University of Bahrain

College of Information Technology

Department of Computer Science

First Semester, 2018-2019

**ITCS214 (Data Structures)**

**Tutorial 3**

**Question 1 (A)** Write a generic class called **SingleLinkedList**having following data fields (private):

private Node<E> head; // reference to the first node

private int size; // number of nodes

This class is also having an inner class called **Node** (as discussed in the lectures).

The class **SingleLinkedList** will have the following private methods:

|  |  |
| --- | --- |
| **Private Method** | **Behavior** |
| private void addFirst (E item) | Inserts a new node in the beginning of the list with item as *data* |
| private void addAfter (Node<E> nodeRef, E item) | Inserts a new node with item as data after the node with reference *nodeRef* |
| private E removeAfter (Node<E> nodeRef) | Deletes the node after the node referenced by *nodeRef* and returns the *data* of the node deleted |
| private E removeFirst () | Deletes the first node of the list and returns the *data* of the node deleted |
| private Node<E> getNode(int index) | Returns reference to the node at position given by *index* |

The class **SingleLinkedList** will have the following public methods:

|  |  |
| --- | --- |
| **Public Method** | **Behavior** |
| public SingleLinkedList ( ) | Constructor to initialize *head* to null and *size* to 0 |
| **public int size()** | Returns current *size* |
| **public boolean contains(E obj)** | Checks whether the given object *obj* is present in the list. If it is there then it returns true else it returns false. |
| **public void clear()** | Removes all the elements of the linked list and makes it empty |
| public boolean isEmpty() | Checks whether the list is empty or not |
| public boolean add ( E anEntry) | Adds object *anEntry* at the end of the list and returns true. |
| public void add (int index, E anEntry) | Adds object *anEntry* in the list at the location given by *index* |
| public E get (int index) | Returns the element of the list at position given by *index* |
| public E set (int index, E newValue) | Updates the element at position index by *newValue* and returns the old value |
| public E remove (int index) | Removes the element at position *index* and returns the element being removed |
| public boolean remove (E obj) | Removes the first occurrence of the object *obj* from the list, if present and returns true, else returns false. |
| public int indexOf(E obj) | Returns the *index* of the first occurrence of the specified element obj in this list, or -1 if this list does not contain the element. |
| public String toString() | Returns the String equivalent of the list object |

**(B)** Write a class called SLListTest having only main method to test all

functionalities.

**Question 2** Rewrite the following methods of class **SingleLinkedList** without calling any other method:

1. public void add (int index, E item)
2. public boolean remove (E item)

**Solution**: **(A)**

public void add (int index, E item)

{

if (index < 0 || index > size) {

throw new

IndexOutOfBoundsException(Integer.toString(index));

}

if (index == 0) // insert a new node at the front

{

Node<E> temp = new Node<E>(item, head); // create a new node

// and link to the first node

head = temp;

size++;

}

else

{

Node<E> node = head;

for (int i = 0; i < index - 1 ; i++)

node = node.next;

// Insert a new node after the node referenced by *node*

Node<E> temp = new Node<E>(item, node.next);

node.next = temp;

size++;

}

}

**Solution**: **(B)**

public boolean remove (E item)

{

Node<E> current, prev;

if (head == null) // list is empty

return false;

else if ( (head.data).equals(item) ) // first node to be deleted

{

head = head.next;

size--;

return true;

}

else

{

// search the node containing *item* starting from the second node

prev = head; // one node behind current

current = head.next;

found = false;

while (current != null && !found )

{

if ( (current.data).equals(item) )

found = true; //*current* refers to the node having *item* as *data*

else

{

// move *prev* and *current* to the next nodes

prev = current;

current = current.next;

}

} // end while

if (found)

{

// *current* references the node to be deleted

prev.next = current.next; // delete the node referenced by *current*

size--;

return true;

}

else

return false; // item not found

} // end else

}

**Question 3** Write following methods to be included in class **SingleLinkedList** of **Question 1**. You may call other methods of the class **SingleLinkedList** in your method.

