Dang Nhat Huy - ITITIU20043

DSA FINAL 2017 - 2018

1. Binary tree

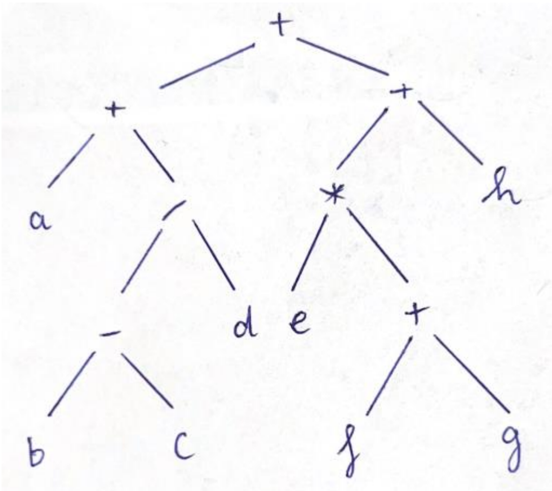
Given an expression: a+(b-c)/d+e*(f+g)+h.

a. Present the expression by a binary tree.

b. Write the expression in postfix format.

<u>Answer:</u>

a) a+(b-c)/d+e*(f+g)+h



b) Postfix format: abc-d/+efg+*h++

2. Hash table

Given a hash table of size 7 where the open addressing procedure is used to solve collisions

a. For a list of items {0, 1, 6, 14, 27, 25}, write the hash function for the hash table, insert the items into the hash table and show the hash table.

b. Change the hash table's size to 11. Show the new hash table.

<u>Answer:</u>

a. Hash function: h(key) = key % table_size

Initialize hash table: [_, _, _, _, _, _, _]

1. Insert 0: h(0) = 0 % 7 = 0

   Since the slot at index 0 is empty, we insert 0 there.

   → Hash Table: [0, _, _, _, _, _, _]

2. Insert 1: h(1) = 1 % 7 = 1

   Since the slot at index 1 is empty, we insert 1 there.

   → Hash Table: [0, 1, _, _, _, _, _]

3. Insert 6: h(6) = 6 % 7 = 6

   Since the slot at index 6 is empty, we insert 6 here

   → Hash Table: [0, 1, _, _, _, _, 6]

4. Insert 14: h(14) = 14 % 7 = 0

   Collision! The slot at index 0 is already occupied by 0

   We probe to the next slot by linear probing: h(14) = (0 + 1) % 7 = 1

   Collision! The slot at index 1 is already occupied by 1

   We probe to the next slot by linear probing: h(14) = (0 + 2) % 7 = 2

   Since the slot at index 2 is empty, we insert 14 here

   → Hash Table: [0, 1, 14, _, _, _, 6]

5. Insert 27: h(27) = 27 % 7 = 6

   Collision! The slot at index 6 is already occupied by 6

   We probe to the next slot by linear probing: h(27) = (6 + 1) % 7 = 0

   Collision! The slot at index 0 is already occupied by 0

   We probe to the next slot by linear probing: h(27) = (0 + 1) % 7 = 1

   Collision! The slot at index 1 is already occupied by 1

   We probe to the next slot by linear probing: h(27) = (0 + 2) % 7 = 2

   Collision! The slot at index 1 is already occupied by 14

   We probe to the next slot by linear probing: h(27) = (0 + 3) % 7 = 3

   Since the slot at index 3 is empty, we insert 14 here

   → Hash Table: [0, 1, 14, 27, _, _, 6]

6. Insert 25: h(25) = 25 % 7 = 4

   Since the slot at index 4 is empty, we insert 25 here

   → Hash Table: [0, 1, 14, 27, 25, _, 6]

b. Hash function: h(key) = key % table_size

Initialize hast table: [_, _, _, _, _, _, _, _, _, _, _]

1. Insert 0: h(0) = 0 % 11 = 0

   Since the slot at index 0 is empty, we insert 0 here

   → Hash Table: [0, _, _, _, _, _, _, _, _, _, _]

2. Insert 1: h(1) = 1 % 11 = 1

   Since the slot at index 1 is empty, we insert 1 here

   → Hash Table: [0, 1, _, _, _, _, _, _, _, _, _]

