Assignment 3

Version 1.00 (last update: Oct. 4, 20:00)

Changes highlighted in yellow

Due date: Thu, Oct 14, 11:59 PM

Summary and Purpose

For this assignment, you will be writing a collection of C functions that operate on linked lists. You will use these to understand the underlying properties of linked lists, their strengths and weaknesses. In particular, you can compare the performance of the linked lists in this assignment to the arrays of Assignment 2.

Deliverables

You will be submitting:

1) A file called `list.h` that contains your function prototypes (see below).
2) A file called `list.c` that contains your function definitions. Do not include any functions written by the instructor.
3) A `makefile` that contains the instructions to compile your code.

You will submit all of your work via git to the School's gitlab server. (As per instructions in the labs.)

This is an individual assignment. Any evidence of code sharing will be investigated and, if appropriate adjudicated using the University's Academic Integrity rules.

Structures for your assignment

You will be working with variables having the following structure which you must declare in your header file.

```
struct Node {
  int data;
  int next;
};
```

This structure represents a node in a linked list (our second data structure used in this course). **data** is an address in the **memsys** memory system where the data of this node is stored, while **next** is an address in the **memsys** system of the next node in the list (or **MEMNULL** if the node is the last in the list).

Additionally, you will be using the following structure to represent a linked list and its meta data.

```
struct List
{
  unsigned int width;
  int head;
};
```

Here, width is the width of the data items stored in the linked list, and head is the address in the **memsys** system of the head of the linked list.

Malloc vs memmalloc

Unless otherwise noted, you should use **memmalloc** to request all the memory required for the functions below from the **memsys** system. All **malloc** and **memmalloc** calls should be **free/memfree**'d at the appropriate time, and definitely before the program ends.

Functions on **Node**s

```
void push( struct memsys *memsys, int *node_ptr,
           void *src, size_t width );
```

(Add an item at the head of the list.) This function will **memmalloc width** bytes of data within **memsys** and copy a **width** bytes of data from **src** into **memsys** using **setval**. It will then initalize a new **struct Node** structure, to hold the address of the first **memmalloc** in **data,** and set the **next** value of the structure to hold the value that was pointed to by **node_ptr**. It will then **memmalloc** memory in **memsys** and use **setval** to copy the structure's data into **memsys**. Finally, it will record the **memsys** address of the **Node** structure in the integer pointed to by node_ptr.

```
void insert( struct memsys *memsys, int *node_ptr,
    void *src, size_t width );
```

(Add an item after a node in the list.) This function will use **getval** to retrieve (from **memsys**) a **Node** structure from the address stored at the location given by **node_ptr**. This function will **memmalloc width** bytes of data within **memsys** and copy a **width** bytes of data from **src** into

memsys using **setval**. It will then initalize a new **struct Node** structure, to hold the address of the first **memmalloc** in **data,** and set the **next** value of the structure to hold the **next** value of the first **Node** that was retrieved (thereby connecting the new node to the rest of the list). It will then **memmalloc** memory in **memsys** and use **setval** to copy the structure's data into **memsys**. Finally, it will record the **memsys** address of the new **Node** structure in the **next** attribute of the original **Node** (thereby attaching the new **Node** after the original **Node**) and use **setval** to write the original **Node** back to **memsys** at the same original location.

```
void delete( struct memsys *memsys, int *node_ptr );
```

(Delete the node after the node referenced.) This function will retrieve the data for the **Node** stored at the address pointed to by **node_ptr** in **memsys**. It will find the next **Node** and delete it after connecting the first **Node** to any following **Nodes**. When it deletes the second **Node**, it will **memfree** both the associated **data** and the **Node** structure within **memsys**. Finally, it will copy the updated first **Node** (with its new **next** value) back into **memsys**.

```
void readHead( struct memsys *memsys, int *node_ptr,
               void *dest, unsigned int width );
```

(Copy data from the head of the list into **dest**.) If the list is empty it should print an error message to the standard error stream and **exit**. Otherwise, this function will copy **width** bytes of data from the **data** pointer in the structure refered to by **node_ptr**, into **dest**.

```
void pop( struct memsys *memsys, int *node_ptr );
```

(Remove an item from the head of the list.) If the list is empty it should print an error message to the standard error stream and **exit**. Otherwise, this function will update the address pointed to by **node_ptr** to the **next Node** in the list, and **memfree** both the original **Node** structure and its **data**.

```
int next( struct memsys *memsys, int *node_ptr );
```

(Return pointer to the pointer to the second item in a list.) If the list is empty it should print an error message to the standard error stream and exit. Otherwise, this function should return the address of the next pointer from the structure pointed to by the pointer that **node_ptr** points to.

```
int isNull( struct memsys *memsys, int *node_ptr );
```

(Tell if a list is empty.) If the list is empty return 1, otherwise return 0.

