Linked List:

① Singly-linked list: each element contains the data for the element and a pointer to the next element

② Doubly-linked list: each element contains the data for the element, a pointer to the next element, and a pointer to the previous element

Example Uses: mailing lists, cooking recipes (ingredients list), file systems, in other data structures/algorithms

↳ what if we wanted to report all the eyes on a go board, not just the number?

Why use lists instead of arrays?:

① Efficiency: many operations can be more efficiently implemented using lists

② Abstraction: Interface provided by a list often more closely follows the intended operation.

Common List Operations:

Initialization: Initialize data structure for list to an empty list

→ list_init()

Destruction: Destroy linked list and free any memory used

→ list_destroy()

Insertion: Insert a new item into the list, often a relative insertion given an existing element

→ list_ins_next()

Removal/Deletion: Remove an item from the list, often a relative removal

→ list_rem_next()

Size: Determine size of list
→ list_size()

Head/Tail: Determine the element at the start (head) or end (tail) of list
→ list_head()
→ list_tail()

Movement: Function to move to next (or prev) list element
→ list_next()

Example (List of strings):

+ Each list element must contain a char * (for string) and a pointer to the next list element.

+ Define a structure using struct declaration to group these items into a single type

```
struct ListElmt {
    char *str;          // List element data
    struct ListElmt *next;   // Pointer to next list element
};
```

⇓

Defines new type "struct ListElmt". Not always convenient to use "struct ListElmt", so it is often combined with a type declaration

```
typedef struct ListElmt_ {
    char *str;
    struct ListElmt_ *next;
} ListElmt;
```

≡ (equivalent to)

```
struct ListElmt_ {
    char *str;
    struct ListElmt_ *next;
};
typedef struct ListElmt_ ListElmt;
```

⇓

Defines a new type "struct ListElmt_" and "ListElmt" that are equivalent

Example (List of strings) (continued):

+ Need to define a structure for the list itself:

```
typedef struct List_ {
    int size;              // keeps track of number of elements
    ListElmt *head;        // pointer to first element
    ListElmt *tail;        // pointer to last element
} List;
```

+ Initialization:

```
void list_init (List *list) {
    list->size = 0;
    list->head = NULL;
    list->tail = NULL;
}
```

"->" operator can be used to access the elements inside a structure from a pointer to that structure.

list->size = 0; ≡ (*list).size = 0;

+ Insertion:

```
int list_ins_next(List *list, ListElmt *element, char *str) {
    ListElmt *new_element;

    if( (new_element = (ListElmt *)malloc(sizeof(ListElmt))) == NULL) return -1;

    new_element->str = str;

    if (element == NULL) {
        if ( list->size == 0) {
            list->tail = new_element;
        }
        new_element->next = list->head;
        list->head = new_element;
    }
    else {
        if(element->next == NULL) {
            list->tail = new_element;
        }
        new_element->next = element->next;
        element->next = new_element;
    }
    list->size++;
```

creates and inits the new element

insert at head of list

insert somewhere else

Example (List of strings) (continued):

+Removal:

```
int list_rem_next(List *list, ListElmt *element) {
    ListElmt *old_element;

    if (list->size == 0) return -1;

    if (element == NULL) {
        old_element = list->head;
        list->head = list->head->next;
        if (list->size == 1) list->tail = NULL;
    }
    else {
        if (element->next == NULL) return -1;
        old_element = element->next;
        element->next = element->next->next;
        if (element->next == NULL) list->tail = element;
    }
    free(old_element->str);
    free(old_element);

    list->size--;
}
```

remove head of list

Example (List of strings) (continued):

+Head / Tail:

```
ListElmt * list_head (List * list) {
    return list->head;
}

ListElmt * list_tail (List * list) {
    return list->tail;
}
```

+Movement:

```
ListElmt * list_next (ListElmt * element) {
    if (element == NULL) return NULL;
    return element->next;
}
```

+Destructor:

```
void list_destroy (List * list) {
    while (list->size > 0) {
        list_rem_next (list, NULL);
    }
    memset(list, 0, sizeof (List));
    return;
}
```