



**NANYANG
TECHNOLOGICAL
UNIVERSITY**

CE1007/CZ1007 DATA STRUCTURES

Lecture 03: Linked List Functions

Dr. Owen Noel Newton Fernando

College of Engineering

School of Computer Science and Engineering

- ListNode structures
- Core linked list data structure functions
 - printList();
 - findNode();
 - insertNode();
 - removeNode();
- Common Mistakes

HOW TO IMPLEMENT DATA STRUCTURE FUNCTIONS WITHOUT MEMORY LEAKS AND ILLEGAL ACCESS ERRORS

- Concept before code
 - Draw all the pictures, step by step
 - Write all the pseudocode (if necessary)
 - Code comes last
 - You should be able to use all the diagrams/pseudocode to implement a linked list in any language

- **ListNode structures**

- Core linked list data structure functions

- `printList();`
- `findNode();`
- `insertNode();`
- `removeNode();`

- Common Mistakes

RECALL: ListNode STRUCTURE

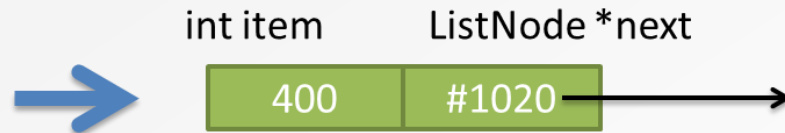
- Our default ListNode for the rest of the class will store an integer item

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

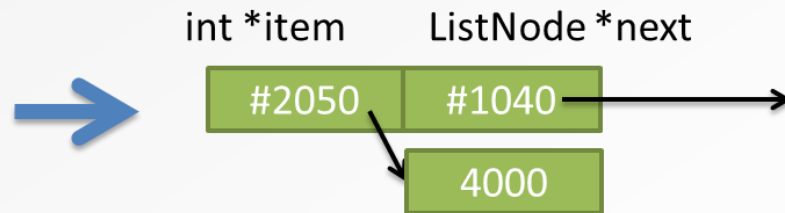
- ListNodes can store anything in the item field
 - int or int*
 - Array of integers
 - char or char*
 - Another struct or a pointer to a struct
 - Whatever you want
 - Can even define int item1, item2

ADVANCED ListNode STRUCTURES

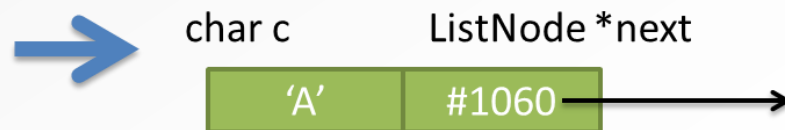
```
typedef struct _listnode{  
    int item;  
    struct _listnode *next;  
} ListNode;
```



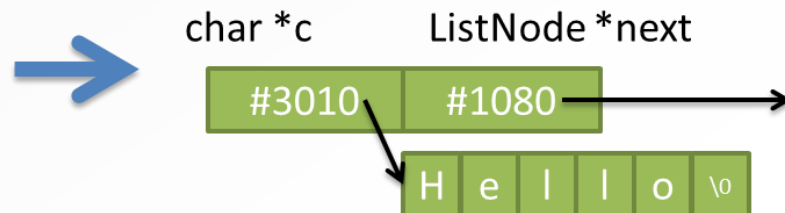
```
typedef struct _listnode{  
    int *item;  
    struct _listnode *next;  
} ListNode;
```



```
typedef struct _listnode{  
    char c;  
    struct _listnode *next;  
} ListNode;
```



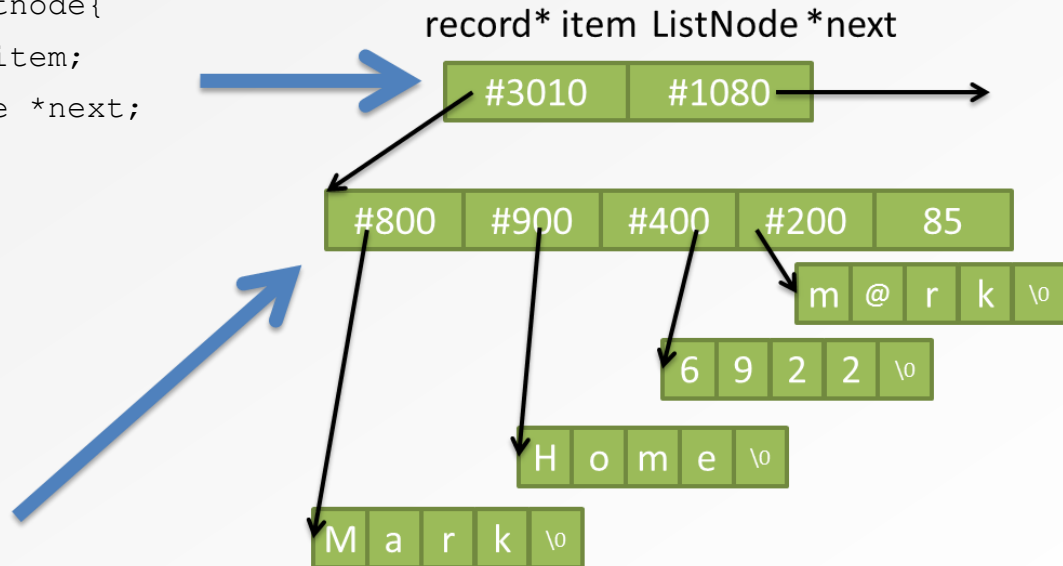
```
typedef struct _listnode{  
    char *c;  
    struct _listnode *next;  
} ListNode;
```



ADVANCED ListNode STRUCTURES

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

```
struct record{
    char *name;
    char *address;
    char *phone
    char *email
    int age;
}
```



- ListNode structures
- **Core linked list data structure functions**
 - printList();
 - findNode();
 - insertNode();
 - removeNode();
- Common Mistakes

SINGLY-LINKED LIST OF INTEGERS

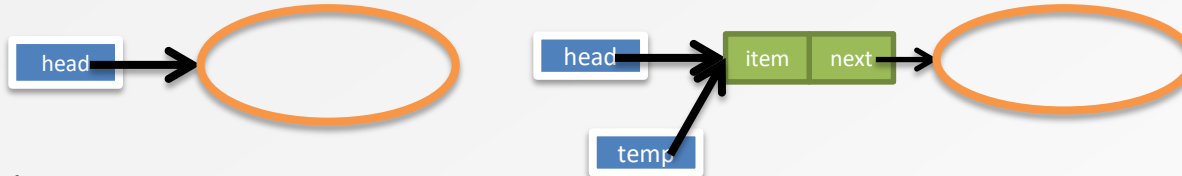
```
1  typedef struct node{
2      int item; struct node *next;
3  } ListNode;
4
5  int main(){
6      ListNode *head = NULL, *temp;
7      int i = 0;
8
9      scanf("%d", &i);
10     while (i != -1){
11         if (head == NULL){
12             head = malloc(sizeof(ListNode));
13             temp = head;
14         }
15         else{
16             temp->next = malloc(sizeof(ListNode));
17             temp = temp->next;
18         }
19         temp->item = i;
20         scanf("%d", &i);
21     }
22     temp->next = null;
23 }
```

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)

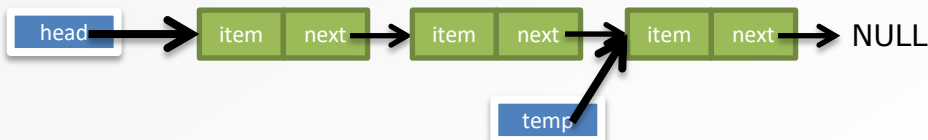
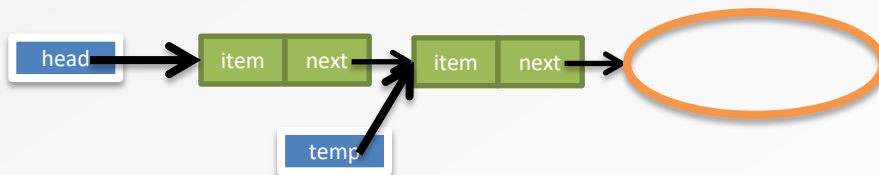
STORE A LIST OF NUMBERS



- $head == NULL$
- Need to update the *head* pointer

```
head = malloc(sizeof(ListNode));
```
- In this case, *temp* also points to the first ListNode

```
temp = head;
```



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

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

CORE FUNCTIONS FOR LINKED LIST

- To support some basic operations

- **Insert a node**

InsertNode()

At the front

At the back

In the middle

- **Remove a node**

RemoveNode()

At the front

At the back

In the middle

- **Print the whole list**

PrintList()

- **Look for the node at index i**

FindNode()

- Etc.

