Review: Linked List

College of Engineering

School of Computer Science and Engineering

ARRAYS

- Items have to be stored in **contiguous** block
- No gaps in between items
- **Easy to random access to items in the sequence.**

  e.g., the ith item can be accessed by arr[i-1]
- Difficult to expand, re-arrange
- When inserting/removing items in the middle or at the front, computation time scales with size of list
- Generally a better choice when data is immutable

arr[0]	arr[1]	arr[2]	arr[3]	arr[4]			
20	30	50	60	70			
No.1	No.2	No.3	No.4	No.5			

NODES

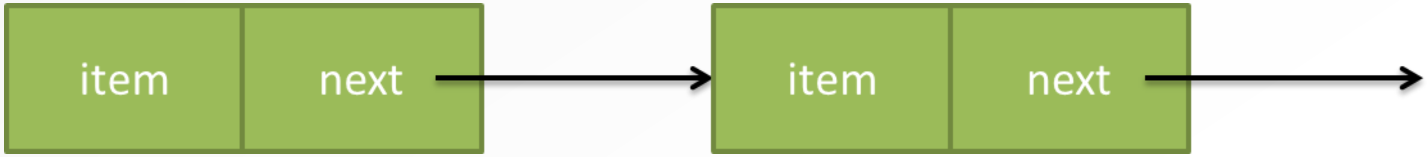
- Node-based data structures
  - Nodes + connections between nodes
- Data structure size is not fixed
  - Can create a node at any point while the program is running
  - Dynamic memory allocation malloc(): malloc(sizeof(…))
  - Deallocation of dynamic memory free()
  - **Common mistakes: memory leak, buffer overflow**
- Pointers vs nodes
  - Pointers create connections between nodes
  - Pointers are not nodes

IMPLEMENTATION OF NODE

- Implementation details differ across languages
- But same fields will always be there:
  - data
  - connection(s) to other node(s)
- In C, ListNode is a C struct with several fields
  - item: this is a data type holding the data stored in the node
  - next: this is a pointer storing the address of the next node in the sequence

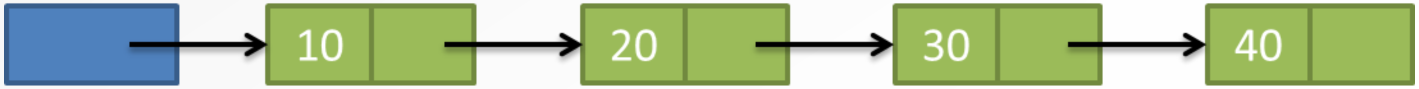
```
typedef struct _listnode{
      int item;
      struct _listnode *next;
 }ListNode;
```

MINIMUM SETTINGS



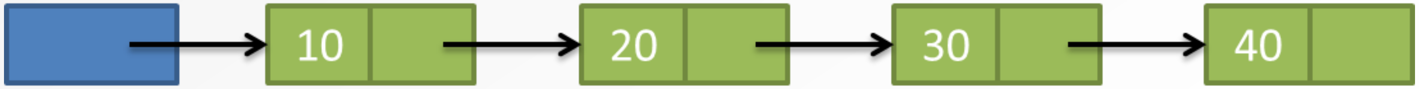
LINKED LISTS

- What is a linked list?
  - Ordered list of items
  - Each item stored in a node
  - Each node connects to the next node in the series
- No need for pointers in definition of a linked list
  - Head pointer, next pointer: all <u>implementation</u> details



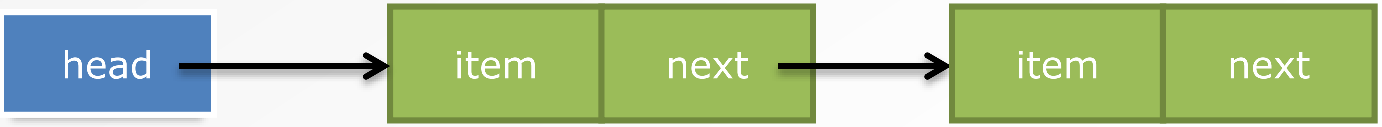
BASIC LINKED LIST

- Different types of data can be stored in a node
- Singly-linked list
  - Each node is connected to at most one other node
  - Each node keeps track of the next node

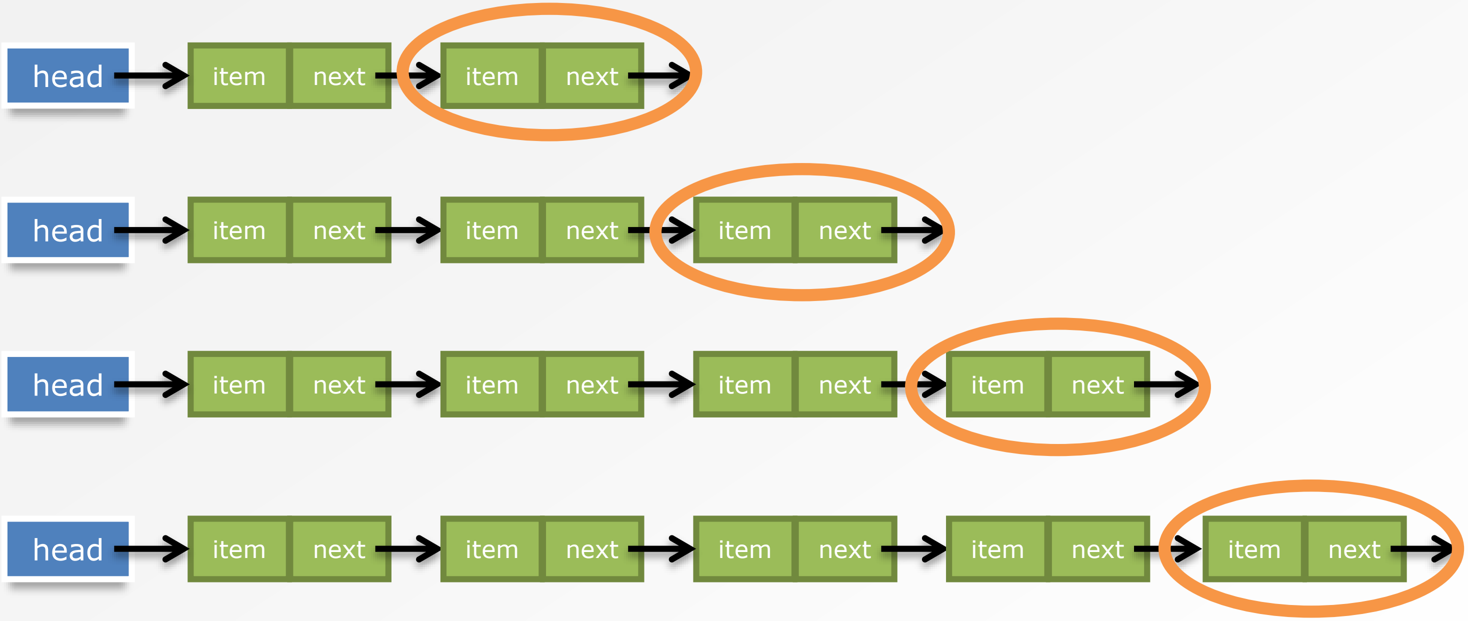


BACK TO LAB QUESTION: STORE A LIST OF NUMBERS

- Previously, we used malloc() to create int array to store all numbers after numOfNumbers was known
- This time, use malloc() to create a new ListNode for each number
  - Get input until input == -1
  - For each input number, create a new node to store the value
  - Arrange all the ListNodes as a linked list

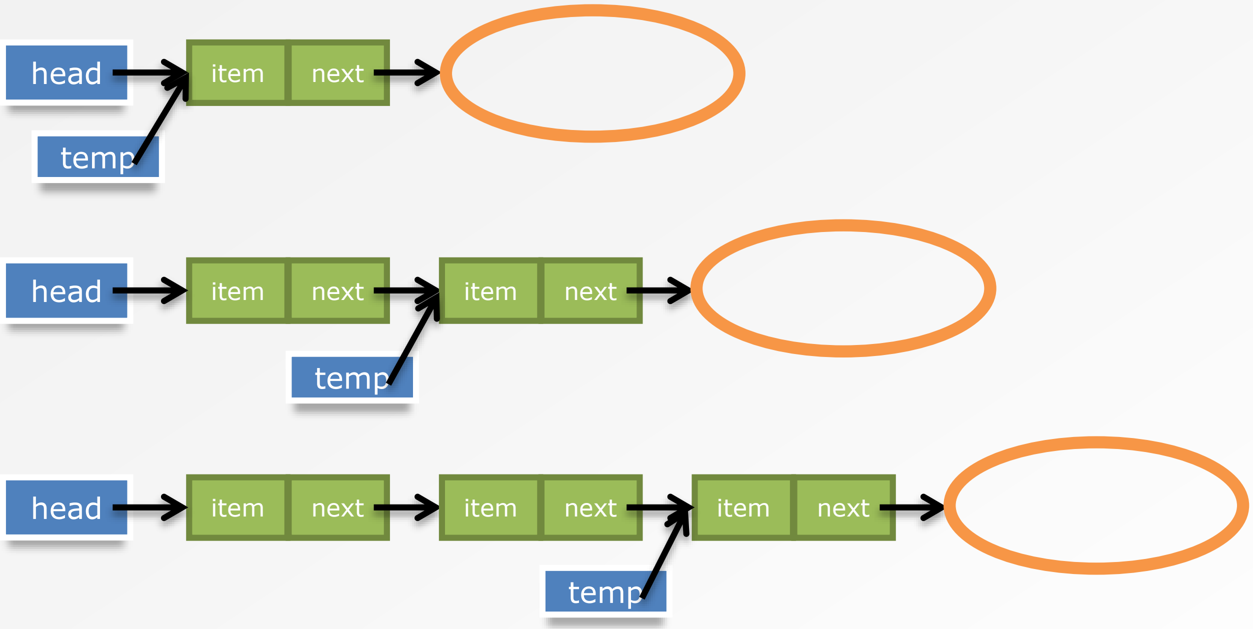


BACK TO LAB QUESTION: STORE A LIST OF NUMBERS



- Address of each new ListNode is saved in next pointer of previous node
- Need a way to keep track of the last ListNode at any time
  - Use another pointer variable