3. Insert 6: h(6) = 6 % 11 = 6

   Since the slot at index 6 is empty, we insert 6 here

   → Hash Table: [0, 1, _, _, _, _, 6, _, _, _, _]

4. Insert 14: h(14) = 14 % 11 = 3

   Since the slot at index 3 is empty, we insert 14 here

   → Hash Table: [0, 1, _, 14, _, _, 6, _, _, _, _]

5. Insert 27: h(27) = 27 % 11 = 5

   Since the slot at index 5 is empty, we insert 27 here

   → Hash Table: [0, 1, _, 14, _, 27, 6, _, _, _, _]

6. Insert 25: h(25) = 25 % 11 = 3

   Collision! The slot at index 3 is already occupied by 14

   We probe to the next slot by linear probing: h(27) = (3 + 1) % 7 = 4

   Since the slot at index 4 is empty, we insert 25 here

   → Hash Table: [0, 1, _, 14, 25, 27, 6, _, _, _, _]

3. Give a graph G represented by the following list of successors.

For each pair (x, y), x is the weight of the edge, y is the terminal extremity of the edge.

A: (5, B)

B: (3, F)

C: (1, A) → (3, D) → (2, F)

D: (9, C)

E: (4, B)

F: (2, E) → (9, D) → (3, G)

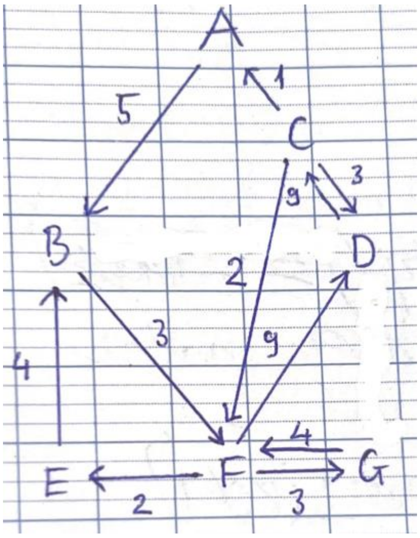
G: (4, F)

i. Draw the graph

ii: Show an adjacency matrix of the graph

<u>Answer:</u>

i.



ii.

	A	B	C	D	E	F	G
A	0	5	0	0	0	0	0
B	0	0	0	0	0	3	0
C	1	0	0	3	0	2	0
D	0	0	9	0	0	0	0
E	0	4	0	0	0	0	0
F	0	0	0	9	2	0	3
G	0	0	0	0	0	4	0

4. In the graph G, use Dijkstra algorithm to compute shortest paths from A to all other nodes. Fill the following table with correspoding values after each step of the algorithm

Node	A	B	C	D	E	F	G
-	**(0, A)**	∞	∞	∞	∞	∞	∞
A	-	**(5, A)**	∞	∞	∞	∞	∞
B	-	-	∞	∞	∞	**(8, B)**	∞
F	-	-	**(10, F)**	(17, F)	(10, F)	-	(11, F)
C	-	-	-	(17, F)	**(10, F)**	-	(11, F)
E	-	-	-	(17, F)	-	-	**(11, F)**
G	-	-	-	**(17, F)**	-	-	-
D	-	-	-	-	-	-	-

5. Rooted trees with unbounded branching

Given a tree whose node may have arbitrary numbers of children. There is a schema to represent that kind of tree name left-child, right-sibling. Each node x contains a parent pointer p, and two other pointers:

- left-child[x] points to the leftmost child of node x, and

- right-sibling[x] points to the sibling of x immediately to the right.

If node x has no children, the left-child[x] = NULL, and if node x is the rightmost child of its parent, then right-sibling[x] = NULL.

