EECS2011: Fundamentals of Data Structures

Assignment 3

Name: Yang Wang

EECS ID: infi999

Student ID: 213894167

November 12, 2016

Problem 1: Card shuffle

For this problem, we need make a method for performing a card shuffle of a list of 2n elements. If the input is (a_1, a_2, a_3, a_4, ... , a_2n), then after splitting it into two halves, we have the two lists L_1 = (a_1, a_2, ..., a_n) and L_2 = (a_(n+1), a_(n+2), ... , a_(2n)). The final result is the single list corresponding to the sequence (a_1, a_(n+1), a_2, a_(n+2), ..., a_n, a_(2n)).

```
Algorithm card_shuffle(Node head):
begin
    If head is null then
    return error
    end if
    Node e ← head                    // create node e
    len ← 0
    while e is not null do
    len ← len + 1                    // get the length of list
    e ← e.next
    end while
    Node last ← e                    // get last node of list
    e ← head                         // set e back to first node
    for i←0; i< len/2; i++ do
    e ← e.next
    end for
    Node mid ← e                     // get mid node of list(also the first node of list 2)
    Node list1 ← head
    Node list2 ← mid
    while list1 is not equal to mid or list2 is not equal to end do
    Node temp1 ← list1.next
    Node temp2 ← list2.next
    list1.next ← list2
    list2.next ← temp1
```

list1 ← temp1

list2 ← temp2

end while

end algorithm

Above is my algorithm, the additional memory cell is O(1). I import the single LinkedList and use len/2 to get mid elements, in other word the first elements in list 2. I used two temp node to help me save the next node for list1 and list2. The whole process is finished within the input list, no additional list used, the additional memory cell is O(1).

Problem 2: Binary Tree Node Balance Factors

For this problem, we need make a method to find the balance factor of an internal position p and print out all the result for all the internal nodes.

Algorithm eulerTourBinarytToCalculateBalanceFactor(T, p, depth):

begin

if *p* has no left child and no right child **then**

return 0

end if

// initialize the left and right depth of p

nLeftDepth ← 0

nRightDepth ← 0

if p has a left child lc **then**

// recursively tour the left subtree of p to print balance factor

nLeftDepth ← eulerTourBinarytToCalculateBalanceFactor (T, lc, depth)

end if

if p has a right child rc **then**

// recursively tour the right subtree of p to print balance factor

nRightDepth ← eulerTourBinarytToCalculateBalanceFactor (T, rc, depth)

end if

// calculate the depth of p

depth ← 1 + (nLeftDepth > nRightDepth ? nLeftDepth : nRightDepth)

// print the element p

perform the "post visit" action for position p

// print the balance factor of p

calculate the balance factor of p that is abs(nRightDepth - nLeftDepth)

return depth

end algorithm

For my algorithm above, I used the reference eulerTourBinary() method on book "Data Structures and Algorithms in Java" page-349.

Since we have an 0(n)-time method for computing the number of descendants of each node by eulerTourBinarytToCalculateBalanceFactor (), therefore the complexity of the above algorithm is O(n).

Problem 3: Priority Search Tree

For this problem, we need to write an algorithm to fill set of n points with x, y-coordinates to binary tree and sort them from maximal y-coordinate and turns it to priority search tree.

Algorithm priority_search_tree_format ():

begin

put points S into integer x and y- coordinates

put them into tree, order them from left to right

create an arraylist to hold all tree elements

// initialize the start index (should start the last entry's parent node)