BACK TO LAB QUESTION: STORE A LIST OF NUMBERS



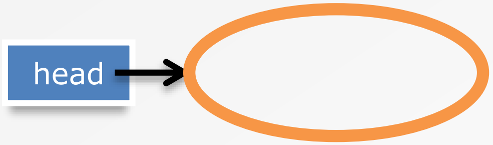
- *temp* pointer stores address of the last ListNode at any time
- Create a new ListNode

```
temp->next = malloc(sizeof(ListNode));
```

BACK TO LAB QUESTION: STORE A LIST OF NUMBERS

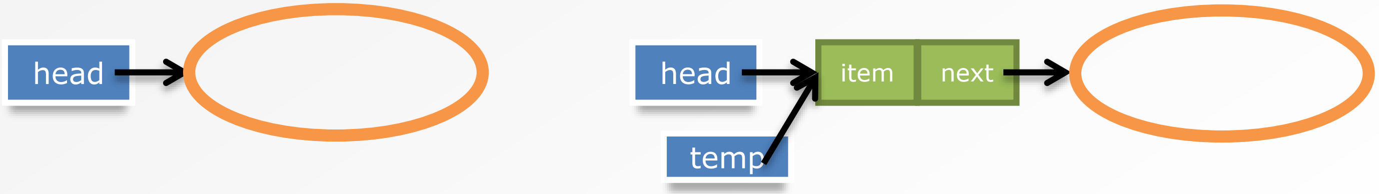
- Watch out for special case
  - First node in the linked list
  - *head* == NULL
  - Need to update the *head* pointer

```
head = malloc(sizeof(ListNode));
```



BACK TO LAB QUESTION: STORE A LIST OF NUMBERS

- After the first ListNode has been created
  - *head* pointer points to first ListNode
  - Can now use *temp* pointer to keep track of last node
  - In this case, *temp* also points to the first ListNode



SINGLY-LINKED LIST OF INTEGERS

```
typedef struct node{
    int item;  struct node *next;
} ListNode;

int main(){
    ListNode *head = NULL, *temp;
    int i = 0;

    scanf(“%d”, &i);
    while (i != -1){
        if (head == NULL){
            head = malloc(sizeof(ListNode));
            temp = head;
        }
        else{
            temp->next = malloc(sizeof(ListNode));
            temp = temp->next;
        }
        temp->item = i;
        scanf(“%d”, &i);
    }
    temp->next = null;
}
```

Quite silly to do this manually every time

Also, this code can only add to the back of a list

Write a function to add a node (other functions too)

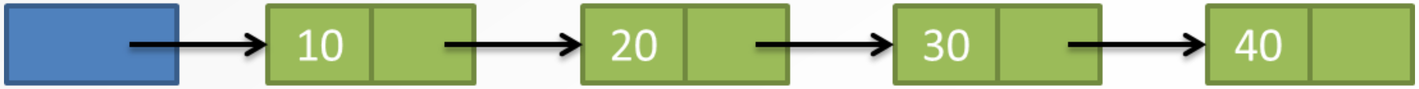
LINKED LIST FUNCTIONS

- Our linked list should support some basic operations
  - Inserting a node `insertNode()`
    - At the front
    - At the back
    - In the middle
  - Removing a node `removeNode()`
    - At the front
    - At the back
    - In the middle
  - Printing the whole list `printList()`
  - Looking for the node at index n `findNode()`
  - Etc.

LINKED LIST FUNCTIONS USING LISTNODE STRUCT

Function prototypes:

- `void printList(ListNode *head);`
- `ListNode * findNode(ListNode *head, int index);`
- `int insertNode(ListNode **ptrHead, int index, int value);`
- `int removeNode(ListNode **ptrHead, int index);`



Content Copyright Nanyang Technological University

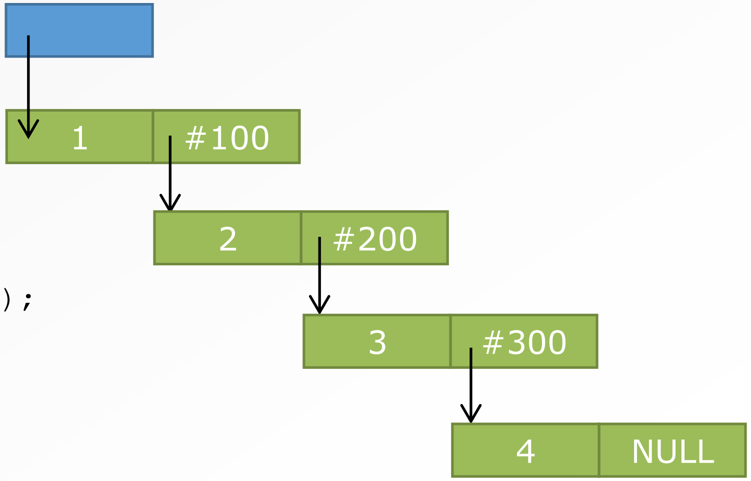
PRINT OUT ITEMS IN LINKED LIST: printList()

- Print all the items by starting from the first node and traversing the list till the end is reached
- Pass head pointer into the function

```
void printList (ListNode *head)
```

  - At each node, use the next pointer to move to the next node

```
void printList(ListNode *head){

    if (head == NULL)
        return;

    while (head != NULL){
        printf("%d ", head->item);
        head = head->next;
    }
    printf("\n");
}
```



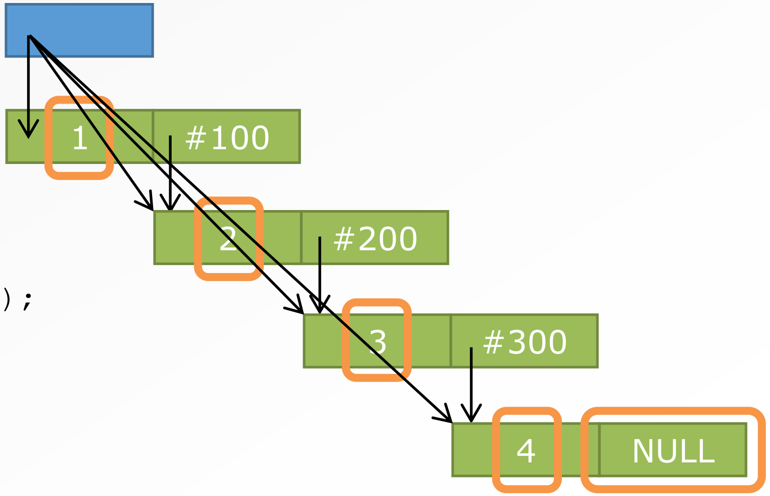
PRINT OUT ITEMS IN LINKED LIST: printList() [ANIMATED]

- Print all the items by starting from the first node and traversing the list till the end is reached
- Pass head pointer into the function

```
void printList (ListNode *head)
```

  - At each node, use the next pointer to move to the next node

```
void printList(ListNode *head){

    if (head == NULL)
        return;

    while (head != NULL){
        printf("%d ", head->item);
        head = head->next;
    }
    printf("\n");
}
```



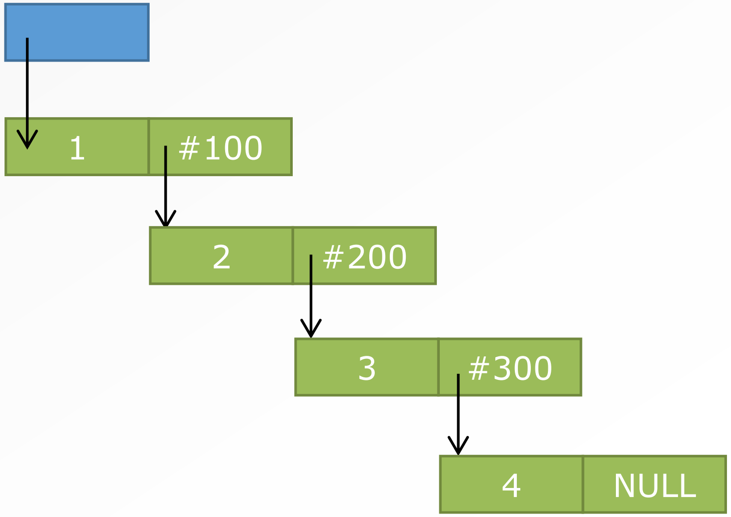
GET POINTER TO NODE AT INDEX i: findNode()

- This function will come in useful later
- Pass head pointer into the function

```
ListNode * findNode(ListNode *head, int index)
```

