NANYANG TECHNOLOGICAL UNIVERSITY
SINGAPORE

# CE1107/CZ1107: DATA STRUCTURES AND ALGORITHMS

## Linked List Functions

**College of Engineering**
School of Computer Science and Engineering

# TODAY

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

# LEARNING OBJECTIVES

After this lesson, 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

# 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 or pseudocode to implement a linked list in any language

# TODAY

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

# RECALL: ListNode STRUCTURE

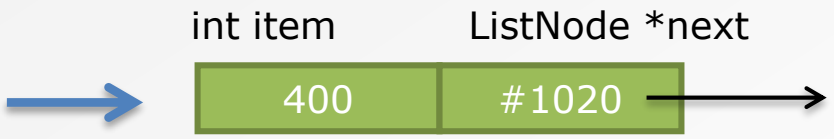
- Our default ListNode for the rest of the class will store an integer item

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

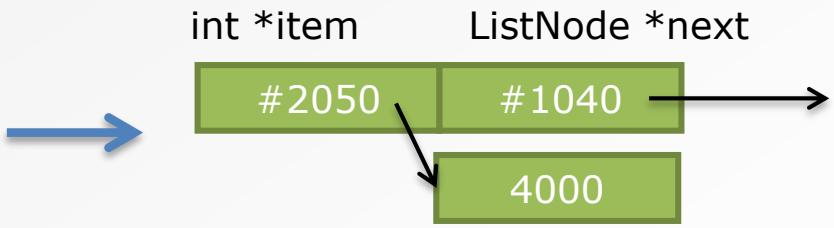
- ListNodes can store anything in the item field
  - int or int*
  - Array of integers
  - char or char*
  - Another struct or a pointer to a struct
  - Whatever you want
  - Can even define int item1, item2

# ADVANCED ListNode STRUCTURES

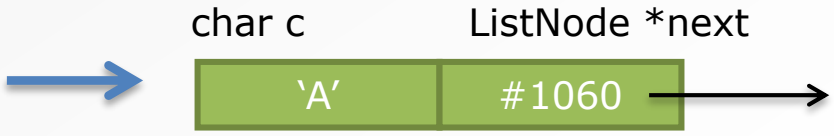
```
typedef struct _listnode{
    int item;
    struct _listnode *next;
} ListNode;
```



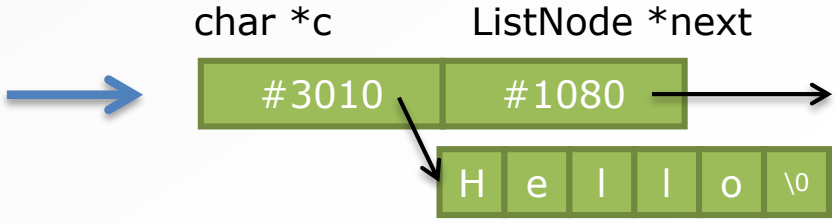

```
typedef struct _listnode{
    int *item;
    struct _listnode *next;
} ListNode;
```




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

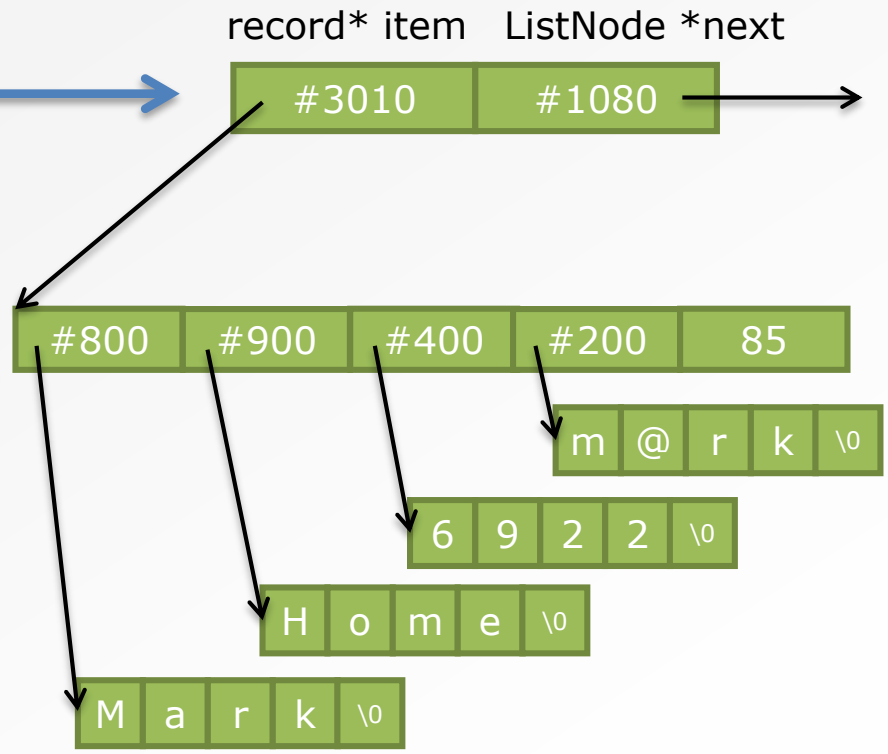



```
typedef struct _listnode{
    char *c;
    struct _listnode *next;
} ListNode;
```



