EECS2101A Assignment 3 Solutions Fall 2023

Instructor: Andy Mirzaian

Problem 1: [30%] Card Shuffle ([GTG] Exercise C-7.44, page 303):

Explanation: We are given a list L of an even number of elements in the form of a Singly Linked Positional List (SLPL). L.cardShuffle() permutes elements of L, alternating between elements of its first half and second half as they appeared in the original order. For example, list $L = (a_1, a_2, a_3, a_4, a_5, a_6, a_7, a_8)$ is changed to $L = (a_1, a_5, a_2, a_6, a_3, a_7, a_4, a_8)$.
Note that head and tail pointers of L remain intact.

Design Idea:. See LS3, pages 22-26 for SSL implementation, and LS7, page 17 for the Positional List (PL) ADT. Even though LS7, and [GTG] pages 277-280, propose a Doubly Linked Positional List (DLPL) implementation of PL, we can still assume a SLPL implementation as explained in class, since we do not intend to do "backward navigation" in the list. See the public method invocations of PL in our algorithm below. As explained in class, this implementation will support these operations, each taking O(1) time. The implementations are similar to the DLPL version discussed in [GTG] (nodes do not have the *prev* field; no need for them), hence its details are omitted here.

L.first():	returns a pointer to the first node of L (returns null if L is empty)
p.getNext():	returns the position in the list just after position p (returns null if p is the last position)
p.setNext(q):	replaces the next field of node p to point to position q
L.size():	returns the number of elements in L
L.isEmpty():	returns true iff L is empty
p.getElement():	returns the element stored in node p **(not used)**
p.setElement(e):	replaces the element field of node p by e **(not used)**

Note: For the sake of having an in-place cardShuffle algorithm, we will avoid using ***addFirst*** and ***addAfter*** methods. We also do not need operations ***removeFirst*** and ***removeAfter***.

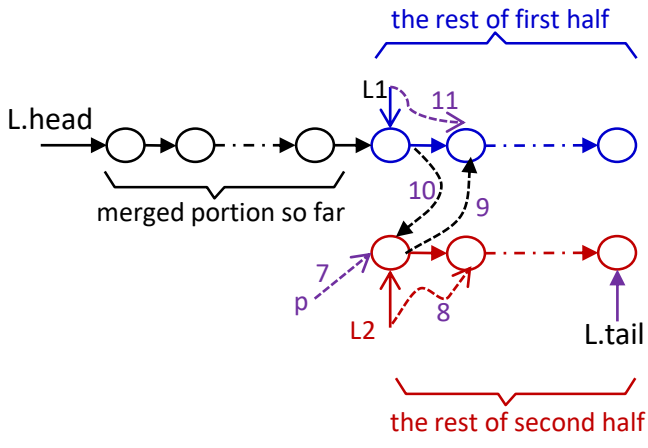
Note 1: As the problem statement required, the method is **in-place**, i.e., it uses only O(1) extra memory cells (int i, and pointers L1, L2, and p) besides the input list. In particular, it does not use any extra nodes or lists such as *firstHalfList* and *secondHalfList* that would require more than O(1) memory cells for storage. Any non-inplace solution will only get partial credit.
Note 2: Even worse, anyone who uses additional *ArrayList* is totally missing the point.

Running Time Analysis: This algorithm clearly takes $O(n)$ time, where $2n$ is the number of elements in the input list, since it spends O(1) time per node in the list.
The figure below shows the **Loop Invariant** and the sequence of local changes (i.e., pointer manipulations) made in each iteration of the while-loop to maintain that LI & make progress.

```
Algorithm cardShuffle( )          //  O(n) time
// Pre-Cond:   input instance list L is a SLPL of 2n elements as explained above.
// Post-Cond:  L is shuffled in-place as described above.
1.  if ( isEmpty() ) then return

    // Now split non-empty list L in two halves L1 and L2:
2.  L1 ⟵ p ⟵ first()
3.  for ( i ⟵ 1 .. size()/2 ) do p ⟵ p.getNext()   // go to the end of the first half
4.  L2 ⟵ p.getNext()   //  second half L2 starts here
5.  p.setNext(null)      //  L1 ends here; L is now split into separate L1 and L2 lists

    // Now merge L1 and L2 with alternating nodes from each:
6.  while ( L1 ≠ null ) do
7.        p ⟵ L2
8.        L2 ⟵ L2.getNext()
9.        p.setNext(L1.getNext())
10.       L1.setNext(p)
11.       L1 ⟵ p.getNext()
12. end  // while-loop
end   // cardShuffle
```



