NANYANG TECHNOLOGICAL UNIVERSITY SINGAPORE

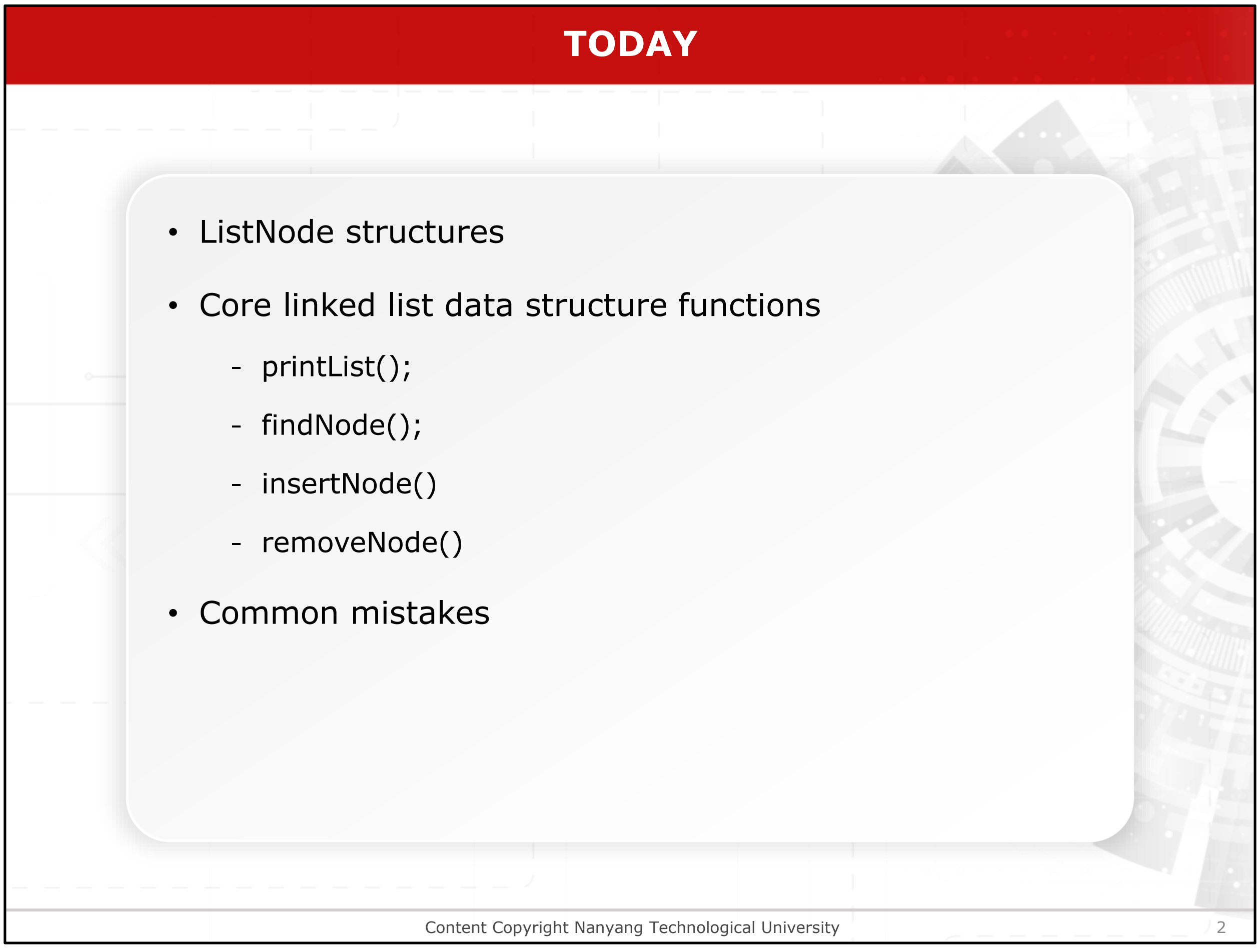
CE1007 DATA STRUCTURES

Lecture 3: **Linked List Functions**

College of Engineering
School of Computer Science and Engineering

Since calling malloc every time we are allocating space for a new list node is troublesome, today we will discuss on linked list functions which will help us to interact with linked list more effectively.

Functions can be used to avoid running the same piece of code or action over and over. They can be improved to act more general and to handle more cases.

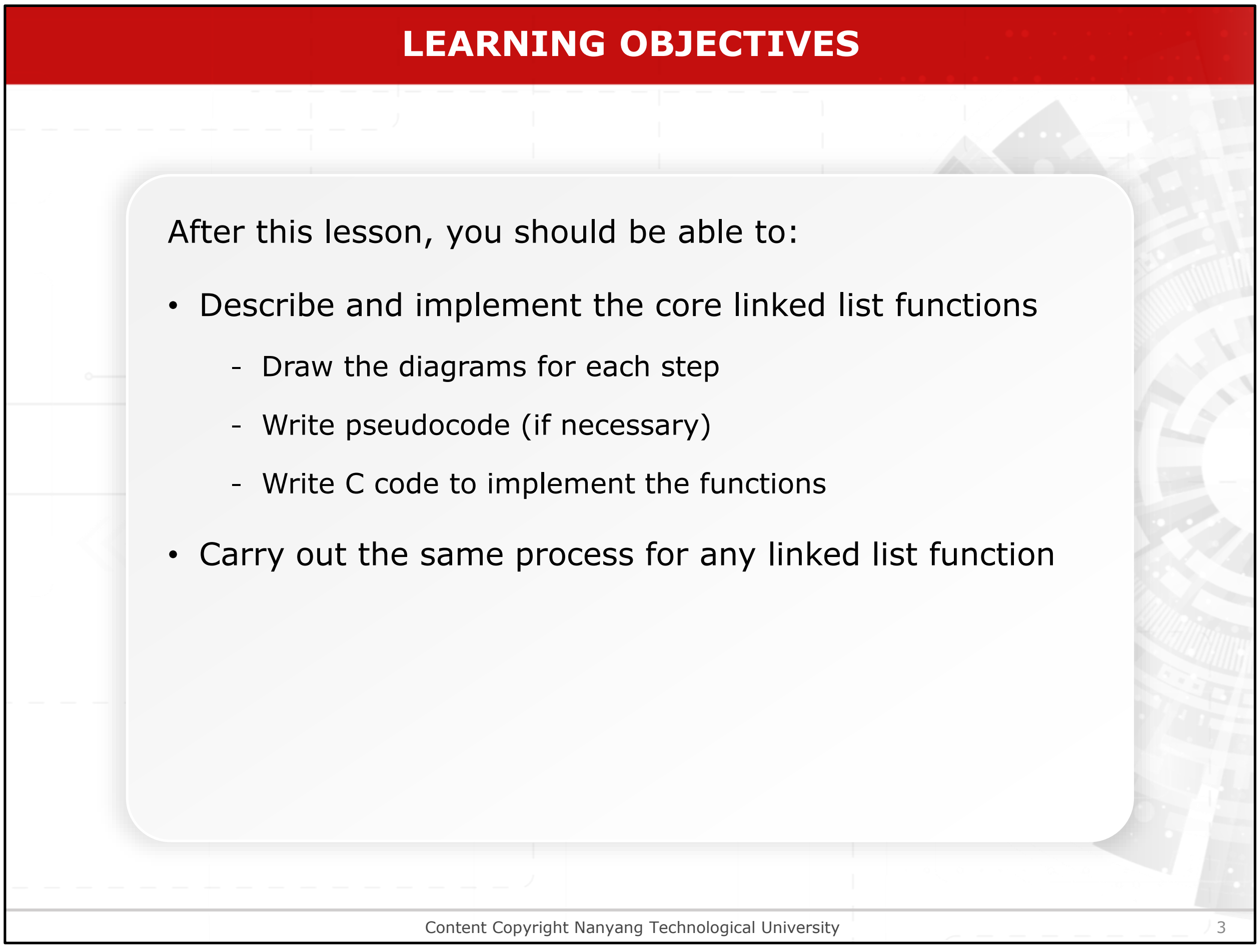


TODAY

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

2

Today we will learn about these 4 functions. The 4th function: removeNode() will be a part of your lab session. Therefore, we will focus on the first 3 functions.



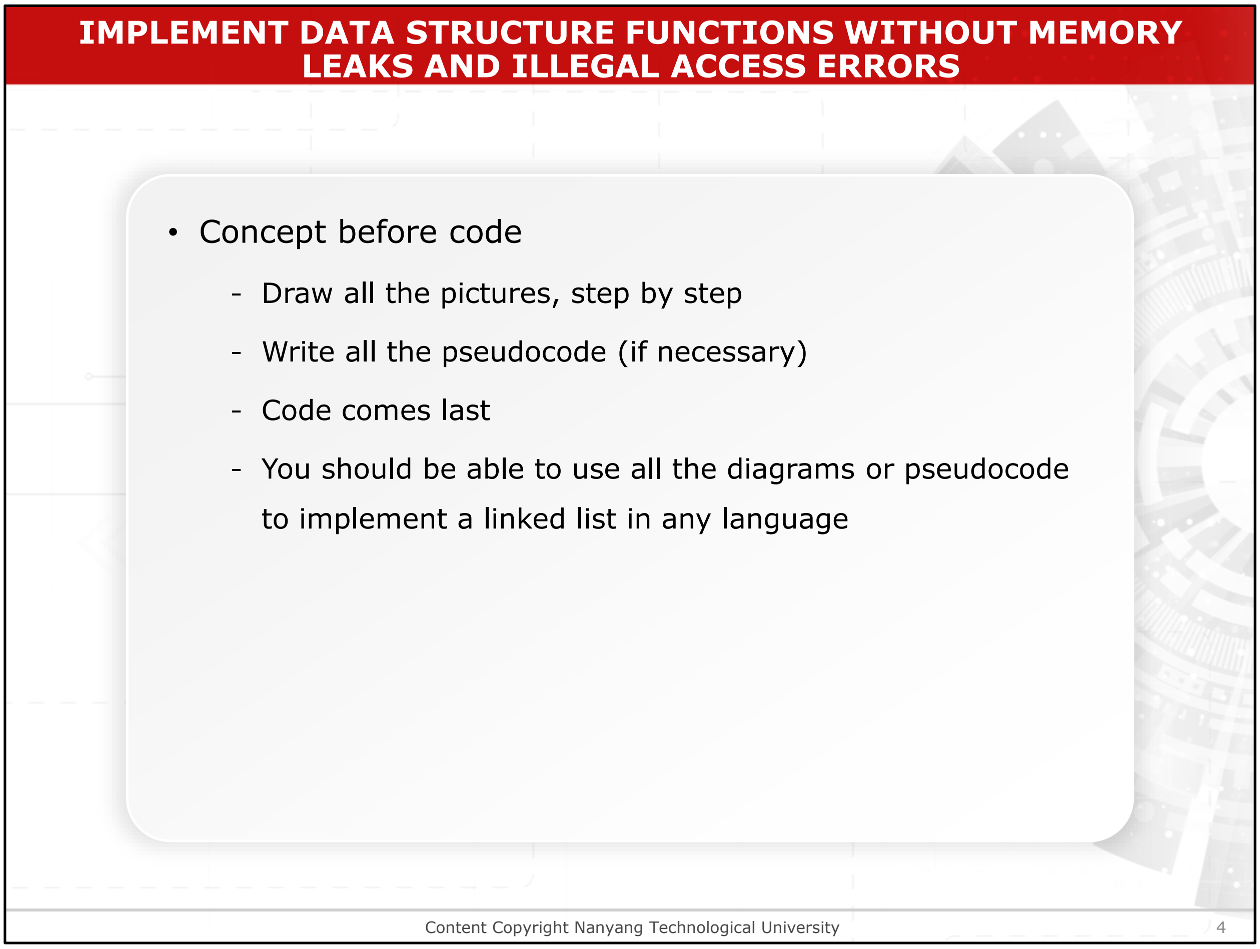
LEARNING OBJECTIVES

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

3

At the end of this lecture, you should be able to describe and implement core linked list functions.

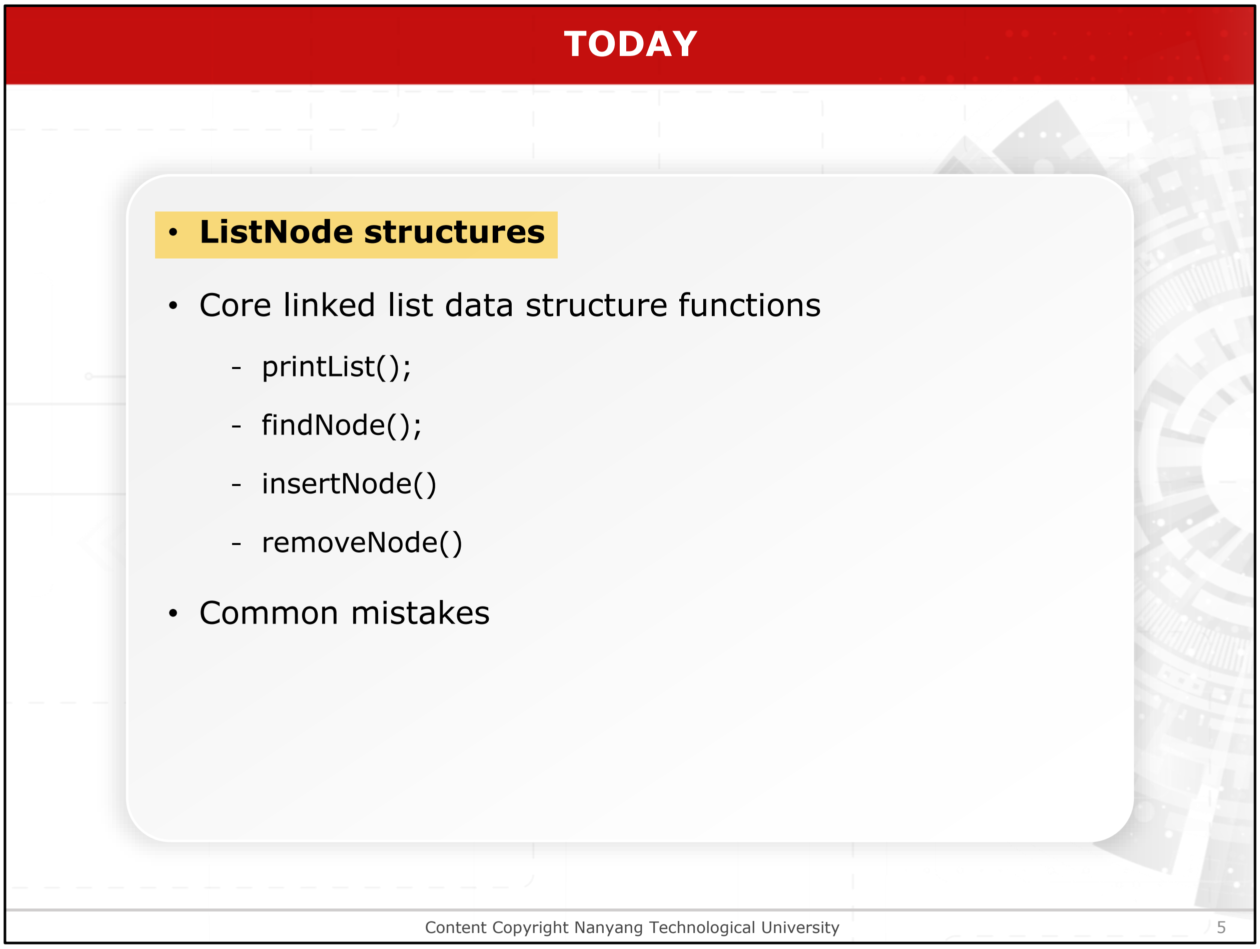


IMPLEMENT DATA STRUCTURE FUNCTIONS WITHOUT MEMORY LEAKS AND ILLEGAL ACCESS ERRORS

- 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

4

To implement data structure functions without memory leaks and illegal access errors, these steps should be followed before you start coding.

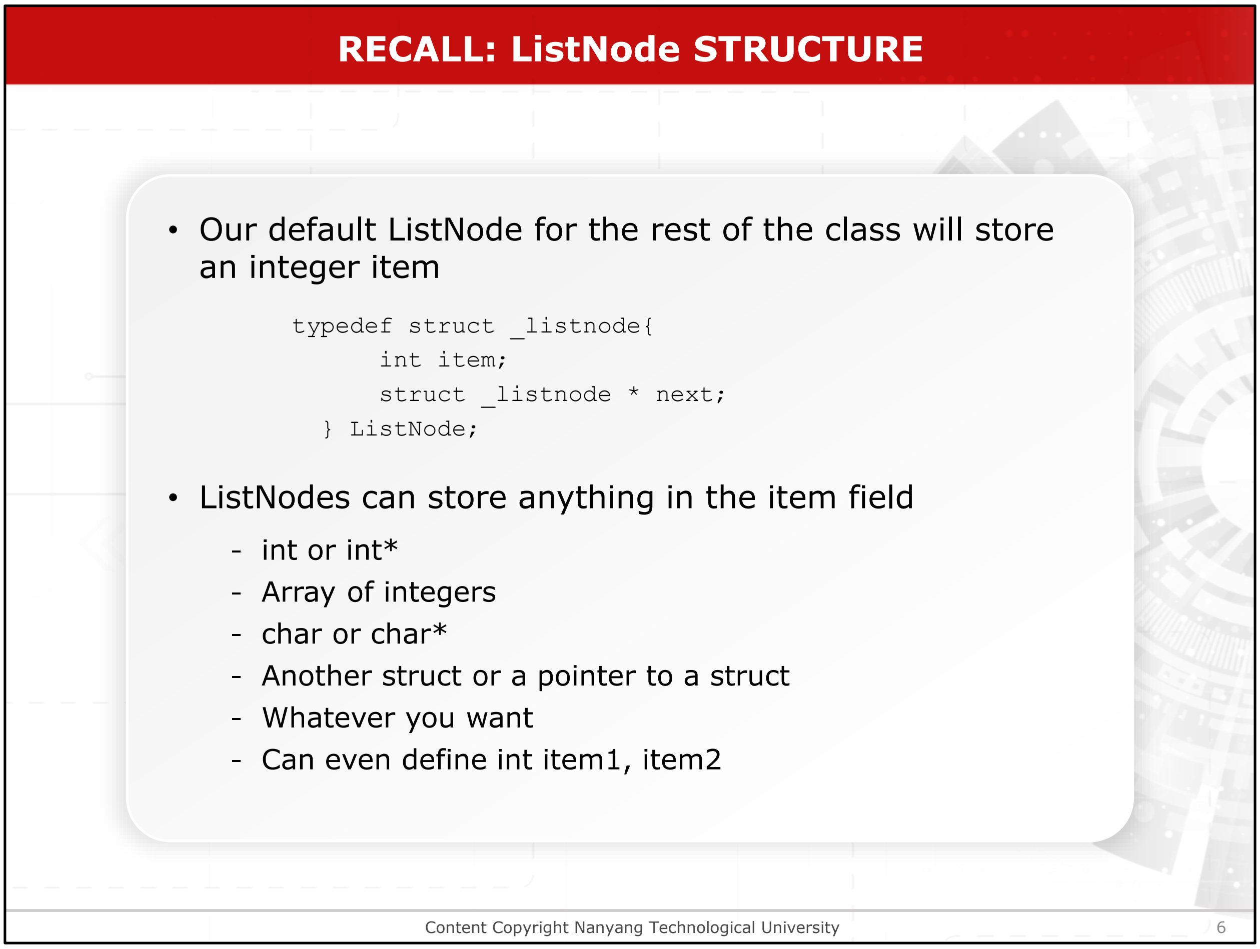


TODAY

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

5

First, we will discuss about ListNode structures.



