Introduction to C++

Linear Linked Lists

Topic #4
CS162 - Topic #4

• Lecture: Dynamic Data Structures
  – Review of pointers and the new operator
  – Introduction to Linked Lists
  – Begin walking thru examples of linked lists
  – Insert (beginning, end)
  – Removing (beginning, end)
  – Remove All
  – Insert in Sorted Order
CS162 - Pointers

• What advantage do pointers give us?
• How can we use pointers and new to allocating memory dynamically
• Why allocating memory dynamically vs. statically?
• Why is it necessary to deallocate this memory when we are done with the memory?
CS162 - Pointers and Arrays

• Are there any disadvantages to a dynamically allocated array?
  – The benefit - of course - is that we get to wait until run time to determine how large our array is.
  – The drawback - however - is that the array is still fixed size.... it is just that we can wait until run time to fix that size.
  – And, at some point prior to using the array we must determine how large it should be.
CS162 - Linked Lists

- Our solution to this problem is to use **linear linked lists** instead of arrays to maintain a "list"
- With a linear linked list, we can grow and shrink the size of the list as new data is added or as data is removed
- The list is ALWAYS sized exactly appropriately for the size of the list
A linear linked list starts out as empty

- An empty list is represented by a null pointer
- We commonly call this the **head** pointer
CS162 - Linked Lists

- As we add the first data item, the list gets one node added to it
  - So, head points to a node instead of being null
  - And, a node contains the data to be stored in the list and a next pointer (to the next node...if there is one)
CS162 - Linked Lists

• To add another data item we must first decide in what order
  – does it get added at the beginning
  – does it get inserted in sorted order
  – does it get added at the end

• This term, we will learn how to add in each of these positions.
CS162 - Linked Lists

- Ultimately, our lists could look like:

```
data  next  data  next  data  next  
```

Sometimes we also have a tail pointer. This is another pointer to a node -- but keeps track of the end of the list. This is useful if you are commonly adding data to the end.
CS162 - Linked Lists

• So, how do linked lists differ than arrays?
  – An array is direct access; we supply an element number and can go directly to that element (through pointer arithmetic)
  – With a linked list, we must either start at the head or the tail pointer and **sequentially traverse** to the desired position in the list
CS162 - Linked Lists

• In addition, linear linked lists (singly) are connected with just one set of next pointers.
  – This means you can go from the first to the second to the third to the forth (etc) nodes
  – But, once you are at the forth you can’t go back to the second without starting at the beginning again.....
CS162 - Linked Lists

• Besides linear linked lists (singly linked)
  – There are other types of lists
    • Circular linked lists
    • Doubly linked lists
    • Non-linear linked lists (CS163)
CS162 - Linked Lists

• For a linear linked lists (singly linked)
  – We need to define both the head pointer and the node
  – The node can be defined as a struct or a class; for these lectures we will use a struct but on the board we can go through a class definition in addition (if time permits)
CS162 - Linked Lists

• We’ll start with the following:
  ```c
  struct video {
    char * title;
    char category[5];
    int quantity;
  };
  ```

• Then, we define a node structure:
  ```c
  struct node {
    video data;  //or, could be a pointer
    node * next;  //a pointer to the next
  };
  ```
Then, our list class changes to be:

```cpp
class list {
public:
    list();   ~list();    //must have these
    int add (const video &);
    int remove (char title[]);
    int display_all();
private:
    node * head;    //optionally node * tail;
};
```
Now, what should the constructor do?

- initialize the data members
- this means: we want the list to be empty to begin with, so head should be set to NULL

```c++
list::list() {
    head = NULL;
}
```
To show how to traverse a linear linked list, let’s spend some time with the display_all function:

```cpp
int list::display_all() {
    node * current = head;
    while (current != NULL) {
        cout << current->data.title << ‘t’ << current->data.category << endl;
        current = current->next;
    }
    return 1;
}
```
• Let’s examine this step-by-step:
  – Why do we need a “current” pointer?
  – What is “current”?
  – Why couldn’t we have said:
    ```
    while (head != NULL) {
      cout << head->data.title << ‘\t’
        << head->data.category << endl;
      head = head->next;    //NO!!!!!!!
    }
    ```
    We would have lost our list!!!!!!
Next, why do we use the NULL stopping condition:

```c
while (current != NULL) {
```

This implies that the very last node’s next pointer must have a NULL value

- so that we know when to stop when traversing
- NULL is a #define constant for zero
- So, we could have said:

```c
while (current) {
```
Now let’s examine how we access the data’s values:

```cpp
cout << current->data.title << \t
     << current->data.category << endl;
```

Since current is a pointer, we use the `->` operator (indirect member access operator) to access the “data” and the “next” members of the node structure.

