Quiz #2

Name: Antonio Rosado

I declare that **this** prequiz is entirely my own work

Submission Date: 2.19.2023

a) The List (ADT) is a collection of data we've been working with where each eleme
nt has a unique index or position in the collection.
The List ADT specifies basic operations that can be performed on the list, such as
inserting an element, deleting an element,
accessing an element at a specified index, iterating through the list.

1.) One implementation of List ADT is the Resizable Array-based (RA) implementat
ion. This implementation stores the list elements
in an adjacent block of memory and uses an index to access the elements. The RA
implementation uses an Array to store the elements
and resizes the array when it reaches its maximum size. The add method is overri
dden to resize the array **if** it reaches its maximum
size. The reverse method is also implemented to reverse the order of elements in
the array.

2.) The second approach of List ADT is the Simply Linked Structure (SLS) impleme
ntation. This implementation is a data structure in which
each element, or node, contains a reference to the next element in the sequence.
This creates a linked list of nodes that are connected
in a linear order. The first element in the list is called the head, the **final** n
ode typically contains a **null** reference, indicating the
end of the list. I

3.) The third approach of List ADT is the Doubly Linked Structure-based (DLS), w
hich is a better implementation than the SLS implementation.
This implementation is a data structure in which the structure itself is acccess
ed via a 'head' or initial state, and a 'tail/back' state.
The head and tail are the two links of the structure that allow **for** traversal al
location.

4.) The fourth approach of List ADT is the Circular Doubly Linked Structure-base
d (CDLS) and its a type of linked list data structure where each
node contains a reference to the previous and next node, creating a circular lin
k between the nodes. The head of the list is a reference to the
first node, and the tail is a reference to the last node.

Implementations:

RA/Array Based:

```
public class ListArrayBased implements ListInterface
      {
            private static final int MAX_LIST = 3;
            protected Object []items; // an array of list items
            protected int numItems; // number of items in list

            public ListArrayBased()
            {
                  items = new Object[MAX_LIST];
                  numItems = 0;
            } // end default constructor
      }
```

Linked List:

```
public class MyListReferenceBased implements ListInterface
{
      private Node head;
      public MyListReferenceBased()
      {
            head = null;
      }
}
```

Doubly Linked List

```
public class DLLIstADT implements ListInterface
{
      private Node head;
      private Node tail;
      int numItems;
      public MyListReferenceBased()
      {
            head = null;
            int = numItems;
            Tail = null
      }
}
```

Circularly Doubly Linked List:

```
public class ListCDLSBased implements ListInterface
{
      private int numItems;
      private DNode head;
      public ListCDLSBased()
      {
            head = null;
            numItems = 0;
      }
}
```

b)

Example 1:

```
Public static ListADT reverse(ListADT list)
{
      ListADT tmp = new ListADT();
      int size = list.size();
      for (int i = 0; i < size; i++)
      {
            tmp.add(i, list.get(list.size()-i-1));
      }
return tmp;
}
```

This reverse implementation takes the list as a parameter, makes temporary
storage, and that
temporary storage gets filled in order from start-end with the opposite or
der of the parameter, giving
the last index in parameter the first index in temp, and returning the tem
porary storage.

Example 2:

```
Public static void reverse(ListADT list)
{
      for (int i = 0, i < list.size(); i++)
      {
            item = list.get(i);
            list.remove(i);
```

```
                        list.add(list.size()-i-1, item);
                    }
                }
```

This reverse implementation grabs the list itself and moves items one at a time from the start of the list, moving index 0 to list.size - 1, 1 to list.size - 2 etc. up to index n-2 to index 1 via traversal.

c)

1.) RA implementation:

Space complexity:

MAX_LIST = 3
Object [] items = ref1 + ref2 + ref3
numItems = **int** (4 bytes)
list = ref
numItems = **int**:
ref + **int** + items.length * ref = (items.length + 2) * 4 bytes

Time complexity:

Operations used: get(**int** index), add(**int** index, Object item)
Traversal: Object items[index] (Direct Index Access)

get(**int** index):
0 shifts

add(**int** index, Object item):
Depends
Best **case**: index n costs 0 + n
Worst **case**: index 0 costs n + n
Average: index n/2 costs n/2 + n

remove(**int** index):
Depends
Best **case**: index n-1 costs 0
Worst **case**: index 0 costs n-1
Average: index (n-1)/2 costs (n-1)/2

find(**int** index):
0 shifts

The RA implementation Pros:

Simple, no additional support. Has the advantage of providing constant time access to elements, which makes it efficient **for** random access to elements. It is also efficient in terms of memory usage since it only allocates memory **for** the number of elements it contains.
Also has direct index access.