RECALL: ListNode STRUCTURE

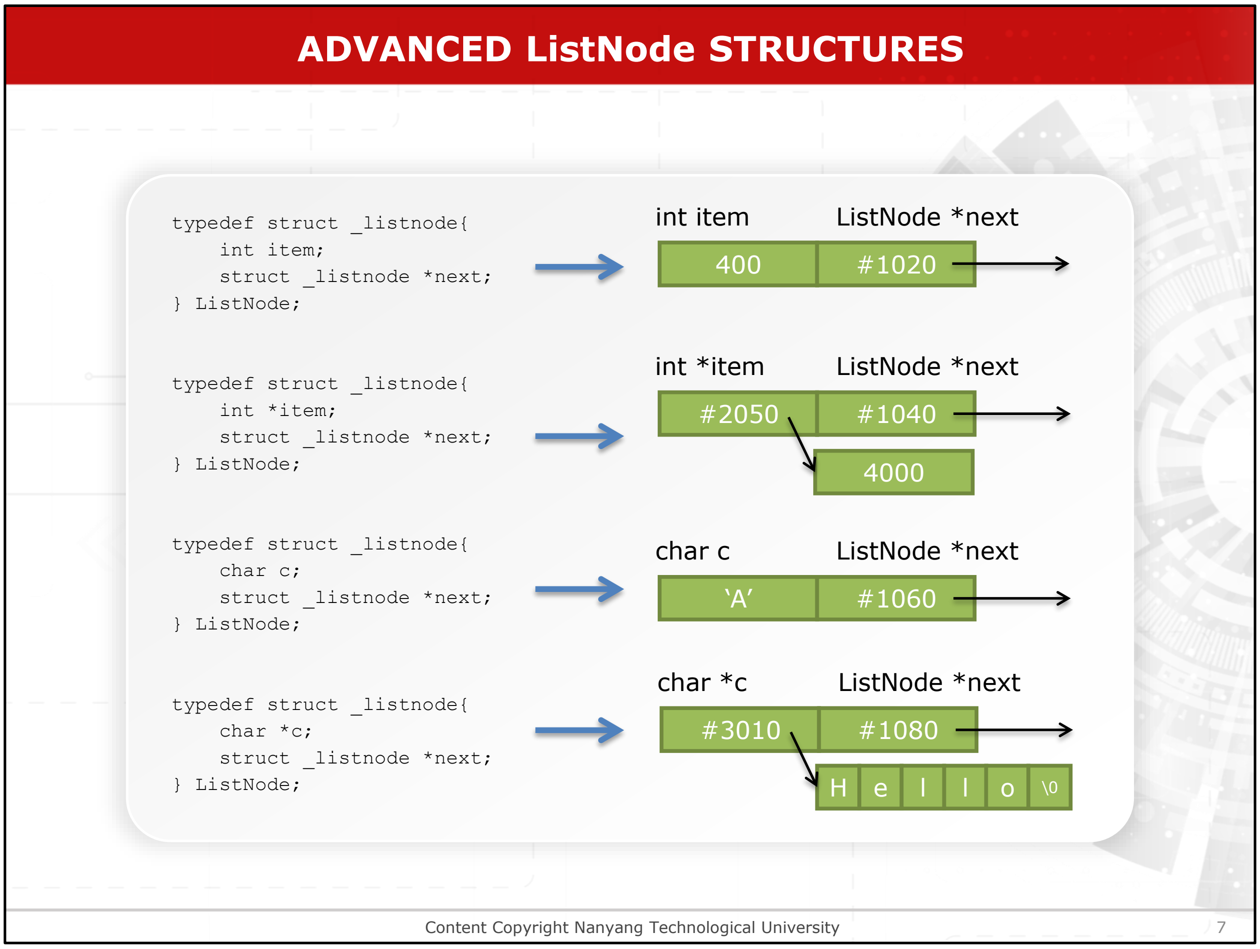
- Our default ListNode for the rest of the class will store an integer item

```
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

6

Here you have a very simple list node structure which has an integer item and a pointer. As we discussed earlier, a list node structure can hold not only integer values; it can hold int*, array of integers, another structure or a pointer to another structure. Also, you can have not only just one item but also multiple items inside the data portion of the list node structure.



ADVANCED ListNode STRUCTURES

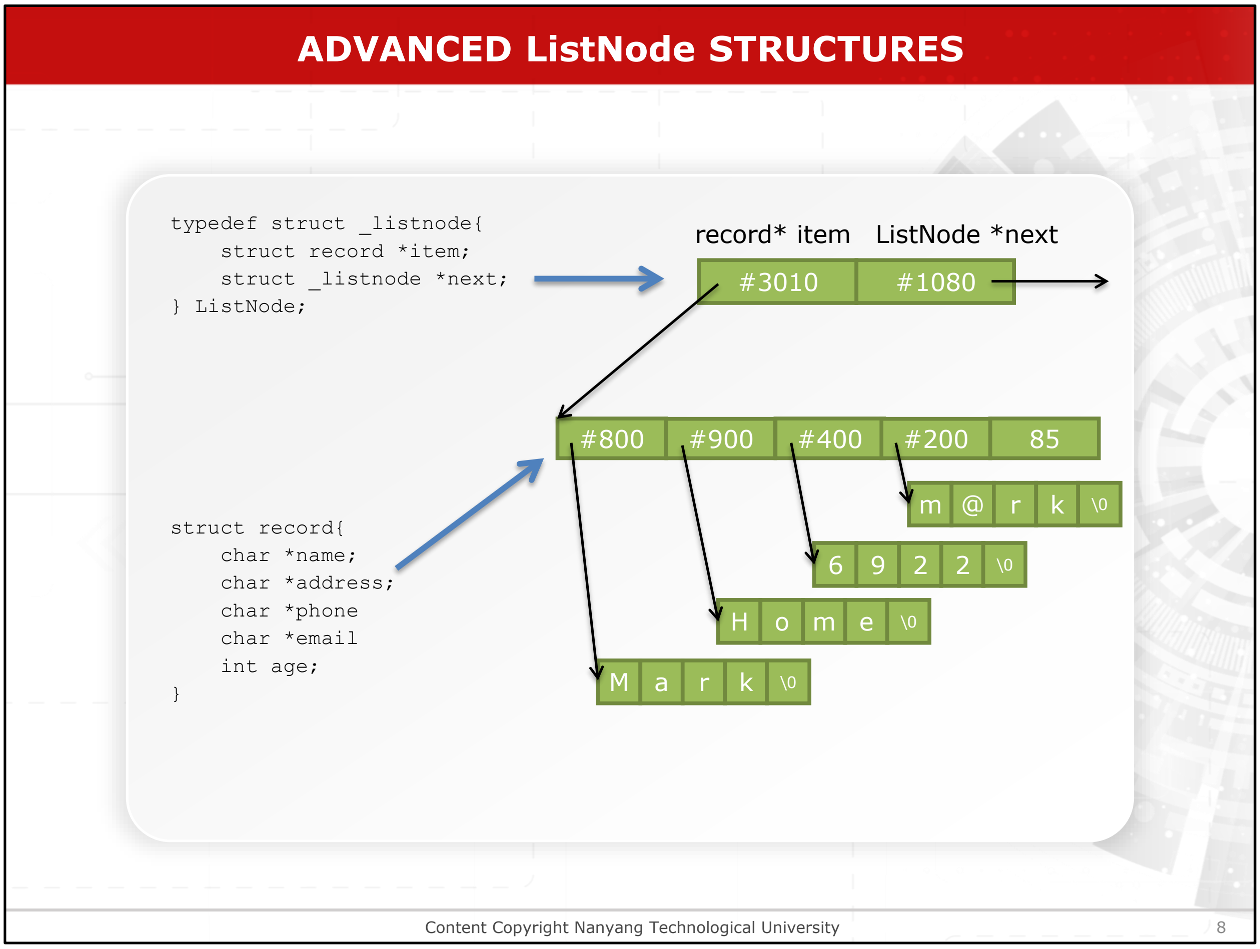
```
typedef struct _listnode{
    int item;
    struct _listnode *next;
} ListNode;
```

```
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;
```

We will start with the basic list node structure that contains an integer value. When you have int *item in your list node structure, that item pointer takes you to the actual integer value in another memory block. Also, instead of an integer, you can store a character inside the structure. We can use *c to point you to a long string somewhere else in the memory.

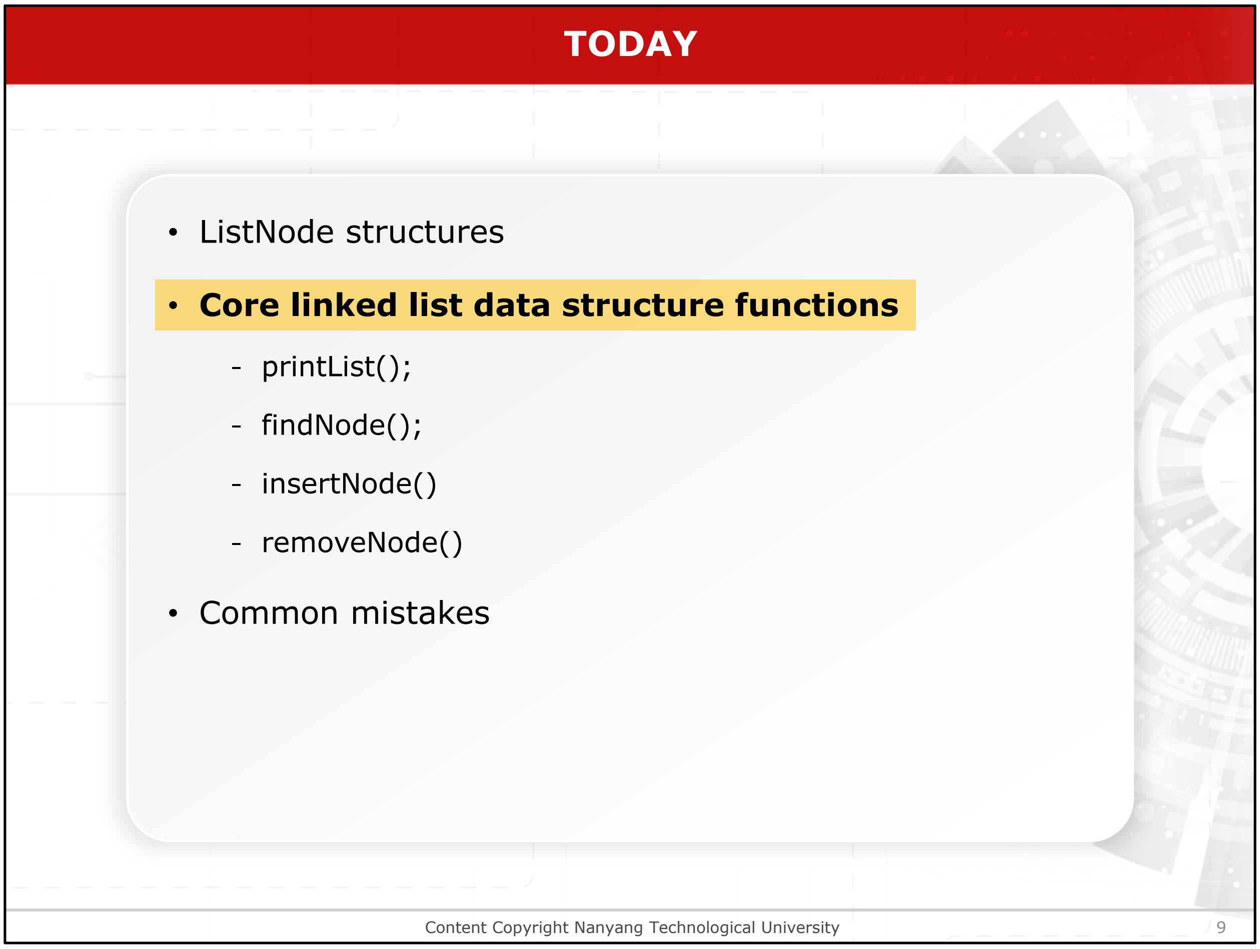


ADVANCED ListNode STRUCTURES

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

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

Here, I have a list node structure. In the data portion, I have a pointer to another structure. As you can see, the other structure has four strings, four character arrays and an integer. Imagine this as an address book. Therefore, instead of having addresses, contacts, names, emails, etc. individually, I can store them all in the list node structure. Each of these record items can store as many member fields as you want.

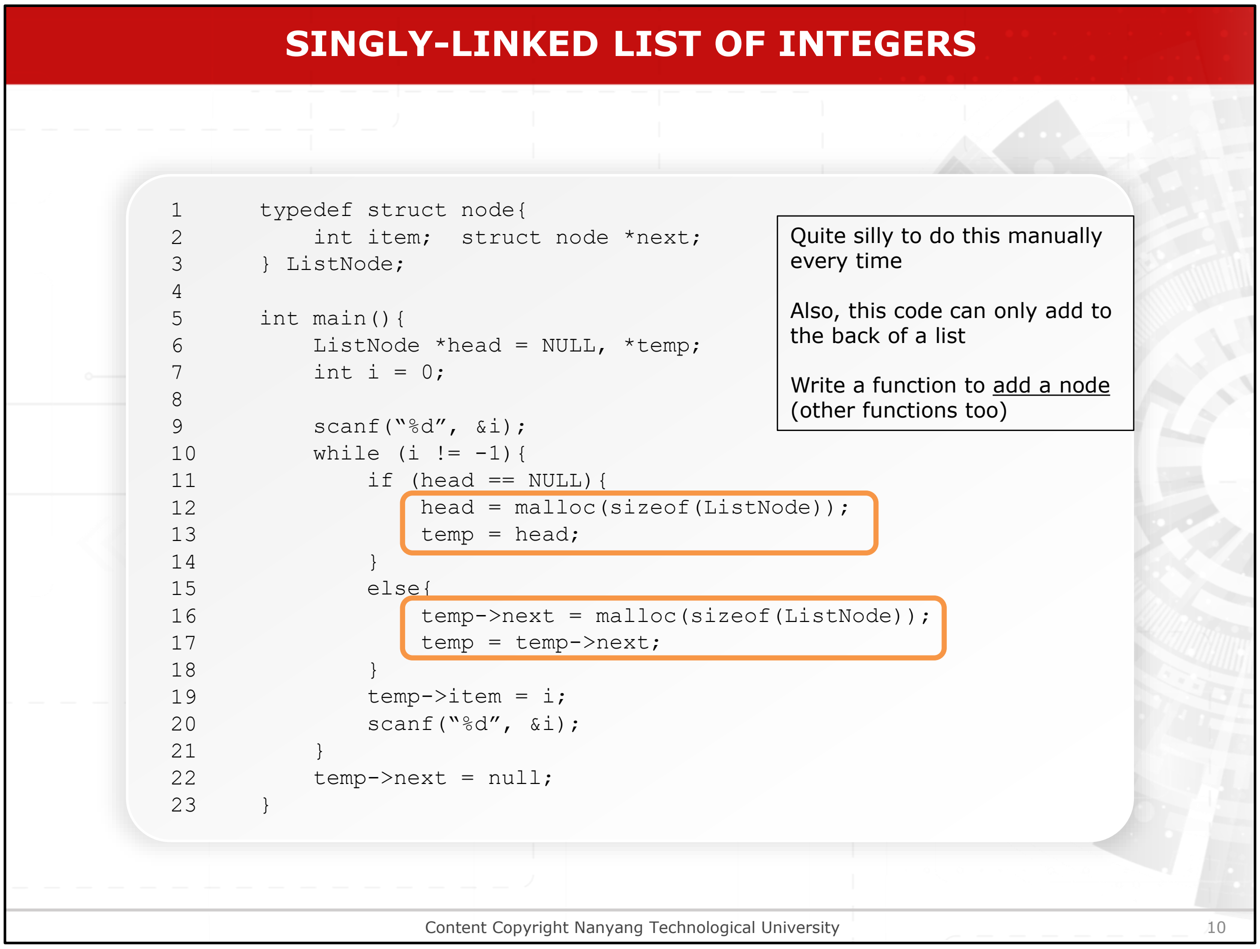


TODAY

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

9

Now, let’s start discussing about core linked list functions.



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

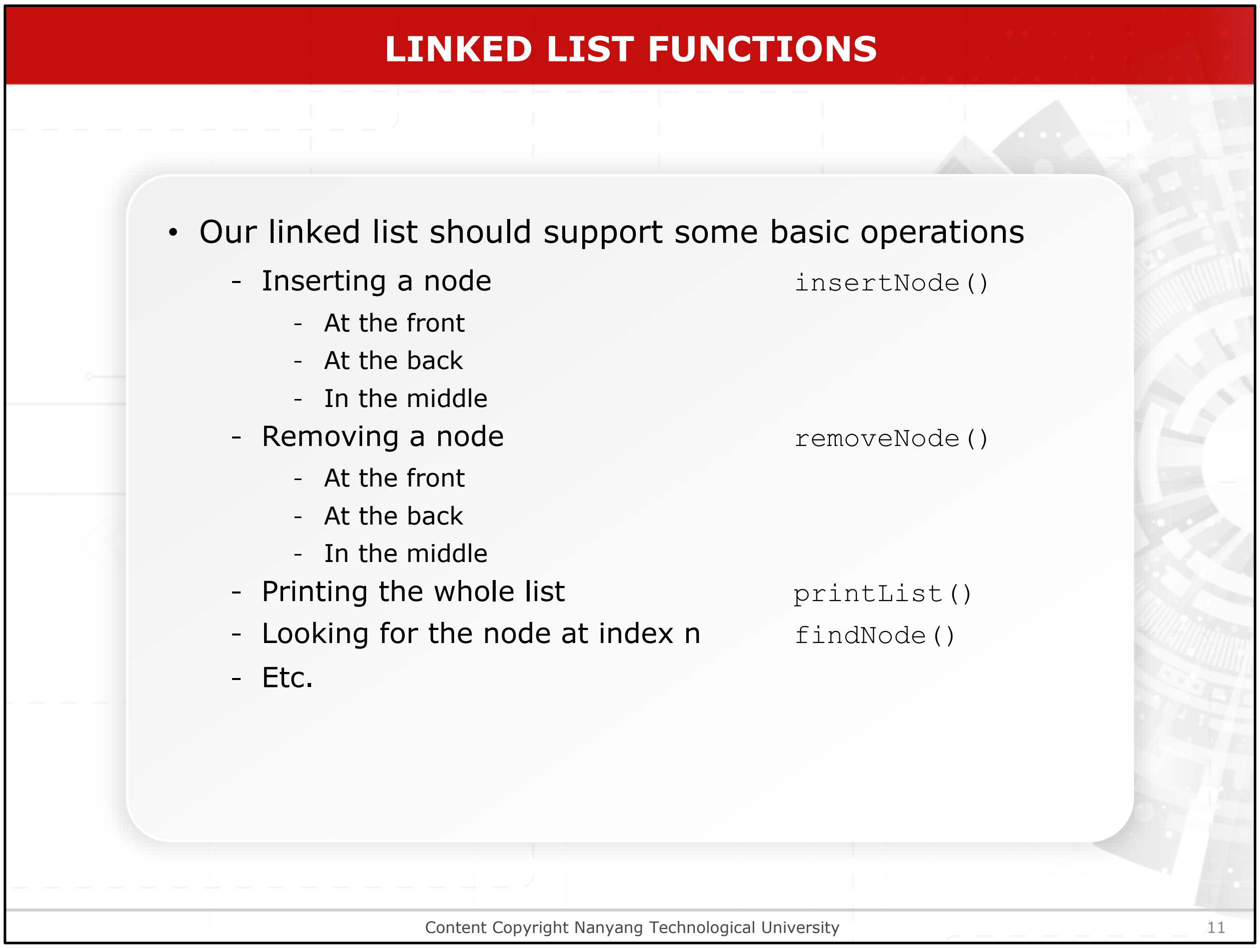
Also, this code can only add to the back of a list

Write a function to <u>add a node</u> (other functions too)

Content Copyright Nanyang Technological University 10

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.

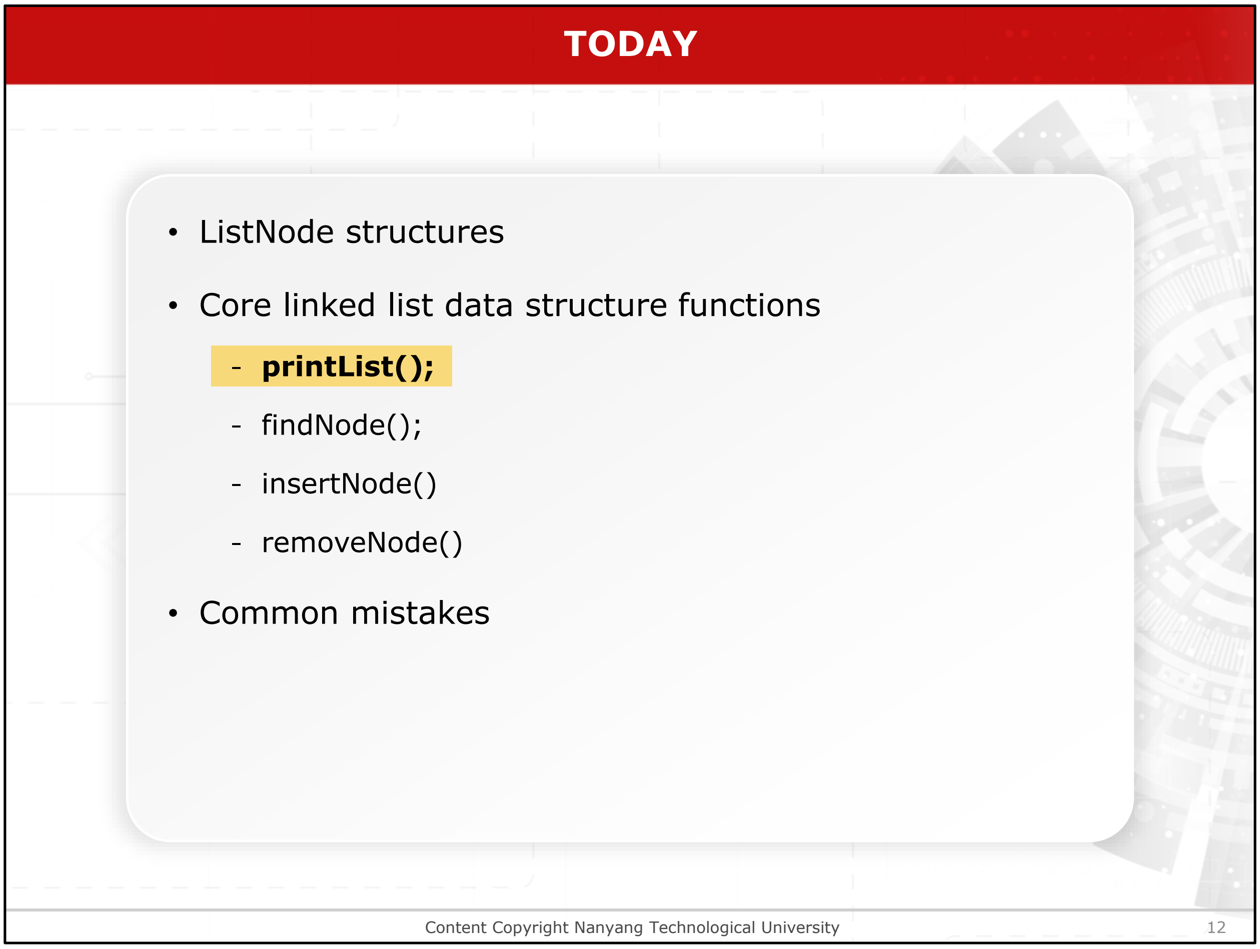
But, every time we run this code, it has to deal with different pointers and it is also limited to insert items only to the back of the linked list. Since we need to run through the code every time we insert a new node, it is better that we put it into a function. The function can build in to add a new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

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.

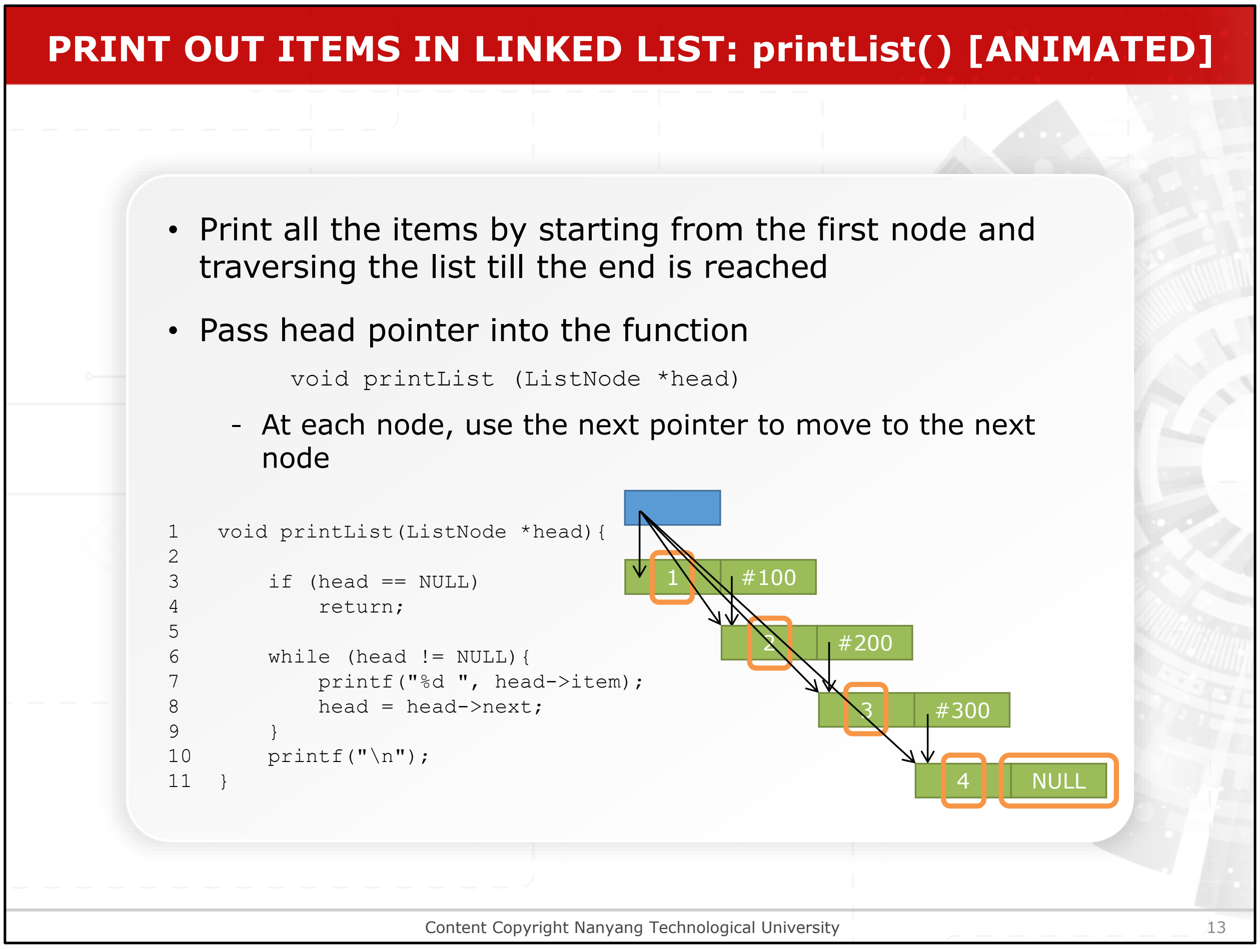


TODAY

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

12

Let's start with the print list function.



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  }
```