Problem 2: [30%] Binary Tree Node Balance Factors ([GTG] Exercise C-8.44, page 353):

Design Idea: Let T be a given proper binary tree. For each node v of T, we can define $height(v)$ and $balance(v)$ recursively as follows:

$$height(v) = \begin{cases} 0 & if\ v\ is\ external \\ 1 + \max\Big(\ height\big(left(v)\big), height\big(right(v)\big)\Big) & if\ v\ is\ internal \end{cases}$$

$$balance(v) = \begin{cases} 0 & if\ v\ is\ external \\ \Big|\ height\big(left(v)\big) - height\big(right(v)\big)\ \Big| & if\ v\ is\ internal \end{cases}$$

We can compute these quantities using the Euler Tour algorithm. These formulas suggest post-order traversal. So, all we need to do is to override *visitExternal( ) and visitRight( )* methods. (See LS8, pages 20-21.) We will choose a more compact solution and use post-order directly as shown below.

Note 1: $height(v)$ and $balance(v)$ are **not stored fields** in node v. Any solution that uses something like *node.height* or *node.balance*, assumes height or balance as a stored field and will only get partial credit.

Note 2: Our algorithm uses *leftHeight, rightHeight, height*, and *balance* as **local variables** of the *balanceHeight( )* method (they are not stored fields in the tree nodes).

Note 3: Our algorithm *balanceHeight( )* computes height and balance in a **single traversal** of the tree. If one writes **two separate traversal algorithms,** one to compute height and the other to use the former to compute and print balance factors, then the worst-case time complexity will be $O(n^2)$. This is because the latter makes linear-time nested calls to the former). Such inefficient algorithms will only get partial credit.

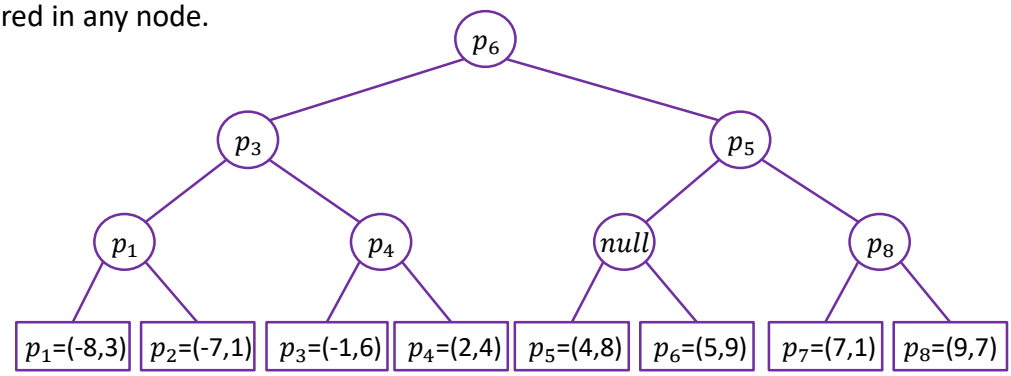
Running Time Analysis: Since this is a post-order traversal algorithm, and visiting a node v (lines 2 and 5-8) takes O(1) time, the entire algorithm takes $O(n)$ time, where n is the number of nodes in the main tree.

```
Algorithm nodeBalanceFactors ( T )          //  O(n) time,  n =  |T|
// Pre-Cond:   T is a proper binary tree.
// Post-Cond:  Prints 〈 element(v) , balance(v) 〉 for each internal node v of T in post-order.
1.  rootHeight ⟵ balanceHeight ( T.root( ) )   // returned value ignored
end
```

```
Algorithm balanceHeight ( v )
// Pre-Cond:    v is root of a proper binary tree.
// Post-Cond:   prints 〈 element(u) , balance(u) 〉 for all internal descendants u of v
                in post-order, and returns height(v).
2.  if v is external then return 0   // height of external node
3.  leftHeight  ⟵ balanceHeight ( left(v) )   // recursively traverse left subtree
4.  rightHeight ⟵ balanceHeight ( right(v) )   // recursively traverse right subtree
    // Lines 5 - 8 :  "visit" node v in post-order:
5.  height  ⟵ 1 + max ( leftHeight, rightHeight )
6.  balance ⟵ | leftHeight – rightHeight |   // absolute value of difference
7.  print ( element(v), balance )   // print element and balance factor of node v
8.  return height   // height of node v
end
```

Problem 3: [40%] Priority Search Tree (PST) ([GTG] Exercise P-9.56, page 400):

Design Idea: See the illustrative figure below which was also given on the Forum. We assume the given data points in the external nodes of T are all distinct, even though some may have equal x or y coordinates. We need to fill in the internal nodes to turn T into a PST. The idea is similar to heapify that applies downHeap on nodes in post-order. The points with largest y-coordinate will bubble up to the top. We use the comparison *yxLexCompare* whose code is shown on the next page (if two points have equal y-coordinates, we break the tie by taking the one with larger x-coordinate). To avoid unnecessary code complication, we use the same identifier, i.e., $point$ (instead of $point$ for external & top for internal nodes) to refer to the point stored in any node.



