Assignment-5

Rahul Motan

1)

# Heap

The (binary) heap data structure is an array object that we can view as a nearly complete binary tree. Each node of the tree corresponds to an element of the array. An array A that represents a heap is an object with two attributes:

$A.length$, which (as usual) gives the number of elements in the array, and $A.heap−size$, which represents how many elements in the heap are stored within array A.

Root : $A\left[1\right]$

Left(i) : $2i$

Right (i): $2i+1$

Parent (i):   $⌊\frac{i}{2}⌋$

# Priority Queue

Priority queues are implemented in such a way that they are dequeued according to the priority of the elements.

Operations supported on a priority queue are:

1. addItem(item, priority)

2. getItemWithHighestPriority()

3. changePriority(item, new\_priority)

To change priority of an element in our priority queue we need to look it up in less than O(n) time and we can do so by maintaining another data structure such as AVL Tree which allows Search operation in $O\left(logn\right)$ time such that each node of the AVL Tree has {item, index}.

Index will point to the index of the item in the array which is our priority queue. In this way when we have to look for an item to change its priority we can look it up in the AVL Tree to know its index in the array.

Also, in our priority queue each node will additionally have a pointer to its corresponding element in the AVL Tree so that once we change the priority of a particular element we can have maximum of logn elements changed in the priority queue and we can update their respective indices in the AVL Tree in O(1) time since we’ll be having their direct address and we can make the change.

$addItem\left(item, priority\right)${

1.  Extend Array by one element

2. pos$\leftarrow $Put the element at the element with the missing leaf.(end of the array)

3. if( priority < lastElement.priority){         // last element in array before adding the new element

                        save memory address in pointer  of A[pos]$\leftarrow $AVL.insert(item, pos)     //AVL.insert returns the address of the node inserted

        } else {

4.            while(pos > 1 and A[*Parent*(pos)].priority < A[pos].priority){                         // O(log n)

                            exchange A[pos] with A[*Parent*(pos)]

                            Update AVL Tree for A[pos] and A[*Parent*(pos)]             //search for A[pos] and update its index.

                                                                                                                             //search for A[Parent(pos)] & update its index.

5.                        pos = *Parent*(pos)

                  }

                   save memory address in pointer  of A[pos]$\leftarrow $AVL.insert(item, pos) //AVL.insert returns the address of the node inserted

         }

**Time Complexity Analysis:**Time complexity will be O(log n)

**Space Complexity Analysis:**O(1) as we are creating the AVL tree before-hand.

$getItemWithHighestPriority\left(\right)${

1.  if A.heap-size < 1

            **error**“heap underflow”

2.   max = A[1]

3.   A[1] = A[A.heap-size]

4.   A.heap-size = A.heap-size - 1

5.   Remove <item, index> from AVL for max.  //  O(log n)

6.   Max-Heapify(A, 1)     //  O(log n)

7.    **return** *max*

**Time Complexity Analysis:**Worst case could take maximum of O(log n) time.

**Space Complexity Analysis:**Max heapify may take log n  space to save log n recursivecalls to itself. Thus O(log n)

$changePriority\left(item, newPriority\right)$*{*

*1.    pos*$\leftarrow $ AVL.search(item)    // look-up reduced from O(n) to O(logn)

2.   swap A[pos] with A[size] // swap with last element

3.  Update AVL Tree for A[pos] //swapped element has changed its index position so need to update AVL tree

4.   remove A[size] from Priority Queue

5.   AVL.remove(item)

6.   addItem(item, newPriority)

**Time Complexity Analysis:**

**Space Complexity Analysis:**

    $Max−Heapify\left(A, i\right)${

1.  l = Left(i)

2.  r = Right(i)

3.  $if l\leq A.heap−size and A\left[l\right] > A\left[i\right]$

4.         largest = l

5.  $else$ largest = i

6.   $if r\leq A.heap−size and A\left[r\right] > A\left[i\right]$

            largest = r

7.   $if largest \ne  i$

8.            exchange $A\left[i\right] with A\left[largest\right]$

9.            Update item-index pair in BST for A[i] and A[largest] using pointer\* // done in constant time

10.         $Max−Heapify\left(A, largest\right)$

**Time Complexity Analysis: O(log n)**

**Space Complexity Analysis: O(log n)** Max recursive calls to be saved in stack can be the height of the heap which is log n

2) a.  Given a BST ‘T’ we can create a balanced tree T’ without using balancing mechanisms by following the below given steps:-

        1. Traverse the given Binary Search Tree in inorder way and store the result in an array. Since inorder traversal of a Binary Search              Tree gives us all the elements in a sorted order. [ done in $O\left(n\right)$ time]

        2.  We need our array to be sorted because to balance the binary tree we need to make root of the balanced tree that has equal               number of smaller elements and equal number of larger elements that is we need get the middle element.

