Week 5 Lecture Notes

Linked Lists using C

Contents

[Dynamic Memory Allocation 1](#_Toc45193277)

[Dynamic linked list 3](#_Toc45193278)

# Dynamic Memory Allocation

Allocating memory statically, which is what we’ve been doing, means the memory for a variable such as an array is allocated when your program compiles. That’s why we have always had to know how big our array needs to be, so the compiler knows how much space to reserve for the array. We can’t change its size as our program executes. The following statement statically allocates 100 elements for an array of integers along with the array variable, a, itself which holds the address of the first byte of element 0.

int aPtr[100];

Allocating memory dynamically means getting space for that variable when your program executes. To dynamlically allocated an array of 100 integers instead of statically as above, we could declare a pointer to an integer like so:

int \*aPtr;

and not say what  it points to just yet. Later in our code, we could dynamically allocate memory for the array. Space for the array wouldn’t exist in a.out (or whatever our executable file is called). It wouldn’t be created when we compiled. It would be created when we executed our program. We might not even know how many elements we need at compile time. Assume as the program executes that the variable count contains the number of integers needed, then the following statement could be used to dynamically allocate an array of count integers:

aPtr = (int\*) malloc( sizeof(int)\*count );

A bit more about variables and what’s called their storage class:

* A variable declared globally or in main exists during the entire execution of the program and is called a static variable. Space for it is allocated in your executable file (a.out or whatever you called it).
* An automatic variable is allocated and deallocated automatically when the program enters/exits the variable’s scope. Function parameters and local variables inside functions other than main are examples of this, and they are allocated on the program’s stack whose space is also in your executable file which means this space is also created at compile time.
* Variables that you allocate dynamically do not have any space in your executable file. In general, what’s called the heap (also known as the free store) is used by the operating system. When a call to a library function such as malloc executes, the operating system finds some free memory in the heap, marks it as belonging to your program until you free it, and gives the address to malloc which then returns this address. If a program that dynamically allocates memory does not deallocate it, the entire system is adversely affected since there’s only one heap and other programs also need to us it. Failing to free dynamically allocated memory is called a memory leak.

There are several C functions for dynamically allocating memory. All prototypes are in stdlib.h. The type size\_t used in these functions is a type definition in stdlib.h. It is an unsigned integer.

* void\* malloc( size\_t numberOfBytes): allocates numberOfBytes bytes and returns a pointer to it or NULL if the function failed. Does not initialize memory. <http://www.tutorialspoint.com/c_standard_library/c_function_malloc.htm>
* void\* calloc( size\_t numberOfItems, size\_t sizeOfEachItem ): returns a pointer to numberOfItems \* sizeOfEachItem number of bytes. Returns NULL upon failure. Initializes all memory bytes to 0. <https://www.tutorialspoint.com/c_standard_library/c_function_calloc.htm>
* void\* realloc( void\* ptr, size\_t size ): resizes the memory block pointed to by ptr that was previously allocated by malloc or calloc. Returns the new pointer value or NULL if the function failed. <http://www.tutorialspoint.com/c_standard_library/c_function_realloc.htm>
* void free( void\* ptr): deallocates memory pointed to by parameter ptr.  <http://www.tutorialspoint.com/c_standard_library/c_function_free.htm>. **Always free dynamically allocated memory**!

Note that malloc, calloc, and realloc have return types of void\* - a pointer to “the official nothing type”. Well, what if the return type was int\*? Then, we couldn’t use malloc to allocate any doubles. Or any Employee structures. So, these functions return a pointer to “whatever” and we can cast the return type to be a pointer to the kind of thing that we want. Technically the type casting isn’t necessary if we use a C compiler, but it is if we use a C++ compiler.

