DS

Intro To Linked Lists

Bonus Reading

A Book On C: 10.1 to 10.4

Why Data Structures?

Increase efficiency

in

storage

and/or

speed

First:

linked list

What is it?

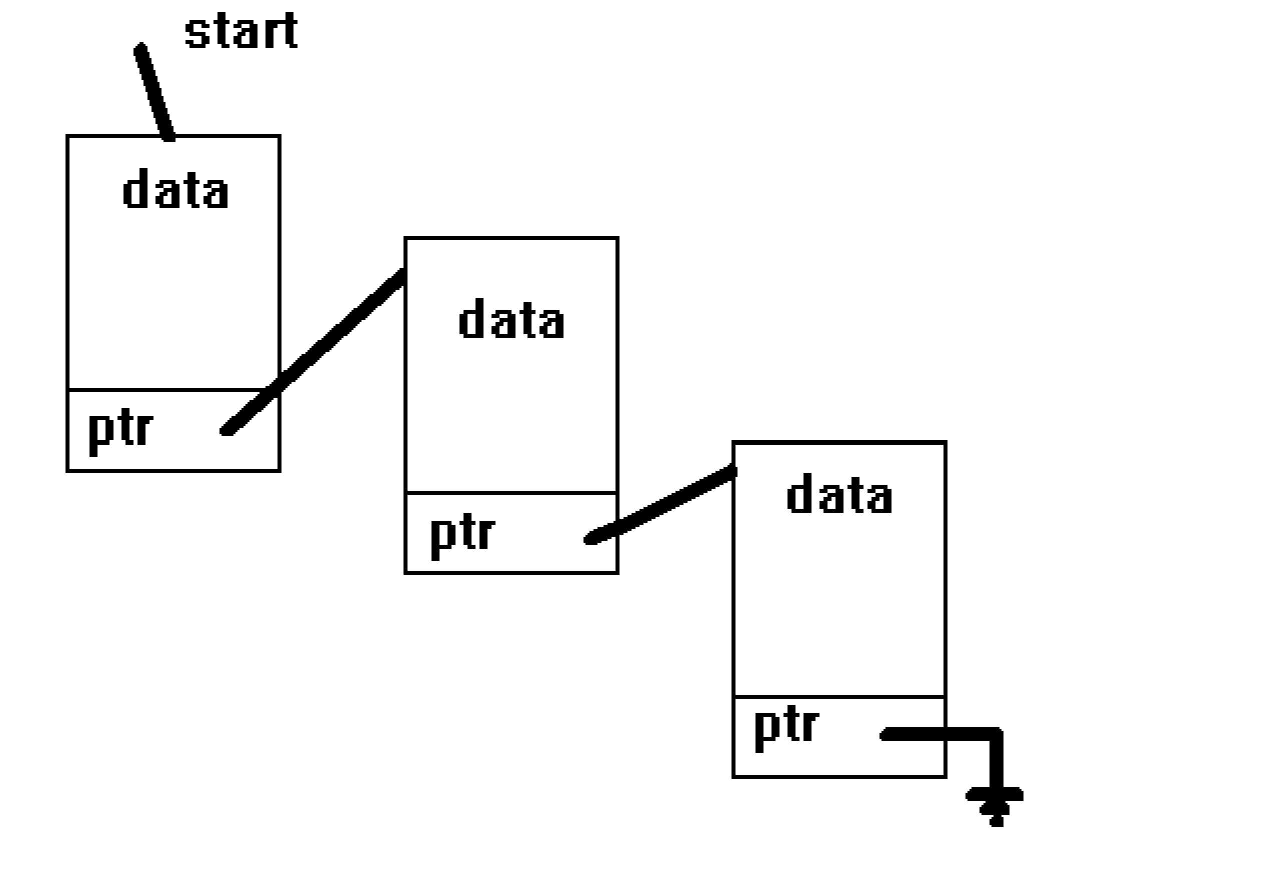
series of data blocks

linked together

through pointers

One block points at the next in a chain until there’s nothing left to point at

Conceptualization



How does it work?

Each block is made up of a data part and a pointer

The data part looks very much like what you're already familiar with (from structs)

The pointer is a pointer to a data block

Example of a block

```
struct addressInfo
{
     char name[51];
     char address[51];
     char phone[11];
     struct addressInfo *next;
};
typedef struct addressInfo
   addrInfo;
```

In this example, the data that we want to store is in name, address, and phone

The pointer variable is called next

The typedef is used solely for convenience in this PowerPoint presentation

Typical Conventions

The pointer field is usually called *next*

The pointer field is usually after the data, not before and never inside

The struct is often referred to as a *node* or a *cell*

Review!

