York University
EECS 2011 Winter 2018 – Problem Set 2
Instructors: James Elder, Suprakash Datta

Solutions

1. Choosing a data structure

   State in one or two words the simplest ADT and implementation we have discussed that would meet each requirement.

   (a) O(1) time removal of the most recently added element
   ADT: Implementation:

   - Answer: ADT: Stack, Implementation: Array

   (b) O(1) average time addition, removal, access and modification of (key, value) pairs with unique keys
   ADT: Implementation:

   - Answer: ADT: Map, Implementation: Hash table

   (c) O(1) time insertion and removal when you are given the position
   ADT: Implementation:

   - Answer: ADT: Node List, Implementation: Doubly-linked list.

   (d) O(1) time index-based access and modification and amortized O(1) addition of elements
   ADT: Implementation:

   - Answer: ADT: Array List, Implementation: Array

   (e) O(log n) time insertion of (key, value) entries and O(log n) removal of entry with smallest key
   ADT: Implementation:

   - Answer: ADT: Priority Queue, Implementation: Heap

   (f) O(1) time removal of the least recently added element
   ADT: Implementation:

   - Answer: ADT: Queue, Implementation: (circular) array

2. Binary Trees

   You are to design a recursive algorithm **btDepths(u, d)**, where u is a node of a binary tree and d is the depth of u. Your algorithm will determine the minimum and maximum depths of the external nodes descending from u. Note that if u has no parent (i.e., is the root of the whole tree), then $d = 0$. You can assume that each node v of the tree supports the following four binary tree accessor methods: **left(v), right(v), hasLeft(v)** and **hasRight(v)**. You can also assume that u is not null. Your algorithm should run in $O(n)$ time, where n is the number of nodes descending from u.

   Input: A non-null node **u** of a binary tree, and its depth d.

   Output: An object **depths** consisting of the two integer fields **depths.min** and **depths.max**, containing the minimum and maximum depth over all external nodes descending from u.

   (a) (20 marks) Your algorithm (in pseudocode or Java):
   Algorithm btDepths(u, d):

   - Answer:

```
if hasLeft(u) & hasRight(u)
      leftDepths = btDepths(left(u),d+1)
      rightDepths = btDepths(right(u),d+1)
      depths.min = min(leftDepths.min,rightDepths.min))
      depths.max = max(leftDepths.max,rightDepths.max))
      return depths
elseif hasLeft(u) & !hasRight(u)
      return btDepths(left(u),d+1)
else if !hasLeft(u) & hasRight(u)
      return btDepths(right(u),d+1)
else // External node
      depths.min = d
      depths.max = d
      return depths
```

(b) (5 marks) Provide a brief justification for why you think your algorithm is $O(n)$.

- Answer: The algorithm visits each node exactly once, and does a constant amount of work at each node. Thus the total work is $O(n)$.

3. Suppose you have a stack S containing n elements and a queue Q that is initially empty. Describe (in pseudocode or English) how you can use Q to scan S to see if it contains a certain element x, with the additional constraint that your algorithm must return the elements back to S in their original order. You may not use an array or linked list only S and Q and a constant number of reference variables.

- Answer: We use the queue Q to process the elements in two phases. In the first phase, we iteratively pop each element from S and enqueue it in Q, and then we iteratively dequeue each element from Q and push it into S. This reverses the elements in S. Then we repeat this same process, but this time we also look for the element x. By passing the elements through Q and back to S a second time, we reverse the reversal, thereby putting the elements back into S in their original order.

4. Describe the structure and pseudo-code for an array-based implementation of the array list ADT that achieves O(1) time for insertions and removals at index 0, as well as insertions and removals at the end of the array list.

- Answer: A simple solution is to adapt the array-based queue implementation described in Section 6.2.2 of the text. This employs front and rear indices f and r, and modular arithmetic to use the array in a circular way. We modify the operation slightly, so that f and r will always point to the next available element at the beginning and end of the list, respectively. Thus f and r will be initialized to locations 0 and 1 of the array respectively, and the array will be considered full when f=r. Thus an array of size N will support an Array List of size N-1.