13

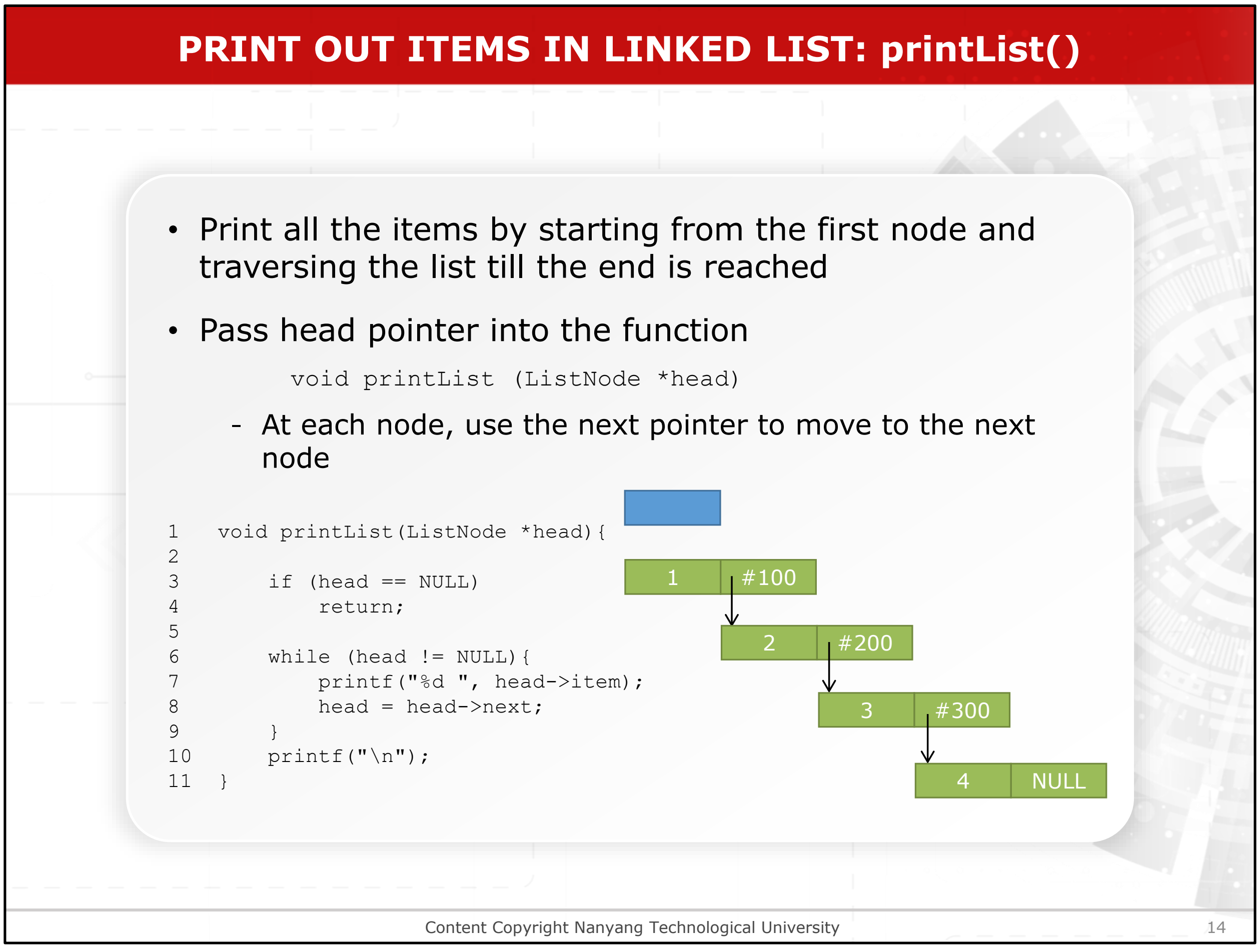
The general concept of printList() is that starting from the first node, the function will print the value stored inside each node while traversing the entire list.

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.



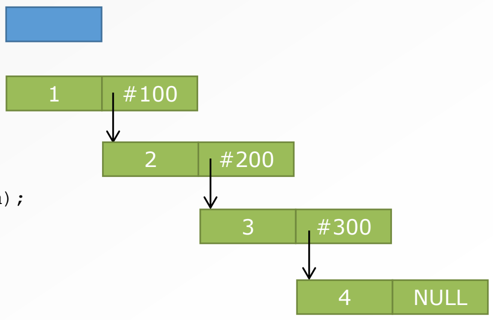
PRINT OUT ITEMS IN LINKED LIST: printList()

- 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

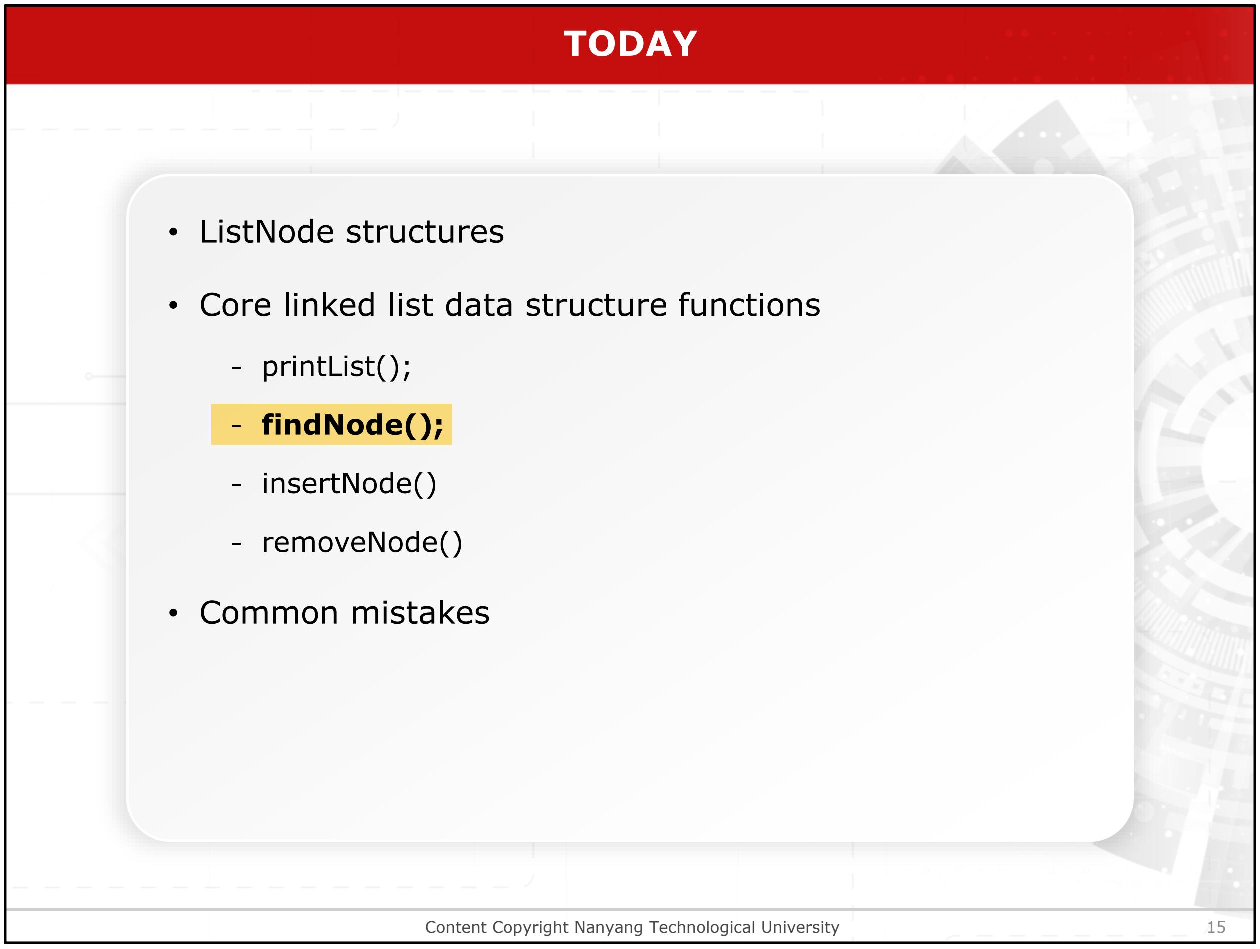
```
void printList (ListNode *head)
```

  - 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.