What's *next?*

Review!

Why use a struct and not an array?

Hint: Separating Data from Pointer

It is not uncommon to have the data defined with its own struct

e.g.

```
struct addressInfo
{
        char name[51];
        char address[51];
        char phone[11];
};
struct addressNode
{
        struct addressInfo      info;
        struct addressNode *next;
};
```

Head of the List

The start of the linked list is often called the *head*

You always need to use a variable to keep track of the head

- Its data type is a pointer to the data block
- Its initial value is always NULL

E.g. struct addressInfo *head = NULL;

Review!

Why do we need a head pointer?

Review!

Why does the head pointer need to be initialized to NULL?

Getting A Node

When you need to store something in the linked list, allocate the block (e.g. using *malloc*())

E.g.

- block = (addrInfo *) malloc(sizeof(addrInfo));

Don’t forget to check return value from *malloc*()!

Starting the List

If there is nothing in the list so far, then the node will be the only member

You can determine if there is anything in the list by checking the value of the *head* variable

E.g. if(head == NULL)

If the *head* is NULL, simply assign the address of the newly allocated block to the *head* variable

E.g.

```
if( head == NULL )
{
    head = block;
    block->next = NULL;
}
```

block->next = NULL

The value of the *next* field **must** always be meaningful

It must either point to a valid block in the list or be *NULL*

If the *next* field is *NULL*, that means that the list has ended

It is usual to assign *NULL* to the *next* field when you add the node to the list

Review!

What would happen if you set block->next = block?

Adding to an Existing List

If the *head* is not *NULL*, then the list has entries already

You could add the node to the start of the list, but this would result in the list being backwards

It's conventional to add the new node to the end of the list

List Traversal

The end of the list is found through list traversal

“List traversal” means going through the list from one end until you either find the other end or something that you are looking for

Example of List Traversal

```
addrInfo *ptr = head;
 while( ptr->next != NULL )
 {
     ptr = ptr->next;
 }
 // when leaving the loop, points to the last node
```

Adding the New Node to the End of the List

Assuming that a variable called *ptr* points to the node at the end of the list, simply do:

```
ptr->next = block;
```

Review!

What would happen if you set ptr->next = NULL?

Listing List Contents

Listing the list contents is easy ... Simply traverse the list and print the contents of each cell as you go!

Setting variables and fields to NULL

Reminders:

- **Always** set the *head* to *NULL* upon declaration
- **Always** set the *next* field of a new node to *NULL* upon either allocation or linking to the list

Result

This produces the simplest type of linked list: an unsorted singly-linked list

We will be looking at variations that make the linked list more convenient to use

E.g.:

- Pointer to end of list
- Sorting a list
- Linking backwards as well as forwards

Review!

The simple linked list that we were introduced to **is** sorted. What is it sorted by?

Memory Allocation

It is vitally important to free the memory in the program before exiting

Freeing Allocated Memory

It's not as easy as just calling free() once, though

You have to free each node individually, through list traversal

You can't free a node and then use its *next* pointer, though!

Example of Freeing Allocated Memory

```
void delete_info (Node *head)
{
    Node *curr = NULL, *ptr = NULL;
    ptr = head;
    // follow the chain until the pointer is null
    while (ptr != NULL)
     {
            // first, save the current cell
            curr = ptr;
            // next, move to the next cell
            ptr = ptr->next;

            // can’t free curr before getting the next pointer
            free (curr);
    }       /* end while */
}   /* end delete_info ()*/
```

Review!

What would happen if you freed the elements without saving the next one?

Finding a Node

Finding a particular node is done through list traversal

Simply set up a traversal algorithm and quit the loop if you find your node

When done the loop, check to see if the pointer that you used is NULL or not

- If it's NULL, you know that you didn't find it

- If it isn't, you know that the pointer points to the node you wanted to find

Deleting a Specific Node

You can’t simply find the node and delete it

- If you do so, you’ll end up orphaning the rest of the list

You have to keep another variable around that keeps track of the **previous** node, since you'll have to link the previous node to the one **after** the one that you're deleting

Exercise

- Set up example provided.
- Run example.
- Understand how it works.
- Change the example to add an option to find a person and print their information.

Review!

What is the main advantage of a simple linked list over an array?

Summary

1. A simple linked list can be used to store data more efficiently.
2. It uses nodes and pointers to nodes.

3. NULL is very important.
4. List traversal is versatile.
5. There’s a lot that can be done with a linked list, especially if you embellish it (next class).