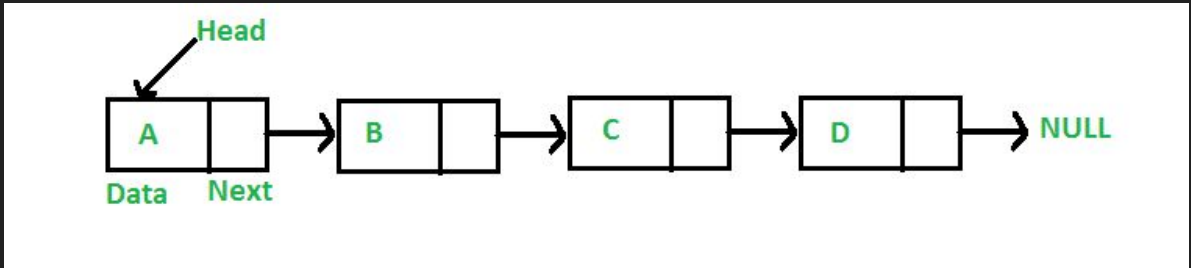
The basics of Algorithms in C++

Nodes, Lists, and Data Structures

The Basic Linked List

All linked lists are built of nodes, these nodes are what we will look at first. A normal node consists of some data, and the next node in sequence. You can think of these as individual containers for data in the list.



Each node (A,B,C and D) contain some data and point to the next node in sequence. The start is called the Head. The final node (The Tail) points to NULL, in C++ a nullptr.

Node Code; Entirely

```
class listNode{
        public:
        int data;
        listNode* next;
        //methods
        listNode(int newData){
                this->data = newData;
                this->next = nullptr;
        }
};
```

Here is a base node class. We will chop up each line of this and discuss what it means. The simplest thing to start with is that we are declaring a **class**. Classes in C++ are just modern structs, and can be used in similar ways. The most important thing to remember is that when instantiating a class, **it returns a pointer to the new object.**

Variables in the class

```
class listNode{
        public:
        int data;
        listNode* next;
        ….
```

Public defines the data as accessible with syntax like **thisNode->data**

Data is our data, an **int** in this case

listNode* next is our pointer to the next node

Let’s focus only on these lines. As we know from the previous slide, the first line is declaring the class. **Public** designates that the below variables and methods are accessible outside the scope of the class. **Data** is obviously the data we are storing in the node. **Next** is important, it is a **pointer** to the next node in sequence.

About Pointers

Pointers are the most important part of all data structures you'll learn. When you declare a pointer variable with * it does not use up memory. When you initialize a class using a line like: **MyClass* test = new MyClass();** you'll notice that you declare it as a pointer. This is because a class is just a fancy struct. You don't need to use malloc or anything as it does it automatically. Like all heap memory objects though **it does not free itself**, therefore you will need to use something like **delete** to free the memory that the object uses. If this isn't done, the program will have a **memory leak**.

The Constructor

```
listNode(int newData){
        this->data = newData;
        this->next = nullptr;
}
```

It’s important that anything that needs to be initialized for a class is done in the constructor. Failure to do so can lead to difficult to debug **Segmentation Faults**, which are the leading cause of dropping out since trade schools.

The name of the class and any variables you want it to be initialized with are done in the constructor. We also use this to define our variables from before. Notice how here we define our data, but we set **next** to be a **nullptr**. A nullptr is essentially NULL, but for pointers. It’s good practice to use this over NULL.

Applying Nodes to a Linked List

A linked list is the simplest application of our nodes. Each node points to the next node in sequence. How you use the Linked List is what defines its application. For example, if you add all new nodes to the front, and only allow retrieval of data from the first node, then you've effectively made a **Stack**. Likewise, if you allow adding data to the front, but only retrieve data from the end, then you've made a **Queue**.

It's important to know the application, but it's more important to know how to design your node.

Printing Data from a Node

Below is a print method. We access data in the method using **this** and **->**. **This** simply refers to the object itself. **->** is a **pointer arrow** and it is how we retrieve data from a pointer. Since **this** is a pointer to itself, we must use the arrow. Anything after the arrow is just data from our object, be it variables or methods.

