MTH 4300, Lecture 19

Linked Lists;
Removing and Adding Elements

E Fink

April 21, 2025

1. Linked Lists

Consider the following scenario: you have a list of values that you wish to store in a specified order. However, you frequently have to insert or erase elements – sometimes in the front, sometimes in the back, maybe even in the middle.

`vectors` are not terribly well-suited to storing this data in these conditions. Inserting and removing elements at the end of the vector is relatively inexpensive – although if you do it enough, you'll have to resize. But inserting near the front of a vector is super-expensive, especially if it happens a lot: inserting data near the front of a vector requires moving each of the elements in the underlying array back by one position, which can take a long time, and removing is similarly slow.

The *linked list* data structure is an alternative structure. While it has its own drawbacks, it makes insertion and removing into very quick operations.

Linked Lists

Let's say we want to store the strings `"Alice"`, `"Bob"`, `"Carol"`, `"Evan"`, in that order. The first step is slightly bizarre: we are not going to put these names into some sort of array. Instead, we are going to make objects for each of the elements.

We will create objects of the following struct:

```
struct Node {
   string data;
   Node *next;
   Node(string s, Node* n): data{s}, next{n} {}
};
```

A *struct* is just a class where everything is public by default. While they are close to identical to classes in functionality, in practice `structs` are usually used to indicate that we view our datatype as a simple aggregation of data, where we don't wish to hide implementation details, or make complex methods.

Linked Lists

Now, we can start creating nodes by

```
Node n1("Alice", nullptr);
Node n2("Bob", nullptr);
Node n3("Carol", nullptr);
Node n4("Evan", nullptr);
```

`nullptr` is a special "pointer to nothing" value (and here it's only temporary).

L19x1_linked.cpp

So far, this is a pretty pathetic excuse for a list – there's no order to it, it's just a collection of nodes.

Linked Lists

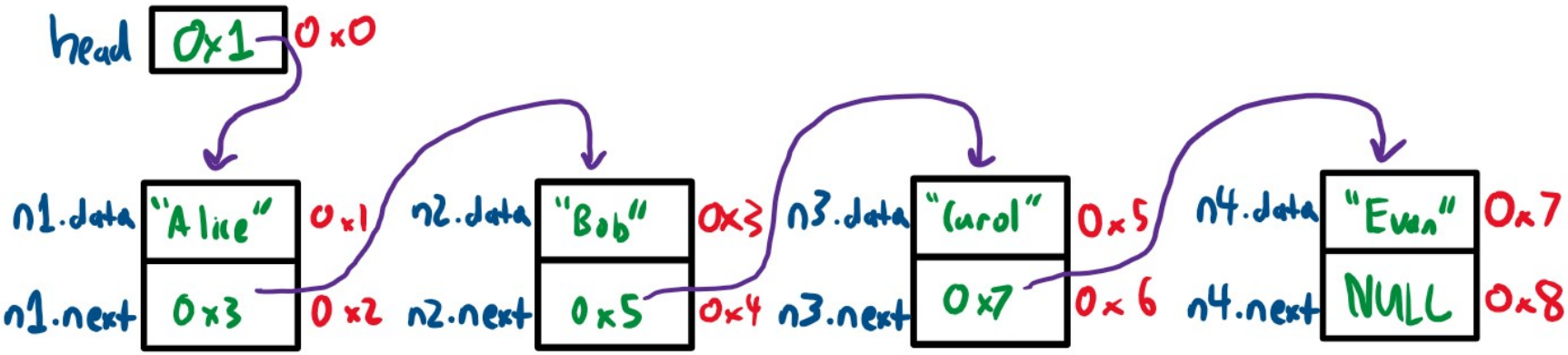
To make it more listy, you can start doing the following:

```
n1.next = &n2;
n2.next = &n3;
n3.next = &n4;
```

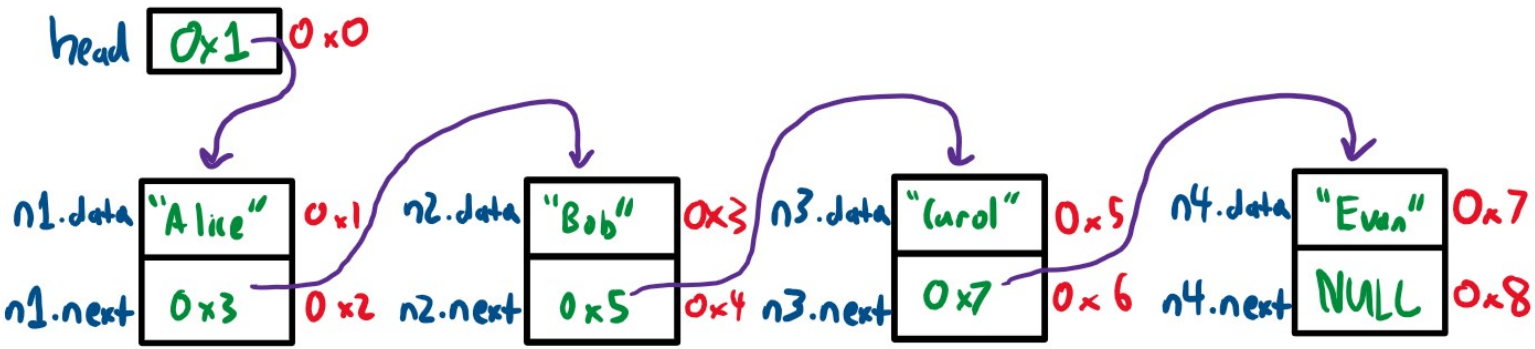
Additionally, I'll introduce a pointer `head` to the first element, by