- ListNode structures
- Core linked list data structure functions
 - **printList();**
 - findNode();
 - insertNode();
 - removeNode();
- Common Mistakes

PrintList()

- Print all items in the linked list
 - 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

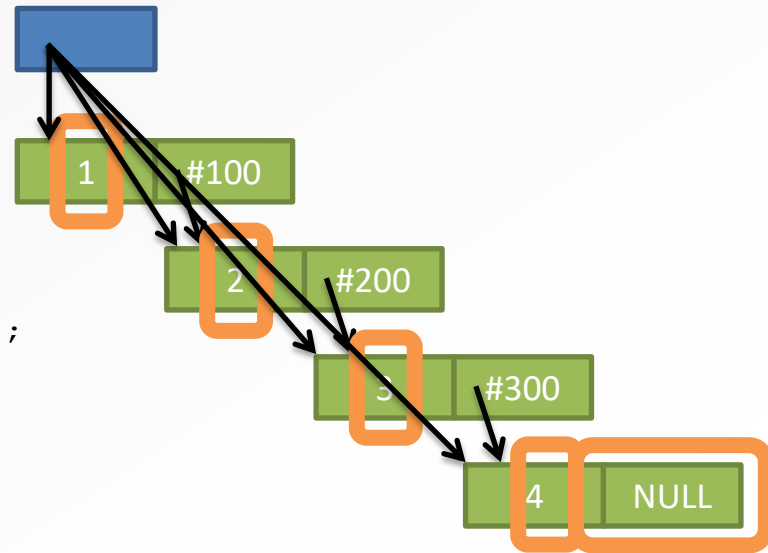
PRINT OUT ITEMS IN LINKED LIST: printList()

- 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

```
1 void printList(ListNode *head){
2
3     if (head == NULL)
4         return;
5
6     while (head != NULL){
7         printf("%d ", head->item);
8         head = head->next;
9     }
10    printf("\n");
11 }
```



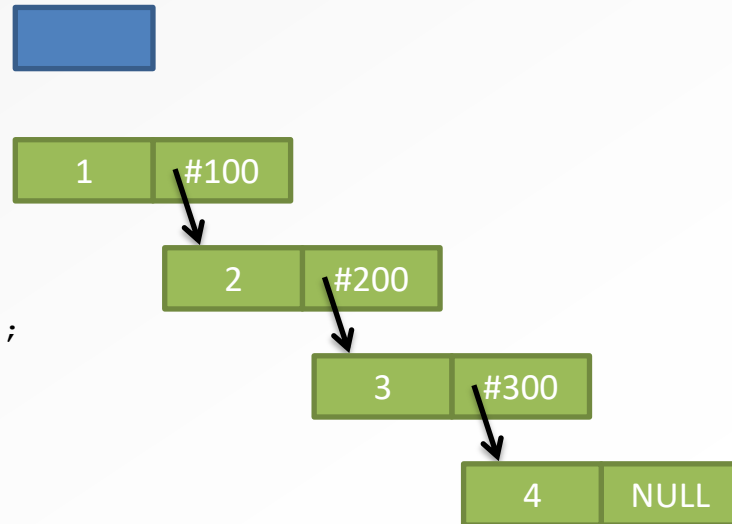
PRINT OUT ITEMS IN LINKED LIST: printList()

- 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

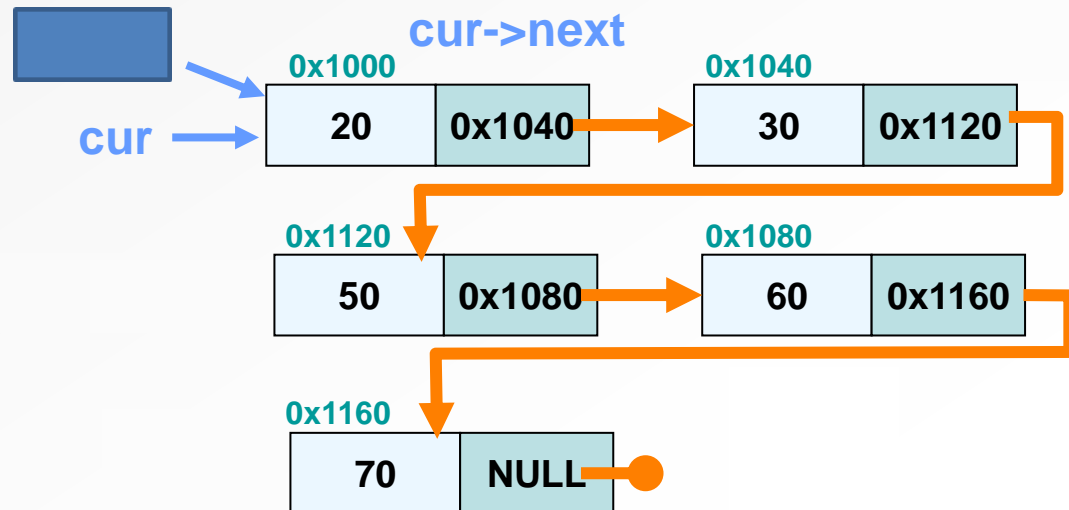
```
1 void printList(ListNode *head){
2
3     if (head == NULL)
4         return;
5
6     while (head != NULL){
7         printf("%d ", head->item);
8         head = head->next;
9     }
10    printf("\n");
11 }
```



PRINT OUT ITEMS IN LINKED LIST: printList()

```
1 void printList(ListNode *head){
2     ListNode *cur
3     cur=head;
4
5     if (cur== NULL)    return;
6
7     while (cur!= NULL){
8         printf("%d\n", cur ->item);
9         cur = cur ->next;
10    }
11    printf("\n");
12 }
```

20
30
50
60
70



- ListNode structures
- Core linked list data structure functions
 - printList();
 - **findNode();**
 - insertNode();
 - removeNode();
- Common Mistakes

findNode()

- **Get the pointer to node at index I**
- This function will come in useful later
- Pass head pointer into the function

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

Count down *index* times

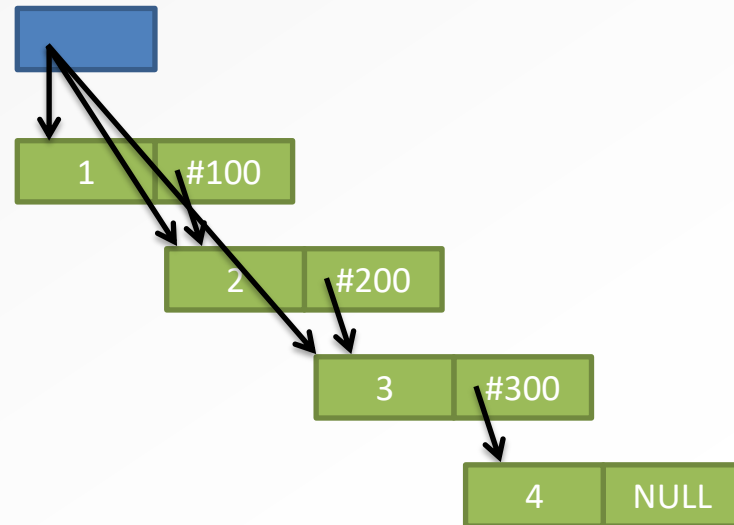
GET POINTER TO NODE AT INDEX I: findNode()

