Linked Lists
Part One
Outline for Today

● *Linked Lists, Conceptually*
  ● A different way to represent a sequence.

● *Linked Lists, In Code*
  ● Some cool new C++ tricks.
Changing Offices
Dr. Cynthia Lee is no longer in room 100.
She can be found in room 108.
Dr. Cynthia Lee is no longer in room 108.
She can be found in room 190.
Dr. Cynthia Lee is no longer in room 190.

She can be found in room 192.
The Sign on Room 192

Welcome to Cynthia’s Office!

Room 100 → Room 108 → Room 190 → Room 192
Linked Lists at a Glance

- A **linked list** is a data structure for storing a sequence of elements.
- Each element is stored separately from the rest.
- The elements are then chained together into a sequence.
- The end of the list is marked with some special indicator.
Linked Lists at a Glance

- A **linked list** is a data structure for storing a sequence of elements.
- Each element is stored separately from the rest.
- The elements are then chained together into a sequence.
- The end of the list is marked with some special indicator.
Linked Lists at a Glance

- A **linked list** is a data structure for storing a sequence of elements.
- Each element is stored separately from the rest.
- The elements are then chained together into a sequence.
- The end of the list is marked with some special indicator.
Linked Lists at a Glance

- A **linked list** is a data structure for storing a sequence of elements.
- Each element is stored separately from the rest.
- The elements are then chained together into a sequence.
- The end of the list is marked with some special indicator.
A Linked List is Either...

...an empty list,

or...

a single cell...

... that points at another linked list.
Representing Linked Lists
A Linked List is Either...

...an empty list,
or...

...a single cell...

...that points at another linked list.
A Linked List is Either...

...an empty list, or...

a single cell... ...that points at another linked list.
struct Cell {
    string value;
    Cell* next;
};

Hi Mom!

a single cell... 

... that points at another linked list.
struct Cell {
    string value;
    Cell* next;
};
```cpp
struct Cell {
    string value;
    Cell* next;
};

Cell* list = new Cell;
```

We just want a single cell, not an array of cells. To get the space we need, we’ll just say `new Cell`.

Notice that `list` is still a `Cell*`, a pointer to a cell. It still says “look over there for your Cell” rather than “I’m a Cell!”

Yes, it’s a bit confusing that C++ uses the same types to mean “look over there for an array of Cells” and “look over there for a single Cell.”
**struct** Cell {
  string value;
  Cell* next;
};

Cell* list = new Cell;
list->value = "dikdik!";

Because list is a pointer to a Cell, we use the arrow operator `->` instead of the dot operator.

Think of list->value as saying “start at list, follow an arrow, then pick the value field.”
struct Cell {
    string value;
    Cell* next;
};

Cell* list = new Cell;
list->value = "dikdik!";
list->next = new Cell;
list->next->value = "quokka!";
list->next->next = new Cell;
list->next->next->value = "pudu!";
list->next->next->next->next = nullptr;
```cpp
struct Cell {
    string value;
    Cell* next;
};

Cell* list = new Cell;
list->value = "dikdik!";
list->next = new Cell;
list->next->value = "quokka!";
list->next->next = new Cell;
list->next->next->value = "pudu!";
list->next->next->next = nullptr;
```
A Linked List is Either...

...an empty list, represented by `nullptr`, or...

...a single linked list cell that points...

...at another linked list.
Measuring a Linked List
A Linked List is Either...

...an empty list, represented by `nullptr`, or...

...a single linked list cell that points...

...at another linked list.
A Linked List is Either...

...an empty list, represented by `nullptr`, or...

A single linked list cell that points...

...at another linked list.

---

dikdik! → quokka! → pudu!
Printing a Linked List
A Linked List is Either…

...an empty list, represented by `nullptr`, or…

...a single linked list cell that points…

...at another linked list.
A Linked List is Either...

...an empty list, represented by \texttt{nullptr}, or...

\begin{itemize}
\item a single linked list cell that points...
\item \ldots at another linked list.
\end{itemize}

\begin{itemize}
\item \texttt{dikdik!}
\item \texttt{quokka!}
\item \texttt{pudu!}
\end{itemize}
Building a Linked List

(without hardcoding it)
A Linked List is Either...

...an empty list, represented by `nullptr`, or...

```
        (empty)

        (cell that points...)
```

...at another linked list.
Cleaning Up a Linked List
Endearing C++ Quirks

- If you allocate memory using the `new[]` operator (e.g. `new int[137]`), you have to free it using the `delete[]` operator.
  
  ```
  delete[] ptr;
  ```

- If you allocate memory using the `new` operator (e.g. `new Cell`), you have to free it using the `delete` operator.
  