- Count down *index* times (let’s try index = 2)
  - To get to index 2 (the 3rd node), we need to follow 2 next pointers

```
ListNode * findNode(
    ListNode *head, int index){

    if (head == NULL || index < 0)
        return NULL;

    while (index > 0){
        head = head->next;
        if (head == NULL)
            return NULL;
        index--;
    }
    return head;
}
```



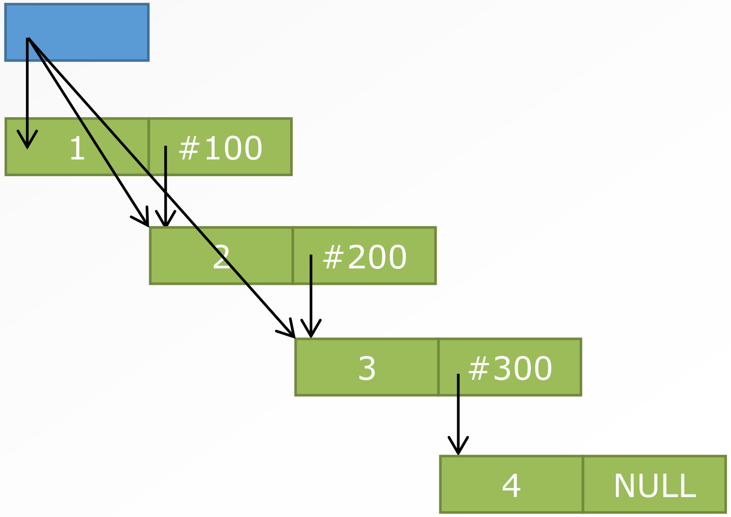
GET POINTER TO NODE AT INDEX i: findNode() [ANIMATED]

- This function will come in useful later
- Pass head pointer into the function

ListNode * findNode(ListNode *head, int index)

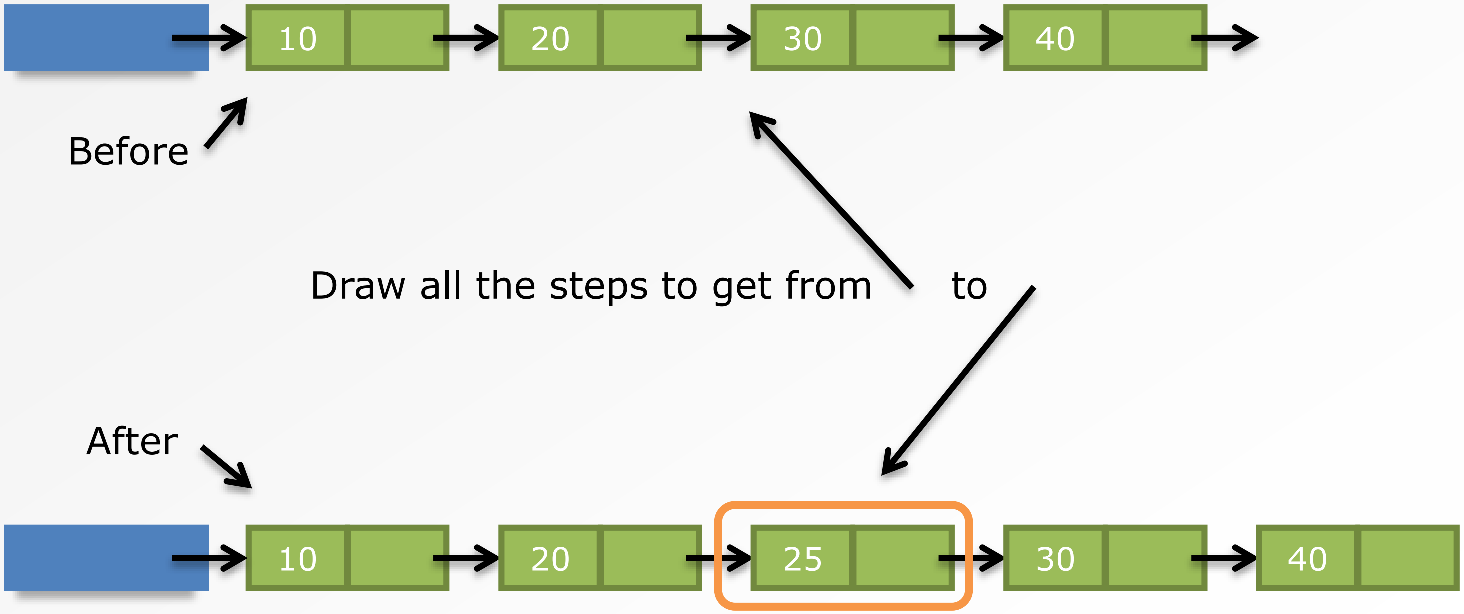
- Count down *index* times (let's try index = 2)
  - To get to index 2 (the 3rd node), we need to follow 2 next pointers

```
ListNode * findNode(
    ListNode *head, int index){

    if (head == NULL || index < 0)
        return NULL;

    while (index > 0){
        head = head->next;
        if (head == NULL)
            return NULL;
        index--;
    }
    return head;
}
```



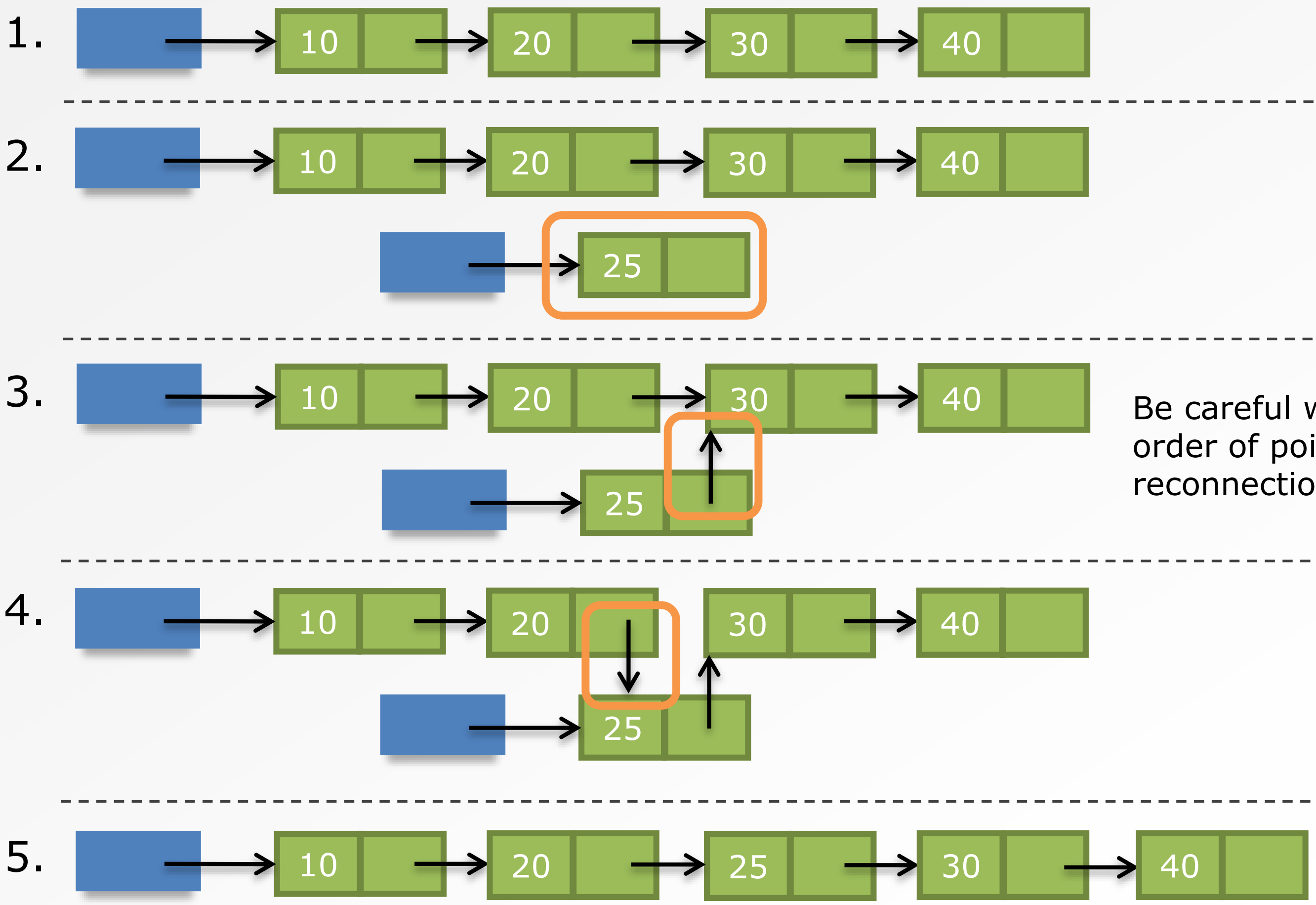
INSERT A NODE: insertNode()