- 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

```
1  ListNode * findNode(  
2      ListNode *head, int index){  
3  
4      if (head == NULL || index < 0)  
5          return NULL;  
6  
7      while (index > 0){  
8          head = head->next;  
9          if (head == NULL)  
10             return NULL;  
11         index--;  
12     }  
13     return head;  
14 }
```



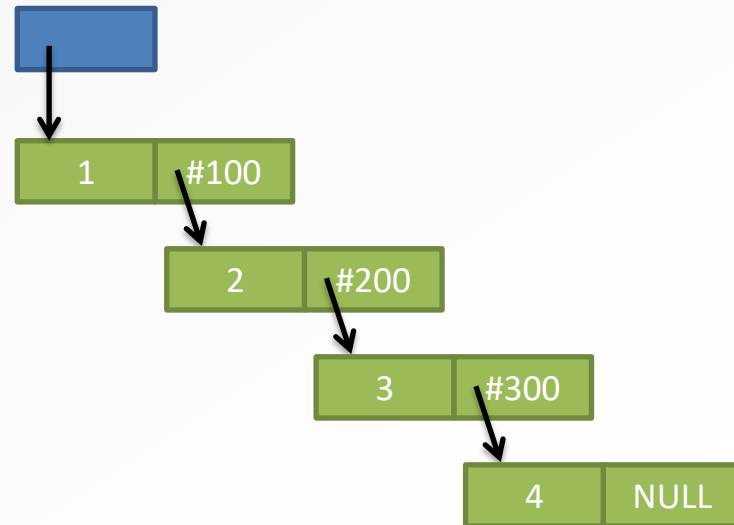
GET POINTER TO NODE AT INDEX I: findNode()

- 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

```
1  ListNode * findNode(  
2      ListNode *head, int index){  
3  
4      if (head == NULL || index < 0)  
5          return NULL;  
6  
7      while (index > 0){  
8          head = head->next;  
9          if (head == NULL)  
10             return NULL;  
11             index--;  
12     }  
13     return head;  
14 }
```

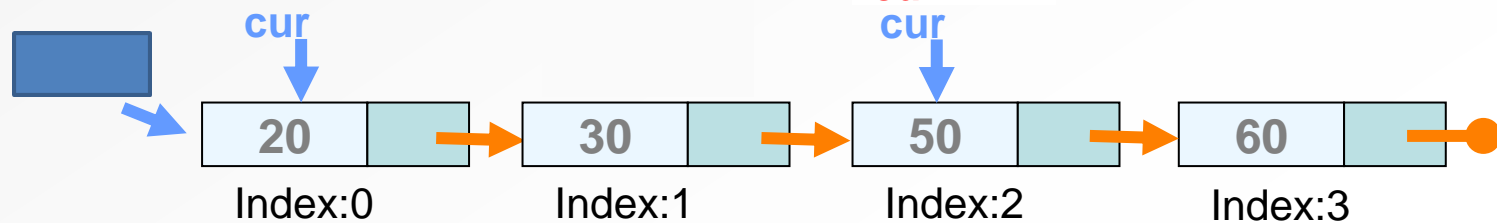


GET POINTER TO NODE AT INDEX I: findNode()

```
1  ListNode *findNode(ListNode*head, int index)
2  {  ListNode *cur; int index
3      cur=head;
4      if (cur==NULL || index<0) return NULL;
5      while(index>0){
6          cur=cur->next;
7          if (cur==NULL)
8              return NULL;
9          index--;
10     }
11     return cur;
12 }
```

When the list is empty
or the index i is illegal

When the list is shorter
than the index



- ListNode structures
- Core linked list data structure functions
 - printList();
 - findNode();
 - **insertNode();**
 - removeNode();
- Common Mistakes

INSERT A NODE: insertNode()

- Add a node anywhere in the linked list
- Let's work through the process of adding a node
- Have to consider various special cases
- Pass in the head pointer
- What is the correct parameter list?

`int insertNode(.....)`

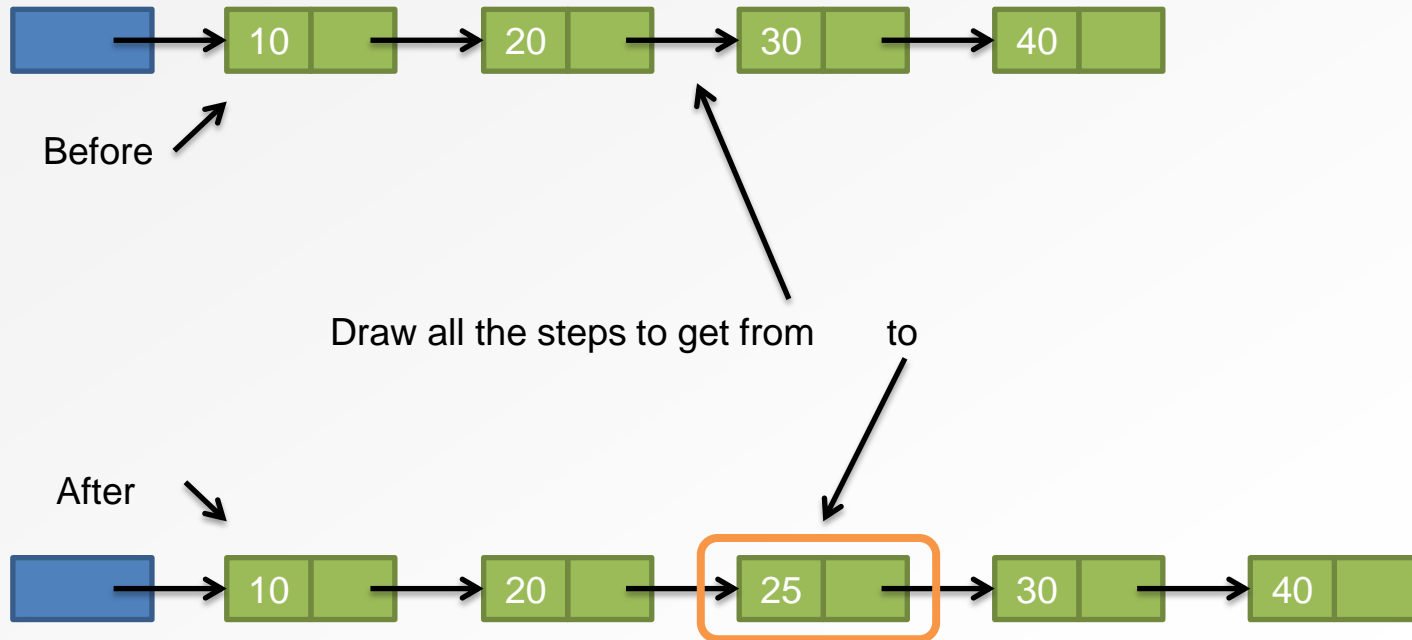
- KIV – this will become obvious later
- There is an apparently correct but actually wrong answer

INSERT A NODE: insertNode()

