CS163 at CCUT Week 2: Linear Linked Lists

David Lu
4/25/17

Data Structure

In computer science, a data structure is a particular way of organizing data in a computer so that it can be used efficiently.

This week, we will look at the linear linked list data structure

Topics to cover:

- Lists
  - Linear Linked Lists
    - Compare to arrays
    - Complexity
    - Basic operations:
      - Insert
      - Delete
    - Traversal
    - Recursion

        - Searching
        - Sorting (if we have time)
    - Practice Programming!
    - Practice Exercises

Lists

In computer science, a list or sequence is an abstract data type that represents a **countable** number of **ordered** values, where the same value may occur more than once.

An *instance* of a list is a computer representation of the mathematical concept of a finite sequence.

A list can also be defined inductively:

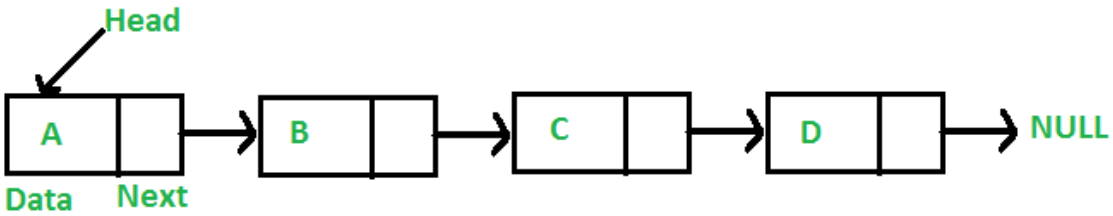
> A list is either empty or it is an item followed by a list.

Linear Linked List

In computer science, a linked list is a linear collection of data elements, called *nodes*, each pointing to the next node by means of a pointer.

It is a data structure consisting of a group of nodes which together represent a sequence.

In a simple linear linked list, each node is composed of a piece of data and a reference (in other words, a link) to the next node in the sequence.



Linked Lists are among the simplest and most common data structures because it allows for efficient insertion or removal of elements from any position in the sequence.

Back to Index

Compare to Arrays

It is important to understand that a linked list is different than an array. Linked lists are not indexed and does not support arbitrary access.

Do you recall what an array is and how arrays work?

Time Complexity

Average for Linear Linked List:

Access	Search	Insert	Delete
$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$

Basic Operations

Abstract Data Type (ADT)

- A constructor for creating an empty list
- A test to see if a list is empty
- A way to insert an item in front of the list
- A way to insert an item at the back of the list
- Search for a particular item
- What else?

Code Basics

To implement a linear linked list in C, we need to understand a few C concepts.

We can use a struct or a class to define each node inductively.

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

Back to Index

Insertion

Let's keep things simple and consider linear linked lists of integers.

A basic operation for lists is to put something on the list. If our list is written on paper, we just find a place to write our item down. If our list is in markdown, we can do this:

1. First item
2. Second item

What is the algorithm for insertion in a linear linked list?

```
def insert(head, data):
  if head is empty:
    head <- data
  else:
    insert(head.next, data)
```

What will this do?

Insertion location

Should we insert at the front, back, or middle of the list?

What are the algorithms to do these?

Deletion

Another basic operation is to remove something from the list. Remember in C++ we need to free the memory that we allocated.

Back to Index

Traversal

You should have seen how to traverse the linear linked list by looping. Perhaps you've seen code that looks like this:

```
node* temp = head;
while(temp != null)
{
  //Do stuff
  temp = temp->next;
}
```

Recursion

Since a list is a recursive data type, it makes sense to make use of recursion in programming operations on lists.

Recursion is a tool a programmer can use to invoke a function call on itself.

Recursive programming is directly related to mathematical induction.
In recursive programming we need two things:

- A **base case**
- A recursive case

Consider the pseudocode to add an item at the end of a linear linked list.

Iteratively

```
def addLast(item):
  if head = None:
    head <- new node(item)
  else:
    node temp <- head
    while temp.next != None:
      temp <- temp.next
    temp.next <- new node(item)
```

Recursively

```
def addLast(head, item):
  if head = None:
    head <- new node(item)
  else
    addLast(head->next, item)
```

Let's try in C++ together.

Back to Index

Searching

An operation we may be interested in doing on lists is to find whether an item is in a list or not. This is called *search*.

Back to Index

Sorting

Another operation we may be interested in performing on lists is to sort it. We called this *sort*.

Back to Index

Creating a random list to practice with in C++m

There are some useful functions in the C++ standard library to generate random numbers.
We will need to include two libraries: ctime and cstdlib
We need ctime to “seed” the peudo random number generator.
And cstdlib contains a pseudo random number generator function.

```
#include <ctime>
#include <cstdlib>

...

// To seed the PRNG:
srand(time(NULL)); // Do this only once!

// To get a random number from 0 to 100:
int num = rand() % 101;
```

So we may have a loop to insert some number of random numbers into our list to practice with.
For example:

```
for(int i = 0; i < 10; ++i)
{
  list.insert(rand() % 101);
  // Or insert(head, rand() % 101); if we do not have a class.
}
```

Back to Index

Some Practice with Linear Linked Lists

Print a list in reverse

> Given a linear linked list (3->2->4->5), print the list in reverse order (5->4->2->3)

Compare two Lists

> Given two linear linked lists, return true if they are the same list. (Order and values matter)

Sorted copy

> Given a linear linked list a, return a sorted list s such taht s contains the same values as a
>
> Example: Given 4->9->2->3->2, return 2->2->3->4->9

Delete duplicate

> Given a sorted linear linked list, delete duplicates from the list.
>
> For example, if given 1->2->2->3->3->3->4, the resulting list should be 1->2->3->4.

Merge two sorted linear linked lists

> Given sorted linear linked lists a and b, return a sorted linear linked list c that contains all elements from a and b
>
> Example: a = 2->3->4->6, b = 1->3->7
> c = 1->2->3->3->4->6->7

Back to Top