CSCI 245 Implementing ArrayList and LinkedList

Overview

In this lab, we will be creating C data structures similar to Java's ArrayList and LinkedList; both of these implement the List data structure, which is an ordered sequence of elements.
An ArrayList functions similarly to a "variable length array", in that access to one of the elements in the list takes the same amount of time no matter how many elements there are in the list. Inserting into an ArrayList can take time proportional to the number of elements in the list, since its size might need to be increased in order to hold the new element.
A LinkedList, on the other hand, takes the same amount of time to insert a new element into the list, while finding an element at a particular element takes time proportional to the number of elements in the list. Note that this is the exact opposite of the ArrayList!
You will need to make extensive use both pointers and the malloc and free memory management functions in order to complete this exercise.

Part 1 – Implementing an ArrayList class
Getting Started
1. Copy the folder \\torque3\245\McFall\KR\array-list into an appropriate location in your torque3 space.
2. Start a Putty session to rizzo, and change directories to the array-list directory you just created.
3. Type the command
make get
and press Enter. This will cause a series of compilation steps to occur.
4. Execute the program by typing ./get and pressing Enter. This will cause the program to be executed; in its current state it doesn't do too much of anything. You can find out what is supposed to happen by executing the solution (on rizzo), by typing
/home/samba/dept/CompSci/course/245/McFall/KR/array-list-solution
and then pressing Enter.

CSCI 245 Implementing ArrayList and LinkedList
Implementing the ArrayList functions

5. Open the file ArrayList.h and write down the two components of the list structure, including their types and names:
Component 1:
Type: _____
Name: _____ Component 2:
Type: _____ Name: _____
Note that since C is not an object-oriented language, when we create functions that implement operations on an Array List, we have to pass the array list to be operated on as a parameter.
Task 1 – Implementing the get function

6. In the space below, write down the prototype for this function (you can find it in ArrayList.h), and give an interpretation of the types of its arguments: Prototype:

Argument 1: _____
Argument 2: _____
Compare your answers with a neighbor to make sure you agree.

7. Open the file ArrayList.c and write an implementation of the get function. Remember that you can treat a pointer like it's an array, and that you can use:

```
list->address
```

as a shortcut for

```
(*list).address
```

Once you have an implementation, you can recompile by typing make get and pressing Enter. Run the program by typing ./get and pressing Enter. Check the output of your implementation against that of the solution to verify that your get implementation is correct.

Task 2 – Implementing the initialize function

8. Copy the prototype from ArrayList.h into ArrayList.c and then write an implementation of the function based on the description in the comments. You will have to use malloc twice here:

- To allocate enough memory to hold a struct list; the location returned by malloc for this will be the address returned from the initialize function.
- To allocate the memory to hold the array contents; the address member of the newly allocated struct list should point to this memory.

You will also need to set the capacity value for the newly allocated struct list appropriately.

To test your implementation, execute make initialize. This will generate an executable named initialize that uses your implementation of both get and initialize, while using my implementations of the other functions.

Task 3 – Implementing the set function

9. As stated in the comments describing the function in ArrayList.h, this function should change the value stored at a particular index in the array, extending the capacity of the array if necessary.

If the capacity of the array needs to be increased, you should be sure to note the following statement from the free man page:

This means that you can't pass free an address that is in the middle of a block of previously allocated memory. Instead, you'll need to allocate some new memory, copy the appropriate contents into it, and then free the entire block of memory that was previously allocated.

You can execute make set to generate an executable named set which can be used to test your implementation of the set function.

The free() function frees the memory space pointed to by ptr, which must have been returned by a previous call to malloc(), calloc() or realloc(). Otherwise, or if free(ptr) has already been called before, undefined behavior occurs. If ptr is NULL, no operation is performed.

Task 4 – Implementing truncate

10. Next comes the implementation of truncate. Here's the description of this

function from ArrayList.h:

```
/*
  Reduces the capacity of the list pointed to by lst
  to the given value; any data that exists with
  index -value- or greater are lost
*/
void truncate(struct list* lst, int value);
```

In all invocations of truncate, you will need to first allocate a new block of memory using malloc. 1 After that allocation, you will need to copy the contents of the original block of memory to the new one. After doing so, you can use free to release the original block of memory.

