Data Structures and Algorithms

Assignment 2(PART 1)

Jan 1 Semester

Name: Jeslyn Ho Ka Yan:

ID: 8535383

Date of Submission: 15 Feb 2024

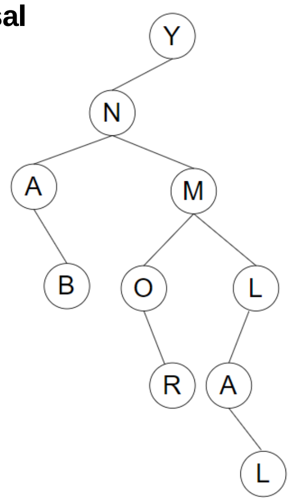
Table of Content

1 Question 1

1.1 INORDER, PREORDER, POSTORDER Traversal

INORDER: A B N O R M A L L Y
PREORDER: Y N A B M O R L A L

POSTORDER: B A R O L A L M N Y

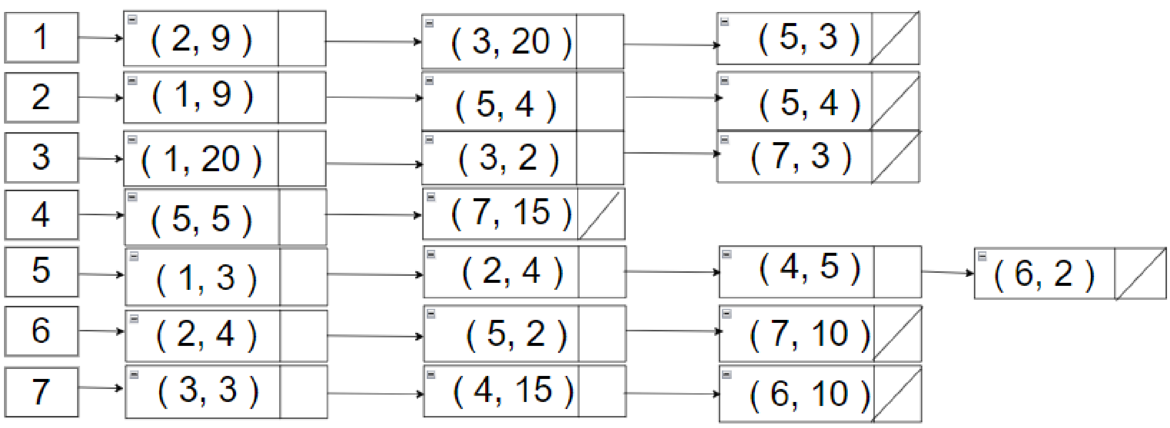


1.2 Graph Representaion

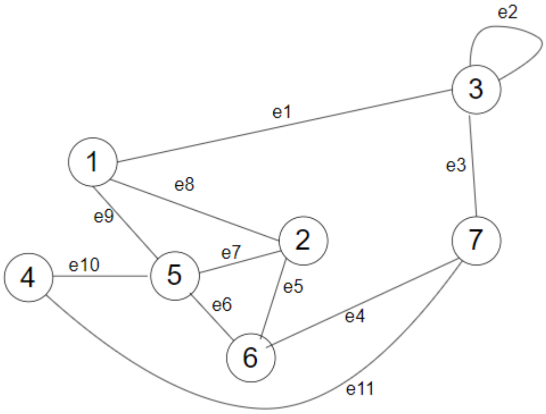
1.2.1 Adjacency Matrix

A	1	2	3	4	5	6	7
1	∞	9	20	∞	3	∞	∞
2	9	∞	∞	∞	4	4	∞
3	20	∞	2	∞	∞	∞	3
4	∞	∞	∞	∞	5	∞	15
5	3	4	∞	5	∞	2	∞
6	∞	4	∞	∞	2	∞	10
7	∞	∞	3	15	∞	10	∞

