Overview

- Our first data structure: Linked List
- Practice some old (or recently new) concepts:
  - Pointers
  - Classes & objects
  - Encapsulation
- Introduces some new concepts at high level:
  - Dynamic memory allocation
  - Analysis of algorithms

We'll describe, create, and analyze a very simple linked list class in this lecture sequence.

Linked Lists

- A linked structure composed of node objects
  - Each node contains one data value
  - Each node contains a pointer to the next node
  - Nodes can be independently created/destroyed
    - ...through the use of dynamic allocation – more soon!

Linked Lists Illustrated

Node Class

Here's a very simple implementation of a node:

```cpp
class node {
    public:
        int data;
        node* next;
};
```

Note NULL pointer in tail node!
(Should use nullptr, but NULL fit better.)

Creating Nodes

Where do nodes come from...?
Don't worry about where just yet, but here's how we do it:

```cpp
node* ptr = new node;
```

Declare a pointer variable (of type pointer to node)
Create (dynamically allocate) a node object.
Make a Linked List

class node {
public:
    int data;
    node* next;
};

int main() {
    node* head;
    head = new node;
    head->data = 14;
    head->next = nullptr;
}

We just made this: 14

Make a Linked List

class node {
public:
    int data;
    node* next;
};

int main() {
    node* head;
    head = new node;
    head->data = 14;
    head->next = nullptr;
}

The -> operator lets us dereference the pointer head and access the member variables of the node pointed to in one step. head->data is equivalent to (*head).data

Making it Longer

int main() {
    node* head;
    head = new node;
    head->data = 14;
    head->next = new node;
    head->next->data = 36;
    head->next->next = new node;
    head->next->next->data = 42;
    head->next->next->next = nullptr;
}

14 36 42

Print List

Let’s print our list. Remember, head is a pointer to our first node.

... for (node* p = head; p != nullptr; p = p->next) {
    cout << p->data << endl;
}

The pointer p moves down the list, pointing to each node in turn.

Why Linked Lists

Seems overly complex: why not just use a vector?

It’s all about trade-offs:

- Vectors are built on arrays
  - On the plus side: random access!
  - Low cost to get/set values at a particular index
  - However:
    - (Relatively) difficult to grow and shrink
    - Insert/remove operations expensive
- Linked lists built on independent nodes
  - Grow/shrink trivial
  - Insert/remove inexpensive-ish
  - However, we lose random access
**Thought Exercise**

What is involved in inserting a value at some index in an array?

E.g., insert the value 27 before value 88.

Do you think it is easier or harder with a linked list?

---

**Linked List Operations: add**

Making a linked list like we did a few slides ago seems a bit silly...

Let’s do it smarter with a function:

- Start with existing list
- Create new node containing requested value
- Add to end

```c
void add_to_tail(node* head, int val)
{
    // start from head, travel down links to find tail.
    node *ptr = head;
    while (ptr->next != nullptr)
    {
        ptr = ptr->next;
    }
    // ptr now points to tail node
    ptr->next = new node;
    ptr->next->data = val;
    ptr->next->next = nullptr;
}
```

---

**Making a Linked List (Do-over)**

```c
int main()
{
    node* head = new node;
    head->data = 14;
    head->next = nullptr;
    add_to_tail(head, 36);
    add_to_tail(head, 42);
    add_to_tail(head, 9);
}
```

---

**Encapsulating Linked List**

Before we add more operations, we should probably make a class.

Advantages:

- Keep track of head/tail pointers so user doesn’t have to
- Prevent user from accidentally messing up list structure
- Keep useful metadata (e.g., size)
- **Encapsulate** data with operations on the data
Linked List Class: Declaration

class linked_list {
public:
    void add_to_tail(int val);
    void print();  // more to come!
private:
    class node {
        public:
            int data;
            node* next;
        };
    node* head = nullptr;
    node* tail = nullptr;
    int size = 0;
};

This declares a class that can only be used within the linked_list class – great for our purposes, because user doesn’t need to know about it!

Note the initializers on these variables; only allowed in C++ 11 and later. We’re using these in place of a default constructor.

Linked List Class: Methods

void linked_list::add_to_tail(int val) {
    // make new tail node
    node* p = new node;
    p->data = val;
    p->next = nullptr;
    // if list is empty, new node becomes both head and tail
    if (_head == nullptr) {
        _head = _tail = p;
    } else {
        _tail->next = p;
        _tail = p;  // update tail only
    }
    _size++;
}

void linked_list::print() {
    for (node* p = _head; p != nullptr; p = p->next) {
        cout << p->data << endl;
    }
}

Stay Tuned…

Part 2 of the lecture:
- More operations
- Analysis of performance
- Applications

Up Next

- Friday, January 18
  - Lab 2 – I/O
  - APT 1 due
  - Project 1 – Personality Test assigned
- Monday, January 21
  - Martin Luther King, Jr. Day – No class
- Wednesday, January 23
  - Linked lists, part 2
  - Lab 2 due
  - Reading: Chapter 14.4 – 14.6