- Adding a node (25) in the middle of a linked list with many existing nodes



Draw all the steps to get from to

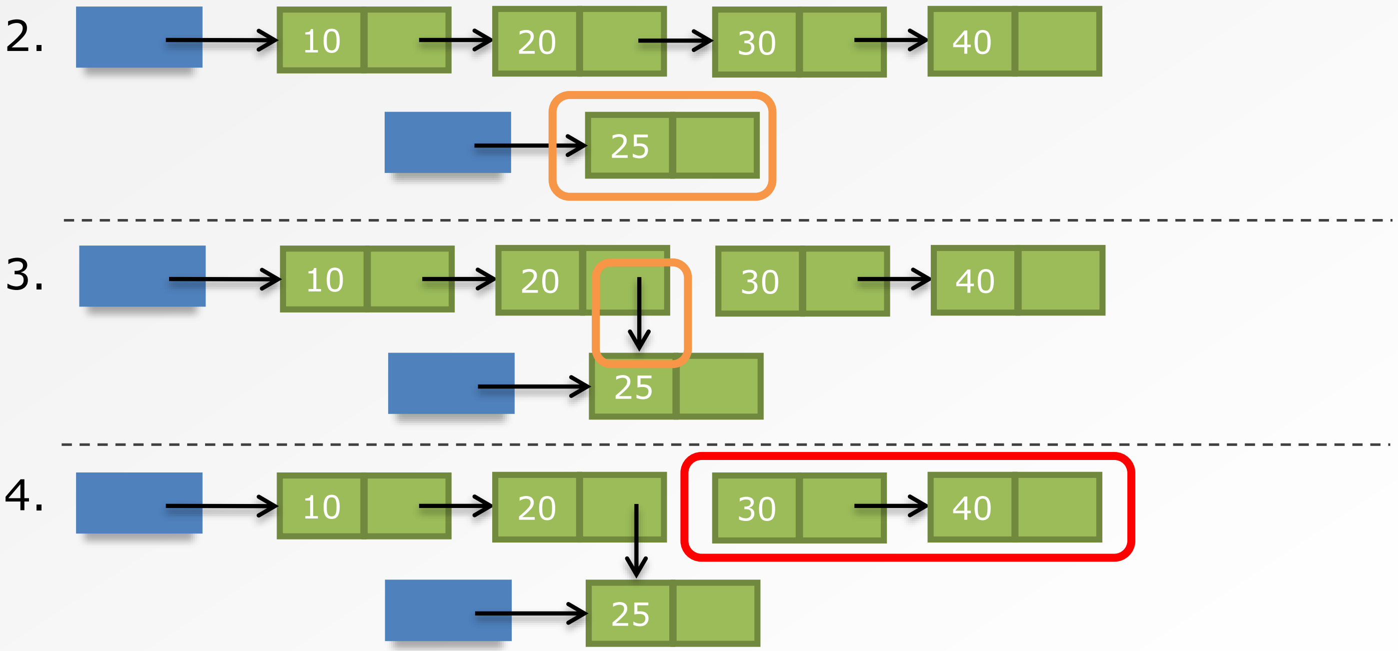
INSERT A NODE: insertNode()



Be careful with the order of pointer reconnections

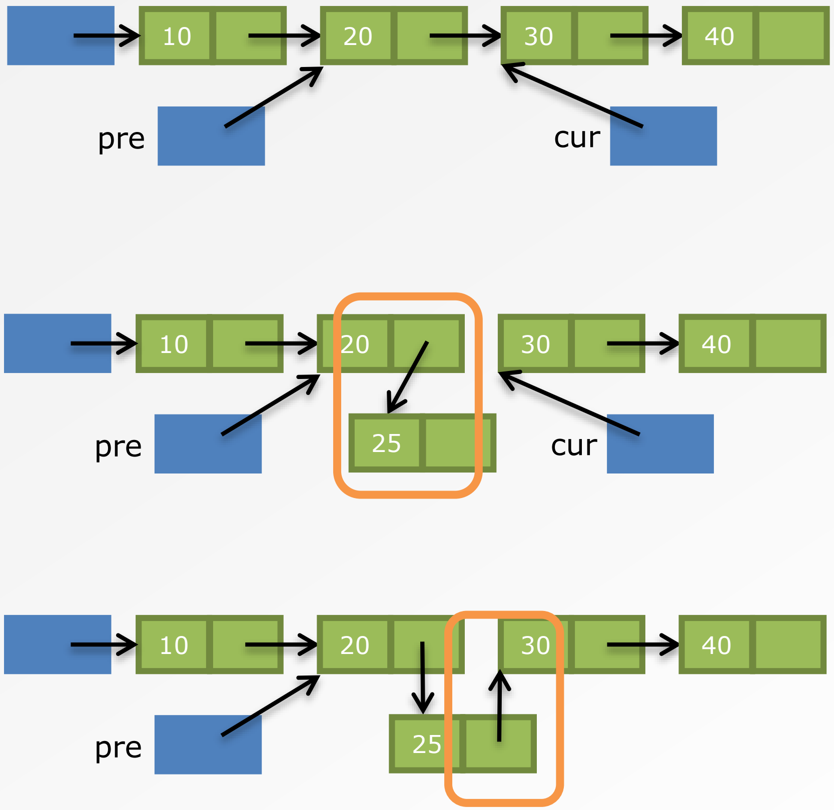
INSERT A NODE: insertNode()

- What if I first connect [20] to [25]?



All gone! Inaccessible in memory since we lost the address of [30]

INSERT A NODE: insertNode()



Slightly different idea:
Use two pointers (pre, cur) to keep track of the nodes before and after where the new node will go

1. Set pre, cur
   Remember findNode()?

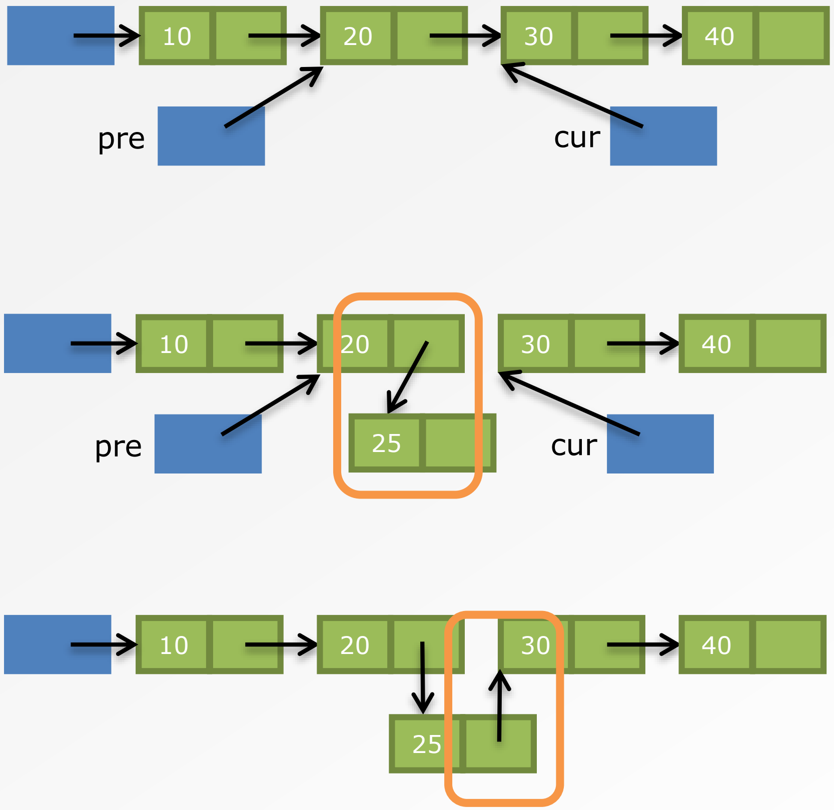
2. Create a new node and store its address in pre->next

```
Pre->next = malloc(sizeof(ListNode));
```

3. Set the new node's next pointer
   New node currently at pre->next
   Next pointer of new node is pre->next->next

```
Pre->next->next = cur
```

INSERT A NODE: insertNode()



Slightly different idea:
Use two pointers (pre, cur) to keep track of the nodes before and after where the new node will go

1. Set pre, cur
   Remember findNode()?

2. Create a new node and store its address in pre->next

```
Pre->next = malloc(sizeof(ListNode));
```

3. Set the new node's next pointer
   New node currently at pre->next
   Next pointer of new node is pre->next->next

```
Pre->next->next = cur
```

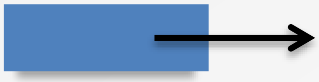
insertNode() [“NORMAL CASE” PART]

- Use findNode() to get address of the pre pointer
- If inserting a new node at index 2, pre should point to node at index 1
    - findNode(… , index-1)

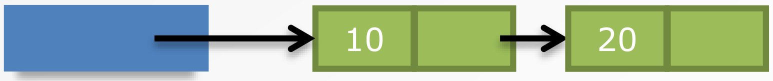
```
 // Find the nodes before and at the target position
    // Create a new node and reconnect the links
    if ((pre = findNode(*ptrHead, index-1)) != NULL){
        cur = pre->next;
        pre->next = malloc(sizeof(ListNode));
        pre->next->item = value;
        pre->next->next = cur;
        return 0;
    }

    return -1;
}
```

INSERT A NODE: insertNode()

- Now deal with special cases
  - Empty list



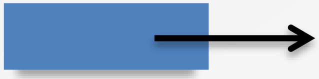
  - Inserting a node at index 0



- What is common to both special cases?