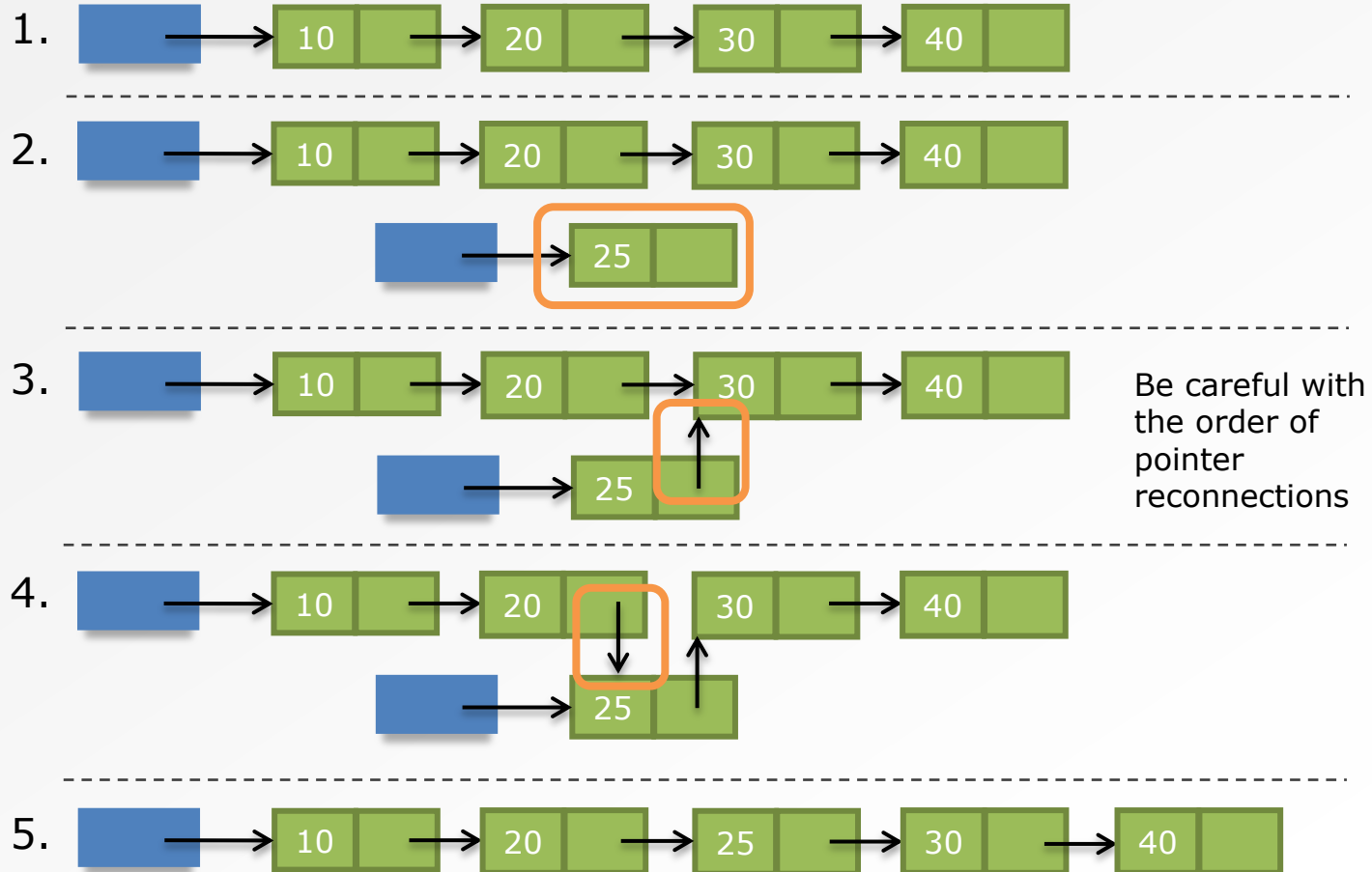
- Consider all **the different places** we want to add a node
 - **Front**
 - **Back**
 - **Middle**
- Consider all the **different starting states** of the linked list
 - **Empty list**
 - **One node**
 - **Many nodes**
- Ok to create many special cases and merge them later when we see similar code
- Start with the case of adding a node in the middle of a linked list with many existing nodes
 - Several pointers to move around

INSERT A NODE: insertNode()

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

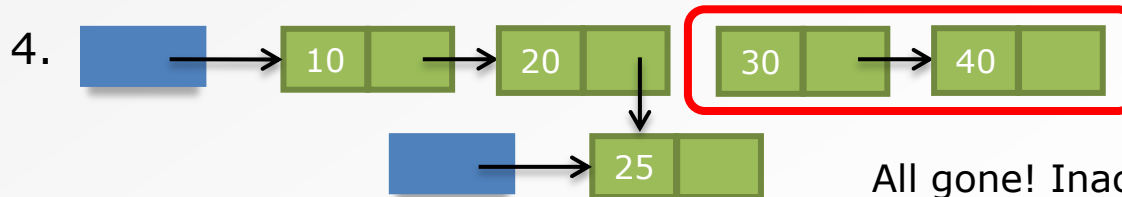
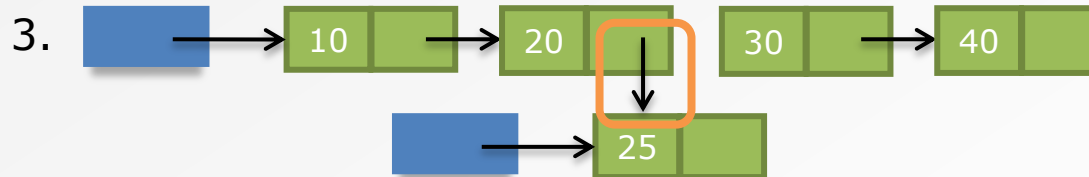
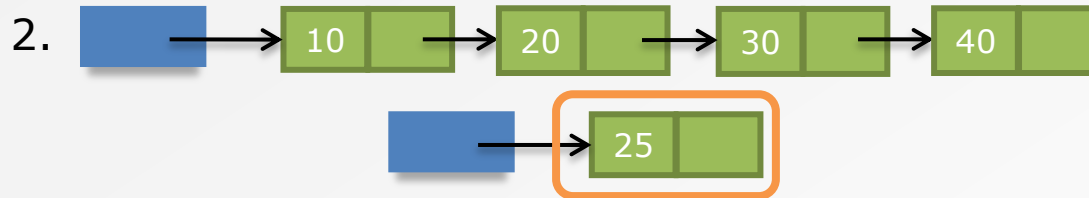


INSERT A NODE: insertNode()



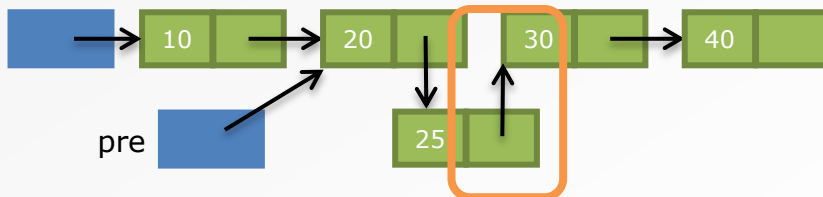
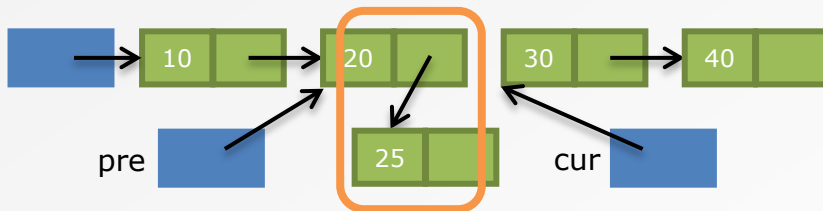
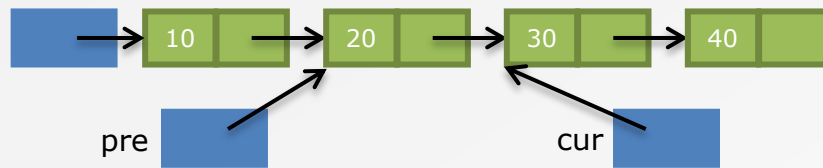
INSERT A NODE: insertNode()

