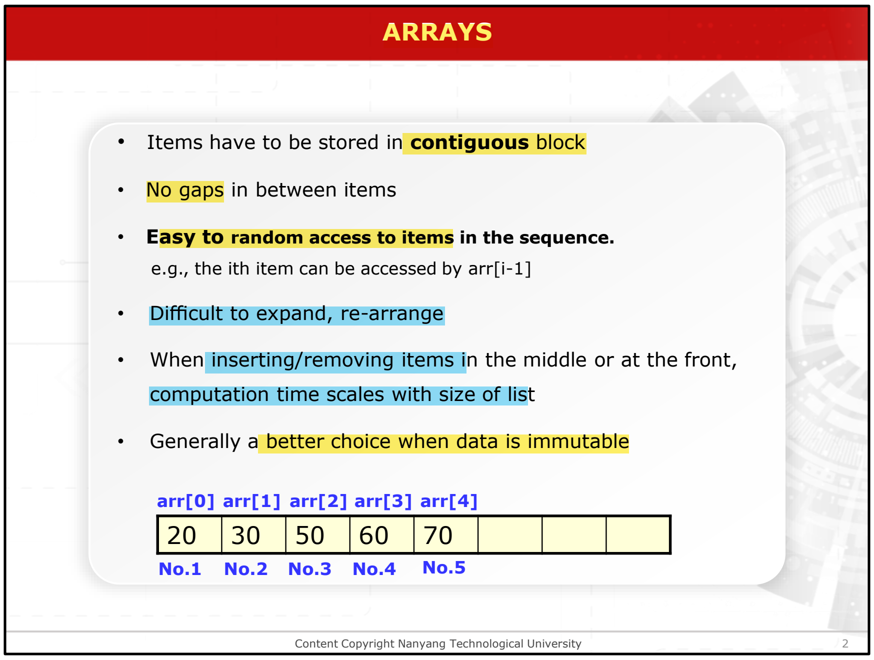
NANYANG TECHNOLOGICAL UNIVERSITY

Review: Linked List

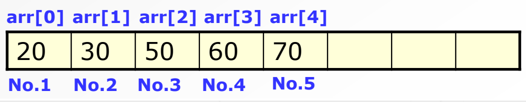
College of Engineering
School of Computer Science and Engineering

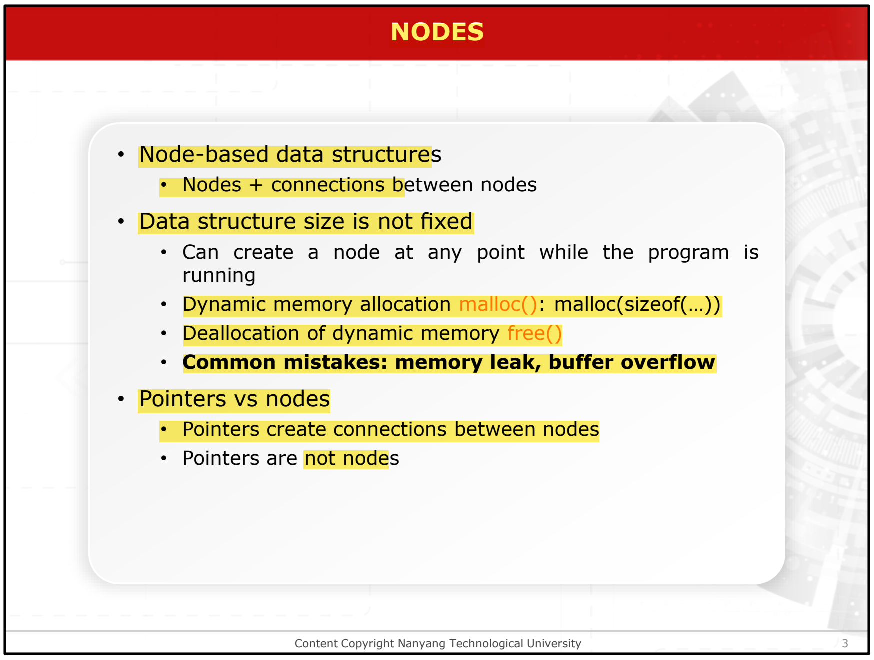
Brief Overview



ARRAYS

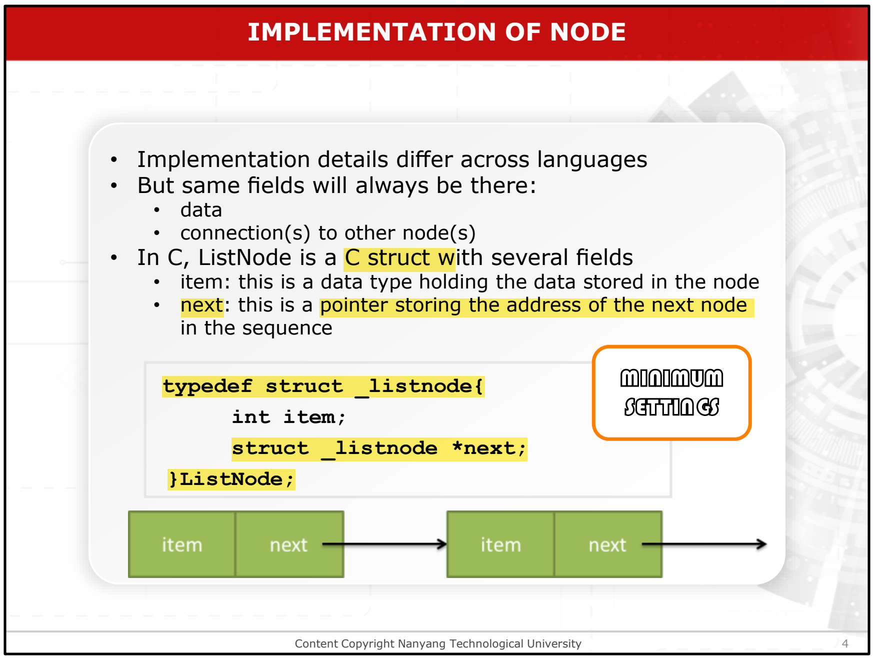
- Items have to be stored in **contiguous** block
- No gaps in between items
- **Easy to random access to items in the sequence.**
  e.g., the ith item can be accessed by arr[i-1]
- Difficult to expand, re-arrange
- When inserting/removing items in the middle or at the front, computation time scales with size of list
- Generally a better choice when data is immutable





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

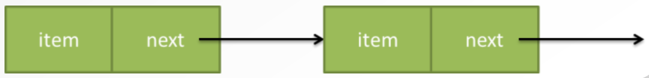


IMPLEMENTATION OF NODE

- 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

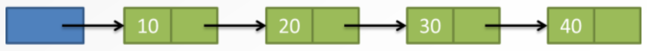
```
typedef struct _listnode{
        int item;
        struct _listnode *next;
 }ListNode;
```

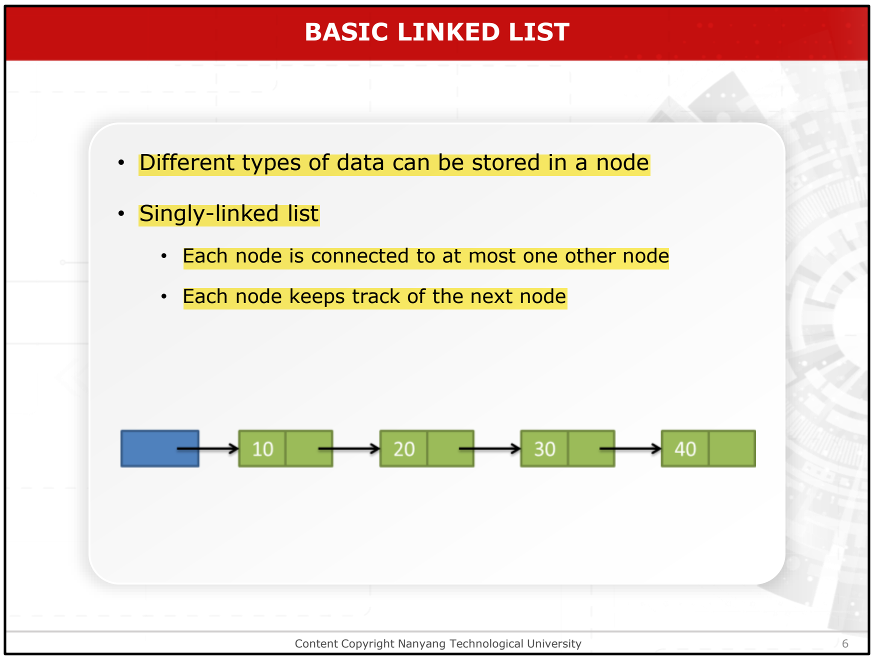
MINIMUM SETTINGS



LINKED LISTS

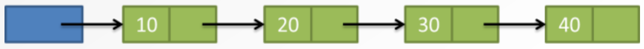
- What is a linked list?
  - Ordered list of items
  - Each item stored in a node
  - Each node connects to the next node in the series
- No need for pointers in definition of a linked list
  - Head pointer, next pointer: all <u>implementation</u> details

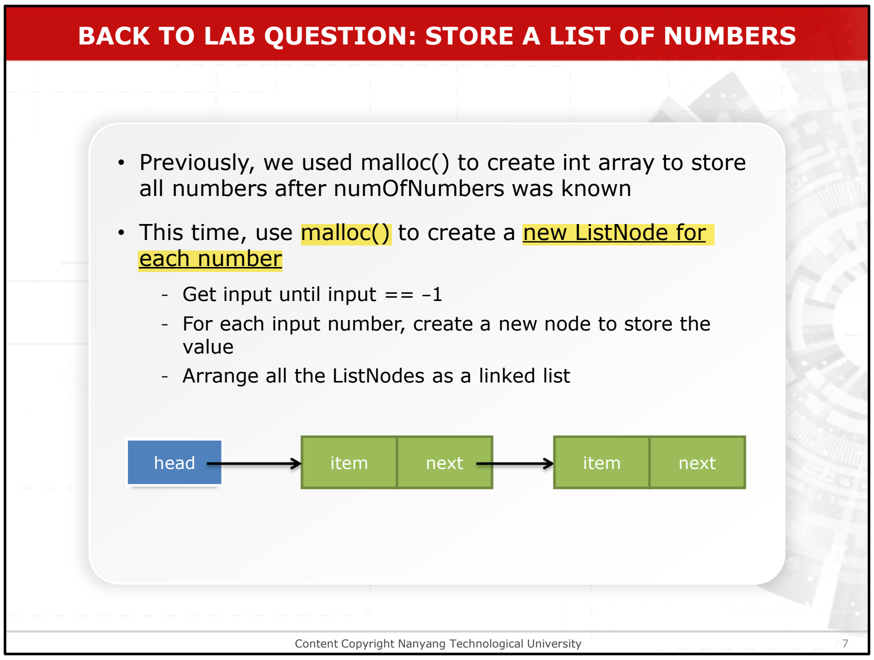




BASIC LINKED LIST

- 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





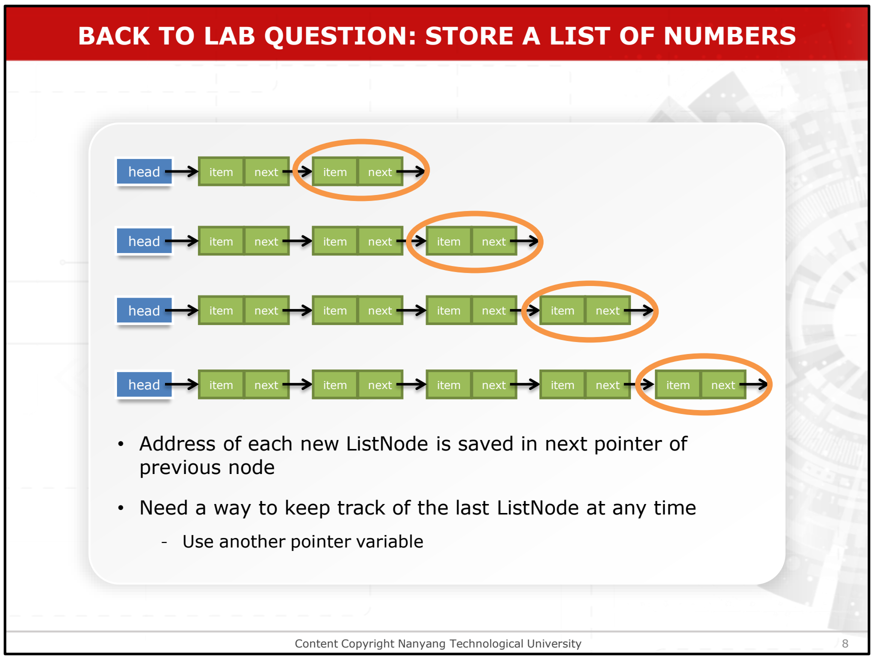
BACK TO LAB QUESTION: STORE A LIST OF NUMBERS

- 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

head
item
next
item
next

7

Remember the lab question where you were asked to calculate the average of the list of numbers? You were supposed to keep running the program until you get a sentinel value of -1 which informs that you have reached the end of the list of numbers. Previously you only knew about arrays and static allocation of arrays. Therefore, you used a pre-allocated array with a larger value. But, using malloc(), you can allocate just enough space you need to store the numbers. Yet, you don't know how many numbers in the list until you reach the end of the list. This where linked list is useful. You just have to use malloc() as we did on the previous slide. Every time I see a new number, and we want to create a new node to store that value, I'm going to add it to the back of my list. Eventually, this whole linked list of nodes gives me my list of numbers.



BACK TO LAB QUESTION: STORE A LIST OF NUMBERS

head
item
next

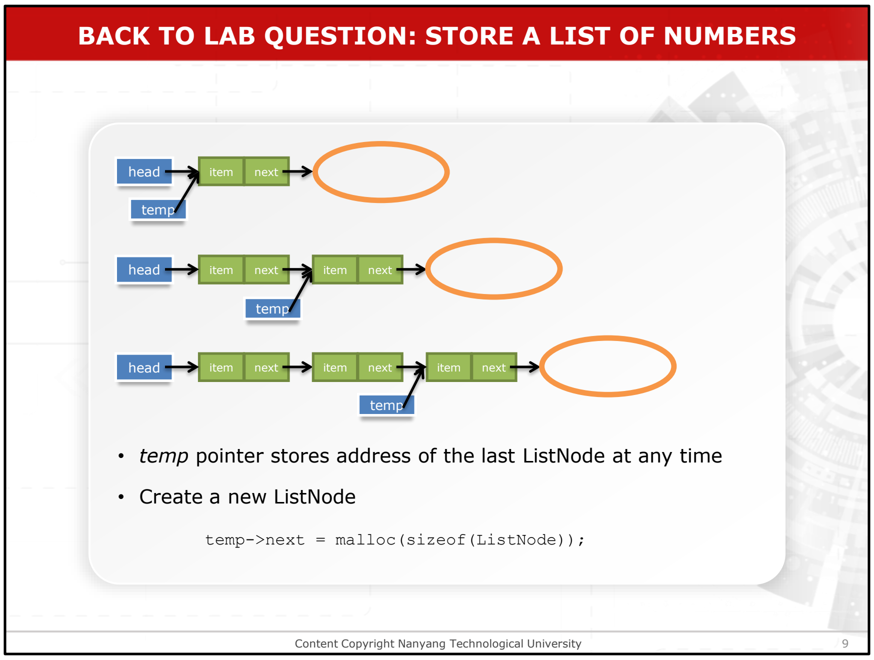
- Address of each new ListNode is saved in next pointer of previous node
- Need a way to keep track of the last ListNode at any time
  - Use another pointer variable

8

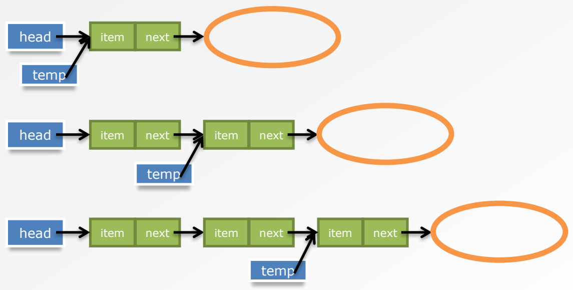
The image shows the different stages of linked list as I build it. Every time I am adding a new number, I create a new node by using malloc() function and hook the node to the back of the existing linked list. I do not need to worry about the sized, whether I have pre-allocated enough memory, etc.