Definitions: First let us define the following terms for any internal node v.

- $S(v) =$ the set of points in external descendants of v.
- $S_p(v) = \{q \in S(v) \mid q <_{yxLex} p\}$ is the set of points in $S(v)$ that are yxLex < p.
- $\overline{S_p}(v) = S(v) - S_p(v)$ is the complementary set in $S(v)$.
- Let INF_POINT $= p_\infty = (\infty, \infty) >_{yxLex}$ any point in the xy-plane.
- For any set of points A, let $\max(A) =$ the yxLex maximum point in A, and $\min(A) =$ the yxLex minimum point in A.
- If $A = \emptyset$ then $\min(A) = p_\infty$ and $\max(A) = null$. Also note that $S_{p_\infty}(v) = S(v)$.
- Now define $point(v)$ inductively (top-down) as follows:

$$point(v) = \max\left(S_p(v)\right), \quad \text{where } p = \begin{cases} p_\infty & \text{if } v = root(T) \\ point(parent(v)) & \text{otherwise} \end{cases}$$

Note the max-heap ordering: if not null, then $point(v) <_{yxLex} point(parent(v))$.

Running Time Analysis: The running time of makePrioritySearchTree (T) is similar to heapify() which is O(n). (See LS9, page 40.)

Algorithm *makePrioritySearchTree* (T) // O(n) time, n = # of external nodes in T
// Pre-Cond: T is a proper complete binary tree with distinct 2D points stored in its external nodes left-to-right in x-sorted order.
// Post-Cond: Turns T into a PST.
1. INF_POINT ⟵ (∞, ∞) // global constant point is yxLex larger than any data point
2. *makePST* (T.root()) // similar to heapify() with post-order downHeaps
end

Algorithm *makePST* (v) // O(size of subtree rooted at v) time
// Pre-Cond: v is root of a sub-tree of the main tree T.
// Post-Cond: this subtree is turned into a PST.
3. **if** v is external **then return**
4. *makePST* (left(v))
5. *makePST* (right(v))
6. *downHeapPST* (v , INF_POINT) // plays the role of "visit(v)" in post-order
end

Algorithm *downHeapPST* (v , p) // O(height(v)) time
// Pre-Cond: left and right subtrees of internal node v are PSTs.
(Points in $\overline{S_p}(v)$ are reserved for proper ancestors of v .)
// Post-Cond: the tree rooted at v becomes a PST with internal nodes restricted to $S_p(v)$.
7. point(v) ⟵ null
8. c ⟵ bestChild (v , p) // $\text{point(c)} = \max\left(S_p(v)\right)$, c = null if $S_p(v) = \emptyset$
9. **if** c = null **then return**
10. point(v) ⟵ point(c) // bubble up $\max\left(S_p(v)\right)$ from c to v
11. **if** c is internal **then** *downHeapPST* (c , point(v)) // v = parent(c)
end

Algorithm *bestChild* (v , p) // O(1) time
// Pre-Cond: v is internal and p is smallest yxLex point among its proper ancestors.
// Post-Cond: returns the child of v that contains $\max\left(S_p(v)\right)$, null if no such child.
12. **if** point(right(v)) ≠null and yxLexCompare(point(right(v)) , p) < 0
13. **then** c ⟵ right(v) **else** c ⟵ null // does right child qualify?
14. **if** point(left(v)) = null or yxLexCompare(point(left(v)) , p) ≥ 0
15. **then return** c // here left child doesn't qualify
16. **if** c ≠null and yxLexCompare (point(c) , point(left(v))) > 0
17. **then return** right(v) **else return** left(v) // choose best qualified child
end

Algorithm *yxLexCompare* (p, q) // O(1) time, similar to LS9, page 7
// Pre-Cond: p and q are two non-null points in 2D plane.
// Post-Cond: returns $-ve$, 0, $+ve$ if p is yxLex < q, = q, or > q, respectively.
18. **if** p.getY() ≠ q.getY()
19. **then return** p.getY() − q.getY()
20. **else return** p.getX() − q.getX()
end