- 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

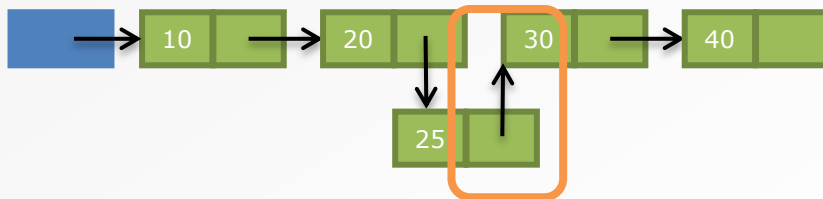
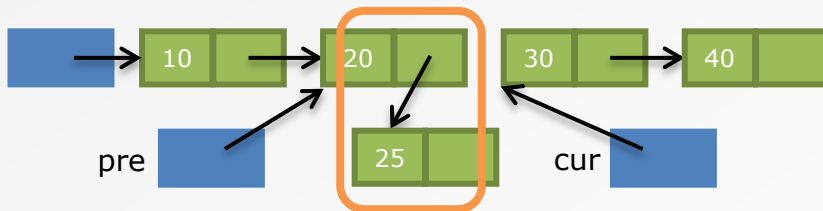
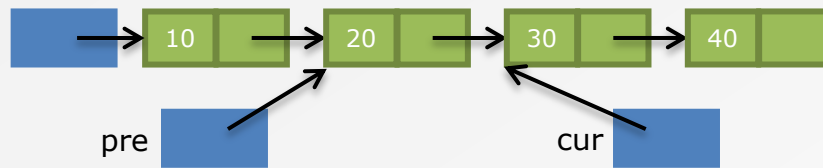
1. Set pre, cur
Remember findNode()?
2. Create a new node and store its address in pre->next

```
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))
```

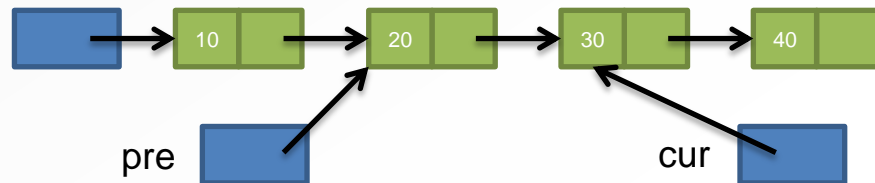
3. Set the new node's next pointer
New node currently at pre->next
Next pointer of new node is
pre->next->next

```
pre->next->next = cur
```

insertNode() ["NORMAL CASE"]

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

```
14     // Find the nodes before and at the target position
15     // Create a new node and reconnect the links
16     if ((pre = findNode(*ptrHead, index-1)) != NULL){
17         cur = pre->next;
18         pre->next = malloc(sizeof(ListNode));
19         pre->next->item = value;
20         pre->next->next = cur;
21         return 0;
22     }
23
24     return -1;
25 }
```



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 = [address of first node]
```


INSERT A NODE: insertNode()

- Answer:
 - The address stored in the head pointer must be changed
- Back to the actual `insertNode()` code
- Earlier question:
 - What is the parameter list?

- Does this work?

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

- Hint:
 - Can you change the address stored in the actual head pointer from inside the `insertNode()` function?

REVISION: POINTERS AND PARAMETER PASSING

main()

```
int n;  
ListNode *head=NULL;  
  
...  
insertNode(head,30);
```

insertNode(ListNode *head, int value)

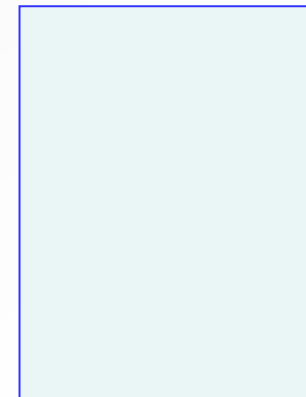
```
if (head==NULL) {  
    head=malloc(sizeof(ListNode));  
    head->item=value;  
    head->next=NULL;  
}  
...
```

head(main) →

n(main) →



stack



heap

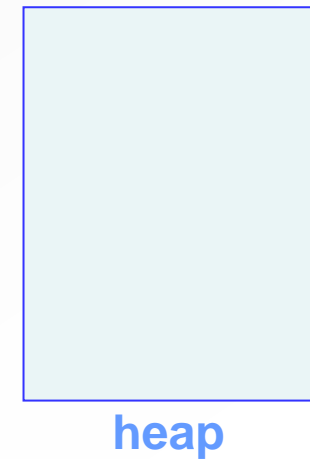
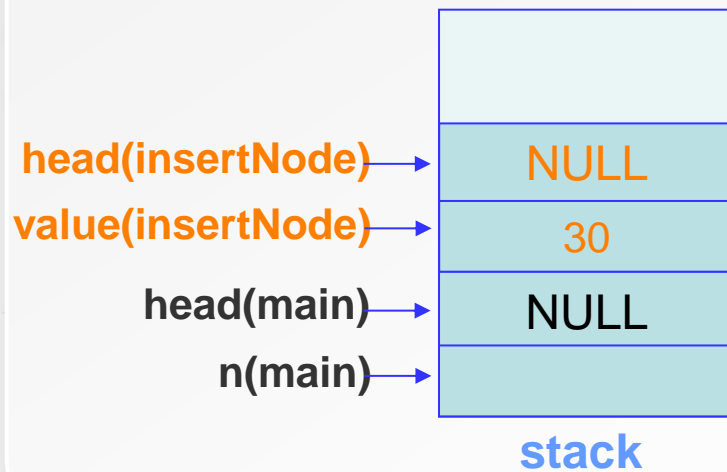
REVISION: POINTERS AND PARAMETER PASSING

main()

```
int n;  
ListNode *head=NULL;  
  
...  
insertNode(head,30);
```

insertNode(ListNode *head, int value)

```
if (head==NULL) {  
    head=malloc(sizeof(ListNode));  
    head->item=value;  
    head->next=NULL;  
}  
...
```



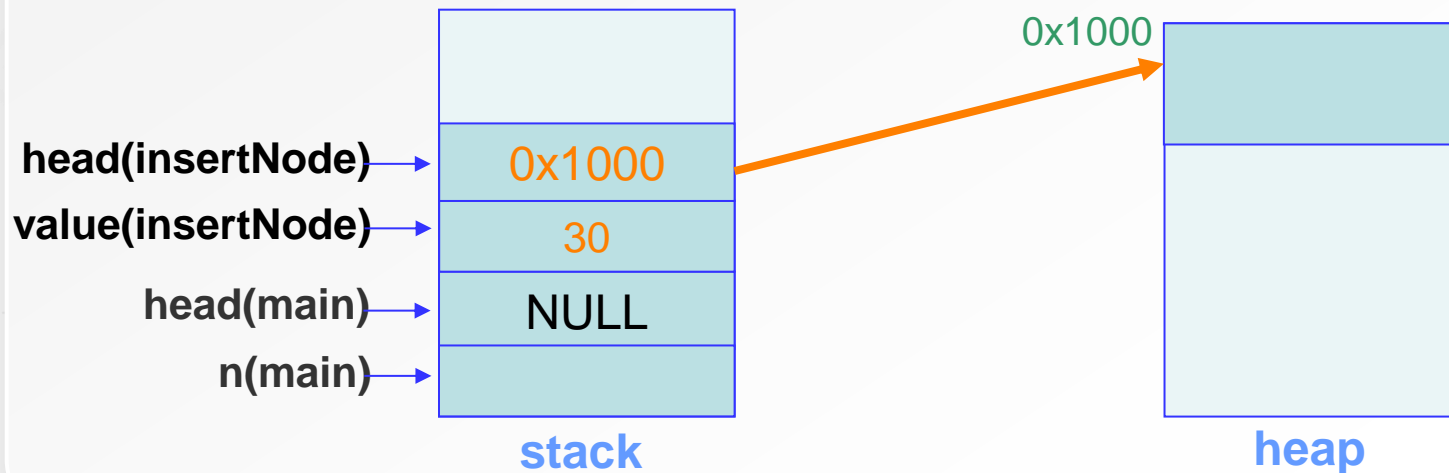
REVISION: POINTERS AND PARAMETER PASSING

main()

