# CSE 1320 - Intermediate Programming

## Linked Lists

Alex Dillhoff

University of Texas at Arlington

# Linked Lists

Pointers and dynamic memory allocation enable a new data structure called a **Linked List**.

A linked list is a dynamic and aggregate data structure that is used to store any type.

# Linked List - Definition

A linked list is defined as a linear sequence of connected nodes.

In its most basic form, each node has a pointer to the subsequent node.

# Linked List - Definition

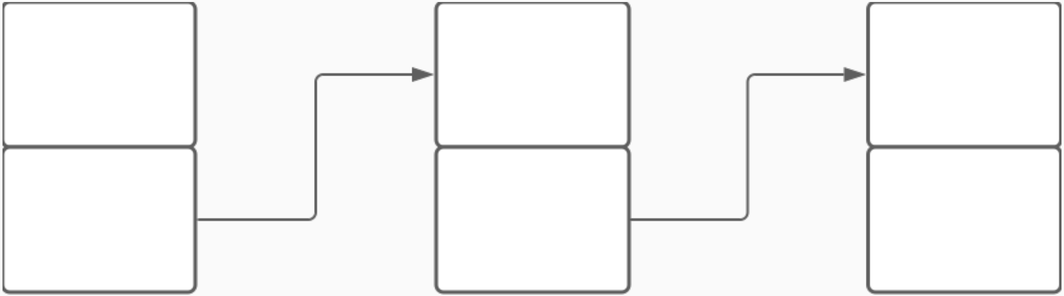

Figure 1: Diagram of a linked list with 3 nodes.

# Linked List - Definition

In C, a node of a linked list is implemented as a **`struct`**.

```
struct node {
    int val;
    struct node *next;
};
```

# Linked List - Definition

```
struct node {
    int val;
    struct node *next;
};
```

The value `val` can be any type, even a complex type like another `struct`.

# Linked List - Definition

```
struct node {
    int val;
    struct node *next;
};
```

This definition allows the list to grow dynamically.

The `next` pointer points to the next node in the list.

The last node points to `NULL`.

# Linked List - Definition

```
struct node {
    int val;
    struct node *first;
    struct node *previous;
    struct node *next;
    struct node *last;
};
```

Each node can include more useful pointers, such as a pointer to the first node, previous node, and last node.

# Linked List - Operations

Nodes can be inserted and removed by modifying the node pointer(s).

# Linked List - Insert

A node is inserted by first allocating memory for the node itself and then setting the pointer of the previous node.

```
struct node *a = malloc(sizeof(struct node*));
struct node *b = malloc(sizeof(struct node*));
a->next = b;
```

# Linked List - Remove

A node is removed by `free`ing the memory allocated to the node and modifying any dependent nodes.

```
free(b);
a->next = NULL;
```

# Linked List - Remove

If the node being removed has dependent's of its own, those links are updated as well.

For example, given a linked list a -> b -> c and a removal of b.

```
a->next = b->next;
free(b);
```

# Linked Lists - Examples

1. Create a function which allocates and adds a node to a list.
2. Traverse a linked list.
3. Release all data of a linked list.
4. Reverse a linked list in a single loop.
5. Explore lists with multiple nodes.

# Linked Lists - Efficiency Analysis

Compared to an array, how efficient is each operation of a linked list?

**Indexing**

An array member can be accessed in constant time (e.g. `arr[i]`).

For a linked list, the node must be found by cycling through the list. In the worst case, it will take as many cycles as there are nodes.

# Linked Lists - Efficiency Analysis

**Insert/Delete at Beginning**
Since the first node is always known, this can be done in constant time by simplying adding the node and updating the `next` pointer.

A fixed array cannot perform this operation and a dynamic array must reallocate the memory, move all members forward by 1, and then insert the new data.

# Linked Lists - Efficiency Analysis

**Inserting/Deleting at the End**
In a linked list, this only takes a single operation assuming there is a pointer to the last node in the list. If not, the list must be traversed from the beginning before adding the new node.

For an array, once new memory is allocated the new data can be added in a single operation.

# Linked Lists - Efficiency Analysis

**Memory Considerations**

In its most basic form, each node of a linked list requires space for the `struct node` which includes whatever data type the value is. For example, using 8 byte addresses, a node which contains a single character requires 9 bytes.

A similar array would requires 9 times **LESS** memory to represent the same data.