Agenda (3/27)
- more linked lists
- stacks
- queues.

Reminder: Exam 2 is next Tuesday (4/2)
- recursion
- sorting
- strings
- 2D Arrays.

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

struct Node * head = NULL; // list has no elements

void insert ( struct Node * * locptr, int key) {
    struct Node * newNode = Malloc (sizeof(struct Node))
        newNode -> val = key;
        newNode -> next = * locptr ;
        *locptr = newNode;
        return;
}
```

to add 7 to the list:

```
insert (& head, 7);
```

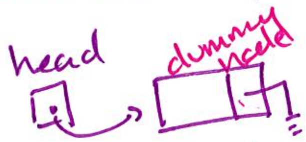
There is no one way to write Linked List functions

key feature: a node structure that has data and a pointer to the next Node.

=> a function like insert needs to get direct access to the Node pointer it wants to change. Not a copy some times you might see linked lists that use a "dummy" head node — a node that acts as the head of the list without holding data.

for example:

```
struct Node * head = malloc( sizeof(struct Node *));
//allocate a Node that doesn't hold data.
```

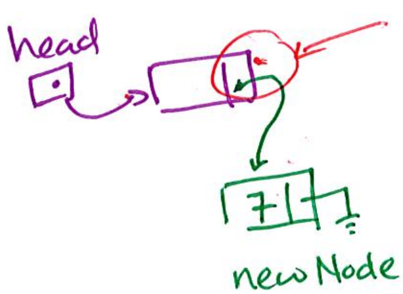



```
// write insert in a different way:

void insert ( struct Node * head, int key) {
    struct Node * newNode = malloc( ——— )
    newNode -> val = key;
    newNode -> next = head -> next
    head -> next = newNode;

}

insert (head, 7)
```

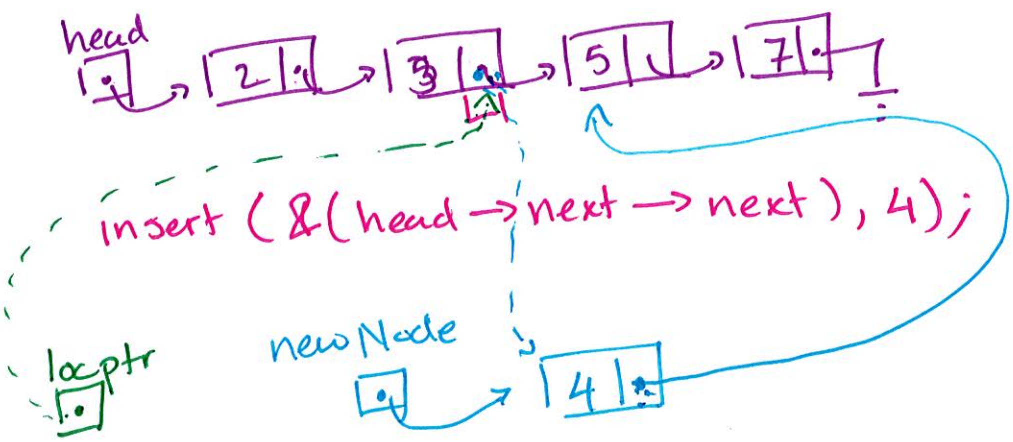


the key thing we needed was direct access to this next pointer

"head" is still a pointer to the thing we want to change.

Note: because insert works on the address of the pointer you want to change, let's see what happens if we pass in a different pointer's address.

insert(&head, 2);



rewired the linked list <u>from the middle</u>.

insert puts a new node right after the next pointer you pass in.

IV) delete a node from the linked list

try one: the node head points to.

```
head = head->next;
```

leaks memory, so let's first remember what we need to free

```
struct Node * toDelete = head;
head = head->next;
free(toDelete);
```

VI) lift that into a function.

```
void delete ( struct Node ** locptr) {    // the address of the pointer that points to the node we want to delete
    struct Node * toDelete = * ~~head~~ locptr;
    *locptr = (*locptr) -> next;
    free (toDelete)
}
```

```
delete (&head) ; // deletes the first node from the list.
```

```
delete ( &(head -> next)); // deletes the second node from the list.
```

VII) delete all the nodes in a list?