Let’s dynamically allocate an array of integers with the aPtr variable declared above then do something with it.

aPtr = (int\*) malloc( sizeof(int)\*10 );

or using calloc if we want all memory bytes initialized to 0:

aPtr = (int\* ) calloc( 10, sizeof(int ) );

Then, we can do the same things that we could do to a statically (or automatically) allocated array. For example,

for( i = 0; i < 10; i++ ) {

aPtr[i] = 2\*i;

}

Don’t forget to free when you’re done with the array! In many cases it’s also a good idea to set the pointer to NULL after memory has been freed.

free( aPtr );

aPtr = NULL;

**Pair Programming 5a:** copy ~caarnold/cisp1020/week5/gradesStart.c and update it to ask the user how many grades he/she has, dynamically allocate an array of that many. Then, get that many grades from the user and put them into your dynamically allocated array. Calculate the average and print.

How do we dynamically allocate a single structure?

Employee\* ePtr;

ePtr = (Employee\*) malloc( sizeof(Employee) );

How do we dynamically allocate an array of structures?

Employee \*eArray;

eArray =(Employee\*)malloc( sizeof(Employee)\* numElements );

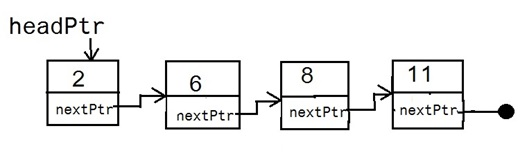
or

eArray =(Employee\*)calloc( numElements, sizeof(Employee) );

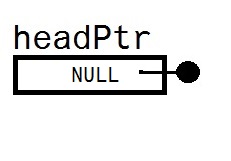
Pitfalls:

* Not checking to make sure memory allocation was successful, although, if memory allocation is not successful, the program can’t do anything about it, and the entire system is in trouble. This is a pretty catastrophic error.  
  ePtr = (Employee\*) malloc( sizeof(Employee) );  
  // if the OS couldn’t dynamically allocate   
  // save everything, write an error log, we’re going down!  
  if ( ePtr == NULL ) {
* Not freeing memory you have allocated as soon as you are done with it. This is called a memory leak and can cause your program and even the entire system to perform poorly. It can affect every other program running on the computer.
* Using a pointer as if it still points to something when you have, in fact, already freed the memory it used to point to. Usually results in a segmentation fault. The following code example checks to see if a pointer actually points to anything before using it. It doesn’t dereference the pointer until it checks to ensure it points to something. This code won’t work if ePtr was not initialized to NULL or set to NULL after it was freed which is one why it’s a good idea to initialize/set pointers to NULL.  
  if ( ePtr != NULL && ePtr->age >= 18 ) {

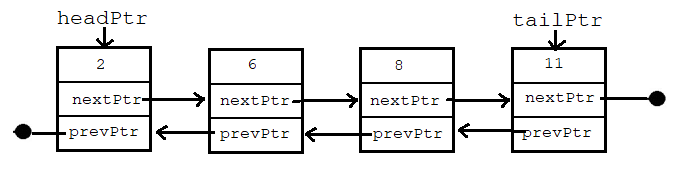
# Dynamic linked list

What if we can’t know how many items we want in an array until the moment we get a new item? And, what if we’d like to be able to do operations like inserting in order without having to shift a bunch of elements in the array to make room for the new one? We can use a linked list. Below is a singly linked list of integers. It has a “headPtr” which is a pointer that points to the beginning of the list- the first “node” in the list. A node typically has an identifying key and data associated with that key For example, a Pellissippi P-number and information about that particular student. This data would be stored in a structure inside the node. Each node also has a pointer to the next node in the list, nextPtr. The image below is a singly linked list because each node has a single pointer which points to the next node in the list. In the singly linked list below, the headPtr points to a the first node in the list which has a 2 in it. That node’s nextPtr points to the next node in the list with a 6 in it. The node with a 6 in it has a nextPtr that points to the node with the 8 in it whose nextPtr points to the next node in the list which has an 11 in it. The node with the 11 in it is the last node in the list so its nextPtr’s value is set to NULL so we know that’s the end of the list. 

When we initially declare a singly-linked list, it will be empty. What does an empty singly-linked list look like? It just has a headPtr whose value is NULL. It doesn’t point to a node because there are no nodes in the list.