`Node *head = &n1;`

At this point, a picture is in order. (Recall that the address of an object is the least address of the storage of its members!)



Linked Lists



Note that `head` is a good way to represent the entire list – because one can access all the data in the list just using the `head` variable, without having the names of the individual Nodes. For example, if I wanted to print out `Alice` I could write

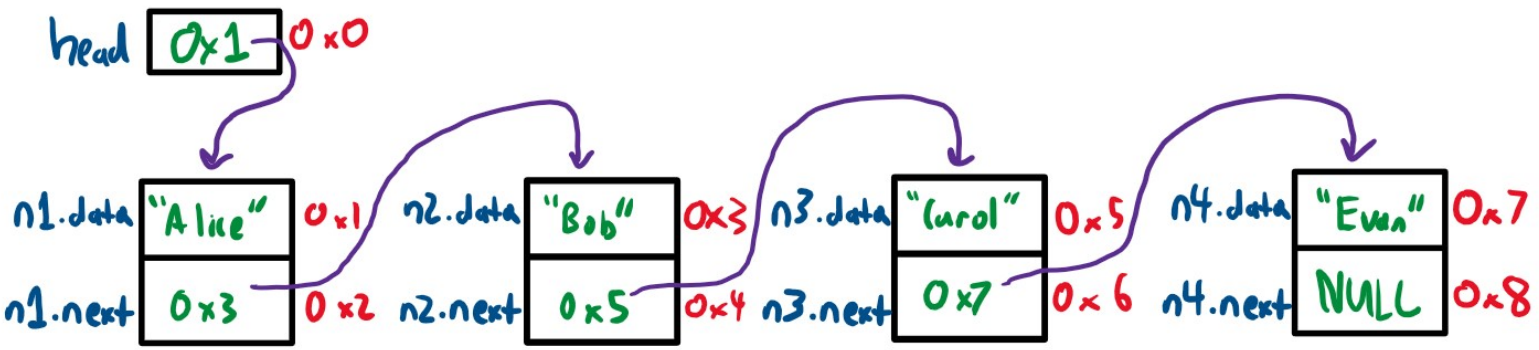
```
cout << (*head).data << endl;
```

And to print out `Bob`, I could print out

```
cout << ( *( (*head).next ) ).data << endl;
```

because `*head` is `n1`,
and `(*head).next` is `0x3`,
and `*( (*head).next )` is therefore `n2`,
and then you can print the `data` of that. Yeesh.

Linked Lists



Recall that the `->` operator means star-then-dot, so that e.g. `current->data` is `(*current).data`. So `current` will be the address of a Node,
`*current` will be the Node at that address,
and `current->data = (*current).data` will be the string in the Node at that address.

That helps with all the parentheses in the above:

```
cout << head->data << " " << head->data->data << endl;
```

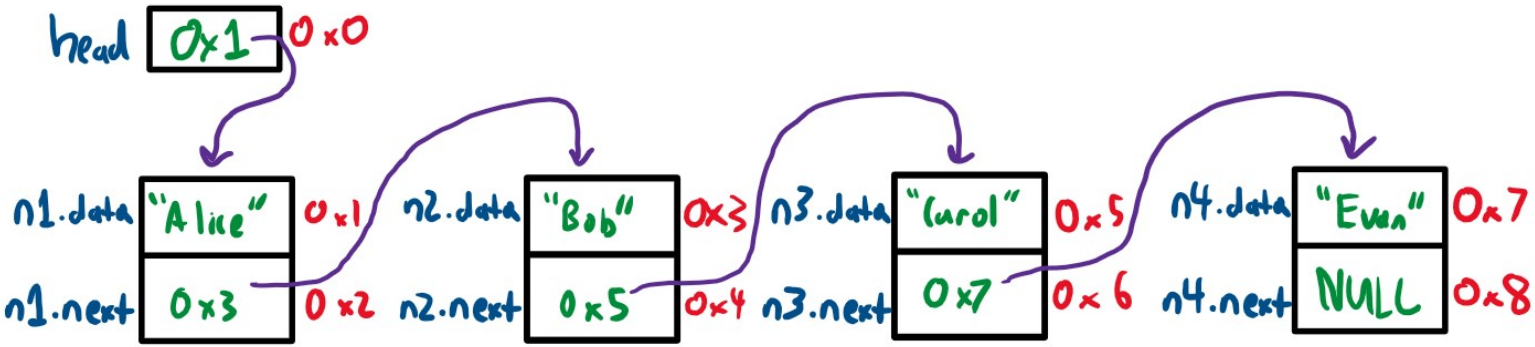
prints out `Alice Bob`.