The RA implementation Cons:

Copy upon resizing, pre-allocation of memory. Can be inefficient **for** inserting and deleting elements in the middle of the list since it requires shifting all elements after the insertion or deletion point. Additionally, resizing the array can be an expensive operation, especially **for** large arrays.

2.) SLS implementation:

Space complexity:

head = ref
numItems = **int**
ref + ref + **int** + n * (ref1 + ref2) = (2n+2)*4 bytes

Time complexity:

Operations used: get(**int** index), add(**int** index, Object item), remove(**int** index), find(**int** index)
Traversal: Via links with 'head'. item.getHead()

get(**int** index):
Depends
Best **case**: index 0 costs 0
Worst **case**: index n-1 costs n-1
Average: index (n-1)/2 costs (n-1)/2

add(**int** index, Object item):
Depends
Best **case**: index 0 costs 0
Worst **case**: index n costs n
Average: index n/2 costs n/2

remove(**int** index):
Depends
Best **case**: index 0 costs 1
Worst **case**: index n-1 costs n
Average: index (n+1)/2 costs (n+1)/2

find(**int** index):
Depends
Best **case**: index 0 costs 0
Worst **case**: index n-1 costs n-1
Average: index (n-1)/2 costs (n-1)/2

The SLS implementation Pros:

It's simple to implement making the operations are efficient, and on demand memory allocation. It is also easy to add or remove elements from the top of the stack, which makes it suitable **for** certain applications.

The SLS implementation Cons:

Does not allow **for** efficient access to elements other than the top element of the stack. Additionally, the use of an array to implement the stack in **this** particular code results in the need to resize the array, which can be inefficient **for** larger stacks.

3.) DLS implementation:

Space complexity:

head = ref
back = ref
numItems = **int**
ref + ref + **int** + n * (ref1 + ref2) = (2n+2) * 4 bytes

Time complexity:

Operations used: get(**int** index), add(**int** index, Object item), remove(**int** index), find(**int** index)
Traversal: Via links with 'head' and 'back/tail'. item.getBack(), item.getHead()

get(**int** index):
Depends
Best **case**: index 0 costs 0
Worst **case**: index (n-1)/2 costs (n-1/)2
Average: index (n-1)/2 costs (n-1)/2

add(**int** index, Object item):
Depends
Best **case**: index 0 costs 0
Worst **case**: index n/2 costs n/2
Average: index n/4 costs n/4

```
                remove(int index):
                Depends
                Best case: index 0 costs 1
                Worst case: index (n-1)/2 costs n/2
                Average: index (n+1)/4 costs n/4

                find(int index):
                Depends
                Best case: index 0 costs 0
                Worst case: index (n-1)/2 costs (n-1)/2
                Average: index (n-1)/4 costs (n-1)/4

        The DLS implementation Pros:
                Better referencing, and good linked traversal.

        The DLS implementation Cons:
                More memory for necessary pointers: i.e. 'head'.

       4.) CDLS implementation:
        Space complexity:
                The total space complexity of the CDLS implementation is 12 bytes.

                head = ref1 (next) + ref2 (back) (DNode)
                numItems = int: ref + int + n * (ref1 + ref2 + ref3) = (3n + 2) *
4 bytes.
        Traversal: Via links with 'head' and 'back/tail'. item.getBack(), item.get
Head(),
                curr = curr.getNext() or curr = curr.getBack();

        Time complexity:
                Operations used: get(int index), add(int index, Object item), remo
ve(int index),
                find(int index) and shifting (items[i] = items[i + or -] different
 size)

                get(int index):
                Depends
                Best case: index 0 costs 0
                Worst case: index (n-1)/2 costs (n-1)/2
                Average: index (n-1)/4 costs (n-1)/8


                add(int index, Object item):
                Depends
                Best: index 0 costs 0
                Worst: index n/2 costs n/4
                Average: index n/4 costs n/8

                remove(int index):
                Depends
                Best case: index 0 costs 1
                Worst case: index (n-1)/2 costs n/4
                Average: index (n+1)/4 costs (n+1)/8

                find(int index):
                Depends
                Best: index 0 costs 0
                Worst: index (n-1)/2 costs (n-1)/2
                Average: index (n-1)/4 costs (n-1)/8

        The CDLS implementation Pros:
        Supports fast traversal in both directions, suitable for applications wher
e frequent modifications of the list are necessary.

        The CDLS implementation Cons:
        The space complexity is less efficient compared to the RA based implementa
tion, and the implementation is more complex.
```