  Instead of implementing the front(), enqueue(e) and dequeue() methods of the queue ADT, we implement the get(i), set(i, e), add(i, e) and remove(i) methods of the Array List ADT in the following way, based upon an array of size N:

  - get(i): return the element from location (f+i+1) mod N
  - set(i, e): set the element at location (f+i+1) mod N to e
  - add(i, e):
    - If i <size()/2
      - Shift all elements with indices less than i to the left one position (using modular arithmetic).

        - $f \leftarrow (f-1) mod N$
        - set the element at location (f+i+1) mod N to e
      * Otherwise
        - Shift all elements with indices greater than or equal to i to the right one position (using modular arithmetic).
        - $r \leftarrow (r+1) mod N$
        - set the element at location (f+i+1) mod N to e
  - remove(i):
    * If i < size()/2
      - Shift all elements with indices less than i to the right one position (using modular arithmetic).
      - $f \leftarrow (f+1) mod N$
    * Otherwise
      - Shift all elements with indices greater than i to the left one position (using modular arithmetic).
      - $r \leftarrow (r-1) mod N$

  If an add(i,e) message is received for a full array, the array must be extended, as for the standard Array List implementation.

5. Describe how to implement an iterator for a circularly linked list. Since hasNext() will always return true in this case, describe how to perform hasNewNext(), which returns true if and only if the next node in the list has not previously had its element returned by this iterator.

   - Answer: The iterator simply maintains two instance variables called start and next, which are both initialized to cursor. If start is encountered while incrementing next, the next variable is set to null, signaling a complete cycle of the list.

   Algorithm next()
   if next = null **then**
   throw exception
   end if
   curr ← next
   if next.getNext() = start **then**
   next ← null
   else
   next ← next.getNext()
   end if
   return curr

   Algorithm hasNewNext()
   if next ≠ null **then**
   return true
   else
   return false
   end if

6. Describe (in pseudocode or English) an O(n) recursive algorithm for reversing a singly linked list L, so that the ordering of the nodes becomes opposite of what it was before.

- Answer: We simply recurse to the end of the list and then reverse pointers as the recursion unwinds, returning the last node on the way back, and making it the new head.

  From an inductive point of view, at any intermediate node we have a "friend" provide us with the reversal of the tail section of the list to the right of our node, and then simply update the new tail of that tail section to point to our node.

```
Algorithm reverse()
  newHead ← reverseSub(head)
  head.next ← null
  head ← newHead
```

```
Algorithm reverseSub(node)
  if node.next ≠ null then
    newHead ← reverseSub(node.next)
    node.next.next ← node
    return newHead
  else
    return node
  end if
```

7. Let T be a tree with n nodes. Define the lowest common ancestor (LCA) between two nodes v and w as the lowest node in T that has both v and w as descendents (where, by definition, a node is a descendent of itself). Given two nodes v and w, describe (in pseudocode or English) an efficient algorithm for finding the LCA of v and w. Assume that each node is extended to include an instance variable *depth* that contains the depth of the node. What is the running time of your algorithm?

   - Answer: Our algorithm simply takes the deeper node and traces a path back toward the root until the depths of the two nodes match. Then both nodes are backed up toward the root in tandem until they meet. The algorithm is O(h), where h is the height of the tree.

```
Algorithm LCA(Node v, Node w)
  while v.depth > w.depth do
    v ← v.parent
  end while
  while w.depth > v.depth do
    w ← w.parent
  end while
  while v ≠ w do
    v ← v.parent
    w ← w.parent
  end while
  return v
```

8. Given a min heap T and a key k, give an algorithm to compute all of the entries in T with key less than or equal to k. The algorithm should run in time proportional to the number of entries returned.

   - Answer: Starting at the root of the tree, recursively search the left and right subtrees if the root of the subtree has a key value less than or equal to k, returning each node visited.