But, there is a slight problem with this sequence. As you may remember, the head pointer directs you to the entire list. Head pointer points you to the first node and the first node points you to the second, likewise. Now, if you look at the third diagram in the slide, we have three nodes and wants to add the fourth node which is circled in orange. To add the fourth node, we need to get the address of the third node to get its next pointer. To get the address of the third node, we need the address of the second node to get to its pointer. Likewise, to insert a new node, we still need to start from the head pointer and trace all the way down to traverse the entire linked list.

Since this way is inefficient, we need to keep track of the last list node to make it efficient.



BACK TO LAB QUESTION: STORE A LIST OF NUMBERS

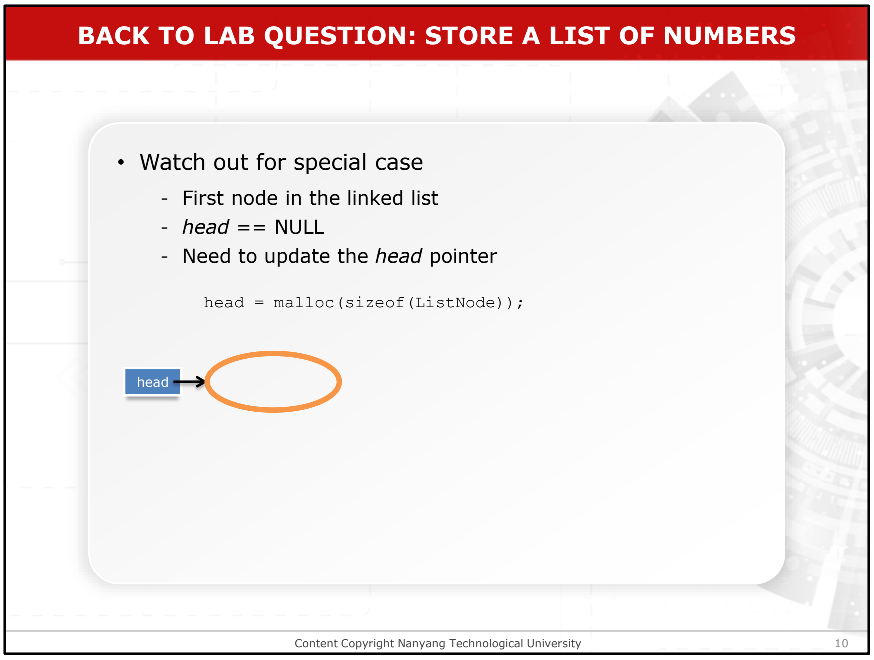


- *temp* pointer stores address of the last ListNode at any time
- Create a new ListNode

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

9

We can start adding temporary pointers to keep track of the last list node. The temporary pointer moves down every time we add a new node so that it is always pointing at the current last list node of the linked list.



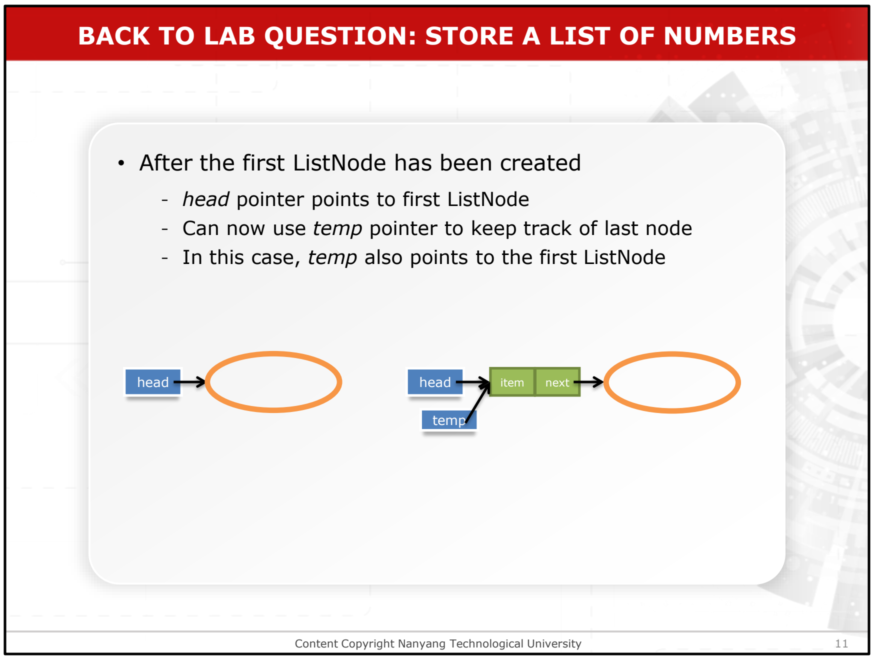
BACK TO LAB QUESTION: STORE A LIST OF NUMBERS

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

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

Content Copyright Nanyang Technological University 10

There are certain special cases when adding a temporary pointer. If we start with an empty linked list, we are not actually changing and next pointer from a node. Therefore, we change the value of the head pointer to point at the address of the first node. Still, we use malloc() to allocate memory for the first list node, but this time we store the address in the head pointer.



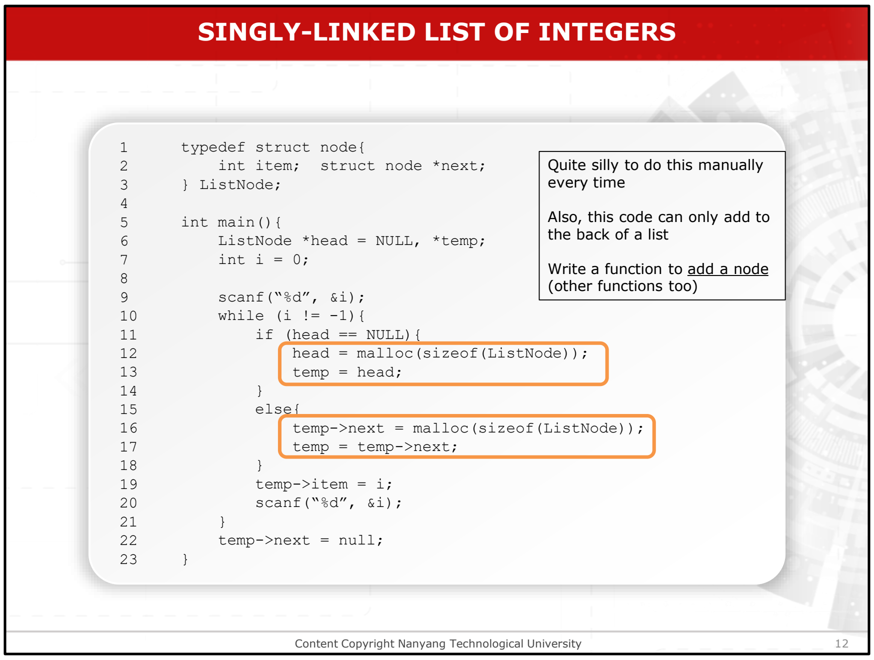
BACK TO LAB QUESTION: STORE A LIST OF NUMBERS

- 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

head
head
item
next
temp

11

When we add a new node to an existing linked list which has one node, we change the value of the next pointer of the first node. But in an empty linked list, since we do not have any nodes, we change the value of the head pointer.



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;
}
```

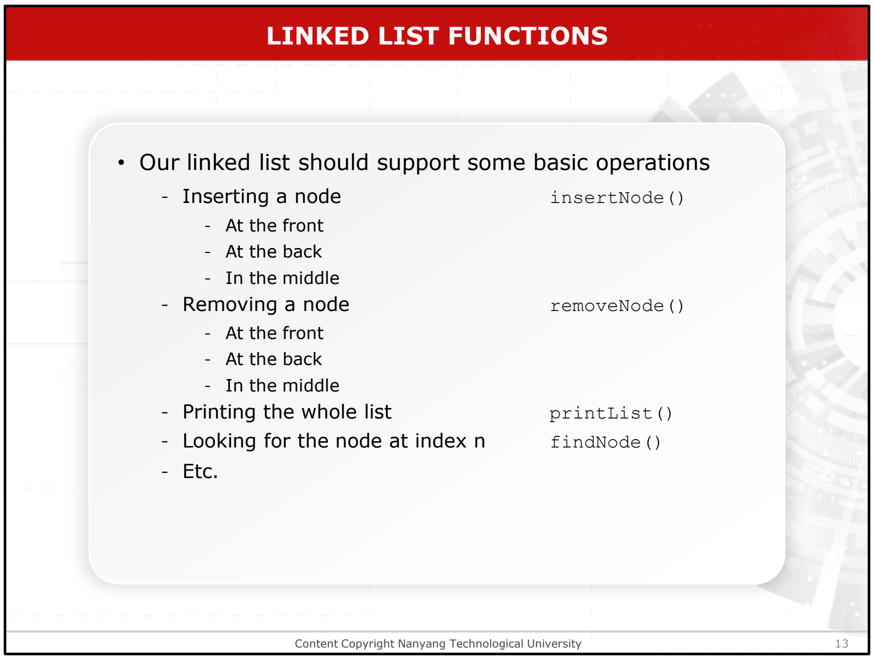
Quite silly to do this manually every time

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

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

We had discussed about the singly linked list of integers. There are two cases to be considered. If the head pointer equals to NULL, we will create a new node and set its address to the head pointer because that is the first node of the linked list. Otherwise, we will add it to the back of the linked list, next to the last list node. If we are creating the first node of the list, the return value of malloc() will be set to the head pointer. If not, the return value of malloc() will be set to the temp pointer which is pointing to the current last list node.

But, every time we run this code, it has to deal with different pointers and it is also limited to insert items only to the back of the linked list. Since we need to run through the code every time we insert a new node, it is better that we put it into a function. The function can build in to add a new node to any given position of the linked list with any given value.



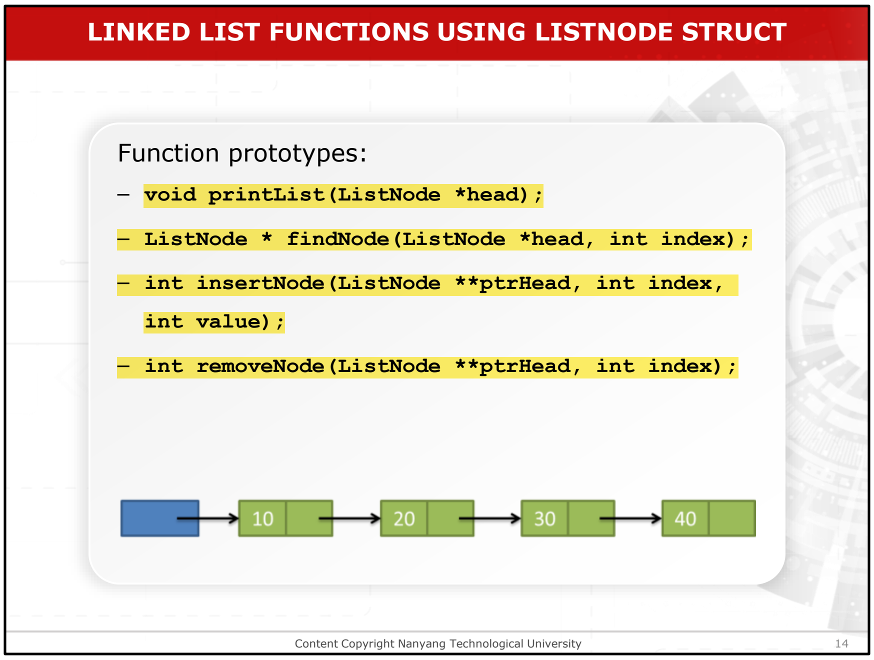
LINKED LIST FUNCTIONS

- 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.	

13

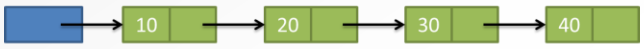
These are the functions that we are going to learn during this lecture. The insertNode() function can be used to insert nodes not only to the back but also to the middle and the front of the linked list. The removeNode() function allows you the remove a node from the back, from the middle and the front of the linked list. The printList() function prints the entire list of numbers and findNode() function look for a node at a specific index position.

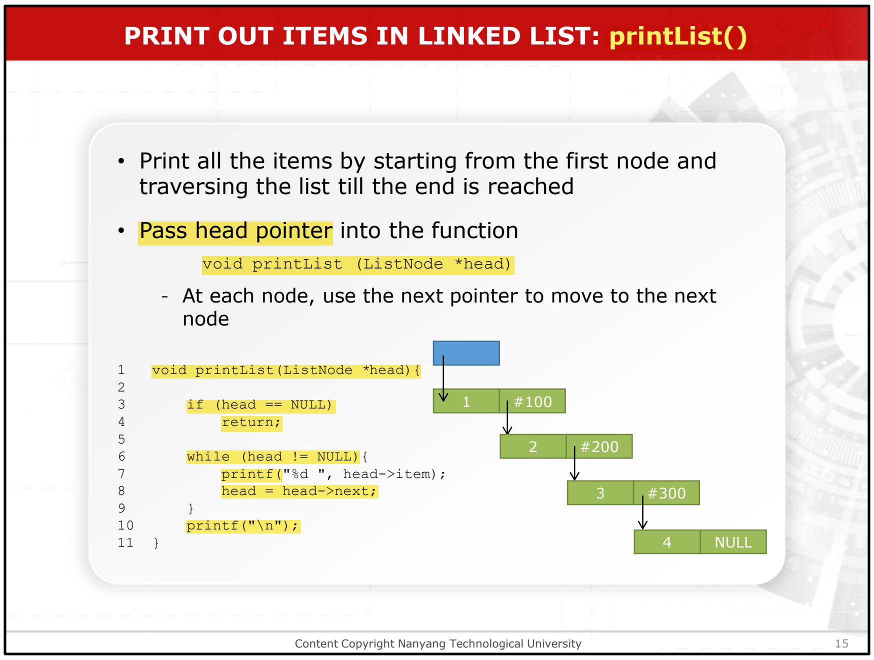


LINKED LIST FUNCTIONS USING LISTNODE STRUCT

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);`





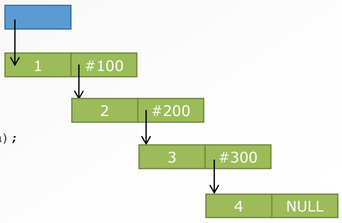
PRINT OUT ITEMS IN LINKED LIST: printList()

- 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

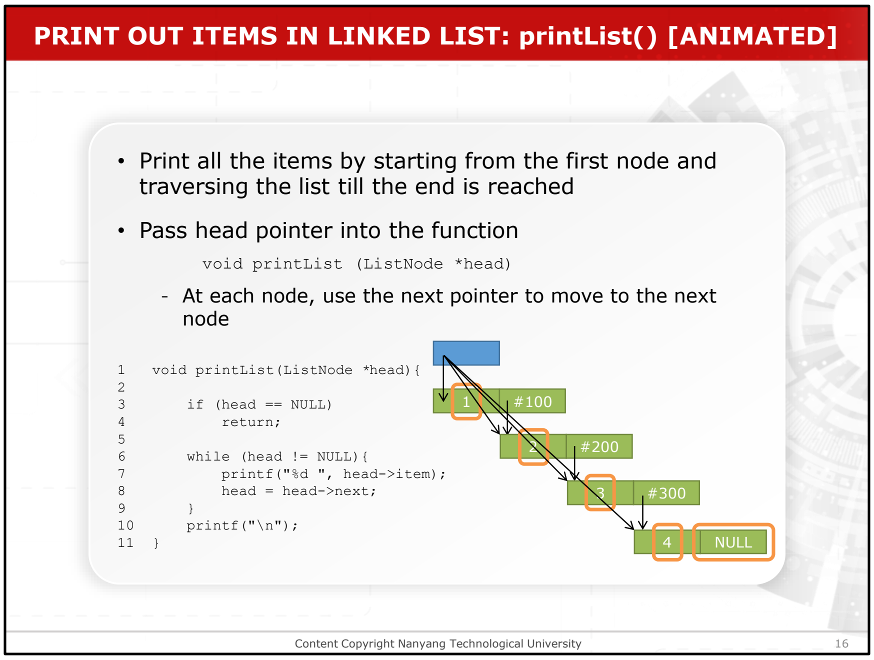
  `void printList (ListNode *head)`

  - 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");
}
```



That is how we print out items in linked list.



