COMP 1633: Intro to CS II

Linked lists

Charlotte Curtis

March 11, 2024

Where we left off

- Dynamic memory allocation
- Assignment 2
- Midterm

> *Textbook Section 9.2*

```
Time *t = new Time;
t->hour = 5;
do_something_with_time(t);
delete t;
```

Today's topics

- Intro to linked list structures
- Nodes and node pointers
- Building linked lists
- Traversing linked lists
- Linked Lists vs Arrays

> *Textbook Sections 13.1*

Another list structure? Why?

- Arrays, even dynamically allocated, are a fixed size and order

```
int *arr = new int[10];
```

- Growing or shrinking an array is painful

```
int *new_arr = new int[20]; // allocate a new bigger array
for (int i = 0; i < 10; i++) { // copy the old array into the new one
    new_arr[i] = arr[i];
}

delete [] arr; // free the memory at the old array
arr = new_arr; // point the original memory to the new array
new_arr = NULL; // good practice to reset the temporary pointer
```

- This is expensive, particularly with an array of structs

Linked lists overview

- Instead of allocating a whole array, why not string together a bunch of pointers?
- A **linked list** is a data structure where each element contains a value (or multiple values), as well as a **pointer** to the next item in the list
- The memory addresses of each item are **random**, and that's okay!
- To accomplish this, we need a `struct` that contains a value and a pointer to the next item in the list

```
struct Node {
    <data type> data;
    Node *next;
};
```

That `Node` structure looks funky

- `Node` is a `struct` with a field that's a pointer to another `Node`
- This is a **self-referential** structure. The name `Node` is common for linked lists, but we could have named it anything, for example:

```
struct Person {
    string name;
    Person *spouse;
};
```

- Why is `spouse` a pointer instead of just a `Person`?

> *Yes, that's a `std::string` - you can use them in assignment 3!*

Pointers so far

So far we've learned how to:

- Allocate memory on the **stack** by declaring variables
- Allocate memory on the **heap** by using `new`
- Define pointers to **named memory addresses** (on the stack)
- Define pointers to **unnamed memory addresses** (on the heap)

> *Problem: we still don't get "unlimited" memory because we need to associate a named variable with each memory address*

Linked Lists

- We only need to declare one variable to get started

```
Node *head = NULL;
```

- We then use the self-referential pointer to **link** to the next chunk of data

```
head = new Node;
head->data = 5;
head->next = NULL;
```

- We can add or remove items from the list dynamically, and only need to keep track of the `head` pointer

> *Concept demo time*

Starting a linked list

- For this simple `Node` struct (which only holds an `int`):

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

- We start with the head pointer:

```
Node *head = NULL;
```

- This is an **empty list**
- All we've done so far is allocate space for a **pointer** to a `Node`
- Don't lose track of the `head` pointer!

Building the list

- Unlike arrays, list nodes need to be allocated **one at a time**

```
head = new Node; // allocate a new node on the heap
head->data = 7; // assign its value
head->next = NULL; // point to the next one in the list
```

- The **last** node of the list should always point to `NULL`
- This is now a **singleton list**, where the start and end are the same item
- We can add another item to the end by allocating the `next` pointer:

```
head->next = new Node;
head->next->data = 10;
head->next->next = NULL;
```

Adding to the front

- We can add a third node to the front of the list, but be careful!

```
head = new Node;
head->data = 1;
head->next = ???; // uh oh, we lost the old head!
```

- We should instead declare a temporary pointer for the new node:

```
Node *temp = new Node;
temp->data = 1;
temp->next = head;
head = temp; // reassign head to the address of temp
```

- Question: should we `delete` `temp`?

Adding to the middle (inserting a node)

- At this point we have a list with 3 nodes: `1 -> 7 -> 10`
- Let's add a `5` between `1` and `7`
- Once more, we'll need a temporary pointer for the new node:

```
Node *temp = new Node;
temp->data = 5;
temp->next = head->next; // steal the pointer to 7
head->next = temp; // point 1 to 5
```

> *Adding to the middle is more expensive as it means **traversing** the list*

Linked list check-in 1/2

We can't just dereference `head + 1` to get the next item in the list because...

A. The `head` pointer is a `NULL` pointer
B. The linked list is not contiguous in memory
C. The `head` pointer does not point to the first item in the list
D. The memory allocated to `head` is not the size of a `Node`

Linked list check-in 2/2

`next` needs to be a `Node *` and not a `Node` because...

A. A `struct` cannot be a member of another `struct`
B. All `struct` members must be pointers
C. The memory would be allocated on the stack
D. The memory allocation would be recursive

Basic list traversal

- **Pointer arithmetic** only works because arrays are **contiguous** in memory
- Since a list is only defined by its `head` pointer, we need to **traverse** the list to find any other item - we can't just say `head + 1` or `head[1]`
- The basic algorithm is something like:

```
set travelling pointer to head
while travelling pointer points to something
    do something with the current node
    advance the travelling pointer to the next node
```

- "Do something" can be printing, searching, summing, etc

Traversing a list

In C++ syntax:

```
Node *current = head;
while (current) { // or while (current != NULL)
    // do something with current->data
    current = current->next;
}
```

- This is a **sentinel loop** that stops when `current` is `NULL`
  - As always, the LCV update must be the **last line** of the loop body
  - Very important that the last node point to `NULL`!
- `current` does not allocate new memory (other than for the pointer itself)

Inserting a node

Basic approach:

- Find the proper position for insertion (We'll deal with this later)
- Allocate a new node
- Steal the `next` pointer from the previous node
- Assign the previous node's `next` pointer to the new node

> *Suppose the list already has the values `1 -> 5 -> 7 -> 10`. We want to add the value `6` and keep the list sorted.*

Handling special cases

We want to insert a node at the "proper" position, which may be:

- At the beginning of the list
- In the middle of the list
- At the end of the list
- In an empty list

> *Which of these need special handling?*
>
> *Next lecture we'll look at some common approaches*

Arrays vs Linked Lists

Arrays	Linked Lists
Contains only data	Contains data and pointers (more memory)
Fixed number of elements	Variable number of nodes
Minimum size is 1	Minimum size is 0
Supports random access	Must traverse to find an element
Insertion/deletion requires shifting	Insertion/deletion is easy
Easiest insertion/deletion at the end	Easiest insertion/deletion at the front
Need to know the length and fill level	End is marked by `NULL`

Coming up next

- Lab tomorrow: continue dynamic allocation lab
- Lecture: More linked lists
- Assignment 3: Linked lists

> *Textbook Chapter 13*