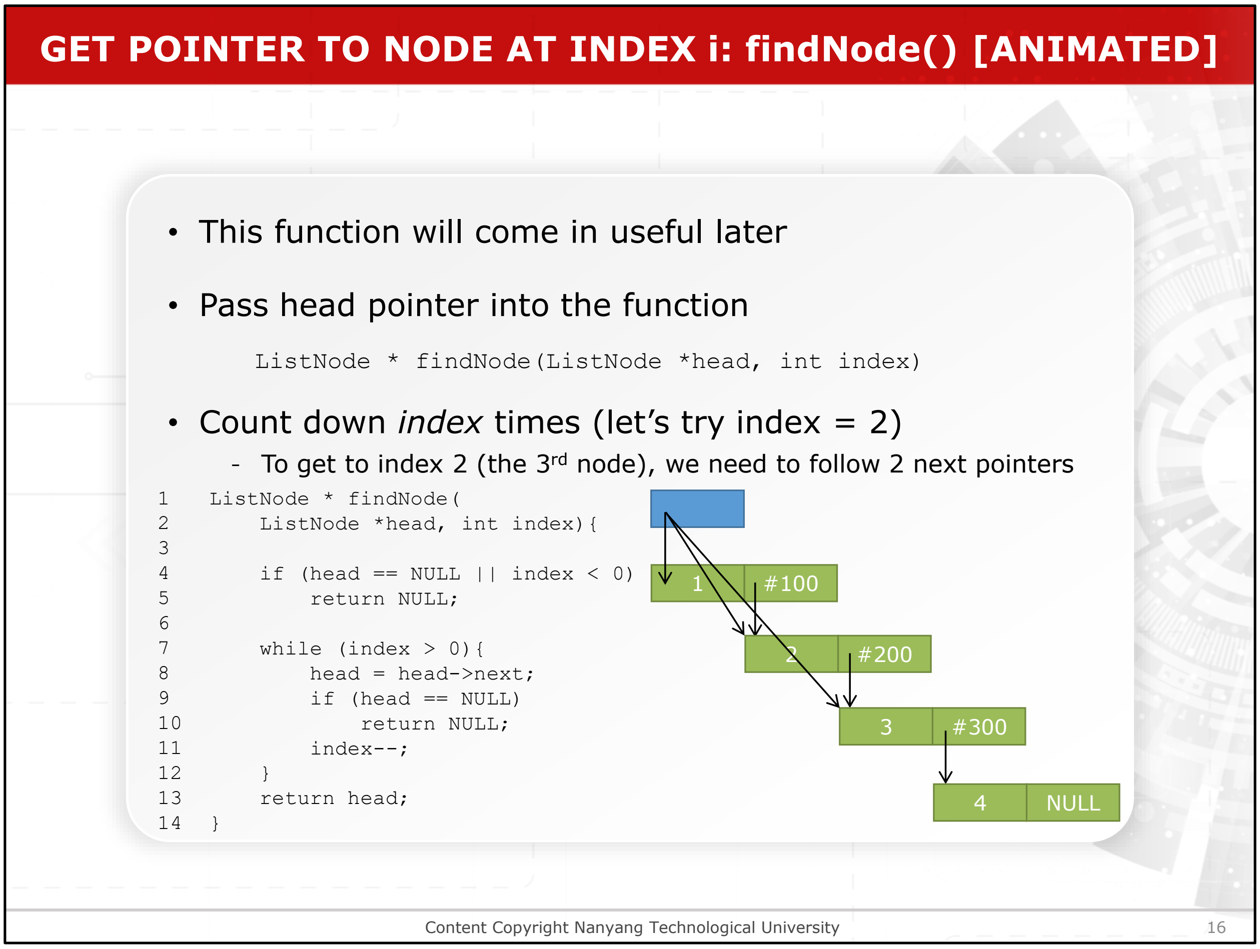


TODAY

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

15

Now, let’s discuss about findNode function.



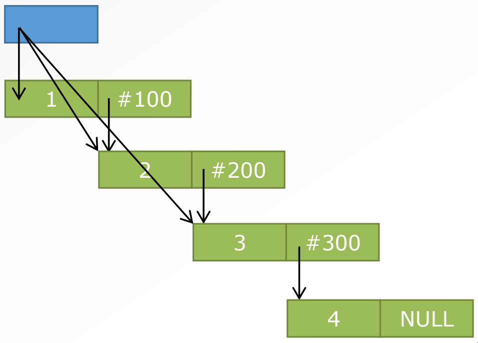
GET POINTER TO NODE AT INDEX i: findNode() [ANIMATED]

- 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  }
```



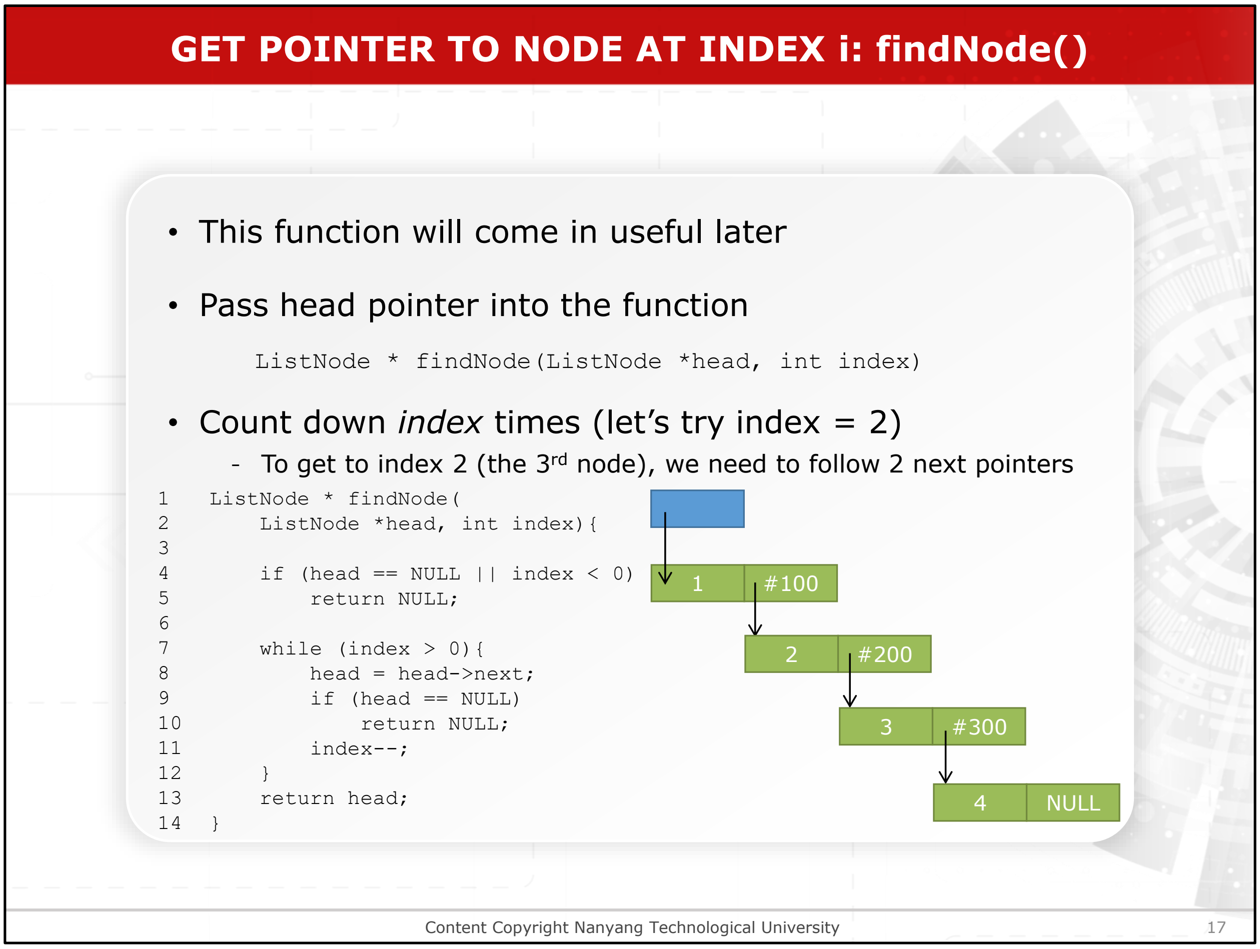
16

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.



GET POINTER TO NODE AT INDEX i: findNode()

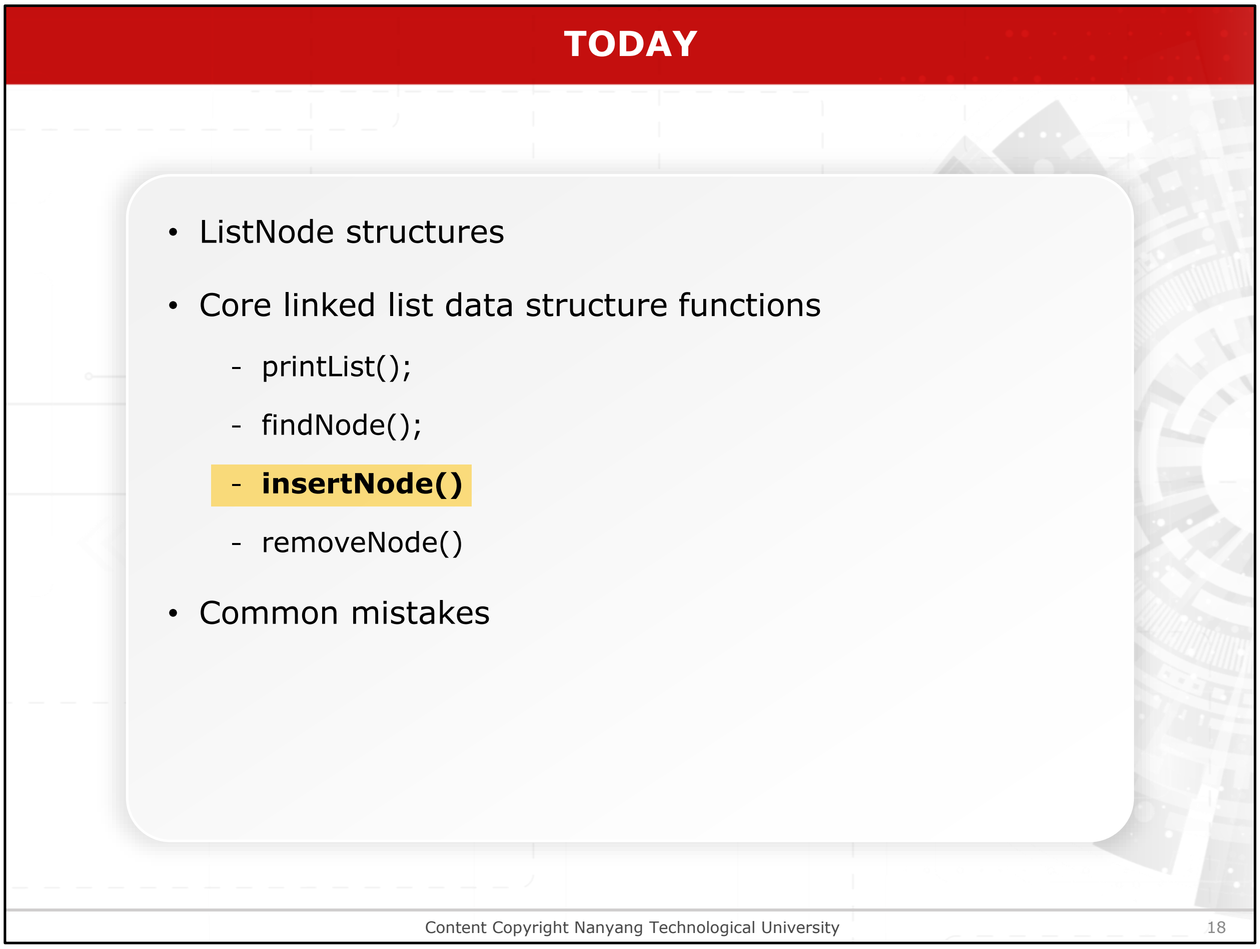
- This function will come in useful later
- Pass head pointer into the function

```
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.

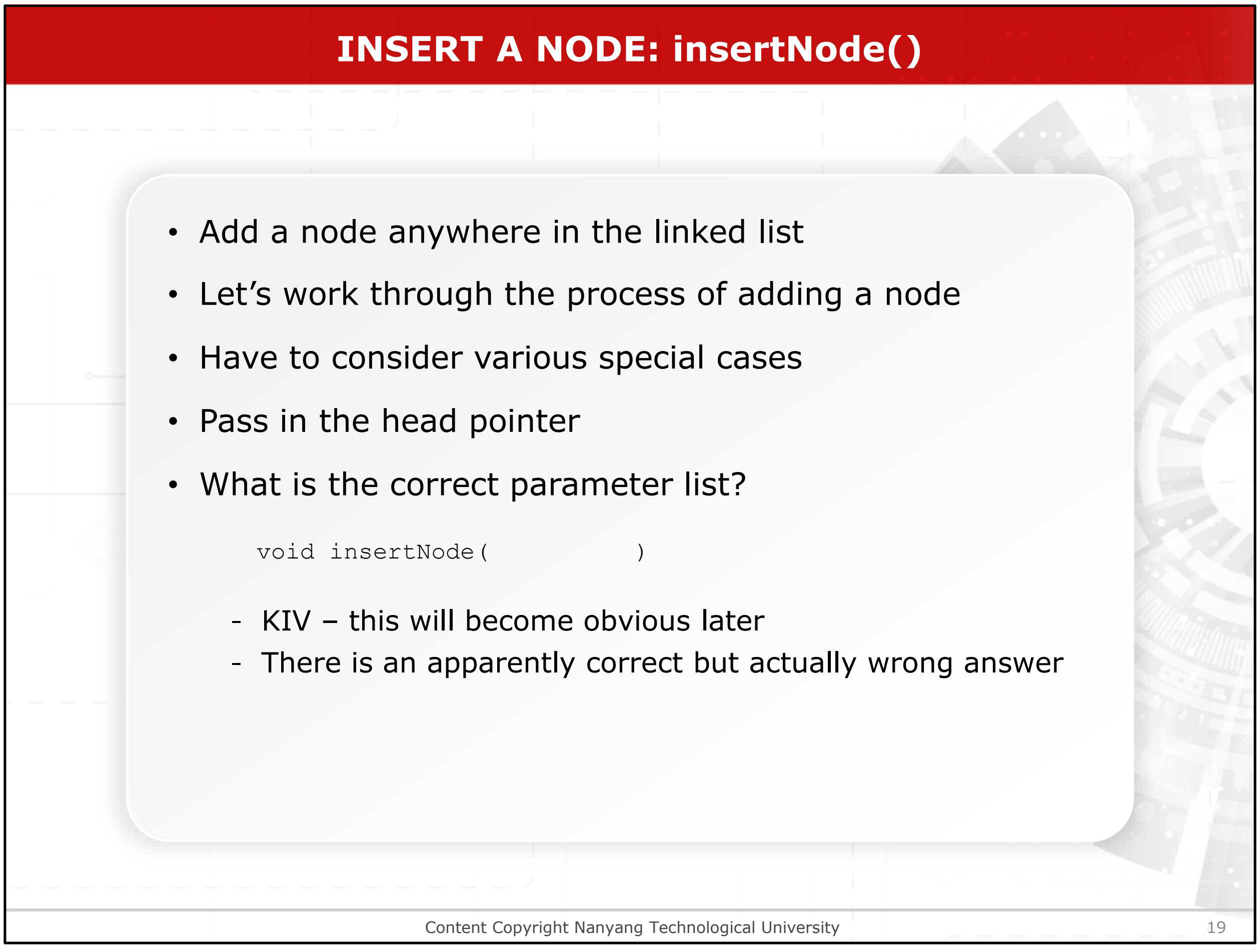


TODAY

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

18

Now I want to go on to the insert node function; this is the most important part of today's lecture.



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?

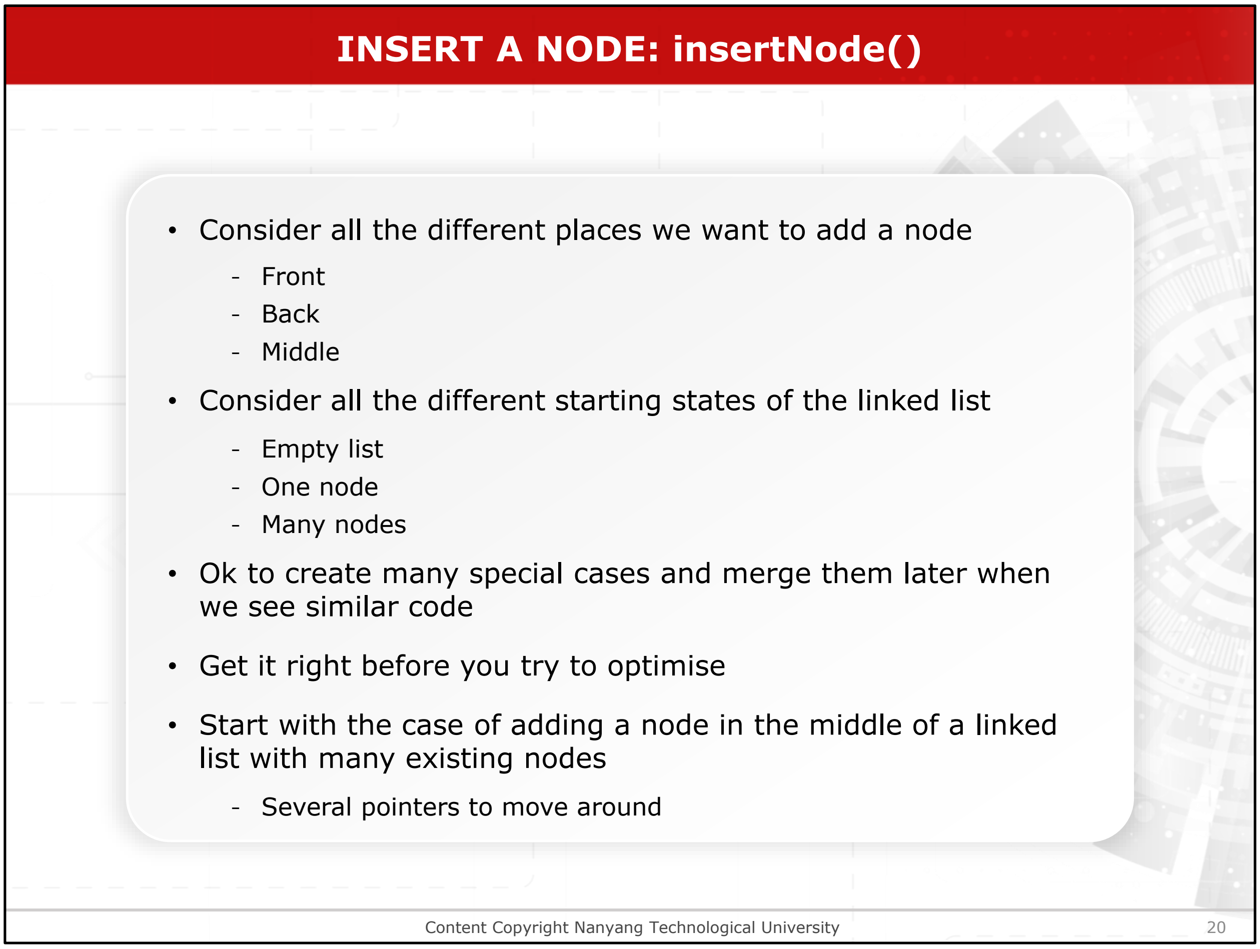
```
void insertNode(          )
```

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

19

Using insertNode() function, we can add nodes to anywhere in an existing linked list.

Previously discussed functions used listNode * head parameter to keep track of the first node. For insertNode(), what will be the parameter list?



