Assignment 3

Version 1.02 (last update: Oct. 5, 19:45)
Changes highlighted in yellow
Due date: Thu, Oct 15, 11:59 PM

Fixed some typos, clarified there are two mallocs in push, and modified the functions to consistently use unsigned integers. Replaced the figures with better ones.

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. You will also use double pointers extensively to pass pointers into functions by reference (so they can be changed).

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.
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
{
  void *data;
  struct Node *next;
};
```

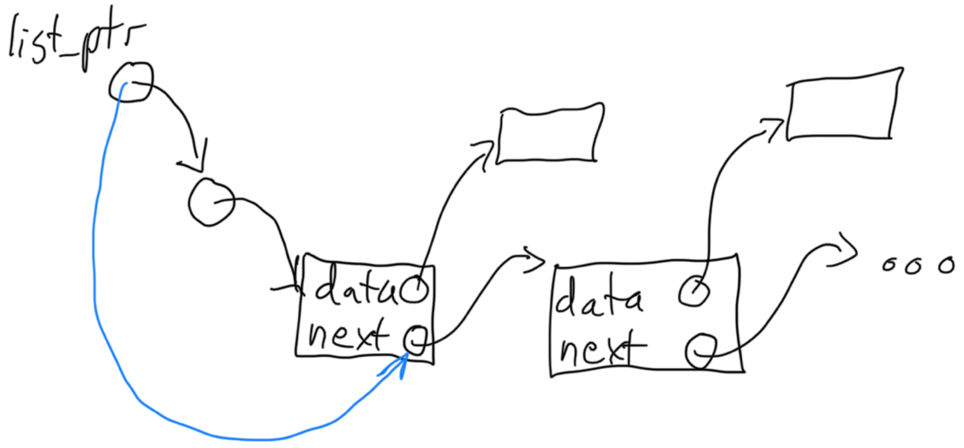
This structure represents a node in a linked list (our second data structure used in this course). **`data`** is a pointer to the data stored at this node, while **`next`** is a pointer to the next node in the list (or null if the node is the last in the list).

Additionally, you will be using the following structure to measure the performance of your code and count the number of memory read, memory write, malloc and free operations.