1. **removeAll**: Removes all occurrences of the object **obj** from the list, if present and returns true, else returns false.

Method heading: **public boolean removeAll(E obj);**

1. **equals**: Returns true if “this list” and parameter **list** contains the same elements in the same order, else returns false.

Method heading: **public boolean equals(SingleLinkedList<E> list);**

1. **Copy constructor:** creates a copy of the list object of type **SingleLinkedList.**

Method heading: **public SingleLinkedList(SingleLinkedList<E> list);**

**(D) reverse**: The method reverses the elements of the linked list. If the list is empty or has only one node return false, otherwise return true, after reversing the list.

Method heading: **public boolean reverse()**

Example:

Before Calling list: 10 20 30 55 60 70 80

After Calling list: 80 70 60 55 30 20 10

**Solution**: **(A)**

public boolean removeAll(E obj)

{

boolean found = false;

int index;

while ( (index = indexOf(obj)) != -1) {

remove(index); // remove node at the index

found = true; // item found

}

return found;

}

**Solution**: **(B)**

public boolean equals(SingleLinkedList<E> list)

{

if (size != list.size)

return false;

Node<E> current1 = head; // Reference to the nodes of *“this list”*

Node<E> current2 = list.head; //Reference to the nodes of the parameter

// *list*

while (current1 != null)

{

if ( (current1.data).equals( current2.data ) ) {

current1 = current1.next;

current2 = current2.next;

}

else

return false;

}

return true;

}

**Solution**: **(C)**

**// Copy constructor:** Parameter *list* is the original list and copy will be made in

// “this list”

public SingleLinkedList (SingleLinkedList<E> list)

{

Node<E> current, newNode, last;

if (list.head == null) // If original list is empty

{

// Make copy list as empty list

head = null;

size = 0;

}

else

{

current = list.head; // Reference to the original list

size = list.size;

head = new Node<E>(current.data); //First node of copy list

last = head; // Reference to the last node of the copy list, used to

// connect new nodes

current = current.next;

// Copy remaining nodes

while (current != null)

{

newNode = new Node<E>(current.data);

last.next = newNode; // Link last node to the new node

last = newNode;

current = current.next;

} // end while

} // end else

}

**Solution**: **(D)**

public boolean reverse( )

{

if (size <= 1) // list is empty or has only one node

return false;

Node<E> temp, revList;

// Delete node from the front of the current list and add it at the front of

// the reverse list

revList = head;

head = head.next;

revList.next = null;

while(head != null)

{

temp = head;

head = head.next;

temp.next = revList;

revList = temp;

}

head = revList;

return true;

}

**Question 4** Write a method **compareHalfs** to be included in the **SingleLinkedList** class that does not accept any parameter. The method returns true, if all the elements of the first half of the linked list are greater than all the elements of the second half of the list, otherwise it returns false. If the list is empty or does not have even number of nodes, the method returns false.

Hint: Find the minimum of the first half of the linked list and the maximum of the second half of the linked list and compare. Use **compareTo** method of interface **Comparable** to compare data items.

**Solution**:

public boolean compareHalfs()

{

Node <E> ptr = head;

if (head == null || size%2 ! = 0 ) return false;

int half = size/2;

// Finding minimum of the first half of the list

E min = ptr.data;

ptr = ptr.next;

int i;

for (i = 1; i < half; i++)

{

if ( ((Comparable)ptr.data).compareTo( (Comparable) min ) < 0 )

min = ptr.data;

ptr= ptr.next;

}

// Finding maximum of the second half of the list

E max = ptr.data;

ptr = ptr.next;

for (i = half + 1;i < size; i++)

{

if ( ((Comparable)ptr.data).compareTo((Comparable) max ) > 0 )

max = ptr.data;

ptr = ptr.next;

}

if ( ((Comparable)min).compareTo((Comparable)max) > 0 )

return true;

else return false;

}