CSCE 2211 Exercises

Exercises (1): Linked Lists, Stacks & Queues (with some solutions)

Linked Lists

In the following exercises, assume that you have a Simple Linked List class ***List*** with the following definition:

```
template <class keyType, class dataType>
class List
{
        public:
        // Member Functions
        List();         // Create an empty List
        ~List();        // Class Destructor
        // Functions Prototype Definitions
        bool listIsEmpty() const;
        bool curIsEmpty() const;
        void toFirst();         bool atFirst() const;
        void advance();         void toEnd();
        bool atEnd() const;     int  listSize() const;
        void updateData (const dataType & );
        void retrieve (keyType &, dataType &) const;
        void insertFirst (const keyType &, const dataType & );
        void insertAfter (const keyType &, const dataType & );
        void insertBefore (const keyType &, const dataType & );
        void insertEnd (const keyType &, const dataType & );
        void deleteNode();      void deleteFirst();
        void deleteEnd();       void makeListEmpty();
        bool search (const keyType & );
        void orderInsert(const keyType &, const dataType & );
        void traverse();
        private:
        // Node Class
          class node
          {
                public:
                        keyType key;                    // key
                        dataType data;                  // Data
                        node *next;             // pointer to next node
           }; // end of class node declaration
        typedef node * NodePointer;
        NodePointer head, cursor, prev;     // Pointers
    }; // End of class List declaration
```

Group 1
Assume that both key and data fields are integers
Write ***user application*** functions to:

1. Receive a list and to return how many of the nodes contain zeros in the data field.
2. Receive a list and to return how many of the nodes contain even numbers in the key field.
3. Receive a list and to return the sum of data stored in the list.
4. Receive a list (L) and to split it at the approximate middle into two separate lists, returning two new lists (L1) and (L2). After the split, (L) will become empty.
5. Receive a list (L) and to split it into two separate lists, returning two new lists (L1) containing positive data numbers and (L2) containing negative data numbers. After the split, (L) will become empty.
6. Receive two lists and return true if they are identical, ordered in the same way. Assume that the two lists are not empty.
7. Receive a list and a number (m) and to return the list after deleting the first m nodes. If the list size is less than or equal to (m), delete the whole list.
8. Receive a non-negative integer **n** and return a list in which each decimal digit of the number is placed in the data field in a node by itself and the order of the digit is to be put in the key field of the node. The order of the nodes should preserve the order of the digits in the original integer from left to right.
9. Receive a list (L1) and a number (m) and to return another list (L2) that contains an exact copy of the first (m) elements from (L1). If the size of (L1) is less than or equal to (m), copy the whole list into (L2). In any case, L1 should remain unchanged.
10. Receive two lists (L1) and (L2) and append (L2) to the end of (L1), returning the result in a new list (L) without changing the original lists.
11. Receive a list and to return true if the keys in the node are in ascending order.
12. Receive a long integer **(Num)** in the form of a string of digits
    (e.g. "546787656789087664324676543454345 7")
    and to return a list containing its digits in the data field (one digit per node). The order of the digit is to be put in the key field of the node. The digits will be ordered such that the least significant digit will be in the first node and the most significant digit in the last node.

Group 2
Add ***member*** functions to the ***List*** class with the following specifications:

1. A public function ***.Recursive_List_Size( )*** to call a recursive private function to return the size of the list.
2. A public function ***.DisplayList( )*** to call a recursive private function to display the contents of the nodes in the list.
3. A public function ***.SameAs(List &L2)*** to return true if the list is the same as another list ***L2***

Solutions of Some Exercises on Linked Lists

In the application, we define :

typedef List<int , int> LList;

Question (1) Answer:

```
int Fun1 (LList &L)
{
        int count = 0;   int k, int d;
        L.toFirst();
        while (! L.curIsEmpty ( ))
        {  L.retrieve (k,d);      if (d == 0) count++;    L.advance ( );
        }
        return count;
}
```

Question (3) Answer:

```
int Fun3 (LList &L)
{
        int sum = 0;   int k, int d;
        L.toFirst();
        while (! L.curIsEmpty ( ))
        {  L.retrieve (k,d);      sum += d;       L.advance ( );
        }
        return sum;
}
```

Question (5) Answer:

```
void Fun5 (LList &L , LList &L1, LList &L2)
{
        int k, int d;
        L.toFirst();
        while (! L.curIsEmpty ( ))
        {  L.retrieve (k,d);
           if ( d >= 0) L1.insertEnd (k,d); else L2. insertEnd (k,d);
           L.advance ( );
        }
        L.makeListEmpty( );
}
```