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  }
```

Content Copyright Nanyang Technological University 16

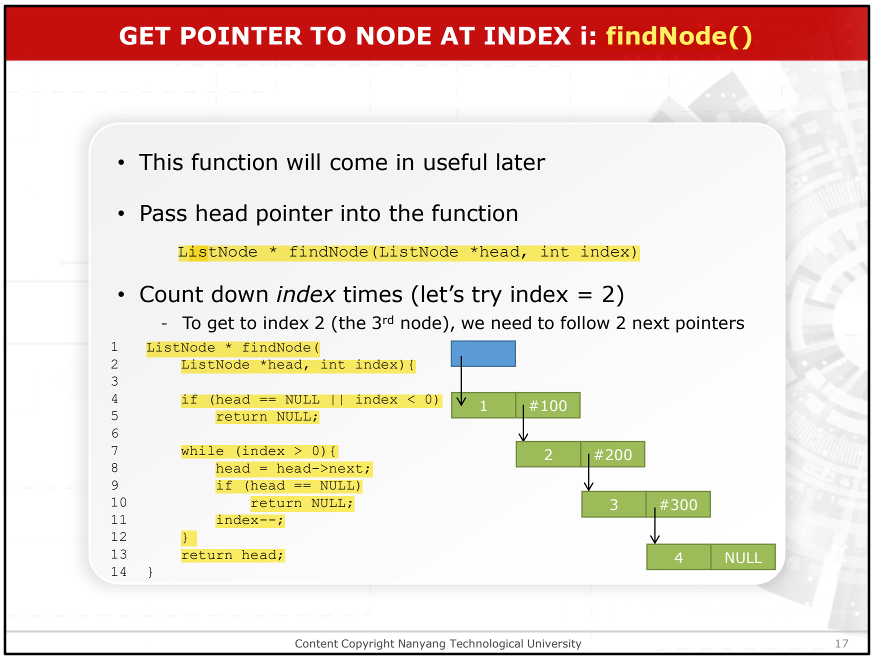
The general concept of printList() is that starting from the first node, the function will print the value stored inside each node while traversing the entire list.

We pass the head pointer to the printList() function. Here, the head pointer performs as a local variable. Therefore, any changes occur with the head pointer inside the function will not apply to the head pointer which points to the first list node.

We can traverse the linked list using a temporary pointer. Every time the temporary pointer gets to a node, the value inside of that node will be printed. Since the head pointer variable within this function is a local variable, we can use the head pointer variable as the temporary variable to traverse the list.

In the given code sample, line 3 and 4 represents the sanity check to make sure that the list is not empty. While the list is not empty, we check the value of the first node using heal->item and print it. Then we move the temporary head pointer to the next node. The loop will continue running until the pointer reaches to the last list node.

When the pointer hits the last list node, and when you try to get to the next node using head->next, since the next pointer of the last node is NULL because it is not pointing at another node, therefore, the head gets a NULL value. Therefore, it ends the while loop because now the head is NULL. Therefore, we get out of the loop and end the printing by printing a new line.



GET POINTER TO NODE AT INDEX i: findNode()

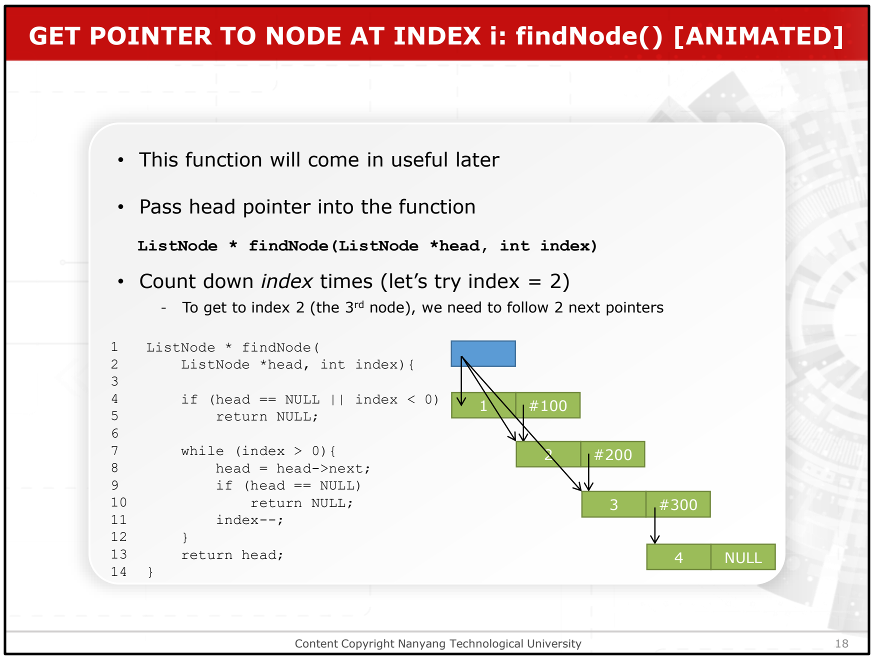
- This function will come in useful later
- Pass head pointer into the function

```
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;
}
```

That is how we get a pointer to the node at index i.



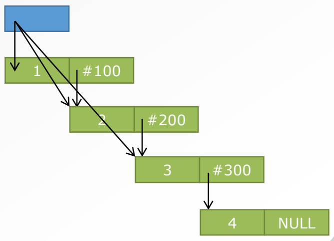
GET POINTER TO NODE AT INDEX i: findNode() [ANIMATED]

- 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  }
```



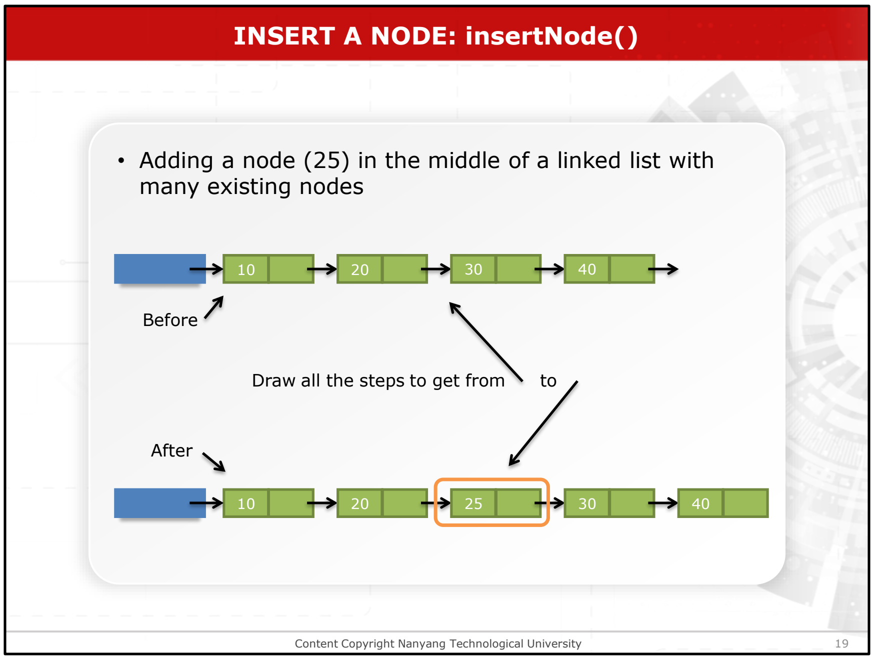
Content Copyright Nanyang Technological University 18

The findNode() function is important especially when you try to insert and remove nodes from the linked list.

This time we return ListNode* because we are now returning a pointer to one of the nodes inside the list. For example, if I have five nodes on my list and I pass findNode() index=2, I should get the address to the third node of my list.

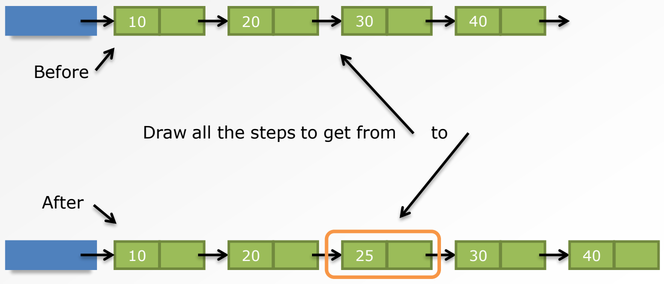
Let's understand the code I have given on the slide. As we have discussed in a previous slide, we cannot perform the findNode() function if the list is empty, therefore using the code chunks in line 4 and 5, we need to check if the list is empty or not. At the same time, we need to check whether the index value we are passing to the function is not less than 0 because negative index nodes do not exist.

To get to the correct index, we use a while loop as in line 7 to 12. For example, if I want to get to the third node of the list, I should pass 2 as the index value. Since the head pointer is already pointing at the first node and we need to get to the third pointer, we need to jump 2 nodes down. Therefore, when I pass a certain index value, we follow the next pointer to the next node index number of times. Thereafter, every time we follow one next pointer to the next node, we decrement the index value until index value equals to 0. Once the index equals 0, the while loop stops and return the head value which is the address of the requested node.

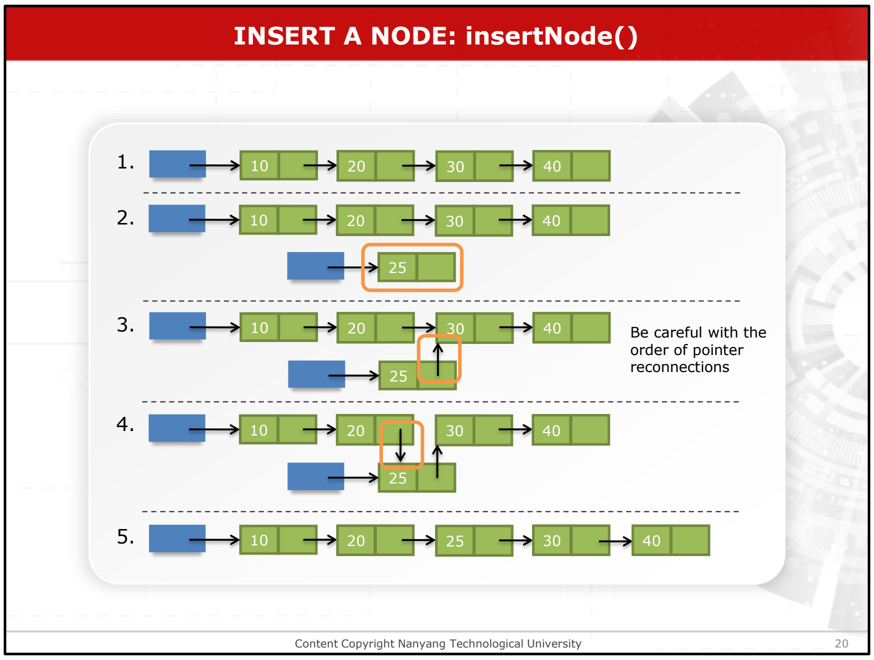


INSERT A NODE: insertNode()

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



We can start by adding node 25 in the middle of an existing linked list with many nodes as shown in the diagram. The linked list contains four nodes with values 10, 20, 30 and 40. We try to add 25 between node 20 and node 30.



INSERT A NODE: insertNode()

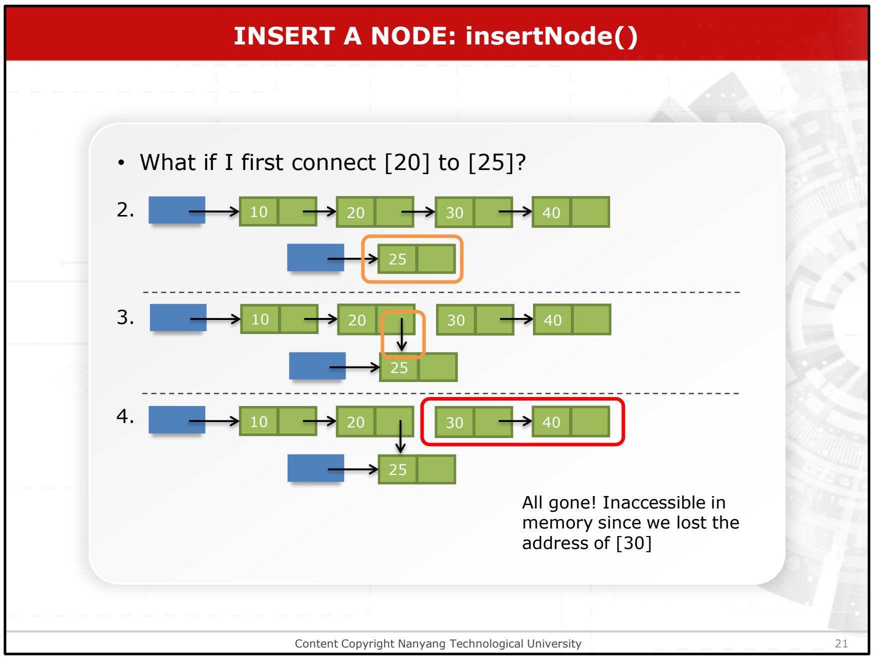
20

Step 1 and 5 have been taken directly from the previous slide. Now, we need to focus on how to get to step 5 starting from step 1.

Firstly, we need to create a new node which has the value of 25.

Then, we will connect the next pointer of the newly created node with the node which is supposed to be next of the node 25 once it is connected with the linked list, which is node 30. Still, node 20 is connected to node 30. Therefore, we need to break the connection between node 20 and node 30; we need node 20 to be connected with node 25.

In step 4 we have changed the link between node 20 and 30 by relinking it with node 25 as shown in the diagram. Now, a temporary pointer which is pointing at the new node, node 25, will be removed. But you have to realise that the order in which you create links in step 3 and 4 is very important. What happens if I swap the order by executing step 4 before step 3?



INSERT A NODE: insertNode()

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

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

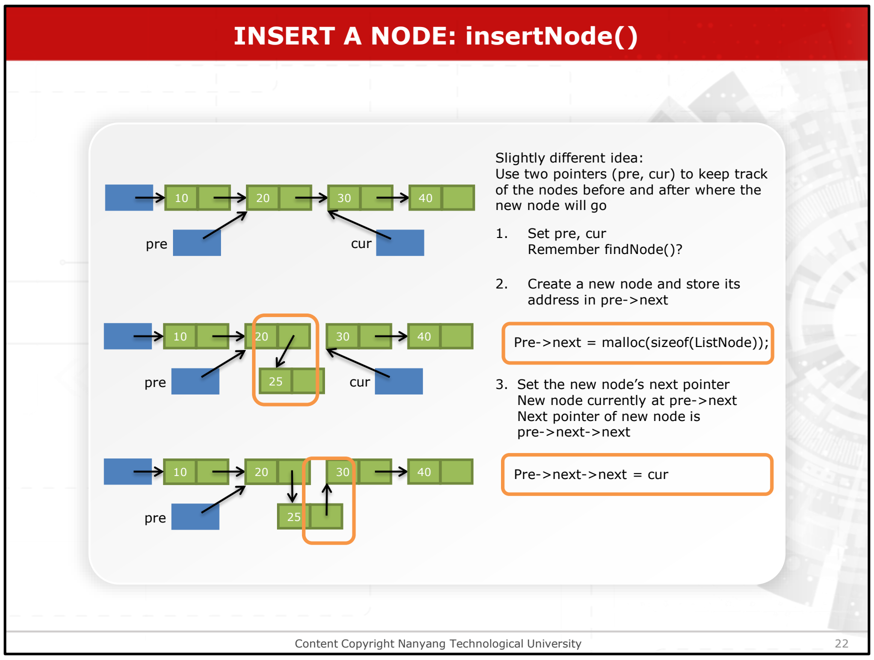
Content Copyright Nanyang Technological University 21

What if I connect node 20 to node 25 first?