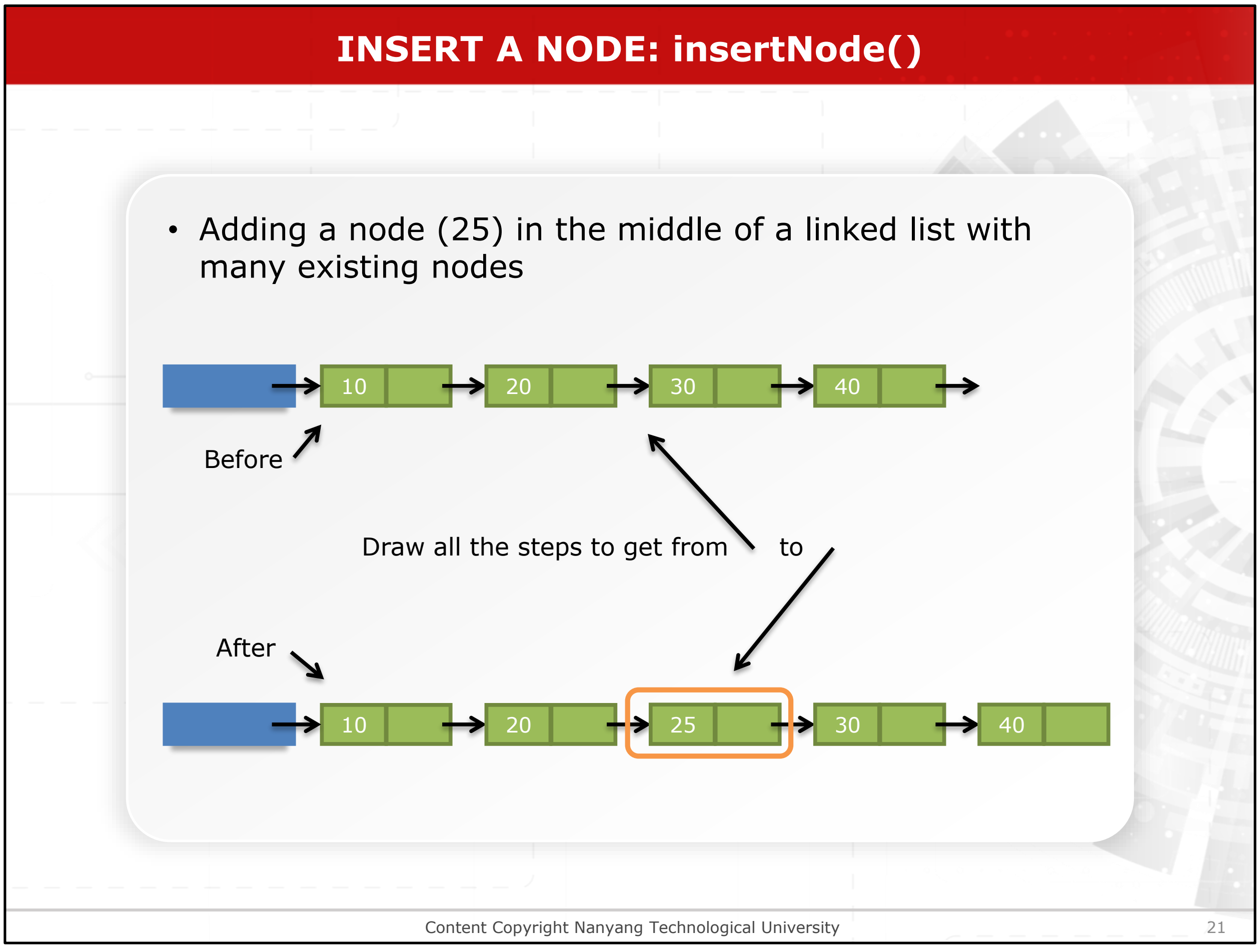
INSERT A NODE: insertNode()

- 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

20

The different places to which we can add a list node are the back, the middle and the front of the linked list. Also, the starting state of the linked list can be an empty list, a list with one node or a list with many nodes.

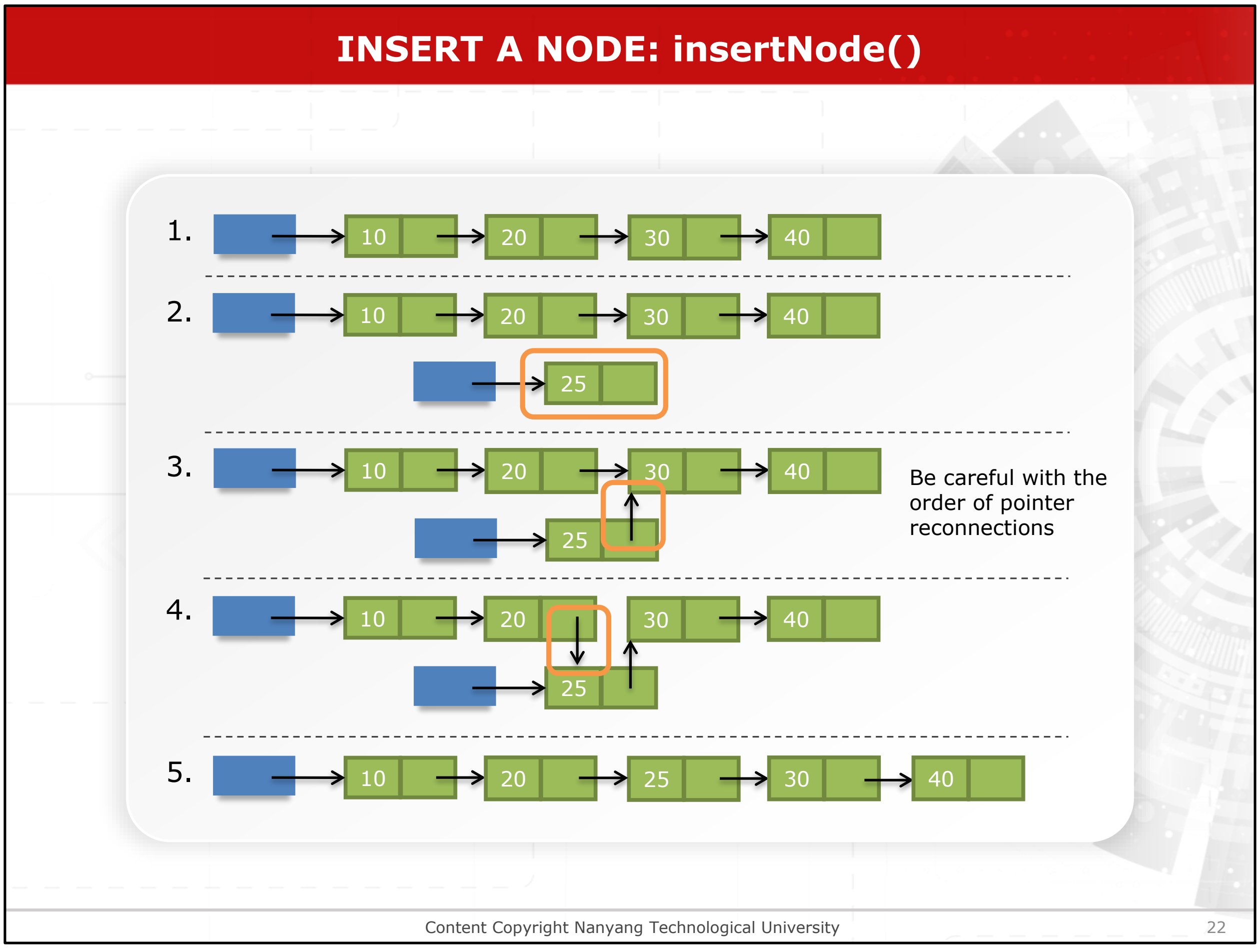
We can implement the code for each case but at the end, you will realise that these cases can be merged. Therefore, we will consider a normal case where we try to add a node to the middle of a linked list which has multiple nodes.



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()

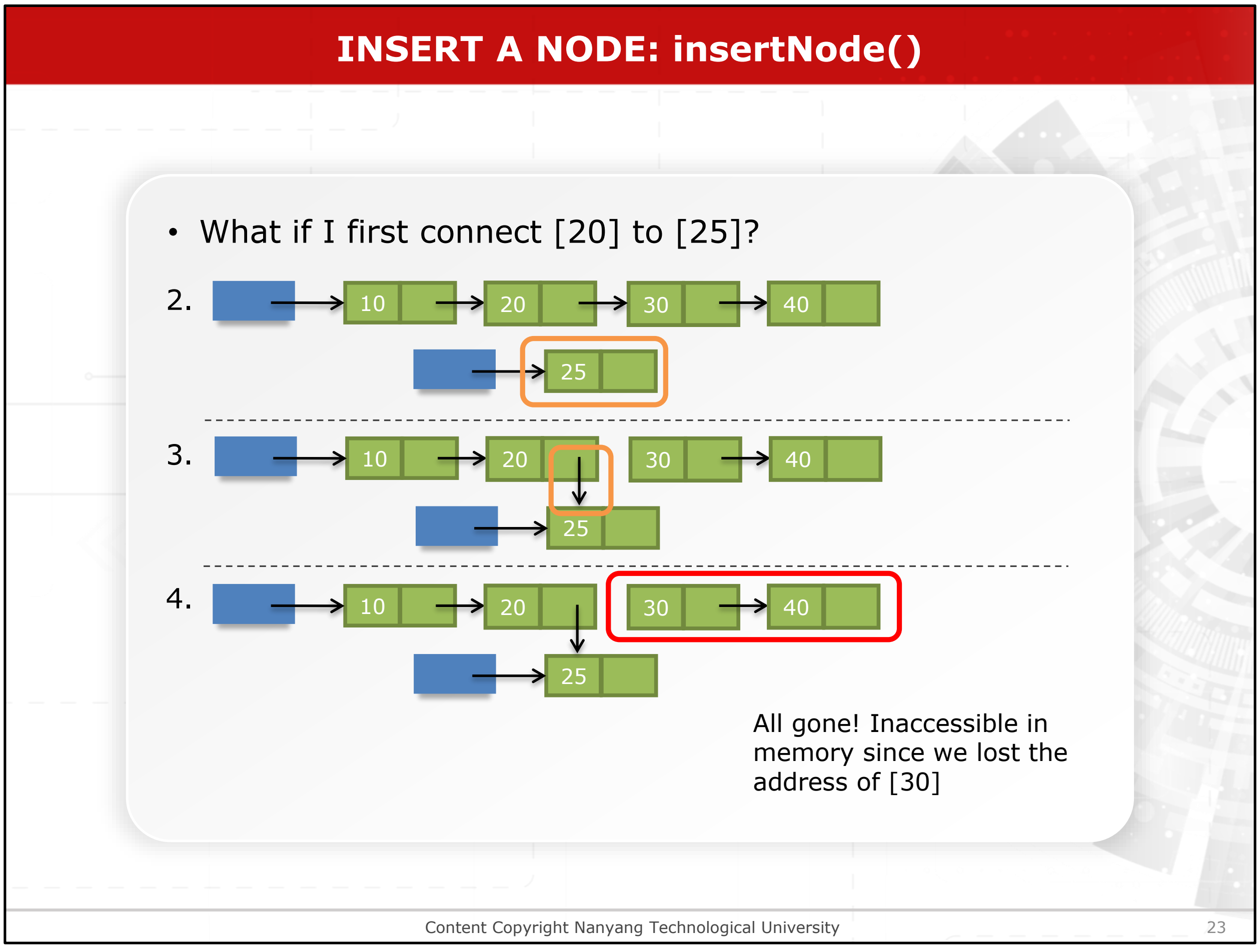
Be careful with the order of pointer reconnections

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]?

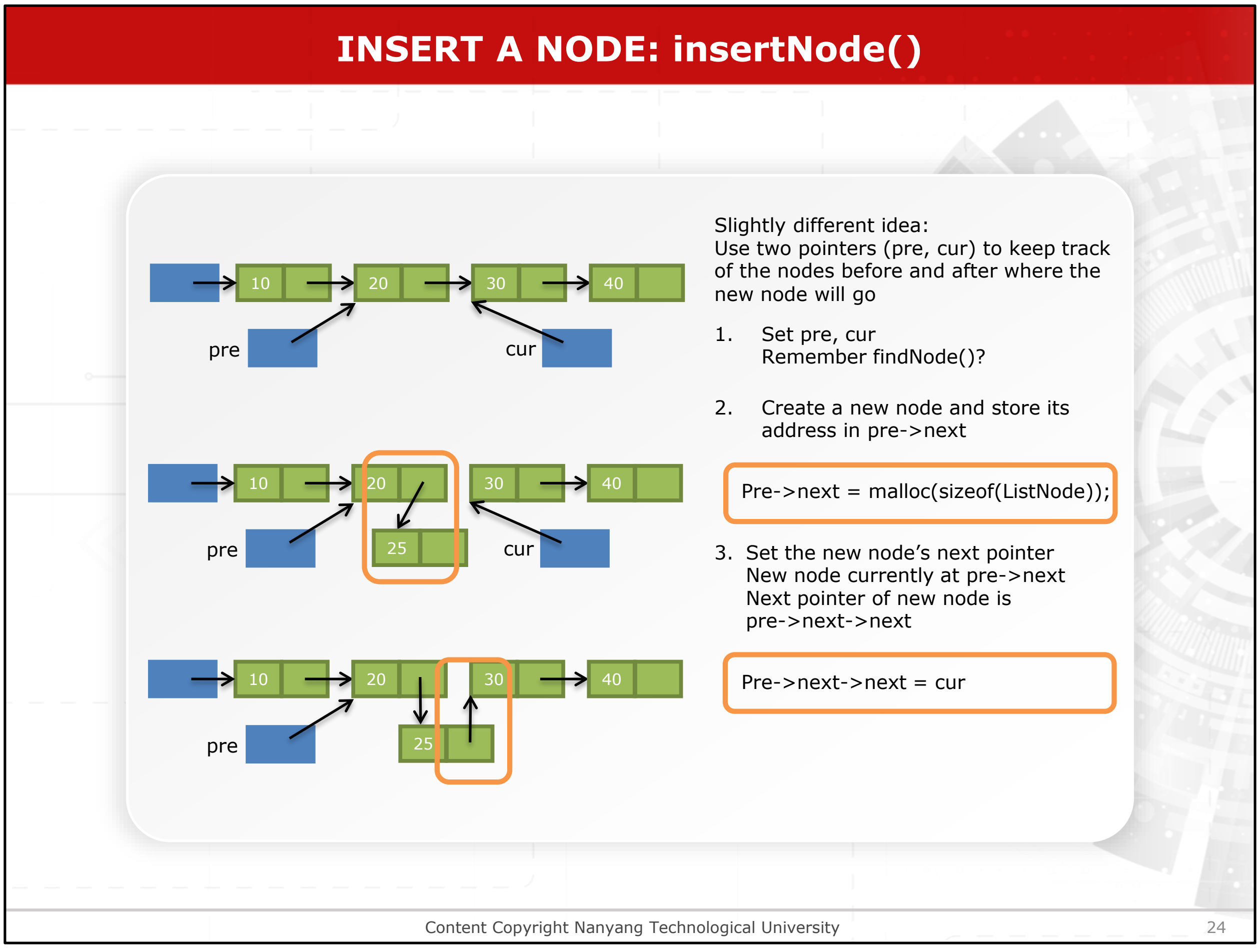
All gone! Inaccessible in memory since we lost the address of [30]