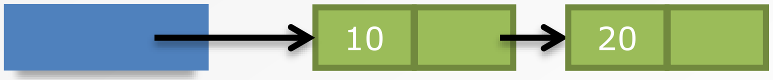
INSERT A NODE: insertNode()

- What is common to both special cases?
  - Empty list



```
head = malloc(sizeof(ListNode))
```

  - Inserting a node at index 0



```
// Save address of the first
node
head = malloc(sizeof(ListNode))
head->next = [addr of first
node]
```

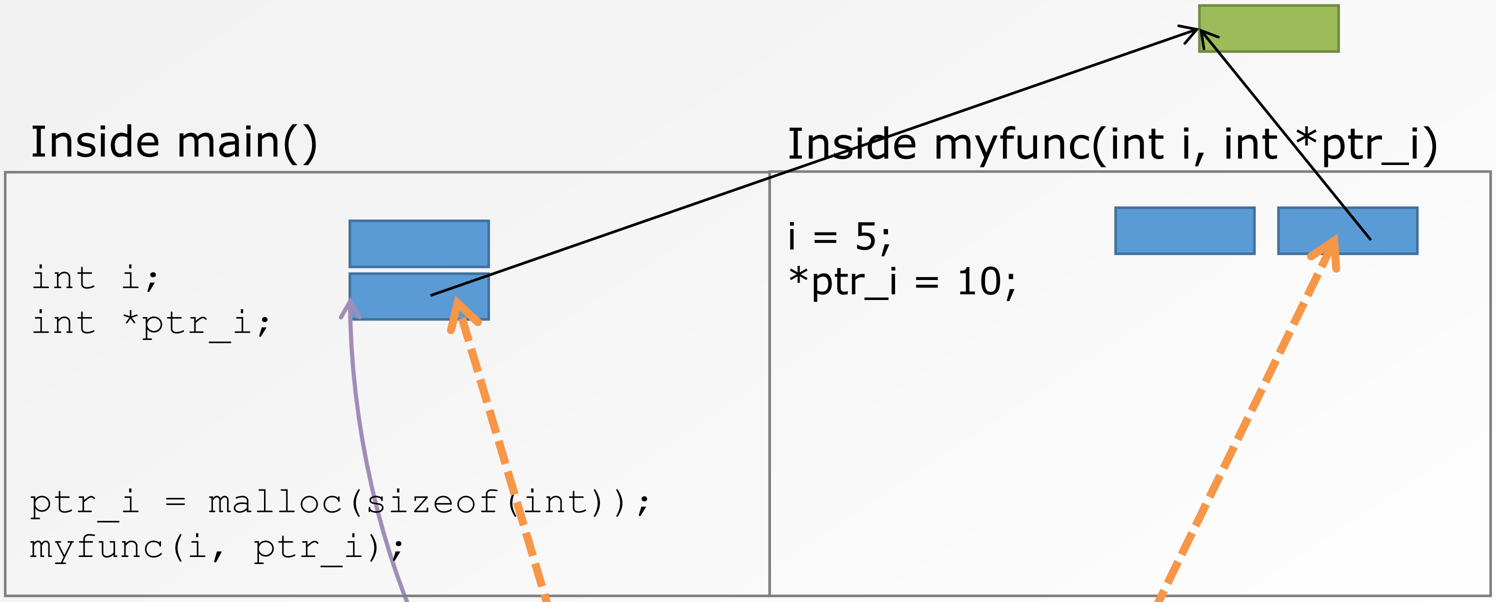
INSERT A NODE: insertNode()

- This does not work!

```
int insertNode(ListNode *head,  …  )
```

- If you are inserting a node into an empty list OR inserting a node at index 0 into an existing list
  - You need to change the address stored in the head pointer
- But you can only change the local copy of head pointer inside the insertNode() function
- Actual head pointer outside insertNode() remains unchanged!
- What is the solution when we want to modify a variable from inside a function?

REVISION: POINTERS AND PARAMETER PASSING



Inside main()

```
int i;
int *ptr_i;


ptr_i = malloc(sizeof(int));
myfunc(i, ptr_i);
```

Inside myfunc(int i, int *ptr_i)

```
i = 5;
*ptr_i = 10;
```

Pass in a pointer: You can change the value at the address store
BUT you cannot change the address stored in the pointer

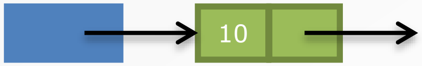
To change the address you must pass in the ADDRESS of the pointer

This is also why we can use the local head pointer as a temporary pointer without destroying the head pointer back in the main() function

INSERT A NODE: insertNode()

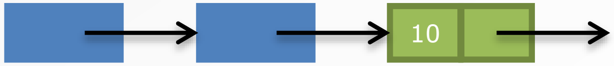
- Pass in a pointer!
- Pointer to the variable we want to change
- The variable to be changed is the head pointer

```
ListNode *head
```



- We need to pass in a pointer to the head pointer

```
ListNode **head
```



- To make things clearer, we will rename this as

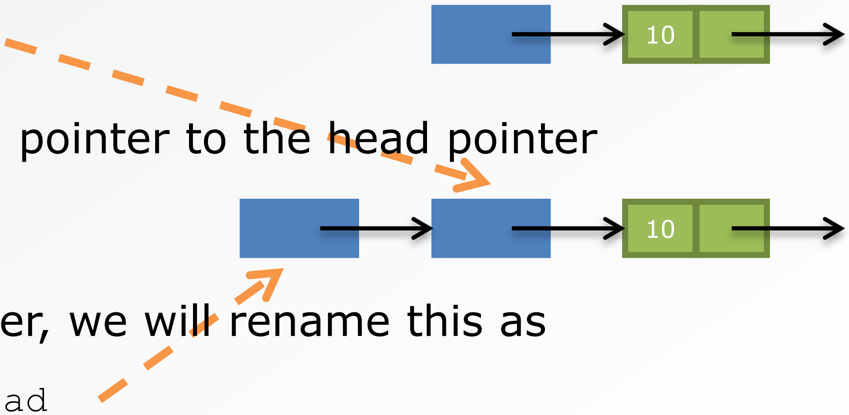
```
ListNode **ptrHead
```

  - Just to remind us that this is a pointer to the head pointer

INSERT A NODE: insertNode()

- Pass in a pointer!
- Pointer to the variable we want to change
- The variable to be changed is the head pointer

```
ListNode *head
```



- We need to pass in a pointer to the head pointer

```
ListNode **head
```

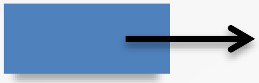
- To make things clearer, we will rename this as

```
ListNode **ptrHead
```

  - Just to remind us that this is a pointer to the head pointer
- **This lets us change the address that the head pointer points to**

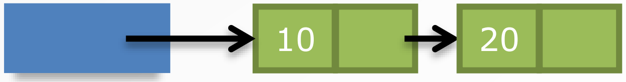
INSERT A NODE: insertNode()

- Can we combine any special cases?
  - Empty list



```
head = malloc(sizeof(ListNode));
head->next = null;
```

  - Inserting a node at index 0

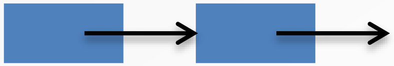


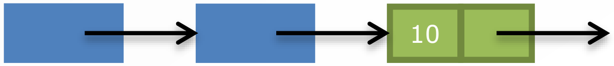
```
cur = head;
head = malloc(sizeof(ListNode))
head->next = cur;
```

- Yes! In an empty list, head = NULL

insertNode()

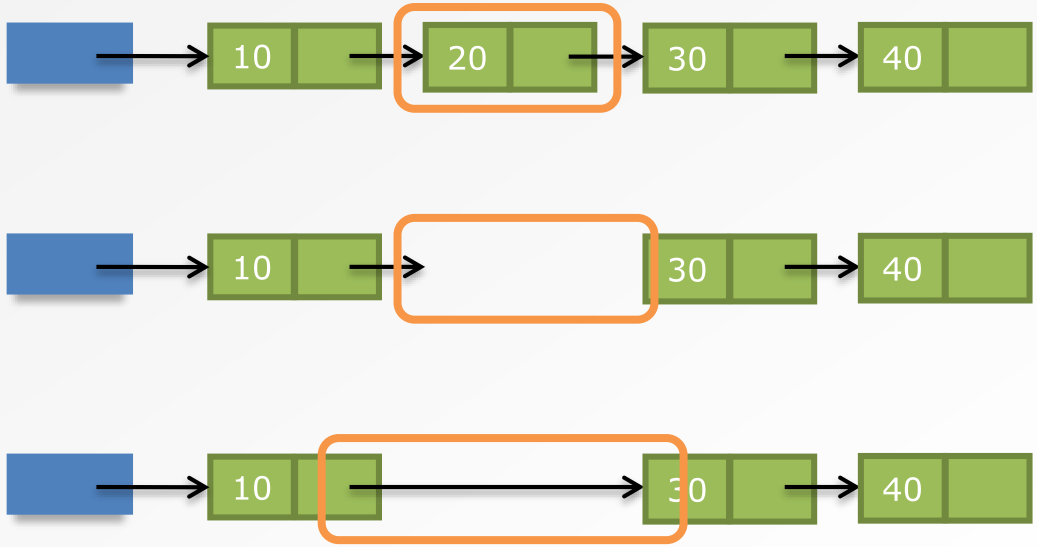
```c
int insertNode(ListNode **ptrHead, int index, int value){

    ListNode *pre, *cur;

    // If empty list or inserting first node, need to update head pointer
    if (*ptrHead == NULL || index == 0){
        cur = *ptrHead;
        *ptrHead = malloc(sizeof(ListNode));
        (*ptrHead)->item = value;
        (*ptrHead)->next = cur;
        return 0;
    }

    // Find the nodes before and at the target position
    // Create a new node and reconnect the links
    if ((pre = findNode(*ptrHead, index-1)) != NULL){
        cur = pre->next;
        pre->next = malloc(sizeof(ListNode));
        pre->next->item = value;
        pre->next->next = cur;
        return 0;
    }

    return -1;
}
```