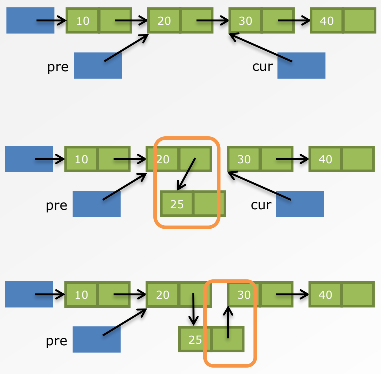
As in the previous slide, first of all, I create a new node.

Then, I change the next pointer of node 20 to point at the new node.

After connecting node 20 to node 25, I am going to change the next pointer of node 25 to point at node 30. But, we cannot point at node 30 because we lost the address of node 30. Therefore, it is very important to keep the order of the pointer operations.
To avoid these manipulations, one pointer is not enough.



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
```

22

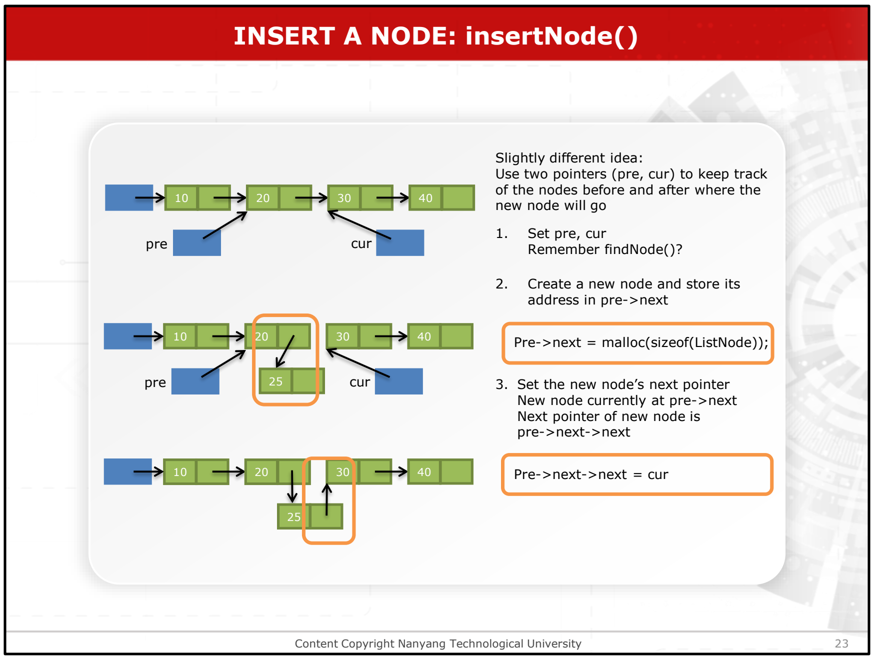
We introduce two pointers as previous and current to keep track of the before and after nodes of the newly added node. Since our new node goes between node 20 and node 30, the previous node is node 20, and the current node is node 30.

So, why do we use the current for the pointer instead of new or after? Because current refers to the node with the index you want. Here, we are trying to insert into index 2, which is the current index of node 30. We are trying to insert a new node to the current index position and move the current index down.
The findNode() function will be used to pass the values for each of these previous and current pointers.
After creating a new node, I can now directly set the address of the new node to the next pointer of the previous node, node 20. By doing this, I have lost the address of node 30 from node 20. But, since the current pointer points at the node 30, I still can retrieve the address of node 30. Finally, I have to reset the next pointer of the new node to node 30 by using the value inside the current pointer.

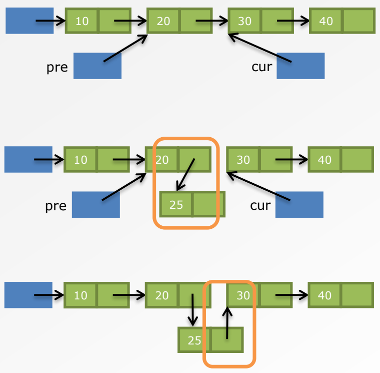
I have created the new node and assigned its address to the pre-> next using malloc(). It created the link between the second node and the new third node.

Now, pre pointer points at node 20 and pre->next pointer points at node 25. Therefore, pre-

>next->next should point at node 30. The address of the node 30 is held by the cur pointer. Therefore pre->next->next should equal to 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

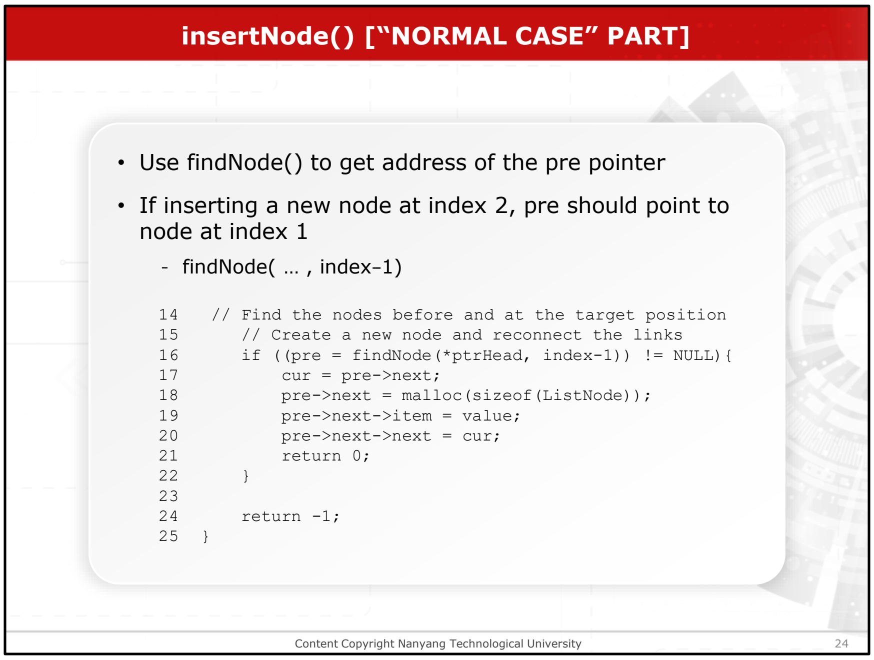
```
Pre->next = malloc(sizeof(ListNode));
```

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
```

23

Since we know that current node is next to the previous node, we can use pre-> next to find the current node and assign that address to cur pointer. The rest of the code is just a combination of the lines of code that I showed you on the previous slide.



insertNode() ["NORMAL CASE" PART]

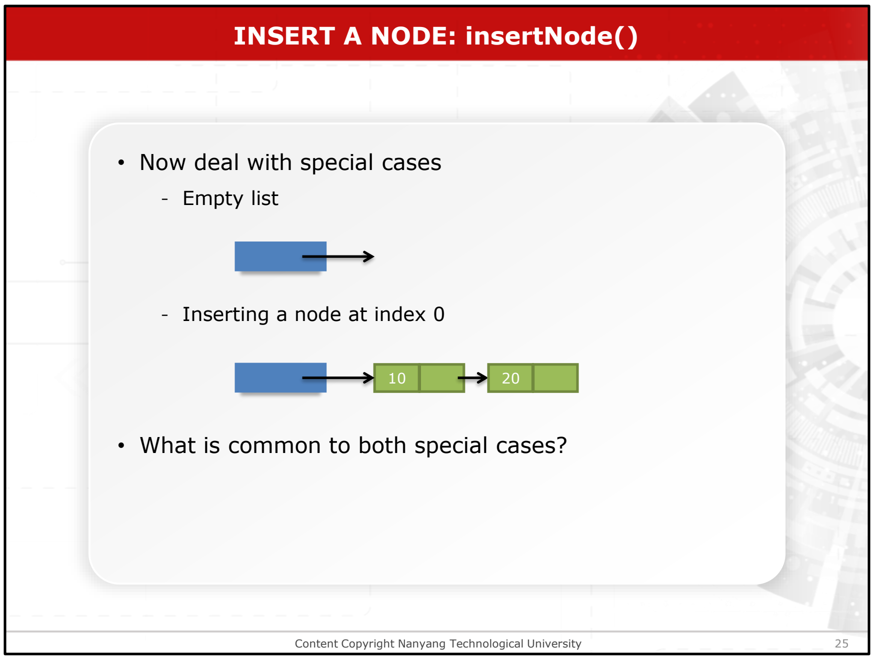
- Use findNode() to get address of the pre pointer
- If inserting a new node at index 2, pre should point to node at index 1
  - findNode(… , index-1)

```
  // 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;
 }
```

24

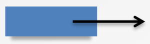
As we discussed previously, the findNode function will be used to find the location on which we need to insert the new node. Therefore, here we use the findNode function to get the address for the pre pointer. So if we are inserting at index 2, pre pointer should be pointing at the node in index 1. Therefore, we use the findNode function to get to index−1 and set that address to pre-pointer.

We also need to find the current node to point to the cur pointer. But, running findNode twice is inefficient since it needs to start from the beginning of the list and traverse down.

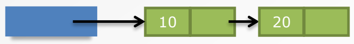


INSERT A NODE: insertNode()

- Now deal with special cases
  - Empty list



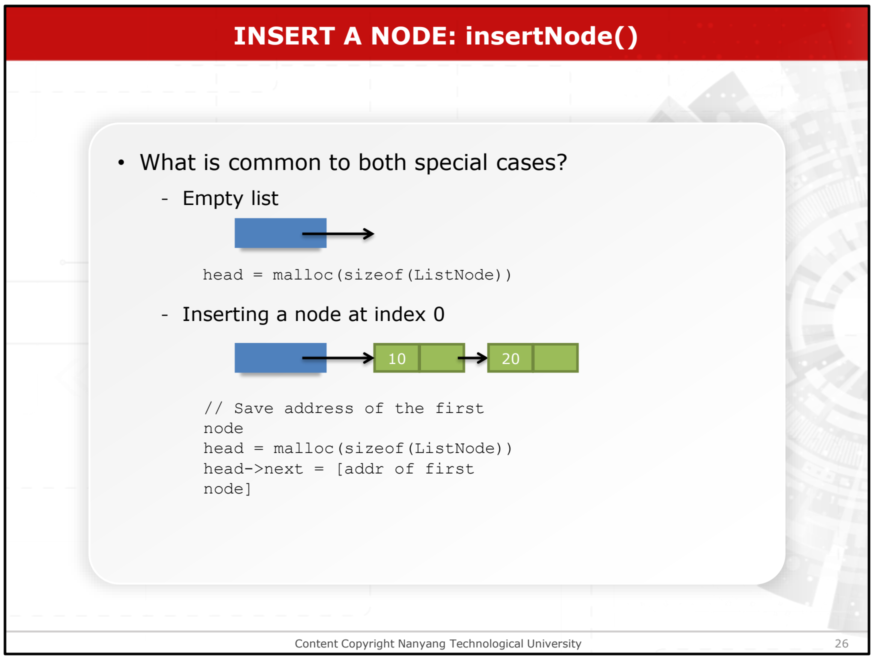
  - Inserting a node at index 0



- What is common to both special cases?

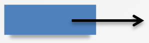
25

Special cases of insertNode are inserting a node to an empty list and inserting a node at index 0 of a list. What is common to these special cases?



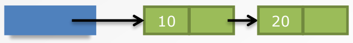
INSERT A NODE: insertNode()

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



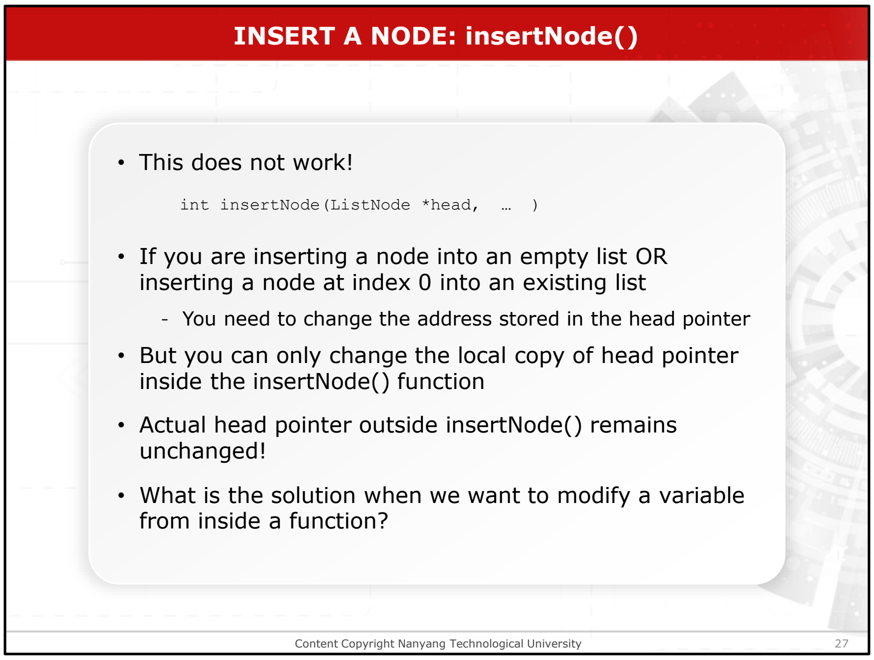
```
head = malloc(sizeof(ListNode))
```

  - Inserting a node at index 0



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

To insert a node to an empty list, you have to create a new node using malloc() and assign it to the head pointer.
To insert a node at index 0 of a list, we need to first save the address of the current first node in a temporary pointer. Then we create the new node using malloc and assign its address to the head pointer. The value of the temporary pointer will be assigned to head->next.