Singly-linked lists aren’t as easy to use or as useful as doubly-linked lists, so let’s jump straight to doubly-linked lists. Below is a doubly linked list where each node not only has a pointer to the next node in the list but, also, a pointer to the previous node in the list. A doubly linked list also has a tail pointer, a pointer to the last node in the list. The prevPtr in the headPtr’s node is set to NULL as is the nextPtr of the tailPtr’s node. An empty doubly-linked list has a headPtr and tailPtr whose values are NULL.



Below is a type definition for a node that has data of an integer, a nextPtr, and a prevPtr. When initializing a node, both of these pointers must be set to NULL because, a node is not in a list until we put it there. Notice the additional node\_t after the struct keyword. This is so we can declare a pointer to a node\_t inside the node\_t structure itself.

typedef struct node\_t {

int i;

struct node\_t\* nextPtr;

struct node\_t\* prevPtr;

} node\_t;

Operations we need for a node must include initializing a node. Another operation could be printing a node or comparing the data in one node to the data in another.

* node\_t\* initNode( int data ): dynamically allocates a node and initializes the data, the nextPtr, and the prevPtr of a this new node that isn’t yet in a list. The algorithm:

1. nodePtr = Allocate memory for the new node
2. nodePtr’s data = data from parameter
3. nodePtr’s nextPtr = NULL
4. nodePtr’s prevPtr = NULL
5. Return nodePtr

How would we call the function initNode from a function like main? You have to assign a node\_t pointer to the function to get the return value, the pointer to the newly created node.

node\_t\* newNodePtr;

nodePtr = initNode( someIntegerGoesHere );

Below is the type definition for a double-linked list. It is a structure that has a headPtr and a tailPtr. It does not have an array or any other pointers. It has a count of the number of nodes in the list because that could be handy.

typedef struct {

node\_t\* headPtr;

node\_t\* tailPtr;

int count;

} dbl\_linked\_list\_t;

Since a doubly-linked list can be traversed from the head to the tail or the tail to the head, it can be used to implement a stack or a queue. Imagine the list as a stack: push an item at the head of the list and pop an item from the head of the list. To use the list as a queue, enqueue at the head of the list and dequeue from the tail of the list. Some operations we need for the list:

* void createList( dbl\_linked\_list\_t\* listPtr ): creates an empty doubly linked list by setting count to 0 and the head and tail pointers to NULL.
* void insertNode( dbl\_linked\_list\_t\* listPtr, node\_t\* nPtr ): inserts node n at the head of the list. This function could be used to push onto the list as a stack or enqueue into the list as a queue. The pre-condition is that the node pointed to by nPtr has already been allocated, has data in it, and has NULL next and previous pointers. Basically, this means that initNode was called to create the node before calling insertNode to insert it.
* node\_t\* popNode( dbl\_linked\_list\_t\* listPtr ): pops the last node inserted from the head of the list and returns a pointer to it. It removes the node from the list but does not deallocate the node’s memory.
* node\_t\* dequeueNode( dbl\_linked\_list\_t\* listPtr ): dequeues the first node inserted from the end of the list and returns it. It removes the node from the list but does not deallocate the node’s memory.
* void deleteList( dbl\_linked\_list\_t\* listPtr ): deletes entire list by deallocating all of the memory for all of the nodes. It sets the head and tail pointers to NULL to indicate an empty list.
* void traverseStack( const dbl\_linked\_list\_t\* listPtr ): traverses and prints the data in the list from head to tail
* void traverseQueue( const dbl\_linked\_list\_t\* listPtr ): traverses and prints the data in the list from tail to head

Let’s look at algorithms and some code for some of these functions. Initializing this doubly linked list involves setting both the head and tail pointers to NULL and count to 0.

createList algorithm:

1. if listPtr != NULL
   1. listPtr’s headPtr = NULL
   2. listPtr’s tailPtr = NULL
   3. listPtr’s count = 0

Checking to make sure that the parameter listPtr isn’t NULL is just a basic check against passing a non-existent list. If listPtr were NULL, that would mean there wasn’t a list to begin with. That no variable of type dbl\_linked\_list\_t existed. This would only be an issue if the list structure itself was dynamically allocated. Nodes have to be dynamically allocated, but the list structure doesn’t necessarily have to be.

