Specifics of a “list”

- List is a set of ordered elements
  - $A_1, A_2, A_3, A_4, \ldots$
  - Refer to the location of an element by its index ($i$)
  - $A_i$ always precedes $A_{i+1}$, etc.
- The size of a list is called $N$
- List with $N = 0$ is called an “empty list”
List operations

- A List defines/provides certain operations:
  - insert (at end, at front, before “i”, etc)
  - remove (end, front, i)
  - get (i)

There are others too… you’ll see more in CMPS 231
Recall our List Container

- Represented as a class / object
- Held a dynamic array as private member to store data
- Could be resized by creating a new sized array and copying over the data
  - **Problem:** This is expensive!

- In addition – inserting at front, in middle forces us to move elements to make room!
- Removal at front in middle? Same problem…
Solution

- To solve this problem — we’re going to combine two important concepts we’ve seen this semester:
  - Pointers and Dynamic Allocation
  - Objects

- We’ll develop a “container” that:
  - Allocates each element as needed.
  - Allows fast insert/remove at front and at end
A chain of pointers

How does this help us?
New Data Structure: Node
Double Linked Lists

Seems more complicated – but makes implementation a bit easier

There are some downsides to this though…

CMPS 231
Initial State

- Let's write the constructor to ensure the list is empty
  - Head
  - NULL
  - Tail

  Might be a good idea to have a public function to determine if list is empty

- Let's write figure out how to add the first item
  - Let's test it by ensuring isEmpty now returns false too.
Insertion – the general case
Insertion – the special cases

- Pointer are great... until they aren’t 😊
  - What happens when we insert into the front of a list?
  - What happens when we insert into the end of a list?

Hope is not a plan... you **must think** about special cases!

Sometimes they work out without special code
... sometimes they don’t
Insertion

Head

Node 0

Node 3

Node 4

Node 12

NULL

Head is just a Node *

Insert after 3

Insert at front

Insert at end

New Node

Insertion
Deletion

Head is just a Node *

Delete 4
Delete first node
Delete last node
Testing

- We don’t know if our code works yet..
  - Print the nodes
  - Implement get(index) to see if things are where we thing they are…
  - Both operations require traversal
List Traversal

```cpp
Node * current = this->head;
while ( current != NULL ) {
    cout << current->getData();
    current = current->getNext();
}
return;
```

console:  4.8  5.2  2.1  11.8
Removal

- We should implement removal code:
  - Remove at front
  - Remove at end
  - Remove at a specified index

Same overall principals

Special Case: Remove from list with 1 element?
Linked List

- **Linked List:**
  - Group of Nodes
  - Each Node contains three things:
    - Data!
    - a pointer to the next Node
    - A pointer to the previous Node

- **LinkedList supports**
  - insert (at end, at front, before “i”, etc)
  - remove (end, front, i)
  - print()
  - get (i)
Memory Management

- Often you’ll want to “clear” the list
  - `makeEmpty()`

- If a list provides a “makeEmpty” operation, there is no guarantee that the programmer will call it.

- Inserting items causes Nodes to be allocated by heap

- Must provide way to automatically call “makeEmpty”
Destructor

- C++ provides special syntax for this:

```cpp
~LinkedList() {
    this->makeEmpty();
}
```

- The destructor is automatically called when the object goes out of scope.
- Also called when delete operator is used.
- Destructor should delete any heap allocated memory within the object.
- Always write a destructor!
Next time...

- This seems like a useful class... but it’s a lot of work
  - It would be helpful if we didn’t have to rewrite the entire thing to hold lists of doubles, characters, strings, circles.....
  - Templates.