INSERT A NODE: insertNode()

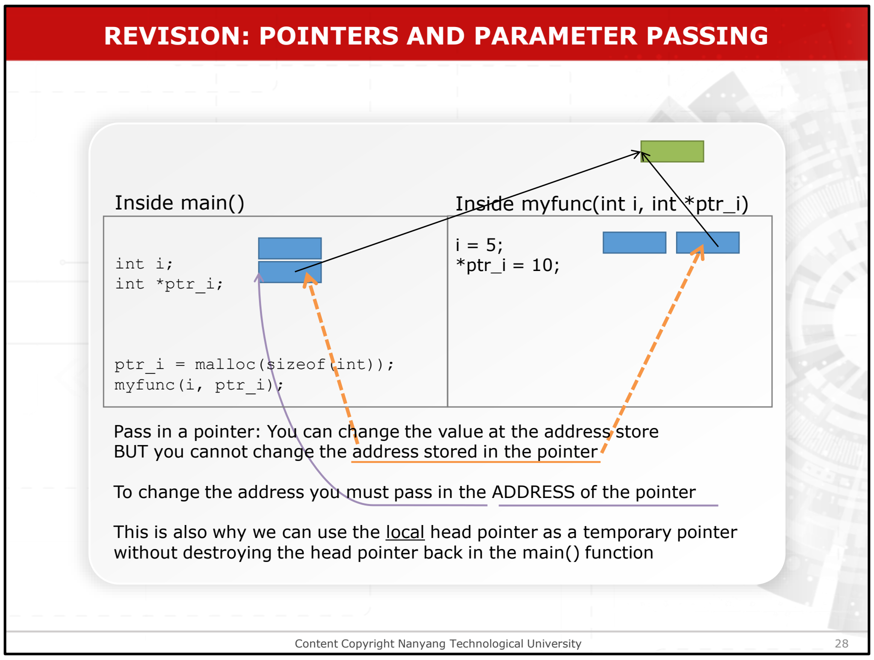
- This does not work!

```
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?

27

The answer is NO. Because when we insert a node to an empty list of at index 0 of a list, we need to change the value of the head pointer. Still, you can only change the local copy of the head pointer inside the function. The actual head pointer remains unchanged.



REVISION: POINTERS AND PARAMETER PASSING

Inside main()
Inside myfunc(int i, int *ptr_i)

```
int i;
int *ptr_i;

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

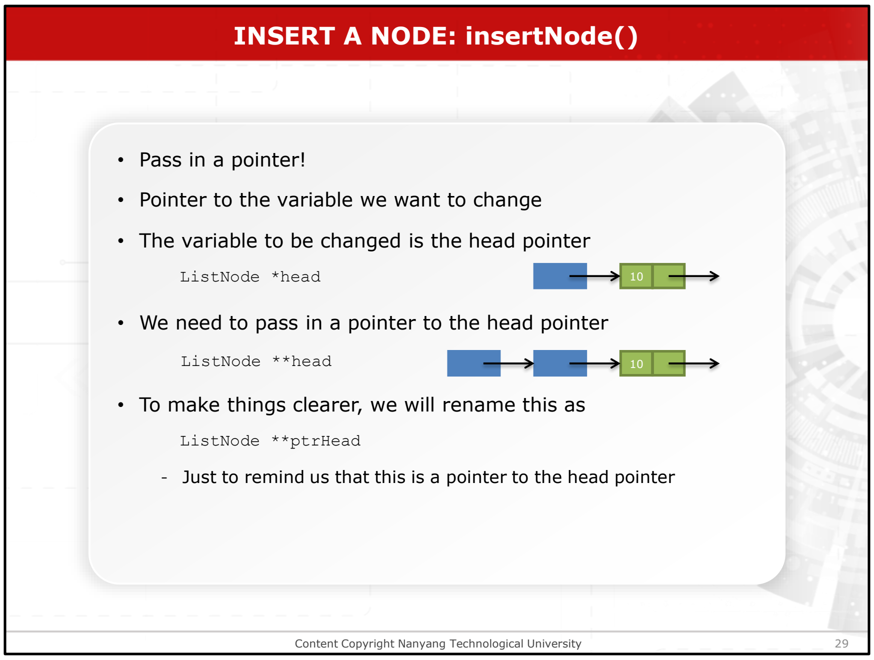
This is also why we can use the local head pointer as a temporary pointer without destroying the head pointer back in the main() function

28

For example, I declare integer i and integer *ptr_i. Then I declare myfunc() passing these two integer variables i and ptr_i. Let me quickly show you this. I have a pointer, and this is int star pointer i.

There is an address stored inside the pointer so pointer i has the value which is the address of space in memory out there, and that's passed into the function. So inside the function, I can actually change the value of whatever is stored in this space in memory. But I cannot change the actual value of the pointer variable.

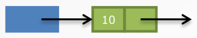
To change it, we need to pass the address of local pointer variable by reference.



INSERT A NODE: insertNode()

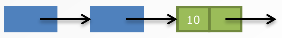
- Pass in a pointer!
- Pointer to the variable we want to change
- The variable to be changed is the head pointer

```
ListNode *head
```



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

```
ListNode **head
```



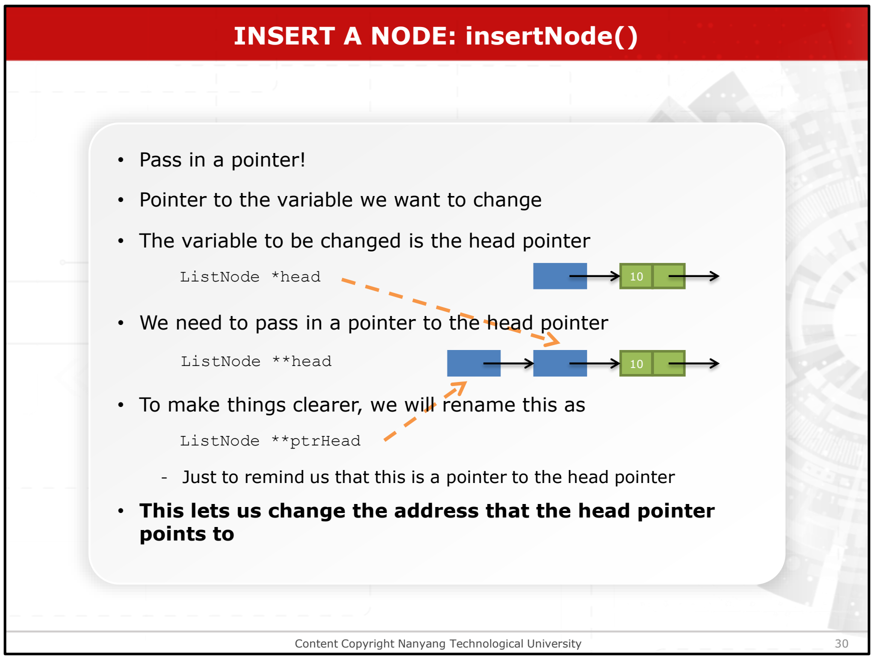
- To make things clearer, we will rename this as

```
ListNode **ptrHead
```

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

29

I need to pass in a pointer to the variable that I want to change which is the head pointer. Therefore, instead of passing in the head pointer, we pass a pointer which points at 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
```

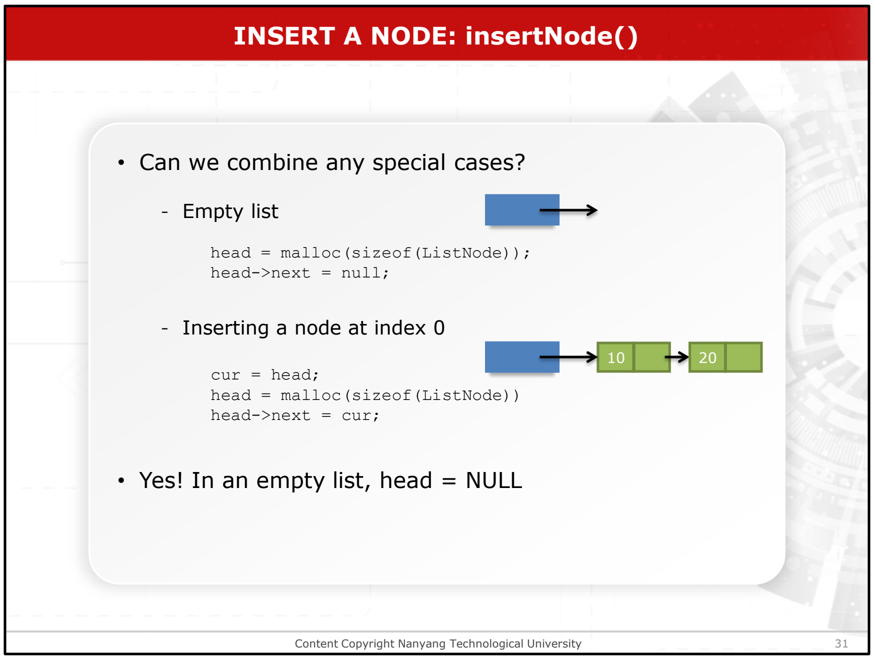
- To make things clearer, we will rename this as

```
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**

Content Copyright Nanyang Technological University 30

Head pointer points to the first node while the pointer to the head pointer points to the head pointer.



INSERT A NODE: insertNode()

- Can we combine any special cases?
  - Empty list

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

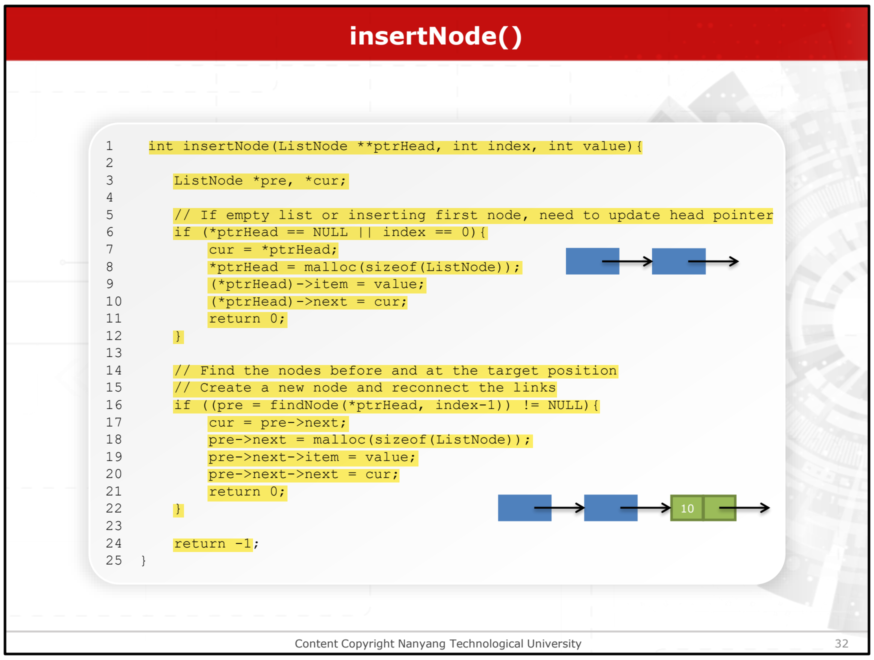
  - Inserting a node at index 0

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

- Yes! In an empty list, head = NULL

Finally, we can combine the special cases we discussed. For an empty list, head->next = 0. When inserting a node at the index 0, cur= head since it has the address of the first node. Then we create a new node and assign its address to head. Now, head->next=cur because the head->next is the original first node.

When the list is empty, head=NULL, therefore cur=NULL and head->next=cur=NULL. We can then combine these two cases with similar code chunks.

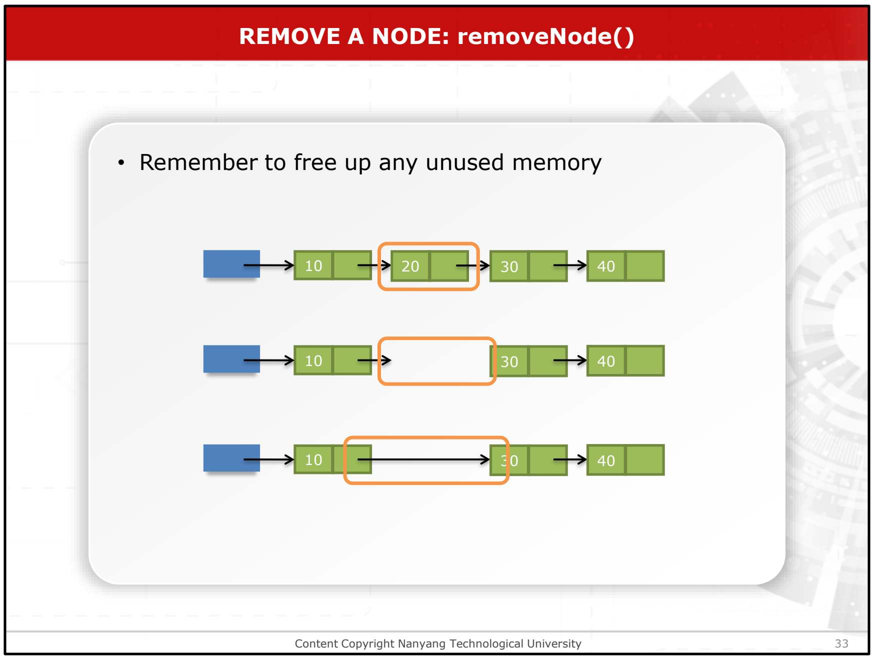


insertNode()

```
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;
}
```

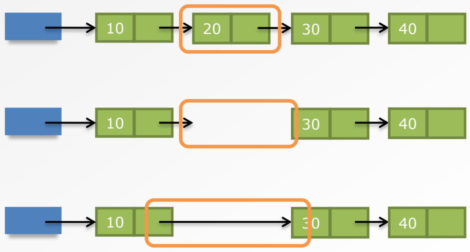
32

We can put all we learned from insertNode like this to form the complete function.



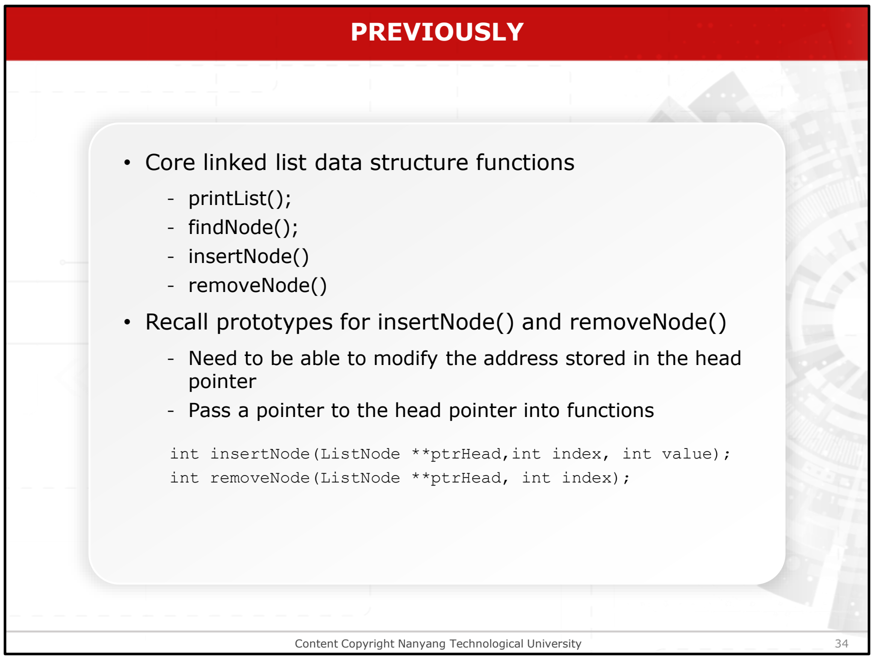
REMOVE A NODE: removeNode()

- Remember to free up any unused memory



33

Remember to free up any unused memory.



PREVIOUSLY

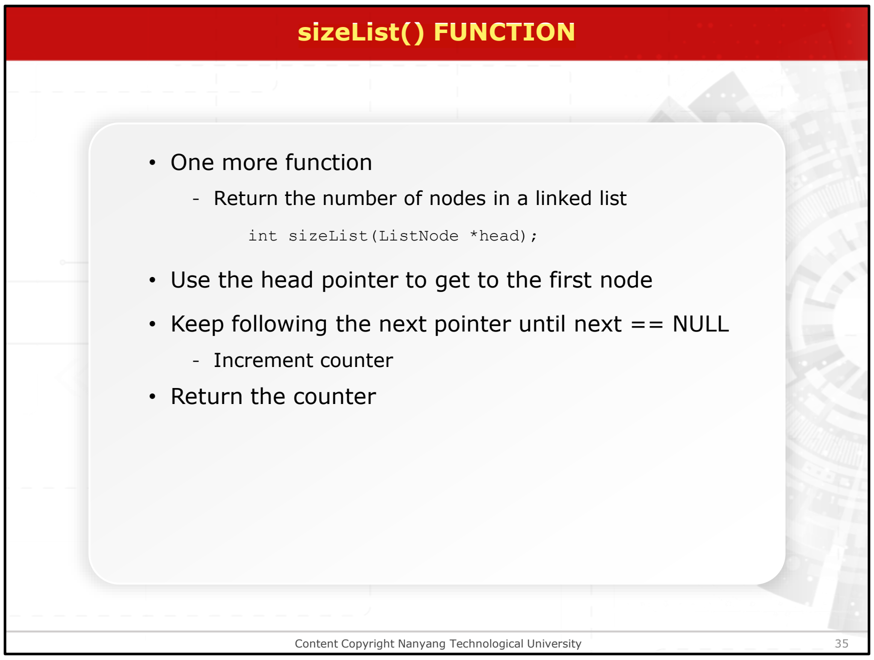
- Core linked list data structure functions
  - printList();
  - findNode();
  - insertNode()
  - removeNode()
- Recall prototypes for insertNode() and removeNode()
  - Need to be able to modify the address stored in the head pointer
  - Pass a pointer to the head pointer into functions

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

34

Earlier we learned about four main functions. The printList function prints every item that is stored in a linked list. The findNode function takes the index value and returns you a pointer to the node you are looking for. The insertNode function allows you to add a new value to anywhere in the linked list. You can try the RemoveNode function in the lab.