  ```
  delete ptr;
  ```

- **Make sure to use the proper deletion operation.** Mixing these up leads to Undefined Behavior.
Cleaning Up Memory

- To free a linked list, we can't just do this:
  ```
  delete list;
  ```
- Why not?
Cleaning Up Memory

- To free a linked list, we can’t just do this:

```cpp
delete list;
```

- Why not?

```
list
```

---

**Dynamic Deallocation!**
Cleaning Up Memory

- To free a linked list, we can’t just do this:
  ```
  delete list;
  ```
- Why not?

```
```

```
A Linked List is Either...

...an empty list, represented by `nullptr`, or...

...a single linked list cell that points...

...at another linked list.
void deleteList(Cell* list) {
    if (list == nullptr) return;
    delete list;
    deleteList(list->next);
}
void deleteList(Cell* list) {
    if (list == nullptr) return;
    delete list;
    deleteList(list->next);
}

Watch Out!
void deleteList(Cell* list) {
    if (list == nullptr) return;
    delete list;
    deleteList(list->next);
}

Watch Out!

void deleteList(Cell* list) {
    if (list == nullptr) return;
    delete list;
    deleteList(list->next);
}
void deleteList(Cell* list) {
    if (list == nullptr) return;
    delete list;
    deleteList(list->next);
}

Watch Out!

Dynamic Deallocation!
void deleteList(Cell* list) {
    if (list == nullptr) return;
    delete list;
    deleteList(list->next);
}
void deleteList(Cell* list) {
    if (list == nullptr) return;
    delete list;
    deleteList(list->next);
}
void deleteList(Cell* list) {
    if (list == nullptr) return;
    delete list;
    deleteList(list->next);
}

Watch Out!
void deleteList(Cell* list) {
    if (list == nullptr)
        return;
    delete list;
    deleteList(list->next);
}

Watch Out!

Undefined behavior!
In the Land of C++, we do not speak to the dead.

What should we do instead?
void deleteList(Cell* list) {
    if (list == nullptr) return;
    Cell* next = list->next;
    delete list;
    deleteList(next);
}

One Option

Gerenuk ➔ Quokka ➔ Pudu ➔
```c
void deleteList(Cell* list) {
    if (list == nullptr) return;
    Cell* next = list->next;
    delete list;
    deleteList(next);
}
```
```c
void deleteList(Cell* list) {
    if (list == nullptr) return;
    Cell* next = list->next;
    delete list;
    deleteList(next);
}
```
```c
void deleteList(Cell* list) {
    if (list == nullptr) return;
    Cell* next = list->next;
    delete list;
    deleteList(next);
}
```
void deleteList(Cell* list) {
    if (list == nullptr) return;
    Cell* next = list->next;
    delete list;
    deleteList(next);
}
```c
void deleteList(Cell* list) {
    if (list == nullptr) return;
    Cell* next = list->next;
    delete list;
    deleteList(next);
}
```
void deleteList(Cell* list) {
    if (list == nullptr) return;
    Cell* next = list->next;
    // delete list;
    deleteList(next);
    // list
    // next
    delete
}

Dynamic Deallocation!
```c
void deleteList(Cell* list) {
    if (list == nullptr) return;
    Cell* next = list->next;
    delete list;
    deleteList(next);
}
```
void deleteList(Cell* list) {
    if (list == nullptr) return;
    Cell* next = list->next;
    delete list;
    deleteList(next);
}
void deleteList(Cell* list) {
    if (list == nullptr) return;

    Cell* next = list->next;
    delete list;
    deleteList(next);
}

Recursion!
```c
void deleteList(Cell* list) {
    if (list == nullptr) return;
    Cell* next = list->next;
    delete list;
    deleteList(next);
}
```
void deleteList(Cell* list) {
    if (list == nullptr) return;
    deleteList(list->next);
    delete list;
}
void deleteList(Cell* list) {
    if (list == nullptr) return;
    deleteList(list->next);
    delete list;
}
void deleteList(Cell* list) {
    if (list == nullptr) return;
    deleteList(list->next);
    delete list;
}

Another Option
```c
void deleteList(Cell* list) {
    if (list == nullptr) return;
    deleteList(list->next);
    delete list;
}
```
void deleteList(Cell* list) {
    if (list == nullptr) return;
    deleteList(list->next);
    delete list;
}

Another Option
```c
void deleteList(Cell* list) {
    if (list == nullptr) return;
    deleteList(list->next);
    delete list;
}
```
void deleteList(Cell* list) {
    if (list == nullptr) return;
    deleteList(list->next);
    delete list;
}

Another Option

Dynamic Deallocation!
void deleteList(Cell* list) {
    if (list == nullptr) return;
    deleteList(list->next);
    delete list;
}
Your Action Items

• **Read Chapter 12.1 - 12.3.**
  • There’s lots of useful information in there about how to work with linked lists.

• **Work on Assignment 6.**
  • Aim to complete Linear Probing and to have started Robin Hood hashing by Wednesday.
Next Time

- **Linked Lists, Iteratively**
  - How do you manually walk a linked list?
- **Pointers by Reference**
  - Combining two methods of indirection!