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
```

Content Copyright Nanyang Technological University 24

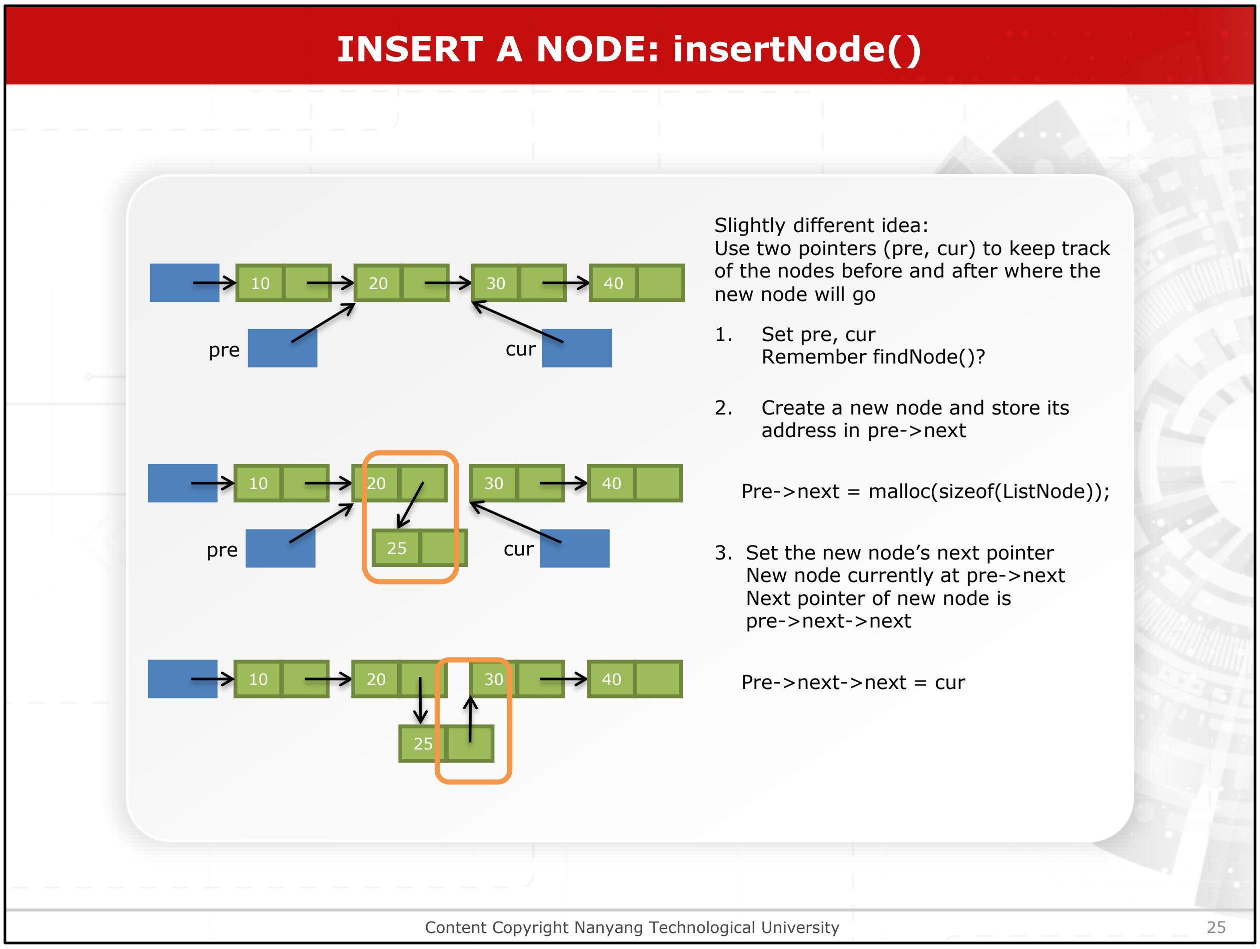
We introduce two pointers as previous and current to keep track of the before and after nodes of the newly added node. Since our new node goes between node 20 and node 30, the previous node is node 20, and the current node is node 30.

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.

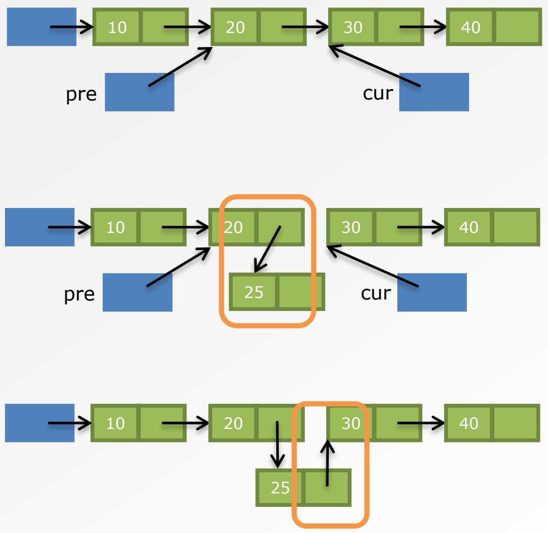
I have created the new node and assigned its address to the pre-> next using malloc(). It created the link between the second node and the new third node.

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()?

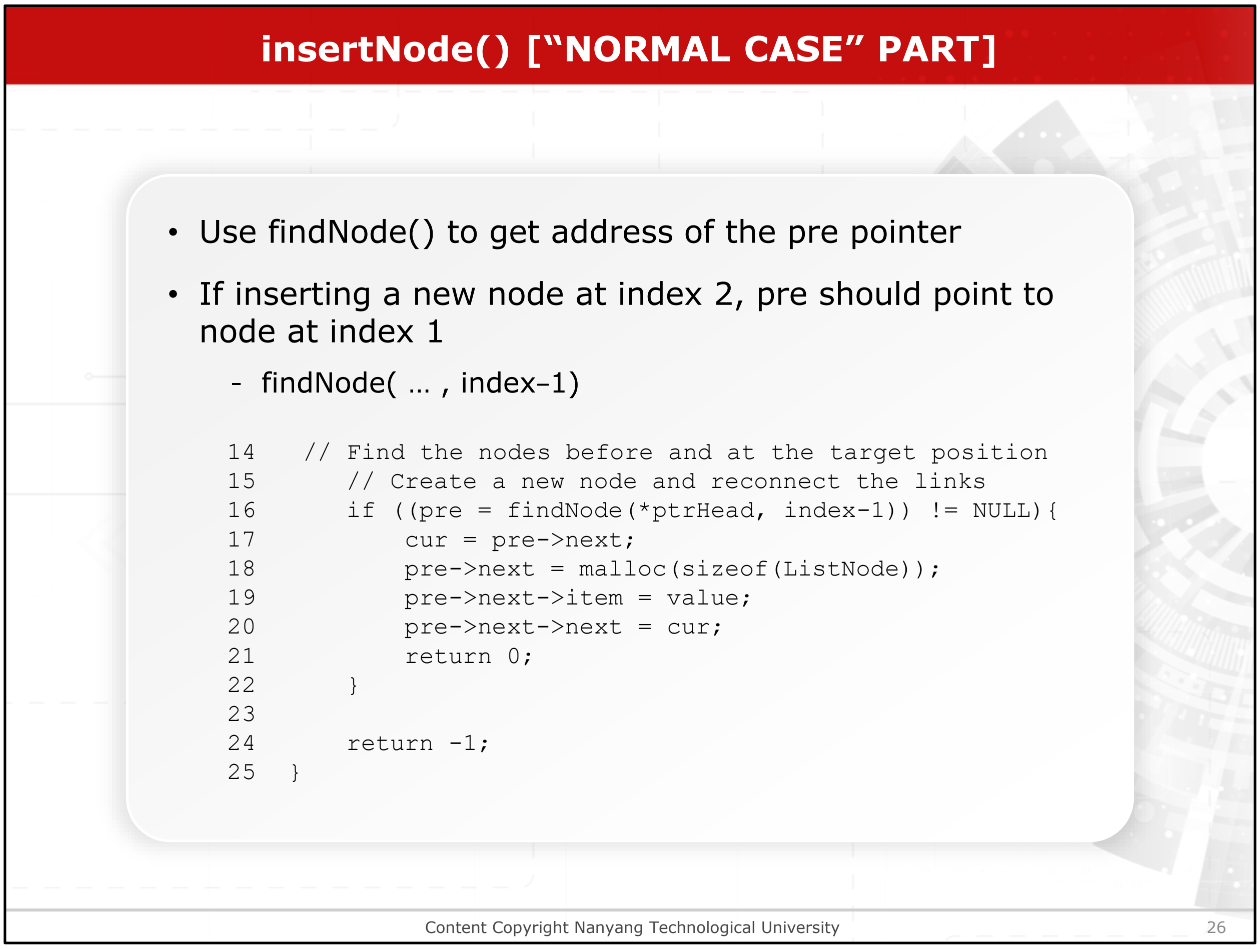
2. Create a new node and store its address in pre->next

```
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
```

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]

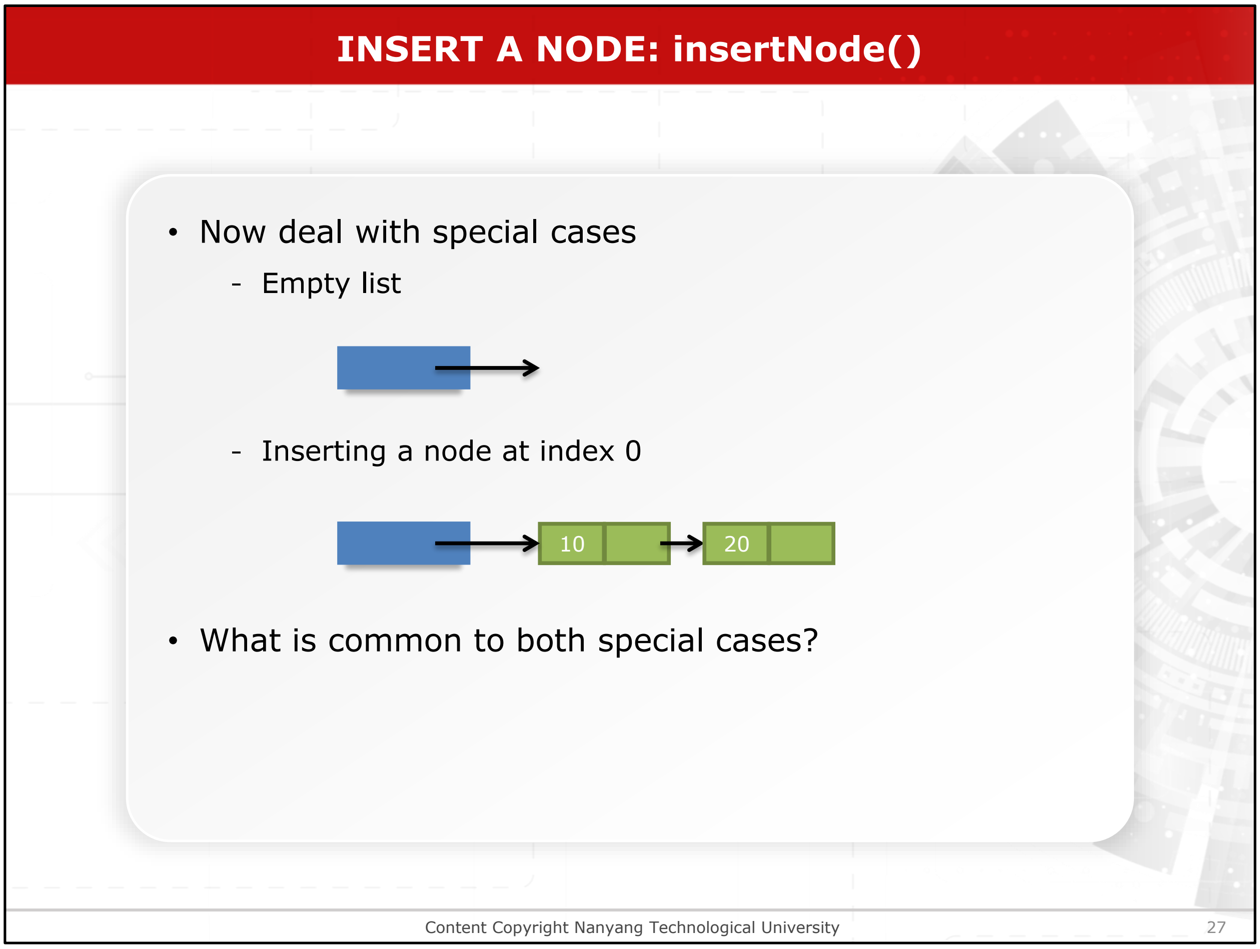
- Use findNode() to get address of the pre pointer
- If inserting a new node at index 2, pre should point to node at index 1
  - findNode(… , index-1)

```
    // 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;
    }
```

Content Copyright Nanyang Technological University 26

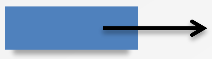
As we discussed previously, the findNode function will be used to find the location on which we need to insert the new node. Therefore, here we use the findNode function to get the address for the pre pointer. So if we are inserting at index 2, pre pointer should be pointing at the node in index 1. Therefore,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

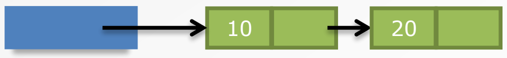


INSERT A NODE: insertNode()

- Now deal with special cases
  - Empty list



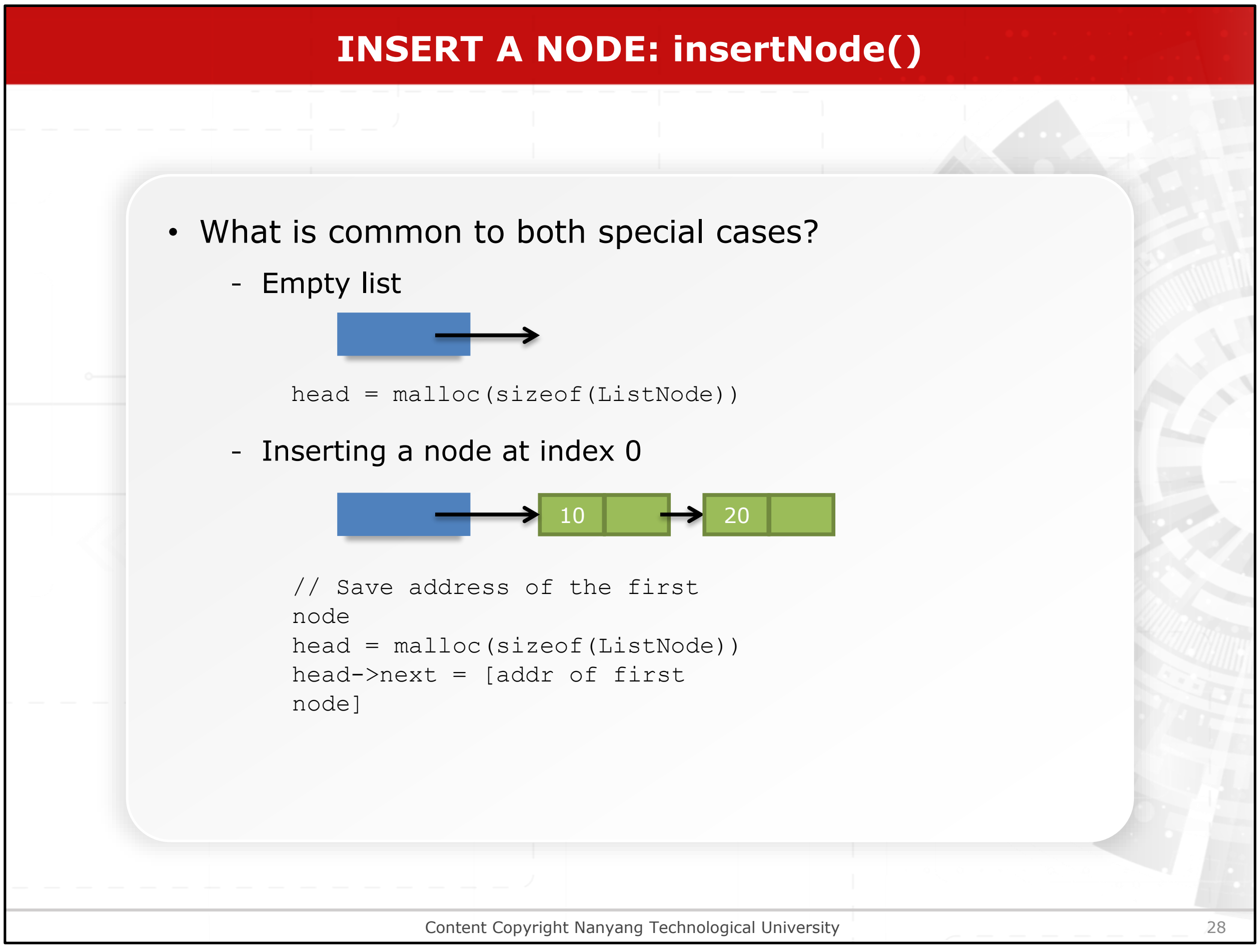
  - Inserting a node at index 0



- What is common to both special cases?

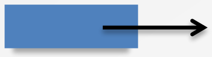
Content Copyright Nanyang Technological University 27

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?



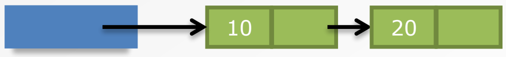
INSERT A NODE: insertNode()

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



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

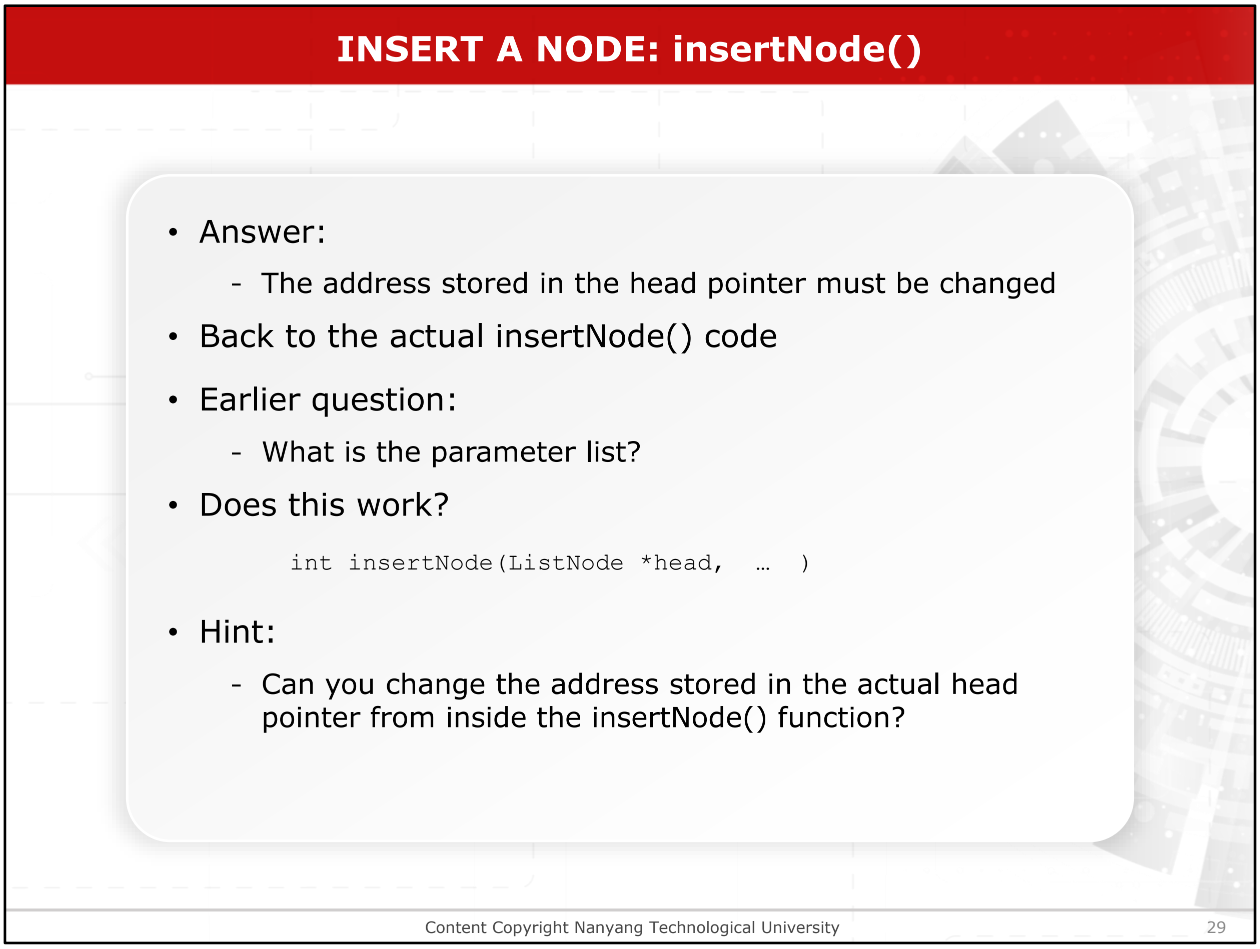
  - Inserting a node at index 0



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

28

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()

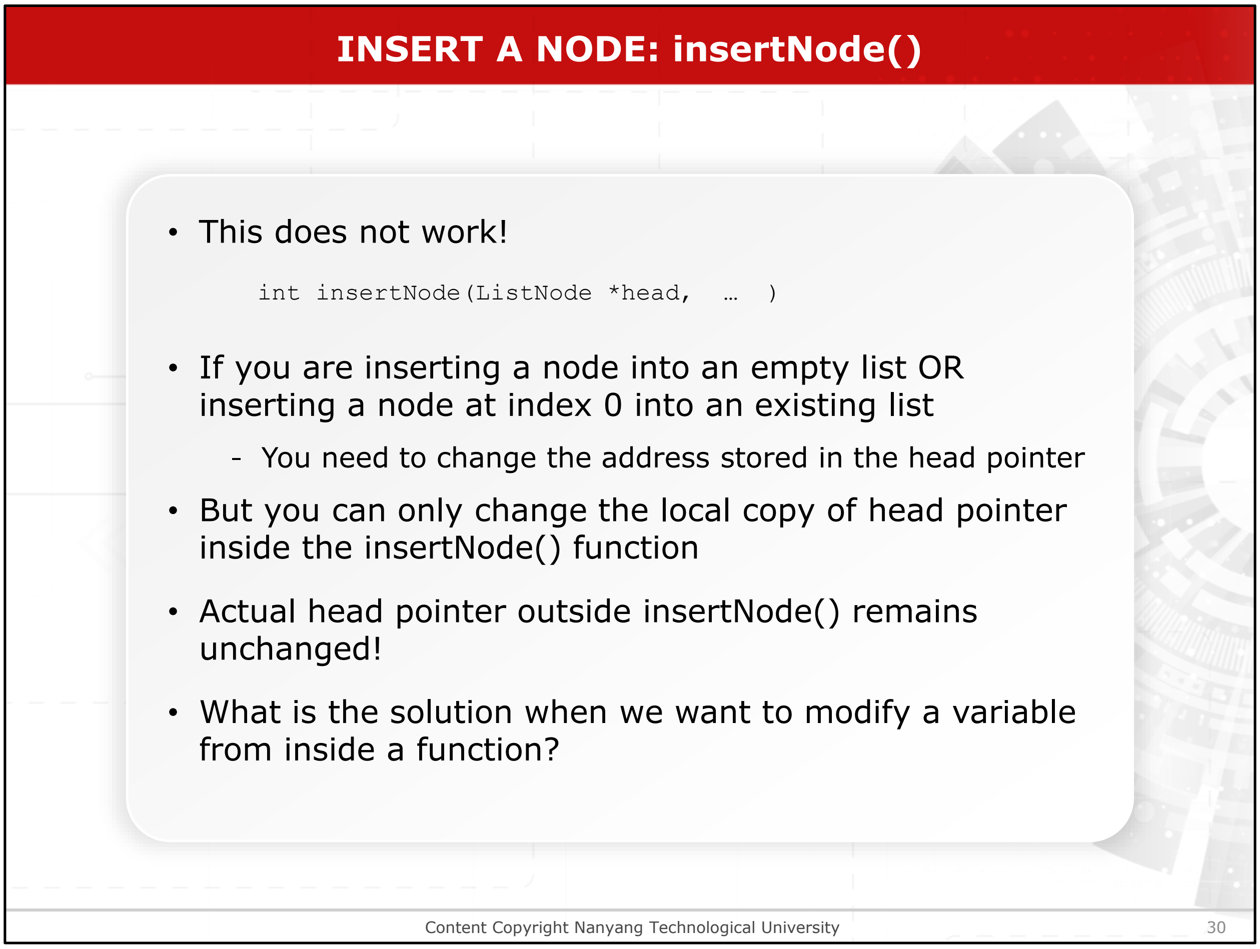
- Answer:
  - The address stored in the head pointer must be changed
- Back to the actual insertNode() code
- Earlier question:
  - What is the parameter list?
- Does this work?

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

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

29

The common feature of these two cases is that the address stored in the head pointer is changed.
Now, if we go back to the question at the beginning of the insertNode() lecture, can we pass ListNode *head as a parameter to the insertNode() function? After passing the head pointer, can we change the value of it inside the insertNode() function?



INSERT A NODE: insertNode()

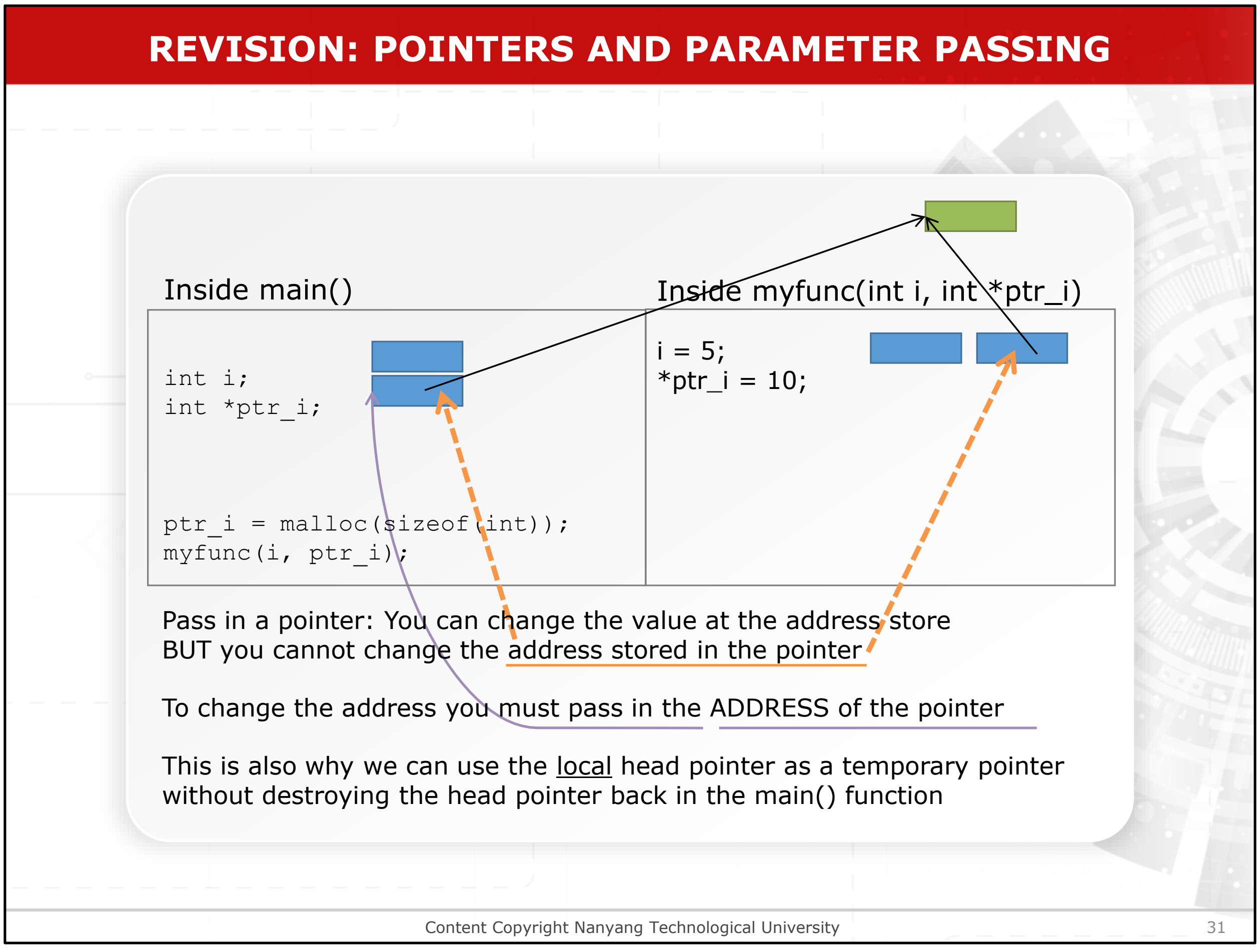
- This does not work!