        3.   The recursive algorithm is shown below:-

            $balanceBST\left(A, lo, hi\right)${

       1.        $if hi<lo$

       2.                return

       3.        $mid = \frac{\left(hi+lo\right)}{2}$

       4.        element.data $\leftarrow $ A[mid]

       5.        element.left $\leftarrow $ $balanceBST\left(A, lo, mid−1\right)$

       6.        element.right$\leftarrow $$balanceBST\left(A, mid+1, hi\right)$

       7.         return element

**Time Complexity Analysis**: The inorder list takes $O\left(n\right)$ time and then we go over the entire list which also takes $O\left(n\right)$ time as we process each element once in $O\left(1\right)$ time.

**Space Complexity Analysis:**The inorder list takes $O\left(n\right)$ space and the recursive calls are made for each element which occupy space upto $O\left(n\right)$

3. a)

Two trees are identical if the element arrangement is similar in both the trees.

To see if two trees are identical we can traverse both trees simultaneously and check if the values at each node are the same.

Step 1: If both trees have null root return true

Step 2: If both roots are not null then check value of root

Step 3: AND the result with recursive call to the same function for left node of both trees AND right node of both trees.

Step 4: return true if result of AND for all the nodes is true

Step 5: Else return false

Psuedocode

$isIdentical\left(root1, root2\right)${

    $if root1 ==null and root2 ==null $

                return true

    $if root1 \ne null and root2 \ne null$

                $if root1.data == root2.data AND isIdentical\left(root1.left, root2.left\right) AND $

                            $isIdentical\left(root1.right, root2.right\right)$

                          return true

return false

**Time Complexity:**The time complexity will depend on the tree which has less number of nodes. Suppose Tree1  has ‘p’nodes and Tree2 has ’q’ nodes and p< q then Time Complexity of the algorithm would become $O\left(p\right)$ as the recursion will terminate when the traversal for either of the tree reaches the leaf node.

**Space Complexity:**The no of stack calls are equivalent to the no of nodes in the tree so the stack memory too depends on the tree with less no of nodes. $O\left(p\right)$

3.b)

We are given a Tree and we need to create an identical copy of the given tree. As in the arrangment of the nodes must be same as in the given tree.

Algorithm:

Step 1: if the given tree’s root is null then return null

Step 2: create a new node.

Step 3: save the value of the root’s data into the node’s data

Step 4: Then recursively make this new node’s left point to the root’s left and node’s right point to the root’s right

Step 5:  return the node

    $Replicate\left(root\right)${

1.      $if root==null$

2.             $return$

3.     $node\leftarrow root.data$

4.     $node.left\leftarrow Replicate\left(root.left\right)$

5.     $node.right\leftarrow Replicate\left(root.right\right)$

6.      $return node$

**Time Complexity:**The time taken by the algorithm depends on the total elements in the tree so the worst case would be $O\left(n\right)$

**Space Complexity:**$O\left(n\right)$ as the recursive calls in the stack would be saved in the memory for the total nodes in the tree.

4. The differences between the two approaches of implementing a binary search tree provided

and extended by us, in terms of how algorithms are implemented, ease of use and understanding, effeciency,

etc are as follows:

1.) The insertion in the tree implemented in RoutineBinarySearchTree.java class is an iterative approach which is proves to be more effecient than the recursive approach implemented in BinarySearchTreeImpl.java which proves to be ineffecient because it is recursive which occupies memory of the stack thus creating overheads.

Also, iterative approach is much simpler and easy to understand as well as to implement than the recursive approach.

2.) The same goes for the implementation of search in ElementNode.java, since it is recursive thus creates overheads and increases ineffeciency. But the same is implemented in an iterative manner in the RoutineBinarySearch.java

3.) BSTreeNode.java gives the flexibility to access right and left of the parent whereas the ElementNode.java only allows to access the parent by default so if the root of the tree is given we need to access left and right nodes to traverse the tree so it’s comparitively easier to use RoutineBinarySearch.java than BinarySearchTreeImpl.java