Now, in order to modify the address stored in the head pointer, we need to pass a pointer to the head pointer. This is ListNode ** pointer. This makes things complicated, and when you write the code, you have to think whether you should dereference it once or twice, or whether it is a direct pointer, etc. Therefore, we need to make things clear.



sizeList() FUNCTION

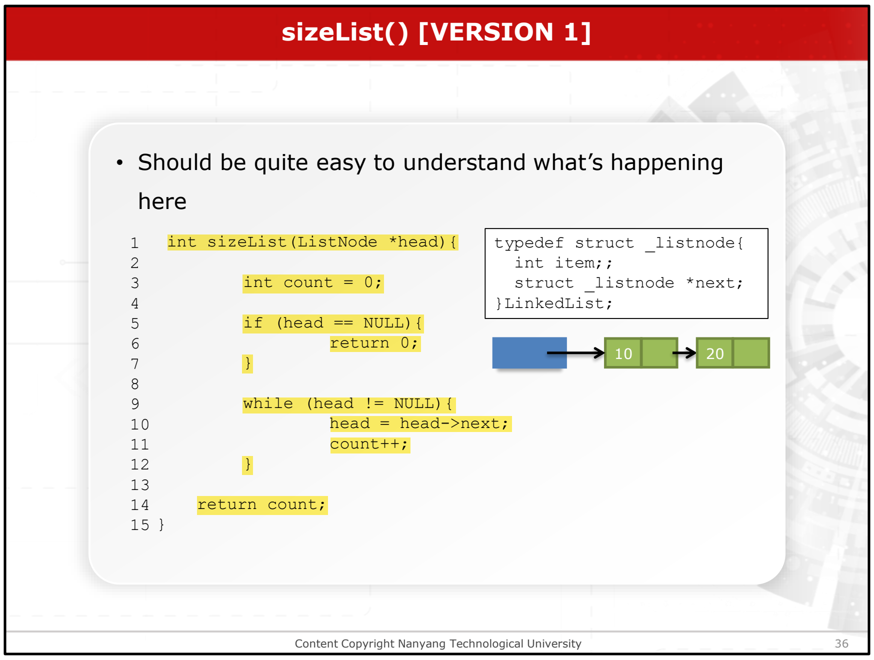
- One more function
  - Return the number of nodes in a linked list

```
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

35

The sizeList function is really simple. You have to return the number of nodes in the linked list. Use the head pointer and keep following the next pointer until you hit the next that equals to NULL. Every time you move one step down, you should increment the counter by 1. This is very similar to the recursive function on counting digits.

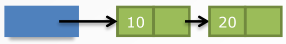


sizeList() [VERSION 1]

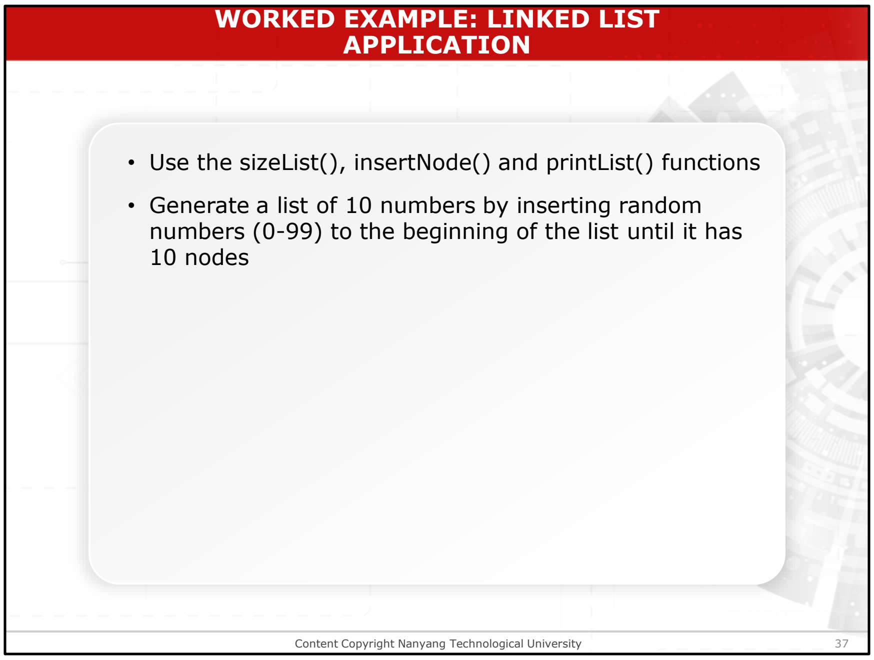
- Should be quite easy to understand what's happening here

```
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;
```



This is the code for the sizeList function. Note that in the code, head is a local variable which initially holds the value of the head pointer, that is the address of the first node. We can use this as a temporary variable. You will see, shortly, the cases that do not allow you to do this.

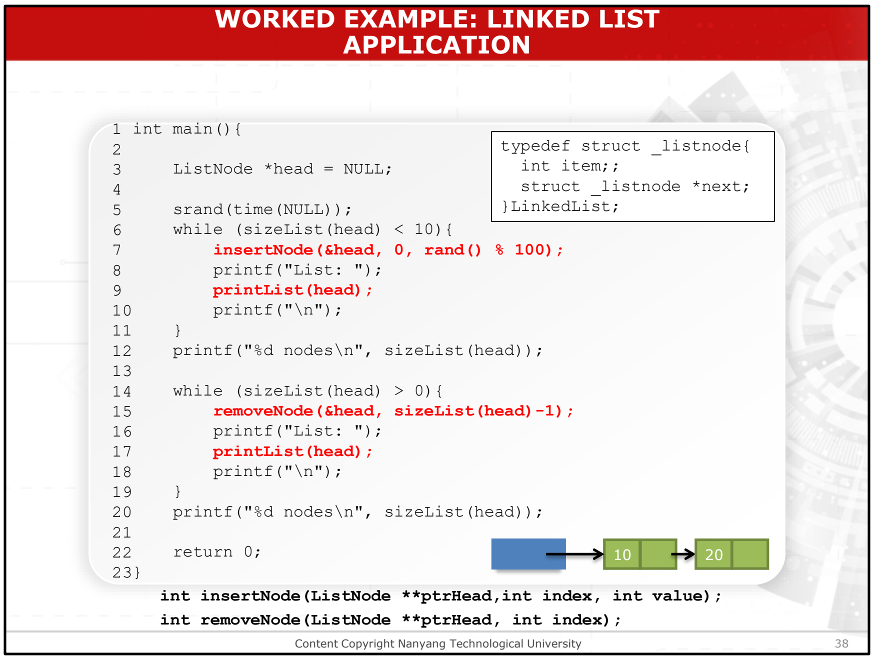


WORKED EXAMPLE: LINKED LIST APPLICATION

- 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

37

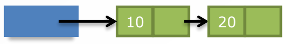
In this application, I want to add values to the linked list until I have a certain number of values inside. We will start inserting random values from the beginning of the linked list. So, we have a while loop which runs while the sizeList(head) is less than 10. I pass in the head pointer, and every time I go through this function and I decide whether to continue, by checking the number of nodes inside the linked list. So, as long as I don't have a certain number of nodes in my list, I will keep inserting a new number.



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;
}
```

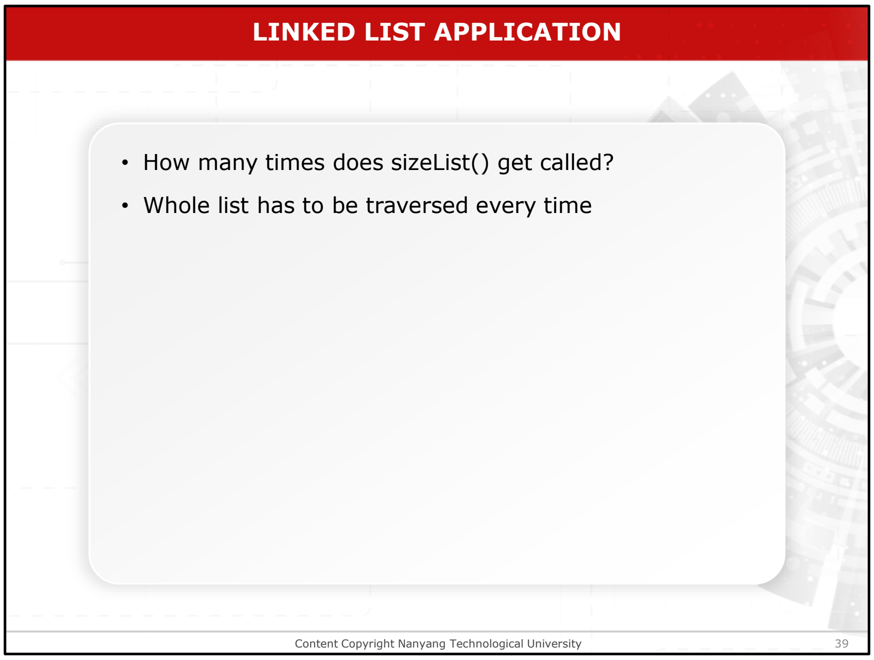
```
typedef struct _listnode{
  int item;;
  struct _listnode *next;
}LinkedList;
```



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

38

In this application, I want to add values to the linked list until I have a certain number of values inside. We will start inserting random values from the beginning of the linked list. So, we have a while loop which runs while the sizeList(head) is less than 10. I pass in the head pointer, and every time I go through this function and I decide whether to continue, by checking the number of nodes inside the linked list. So, as long as I don't have a certain number of nodes in my list, I will keep inserting a new number.

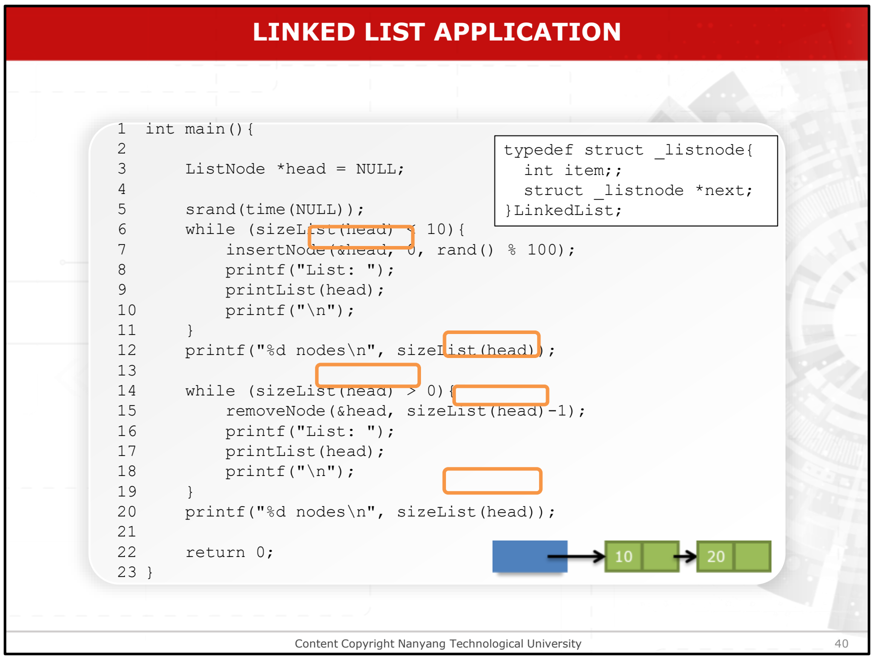


LINKED LIST APPLICATION

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

39

Now, what I want you to think about is how many times does thr size list function get called? Here, we call sizeList function five times. Every time I call sizeList function, it traverses all the way down and increments the counter to figure out how many nodes there are in the linked list. But it is a waste of time to traverse the entire linked list many times.



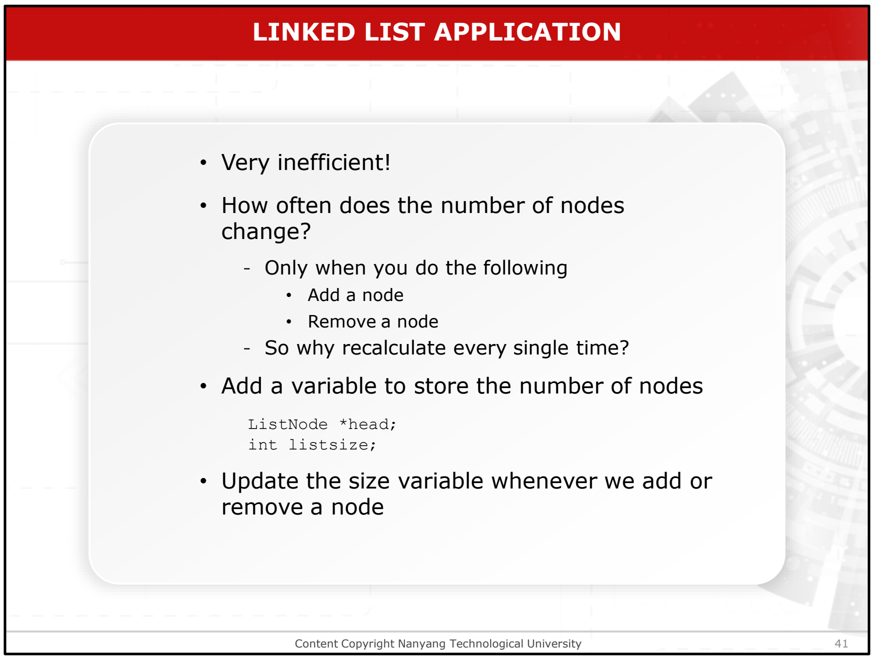
LINKED LIST APPLICATION

```
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;
```

40

Now, what I want you to think about is how many times does thr size list function get called? Here, we call sizeList function five times. Every time I call sizeList function, it traverses all the way down and increments the counter to figure out how many nodes there are in the linked list. But it is a waste of time to traverse the entire linked list many times.