```
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?

30

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

```
int i;
int *ptr_i;

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

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

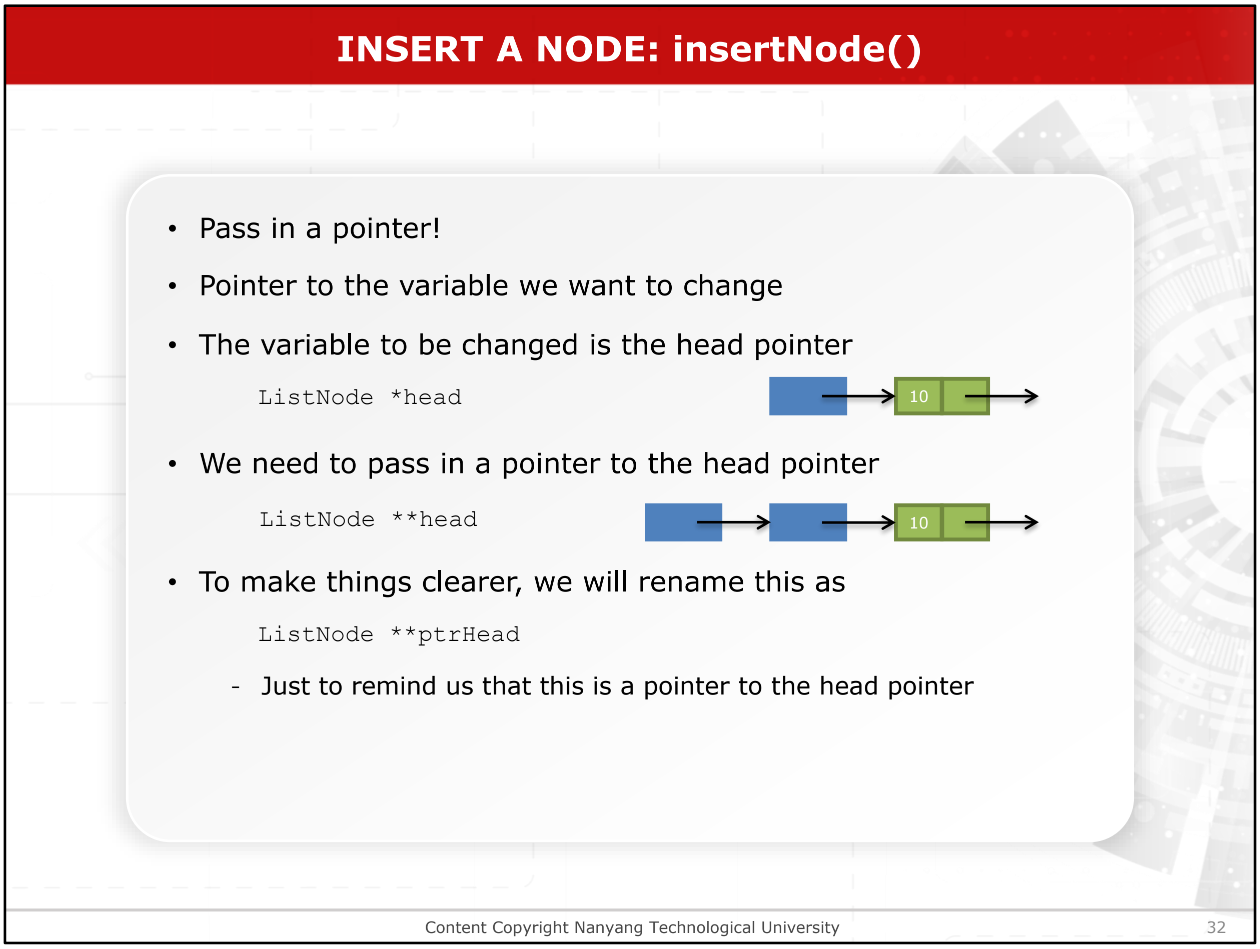
To change the address you must pass in the ADDRESS of the pointer

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

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.

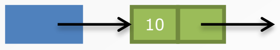
To change it, we need to pass the address of local pointer variable by reference.



INSERT A NODE: insertNode()

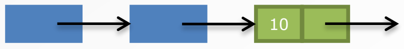
- Pass in a pointer!
- Pointer to the variable we want to change
- The variable to be changed is the head pointer

```
ListNode *head
```



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

```
ListNode **head
```



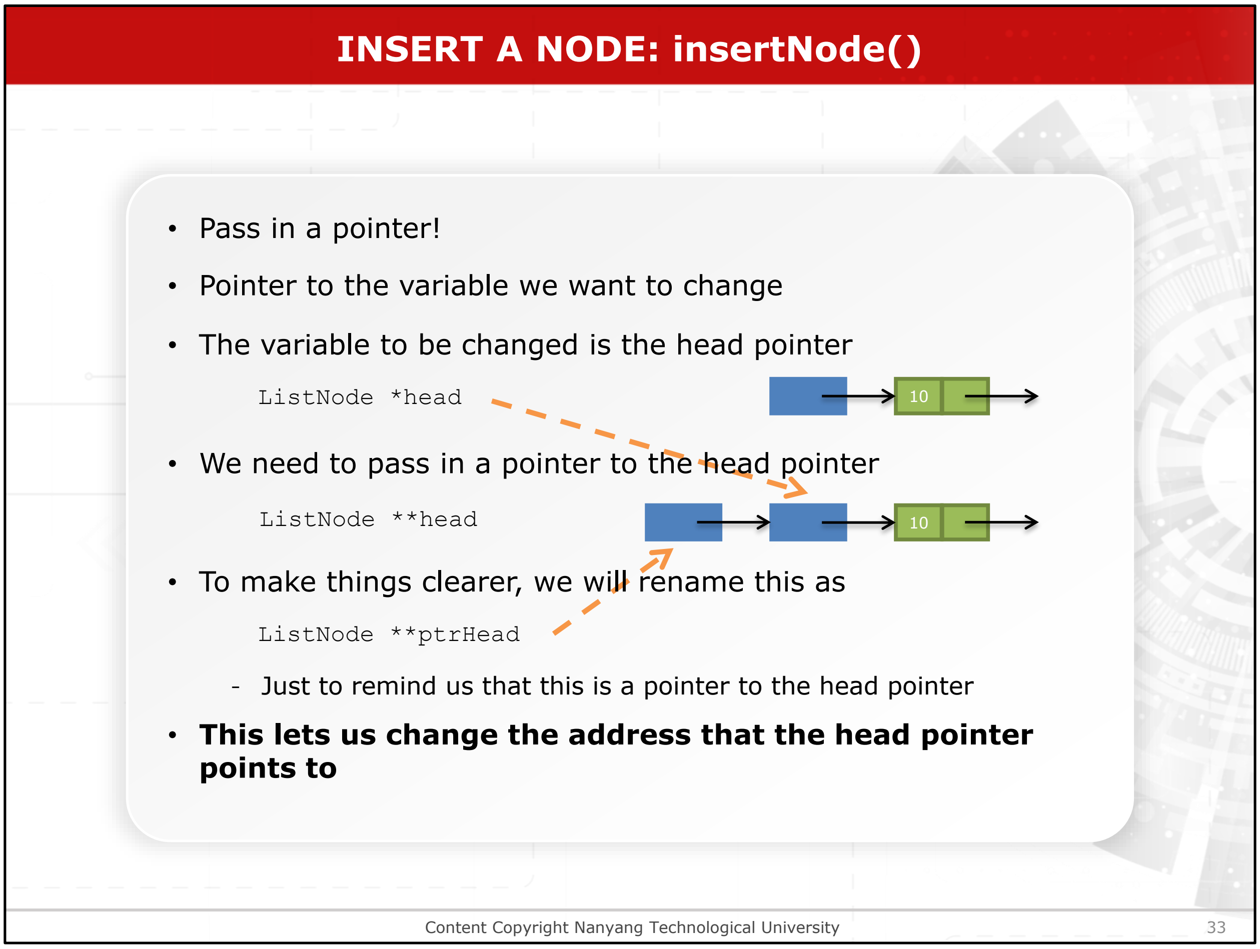
- To make things clearer, we will rename this as

```
ListNode **ptrHead
```

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

32

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
```

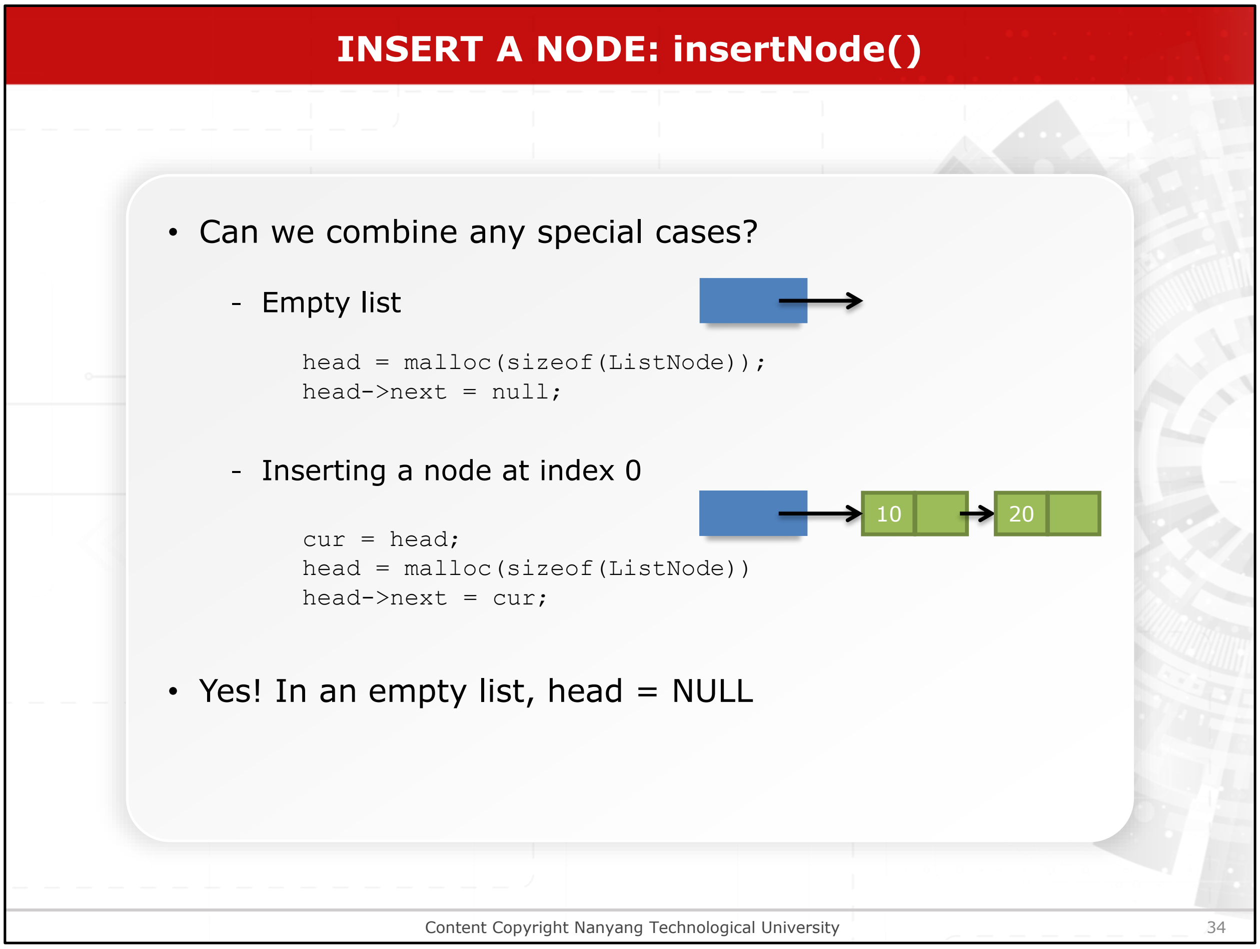
- To make things clearer, we will rename this as

```
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**

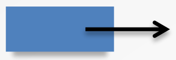
33

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



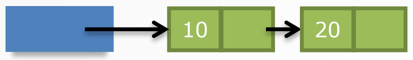
INSERT A NODE: insertNode()

- Can we combine any special cases?
  - Empty list



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

  - Inserting a node at index 0

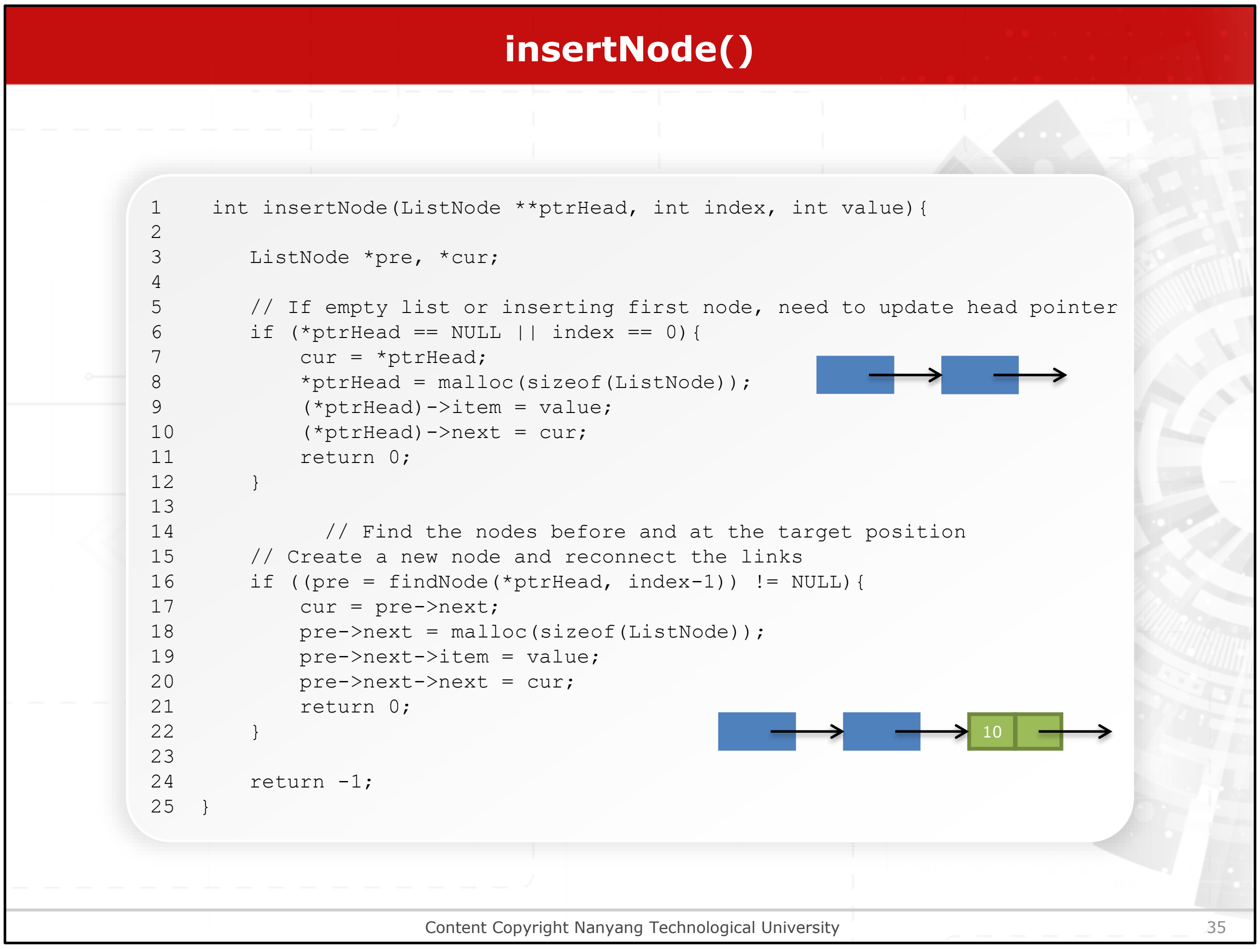