# 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;
}
```

# TODAY

- 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)

# 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.

# TODAY

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

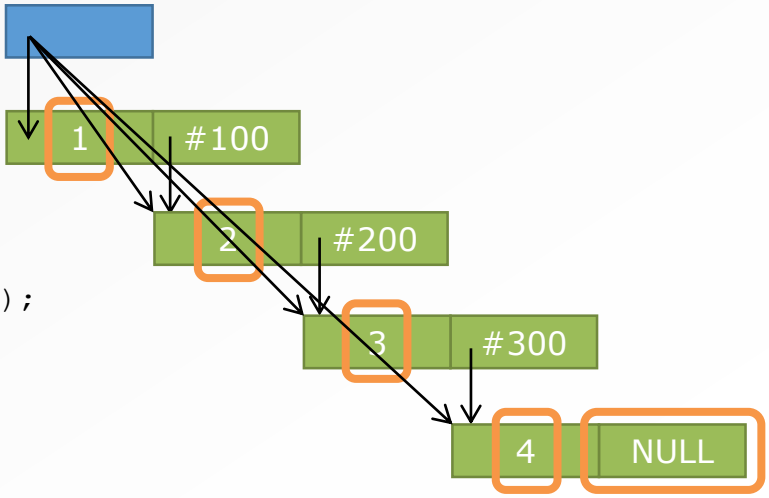
# PRINT OUT ITEMS IN LINKED LIST: printList() [ANIMATED]

- 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

```
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  }
```



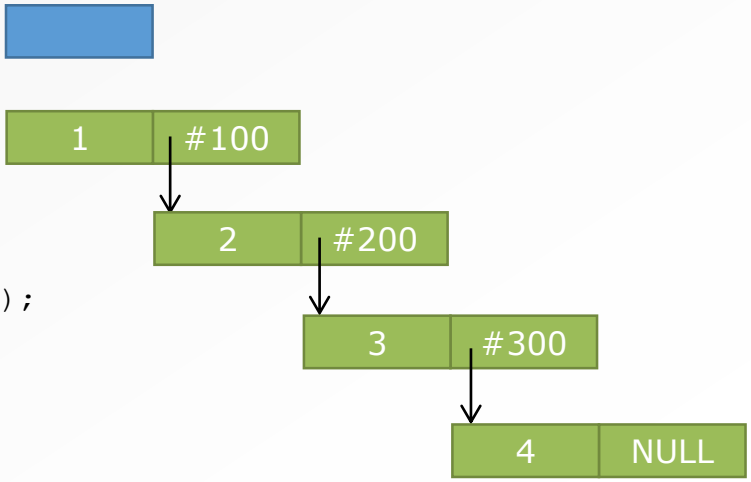
# PRINT OUT ITEMS IN LINKED LIST: printList()

- 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  }
```