```
while ( head != NULL) {
    delete(&head)    // don't forget, we're working on pointers to pointers
}
```

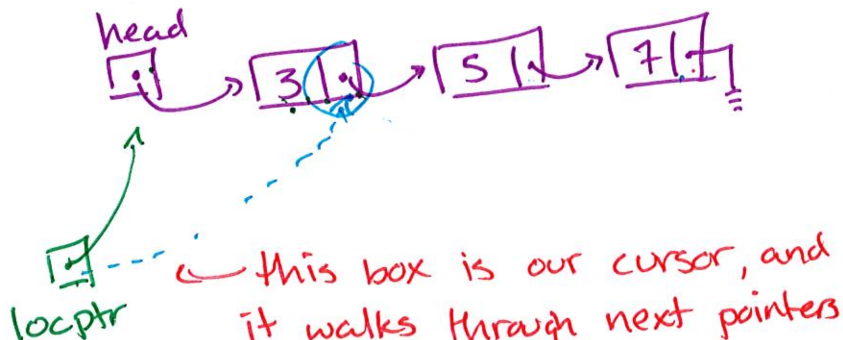
VIII) find a value in the list.

i) if the node is in the list, return the address of the pointer that points to it.

(if it's the first element, return &head.
if it's the second element, return &(head->next))

ii) if the node isn't in the list, return the address of the last next pointer (which points to NULL)

```
struct Node ** findEq(struct Node ** locptr, int key) {
            walk through list
    while ((*locptr) != NULL) {
        if ((*locptr)->val == key) return locptr;
        locptr = &((*locptr)->next);
    }
    return locptr;
}
```



IX) delete a particular key from the list if it exists.

1) figure out if the ~~[illegible]~~ key is in the list

```
struct Node ** toDelete = findEq(&head, key)
```

2) if the key is in the list, remove it:

```
if (*(toDelete) != NULL) { // key is in the list
    delete(toDelete);
}
```

Most common uses of lists are to build two restricted versions:

1: Stack (~~first-in, first-out~~ last-in, first-out)

push i) add an element to the front of stack

pop ii) remove an element from the front of the stack.

peek iii) look at the element at the front.

push is just inserting a node at head:

pop is just removing the node head points to

peek is just looking at the first node.

```
struct Stack {
    struct Node * head;
    int size;
}

void push (struct Stack * s, int key) {
    insert(&(s->head), key);
    size++
}

int pop(struct Stack * s) {
    if (size < 1) { handle error }
    int retval = s->head->val;
    delete (&(s->head));
    size--;
    return retval;
}
```

2: Queue (first-in, first-out)

add to one end, delete from the other

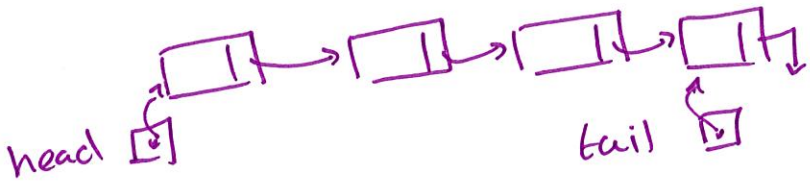
1) enqueue (get in line)

   ↳ ~~add to head~~ add to the end

2) dequeue (leave line)

   ↳ removes from the front

just have a pointer whose job is to point to the end of the list : tail




```
struct Queue {
    struct Node * head;
    struct Node * tail;
    int size;
}

void enqueue (struct Queue * q, int key) {
    if (size == 0) {
      q -> head = malloc ( — );   } special case for
      q -> tail = head;           }   the first node
    size ++
    return
    }
    // insert a new Node after tail

    struct * newNode = malloc( — );
    newNode -> val = key;
    q -> tail -> next = newNode; // put new node
                                    after tail
    q -> tail = q -> tail -> next;
    size ++
    return }
```