Since the parameter to createList, listPtr, is a pointer to a structure, use the -> operator to access its data members such as

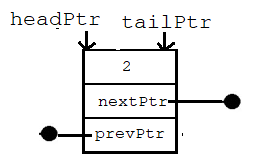
listPtr->count = 0;

To call this function, we need a list then the function call passing the list by reference as shown below. I didn’t dynamically allocate the list so the list variable is not pointer. In general, if you don’t have to dynamically allocate something, then DON’T.

dbl\_linked\_list list;

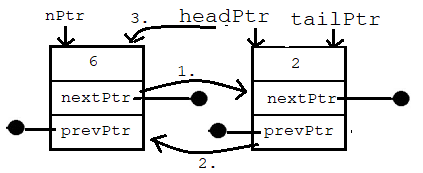
createList( &list );

Let’s look at the insertNode algorithm. Examine the image below. If the list is empty, then adding a node to the beginning of the list means setting the head and the tail pointers to point to that single, new node in the list.

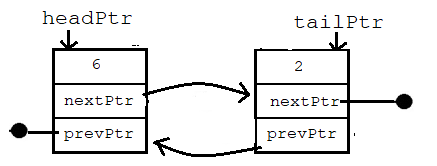


When adding a node to a list that has one or more nodes in it, then the tail pointer is not altered. Examine the image below where the following steps are labelled To insert the new node pointed to by nPtr, we must

1. Set nPtr’ nextPtr to the headPtr
2. Set the headPtr’s next pointer to nPtr
3. Set the headPtr to nPtr



So, the cleaned up image has the headPtr pointing to the new node, the new node’s next pointer pointing to the node that was at the beginning of the list, and that node’s previous pointer pointing to the new node.



A detailed algorithm for insertNode is shown below. It has two parameters, listPtr which points to the list strusture and nPtr which points to the new node to insert in the list. Recall that both listPtr and nPtr are pointers to structures so the syntax of a step such as

list’s headPtr->prevPtr = nPtr

is

listPtr->headPtr->nextPtr = nPtr;

And the syntax for a statement such as

nPtr’s nextPtr = the list’s headPtr

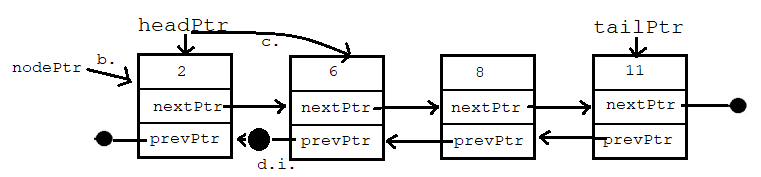
is

nPtr->nextPtr = listPtr->headPtr;

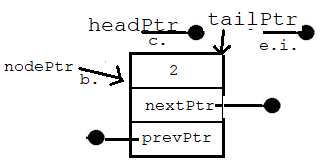
1. if listPtr != NULL
   1. count++
   2. if the lists’s headPtr is NULL (it’s an empty list)
      1. list’s headPtr  = nPtr
      2. list’s tailPtr = nPtr
   3. else (the list already has one or more nodes in it)
      1. nPtr’s nextPtr = the list’s headPtr
      2. list’s headPtr->prevPtr = nPtr
      3. list’s headPtr = nPtr

The popNode algorithm is below. The function has one parameter, a pointer to a list. It returns a pointer to the node removed from the head of the list or NULL if the list was empty so no node could be removed.

