(3) is 3%10 = 3. However, index 3 is already occupied with 13. Therefore, using linear probing, 3 will be placed at index 5 as index 3 and 4 are already occupied.
Similarly, 23, 5 and 15 will be placed at index 6, 7, 9 respectively.

Correct result:

0	
1	
2	12
3	13
4	2
5	3
6	23
7	5
8	18
9	15

5 a)



b)

```
Input :
in[]   = {4, 8, 2, 5, 1, 6, 3, 7}
post[] = {8, 4, 5, 2, 6, 7, 3, 1}

Output : Root of below tree
          1
        /   \
      2       3
    /   \   /   \
  4      5  6    7
    \
      8
```

6 a)

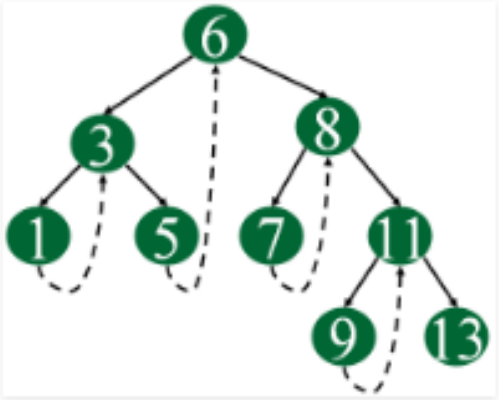
The answer is: ABDCEGHF

b)

A **threaded binary tree** defined as follows: "A**binary tree** is **threaded** by making all right child pointers that would normally be null point to the inorder successor of the node (if it exists), and all left child pointers that would normally be null point to the inorder predecessor of the node."

There are two types of threaded binary trees.
Single Threaded: Where a NULL right pointers is made to point to the inorder successor (if successor exists)
Double Threaded: Where both left and right NULL pointers are made to point to inorder predecessor and inorder successor respectively. The predecessor threads are useful for reverse inorder traversal and postorder traversal.
The threads are also useful for fast accessing ancestors of a node.

Following diagram shows an example Single Threaded Binary Tree. The dotted lines represent threads.



7 a)

```
deleteNode(struct node* root, int key)
{
   // base case
   if (root == NULL) return root;

   // If the key to be deleted is smaller than the root's key,
   // then it lies in left subtree
   if (key < root->key)
      root->left = deleteNode(root->left, key);

   // If the key to be deleted is greater than the root's key,
   // then it lies in right subtree
   else if (key > root->key)
      root->right = deleteNode(root->right, key);

   // if key is same as root's key, then This is the node
   // to be deleted
   else
   {
      // node with only one child or no child
      if (root->left == NULL)
      {
         struct node *temp = root->right;
         free(root);
         return temp;
      }
      else if (root->right == NULL)
      {
         struct node *temp = root->left;
         free(root);
         return temp;
      }

      // node with two children: Get the inorder successor (smallest
      // in the right subtree)
      struct node* temp = minValueNode(root->right);
root->key = temp->key;

      // Delete the inorder successor
      root->right = deleteNode(root->right, temp->key);
   }
   return root;
}
```

b)

```
constructTrees(int start, int end)
{
   vector<struct node *> list;

   /*  if start > end   then subtree will be empty so returning NULL
      in the list */
   if (start > end)
   {
      list.push_back(NULL);
      return list;
   }

   /*  iterating through all values from start to end  for constructing\
      left and right subtree recursively  */
   for (int i = start; i <= end; i++)
   {
      /*  constructing left subtree   */
      vector<struct node *> leftSubtree  = constructTrees(start, i - 1);

      /*  constructing right subtree  */
      vector<struct node *> rightSubtree = constructTrees(i + 1, end);

      /*  now looping through all left and right subtrees and connecting
         them to ith root  below  */
      for (int j = 0; j < leftSubtree.size(); j++)
      {
         struct node* left = leftSubtree[j];
         for (int k = 0; k < rightSubtree.size(); k++)
         {
            struct node * right = rightSubtree[k];
            struct node * node = newNode(i);// making value i as root
            node->left = left;              // connect left subtree
            node->right = right;            // connect right subtree
            list.push_back(node);           // add this tree to list
         }
      }
                  }

   return list;
}
```

8)

a)

Steps to follow for insertion

Let the newly inserted node be w

1) Perform standard BST insert for w.

2) Starting from w, travel up and find the first unbalanced node. Let z be the first unbalanced node, y be the child of z that comes on the path from w to z and x be the grandchild of z that comes on the path from w to z.

3) Re-balance the tree by performing appropriate rotations on the subtree rooted with z. There can be 4 possible cases that needs to be handled as x, y and z can be arranged in 4 ways. Following are the possible 4 arrangements:

a) y is left child of z and x is left child of y (Left Left Case)

b) y is left child of z and x is right child of y (Left Right Case)

c) y is right child of z and x is right child of y (Right Right Case)

d) y is right child of z and x is left child of y (Right Left Case)

Following are the operations to be performed in above mentioned 4 cases. In all of the cases, we only need to re-balance the subtree rooted with z and the complete tree becomes balanced as the height of subtree (After appropriate rotations) rooted with z becomes same as it was before insertion.

b)

Why do Binary Search Trees Have to be Balanced?