# TODAY

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

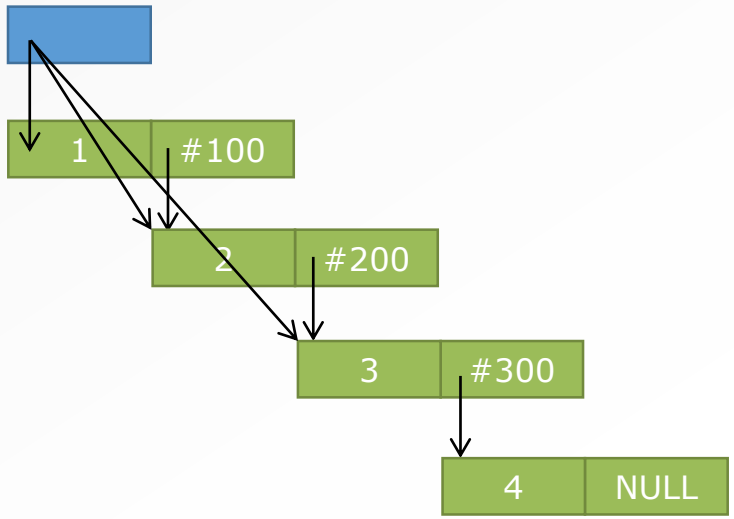
# GET POINTER TO NODE AT INDEX i: findNode() [ANIMATED]

- 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

```
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  }
```



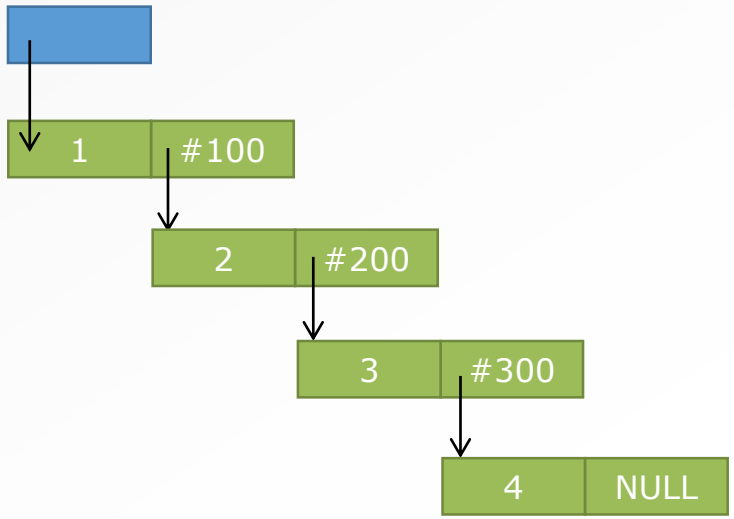
# GET POINTER TO NODE AT INDEX i: findNode()

- 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  }
```



# TODAY

- 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?