// n is the number of points as external nodes for tree

startIndex ← n-2

for j ← startIndex; j >= 0; j-- **do**

// loop until processing the root

using the bottom –up method to move the children with the largest y value up

downheap(j)

end for

end algorithm

Algorithm downheap(j):

begin

while hasLeft(j) || hasRight(j) **do**

// choose the child index k which has the larger y value

the children k with the largest y value up

swap(j,k)

j ← k

end while

end algorithm

Algorithm swap(j,k):

begin

temp ← arrayTree[j]

arrayTree[j] ← arrayTree[k]

arrayTree[k] ← temp

end algorithm

Since the bottom up heap constrictions takes linear time, so the algorithm takes linear time. In order to prove my algorithm is correct, I have also submitted the java code for problem3. And since there is no given the test case input, I used random number for y value and that means when you run the java code, the input and output changed when you run it again. <u>So to save your time, I will attach an output screenshot and explain to you the algorithm is correct.</u>

Input (the x-coordinates from 0 to 7, total 8 external nodes:

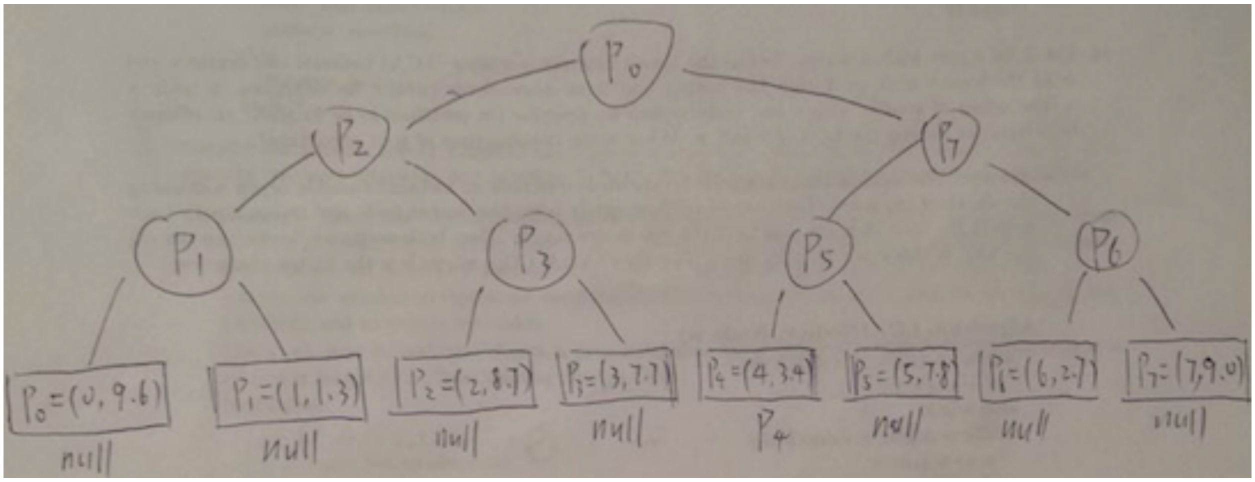
```
0.0
9.678372899301221
1.0
1.384324517472444
2.0
8.73989764514378
3.0
7.748014058520893
4.0
3.482441939369912
5.0
7.8387330837387985
6.0
2.782477511507201
7.0
9.019289367303248
```

p0 = (0, 9.6) p1 = (1, 1.3) p2 = (2, 8.7) p3 = (3, 7.7) p4 = (4, 3.4) p5 = (5, 7.8) p6 = (6, 2.7) p7 = (7, 9.0)

Output (the priority search tree):

```
0.0
9.678372899301221
2.0
8.73989764514378
7.0
9.019289367303248
1.0
1.384324517472444
3.0
7.748014058520893
5.0
7.8387330837387985
6.0
2.782477511507201
null
null
null
null
4.0
3.482441939369912
null
null
null
```

To let you check the output clearly, it's the same as below picture. (next page)



P0
P2
P7
P1
P3
P5
P6
P0=(0, 9.6)
P1=(1, 1.3)
P2=(2, 8.7)
P3=(3, 7.7)
P4=(4, 3.4)
P5=(5, 7.8)
P6=(6, 2.7)
P7=(7, 9.0)
null
null
null
null
P4
null
null
null