Agenda (3/25)

- Linked Lists

Exam 2: Tuesday, 4/2
- recursion
- 2D arrays
- sorting
- strings

Linked lists are dynamic data structures
- grow and shrink on demand (unlike array)

+ convenient to add/remove elements in the middle of a list of data
+ convenient to add/remove elements from either end of the list.

- looking up a specific element in the list ("what is the q^{th} element)
- finding elements in the list ("is the number 12 in the list")

- holds a list of elements by maintaining a sequence of structures; each structure holds two things:
  1) a piece of data (e.g. a number)
  2) a pointer to the next part of the linked list

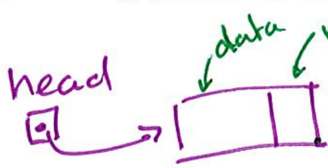
"nodes"

```
struct Node {
    int data;          // piece of data
    struct Node *next; // pointer to the next node
};
```

Build a linked list with the ~~elem~~ numbers 3, 7, 12.

```
struct Node * head; // a pointer to the beginning of the list

head = malloc(sizeof(struct Node));
```




```
head->data = 3; // (*head).data = 3
```


```
head->next = malloc(sizeof(struct Node));
```


```
head->next->data = 7;
head->next->next = malloc(sizeof(struct Node));
head->next->next->data = 12;
head->next->next->next = NULL;
```

Print out the data in the list

```
printf("%d", head->data);
   ____ , head->next->data);
   ____ , head->next->next->data);
```

```
printf("%d", head->next->next->next->data)
```

← NULL, segfault

All of this code sucks to write:

- tedious
- error prone (use too many/few next pointers)
- isn't dynamic — assumes the list is a certain size, which defeats the whole purpose of a linked list.

Let's write code that actually works on dynamic data structures — doesn't care how big the list is.

I) prints out data in the list.

standard trick: use a pointer that points to the "current" node in the list.

```
struct Node * cur; // "current" or "cursor"
cur = head; // cur points to the ~~current~~ head of the list
```

idea: if ~~current~~ points to a valid node:
- print the data
- move cur down the list and repeat

if it doesn't point to a valid node, we're done.

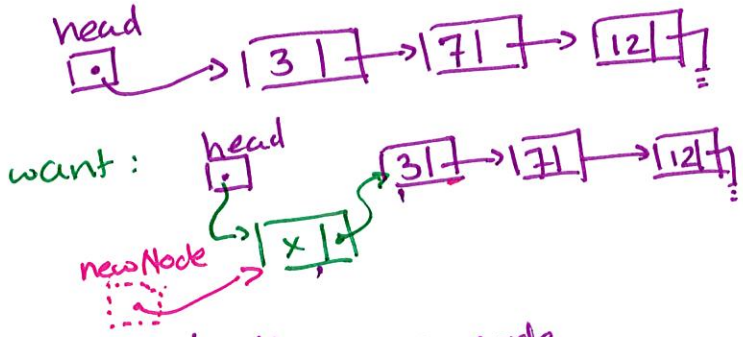
```
while (cur != NULL) { // while cur doesn't point to NULL
    printf("%d ", curr -> data); // print the data in the
                                 Node cur points to.
    cur = cur -> next; // the next Node is pointed to
                       by the next field of the
                       current node
}
```

II) write a function that decides whether a value is in a list.

```
bool contains(int key, struct Node * head){
    struct Node * cur = head;
    while (cur != NULL) {
        if (cur->data == key) { return true; }
        cur = cur->next;
    }
    return false;
}
```

(key: the value you are looking for; head: ptr of the list to search)

III) write a piece of code (not a function) that adds an element to the front of the list.



1. create the new node
2. make the new node point to the beginning of the list
3. update head to point to the new node.

step 1
```
struct Node * newNode = malloc(sizeof(struct Node));
newNode->data = x;
```

step 2
```
newNode->next = head;
head = newNode;
```

careful: data winds up in the list in the reverse ~~opposite~~ order that you added it.

IV) lift this out into a function.

attempt #1: (this is wrong)

let's just move the code into a function, and make head & x arguments.

```
void add(int x, struct Node * head) {
    struct Node * newNode = malloc(sizeof(struct Node
    newNode -> data = x;
    newNode -> next = head;
    head = newNode;
    return;
}
```

why doesn't this work?

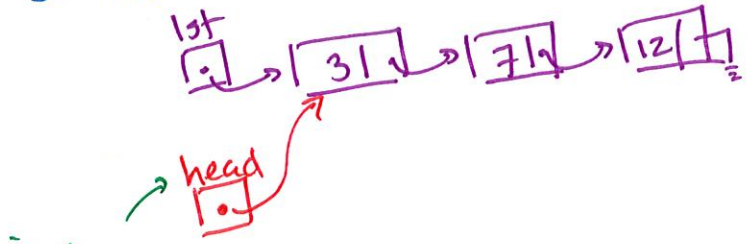
1. built a list pointed to by lst:

```
struct Node * lst = malloc ...
        :
```

2. try to add 2 to lst:

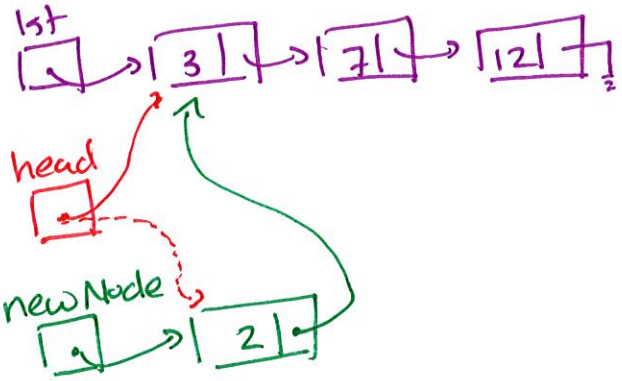
```
add(2, lst);
```

3. inside of add, head is a copy of lst. inside the function:



is a copy of lst.

4. when I add 2, I'm messing with head not lst.



lst never changes. so it still points to the old data. and the new Node leaks.

to fix this, we pass in a pointer to the ~~head of~~ the head pointer (pass the address of the head pointer)

```
                                          pointer to the head pointer
void add (int x, struct Node ** headPtr) {
    struct Node * newNode = malloc (sizeof (struct Node));
    newNode -> data = x;
    newNode -> next = *headPtr; // dereference headPtr to get to head.
    *headPtr = newNode;
}
```

add (2, &lst) ← pass in the address of the pointer we want to change.