Question (7) Answer:

```
void Fun7 (LList &L , int m)
{
        int n;
        n = L.listSize ( );
        if (n <= m) L.makeListEmpty( );
        else for (int i = 1; i <= m; i++) L.deleteFirst ( );
}
```

Question (9) Answer:

```
void Fun9 (LList &L1 , LList &L2, int m)
{
        int n = 0;
        int k,d;
        L2.makeListEmpty( );
        L1.toFirst();
        while (! L1.curIsEmpty ( ) && n < m)
        {  L1.retrieve (k,d);       L2. insertEnd (k,d);
           L1.advance ( );    n++;
        }
}
```

Question (11) Answer:

```
bool Ascending_Order (LList &L)
{
        int x,y,d;  bool ordered;
        n = L.listSize ( );
        if (n == 0) return false; else if (n == 1) return true;
                else
                        ordered = true;  L.toFirst( );
                        L.retrieve (x,d); L.advance( );
                        do
                        {
                                L.retrieveData(y,d);
                                if (x > y) ordered = false;
                                x = y;
                                L.advance( );
                         } while ( (! L.curIsEmpty( )) && (ordered))
                        return ordered;
}
```

Question (12) Answer:

```
void Fun12 (string num, LList &L)
{
        Int d;
        int n  = num.length ( );
        if (n > 0)
                for ( int k = 0; k < n; k++)
                        {
                          d = int (num.at (k)) – int ('0');
                          L.insertFirst (k,d);
                        }
}
```

Group 2

Add ***member*** functions to the ***List*** class with the following specifications:

1. A public function ***.Recursive_List_Size( )*** to call a recursive private function to return the size of the list.
4. A public function ***.DisplayList( )*** to call a recursive private function to display the contents of the nodes in the list.
5. A public function ***.SameAs(List &L2)*** to return true if the list is the same as another list ***L2***

Question (1) Answer:

```
public:
        template <class keyType, class dataType>
        int List <keyType, dataType>:: Recursive_List_Size( )
        {
                return List_Size2(head);
        }

private:
        template <class keyType, class dataType>
        int List <keyType, dataType>:: List_Size2( NodePointer h)
        {
                if (h == NULL) return 0;
                        else return 1 + List_Size2(h->next);
        }
```

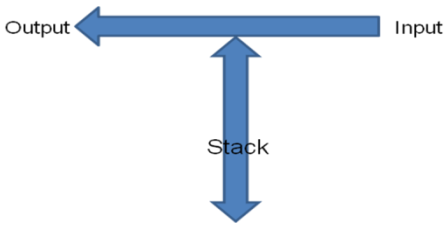
Stacks

Given an input sequence of integers 1, 2, 3, 4, 5, 6 and only three operations on a stack:

1. C: Copy next input directly to output list.
2. S: Push next input onto stack.
3. P: Pop stack and send popped integer to output.

For example, the sequence of operations: 'SSSCCPSPPP' produces the output '453621'.
Notice that the no. of pops (P) must be equal to the no. of pushes (S) and that the sum of pops (P) and copies (C) must be equal to 6.
Write the sequences of operations needed to produce the following outputs:
(a) 123456 (b) 654132 (c) 143526 (d) 123654 (e) 243561 (f) 345621

Answer:



The input sequence is like a wagon train. At the input, the wagons are numbered 1 2 3...
The stack is like a garage; we use it to obtain a certain "permutation" of the wagons on the output. For example for (f):
'SSCCCCPP' produces 345621

The exercise shows that a stack can be used to produce permutations of an input sequence. Can we use a **queue + stack** to permute an input sequence? If yes, work out your own example.

Convert the following infix expressions to postfix notation:
A + B * C A + B – C (A + B) * C (A * B / C) / (A – D)

Answer:
The first will be ABC*+. Work out the others.

Evaluate the postfix expression A B C D - * +, where A = 25, B = 2, C = 18 and D = 13.

Answer:
The stack will produce A+B*(C-D) = 35

In the following exercises, assume that you are developing an "application" using the "Stackt" class. You have no access to the implementation of the class.

Write a function **GetBottom** to retrieve the bottom element from a non-empty stack of characters. The procedure should leave the stack unchanged.

Write a recursive function **Append (S,U)** to append a stack (U) on top of a stack (S) so that U's top is on the top, leaving U empty.

Answer:

```
void Append(Stackt <Type> & s, Stackt <Type> & u)
{
        Type el;
        if(!u.stackIsEmpty())
        {
                u.pop(el);
                Append(s, u);
                s.push(el);
        }
}
```

Write a function **RemoveBlanks** to remove all blanks from a stack of characters, leaving the stack otherwise unchanged. What is the complexity of this function?

