York University
EECS 2011Z Winter 2015 – Problem Set 2
Instructor: James Elder

Solutions

1. Array Lists

   What is the asymptotic running time of the method makeList below as a function of N? Please justify your answer.

```
1  public static List<Integer> makeList( int N )
2  {
3    ArrayList<Integer> list = new ArrayList<Integer>();
4
5    for( int i = 0; i < N; i++ )
6    {
7      list.add( i );
8      list.trimToSize( );
9    }
10 }
11
12 /**
13 * Trims the capacity of this ArrayList instance to be the
14 * list's current size. An application can use this operation to minimize
15 * the storage of an ArrayList instance.
16 */
17 public void trimToSize() {
18     modCount++;
19     int oldCapacity = elementData.length;
20     if (size < oldCapacity) {
21         Object oldData[] = elementData;
22         elementData = (E[])new Object[size];
23         System.arraycopy(oldData, 0, elementData, 0, size);
24      }
25 }
```

   - Answer: makeList calls methods **add** and **trimToSize** once each on each iteration of the for loop. On iteration i, both **add** and **trimToSize** require that i array elements be copied to a new array. Thus the asymptotic complexity is $\sum_{i=1}^{N} i = N(N+1)/2$.

2. Linked Lists

   You are to design an efficient iterative algorithm **merge**(A, B) that accepts two singly-linked lists, each containing a strictly increasing sequence of integers (i.e., with no repeating elements), and merge them (take their union) into a single strictly increasing list of integers (again, with no repeating elements). For example, the input $A = [1, 4, 8, 10, 11, 20]$, $B = [-5, 1, 8, 9, 20]$ should return a reference to a list containing $[-5, 1, 4, 8, 9, 10, 11, 20]$.

   Your algorithm should use only O(1) additional memory beyond the two input lists, and should run in $\mathcal{O}(\max(m, n))$ time, where m and n are the lengths of the two input lists.

   Your algorithm is given only references A and B to the first nodes in each of the two input lists. Each node is comprised only of *val* and *next* instance variables. You may assume that empty lists are represented as *null* nodes and the *next* field of the last node in each list is set to *null*. No other list variables are available to you.

Input: Two singly linked lists A and B, each containing a strictly increasing sequence of integers.

Output: The union of A and B as a singly linked list of strictly increasing integers.

(a) (20 marks) Your algorithm (in pseudocode or Java):
Algorithm merge(A,B):

- Answer:

```
1  package Midterm;
2
3  public class Merge {
4      IntNode merge(IntNode A, IntNode B) {
5          IntNode C = null;
6          IntNode curr = null;
7          if (A == null) {
8              return (B);
9          } else if (B == null) {
10             return (A);
11         } else if (A.val < B.val) {
12             C = A;
13             A = A.next;
14         } else if (B.val < A.val) {
15             C = B;
16             B = B.next;
17         } else {
18             C = A;
19             A = A.next;
20             B = B.next;
21         }
22         curr = C;
23         while (A != null && B != null) {
24             if (A.val < B.val) {
25                 curr.next = A;
26                 A = A.next;
27             } else if (B.val < A.val) {
28                 curr.next = B;
29                 B = B.next;
30             } else {
31                 curr.next = A;
32                 A = A.next;
33                 B = B.next;
34             }
35             curr = curr.next;
36         }
37         if (A != null) {
38             curr.next = A;
39         } else {
40             curr.next = B;
41         }
42         return C;
43     }
44 }
```

(b) (5 marks) Your algorithm is efficient in both time and memory. What is the biggest sacrifice you have made in achieving this efficiency?

- Answer: The algorithm destroys the input lists, which could lead to error if the caller is not careful.

3. 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

4. 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 $O(n)$.

5. **Largest Imbalance**

We define the imbalance of a node x of a proper binary tree as the difference between the lengths of the shortest and longest paths from x to a descendent external node.

Provide the pseudocode for a recursive algorithm $Imbalance(x)$ that returns the largest imbalance i in the tree with root node x, as well as the lengths s and l of the shortest and longest paths from x to an external node. You may assume methods getLeft(x) and getRight(x) return the left and right child nodes of node x, and method isExternal(x) returns true if the node is external, false if internal.

Input: The root x of a proper binary tree.

Output: The largest imbalance i in the tree as well as the lengths s and l of the shortest and longest paths from x to an external node.