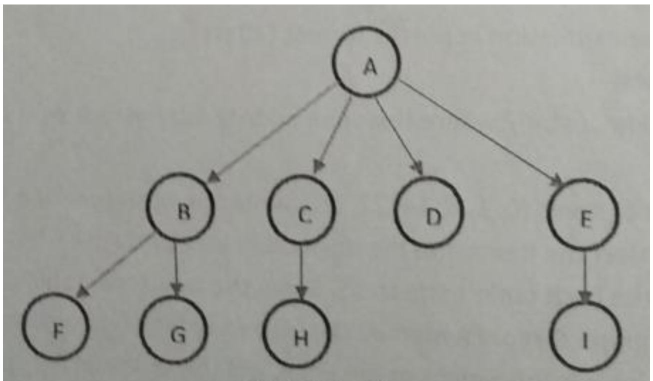


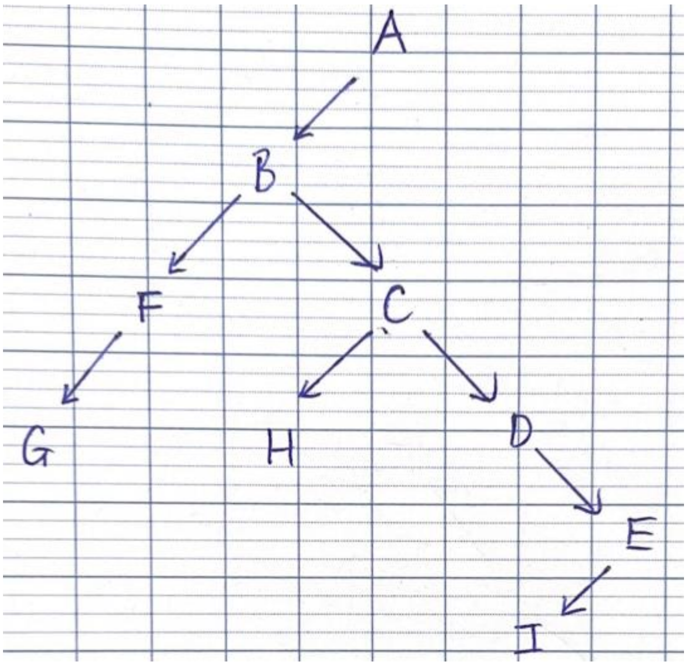
Figure 1 - Tree T

i. Draw a left-child, right-sibling representation of the tree T in figure 1.

ii. Write an O(n) non-recursive procedure that prints all the keys of an arbitrary rooted tree with n nodes, where the tree is stored using the left-child, right-sibling representation.

<u>Answer:</u>

i.



ii. Pseudocode:

```
Traverse(node):
      Stack container = new Stack();
      // or Queue container = new Queue();

        while (!container.isEmpty()) {
            // Push left, move right
            container.push(node.left);
            print(node.value);
            node = node.right;
```

```
            // Or push right, move left
            container.push(node.right);
            print(node.value);
            node = node.left;

            // The node is leaf
            if (node.left == node.right == null) {
                  node = container.pop();
                  // Or node = container.dequeue();
            }
        } // End while loop
// End algorithm
```

Sample code:

```
class Node {
    int key;
    Node parent;
    Node leftChild;
    Node rightSibling;

    Node(int key) {
        this.key = key;
        this.parent = null;
        this.leftChild = null;
        this.rightSibling = null;
    }
}

public class TreeTraversal {
    public static void printTreeKeys(Node root) {
        if (root == null) {
            return;
        }

        Node current = root;

        while (current != null) {
            System.out.println(current.key); // Print the key of the current node

            if (current.leftChild != null) {
                current = current.leftChild;
            } else if (current.rightSibling != null) {
                current = current.rightSibling;
            } else {
                // Move up the tree until finding a node with a right sibling or
reaching the root
                while (current.parent != null && current.rightSibling == null) {
                    current = current.parent;
                }
                if (current.parent == null) {
                    break; // Reached the root and finished traversal
                }
                current = current.rightSibling;
            }
        }
    }

    public static void main(String[] args) {
        // Create the tree structure
        Node root = new Node(1);
        Node node2 = new Node(2);
        Node node3 = new Node(3);
        Node node4 = new Node(4);
        Node node5 = new Node(5);
```

```java
        Node node6 = new Node(6);
        Node node7 = new Node(7);

        root.leftChild = node2;
        node2.parent = root;
        node2.rightSibling = node3;
        node3.parent = root;
        node3.rightSibling = node4;
        node4.parent = root;
        node4.leftChild = node5;
        node5.parent = node4;
        node5.rightSibling = node6;
        node6.parent = node4;
        node6.rightSibling = node7;
        node7.parent = node4;

        // Call the printTreeKeys method
        printTreeKeys(root);
    }
}
```