```
void insertNode(          )
```

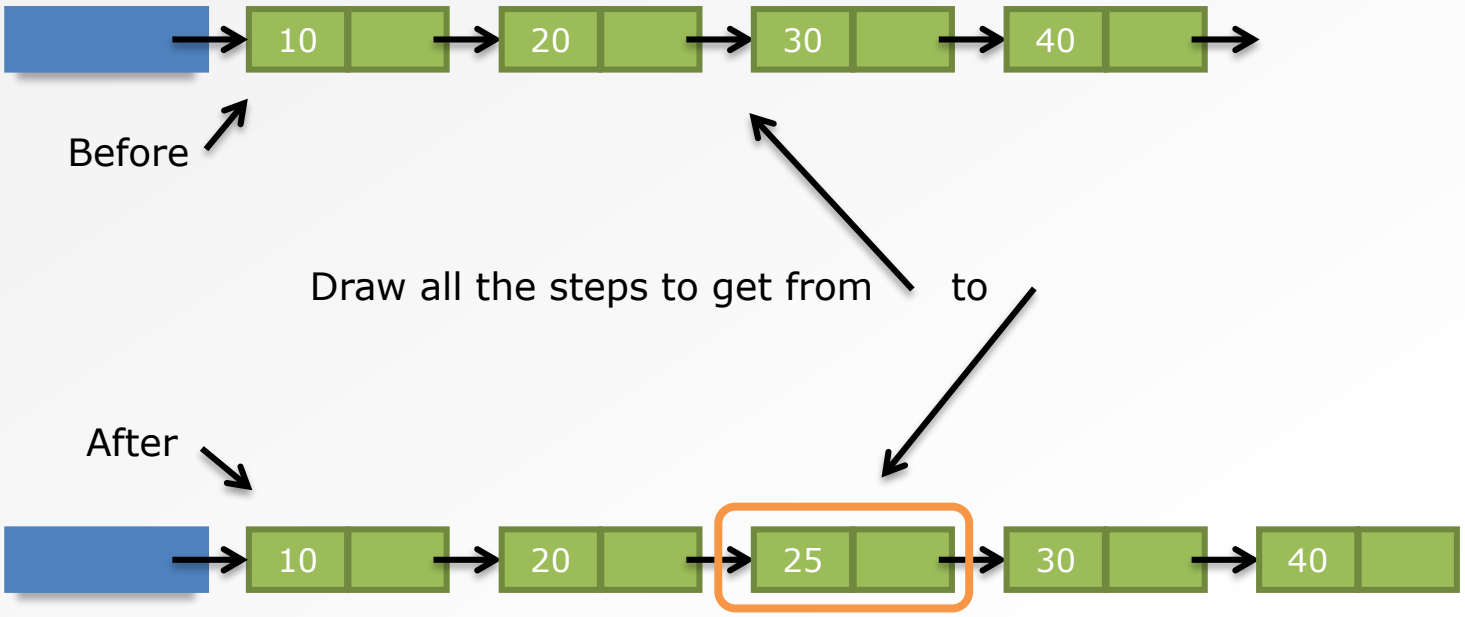
  - KIV – this will become obvious later
  - There is an apparently correct but actually wrong answer

# INSERT A NODE: insertNode()

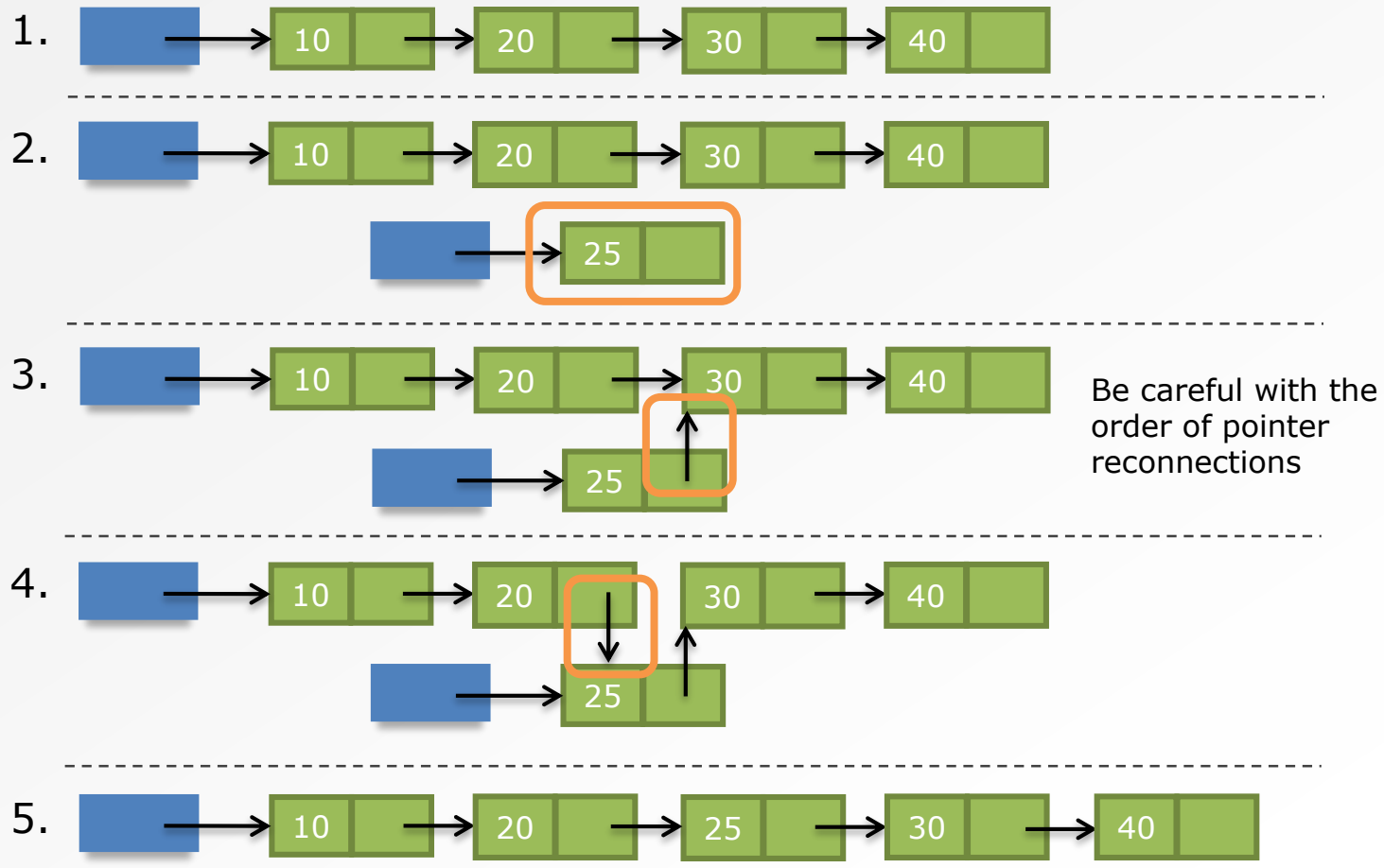
- Consider all the different places we want to add a node
  - Front
  - Back
  - Middle
- Consider all the different starting states of the linked list
  - Empty list
  - One node
  - Many nodes
- Ok to create many special cases and merge them later when we see similar code
- Get it right before you try to optimise
- 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



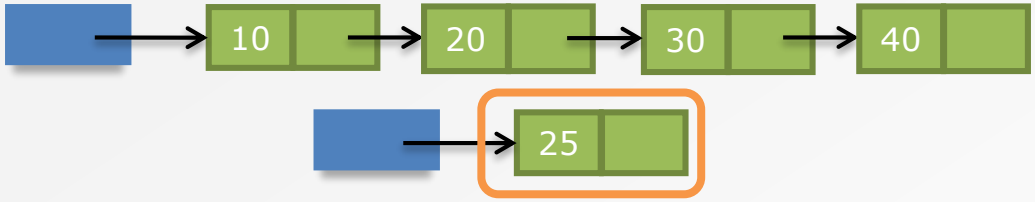
# INSERT A NODE: insertNode()



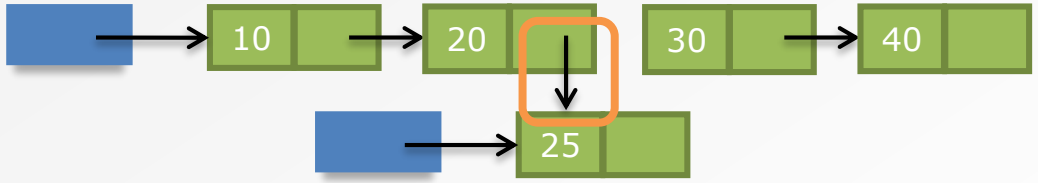