But, since “data” is an object (and not a pointer), we use the `. ` operator to access the title, category, etc.
If our node structure had defined data to be a pointer:

```c
struct node {
    video * ptr_data;
    node * next;
};
```

Then, we would have accessed the members via:

```c
cout << current->ptr_data->title << "\t"
    << current->ptr_data->category << endl;
```

(And, when we insert nodes we would have to remember to allocate memory for a video object in addition to a node object...)
CS162 - Traversing

- So, if current is initialized to the head of the list, and we display that first node
  - to display the second node we must traverse
  - this is done by:
    ```
    current = current->next;
    ```
- why couldn’t we say:
  ```
  current = head->next; //NO!!!!!
  ```
CS162 - Building

• Well, this is fine for traversal

• But, you should be wondering at this point, how do I create (build) a linked list?

• So, let’s write the algorithm to add a node to the **beginning** of a linked list
CS162 - Insert at Beginning

- We go from:
  - head
    - data
    - next

- To:
  - head
  - data
  - next
  - new node
  - previous first node

- So, can we say:
  - head = new node;
  //why not???
• If we did, we would lose the rest of the list!
• So, we need a temporary pointer to hold onto the previous head of the list

```c
node * current = head;
head = new node;
head->data = new video; //if data is a pointer
head->data->title = new char [strlen(newtitle)+1];
strcpy(head->data->title, newtitle);
//etc.
head->next = current;  //reattach the list!!!
```
• Add a node at the end of a linked list.
  – What is wrong with the following. Correct it in class:

    node * current = head;
    while (current != NULL) {
      current = current->next;
    }
    current = new node;
    current->data = new video;
    current->data = data_to_be_stored;

LOOK AT THE BOLD/ITALICS FOR HINTS OF WHAT IS WRONG!
We need a temporary pointer because if we use the head pointer
  • we will lose the original head of the list and therefore all of our data

If our loop’s stopping condition is if current is not null -- then what we are saying is loop until current IS null
  • well, if current is null, then dereferencing current will give us a segmentation fault
  • and, we will NOT be pointing to the last node!
• Instead, think about the “before” and “after” **pointer diagrams:**

Before

```
head → 1ST → ... → NTH
```

After

```
head → 1ST → ... → current → new node
```
CS162 - Inserting at End

- So, we want to loop until current->next is not NULL!
- But, to do that, we must make sure current isn’t NULL
  - This is because if the list is empty, current will be null and we’ll get a fault (or should) by dereferencing the pointer

```c
if (current)
    while (current->next != NULL) {
        current = current->next;
    }
```
Next, we need to connect up the nodes

- having the last node point to this new node
  ```c
  current->next = new node;
  ```
- then, traverse to this new node:
  ```c
  current = current->next;
  current->data = new video;
  ```
- and, set the next pointer of this new last node to null:
  ```c
  current->next = NULL;
  ```
• Lastly, in our first example for today, it was inappropriate to just copy over the pointers to our data
  - we allocated memory for a video and then immediately lost that memory with the following:
    
    ```c++
    current->data = new video;
    current->data = data_to_be_stored;
    ```
  - the correct approach is to allocate the memory for the data members of the video and physically copy each and every one
Now let’s look at the code to remove at node at the beginning of a linear linked list. Remember when doing this, we need to deallocate all dynamically allocated memory associated with the node. Will we need a temporary pointer? – Why or why not...
CS162 - Removing at Beg.

- What is wrong with the following?

  node * current = head->next;
  delete head;
  head = current;

  – everything? (just about!)
First, don’t dereference the head pointer before making sure head is not NULL

if (head) {
    node * current = head->next;
    – If head is NULL, then there is nothing to remove!

Next, we must deallocate all dynamic memory:

delete [] head->data->title;
delete head->data;
delete head;
head = current; //this was correct....
CS162 - Removing at End

• Now take what you’ve learned and write the code to remove a node from the end of a linear linked list

• What is wrong with: (lots!)

```c
node * current = head;
while (current != NULL) {
    current = current->next;
}
delete [] current->data->title;
delete current->data;
delete current;
```
CS162 - Removing at End

• Look at the stopping condition
  – if current is null when the loop ends, how can we dereference current? It isn’t pointing to anything
  – therefore, we’ve gone too far again
    
    ```
    node * current = head;
    if (!head) return 0; //failure mode
    while (current->next != NULL) {
        current = current->next;
    }
    ```

  – is there anything else wrong? (yes)
CS162 - Removing at End

• So, the deleting is fine....

\[
delete [] current->data->title; \\
delete current->data; \\
delete current; \\
\]