REMOVE A NODE: removeNode()

- Remember to free up any unused memory



PREVIOUSLY

- Core linked list data structure functions
  - printList();
  - findNode();
  - insertNode()
  - removeNode()
- Recall prototypes for insertNode() and removeNode()
  - Need to be able to modify the address stored in the head pointer
  - Pass a pointer to the head pointer into functions

```
int insertNode(ListNode **ptrHead,int index, int value);
int removeNode(ListNode **ptrHead, int index);
```

sizeList() FUNCTION

- One more function
  - Return the number of nodes in a linked list

```
int sizeList(ListNode *head);
```

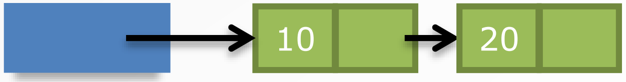
- Use the head pointer to get to the first node
- Keep following the next pointer until next == NULL
  - Increment counter
- Return the counter

sizeList() [VERSION 1]

- Should be quite easy to understand what’s happening here

```
int sizeList(ListNode *head){

        int count = 0;

        if (head == NULL){
                return 0;
        }

        while (head != NULL){
                head = head->next;
                count++;
        }

   return count;
}
```

```
typedef struct _listnode{
  int item;;
  struct _listnode *next;
}LinkedList;
```



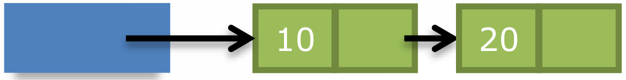
WORKED EXAMPLE: LINKED LIST APPLICATION

- Use the sizeList(), insertNode() and printList() functions
- Generate a list of 10 numbers by inserting random numbers (0-99) to the beginning of the list until it has 10 nodes

WORKED EXAMPLE: LINKED LIST APPLICATION

```
int main(){

    ListNode *head = NULL;

    srand(time(NULL));
    while (sizeList(head) < 10){
        insertNode(&head, 0, rand() % 100);
        printf("List: ");
        printList(head);
        printf("\n");
    }
    printf("%d nodes\n", sizeList(head));

    while (sizeList(head) > 0){
        removeNode(&head, sizeList(head)-1);
        printf("List: ");
        printList(head);
        printf("\n");
    }
    printf("%d nodes\n", sizeList(head));

    return 0;
}
```

```
typedef struct _listnode{
  int item;;
  struct _listnode *next;
}LinkedList;
```



```
int insertNode(ListNode **ptrHead,int index, int value);
int removeNode(ListNode **ptrHead, int index);
```

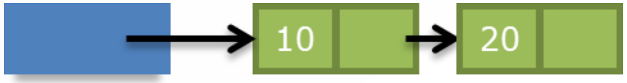
LINKED LIST APPLICATION

- How many times does sizeList() get called?
- Whole list has to be traversed every time

LINKED LIST APPLICATION

```
int main(){

    ListNode *head = NULL;

    srand(time(NULL));
    while (sizeList(head) < 10){
        insertNode(&head, 0, rand() % 100);
        printf("List: ");
        printList(head);
        printf("\n");
    }
    printf("%d nodes\n", sizeList(head));

    while (sizeList(head) > 0){
        removeNode(&head, sizeList(head)-1);
        printf("List: ");
        printList(head);
        printf("\n");
    }
    printf("%d nodes\n", sizeList(head));

    return 0;
}
```

```
typedef struct _listnode{
    int item;;
    struct _listnode *next;
}LinkedList;
```



LINKED LIST APPLICATION

- Very inefficient!
- How often does the number of nodes change?
    - Only when you do the following
        - Add a node
        - Remove a node
    - So why recalculate every single time?
- Add a variable to store the number of nodes

```
ListNode *head;
int listsize;
```

- Update the size variable whenever we add or remove a node

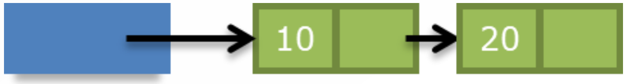
LINKED LIST APPLICATION [VERSION 2]

- Now sizeList() is redundant AND we have to manually manage the count of nodes in the list
- Still not a complete solution to our problems

LINKED LIST APPLICATION [VERSION 2]

```
int main(){
    ListNode *head = NULL;
    int listsize = 0;
    srand(time(NULL));
    while (listsize < 10){
        insertNode(&head, 0, rand() % 100);
        listsize++;
        printf("List: ");
        printList(head);
        printf("\n");
    }
    printf("%d nodes\n", listsize);

    while (size > 0){
        removeNode(&head, listsize-1);
        listsize--;
        printf("List: ");
        printList(head);
        printf("\n");
    }
    printf("%d nodes\n", listsize);

    return 0;
}
```

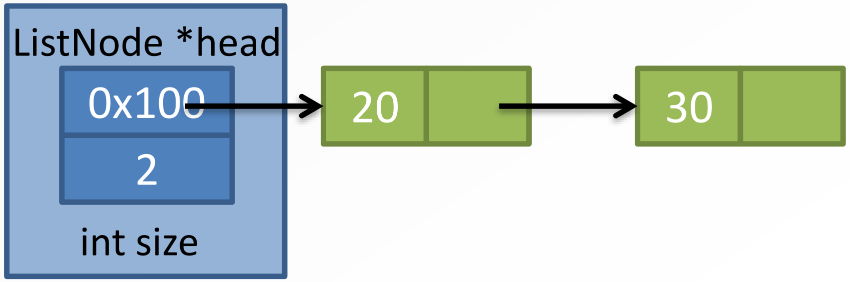
```
typedef struct _listnode{
  int item;;
  struct _listnode *next;
}LinkedList;
```



LINKEDLIST C STRUCT

- Implementation of Linked List
  - Define another C struct, LinkedList
  - Wrap up all elements that are required to implement the Linked List data structure

```
typedef struct _linkedlist{
    ListNode *head;
    int size;
} LinkedList;
```



- Why is this useful?
  - Consider the rewritten Linked List functions

LINKED LIST FUNCTIONS USING LINKEDLIST STRUCT

- Original function prototypes:
  - `void printList(ListNode *head);`
  - `ListNode * findNode(ListNode *head, int index);`
  - `int insertNode(ListNode **ptrHead, int index, int value);`
  - `int removeNode(ListNode **ptrHead, int index);`

- New function prototypes:
  - `void printList(LinkedList *ll);`
  - `ListNode * findNode(LinkedList *ll, int index);`
  - `int insertNode(LinkedList *ll, int index, int value);`
  - `int removeNode(LinkedList *ll, int index);`

CALLING NEW VERSION OF LINKED LIST FUNCTIONS

- Two versions of a small application

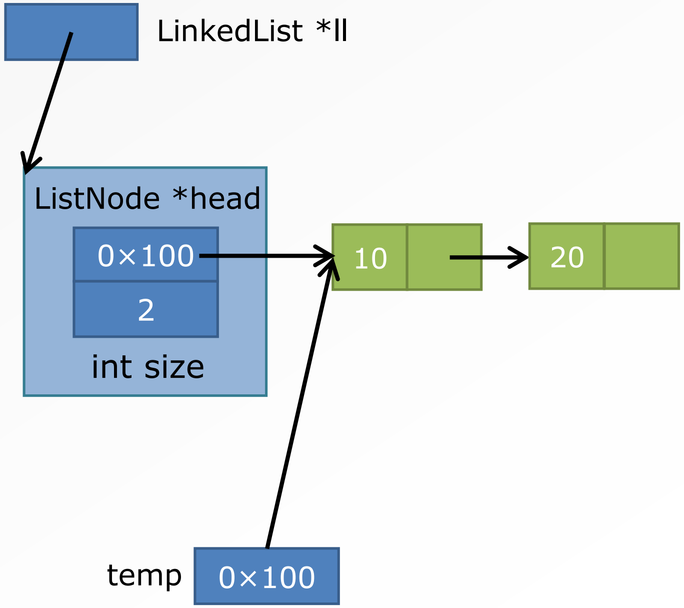
```
int main(){

    LinkedList ll;
    LinkedList *ptr_ll;

    insertNode(&ll, 0, 100);
    printList(&ll);
    printf(“%d nodes\n”, ll.size);
    removeNode(&ll, 0);

    ptr_ll = malloc(sizeof(LinkedList));
    insertNode(ptr_ll, 0, 100);
    printList(ptr_ll);
    printf(“%d nodes\n”, ptr_ll->size);
    removeNode(ptr_ll, 0);
}
```

```
int insertNode(LinkedList *ll, int index, int value);
int removeNode(LinkedList *ll, int index);
```

printList() USING LinkedList STRUCT

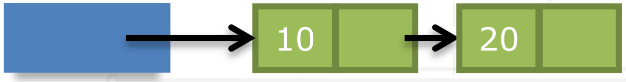
- Declare a temp pointer instead of using head (it is no longer a local variable; it is the actual head pointer)

```
void printList(LinkedList *ll){
    ListNode *temp = ll->head;

    if (temp == NULL)
        return;

    while (temp != NULL){
        printf("%d ", temp->item);
        temp = temp->next;
    }
    printf("\n");
}
```