# INSERT A NODE: insertNode()

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

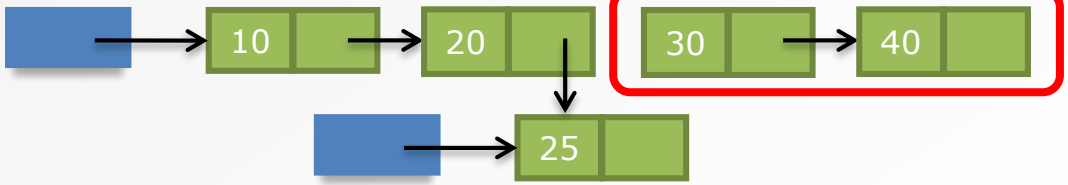
2.




3.

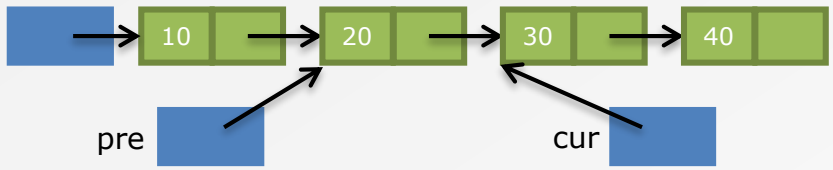


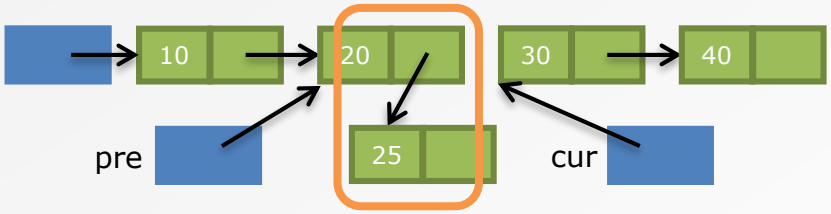

4.

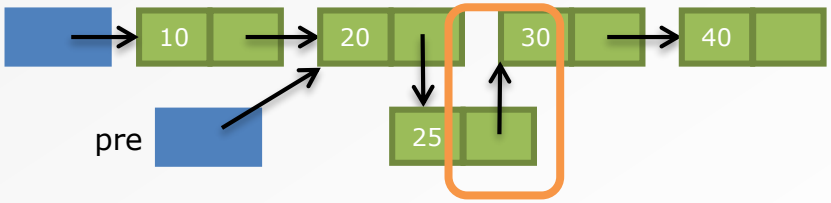



All gone! Inaccessible in memory since we lost the address of [30]

# 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()?

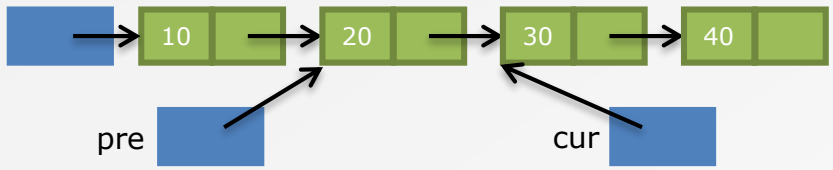
2. Create a new node and store its address in pre->next

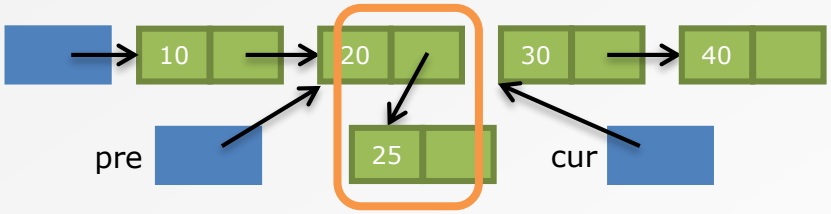
```
Pre->next = malloc(sizeof(ListNode));
```

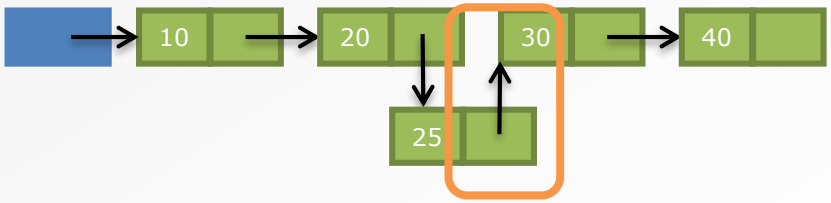
3. Set the new node's next pointer
   New node currently at pre->next
   Next pointer of new node is pre->next->next

