COMP 1633: Intro to CS II

More Linked Lists

Charlotte Curtis

March 13, 2024

Where we left off

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

> *Textbook Sections 13.1*

```
// Start the list
Node *head = new Node;
head->data = 1;

// Add another element
head->next = new Node;
head->next->data = 2;
head->next->next = NULL;
```

Today's topics

- Algorithms for working with linked lists
  - Inserting a node
  - Searching for a value
  - Deleting a node
- Passing linked lists to functions
- Linked list variations

> *Textbook Chapter 13*

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)

Passing linked lists to functions

- Many algorithms, like printing a list, would be most useful as a function
- What should be passed to the function? By value or by reference?
  - The whole list is attached to the `head` pointer
  - Does the `head` need to point somewhere else?

```
// by value
void print(Node *head);
```

```
// by reference
void insert(Node *&head, int value);
```

- If `Node *&` is confusing, you can use a `typedef`:

```
typedef Node * NodePtr;
void insert(NodePtr &head, int value);
```

Linked list + function example

Write a function to calculate and return the length of a linked list.

Inputs: Head pointer (by value or by reference?)

Outputs: Length of list (what datatype?)

Caution: passing by reference vs value

Say we defined an insertion function as:

```
void insert(Node *head, int value); // pass head by value
```

When testing with the value `5`, this will work!

- What about `9`?
- What about `12`?
- What about `0`?

> *Pass-by-reference is only needed when `head` changes, but to keep your function general, you should assume that it might change*

Finding the proper position - v1

- We need to find the node that comes **before** the insertion point
- This means that we need to examine the `next` pointer of each node

```
Node *current = head;
while (current->next && current->next->data < value) {
    current = current->next;
}
```

- Very important to check `current->next` before accessing `current->next->data`!
- What if the list is empty?
- What if the new node needs to be inserted at the head?

Finding the proper position - v2

- Alternatively, we could use two pointers traversing in parallel:

```
Node *prev = NULL;
Node *current = head;
while (current && current->data < value) {
    prev = current;
    current = current->next;
}
```

- Makes handling of special cases easier, but need to keep track of two pointers
- Which is better? Whichever makes more sense to you!

Special cases

`prev`	`current`	Solution
`NULL`	`NULL`	Insert as first node in empty list
`NULL`	`not NULL`	Insert as first node in non-empty list
`not NULL`	`NULL`	Insert as last node in non-empty list
`not NULL`	`not NULL`	Insert in middle of list

> *For every linked list operation, think about the special cases!*

Linked List check-in 1/2

What am I forgetting in the following code? Assume that a list of `Node`s already exists with a pointer to `head` defined.

A. `head = temp;`

B. `delete temp;`

C. `temp = NULL`;

D. Both 1 and 2

E. Both 1 and 3

```
Node *temp = new Node;
temp->data = 0;
temp->next = head;
```

Linked List check-in 2/2

What is the following code doing? Again, assume `head` is defined.

A. Inserting at the start of the list
B. Inserting at the end of the list
C. Inserting in the middle of the list
D. Finding a node
E. Deleting a node

```
Node *temp = new Node;
temp->data = 5;
temp->next = head->next;
head->next = temp;
```

Deleting a node

- We need to:
    - Find the node to delete
    - Copy the address of the `next` node
    - Disconnect the node by changing the `next` pointer of the previous node
    - Free the memory using `delete`
- More traversing!
- More special cases!

Deleting a node - code example

```
Node *prev = NULL;
Node *current = head;
// Find the node to delete
while (current && current->data != value) {
    prev = current;
    current = current->next;
}

prev->next = current->next; // Disconnect the node
delete current; // Free the memory
```

What special cases do we need to consider?

Special cases for deletion

`prev`	`current`	Solution
`NULL`	`NULL`	Empty list, nothing to delete
`NULL`	`not NULL`	Delete the first node, update `head`
`not NULL`	`NULL`	End of list, nothing to delete
`not NULL`	`not NULL`	Delete within the list (could be last)

Linked list variations

- By creating your own data structure, you get to define the rules!
- Maybe you're doing a lot of adding/deleting at the end of the list, so you might want to keep track of a pointer to the **last node** (the `tail`)
- You might want to keep track of the **length** of the list or the **sort order**
- Once you start getting fancy, you probably want to create a `class` to encapsulate all of this information - we'll do this next week
- First though, let's look at **doubly linked lists**

Doubly linked lists

- So far we've only been able to travel one direction - the `next` node has no knowledge of its predecessor
- Solution: add a `prev` pointer to each node

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

- Advantages: easier to delete nodes, easier to traverse backwards
- Disadvantages: more memory, more complexity (two pointers to maintain)

Circularly linked lists

- Instead of the last element pointing to `NULL`, it points to `head`
- There is no real `head` anymore, but you still need to keep a pointer to **somewhere** in the list
- Only really useful when there's no start or end
- Advantages: easier to traverse, no need to check for `NULL`
- Disadvantages: more complexity, usually can't be empty

Lists of lists

- You can have a linked list of linked lists!
- Each node in the "outer" list is the `head` of another list
- Example: list of courses, each course has a list of students

Cross-linked lists

Expanding on the previous example, consider the situation where:

- Each student has a list of courses
- Each class has a list of students
- Each instructor has a list of courses

```
struct Student {
    string name;
    int id;
    Course *courses;
};
```

```
struct Course {
    string name;
    int number;
    Student *students;
    Instructor *instructor;
};
```

```
struct Instructor {
    string name;
    Course *courses;
};
```

> *COMP 2631 is all about various information structure*

Object oriented programming preview

- So far we've been implementing solutions in a **procedural** style
- The **object oriented** approach is based on the idea that different **objects** can be interacted with in a different way
    - You can sit on a chair
    - You can draw with a pen
    - You can (probably) pick up a chair and a pen
    - Can you draw with a chair?
- In the OO approach, we can **encapsulate** data and functions in a `class` - an **abstract data type** that defines how an object can be interacted with

Classes

- A **class** is a blueprint for creating objects, much like how a `struct` is a blueprint for creating data structures

```
struct Student {
    string name;
    int id;
};
```

```
class Student {
    string name;
    int number;

    void print();
};
```

- A **class** is a **type** of object, just like `int` or `string` or `Node`
- **Member functions** are accessed using `.` or `->` just like member variables

In fact, in C++ a `struct` and a `class` are almost identical

Coming up next

- Tutorial: Linked lists
- **Assignment 3** 🎉 Linked lists
- Next week: Classes and objects

> *Textbook Chapter 10.2-10.3*