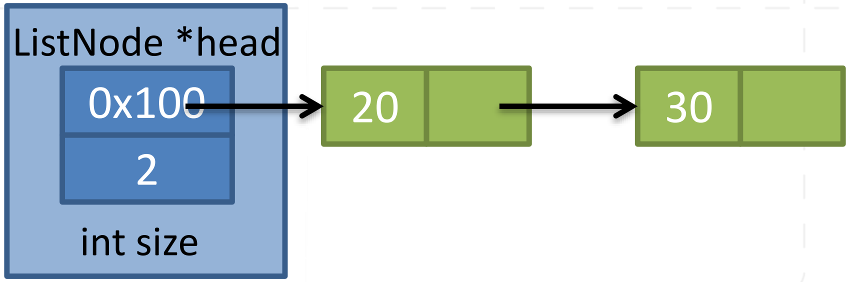
printList() Versions

```
typedef struct _listnode{
  int item;;
  struct _listnode *next;
}LinkedList;
```



```
void printList(ListNode *head){

    if (head == NULL)
        return;

    while (head != NULL){
        printf("%d ", head->item);
        head = head->next;
    }
    printf("\n");
}
```

```
typedef struct _linkedlist{
   int size;
   ListNode *head;
}LinkedList;
```



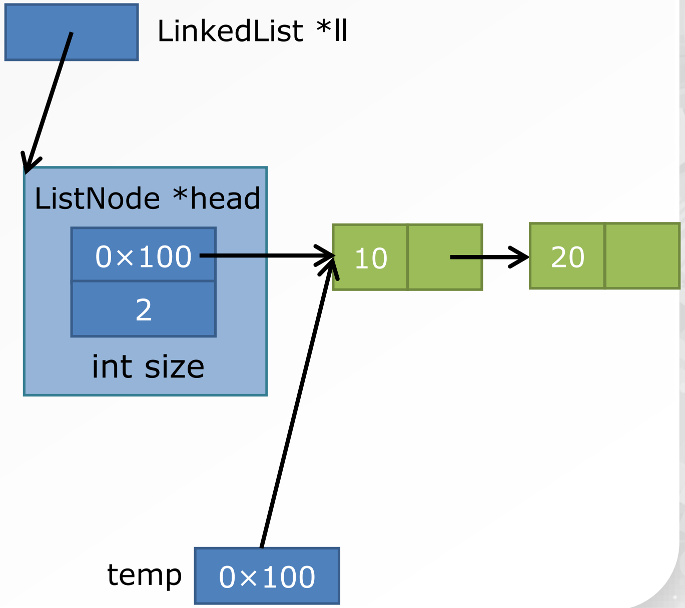
```
void printList(LinkedList *ll){
    ListNode *temp = ll->head;

    if (temp == NULL)
        return;

    while (temp != NULL){
        printf("%d ", temp->item);
        temp = temp->next;
    }
    printf("\n");
}
```

findNode() USING LinkedList STRUCT

- Again, declare a temp pointer to track the node we are looking at

- Also not much change/improvement in development time here

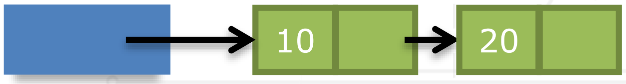
```
ListNode * findNode(
     LinkedList *ll, int index){
     ListNode *temp = ll->head;
     if (temp == NULL || index < 0)
          return NULL;

     while (index > 0){
          temp = temp->next;
          if (temp == NULL)
               return NULL;
          index--;
     }
     return temp;
}
```

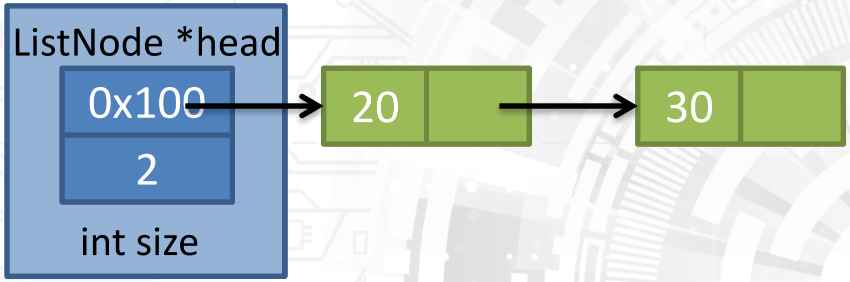


findNode() Versions

```
typedef struct _listnode{
  int item;;
  struct _listnode *next;
}LinkedList;
```

```
typedef struct _linkedlist{
   int size;
   ListNode *head;
}LinkedList;
```



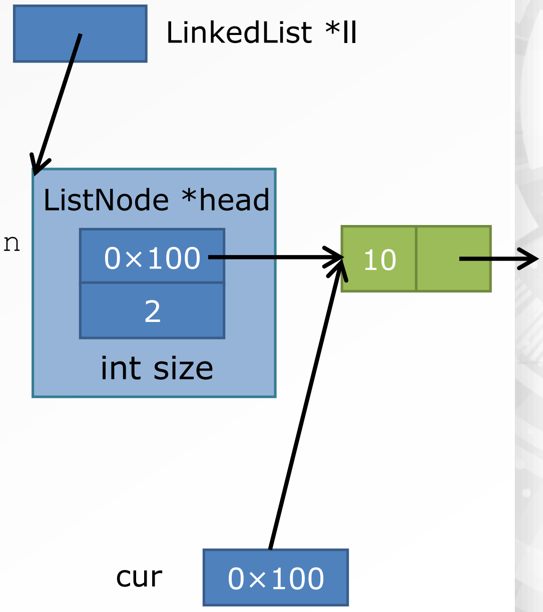


```
ListNode * findNode(
    ListNode *head, int index){

    if (head == NULL || index < 0)
        return NULL;

    while (index > 0){
        head = head->next;
        if (head == NULL)
            return NULL;
        index--;
    }
    return head;
}
```

```
ListNode * findNode(
    LinkedList *ll, int index){
    ListNode *temp = ll->head;
    if (temp == NULL || index < 0)
        return NULL;

    while (index > 0){
        temp = temp->next;
        if (temp == NULL)
            return NULL;
        index--;
    }
    return temp;
}
```

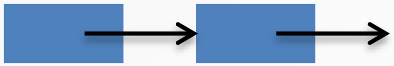
insertNode() LinkedList STRUCT

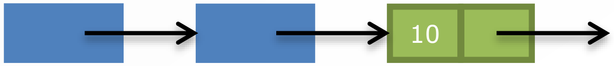
```c
int insertNode(LinkedList *ll, int index, int value){
    ListNode *pre, *cur;

    if (ll == NULL || index < 0 || index > ll->size + 1)
          return -1;
// If empty list or inserting first node, need to update head pointer
    if (ll->head == NULL || index == 0){
       cur = ll->head;
       ll->head = malloc(sizeof(ListNode));
       ll->head->item = value;
       ll->head->next = cur;
       ll->size++;
       return 0;
    }
// Find the nodes before and at the target position
// Create a new node and reconnect the links
    if ((pre = findNode(ll, index - 1)) != NULL){
       cur = pre->next;
       pre->next = malloc(sizeof(ListNode));
       pre->next->item = value;
       pre->next->next = cur;
       ll->size++;
       return 0;
    }
    return -1;
}
```



insertNode() Using ListNode STRUCT

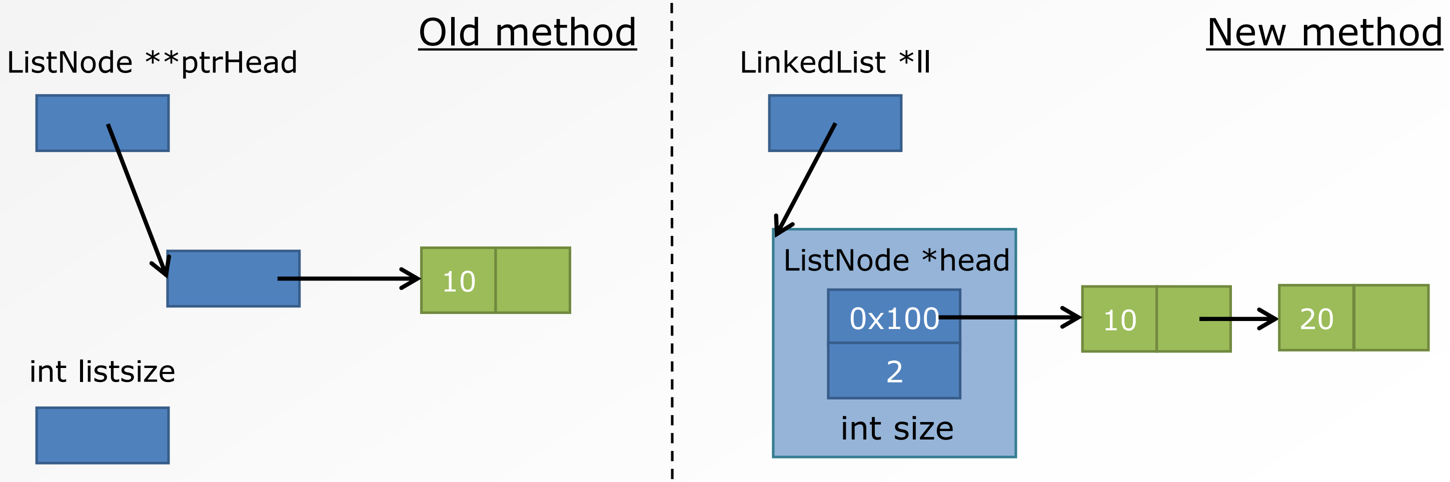
```
 int insertNode(ListNode **ptrHead, int index, int value){

    ListNode *pre, *cur;

    // If empty list or inserting first node, need to update head pointer
    if (*ptrHead == NULL || index == 0){
        cur = *ptrHead;
        *ptrHead = malloc(sizeof(ListNode));
        (*ptrHead)->item = value;
        (*ptrHead)->next = cur;
        return 0;
    }

    // Find the nodes before and at the target position
    // Create a new node and reconnect the links
    if ((pre = findNode(*ptrHead, index-1)) != NULL){
        cur = pre->next;
        pre->next = malloc(sizeof(ListNode));
        pre->next->item = value;
        pre->next->next = cur;
        return 0;
    }

    return -1;
}
```