LINKED LIST APPLICATION

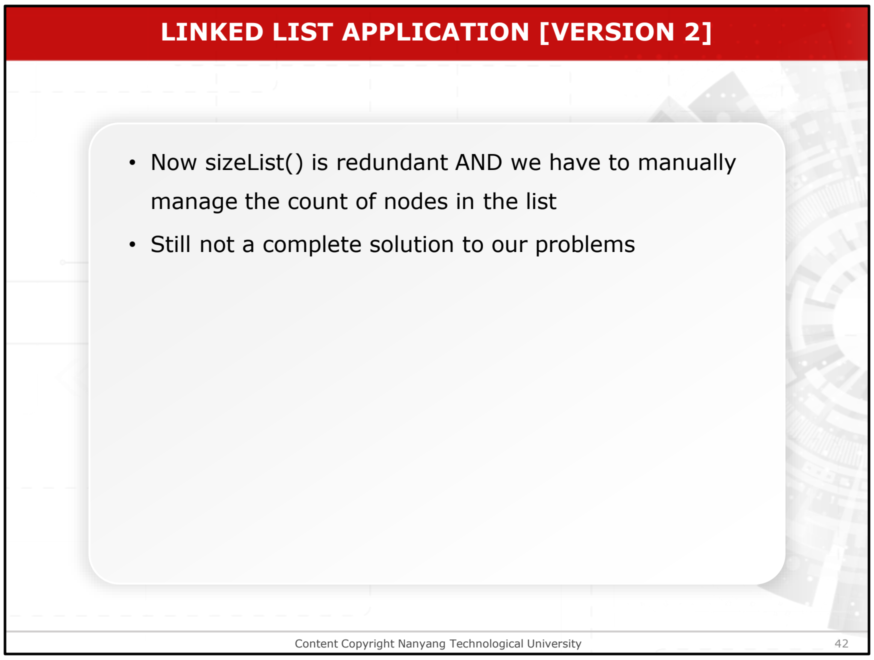
- Very inefficient!
- How often does the number of nodes change?
  - Only when you do the following
    - Add a node
    - Remove a node
  - So why recalculate every single time?
- Add a variable to store the number of nodes

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

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

41

When we add a node, the size of the list goes up by 1. When we remove a node, it goes down by 1. When I print a list or run findNode function, the size of the list won't change. Therefore, we do not want to bother about the size of the linked list each time. I create a separate variable, ''listsize", that will keep track of the size of the linked list. Every time we insert a node, the value of the listsize will increase by 1, and it will decrease by 1 every time we remove a node. The starting value of the listsize is 0 as we start with an empty list.



LINKED LIST APPLICATION [VERSION 2]

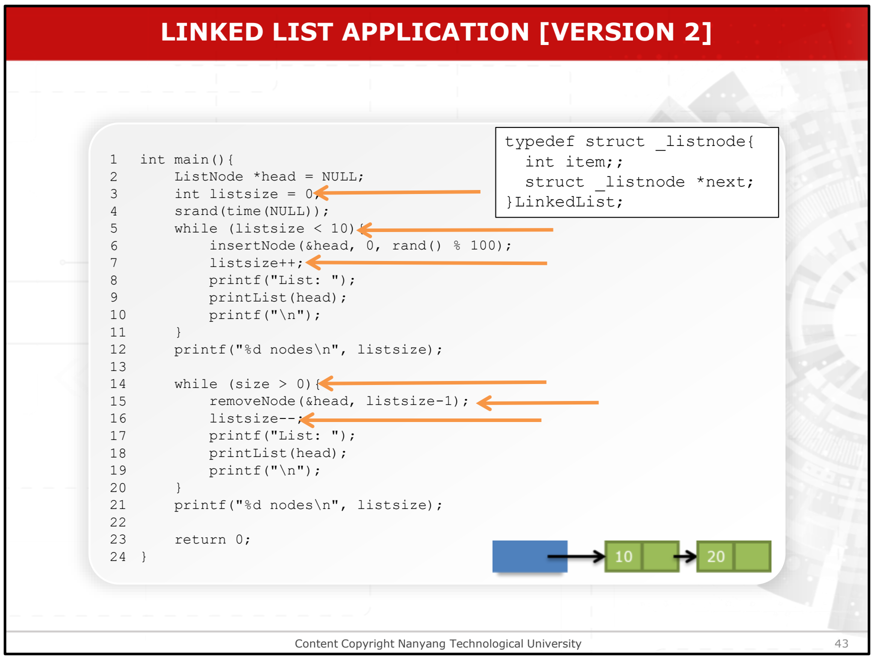
- Now sizeList() is redundant AND we have to manually manage the count of nodes in the list
- Still not a complete solution to our problems

Content Copyright Nanyang Technological University 42

Previously, we called sizeList function several times. But we will now update the listsize variable instead.

Although this is more efficient than the previous version of the code, it is not good for you as a programmer because now you have to keep track of both the head pointer and the listsize variable. You have to make sure that the listsize variable is updated every time you insert or remove a node.

If you have an array of linked lists, you should have an array of listsize values, and make sure that you access the correct one when needed. Now, this is also inefficient. One of the reasons why we learned C struct is that we need to wrap up the things that are related. Therefore, we will apply the same thing here.



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;
```

43

Previously, we called sizeList function several times. But we will now update the listsize variable instead.

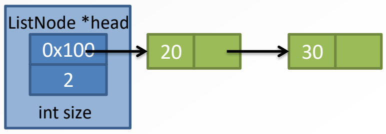
Although this is more efficient than the previous version of the code, it is not good for you as a programmer because now you have to keep track of both the head pointer and the listsize variable. You have to make sure that the listsize variable is updated every time you insert or remove a node.

If you have an array of linked lists, you should have an array of listsize values, and make sure that you access the correct one when needed. Now, this is also inefficient. One of the reasons why we learned C struct is that we need to wrap up the things that are related. Therefore, we will apply the same thing here.

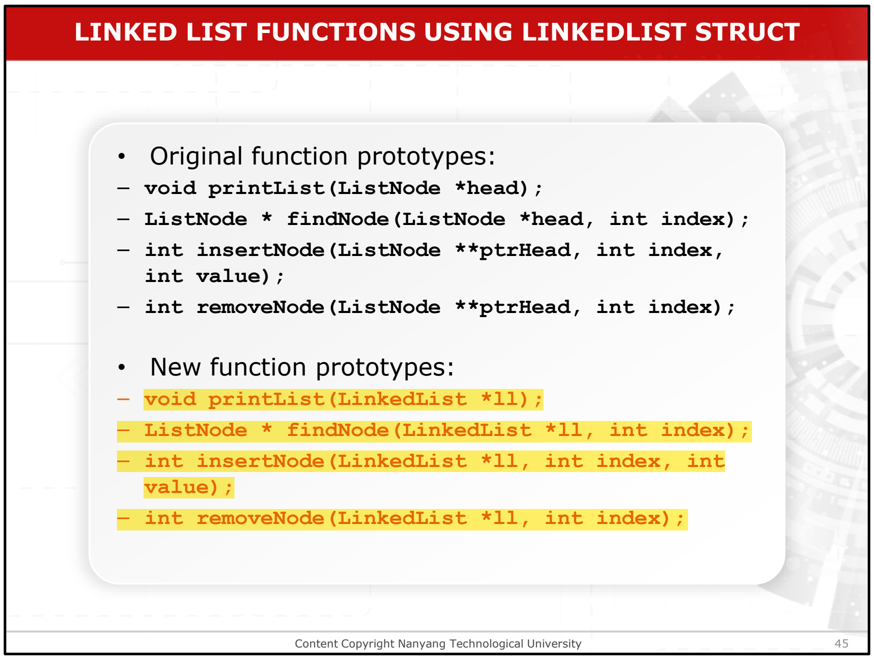
LINKEDLIST C STRUCT

- 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;
```



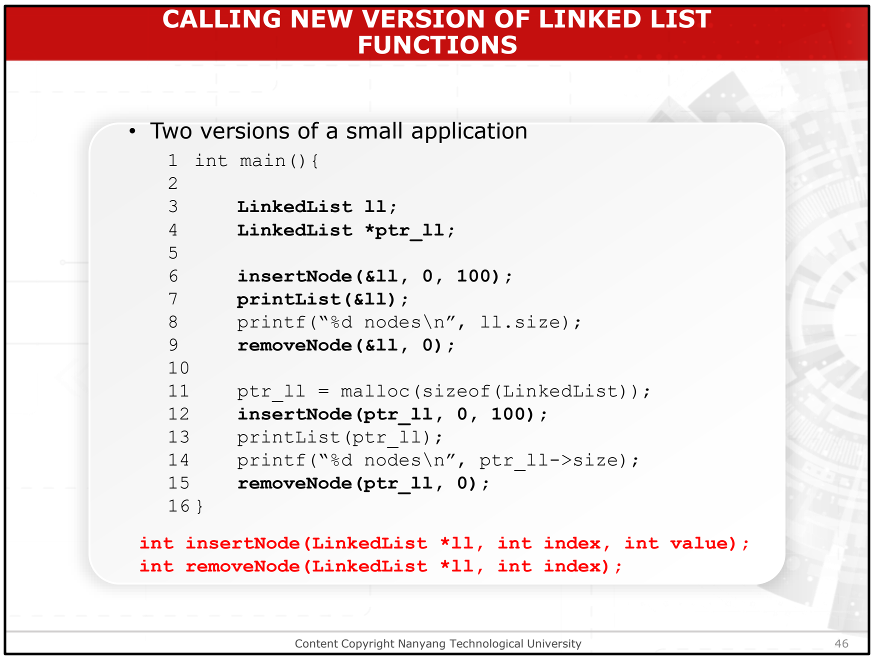
- Why is this useful?
  - Consider the rewritten Linked List functions



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

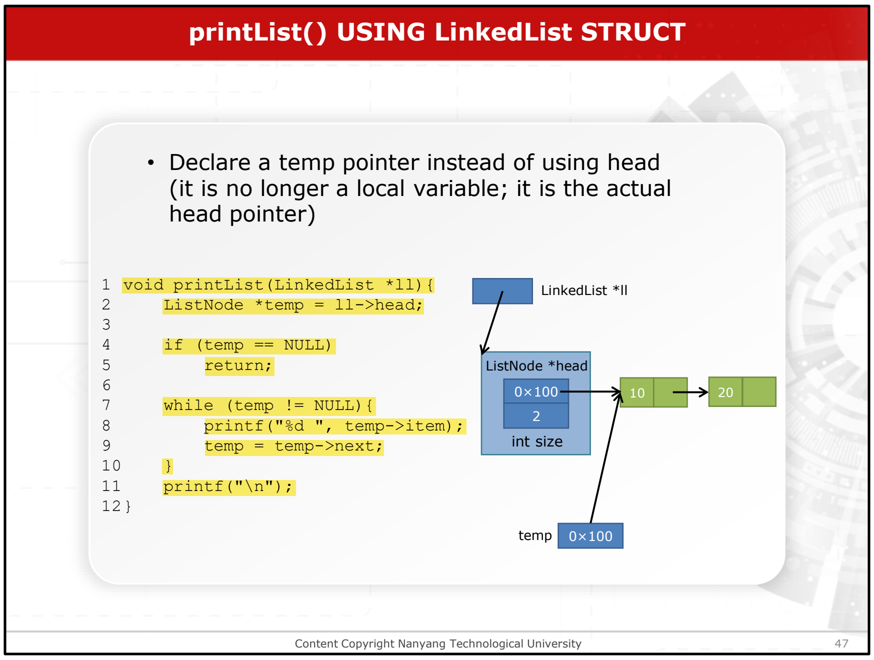
```
1  int main(){
2
3      LinkedList ll;
4      LinkedList *ptr_ll;
5
6      insertNode(&ll, 0, 100);
7      printList(&ll);
8      printf("%d nodes\n", ll.size);
9      removeNode(&ll, 0);
10
11     ptr_ll = malloc(sizeof(LinkedList));
12     insertNode(ptr_ll, 0, 100);
13     printList(ptr_ll);
14     printf("%d nodes\n", ptr_ll->size);
15     removeNode(ptr_ll, 0);
16 }

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

46

Now let’s see how we use the linked list struct with both dynamic and static memory allocation. In here, ll is a proper linked list struct which has two fields and takes up a certain amount of memory. This is a statically declared linked list struct.

In line 4, we declared pointer ll, a linked list object from the heap using malloc. From line 6 to 9, we use the statically declared linked list struct. We pass in a pointer to the C structure so we can print, insert and remove nodes. But, since we have a linked list struct now, we can also use the malloc to create a linked list structure dynamically. Now, I am going to call malloc and get enough memory to hold the linked list struct. This linked list struct keeps track of the list nodes and size of the linked list. If you are dealing with a struct itself, the notation is a dot. But if you are accessing a struct using a pointer, then you should use the arrow notation to get inside the variables.

In line 11, we ask for enough space to hold the linked list struct. I assigned that memory location to pointer ll. And subsequently, I will perform the same functions as inset, print the list, print the size of the list and remove.



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");
}
```

47

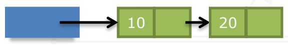
Previously, we passed in a head pointer for printList function, but now we are passing in a linked list struct. In previous parameter list, we used the head as a local pointer variable which gets a copy of the value outside from the head pointer.

If I use ll-> head as the temporary pointer, I am destroying the structure of the linked list itself because I am keep changing the value of the pointer to go down the linked list.

So I need to create the *temp as a temporary pointer just for the function. Now we can use this local variable as a temp pointer. We can use it to go down the list nodes. The outside head pointer points at the actual first node even though we use the temp pointer to move down the list.

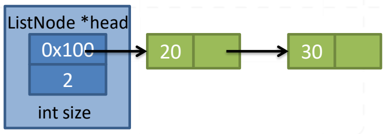
printList() Versions

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

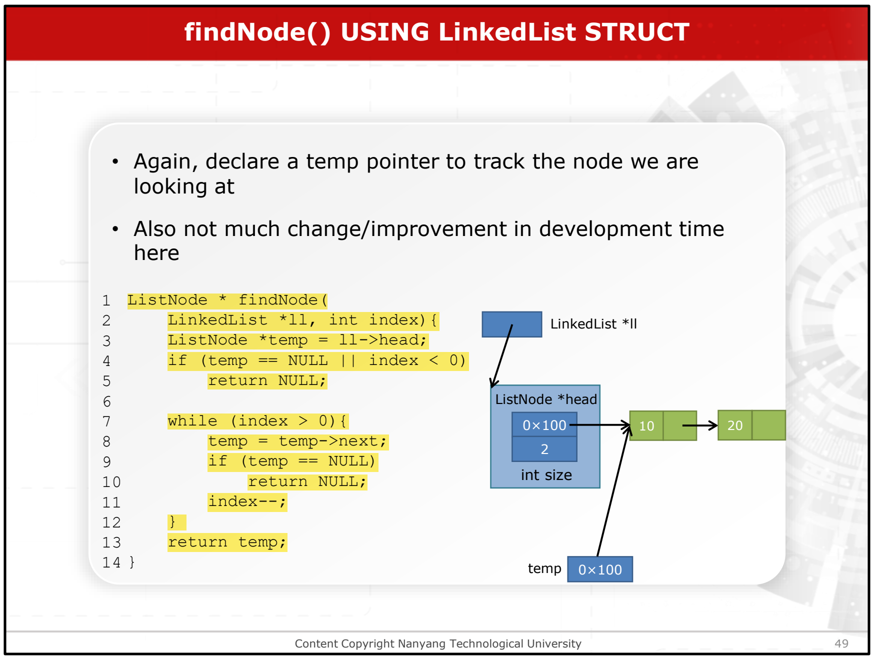


```
void printList(ListNode *head){

    if (head == NULL)
        return;

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

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



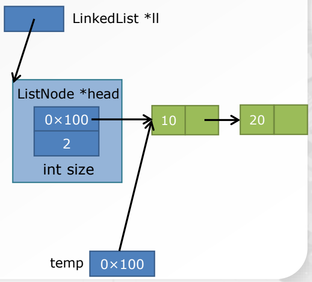
```
void printList(LinkedList *ll){
    ListNode *temp = ll->head;

