CS302

Assignment 1: Stacks of cards

Description

You are to simulate the playing of games of "Accordian" patience, the rules for which are as follows:

Deal cards one by one in a row from left to right, not overlapping. Whenever the card matches its immediate neighbor on the left, or matches the third card to the left, it may be moved onto that card. Cards match if they are of the same suit or same rank. After making a move, look to see if it has made additional moves possible. Only the top card of each pile may be moved at any given time. Gaps between piles should be closed up as soon as they appear by moving all piles on the right of the gap one position to the left. Deal out the whole pack, combining cards towards the left whenever possible. The game is won if the pack is reduced to a single pile.

Situations can arise where more than one play is possible. Where two cards may be moved, you should adopt the strategy of always moving the leftmost card possible. Where a card may be moved either one position to the left or three positions to the left, move it three positions.

In order to solve the problem, you will need to create a custom stack and linked list classes.

```
template <class type>
class myStack
{
public:
  myStack();
  myStack(const myStack<type>&);
  const myStack& operator=(const myStack<type>&);
  ~myStack();

  void push(const type&);
  type peek() const;
  std::size_t getSize() const;
  type pop();
  bool isEmpty() const;
private:
  void clearStack();
  void copyStack(const myStack<type>&);
  type * stackElements;
  std::size_t capacity, size;
};
```

Each member will contain/perform the following

- `type * stackElements` - dynamic array that stores the contents of the stack

- `std::size_t size` - the amount of elements currently stored on the stack
- `std::size_t capacity` - the capacity of the stack (the maximum amount of elements `stackElements` array can store)
- `myStack<type>::myStack()` - default constructor, sets `size = 0`, `capacity = 5`, and `stackElements = new type[capacity]`
- `myStack<type>::myStack(const myStack<type>& copy)` - deep copy constructor, copies all the fields of `rhs` object into the `*this` object (deep copies the `stackElements` array), this function can use the `copyStack` function
- `const myStack<type>& myStack<type>::operator=(const myStac<type>k& rhs)` - assignment operator, deallocates `this->stackElements` array before deep copying `rhs` into `*this` object, this function can use the `clearStack` and `copyStack` functions
- `myStack<type>::~myStack()` - destructor, deallocates `stackElements`, can use `clearStack` function
- `void myStack<type>::push(const type & item)` - inserts `item` on top of `stackElements` at index `size`, then increases size by 1, if the stack is full prior to attempting a push operation, resize the stack by doubling the `capacity` and resize the `stackElements` array
- `type myStack<type>::peek() const` - returns the topmost element of the stack, element at index `size - 1` of `stackElements` array, if the stack is empty then throw an `std::out_of_range` exception
- `std::size_t myStack<type>::getSize() const` - returns `size`
- `type myStack<type>::pop()` - returns the topmost element of the stack, index `size - 1` of `stackElements` array (also removes the topmost element by decreasing `size` value by 1)
- `bool myStack::isEmpty() const` - returns `true` if `size` equals 0 and `false` otherwise
- `void myStack<type>::clearStack()` - deallocates `stackElements` array and sets all the integer fields to 0
- `void myStack<type>::copyStack(const myStack<type>& copyThisStack)` - copies all of the fields of `copyThisStack` (deep copies the `stackElements` array)

You will need to implement the following circular doubly linked list with a dummy node, along with the iterator class

```
template <class type>
class LL
{
private:
  struct node
  {
    type item;
    node * next;
    node * prev;
  };

  void clearList();
  void copyList(const LL<type>&);

  node * dummy;

public:

  class iterator
```

```
  {
  private:
    node * position;

  public:
    friend class LL;
    iterator() : position(nullptr) {}
    iterator(node * n) : position(n) {}
    type& operator*() { return position->item; }
    bool operator==(const iterator& rhs) { return position == rhs.position; }
    bool operator!=(const iterator& rhs) { return position != rhs.position; }

    iterator operator++(int);
    iterator operator++();

    iterator operator--(int);
    iterator operator--();
  };

  LL();
  LL(const LL<type>&);
  const LL<type>& operator=(const LL<type>&);
  ~LL();
  void headInsert(const type&);
  void tailInsert(const type&);
  void headRemove();
  void tailRemove();
  void removeAtPosition(iterator&);
  bool isEmpty() const { return dummy->next == dummy; }
  iterator begin() const { return iterator(dummy->next); }
  iterator end() const { return iterator(dummy); }
};
```

Each member of the iterator class is implemented for you, but here is a description of each member/function

- `node * position` - a pointer that points to a node in the linked list
- `LL<type>::iterator::iterator()` - default constructor that sets the `position` pointer to `nullptr`
- `LL<type>::iterator::iterator(node * n)` - constructor that sets `position` with `n`
- `type& LL<type>::iterator::operator*()` - dereferences the iterator, returns the stack object stored in the node that the iterator points to
- `bool LL<type>::iterator::operator==(const LL<type>::iterator& it)` - compares `*this` with `it`, if they both point to the same node, the function returns `true`, else returns `false`
- `bool LL<type>::iterator::operator!=(const LL<type>::iterator& it)` - compares `*this` with `it`, if they both point to different nodes, the function returns `true`, else returns `false`
- `typename LL<type>::iterator LL<type>::iterator::operator++(int)` - postix operator, moves the iterator over to the next node in the linked list
- `typename LL<type>::iterator LL<type>::iterator::operator++()` - prefix operator, moves the iterator over to the next node in the linked list
- `typename LL<type>::iterator LL<type>::iterator::operator--(int)` - postix operator, moves the iterator over to the previous node in the linked list