```
struct Performance
{
  unsigned int reads;
  unsigned int writes;
  unsigned int mallocs;
  unsigned int frees;
};
```

Double Pointers

This assignment makes extensive use of double pointers. The following diagram illustrates how the double pointer references another pointer.



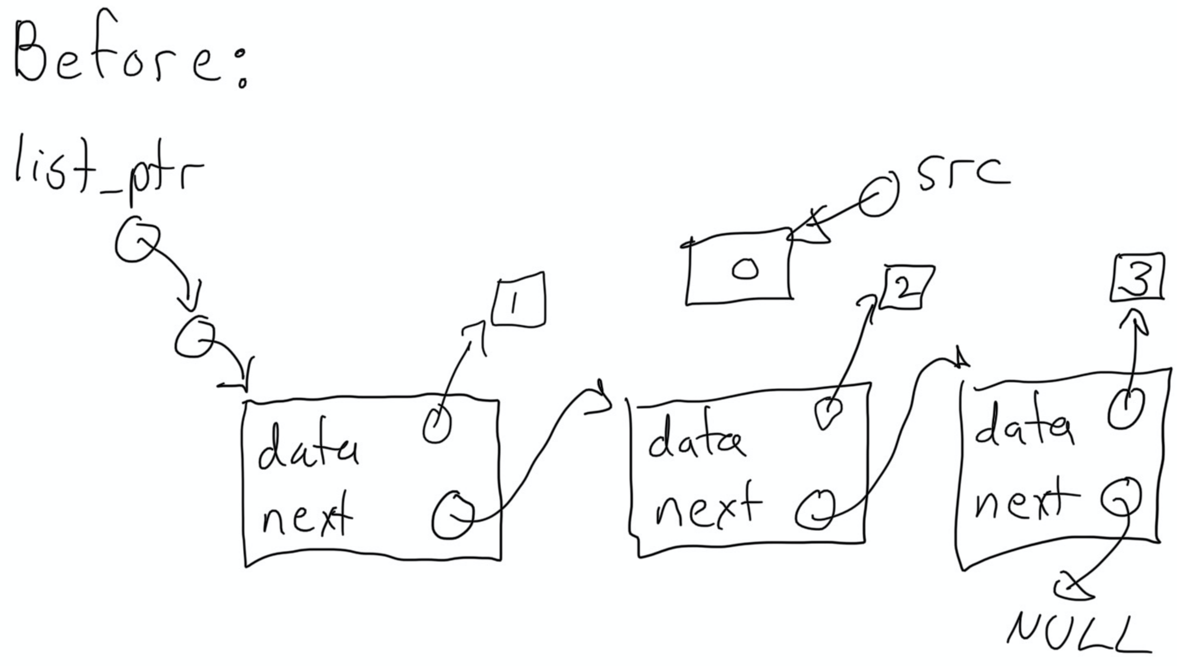
Basic function prototypes and descriptions for your assignment

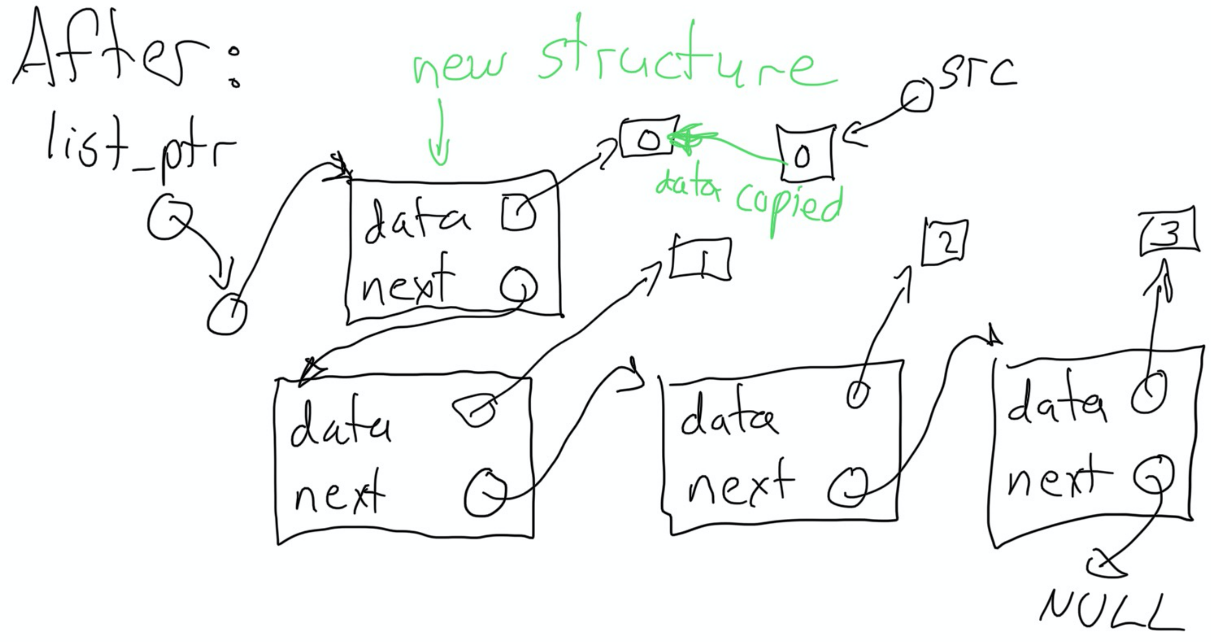
`struct Performance *newPerformance();`

This function will allocate sufficient memory for a **`Performance`** structure, set **`reads`**, **`writes`**, **`mallocs`**, and **`frees`** to zero (yes, I realize there is technically one malloc in this function) and return the address of the structure. Your function should print an error message to the standard error stream and **`exit`** if the **`malloc`** function fails.

`void push( struct Performance *performance, struct Node **list_ptr, void *src, unsigned int width );`

(Add an item at the head of the list.) This function will **`malloc`** a new **`struct Node`** structure, **`malloc width`** bytes of data and save the address in **`data`**, copy **`width`** bytes of data from the parameter **`src`** to the adress **`data`** in the new **`Node`** structure, and set the **`next`** pointer of the structure to be equal to the value pointed to by **`list_ptr`**. Finally, it will store the address of the structure in the pointer that is pointed