```
int n;  
ListNode *head=NULL;  
  
...  
insertNode(head,30);
```

insertNode(ListNode *head, int value)

```
if (head==NULL) {  
    head=malloc(sizeof(ListNode));  
    head->item=value;  
    head->next=NULL;  
}  
...
```



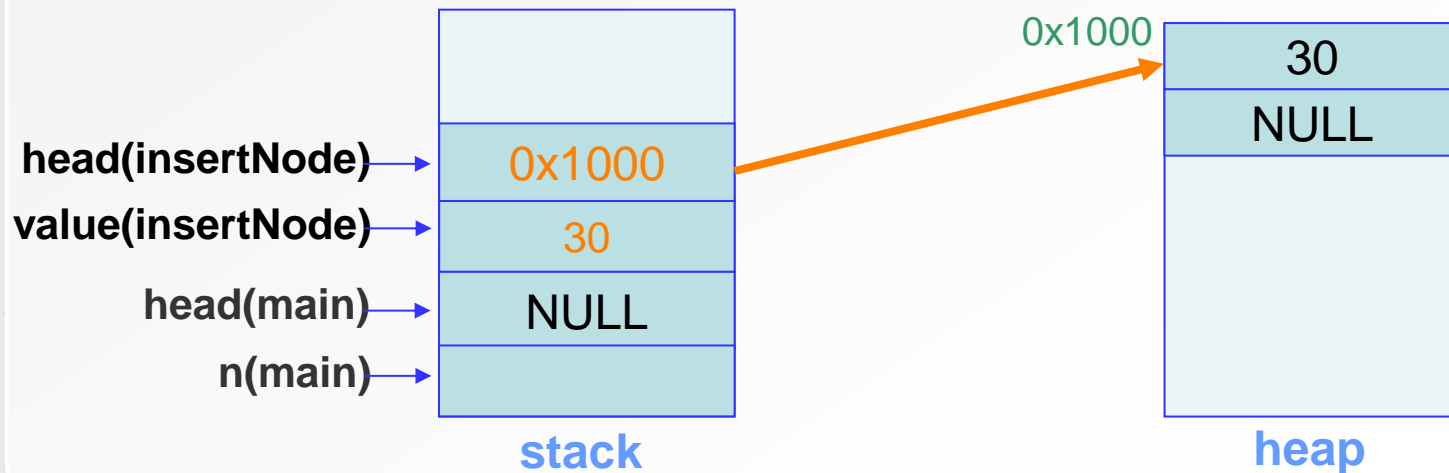
REVISION: POINTERS AND PARAMETER PASSING

main()

```
int n;  
ListNode *head=NULL;  
  
...  
insertNode(head,30);
```

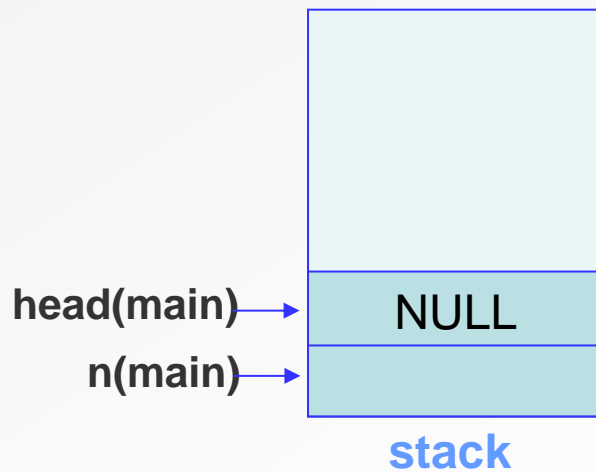
insertNode(ListNode *head, int value)

```
if (head==NULL) {  
    head=malloc(sizeof(ListNode));  
    head->item=value;  
    head->next=NULL;  
}  
...
```



REVISION: POINTERS AND PARAMETER PASSING

```
main()  
  
int n;  
ListNode *head=NULL;  
  
...  
insertNode(head,30);
```



INSERT A NODE: `insertNode()`

- This does not work!

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

- If you are inserting a node into an empty list **OR** you are 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 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?

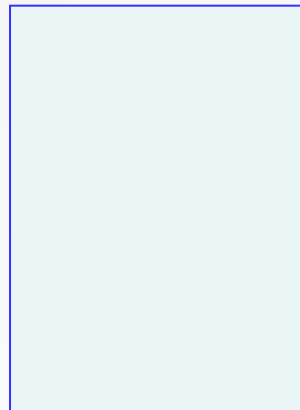
POINTERS AND PARAMETER PASSING

`main()`

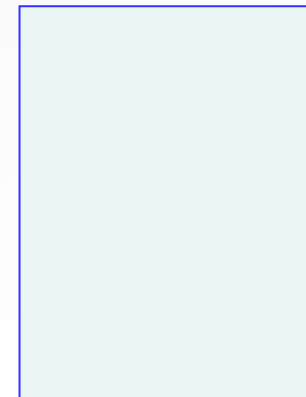
```
int n;  
ListNode **ptrHead,  
        *head=NULL;  
...  
ptrHead=&head;  
insertNode(ptrHead,30);
```

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

```
if (*ptrHead==NULL) {  
    *ptrHead=malloc(sizeof(ListNode));  
    (*ptrHead)->item=value;  
    (*ptrHead)->next=NULL;  
}  
...
```



stack

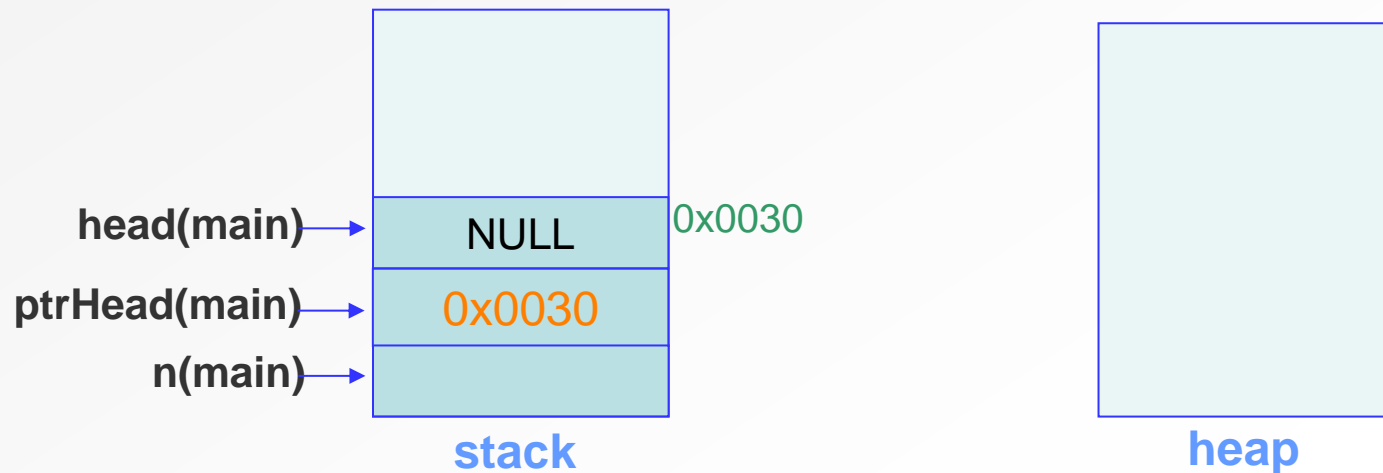


heap

POINTERS AND PARAMETER PASSING