- `typename LL<type>::iterator LL<type>::iterator::operator--()` - prefix operator, moves the iterator over to the previous node in the linked list

Each member of the linked list class contains/performs the following:

- `struct node` - each node of the linked list contains a pointer to the next node, the previous node, and an item field that contains a templated item which will be a `myStack<string>` object when declaring the linked list in the main accordian game file
- `node * dummy` - dummy node, `dummy->next` contains the address of the head node, and `dummy->prev` contains the address of the tail node (also the head node's prev pointer contains the address of the dummy node and the tail node's next contains the address of the dummy node)
- `LL<type>::LL()` - default constructor, allocates a node to dummy, and sets dummy's next and prev to itself
- `LL<type>::LL(const LL<type>& copy)` - copy constructor, allocates a node to dummy and sets its next and prev to itself, then calls `copyList(copy)`
- `const LL<type>& LL::operator=(const LL<type>& rhs)` - assignment operator, calls `clearList()` and `copyList(rhs)`
- `LL<type>::~LL()` - destructor, calls `clearList()`, and then deallocates the dummy node
- `void LL<type>::headInsert(const type& item)` - inserts a new node to the front of the linked list and sets this new node's item field with the item parameter (dummy's next must point to this node and this new node points back to dummy, and if this is the only node in the linked list, then dummy's next and prev point to this new node and this new node's next and prev point to the dummy node)
- `void LL<type>::tailInsert(const type& item)` - inserts a new node to the end of the linked list and sets this new node's item field with the item parameter (dummy's prev must point to this node and this new node's next points to dummy, and if this is the only node in the linked list, then dummy's next and prev point to this new node and this new node's next and prev point to the dummy node)
- `void LL<type>::headRemove()` - removes the front node if the list is not empty, make sure to update dummy node's next field correctly, also make sure to handle the case if the linked list only contains one node
- `void LL<type>::tailRemove()` - removes the end node if the list is not empty, make sure to update dummy node's prev field correctly, also make sure to handle the case if the linked list only contains one node
- `void LL<type>::removeAtPosition(LL<type>::iterator & it)` - removes the node that the iterator points to, and then the iterator is updated to point to the next node (next of the node that was deallocated), make sure to handle cases if the iterator points to the front or the end
- `bool LL<type>::isEmpty() const` - returns `true` if there are no nodes in the linked list and `false` otherwise, if dummy-¿next == dummy then the list is empty
- `typename LL<type>::iterator LL<type>::begin() const` - returns `iterator(dummy->next)` which is an iterator object whose position field points to the front node of the linked list
- `typename LL<type>::iterator LL<type>::end() const—` - returns `iterator(dummy)` which is an iterator object whose position field points to the node after the end node (denotes the iterator went off the edge of the linked list)
- `void LL<type>::clearList()` - deallocate the list by either removeHead or removeTail until tne only node in the list is the dummy node
- `void LL<type>::copyList(const LL<type>& copyThisList)` - deep copies `copyThisList` into `*this`, uses tailInsert several times

Input

Input data to the program specifies the order in which cards are dealt from the pack. The input contains a set of 52 cards. Cards are represented as a two character code. The first character is the face-value (A=Ace, 2–9, T=10, J=Jack, Q=Queen, K=King) and the second character is the suit (C=Clubs, D=Diamonds, H=Hearts, S=Spades). The cards are read in through linux redirection.

Output

Output shows the number of cards in each of the piles remaining along with the card on top of each pile and size of each pile after playing "Accordian patience".

Contents of Main

You will declare the following linked list object

```
LL< myStack<string> > accordian;
```

You would need to populate the linked list by reading a card (`string`) and push it onto a temporary stack and then tail insert this stack of size 1 into the linked list (for each card). You can iterate through the stack using the following technique.

```
LL< myStack<string> >::iterator it;

for (it = accordian.begin(); it != accordian.end(); it++)
  cout << (*it).peek() << " ";

cout << endl;
```

Since each node contains a stack, the `*it` returns a stack stored at the node that `it` points to, and then we call the `peek()` function to retrieve the topmost card (`string`) in the stack. We can call any of the `myStack` functions from the iterator (after dereferencing). Suppose you want to compare the card at position i with j (to see if they have the same face or suit), you can do the following

```
LL< myStack<string> >::iterator i, j;

//suppose i is set to a node

j = i;

j--;
j--;
j--;

if ( (*i).peek()[0] == (*j).peek()[0] or (*i).peek()[1] == (*j).peek()[1] )
{
  //pop stack at node i and push into node j

  //check to make sure i is not empty, if it is
  //then remove the node
}

//If there was no match, then set j as i's immediate left neighbor

//and compare their face and suits again
```

```
//all of this would be inside a loop, because if a card moves
//you want to see if it can keep moving to the left
```

Specifications

- Comment your code and your functions
- Do not add extra class members or remove class members and do not modify the member functions of the class
- Do not use global variables
- Make sure your program is memory leak free

Sample Run

```
% g++ myStack.cpp LL.cpp main.cpp
% ./a.out < cards01.txt

6 Stacks: 2H:40 QC:8 3D:1 9S:1 7H:1 5C:1

% ./a.out < cards02.txt

1 Stacks: KS:52
```

Submission

Upload myStack.cpp, LL.cpp, and main.cpp to codegrade

References

- Link to image(s) can be found at http://pngimg.com/imgs/objects/cards/
- Supplemental Video Link https://youtu.be/tSt6R9SdxXg