Task 5 – Implementing the compact function

11. Your final task is to implement the compact function. The goal of this function is to reduce the amount of memory needed to hold the current capacity of the array list, possibly changing the indexes of the array while doing so. See the prototype and comments in ArrayList.h for more details.

The executable for this task can be generated by executing make compact.

1 Except the case where the new value of capacity is the same as the old one, in which case this function doesn't have to do anything.

CSCI 245 Implementing ArrayList and LinkedList

Part 2 – Implementing a Linked List

A linked list is a second data structure that represents an ordered list of elements. Rather than requiring a contiguous block of memory, the linked list allows each element in the list to be stored anywhere in memory. It does this by representing each element in the list with a node containing two parts: the data associated with the node, and a pointer to the next node in the list. The figure below gives an illustration. ***figure not in file***

In the above example, the data item is shown in the left portion of the node, and the next item is the right portion of the node. The data item is a char* (null terminated string).

The list is represented by a pointer to the first node in the list, which is typically referred to as the "head" of the list. In our example, the first node, labeled Node 1, has its data item pointing to a location in memory which contains the null terminated string "Hello". Its next pointer points to the array of memory allocated for Node 2. This process continues until the end of the list is reached; this is indicated by a value of NULL for the next pointer in Node 4.

Getting Started

1. Copy the folder \\torque3\245\McFall\KR\linked-list into an appropriate location in your torque3 space.
2. Start a Putty session to rizzo, and change directories to the linked-list directory you just created.
3. Compile the code by typing make and then pressing Enter. This produces an executable named main; run it by typing ./main followed by Enter.

CSCI 245 Implementing ArrayList and LinkedList

Task 1 - Implementing a print function

Currently, the main method in main.c has code that prints out the elements of the linked list pointed to by the variable list.

Take this code and move it into a function named print in LinkedList.c (with a prototype in LinkedList.h). Call this function in main.c to ensure that the program still works correctly.

Task 2 – Implementing a contains function

4. Add the following function prototype into LinkedList.h:

```
/*
 Returns 1 if the list named "list" contains a node whose
 value is sValue, 0 if not
 */
 int contains (struct LinkedList* list, char* sValue);
```

5. Write an implementation of the contains method in LinkedList.c.

You can use the process of printing out the nodes in the print function as an example; start at the head of the list, and follow the next pointers until you either find a node with the desired data value, or you reach the end of the list.

You'll want to use the strcmp function to compare the values in the list nodes with the string pointed to by sValue; add #include<string.h> to LinkedList.c in order to get the appropriate prototype, and remember that strcmp returns 0 when the two strings being compared are equal

6. To verify your implementation, add printf statements in main.c that indicate the return value of contains for the search strings "Hello", " " and "world" These cases ensure that the code correctly locates elements at the beginning, interior, and end of the list.

Then, add a printf statement that outputs the return value of contains for the string "Goodbye", ensuring that the result is correct for a value that is not in the list.

Task 3 – Implementing a copy function

Next we'll add a method to make a copy of a linked list.

7. Add a declaration for the copy function into LinkedList.h like this: struct LinkedList* copy (struct LinkedList* src);

8. Add a stub implementation of the function into LinkedList.c as follows:

```
 struct LinkedList* copy (struct LinkedList* src) {
   return src;
}
```

9. Add code to the end of main.c that uses the copy function to create a copy of the list pointed to by list, and then prints out that list using the print function.

10. Add code in main.c (after making the copy but before printing it) so that the word new is inserted after the space in the list named list, and re-run the program.

You should see the additional word printed out in the copied list, since our copy function currently doesn't make a copy, but instead returns a pointer to the original list.

11. Modify the implementation of the copy method so that it actually makes an independent copy of the list pointed to by src (the data pointers can be the same; in other words, you don't have to make copies of the strings themselves). I used the create and insertAfter functions to help in my implementation.

12. Using printf statements or the debugger, obtain the addresses of each of nodes in each list to help you understand what is happening.

