Winter 2019 CMPS 101-01, Shel Finkelstein
Programming Assignment 1 (PA1)
Due Date: Sunday, January 27, 11:59pm (strict) on GitLab

In this assignment you will create a Doubly-Linked List ADT module, and use it to perform an Insertion Sort. Another goal of this project is to make sure everyone is up to speed with C, especially pointers and structures, and to build an ADT that can be reused in future assignments. Be sure to read the two files ADT.pdf (which discussed ADTs in Java and C) and ADTRemarks.pdf (which has more information about ADTs in C) before proceeding, paying special attention to the second handout. Both files are on Piazza, under Resources→General Resources.

The executable file for this project will be called InsertSortLinked, and will be operated by doing

```
% InsertSortLinked <input file> <output file>
```

at the command prompt `%`. The program `FileIO.c`, which is also posted on Resources→General Resources, shows one way that file input and output can be accomplished in C. Your List ADT module will be contained in a file called List.c, whose operations will be used by InsertSortLinked.c. Your List ADT will be a **doubly-linked** list of Node objects. The underlying data structure for the list will be a doubly linked list of Node objects. You may use the examples Queue.c and Stack.c (provided on Piazza under Resources→QueueExample and Resources→StackExample) as starting points for your design. Your List module will export a `List` type along with the following operations. (Some of these operators are not needed for PA1, but you should implement all of them.)

```
// Constructors-Destructors ----------------------------------------------------

List newList(void);    // returns a List which points to a new empty list object

void freeList(List* pL);  // frees all heap memory associated with its List* argument,
                          // and sets *pL to NULL

// Access functions -----------------------------------------------------------

int length(List L);    // Returns the number of nodes in this List.

int frontValue(List L);// Returns the integer in the front Node.
                       // Precondition:  L has at least one Node on it.

int backValue(List L); // Returns the integer in the back Node.
                       // Precondition:  L has at least one Node on it.

int getValue(Node N);  // Returns the integer in Node N.
                       // Precondition:  N is not NULL

int equals(List A, List B); // Returns 1 if if List A and List B are the same integer
                       // sequence, otherwise returns 0.
```

```
// Manipulation procedures -----------------------------------------------

void clear(List L);          // Resets this List to the empty state.

Node getFront(List L);       // If List is non-empty, returns the front Node, without
                             // changing the List.  Otherwise, does nothing.

Node getBack(List L);        // If List is non-empty, returns the back Node, without
                             // changing the List.  Otherwise, does nothing.

Node getPrevNode(Node N);    // Without changing the List that N is on, returns the
                             // Node that is previous to Node N on its List.  If
                             // there is no previous Node, returns NULL.

Node getNextNode(Node N);    // Without changing the List that N is on, returns the
                             // Node that is next after Node N on its List.  If
                             // there is no next Node, returns NULL.

void prepend(List L, int data);  // Inserts a new Node into List L before the front
                             // Node that has data as its value. If List is empty,
                             // makes that new Node the only Node on the list.

void append(List L, int data);  // Inserts a new Node into List L after the back
                             // Node that has data as its value. If List is empty,
                             // makes that new Node the only Node on the list.

void insertBefore(List L, Node N, int data);  // Insert a new Node into Node N's list
                             // before Node N that has data as its value.
                             // Assume that Node N is on List L.  If Node N is
                             // the front of List L, then the new Node becomes the new
                             // front.
                             // Precondition:  Node N is not NULL

void insertAfter(List L, Node N, int data);  // Insert a new Node into Node N's list
                             // after Node N that has data as its value.
                             // Assume that Node N is on List L.  If Node N is
                             // the back of List L, then the new Node becomes the new
                             // back.
                             // Precondition:  Node N is not NULL

void deleteFront(List L);    // Deletes the front Node in List L.
                             // Precondition:  L has at least one Node on it.

void deleteBack(List L);     // Deletes the back Node in List L.
                             // Precondition:  L has at least one Node on it.

// Other operations -----------------------------------------------------------

void printList(FILE* out, List L); // Prints the values in List L from front to back
                                   // to the file pointed to by out, formatted as a
                                   // space-separated string.
                                   // For those familiar with Java, this is similar
                                   // to toString()in Java.
```

```
// Optional Manipulation procedures  ------------------------------------------------

 void detachNode(List L, Node N);   // This operation is optional.
                                 // Removes N from List L by making the Node before
                                 // Node N link to the Node that's after Node N as its
                                 // next Node, and making the Node after Node N link to
                                 // the Node that's before Node N as its previous Node.
                                 //
                                 // After detachNode, Node N should have NULL as both its
                                 // next and previous Nodes.
                                 //
                                 // Special cases:
                                 //
                                 // If Node N is the front the List L, then the Node after
                                 // Node N becomes the front of List L, which should have
                                 // a NULL previous Node.
                                 //
                                 // If Node N is the back of List L, then the Node before
                                 // Node N becomes the back of List L, which should have
                                 // a NULL next Node.
                                 //
                                 // Precondition: N is not NULL and N is a Node on List L.

 void deleteNode(List L, Node N);   // This operation is optional.
                                 // Deletes Node N from List L.
                                 // Removes N from List L by making the Node before
                                 // Node N link to the Node that's after Node N as its
                                 // next Node, and making the Node after Node N link to
                                 // the Node that's before Node N as its previous Node.
                                 //
                                 // Special cases:
                                 //
                                 // If Node N is the front the List L, then the Node after
                                 // Node N becomes the front of List L, which should have
                                 // a NULL previous Node.
                                 //
                                 // If Node N is the back of List L, then the Node before
                                 // Node N becomes the back of List L, which should have
                                 // a NULL next Node.
                                 //
                                 // Precondition: N is not NULL and N is a Node on List L.

 void attachNodeBetween(List L, Node N, Node N1, Node N2);
                                 // This operation is optional.
                                 // Attaches Node N between Nodes N1 and N2.  Makes N1's
                                 // next Node be N, and N's previous Node be N1.  Makes
                                 // N2's previous Node be N, and N's next Node be N2.
                                 //
                                 // Special cases:
                                 //
                                 // If N1 is NULL and N2 is the front of List L, makes N
                                 // the front of List L, which should have a NULL
                                 // previous Node, and whose next Node should be N2.
                                 //
                                 // If N1 is the back of List L and N2 is NULL, makes N
                                 // the back of List L, which should have a NULL next
                                 // Node, and whose previous Node should be N1.
                                 //
                                 // Preconditions:  N1 and N2 are adjacent nodes on List
                                 // L, with N2 being N1's next node and N1's being N2's
                                 // previous node.  If N1 is NULL, then N2 is the front of
                                 // list L.  If N2 is NULL, then N1 is the back of List L.
```