So why do binary search trees have to be balanced? I think the best way to understand the importance is to walk through a base case. And remember that the key reason why a BST offers such great performance is because it allows us to ignore irrelevant values. Thus decreasing the number of comparisons a program has to perform to find a data element.

This may not seem like a huge deal for such a small data set. However what if you had millions of elements that you had to search for? The performance gains of a BST are quite significant, which is one of the reasons why binary search trees are considered such a vital tool for computer scientists.

9 a)

Why Red-Black Trees?

Most of the BST operations (e.g., search, max, min, insert, delete.. etc) take O(h) time where h is the height of the BST. The cost of these operations may become O(n) for a skewed Binary tree. If we make sure that height of the tree remains O(Logn) after every insertion and deletion, then we can guarantee an upper bound of O(Logn) for all these operations. The height of a Red-Black tree is always O(Logn) where n is the number of nodes in the tree.

Comparison with AVL Tree

The AVL trees are more balanced compared to Red-Black Trees, but they may cause more rotations during insertion and deletion. So if your application involves many frequent insertions and deletions, then Red Black trees should be preferred. And if the insertions and deletions are less frequent and search is a more frequent operation, then AVL tree should be preferred over Red-Black Tree.

b)

Properties of B-Tree

1) All leaves are at same level.
2) A B-Tree is defined by the term *minimum degree* 't'. The value of t depends upon disk block size.
3) Every node except root must contain at least t-1 keys. Root may contain minimum 1 key.
4) All nodes (including root) may contain at most 2t – 1 keys.
5) Number of children of a node is equal to the number of keys in it plus 1.
6) All keys of a node are sorted in increasing order. The child between two keys k1 and k2 contains all keys in the range from k1 and k2.
7) B-Tree grows and shrinks from the root which is unlike Binary Search Tree. Binary Search Trees grow downward and also shrink from downward.
8) Like other balanced Binary Search Trees, time complexity to search, insert and delete is O(Logn).

Insertion

1) Initialize x as root.
2) While x is not leaf, do following
..**a)** Find the child of x that is going to to be traversed next. Let the child be y.
..**b)** If y is not full, change x to point to y.
..**c)** If y is full, split it and change x to point to one of the two parts of y. If k is smaller than mid key in y, then set x as first part of y. Else second part of y. When we split y, we move a key from y to its parent x.
3) The loop in step 2 stops when x is leaf. x must have space for 1 extra key as we have been splitting all nodes in advance. So simply insert k to x.

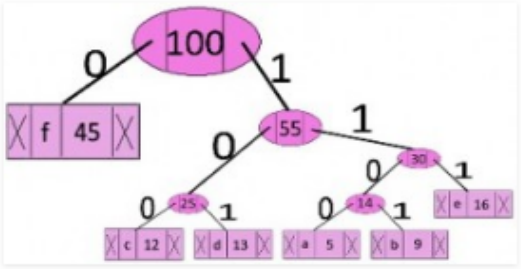
10 a)

Steps to build Huffman Tree

Input is array of unique characters along with their frequency of occurrences and output is Huffman Tree.
1. Create a leaf node for each unique character and build a min heap of all leaf nodes (Min Heap is used as a priority queue. The value of frequency field is used to compare two nodes in min heap. Initially, the least frequent character is at root)
2. Extract two nodes with the minimum frequency from the min heap.
3. Create a new internal node with frequency equal to the sum of the two nodes frequencies. Make the first extracted node as its left child and the other extracted node as its right child. Add this node to the min heap.
4. Repeat steps#2 and #3 until the heap contains only one node. The remaining node is the root node and the tree is complete.

Steps to print codes from Huffman Tree:

Traverse the tree formed starting from the root. Maintain an auxiliary array. While moving to the left child, write 0 to the array. While moving to the right child, write 1 to the array. Print the array when a leaf node is encountered.



The codes are as follows:

```
character   code-word
    f          0
    c          100
    d          101
    a          1100
    b          1101
    e          111
```

b)

algorithm *kmp_search*:
  input:
    an array of characters, S (the text to be searched)
    an array of characters, W (the word sought)
  output:
    an array of integers, P (positions in S at which W is found)
    an integer, nP (number of positions)

  define variables:
    an integer, j ← 0 (the position of the current character in S)
    an integer, k ← 0 (the position of the current character in W)
    an array of integers, T (the table, computed elsewhere)

  let nP ← 0

  while j < length(S) **do**
    if W[k] = S[j] **then**
      let j ← j + 1
      let k ← k + 1
      if k = length(W) **then**
        (occurrence found, if only first occurrence is needed, m← j - k may be returned here)
        let P[nP] ← j - k, nP ← nP + 1
        let k ← T[k] (T[length(W)] can't be -1)
    else
      let k ← T[k]
      if k < 0 **then**

```
    let j ← j + 1
    let k ← k + 1
```