```
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)

```
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 = [addr 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?

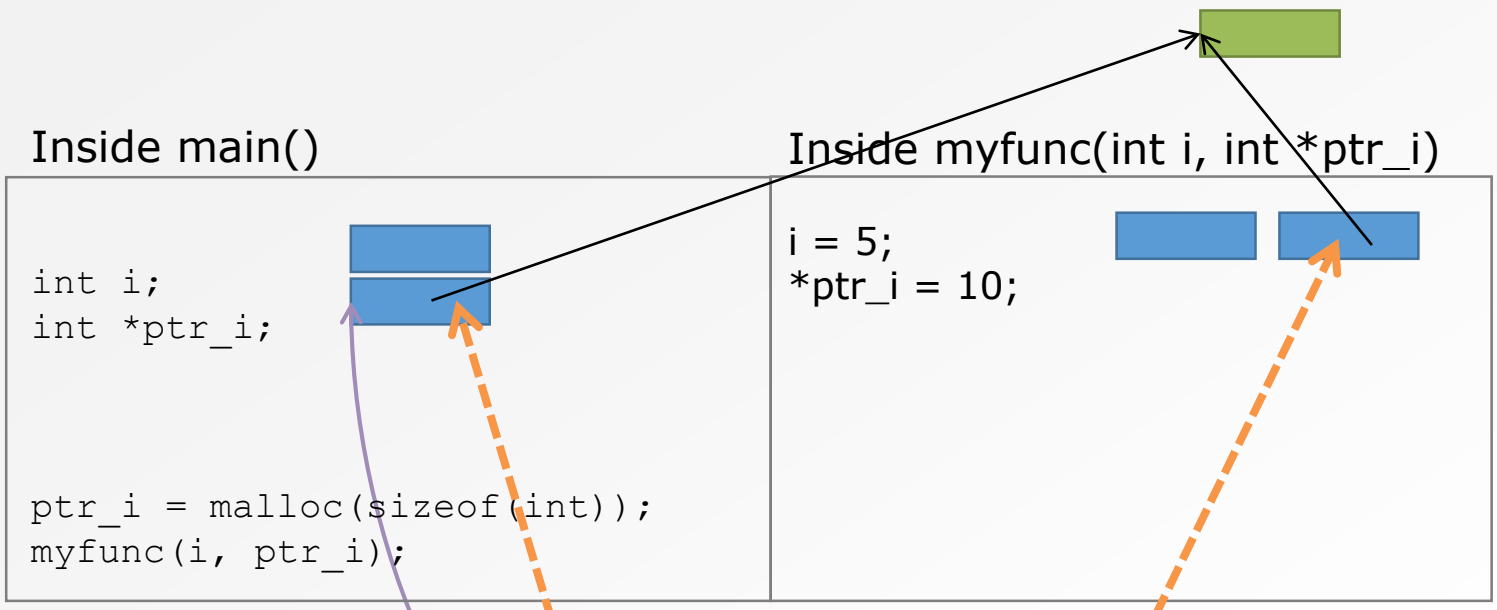
# INSERT A NODE: insertNode()

- 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




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   }
```