to by **`list_ptr`**. If the **`malloc`** fails, it should print an error message to the standard error stream and **`exit`**. **`mallocs`** and **`writes`** in the **`performance`** structure should both be incremented by 1.



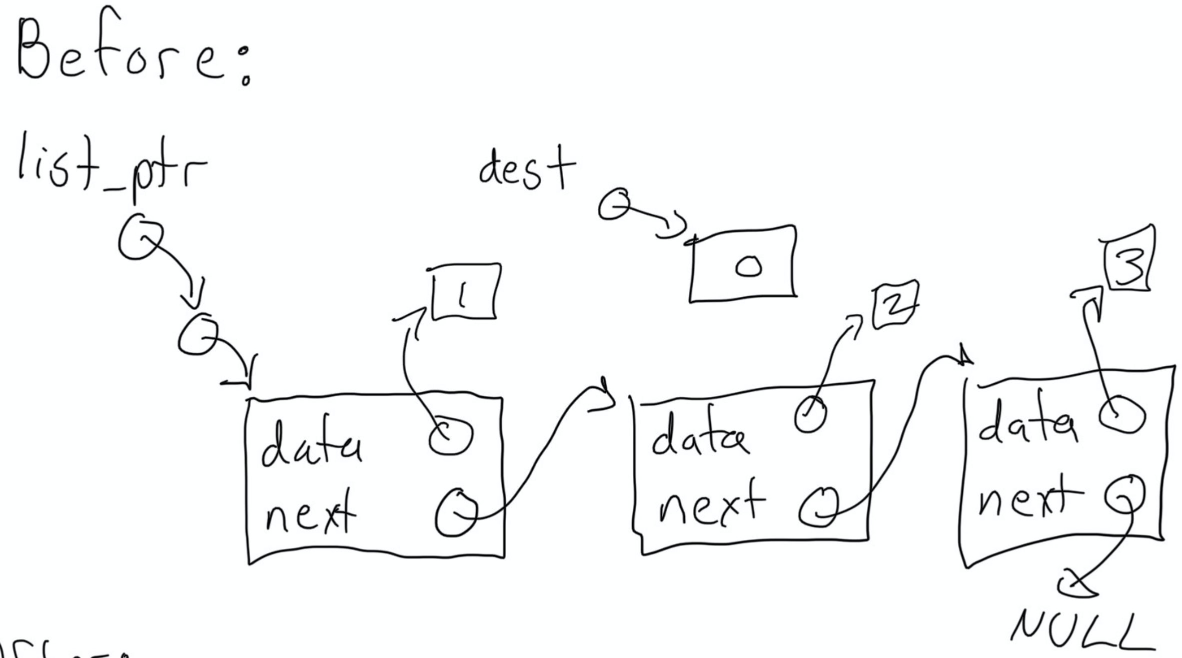


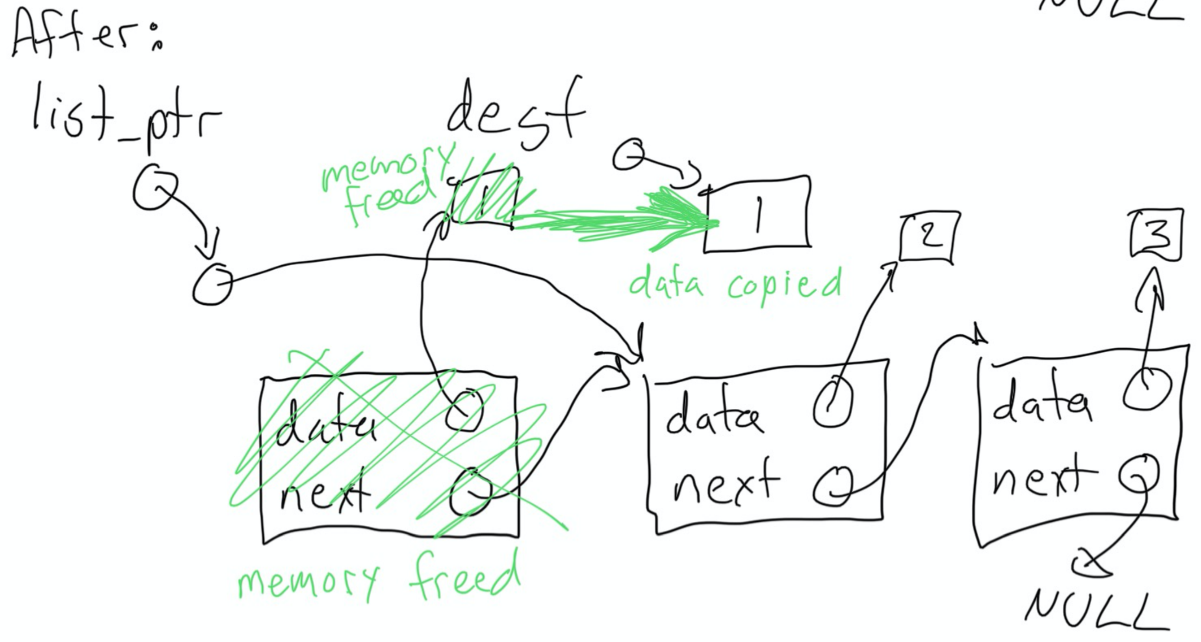
`void readHead( struct Performance *performance, struct Node **list_ptr, void *dest, unsigned int width );`

(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 pointed to by the pointer that **`list_ptr`** points to, into **`dest`**. **`reads`** in the **`performance`** structure should be incremented by 1.

`void pop( struct Performance *performance, struct Node **list_ptr, void *dest, unsigned int width );`

(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 copy **`width`** bytes of data from the **`data`** variable in the structure whose address is stored in the pointer pointed to by **`list_ptr,`** to the address given by **`dest`**. It should update the pointer pointed to by **`list_ptr`** to the next node in the list, and free the node structure that used to be first. **`frees`** and **`reads`** in the **`performance`** structure should both be incremented by 1.





`struct Node **next( struct Performance *performance, struct Node **list_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 **`list_ptr`** points to. **`reads`** in the **`performance`** structure should be incremented by 1.