1. if the list exists and isn’t empty
   1. count--
   2. nodePtr = the node at the head of the list (so we can return it)
   3. set the headPtr to point to next in the list
   4. if the list is still not empty
      1. set headPtr’s previous pointer to NULL



* 1. if the list is empty (headPtr was just set to NULL)
     1. set tailPtr to NULL
  2. set nodePtr’s prev and next pointers to NULL since the node’s not in the list anymore
  3. return nodePtr



1. else
   1. return NULL

The traverseStack algorithm has one parameter, a pointer to the list. It starts at the head of the list and follows the next pointers until next pointer is NULL which means we’re at the end of the list.

1. if list exists
   1. curPtr = list’s headPtr
   2. while curPtr != NULL
      1. print curPtr’s data
      2. update curPtr so it points to the next node: curPtr = curPtr->nextPtr

The dequeueNode function algorithm is synonymous to the popNode algorithm except you’re taking a node from the tailPtr instead of the headPtr side of the list. The same goes for traversing from the tail to the head of the list in the traverseQueue function.

The last function to look at is the function that deletes the entire list. The algorithm for this function is below. The key to this algorithm is to realize that you can’t delete a node then use that node’s next pointer to get to the next node. For example, you \*can’t\*:

free( headPtr );

headPtr = headPtr->nextPtr;

How can you us the nextPtr variable in a non-existent node? So, we have to use a second pointer, curPtr in the algorithm below, to free a node’s memory, and have it “follow” the head pointer as the head pointer moves down the list.

1. if the list exists and it’s not empty
   1. curPtr = the list’s headPtr
   2. while curPtr isn’t NULL
      1. headPtr = curPtr’s nextPtr
      2. deallocate curPtr memory
      3. update curPtr: curPtr = headPtr
   3. Set list’s tail pointer to NULL
   4. Set list’s count to 0

**Pair Programming 5b:** Copy ~caarnold/cisp1020/week5/mainDblLinkedList.c and read through the file then use it to compile and test the files below. If you decide to test functions as you write them, you will need to comment out some code in main that uses those functions you haven’t written yet, or you can just write stubs for some functions. Create C and header files for the doubly linked list and node. The files to create are:

* node.h: contains the type definition for a node and any function declarations/prototypes for a node. Don’t forget to use the #ifndef NODE\_H #define NODE\_H #endif pre-processor directives.
* node.c: contains all node function definitions that were declared in node.h. This file must #include “node.h” since it has code that references the node\_t structure which is in node.h.
* dbl\_linked\_list.h: this file contains the structure definition for the list. It also contains function declarations/prototypes for all list functions
* dbl\_linked\_list.c: this file contains the definitions for all of the linked list functions in dbl\_linked\_list.h. This file must #include “node.h” since it has code that references the node\_t structure and #include “dbl\_linked\_list.h” since it references the dbl\_linked\_list\_t structure.

It is very important that the correct definitions and declarations are put in the correct files. All functions that operate only on nodes such as initNode \*must\* go in the node files. All functions that operate on the entire list such as insertNode which inserts in a list or findNode which finds a node in a list \*must\* go in the list files. If a function has code that operates on a list at all, it \*must\* go in the list files, not the node files.

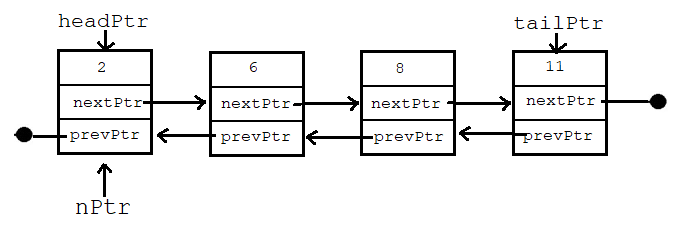
There is more functionality we might want to add to our list which might be needed for a graded lab assignment. For example:

* node\_t\* findNode( dbl\_linke\_list\* l, int key ): returns a pointer to the node with the parameter key in it or NULL if not found
* void removeNode( dbl\_linked\_list\* l, node\_t\* nPtr ): removes node pointed to by nPtr from the list and deallocates memory for it.