Linked Lists

The most basic thing we can do with a linked list is to *traverse* it – i.e., go through it item by item.

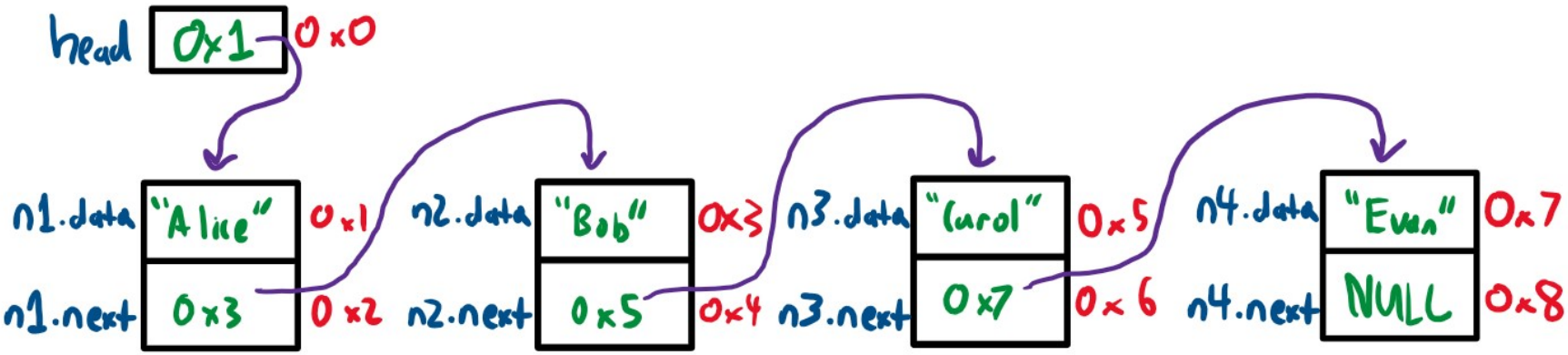
```
Node *current = head;

while(current != nullptr) {
   cout << current->data << endl;
   current = current->next;
}
```

So, with the linked list as depicted below, `current` would be `0x1`, then `0x3`, then `0x5`, then `0x7`, then `nullptr`. At each stop, the data of `*current` is printed.



2. Removing and Adding Elements

Removing an element: Let's remove `Carol` from this list. Since we are lucky enough to know that `Carol` is stored in `n3`, and that the previous entry is `n2`, we can simply edit the `next` pointer of `n2`, to point to `n4` instead.



If you start traversing from the designated head, you will see Alice, Bob, and Evan. The fact that Carol's Node still points to Evan's Node is irrelevant – it is impossible to get to Carol's Node from the head.

Removing and Adding Elements

Perhaps that was too easy. Typically, we only keep track of the `head` pointer of a list, rather than having explicit variable names for each Node. Can we seek *and* remove a Node, given its content data?

It will be helpful to have two pointers, `current` and `prev`. That's because you only know when you've found the correct position for your removed element when you have gone one entry past it. So, we let `current` refer to the element we are looking at, and `prev` refer to the element immediately prior to that.

Additionally, extra care may be required if:

- the element to be removed comes at the front of the list
- the element to be removed comes at the end of the list
- the element to be removed isn't present at all

Let's try to keep our eye on all the possible edge cases.

L19x2_remove.cpp

Removing and Adding Elements

Adding an element: let's take the original list, and add in an element, placing it in its natural place in alphabetical order. This is somewhat similar to before – basically, you just switch some `next` pointers around – but there is the additional challenge of finding the location.

L19x3_add.cpp

It will be helpful to have two pointers, `current` and `prev`. That's because you only know when you've found the correct position for your new element when you have gone one entry past it. So, we let `current` refer to the element we are looking at, and `prev` refer to the element immediately prior to that.

Additionally, extra care is required if the element should be added at the front or the end of the list.

Now that we've pointed out the benefits of linked lists with regard to insertion and deletion, we should mention a drawback: **accessing element #i in the list is slow**. You need a loop with a counter!