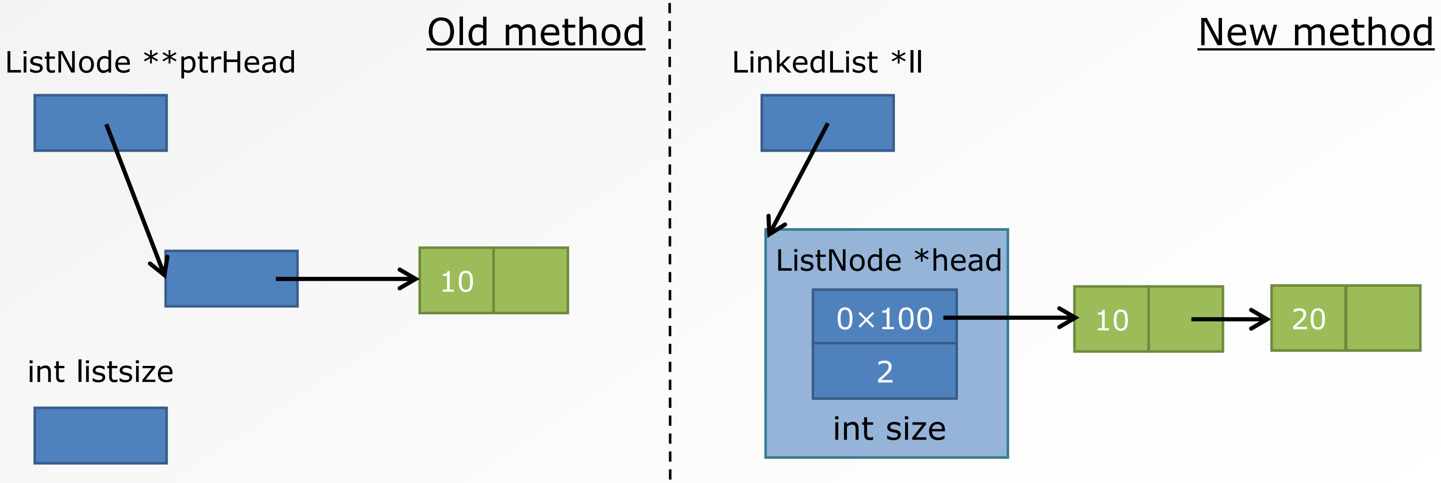
insertNode() USING LinkedList STRUCT

- Pass in pointer to LinkedList struct
- Function has full access to read and write address in head pointer
- Function can also update the number of nodes in the size variable; no need to pass in and listsize
- No need to think about double dereferencing



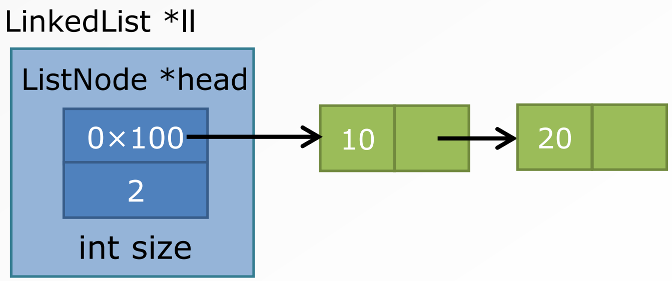
insertNode() USING LinkedList STRUCT

- Rewriting the removeNode() function is left as an exercise for you
- MUCH simpler than writing the original versions with pointer to head pointer



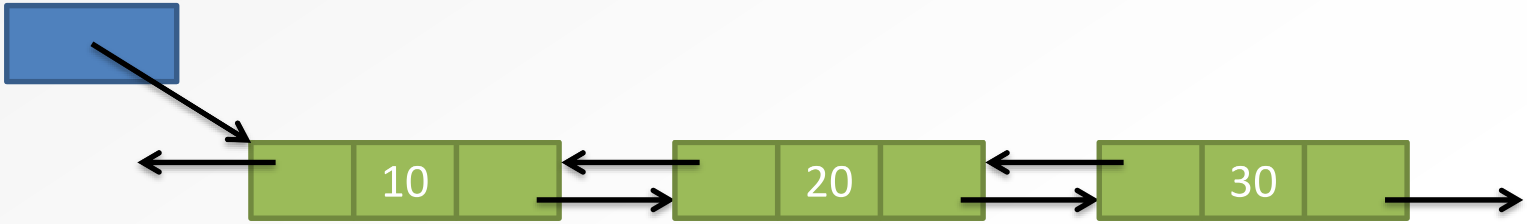
Linkedlist STRUCT

- Allows us to think of LinkedList as an object on its own
- Each LinkedList object has the following components
  - Head pointer that stores the address of the first node
  - Size variable that tracks the number of nodes in the linked list
- Conceptually much cleaner
- Practically much cleaner too
  - Easy to pass the entire LinkedList struct into a function



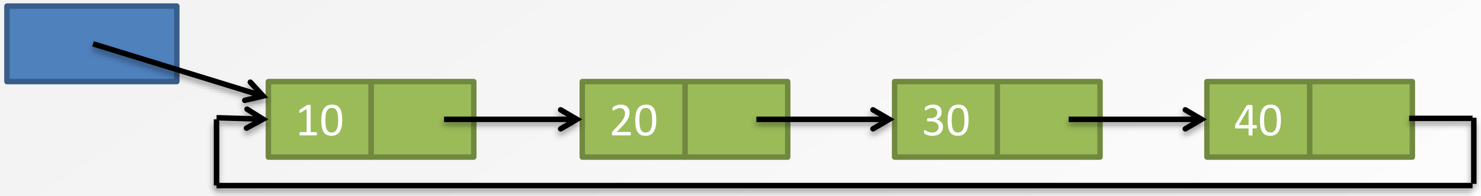
MORE COMPLEX LINKED LISTS - DOUBLY LINKED LIST

- So far, singly-linked list
  - Each ListNode is linked to at most one other ListNode
  - Traversal of the list is one-way only
    - Can’t go backwards
    - What if we want to start from a given node and search EITHER backwards OR forwards
- Doubly Linked List
  - Traversing a doubly linked list in forward direction
  - Traversing a doubly linked list in backward direction

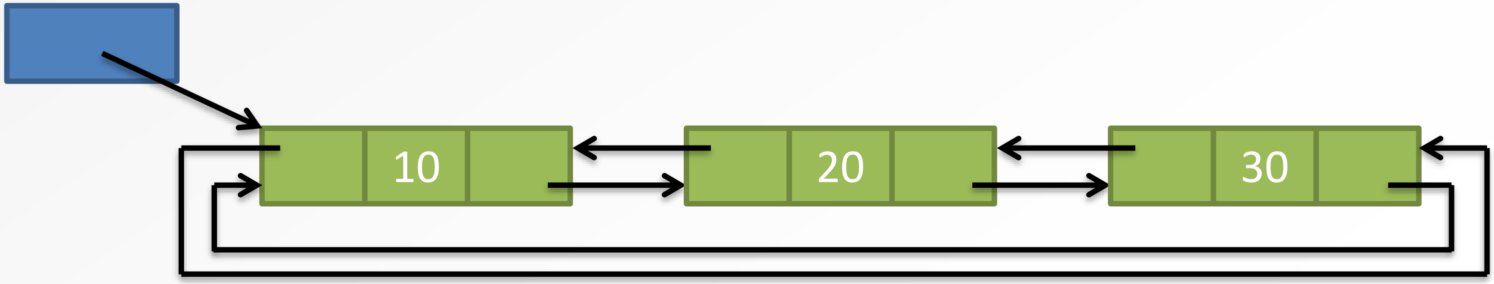


MORE COMPLEX LINKED LISTS - CIRCULAR LINKED LISTS

- Circular singly-linked lists
  - Last node has next pointer pointing to first node



- Circular doubly-linked lists
  - Last node has next pointer pointing to first node
  - First node has pre pointer pointing to last nod



ARRAYS VS. LINKED LISTS

- **Arrays**
  - Efficient random access
  - Difficult to expand, re-arrange
  - When inserting/removing items in the middle or at the front, computation time scales with size of list
  - Generally a better choice when data is immutable
- **Linked lists (dynamic-pointer-based and static-array-based)**
  - “Random access” can be implemented, but more inefficient than arrays
  - cost of storing links, only use internally.
  - Easy to shrink, rearrange and expand (but array-based linked list has a fixed size)
  - Insert/remove operations only require fixed number of operations regardless of list size. no shifting

COMMON MISTAKES

- Very important!
  - head is a node pointer
  - Points to the first node
  - head is not the “first node”
  - head is not the “head node”
- Forget to check whether the list is empty head=NULL
- Forget to deal with the first node differently.
- Forget to deal with the last node differently
- Forget to handle differently when: insert/remove a node at the beginning/tail of the list
- Changes of the links when insert/remove a node. The order matters!!