Task 4 – Implementing an insertBefore function

Next we will implement a function that inserts a new node before an existing node.

The prototype for the method will be:

```
/**
 Inserts a new node with value "value" into the list
 referenced by "list" so that the new node comes before
```

```
  the node with value "sValue"
  Returns 0 if no node with value "sValue" can be found
  in the list.  Returns -1 if no memory can be allocated
  for a new node, and returns 1 if the new node
**/
int insertBefore (struct LinkedList* list, char* sValue,
char* value);
```

You will need to take special care to handle the case when the element is to be inserted before the existing head of the list. Add code to main.c that verifies that this method works correctly.

Task 5 – Implementing a delete function

The prototype for our delete function will be similar to the other functions we have implemented:

```
/**
  Deletes the node in the list with value "value"
  Returns 0 if such a node cannot be found; otherwise
  returns 1
**/
int delete(struct LinkedList* list, char* value);
```

You will again want to be careful to handle the case where the node being deleted is the head node of the list.

Also, be sure that you call the free(3) function to release the memory associated with the node to be deleted. Failure to do so will result in memory that has been allocated to your program, but is not actually referenced by any variable, therefore making it inaccessible.

This is called a memory leak, and is one of the major reasons why languages like Java and C# provide automatic “garbage collection” to free the programmer from having to worry about releasing unused memory.

Task 6 – Preview of functional programming using visit

Soon we will be looking at a language called Racket, which is an instance of a programming language paradigm called functional programming. One hallmark of functional programming is that functions are “first class objects.2” This means in part that functions can be used in all the places that variables can be used. For our example, we’ll use a function as a parameter to a function named visit, allowing the supplied function to be applied to each of the nodes in our linked list.

Here’s the prototype for the visit function, which should be put in LinkedList.h: /**

```
  Visits each node of the list, calling the function
  Pointed to be "visitFunc" for each node
**/
void visit(struct LinkedList* list,
        void (*visitFunc) (struct Node* value));
```

The second parameter to visit is a pointer to a function that returns void and takes a pointer to a struct Node as a parameter. The name of the function pointer parameter is visitFunc.

2 See https://en.wikipedia.org/wiki/First-class_citizen for further discussion

The syntax of function pointers is a bit hard to remember, at least for me. I end up looking it up almost every time I use it! Fortunately, calling a function referenced by a function pointer looks like it calling a function using its name. For example, in the body of visit, we could write:
visitFunc(list->head);
to call the function pointed to by visitFunc with the first node in the list as the parameter to visitFunc.
Use visit to mimic the functionality of the print function. This will require the following steps, both in main.c:
1. Implementing a function named printNode that takes a struct Node* as a parameter, and prints the value of that Node.
2. Calling the visit function, passing printNode as the visitFunc parameter.
Implement this, and then print the list using visit by calling it in main.c.
A last modification
Suppose we want to be able to count the total number of characters in our linked list. At first glance, it seems like visit should provide a nice way to do this, since we can use it do anything we want on each node of the list. A problem, however, is that we won't have a place to store the character count between each call to visitFunc. Below are several modifications to our program that facilitate this.
LinkedList.h

```
void visit(struct LinkedList* list,
        void (*visitFunc) (struct Node* value, void* d),
        void* data);
```

LinkedList.c

```
void visit(struct LinkedList* list, void (*visitFunc)
(struct Node* value, void* d), void* data) {
  struct Node* current = list->head;
  while (current != NULL) {
    visitFunc(current, data);
    current = current->next;
  }
}
```

main.c

```
void totalLength(struct Node* node, void* data) {
  int* currentLength = (int *) data;
  if (node->data != NULL) {
    *currentLength += strlen(node->data);
  }
}
```


We can then count the characters in a list by invoking visit from the main function, using totalLength as the function parameter to vist:

```
int length = 0;
visit (list, totalLength, &length);
printf("Total characters = %d\n", length);
```

Make sure you understand how this code works – in particular, the presence of the parameters named d and data in the prototype for the visit function, and how the data parameter is used in the totalLength function.
Then add this code to your version and verify that it works correctly.