Functions on `Lists` (a.k.a. Derived function prototypes and descriptions for your assignment)

The following functions should all be implemented by calling the "Basic" functions, above. Most importantly, you should not be interacting with Node structures or Node pointers directly, only by calling the Basic functions.

```
struct List *newList( struct memsys *memsys, unsigned int width );
```

This function should use a regular **malloc** to allocate memory for the **struct List**. If the **malloc** command fails, it should print an error to **stderr** and **exit**. The list structure should be initialized with the given **width** and a **head** equal to **MEMNULL**. This function should return a pointer to the structure.

```
void freeList( struct memsys *memsys, struct List *list );
```

This function should use repeated calls of the **pop** command to **memfree** all the nodes in the **list** from the **memsys** system. Finally, it should **free** the list structure.

```
int isEmpty( struct memsys *memsys, struct List *list );
```

This function should return 1 if the given list is empty and 0 otherwise.

```
void readItem( struct memsys *memsys, struct List *list,
               unsigned int index, void *dest );
```

This function will use the **next** and **readHead** functions, above, to find the **Node** at **index** (where **index=0** for the first node in the list), to retrieve **data** from the given position in the list and store it at **dest**.

```
void appendItem( struct memsys *memsys, struct List *list,
                 void *src );
```

This function will add an element to the end of the list. It will do this by calling the **next** function (above) until **isNull** returns true. Then it will call the **insert** function to add a **Node** contain the **data** located at **src** at the end of the list.

```
void insertItem( struct memsys *memsys, struct List *list,
                 unsigned int index, void *src );
```

This function will use **next** and **insert** calls to insert a **Node** at the given index. If **index** is 0 it will insert the item at the head of the list, if **index** is 1, at the second position, etc.

```
void prependItem( struct memsys *memsys, struct List *list,
                  void *src );
```

This function will use `insertItem` to insert data at position 0, but calling the `push` function.

```
void deleteItem( struct memsys *memsys, struct List *list,
                 unsigned int index );
```

This function will use `next` and `delete` calls to remove the node at the given `index`. Pay special attention to the item at `index=0`.

The Last 20%

The above, constitutes 80% of the assignment. If you complete it, you can get a grade up to 80% (Good). The rest of the assignment is more challenging and will allow you to get a grade of 80-90% (Excellent) or 90-100% (Outstanding). Make sure you complete the first part well, before proceeding to the following additional part.

Write the following functions:

```
int findItem( struct memsys *memsys, struct List *list,
              int (*compar)(const void *, const void *), void *target );
```

This function will retrieve elements from `list` using `readHead` (above) starting with the first element in the list and proceeding incrementally by calling `next` (above). For each element it will apply the `compar` function to `target` and the retrieved element. If the `compar` function returns 0 (indicating a match), this function should return the index of the matching element. If they compar function returns a non-zero value (indicating a mismatch) it should proceed with the next element. If they function processes the entire list without finding a match, it should return a value of -1.
This function should use a regular `malloc` to create a temporary variable of width `bytes` to store data values extracted from the list as it is searched.

You can write additional helper functions as necessary to make sure your code is modular, readable, and easy to modify.

Header File

Use the #ifndef…#define…#endif construct in your header file to prevent problems if your header file is included multiple times.

Testing

You are responsible for testing your code to make sure that it works as required. The CourseLink web-site will contain some test programs to get you started. However, we will use a different set of test programs to grade your code, so you need to make sure that your code performs according to the instructions above by writing more test code.

Your assignment will be tested on the standard SoCS Virtualbox VM (http://socs.uoguelph.ca/SoCSVM.zip) which will be run using the Oracle Virtualbox software (https://www.virtualbox.org/wiki/Downloads). If you are developing in a different environment, you will need to allow yourself enough time to test and debug your code on the target machine. We will NOT test your code on YOUR machine/environment.

Full instructions for using the SoCS Virtualbox VM can be found at: https://wiki.socs.uoguelph.ca/students/socsvm.

Makefile

You will create a makefile that supports the following targets:

`all:` this target should generate list.o.

All programs and .o files must be compiled with the –std=`c99 -Wall -pedantic` options and compile without any errors or warning.

`clean:` this target should delete all `.o` files.

`list.o:` this target should create the object file, `list.o`, by compiling the `list.c` file.

All compilations and linking must be done with the `-Wall -pedantic -std=c99` flags and compile and link **without any warnings or errors**.

Git

You must submit your .c, .h and makefile using git to the School's git server. Only code submitted to the server will be graded. Do ***not*** e-mail your assignment to the instructor. We will only grade one submission; we will only grade the last submission that you make to the server and apply any late penalty based on the last submission. So once your code is complete and you have tested it and you are ready to have it graded make sure to commit and push all of your changes to the server, and then do not make any more changes to the A2 files on the server.

Academic Integrity

Throughout the entire time that you are working on this assignment. You must not look at another student's code, now allow your code to be accessible to any other student. You can share additional test cases (beyond those supplied by the instructor) or discuss what the correct outputs of the test programs should be, but do not share ANY code with your classmates.

Also, do your own work, do not hire someone to do the work for you.

Grading Rubric

push	2
insert	2
delete	1
readHead	1
pop	1
next	1
isNull	1
newList	1
freeList	1
isEmpty	1
readItem	1
appendItem	2
insertItem	2
prependItem	1
deleteItem	2
style	2
makefile	2
findItem	6
Total	30

Ask Questions

The instructions above are intended to be as complete and clear as possible. However, it is YOUR responsibility to resolve any ambiguities or confusion about the instructions by asking questions in class, via the discussion forums, or by e-mailing the course e-mail.