    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;
}
```

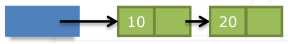


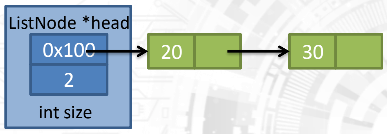
In findNode function, we cannot use the head pointer as the temporary pointer. Therefore, I have declared listNode * temp. The rest of the function is similar to the previous version.

findNode() Versions

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

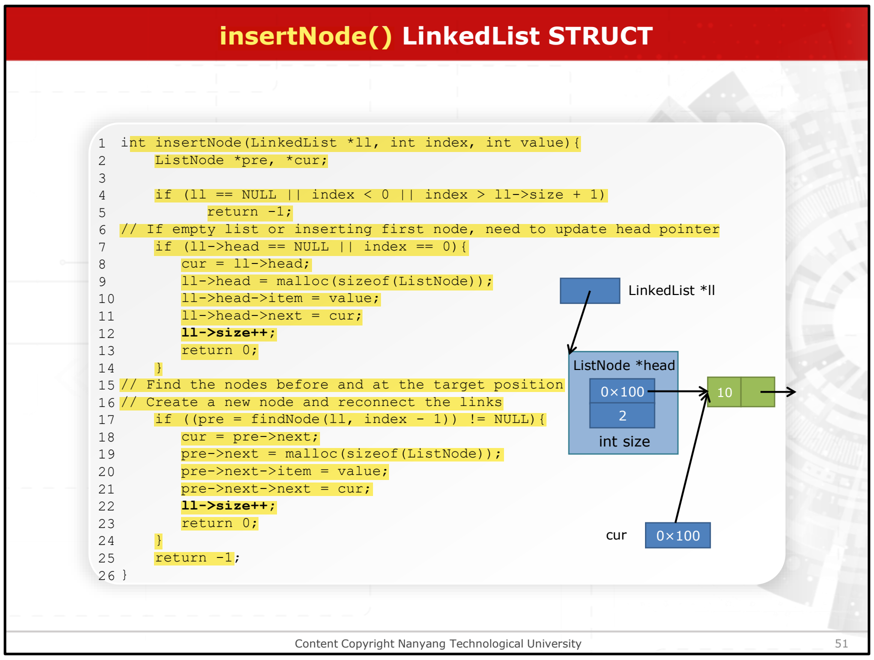
```
typedef struct _linkedlist{
   int size;
   ListNode *head;
}LinkedList;
```





```
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;
}
```

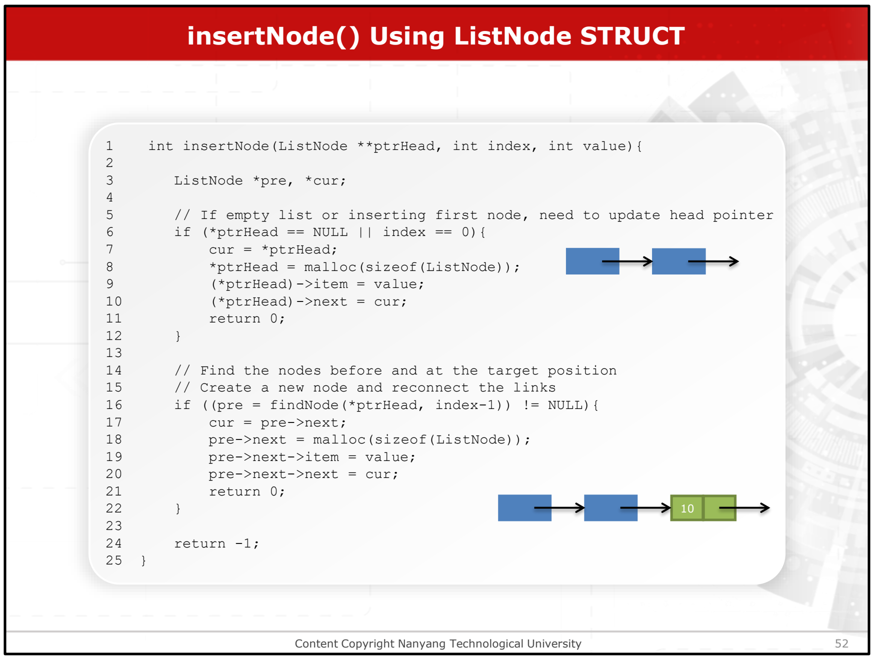


insertNode() LinkedList STRUCT

```
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;
}
```

51

We can put all we learned from insertNode like this to form the complete function.

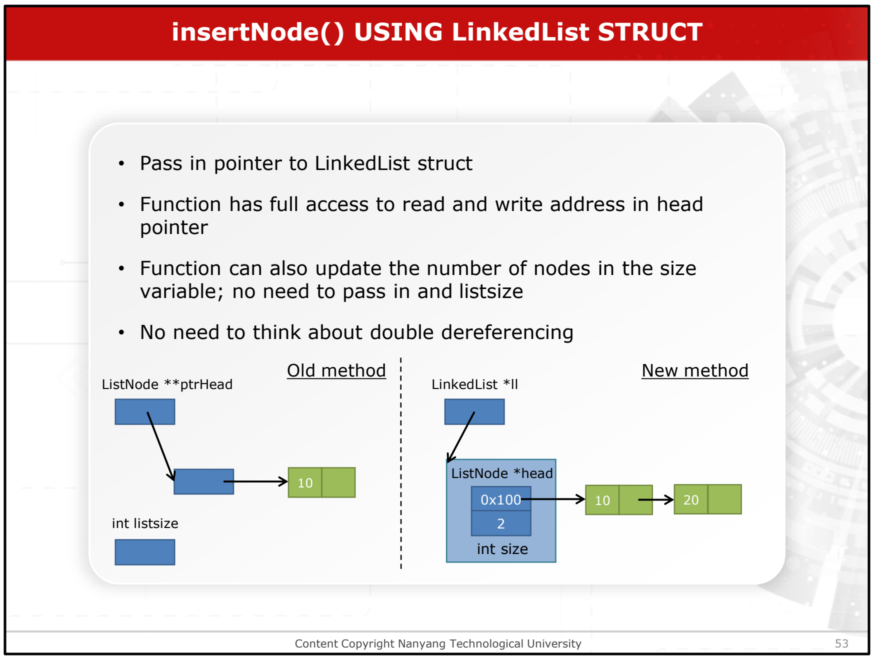


insertNode() Using ListNode STRUCT

```
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;
}
```

52

We can put all we learned from insertNode like this to form the complete function.



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

Old method
ListNode **ptrHead
10
int listsize
New method
LinkedList *ll
ListNode *head
0x100
2
int size
10
20

53

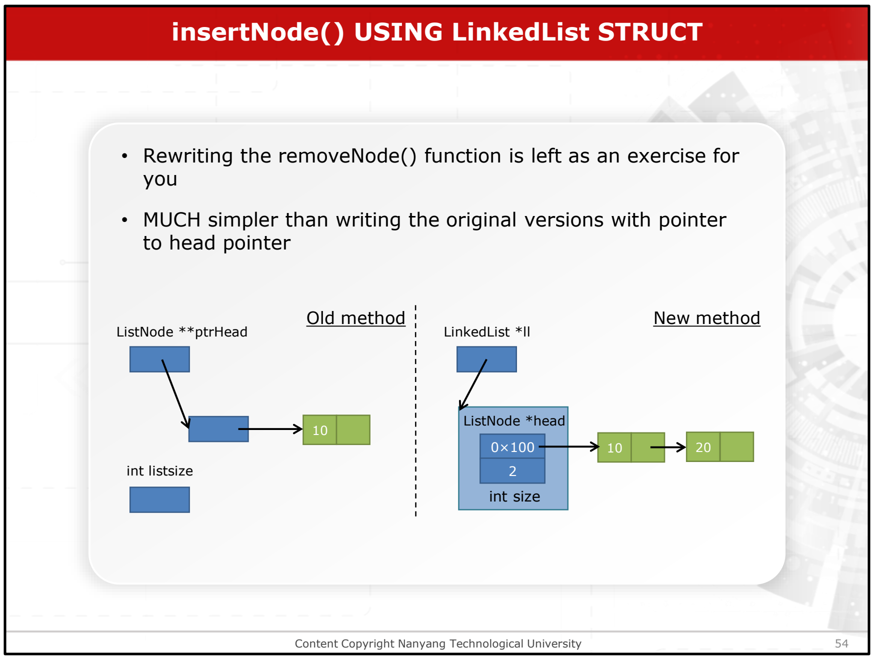
In the old method of inserting node, we have added the listsize variable to keep track of the size of the linked list.

Now, what if I make a mistake when updating the value of the listsize, or if someone else uses the insert node function and modify your linked list instead?

To take care of such a problem, we need to pass in the listsize variable to the insert node function as a parameter.

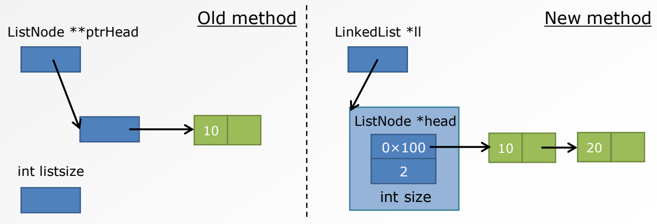
That is the old method. In the new version, instead of worrying about dereferencing the correct things, we just need to pass in the single list node pointer ll. Once we have access to the linked list struct, we have access to the head pointer and the list size variable.

By doing that, you do not need to change the parameter list for the insert node or remove node functions.



insertNode() USING LinkedList STRUCT

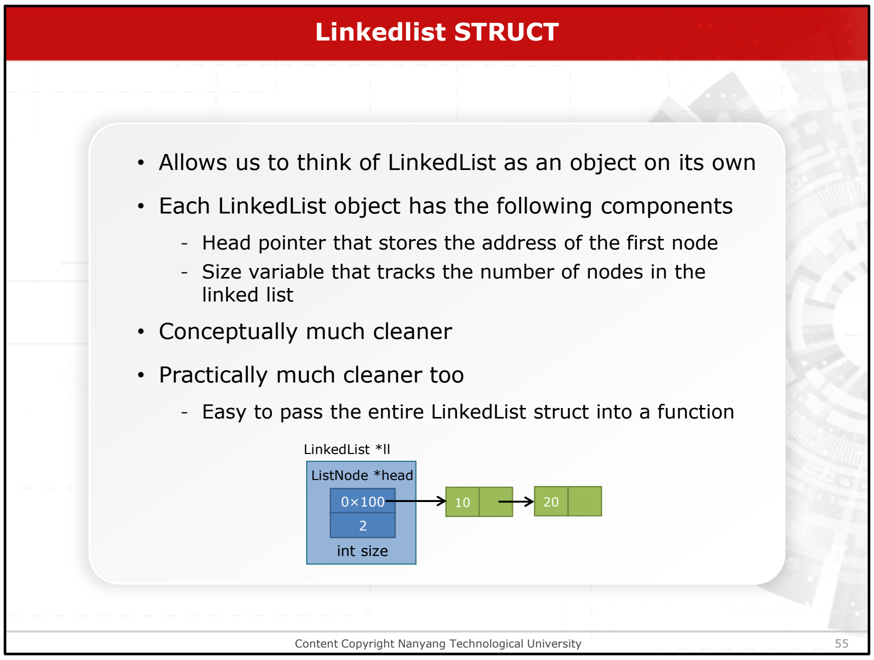
- Rewriting the removeNode() function is left as an exercise for you
- MUCH simpler than writing the original versions with pointer to head pointer



Content Copyright Nanyang Technological University 54

Now you can rewrite the functions with the linked list struct.

I will provide you with the reference code in a week. You can compare it with my code after trying it out yourself.



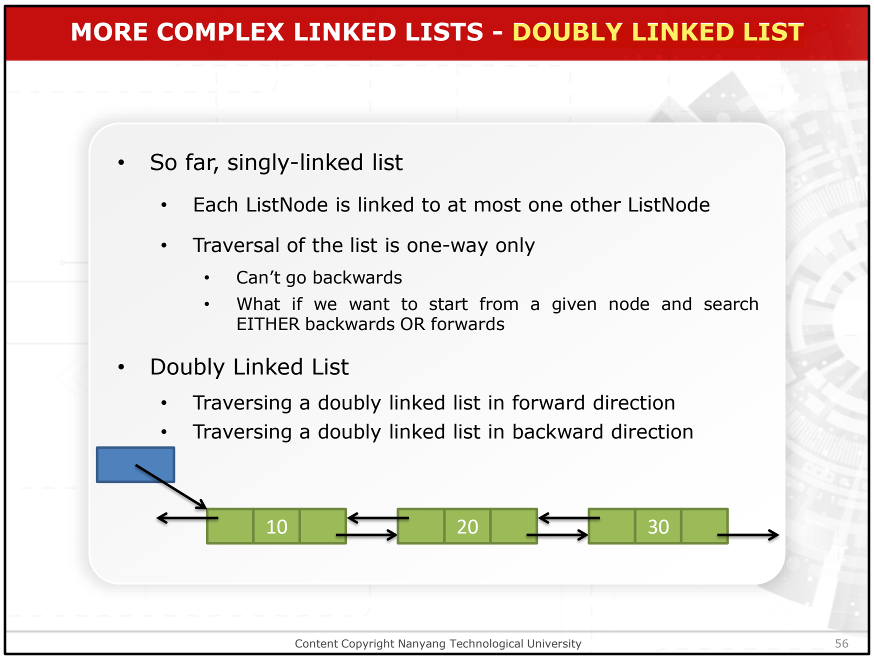
Linkedlist STRUCT

- 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

LinkedList *ll
ListNode *head
0×100
2
int size
10
20

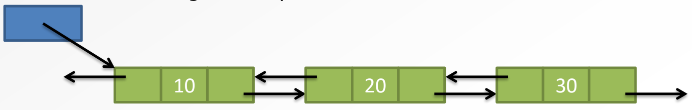
The reason why we have these structures in C is that each object represents a certain concept.

Likewise, now the linked list struct allows us to treat the entire linked list as an object, allowing us to easily create an array of the linked list. We can easily pass in the linked list struct to other functions with a pointer to the linked list struct.



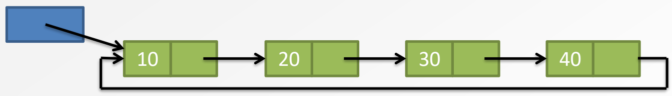
MORE COMPLEX LINKED LISTS - DOUBLY LINKED LIST

- 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

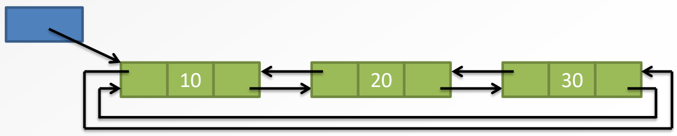


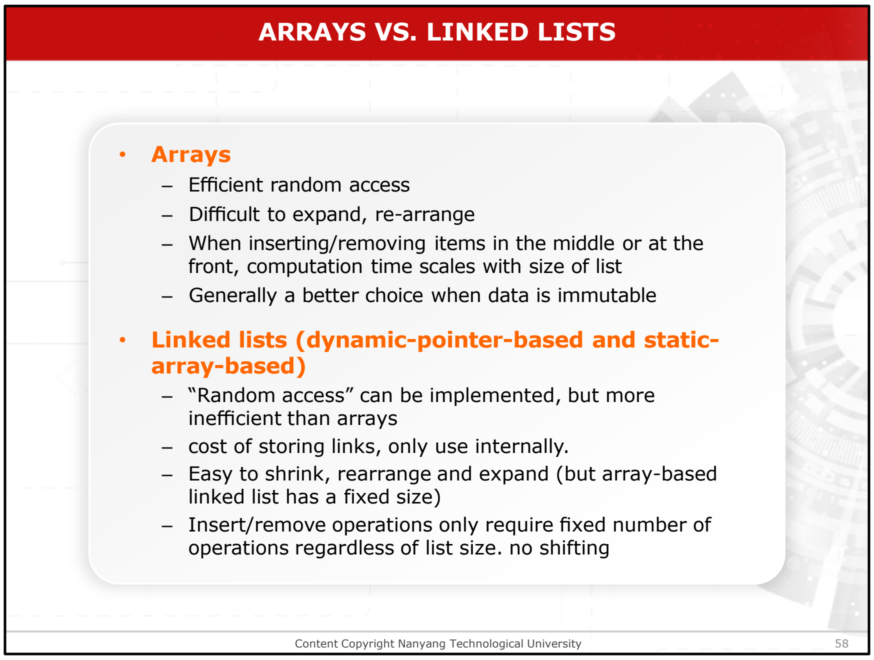
MORE COMPLEX LINKED LISTS - CIRCULAR LINKED LISTS

- 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!!