- Answer:

algorithm $Imbalance(x)$

<pre−cond>: x is the root of a proper binary tree.

<post−cond>: The largest imbalance i in the tree as well as the lengths s and l of the shortest and longest paths from x to an external node.

```
begin
    if (isExternal(x)) then
        return( ⟨0, 0, 0⟩ )
    else
        ⟨i_l, s_l, l_l⟩ = Imbalance(x.getLeft())
        ⟨i_r, s_r, l_r⟩ = Imbalance(x.getRight())
        s = min(s_l, s_r) + 1
        l = max(l_l, l_r) + 1
        i = max(i_l, i_r, l − s)
        return(⟨i, s, l⟩)
    end if
end algorithm
```

The time is $O(n)$.

6. What are the asymptotic running times of the methods add and remove of the class SparseNumericVector that you modified for Programming Question 1?

   - Answer:
     They are both $\Theta(n)$, where n is the number of non-zero elements in the vector.

7. Give an example of a Java code fragment that performs an array reference that is possibly out of bounds, and if it is out of bounds, the program catches that exception and prints the following error message: Dont try buffer overflow attacks in Java!

   - Answer:

```
1 try {
2 val = myArray[idx];
3 System.out.println("The value stored at position "+idx+"  of myArray
      is " + val);
4 } catch (ArrayIndexOutOfBoundsException aioobx) {
5 System.out.println("Don't try buffer overflow attacks in Java!");
6 }
```

8. 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.

9. 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 5.2.2 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.

10. Describe (in pseudocode or English) an algorithm for reversing a singly linked list L using only a constant amount of additional space and not using any recursion.

    - Answer: The solution requires 3 pointers, prev, curr and next. prev is required as the new target for curr.next. next is required so that we do not lose our reference to the rest of the list when we redirect curr.next.

      if head = null **then**
        return head
      end if
      prev← head
      curr ← prev.next
      if cure = null **then**
        return head
      end if
      next ← curr.next
      prev.next ← null
      curr.next ← prev
      while next ≠ null **do**
        prev ← curr
        curr ← next
        curr.next ← prev
        next ← next.next
      end while
      head ← cure

11. 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
```

12. 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
```

13. Describe (in pseudocode or English) an algorithm that will output all of the subsets of a set of n elements (without repeating any subsets). What is the asymptotic running time of your algorithm?

- Answer: Here, our "friend" returns to us the complete set of subsets SS of the input set minus the first element. Then all we have to do is make sure we return the union of this set of subsets SsS with the set of subsets formed by adding this first element to each of the sets in SsS.

  We use the symbol to denote set complement (i.e., S S.first is the set S with the element S. first removed) and to denote set union.

  The work done in each stackframe is proportional to the number of sets in NSsS, which is 2i at a recursion height of i. Thus running time is O(2n).

  Notation:

  - S = input Set of elements
  - SsS = Set of subSets
  - NSsS = New Set of subSets
  - sS = one subSet

  Algorithm subsets(S)
  if S = null **then**
  return null
  end if
  SsS ← subsets(S S.first)
  NSsS ← SsS
  for all sS in SsS **do**
  Add Si S.first to NSsS
  end for
  return NSsS

14. The balance factor of an internal node v of a proper binary tree is the difference between the heights of the right and left subtrees of v. Describe (in pseudocode or English) an efficient algorithm that specializes the Euler tour traversal of Section 7.3.7 to print the balance factors of all the internal nodes of a proper binary tree.

    - Answer: One way to do this is to extend the binary tree node class to support a height instance variable. Then the visitRight() method can be used to both update the height variables and to print the balance factors:

    Algorithm visitRight()
    if v.isInternal(v) **then**
    v.height ← max(v.leftchild.height, v.rightchild.height) + 1
    balance ← absval(v.rightchild.height − v.leftchild.height)
    else
    v.height ← 0
    balance ← 0
    end if

15. 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. What is the running time of your algorithm?

- Answer: We assume that each node is extended to include an instance variable depth containing the depth of the node. Then 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

16. We can represent a path from the root to a given node of a binary tree by means of a binary string, where 0 means go to the left child and 1 means go to the right child. Use this to design an time algorithm for finding the last node of a complete binary tree with n nodes, assuming a linked structure implementation that does not keep a reference to the last node.

    - Answer: The path to the last node in the heap is given by the path represented by the binary expansion of n with the highest-order bit removed.

17. Given a 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 nod visited.