The above methods allow the client of this ADT to iterate over lists. A typical loop in C iterating from the front to the back of List L would be:

```
aNode = getFront(L);
while(aNode != NULL) {
    x = getValue(aNode);
    // do something with x
    aNode = getNextNode(aNode);
}
```

and a similar loop would be used to iterate from back to front.

What should your Insert Sort program based on a doubly-linked list do? It should read in lines from the input file, where each separate line consists of numeric values (separated by blanks). Each individual line should be sorted using Insert Sort, and the result should be printed to the output file using PrintList.

In other words, the list should initially be empty, and each value read on a line should be appended to the list. Then perform Insert Sort on the values from that line, following the same approach as the array-based Insert Sort program described in class, but using the doubly-linked list operations that you've defined. Do this for all lines in the input file, printing the sorted result.

Your program will be structured in three files: a client program `InsertSortLinked.c`, a List implementation file `List.c`, and a List header file `List.h`. You will also turn in three additional files: `README`, `Makefile`, and `ListClient.c`. This last file will be posted on Piazza under Resources→PA1, and must be submitted unaltered. This file has test data that we will use as inputs to test your program, but we will also test your program using other inputs. Thus you will turn in 6 files in all. Note that these file names are *not* optional. Points will be deducted if you turn in wrongly named files, or extra files such as data files or binary files. Each file you write must begin with a comment block that has your name, user id, and the assignment name (PA1, in this case).

You must submit a `README` file for this (and every) assignment. The `README` file should list each file (other than the `README` file itself) that you submitted, together with a brief description of its role in the project, and any special notes to the people grading your assignment. The `README` is essentially a table of contents for the assignment.

Your `Makefile` will create an executable called `InsertSortLinked` and will include a `clean` utility that removes all object files, including `InsertSortLinked`. A simple Makefile for this assignment is posted under Resources→PA1. You may alter it as you see fit. There was also information about Makefiles in the posting made at the beginning of the term that had subject "Some resources on Unix, Editors, C and Makefiles". And the makefile examples posted on Piazza under Resources→QueueExample and Resources→StackExample may also help you.

Note that the compile operations mentioned in the above Makefile invoke the gcc compiler with the flags `-std=c99`. You may develop your programs on any system, using any editor or IDE. But it is a requirement of this and all other assignments in C that your program compile without warnings or errors under gcc, and run properly in the Linux computing environment on the UNIX Timeshare unix.ucsc.edu provided by ITS. In particular, you should not use the cc compiler. Your C programs must also run without memory leaks. Test them using `valgrind` on unix.ucsc.edu by doing:

```
% valgrind program_name argument_list.
```

You must submit your PA1 project on GitLab following the directions provided on Piazza for PA1. The due date is Sunday, January 27, 11:59pm, and that deadline will be strictly enforced.