# TODAY

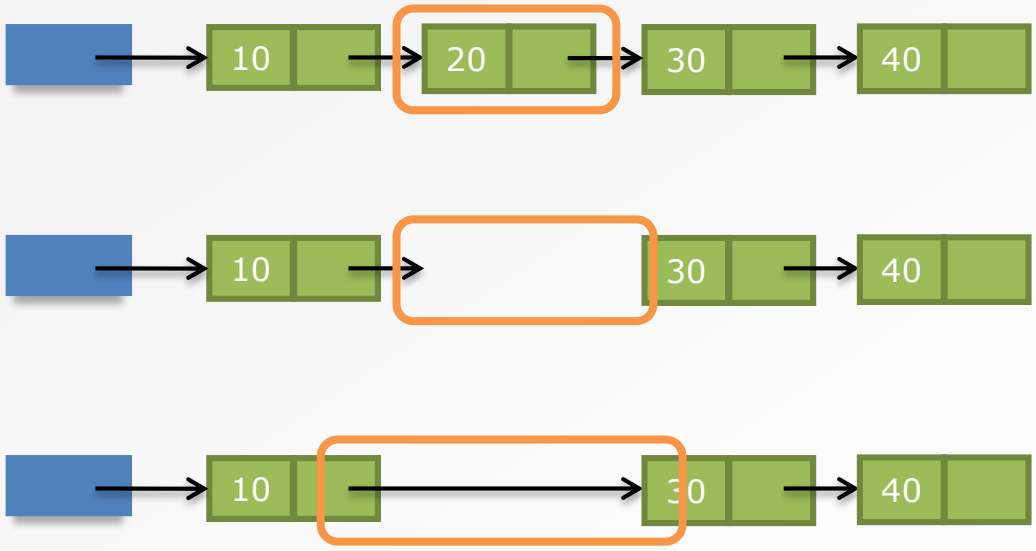
- ListNode structures
- Core linked list data structure functions
  - printList();
  - findNode();
  - insertNode()
  - **removeNode()**
- Common mistakes

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

- Do this as one of your nine lab questions
- We will 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

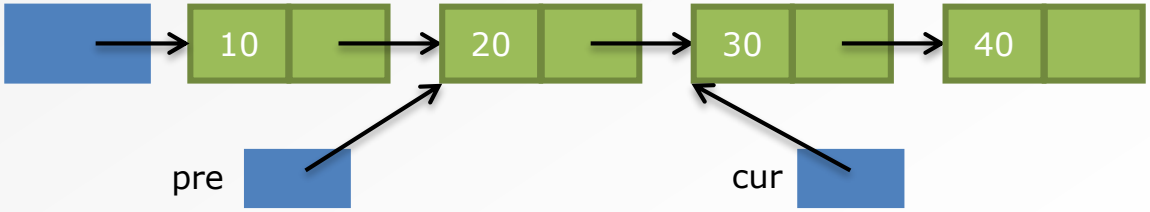


# TODAY

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

# COMMON MISTAKES

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



# NEXT LECTURE

- Application: Worked example
- Advanced linked lists
- Array-based implementations of linked lists