The algorithm for findNode is just a linear search algorithm. A linked list cannot be randomly accessed like an array; there is not list[i]. In order to find something in a list, it must be traversed starting at the beginning (or end) and examining each node’s data until the key is found or the end of the list is reached. In this case, the key is not in the list and NULL is returned (as opposed to -1 in the case of a linear search in an array).

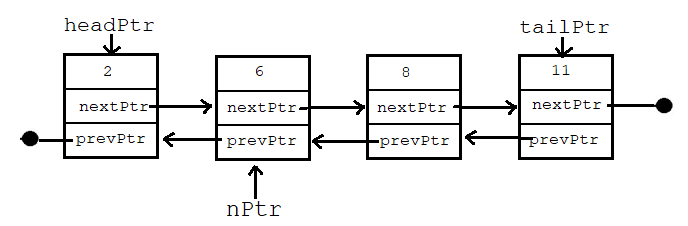
1. for curPtr = list’s headPtr, curPtr != NULL, curPtr = curPtr->nextPtr
   1. if curPtr’s data == key
      1. Return curPtr
2. Return NULL

The algorithm for removing a node given a pointer to the node has some special cases: 1) when the node to be removed is the head, 2) when the node to be removed is the tail, 3) when the node to be removed is the last node, and 4) when the node to be removed is in the middle somewhere. Let’s look at removing the head node first. We’ll also cover the case of removing the last node in the list since it is the head and the tail. Removing the tail node is similar. In the image below, nPtr, the node to remove, points to the same node as the list’s headPtr.



1. if nPtr == list’s headPtr (then nPtr points to the head of the list)
   1. headPtr = nPtr’s nextPtr (so headPtr points to the node with the 6 in it)
   2. if headPtr != NULL (didn’t remove last node in the list
      1. headPtr’s prevPtr = NULL (since the head of the list doesn’t have a node before it
   3. else (headPtr is NULL so removed the last node in the list which is now empty)
      1. tailPtr = NULL

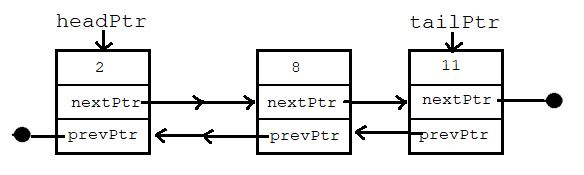
Let’s look at the middle case now. Consider the image below, a list of 4 nodes with keys in order from head to tail of 2, 6, 8, and 11, and nPtr pointing to the second node with key 6.



To remove the node with key 6 pointed to by nPtr, the following algorithm must be followed:

1. nPtr’s prevPtr’s nextPtr = nPtr’s nextPtr (the node with 2’s nextPtr now points to the node with the 8 in it)
2. nPtr’s nextPtr’s prevPtr = nPtr’s prevPtr (the node with the 8’s prevPtr now points to the node with the 2 in it)

The result of removing the node pointed to by nPtr is shown here where the node with a 2 in it now points to the node with the 8 and the node with the 8 points back to the node with the 2.



Putting the algorithms to remove a node all together:

1. If nPtr == list’s headPtr
   1. Steps described above go here
2. else if nPtr == list’s tailPtr (but not the headPtr or the if above would have executed)
   1. list’s tailPtr = nPtr’s prevPtr
   2. list’s tailPtr’s nextPtr = NULL
3. else (nPtr must be in the middle somewhere)
   1. Steps described above go here
4. count--
5. Deallocate nPtr

Algorithm efficiency. Remember, Big-O notation is the worse-case scenario.

* Inserting at the beginning of a singly or doubly-linked list: O(1)
* Inserting in order in a singly or doubly-linked list: O(n) because we may have to traverse the entire list until we find the right spot for the new node.
* Inserting at the end of a singly-linked list: O(n)
* Inserting at the end of a doubly-linked list: O(1)
* Finding in a list: O(n) since finding is just a linear search, and all items may need to be examined.
* Removing a node given a pointer to it: O(1) since there are a constant number of steps no matter how many items are in the list