Ignore the **ofstream&** we will cover this later

```
void printNode(std::ofstream& output){
        if(this->next == nullptr){
                output<<"(nodes data: "<<this->data<<", nexts data: NULL)\n";
        }
        else{
                output<<"(nodes data: "<<this->data<<", nexts data: "<<this->next->data<<")\n";
        }
}
```

Various Node Applications

Linked Lists, Stacks, Queues, Binary Trees, Quad Trees.

The applications are broad since a node itself is just a way of saying “I have this data, here’s where you can go next for more.”

Nodes are limited though. While great for storing data, they have limited functionality alone. The algorithm that defines their sequence (i.e. what’s next) is what gives the algorithm purpose.

Review of What We Learned

We learned about basic node structure. How the Node class is created and what each part of it does. If you want to review classes in C++, you can click this link.

We reviewed pointers, and how they are used in this application. If you want an in depth review of pointers watch this video.

Next we will look at a **Stack Class** and show how it is created. First we will go over what we need, then implement it.

Stacks and What We Will Need

A stack is the most basic data structure. It takes in a value and **pushes** it on top. When we want to retrieve a value, we can only take from that top, so we **pop** it back out. This behavior is known as **Last In First Out(LIFO)** since the last value we add is the first to be removed.

All we need for a stack is it to follow this behavior, so let’s get to defining our stack.

Stack variables

```
class LLStack{
        public:
        listNode* top;
        LLStack(){
                this->top = new listNode(-99999);
        }
        …
```

Fortunately the only variable we need to define for a stack is the **top node**. This node will always be at the top, and all new nodes are added after it. We set it to some dummy value that we can easily identify.

The Push Method

```
void push(listNode* node){
        if(this->top->next == nullptr){
                //empty list add new node
                this->top->next = node;
        }
        else{
                node->next = this->top->next;
                this->top->next = node;
        }
}
```

Our if/else statement saves us from our edge case. The **next** is accessed using **pointer arrows** since our **node** and **this** are both pointers.

To push a new node, we first must give it a node, which is what is in our method header. We must check for an edge case, where our node is the first node in the stack. We find this out by checking if **top**'s next is a **nullptr** if it is, then our new node **will not** have a **next**. Otherwise we set our new nodes **next** to **top**'s **next**, then **top**'s **next** to our node.

The Helpful isEmpty Method

```
bool isEmpty(){
        if(this->top->next == nullptr){
                return true;
        }
        return false;
}
```

Apply yourself! Go back to the previous slide and use **isEmpty()** to simplify our if statement condition.

Sometimes we will want to know if our **stack** is empty, that way we can make decisions based on special cases. We know that our stack is empty if **the next node after top is nullptr**. We simply check for this and return whether if it's true or false. Get used to it, almost every data structure will have to be checked if empty or not.

The Pop Method

```
listNode* pop(){
        if(this->isEmpty()){
                return nullptr;
        }
        else{
                listNode* temp = this->top->next;
                this->top->next = this->top->next->next;
                return temp;
        }
}
```

It's important to make sure your assignment is correct, as failure to do so can easily lead to memory leaks. If we were to just take the data from this node, it would be necessary to **delete** it afterwards to prevent a leak.

The **pop** method will return whatever node we remove. If our stack is empty, we return a **nullptr**. If there is data, since a **stack** is a **LIFO** structure, we want to remove and return the node right after **top**. Our **temp** node is set to the node after top. **Top**'s **next** is then set to the node after it, or **next's next**. Finally we return **temp**.

The Stack, Conclusions

As you can see, with just 3 methods we have created a stack. Being the most simple, it makes sense that it wouldn’t be hard.

As practice you should try implementing it entirely, and running it with some sample data. Click this link to learn about input streams and output streams, an important part in testing since it easily lets you input data.

Beyond The Stack

Everything can be simplified down to this: Determine the structure for your node, then design how the nodes are applied. It’s like building a container and then building the facility that stores it.

Everything though is just a node and a data structure for that node. How you solve the problem is up to you. The key is understanding the basics. Go over it how we did with the stack. We found what we need, what we want to store, and how we want to advance it.

Good Luck with your assignment.

For More:
https://matthewflammia.xyz/

For Music:
https://open.spotify.com/playlist/4bhwDm3U170v7SQB2JEz6h?si=cb523378a54f44a5