– but, doesn’t the previous node to this still point to this deallocated node?
– when we retraverse the list -- we will still come to this node and access the memory (as if it was still attached).
CS162 - Removing at End

• When removing the last node, we need to reset the new last node’s next pointer to NULL
  – but, to do that, we must keep a pointer to the previous node
  – because we do not want to “retraverse” the list to find the previous node
  – therefore, we will use an additional pointer
    • (we will call it “previous”)

CS162 Topic #4
CS162 - Removing at End

• Taking this into account:

```c
node * current= head;
node * previous = NULL;
if (!head) return 0;
while (current->next) {
    previous = current;
    current = current->next;
}
delete [] current->data->title;
delete current->data;
delete current;
previous->next = NULL;  //oops...
```

Can anyone see the remaining problem?
CS162 - Removing at End

• Always think about what special cases need to be taken into account.

• What if...
  – there is only ONE item in the list?
  – previous->next won’t be accessing the deallocated node (previous will be NULL)
  – we would need to reset head to NULL, after deallocating the one and only node
• Taking this into account:
  
  ```
  ... 
  if (!previous) //only 1 node
      head = NULL;
  else
      previous->next = NULL;
  }
  ```

Now, put this all together as an exercise
The purpose of the destructor is to
- perform any operations necessary to clean up memory and resources used for an object’s whose lifetime is over
- this means that when a linked list is managed by the class that the destructor should deallocate all nodes in the linear linked list
- delete head won’t do it!!!
CS162 - Deallocationg all

• So, what is wrong with the following:

```cpp
list::~list() {
    while (head) {
        delete head;
        head = head->next;
    }
}
```

– We want head to be NULL at the end, so that is **not** one of the problems
– We are accessing memory that has been deallocated. Poor programming!
CS162 - Deallocationg all

• The answer requires the use of a temporary pointer, to save the pointer to the next node to be deleted prior to deleting the current node:

```cpp
list::~list() {
    node * current;
    while (head) {
        current = head->next;
        delete [] head->data->title;
        delete head->data;
        delete head;
        head = current;
    }
}
```
Next, let’s insert nodes in sorted order.

Like deleting the last node in a LLL,
– we need to keep a pointer to the previous node in addition to the current node
– this is necessary to re-attach the nodes to include the inserted node.

So, what special cases need to be considered to insert in sorted order?
CS162 - Insert in Order

- Special Cases:
  - Empty List
    - inserting as the head of the list
  - Insert at head
    - data being inserted is less than the previous head of the list
  - Insert elsewhere
CS162 - Insert in Order

• Empty List
  – if head is null, then we are adding the first node to the list:

```java
if (!head) {
    head = new node;
    head->data = ...
    head->next = 0;
}
```

Before

head

After

head

1ST
• Inserting at the Head of the List
  – if head is not null but the data being inserted is less than the first node

Before

After

CS162 Topic #4
CS162 - Insert in Order

• Here is the “insert elsewhere” case:

Before

head

1ST

NTH

After

head

1ST

current

previous

new node

NTH

CS162 Topic #4
CS162 - Special Cases

• When inserting in the middle
  – can the same algorithm and code be used to add at the end (if the data being added is “larger” than all data existing in the sorted list)?
  – Yes? No?
CS162 - In Class, Work thru:

- Any questions on how to:
  - insert (at beginning, middle, end)
  - remove (at beginning middle, end)
  - remove all

- Next, let’s examine how similar/different this is for
  - circular linked lists
  - doubly linked lists
CS162 - In Class, Work thru:

• For circular linked lists:
  – insert the very first node into a Circular L L (i.e., into an empty list)
  – insert a node at the end of a Circular L L
  – remove the first node from a Circular L L

  – Walk through the answers in class or as an assignment (depending on time available)
CS162 - In Class, Work thru:

• For doubly linked lists:
  – write the node definition for a double linked list
  – insert the very first node into a Doubly L L (i.e., into an empty list)
  – insert a node at the end of a Doubly L L
  – remove the first node from a Doubly L L

  – Walk through the answers in class or as an assignment (depending on time available)
CS162 - What if...

- What if our node structure was a class
  - and that class had a destructor
  - how would this change (or could change) the list class’ (or stack or queue class’) destructor?

//Discuss the pros/cons of the following design....

```cpp
class node {
public:
  node(); node(const video &); ~node();
private:
  video * data; node * next;
};
```
• OK, so what if the node’s destructor was:

```cpp
node::~node() {
    delete [] data->title;
    delete data;
    delete next;
};
list::~list() {
    delete head; //yep, this is not a typo
}
```

– This is a “recursive” function.... (a preview of our Recursion Lecture; saying delete next causes the destructor to be implicitly invoked. This process ends when the next ptr of the last node is null.)