Write a function **SwapStack** to exchange the top two elements of the stack, leaving the stack otherwise unchanged.

Answer:

```
void SwapStack(Stackt <char> & s)
{
        char el, temp;
        if(!s.stackIsEmpty())
        {
                s.pop(el);
                if(!s.stackIsEmpty())
                {
                        s.pop(temp);
                        s.push(el);
                        s.push(temp);
                }
                else s.push(el);
        }
}
```

Write a function **AddTop** to replace the top two elements of a nonempty stack of numbers with their sum. If the stack contains one element, we leave the stack unchanged.

Write a recursive function **AddStack** to replace the elements of a nonempty stack of numbers with their sum. If the stack contains one element, we leave the stack unchanged.

Answer:

```
void AddStack(Stackt <int> & s)
{
        int el1, el2;

        if(!s.stackIsEmpty())
        {
                s.pop(el1);
                if(!s.stackIsEmpty())
                {
                        s.pop(el2)
                        s.push(el1 + el2);
                        AddStack(s);
                }
                else s.push(el1);
        }
}
```

Write a function **EqualStacks** to return True if two stacks are identical. The function should leave the stacks unchanged.

In the following, assume that you have access to the implementation of the "Stackt" class:

- Overload the "==" operator to test if two stacks are identical. The operation should leave the stacks unchanged.
- Overload the "+" operator to append a stack on top of another stack.

Answer for the == operator:

```
template <class Type>
bool Stackt<Type>::operator == (const Stackt<Type> & s)
{
        if (s.top != top)   return false;

        for ( int i = 0; i <= top; i++)
             if (stack[i] != s.stack[i])
                  return false;
    return true;
}
```

Queues

In the following exercises, assume that you are developing an "application" using the "Queuet" class. You have no access to the implementation of the class.

Write a function **QueueRear (Q)** to retrieve the rear of a queue (Q), leaving the queue unchanged.

Write a function **NthElement (Q)** to return the n^{th} element in a queue (Q), leaving the queue without that element.

Answer:

```
Type NthElement(Queuet <Type> & Q, int n)
{
        Type el, nth;
        int len = Q.queueLength();
        for(int i=1; i<=len; i++)
        {
                Q.dequeue(el);
                if(i != n)
                        Q.enqueue(el);
                else
                        nth = el;
        }
        return nth;
}
```

Write a boolean function **EqualQueues (Q1 , Q2)** which receives two queues Q1 and Q2 and returns True if they are identical and False otherwise. The function should leave the queues unchanged.

A queue (Q) contains an even number of elements. Write a function **Split (Q,Q1,Q2)** to copy the 1st half of (Q) into (Q1) and the 2nd half into (Q2), leaving the original queue unchanged.

Answer:

```
void Split(Queuet <int> & Q, Queuet <int> & Q1, Queuet <int> & Q2)
{
        int len = Q.queueLength();
        int el;
        for(int i=1; i <= len/2; i++) {
                Q.dequeue(el); Q1.enqueue(el);  Q.enqueue(el);
        }
        for(i=1; i <= len/2; i++) {
                Q.dequeue(el);      Q2.enqueue(el);          Q.enqueue(el);
        }
}
```

Write a function **SwapQueue (Q)** to exchange the front two elements of the queue (Q), leaving the queue otherwise unchanged.

Write a recursive function **Append (Q1,Q2)** to append a queue (Q2) to the end of a queue (Q1), leaving Q2 empty.

Answer:

```
void Append(Queuet <int> & Q1, Queuet <int> & Q2)
{
        int el;
        if(!Q2.queueIsEmpty())
        {
                Q2.dequeue(el);
                Q1.enqueue(el);
                Append(Q1, Q2);
        }
}
```

Write a recursive function **Reverse** (Q) to reverse the order of the elements in a queue (Q).

Write a function **PositionToQueue (Q, Sub, S)** to place into queue (Q) the starting positions, in order, of every occurrence of substring (sub) in the string (S).

Answer:

```
void PositionToQueue(Queuet <int> & Q, string s, string sub)
{
        int pos = s.find(sub);
        int i = 0;
        while(pos>=0 && pos <s.length())
        {
                Q.enqueue(pos + i * sub.length());
                s.erase(pos, sub.length());
                pos = s.find(sub);
                i++;
        }
}
```

In the following, assume that you have access to the implementation of the "Queuet" class:

- Overload the "==" operator to test if two queues are identical. The operation should leave the queues unchanged.
- Overload the "+" operator to append a queue to the rear of another queue.