1.2.2 Adjacency List

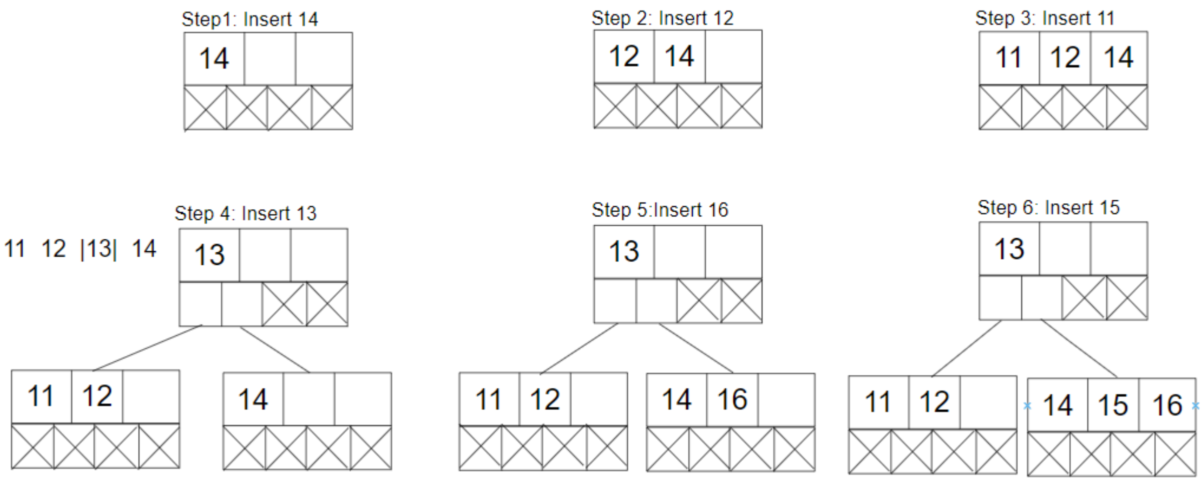


1.2.3 Incidence Matrix



M:	E1	E2	E3	E4	E5	E6	E7	E8	E9	E10	E11
1	20	∞	∞	∞	∞	∞	∞	9	3	∞	∞
2	∞	∞	∞	∞	4	∞	4	9	∞	∞	∞
3	20	2	3	∞	∞	∞	∞	∞	∞	∞	∞
4	∞	∞	∞	∞	∞	∞	∞	∞	∞	5	15
5	∞	∞	∞	∞	∞	2	4	∞	3	5	∞
6	∞	∞	∞	10	4	2	∞	∞	∞	∞	∞
7	∞	∞	3	10	∞	∞	∞	∞	∞	∞	15

1.3 2-4 Tree



2 Question 2

2.1 Explain and State the asymptotic run-time complexity of Two Algorithms

Algorithm 1

Purpose:

Algorithm 1, is design to find the maximum value of the tree by using level order traversal method. It’s a method that will visit all nodes in a level, before procceding to visit the nodes in the next level. After the queue has been set up, it will then explore the binary tree repeatedly and update the varaible call ‘x’ which is the maximum value, when a highest value is found.

Asymptotic Run-time Complexity: O(n)

Algorithm 2

Purpose:

Algorithm 2, is design to find and retrieve infromation from the rightmost non null node in the tree. The algorithm will first verfiy for null root, if the root is null, then the method will return -1. Otherwise, it will then moves to the right child of each node as it traverses the tree. The procedure keeps going until it gets to the last non-null node on the right, at which point it returns that node's information.

Asymptotic Run-time Complexity: O(Log n)

2.2 A1(root)

The Function call A1 (root) will **return ‘89’** for the following binary search tree.

Explantion:

The algorithm will first initialize ‘x’ as ‘32’ as in it is the staring root. The root is then enqueued, and the while loop is initiated. The root is then dequeued, x has been updated to 32, and the right and left children ‘63’ and ‘27’ are enqueued.

The algorithm will proceed to dequeue the nodes in the subsequent iteration. It will update x to the maxiumum number It can find. ‘89’ is the highest number in the binary search tree, it will be the final alue of x.

2.3 Run-time complexity of Algorithm 2

ALGORITHM 2	
Function A2(root)	
$if\ (root == NULL)$	1
$return\ -1$	1
$endIf$	
$t = root$	1
$while\ (t.right\ != NULL)\ do$	Log n
$t = t.right$	n
$endWhile$	
$return\ t.data$	1
End of function A2	

T(n)= 4+n+logn

Asymptotic Run-time Complexity: O(log n)

2.4 Function A2 (pseudocode)

```
Function A2(root)
   if root == NULL
      return NULL

   rightNode = A2(root.right)
   if rightNode == NULL
      return root.data
   else
      return A2(rightNode)

End of function A2
```

2.5 Which Algorithm will you choose?

The superior temporal complexity of **Algorithm A2** makes it the recommended choice for broad use in a BST. By just traversing the right-child pointers until it reaches the rightmost leaf, which represents the maximum value in a BST, it is able to accomplish this efficiency. Alogrithm 1 may have higher time compexity than Alogrithm 2 as it involves with unecessary operation like visitng every nodes in each level of the entire tree.

3 Question 3

3.1 Main Nodes

```
class MainNode {
   int date;              // 0 for even list, 1 for odd list
   SecondaryNode nextNode;
   MainNode nextList;

   MainNode(int d, SecondaryNode nextN, MainNode nextL) {     //constructor
      this.date = d;
      this.nextNode = nextN;
      this.nextList = nextL;
   }
}
```

3.2 Function INSERT (head,x)

```
INSERT(head, int x)
   SecondaryNode newSecondaryNode = new SecondaryNode(x, null);

   MainNode mainNode = new MainNode(x % 2, newSecondaryNode, null);

   if head is null
      head = mainNode;                              // If the main list is empty, set head to mainNode
    else
      mainNode.nextL = head.nextL;
      head.nextL = mainNode;

      if x is even
         newSecondaryNode.nextN = head.nextN;
         head.nextN = newSecondaryNode;
      else
         newSecondaryNode.nextN = head.nextN.nextN;
         head.nextN.nextN = newSecondaryNode;

   return head;
```

3.3 Function SEARCH (head,x)

```
SEARCH(head, int x)
   MainNode currentMainNode = head;

   while (currentMainNode != null)
      if (currentMainNode.data is (x % 2))
         SecondaryNode currentSecondaryNode = currentMainNode.nextN;

         while (currentSecondaryNode != null && currentSecondaryNode.data != x)
            currentSecondaryNode = currentSecondaryNode.nextN;

         while (currentSecondaryNode != null)
            return true;
      currentMainNode = currentMainNode.nextL;

   return false
```

3.4 Function DELETE(head,b)

```
DELETE(head, int b)
   MainNode currentMainNode = head;

   while (currentMainNode != null)
      if (currentMainNode.data is b)
         MainNode tempMainNode = currentMainNode;
         head = currentMainNode.nextL;
         // Assuming Java's garbage collector handles delete operations
        return head

      MainNode prevMainNode is null;
      while (currentMainNode != null && currentMainNode.data != b)
         prevMainNode = currentMainN
         currentMainNode = currentMainNode.nextL

      if (currentMainNode != null)
         prevMainNode.nextL = currentMainNode.nextL;
         // Assuming Java's garbage collector handles delete operations
         return head;

   return head;
```