```
main()
{
    int n;
    ListNode **ptrHead,
           *head=NULL;
    ...
    ptrHead=&head;
    insertNode(ptrHead,30);
}
```

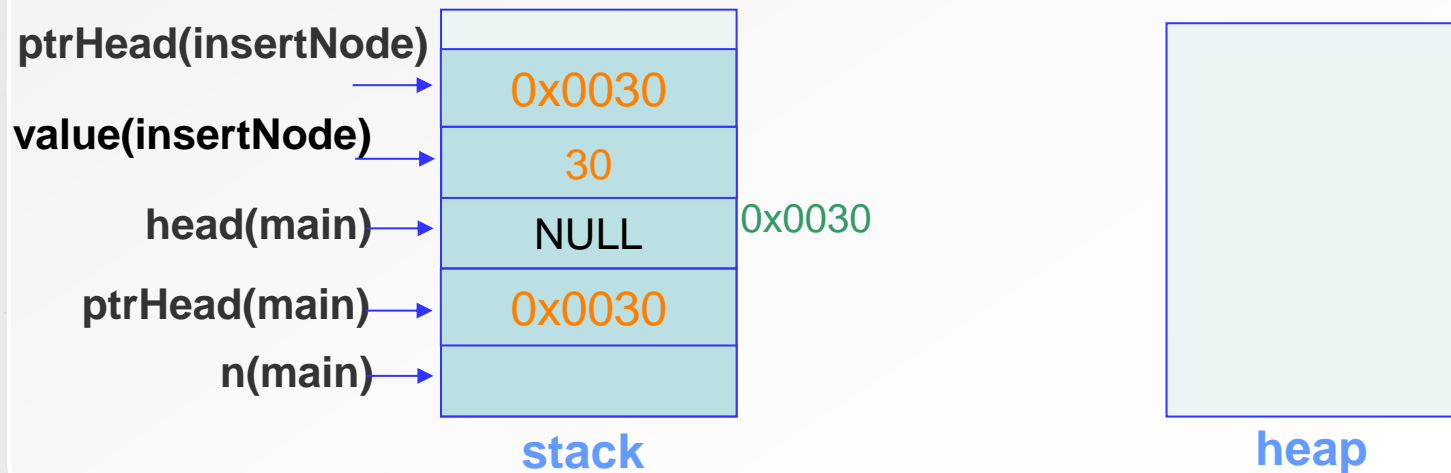
```
insertNode(ListNode **ptrHead, int value)
{
    if (*ptrHead==NULL) {
        *ptrHead=malloc(sizeof(ListNode));
        (*ptrHead)->item=value;
        (*ptrHead)->next=NULL;
    }
    ...
}
```



POINTERS AND PARAMETER PASSING

```
main()
{
    int n;
    ListNode **ptrHead,
           *head=NULL;
    ...
    ptrHead=&head;
    insertNode(ptrHead,30);
}
```

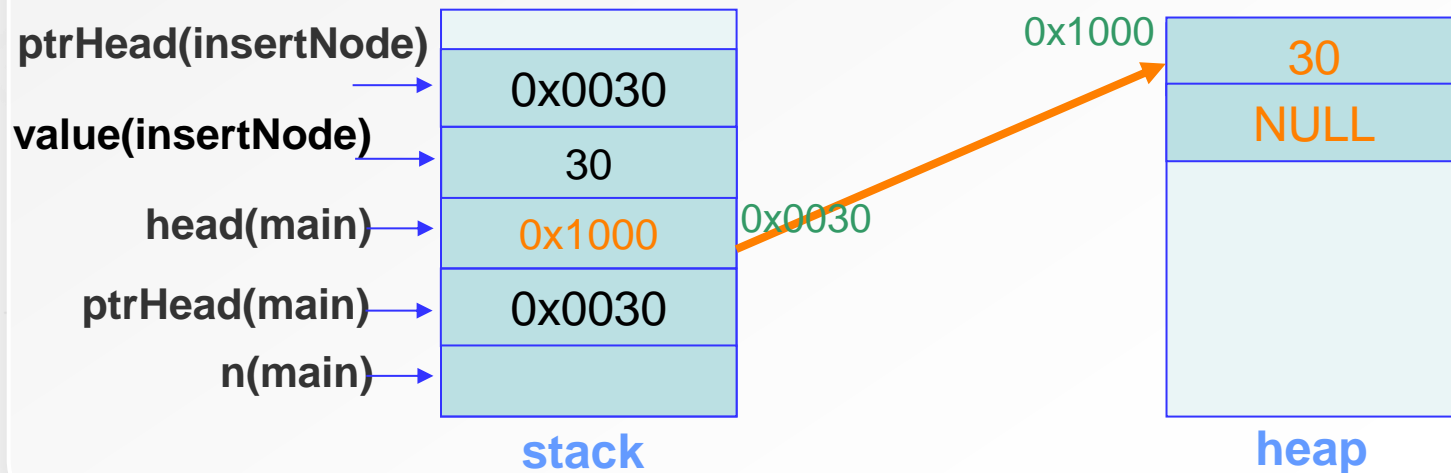
```
insertNode(ListNode **ptrHead, int value)
{
    if (*ptrHead==NULL) {
        *ptrHead=malloc(sizeof(ListNode));
        (*ptrHead)->item=value;
        (*ptrHead)->next=NULL;
    }
    ...
}
```



POINTERS AND PARAMETER PASSING

```
main()
{
    int n;
    ListNode **ptrHead,
           *head=NULL;
    ...
    ptrHead=&head;
    insertNode(ptrHead,30);
}
```

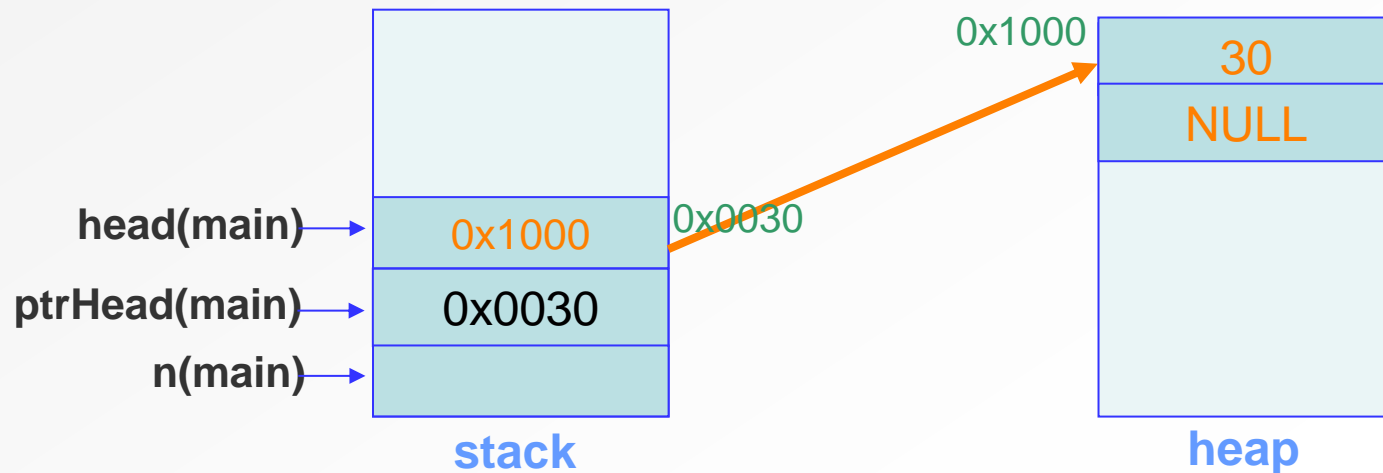
```
insertNode(ListNode **ptrHead, int value)
{
    if (*ptrHead==NULL) {
        *ptrHead=malloc(sizeof(ListNode));
        (*ptrHead)->item=value;
        (*ptrHead)->next=NULL;
    }
    ...
}
```



POINTERS AND PARAMETER PASSING

```
main()
{
    int n;
    ListNode **ptrHead,
            *head=NULL;

    ...
    ptrHead=&head;
    insertNode(ptrHead,30);
}
```



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;
```

