CPSC 1160 SECTION 001 INSTRUCTOR: GLADYS MONAGAN

ASSIGNMENT #8: LINKED LISTS

DUE DATE WITH BRIGHTSPACE: THURSDAY, NOVEMBER 9 AT 11:50 PM

THE PURPOSE OF THIS ASSIGNMENT IS

- to learn to program using linked lists and to practice programming recursively
- to program functionally (e.g. the beginning of the list is always returned) without instance variables and global variables
- to mimic, to a weak extent, programming in LISP which had as its data structure linked lists

READINGS IN THE TEXTBOOK

Read chapter 20 of the textbook (with the caveats mentioned in class e.g. about templates)

- §20.3 is not covered in detail
- §20.5, §20.7, §20.9 not covered in this course

TO IMPLEMENT

We will represent a linked list of integers using the following C++ structure

```
struct Node {
   int value;
   Node *link;
};
```

Here are some basic functions for linked lists.

The functions `first` and `rest` access the `value` and `link` fields of a `Node` (they are like accessor member functions if Node were a class).
`first` returns the value field of the first node in the list
`rest` returns the rest of the list (a pointer to the rest of the list).

The functions `setfirst` and `setrest` set (i.e. mutate or change) the fields `value` and `link` of a `Node`.

The function `cons` is the constructor.

```
#include <cassert>
//returns the data field of the Node
// if p is the nullptr, we cannot get its data ...exit abruptly
int first( Node* p ) {
   assert( p != nullptr );
   return( p->value );
}

// returns the link field of the Node
// if p is the nullptr, we cannot get its link.... exit abruptly
Node* rest( Node* p ) {
   assert( p != nullptr );
   return( p->link );
}

// change the value field (the first) of the Node pointed to by p
void setfirst( Node* p, int x ) {
   assert( p != nullptr );
   p->value = x;
}

// change the link field (the rest) of the Node pointed to by p
void setrest( Node* p, Node* q ) {
   assert( p != nullptr );
   p->link = q;
}

// construct a new node (allocate memory)
Node * cons( int x, Node* p) {
   return new Node{x, p};
}
```

You must use first, setfirst, rest, setrest, nullptr in **your assignment.**
No marks given for the assignment if you use p->link directly.
You must use the file structure given with the Makefile provided.
Your code must compile to get any marks.

Please put in the README.txt any bugs that you know of and/or any functions that you did not finish.

Implement the following functions for linked lists using these function prototypes. There will no marks awarded for these 4 functions which must be included (done in class already or in the lab)

```
std::istream& operator >> (std::istream& in, Node* & p);
std::ostream& operator << (std::ostream& out, Node* p);
Node *reverse(Node *p);
Node *copy(Node *p);
```

For marks, implement these functions. Document them properly in linkedlist.cpp. The function declarations are in linkedlist.h

- `Node*append( int x, Node* p );`
  The `append(x,p)` function adds (appends) a new node with the value of x to the end of the list `p`. The list with x appended to it is returned as the "result" of append. Do not use reverse for append. In general, appending is not an efficient way to insert something into a linked list. The efficient way is to insert at the front of the list using `cons`. But doing this generates the list in reverse order. Hence we need the reverse function. That's what we did in the lab.
  But in this case, just write the append function.
- `bool searchInOrder( int x, Node* p );`
  The function `searchInOrder(x, p)` looks for a value `x` in the list pointed to by `p`. If `x` is in the list, return true, false otherwise. <u>Note that the linked list `p` is sorted in ascending order</u>. Take advantage of this.
- `bool hasDuplicates( Node *p );`
  The function `hasDuplicates` determines if any elements of the linked list `p` appear more than once (you want to define a helper function `search` or `member` to help in `hasDuplicates`: nothing is assumed about the order of `p` in `hasDuplicates`)
- `bool isLonger( Node *p, Node *q );`
  The function `isLonger(Node *p, Node *q)` determines if the linked list `p` is longer than the linked list `q`. Do <u>not</u> use the function length when implementing `isLonger`. Do not use a counter, either. Do not use arrays. Do not use strings. "Traverse" down both lists at the same time to determine if they are equal in length or if `p` is longer than `q`.
- `bool equal( Node *p, Node *q );`
  The function `equal(p, q)` returns true if p and q have the same elements and in the same order.
- `Node* makeDuplicates(const Node *p);`
  Returns a new list with every element repeated in the order in which appears in p.
- `Node* deleteList( Node* p );`
  If we want to delete <u>every</u> node of a linked list, we call `deleteList` so that the dynamic memory used in the linked list is returned to the heap (freestore). Each node of the list should be removed (one by one). `deleteList` should return the null pointer.
- `Node* array2List(int A[], int n );`
  The conversion routine converts an array of n integers to a linked list of n integers. The order in the array A should correspond to the order of the linked list.
- `Node* mergeTwoOrderedLists( Node *p, Node *q );`
  takes two lists that are <u>already sorted in ascending order</u> and it returns a list that contains the elements of both lists, still sorted in ascending order. It's up to you whether you want to write a "destructive" version that destroys the original linked lists or whether you want to write a version that returns a copy of the Nodes in the original lists.
  <u>Put in the comment of the function whether your function is destructive or not.</u>
- `Node* removeAll(int x, Node *p);`
  The function `removeAll` removes every occurrence of x in p and returns a pointer to the list without any x's. Deallocate the memory of each node that has x in it.
  The list is in no special order.

Some of these functions are easier to program recursively, and some are easier to program iteratively. Test your functions properly. Add more doctest tests. Remember that the empty list could be an argument to any function.
You can overload the functions if you need them for the recursive version. Place the helper functions in the file before they are called

BONUS

Program **<u>all</u>** the functions recursively in the file linkedlist_rec.cpp **<u>and</u>** iteratively in the file linkedlist.cpp

TO SUBMIT WITH BRIGHTSPACE AS A SINGLE ZIP FILE

Before submitting the directory (the folder) with all the files listed below (and before submitting this directory as a single zip file)

```
        make clean
or
         make remove
```

1) **linkedlist.cpp** code with your functions: the functions have been document in linkedlist.h but if you need to explain something about your implementation, add the comments in the linkedlist.cpp file. Use helper functions as needed and place them before they are called in the file.
2) do not modify: **linkedlist.h doctest.h Makefile**
3) **dtest_ll.cpp** add tests but they will not be marked
4) **test_linkedlist.cpp** will not be marked but make sure that it compiles
5) **README.txt** indicating whether you did the bonus
   a) if you did the bonus, put all the recursive functions in **linkedlist_rec.cpp**
   b) put all the iterative functions in **linkedlist.cpp**

Do not get any code from the Internet nor from a friend (see course outline).
Ask for help if you need help with this assignment during office hours or at the Help Centre. Start early. You must be proficient in coding with linked lists in this course.

make
compiles, links and produces the executable dtest_ll.exe (or dtest_ll)

make run   or   make check   if on Linux
compiles the executable and it runs the executable

make test
compiles, links and produces the executable test_linkedlist.exe or test_linkedlist

make clean   or   make remove   if on Linux
removes all the object files and the executables (but not the code)