```
int isEmpty( struct Performance *performance, struct Node **list_ptr );
```

(Tell if a list is empty.) If the list is empty return 1, otherwise return 0. Do not modify **`performance`**.

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.

```
void freeList( struct Performance *performance, struct Node **list_ptr );
```

This function will pop items off the list until the list isEmpty.

```
void readItem( struct Performance *performance, struct Node **list_ptr,
unsigned int index, void *dest, unsigned int width );
```

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

```
void appendItem( struct Performance *performance, struct Node **list_ptr,
void *src, unsigned int width );
```

This function will add an element to the end of the list. It will do this by calling the **`next`** function (above) until **`isEmpty`** returns true. Then it will call the **`push`** function to add the item at the end of the list.

```
void insertItem( struct Performance *performance, struct Node **list_ptr,
unsigned int index, void *src, unsigned int width );
```

This function will use **`next`** and **`push`** 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 Performance *performance, struct Node **list_ptr,
void *src, unsigned int width );
```

This function will use **`insertItem`** to insert data at position 0.

```
void deleteItem( struct Performance *performance, struct Node **list_ptr,
unsigned int index );
```

This function will use **`next`** and **`pop`** calls to remove the node at the given inex.

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 Performance *performance, struct Node **list_ptr, int
(*compar)(const void *, const void *), void *target, unsigned int width );
```

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.

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 (Lecture 02) 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

newPerformance	1
push	2
readHead	1
pop	2
next	1
isEmpty	1
freeList	2
readItem	1
appendItem	1
insertItem	2
prependItem	1
deleteItem	1
style	2
makefile	2
findItem	5
Total	25

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.