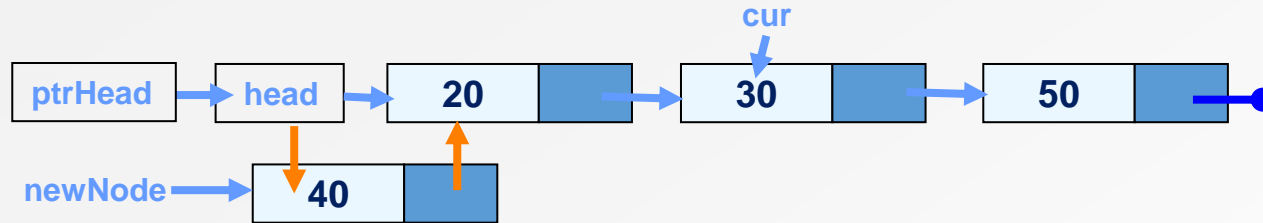
- Yes! In an empty list, head = NULL

insertNode()

```
1  int insertNode(ListNode **ptrHead, int index, int value){
2
3      ListNode *pre, *cur;
4
5      // If empty list or inserting first node, need to update head pointer
6      if (*ptrHead == NULL || index == 0){
7          cur = *ptrHead;
8          *ptrHead = malloc(sizeof(ListNode));
9          (*ptrHead)->item = value;
10         (*ptrHead)->next = cur;
11         return 0;
12     }
13
14     // Find the nodes before and at the target position
15     // Create a new node and reconnect the links
16     if ((pre = findNode(*ptrHead, index-1)) != NULL){
17         cur = pre->next;
18         pre->next = malloc(sizeof(ListNode));
19         pre->next->item = value;
20         pre->next->next = cur;
21         return 0;
22     }
23
24     return -1;
25 }
```

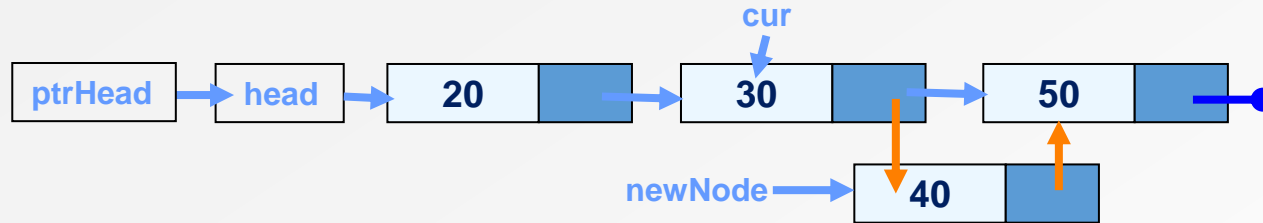


insertNode()



```
1 void insertNode(ListNode **ptrHead, int index, int value){
2     ListNode *cur, *newNode;
3     // If empty list or inserting first node, update head pointer
4     if (*ptrHead == NULL || index == 0){
5         newNode = malloc(sizeof(ListNode));
6         newNode->item = value;
7         newNode->next = *ptrHead;
8         *ptrHead = newNode;
9     }
10    // Find the nodes before and at the target position
11    // Create a new node and reconnect the links
12    else if ((cur = findNode(*ptrHead, index-1)) != NULL){
13        newNode = malloc(sizeof(ListNode));
14        newNode->item = value;
15        newNode->next = cur->next;
16        cur->next = newNode;
17    }
18    else printf(" can not insert the new item at index %d!\n", index);
19 }
```

insertNode()



```
1 void insertNode(ListNode **ptrHead, int index, int value){
2     ListNode *cur, *newNode;
3     // If empty list or inserting first node, update head pointer
4     if (*ptrHead == NULL || index == 0){
5         newNode = malloc(sizeof(ListNode));
6         newNode->item = value;
7         newNode->next = *ptrHead;
8         *ptrHead = newNode;
9     }
10    // Find the nodes before and at the target position
11    // Create a new node and reconnect the links
12    else if ((cur = findNode(*ptrHead, index-1)) != NULL){
13        newNode = malloc(sizeof(ListNode));
14        newNode->item = value;
15        newNode->next = cur->next;
16        cur->next = newNode;
17    }
18    else printf(" can not insert the new item at index %d!\n", index);
19 }
```

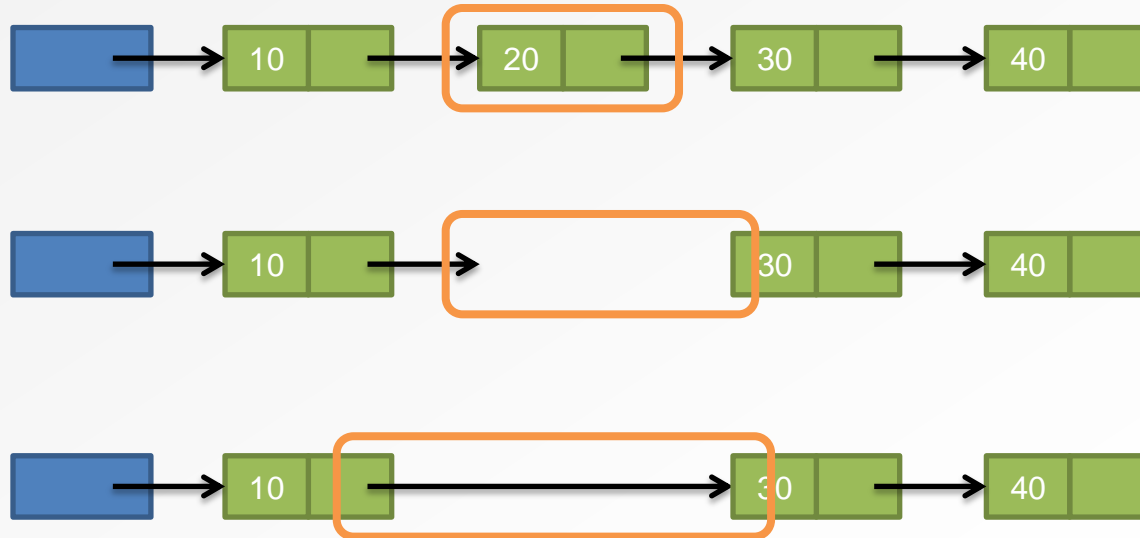
- ListNode structures
- Core linked list data structure functions
 - printList();
 - findNode();
 - insertNode();
 - **removeNode();**
- Common Mistakes

REMOVE A NODE FROM ANY POSITION OF THE LINKED LIST: `removeNode()`

- One of your week 10 lab question
- We'll go through the basic diagrams
- You write the code
- Again, we need to pass in a pointer to the head pointer
 - In case we delete the first node, we have to change the address stored in the head pointer (outside, not the local copy)
 - What are the other special cases?

REMOVE A NODE: removeNode()

- Remember to free up any unused memory



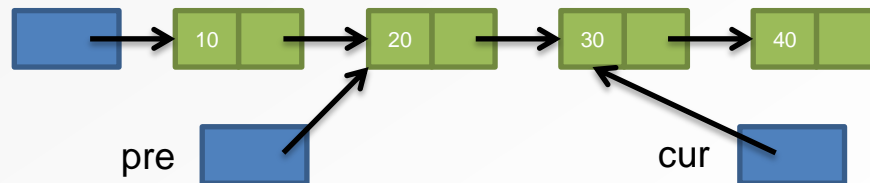
- ListNode structures
- Core linked list data structure functions
 - printList();
 - findNode();
 - insertNode();
 - removeNode();
- **Common Mistakes**

COMMON MISTAKES

- Forget to check whether the list is empty
head=NULL
- Forget to handle differently when: insert/remove a node at the beginning of the list

COMMON MISTAKES

- What is cur?
- What is pre?
- State 3 ways of getting the address of the node at index 2 (3rd node)



YOU SHOULD BE ABLE TO...

- Describe and implement the core linked list functions
 - Draw the diagrams for each step
 - Write pseudocode (if necessary)
 - Write C code to implement the functions
- Carry out the same process for any linked list function

- Advanced linked lists