```
cur = head;
head = malloc(sizeof(ListNode))
head->next = cur;
```

- Yes! In an empty list, head = NULL

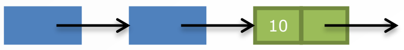
Finally, we can combine the special cases we discussed. For an empty list, head->next = 0. When inserting a node at the index 0, cur= head since it has the address of the first node. Then we create a new node and assign its address to head. Now, head->next=cur because the head->next is the original first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

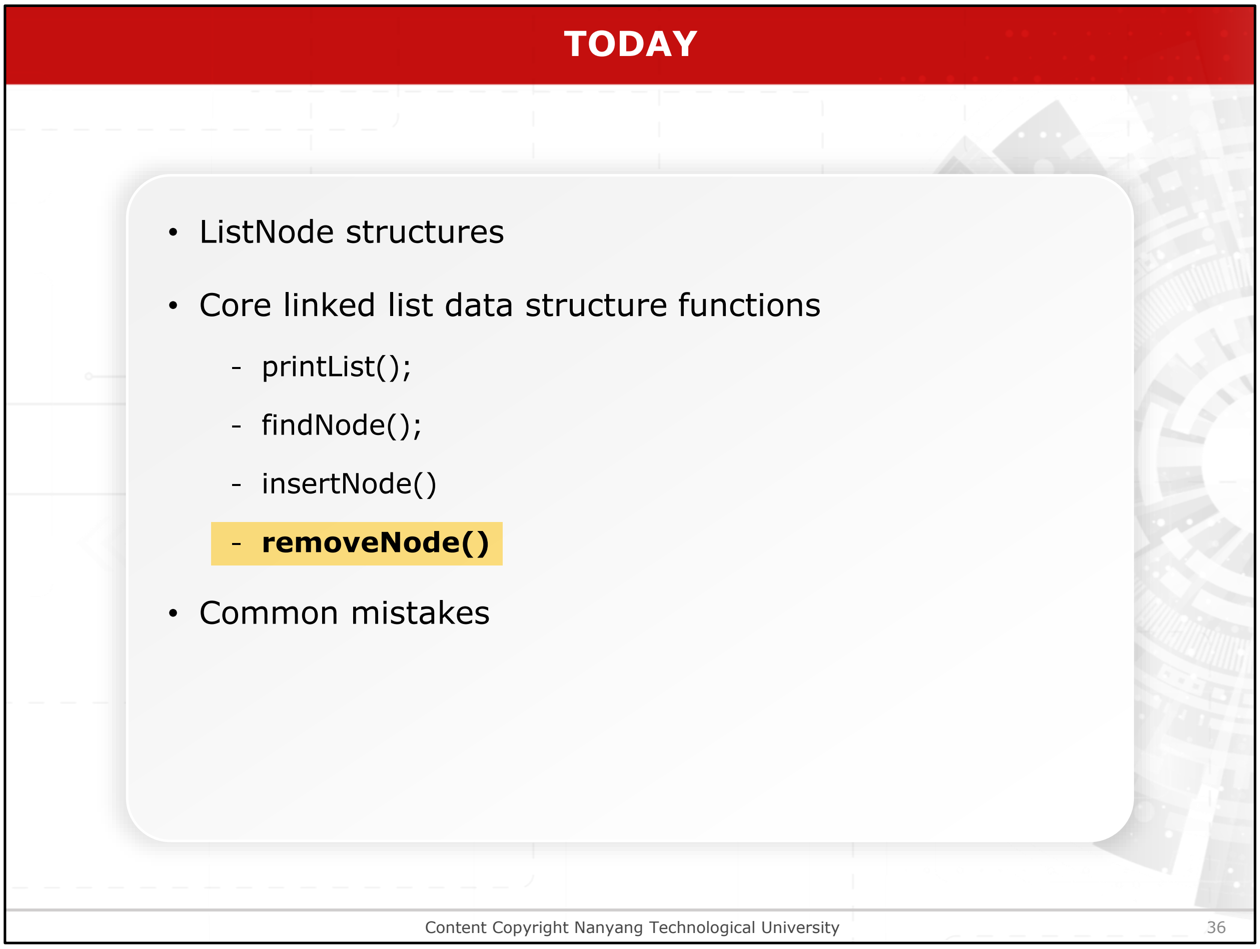


insertNode()

```
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;
}
```



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

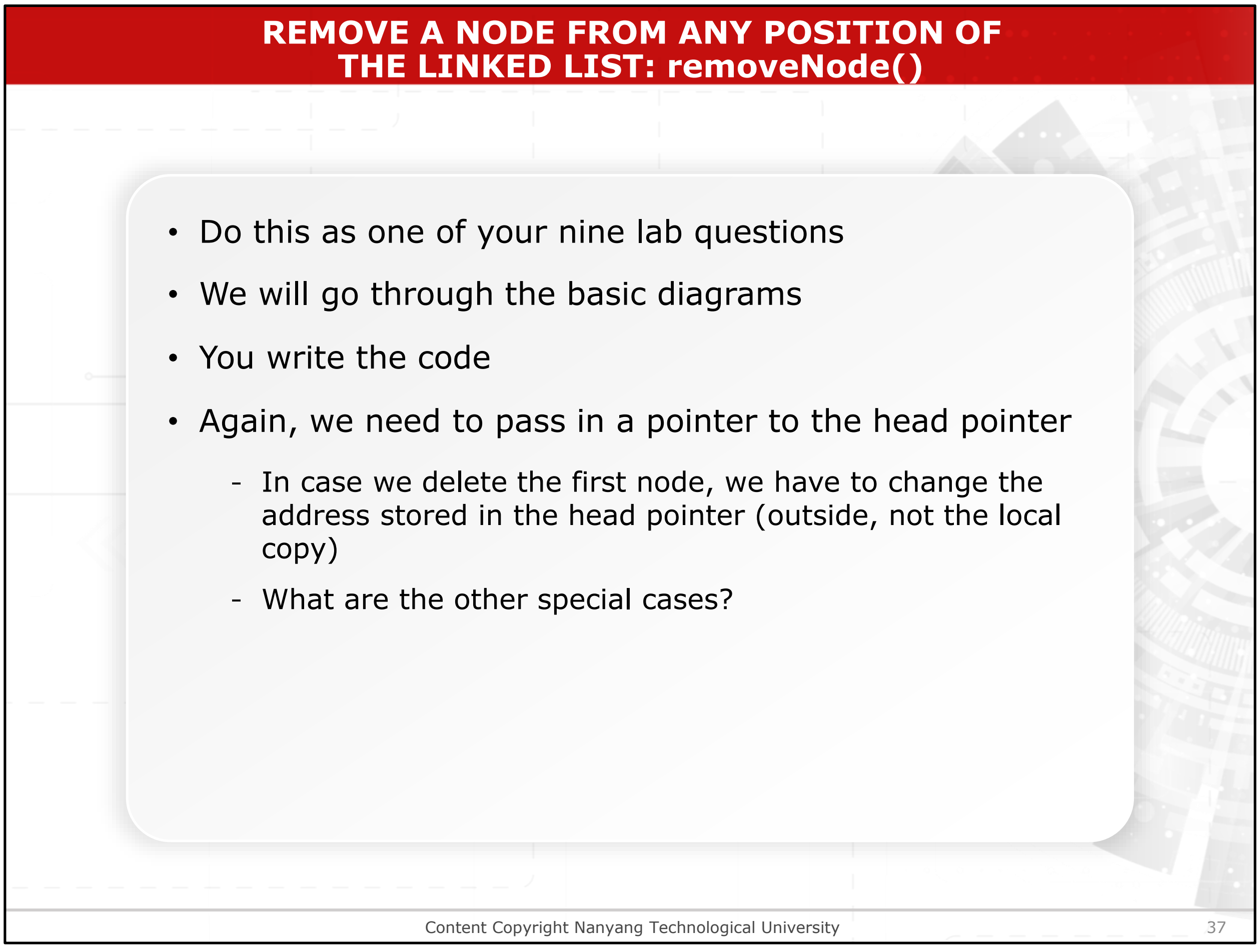


TODAY

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

36

The remove node function is for you to look through for your lab. Use the diagrams given to you as a reference as you write your lab function for remove node.



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?

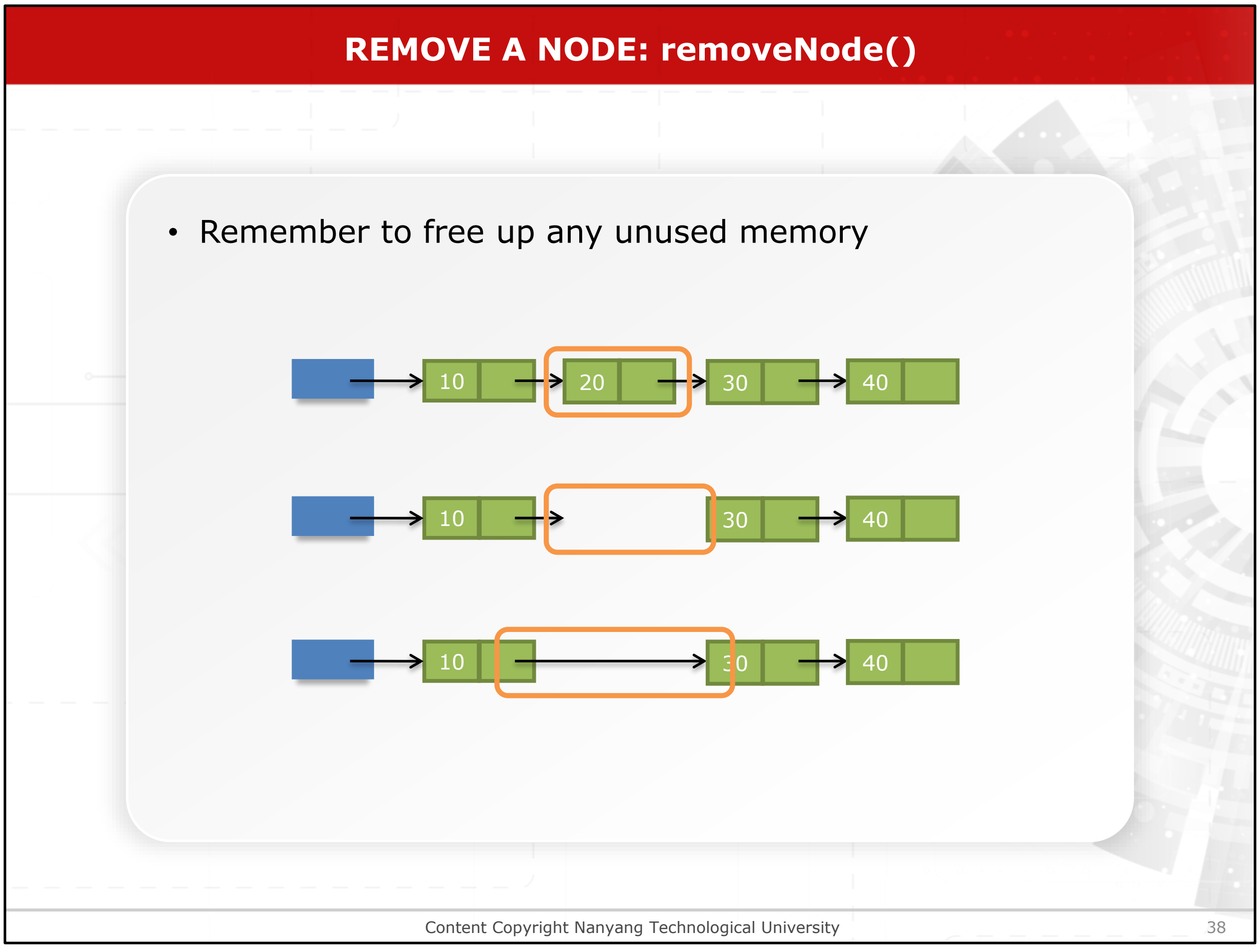
37

This is going to be discussed in your tutorial.

The important thing over here is that when you remove a node, you need to free up the memory that's been given to you. As you may remember, the malloc() function gives you memory from the heap. But if you hold on to that memory when you are not using it anymore, your heap eventually depletes, assuming that your program runs for a long time.

When you remove a node, it's not just as simple as reconnecting the pointers. I would also need to make sure that any memory which I requested using malloc() has been freed properly using the free function.

This list node was allocated using a call to malloc, and when I'm removing this node, I have to make sure that it is properly freed. Yet, there are a few complications here. If we straightaway free the memory for that removing node, we will lose the rest of the list nodes located after it. Even if we link the two nodes besides the nodes that are going to be removed, and free the memory for the node that is going to be removed, we will still lose the address of the node we are going to remove.

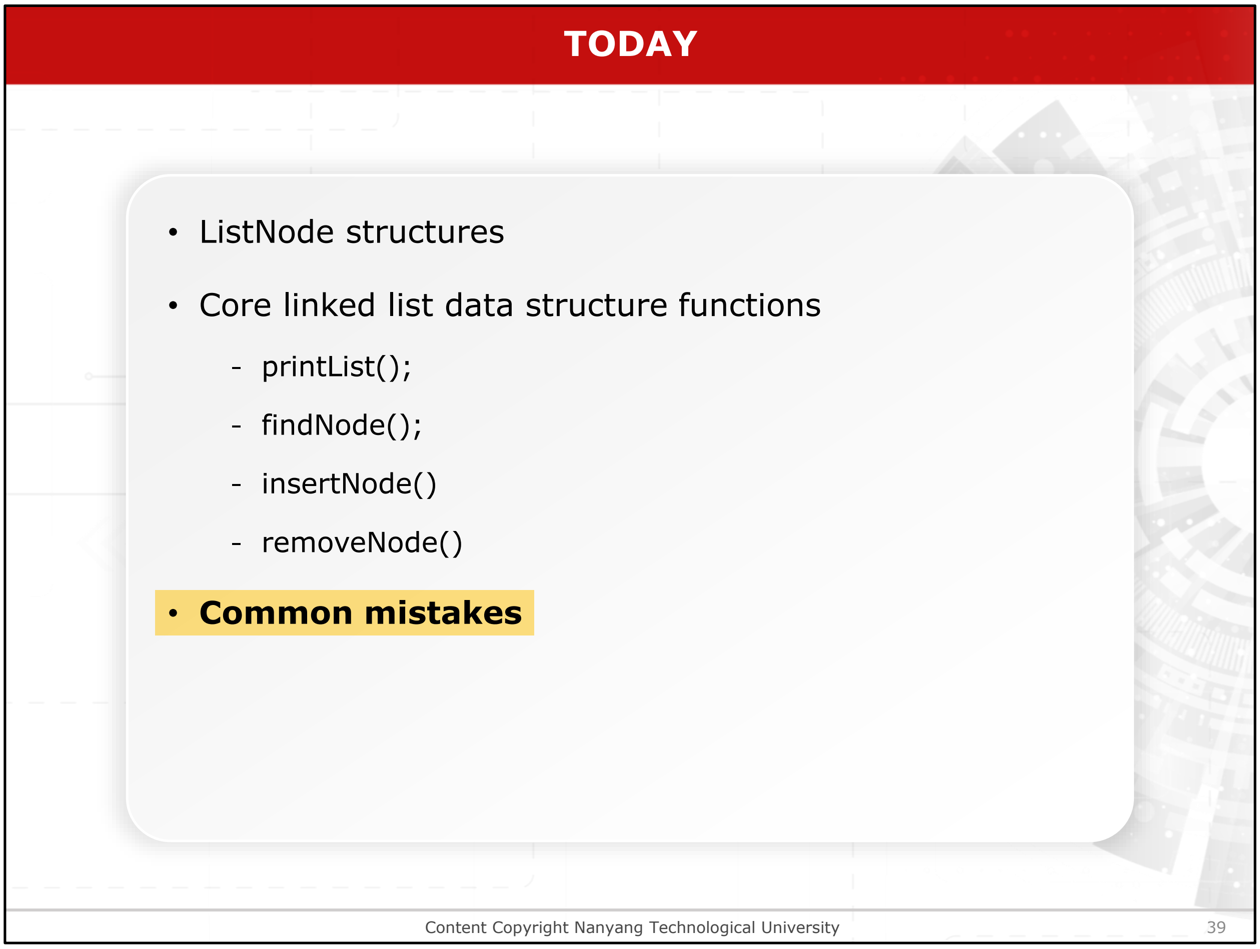


REMOVE A NODE: removeNode()

- Remember to free up any unused memory

38

Remember to free up any unused memory.

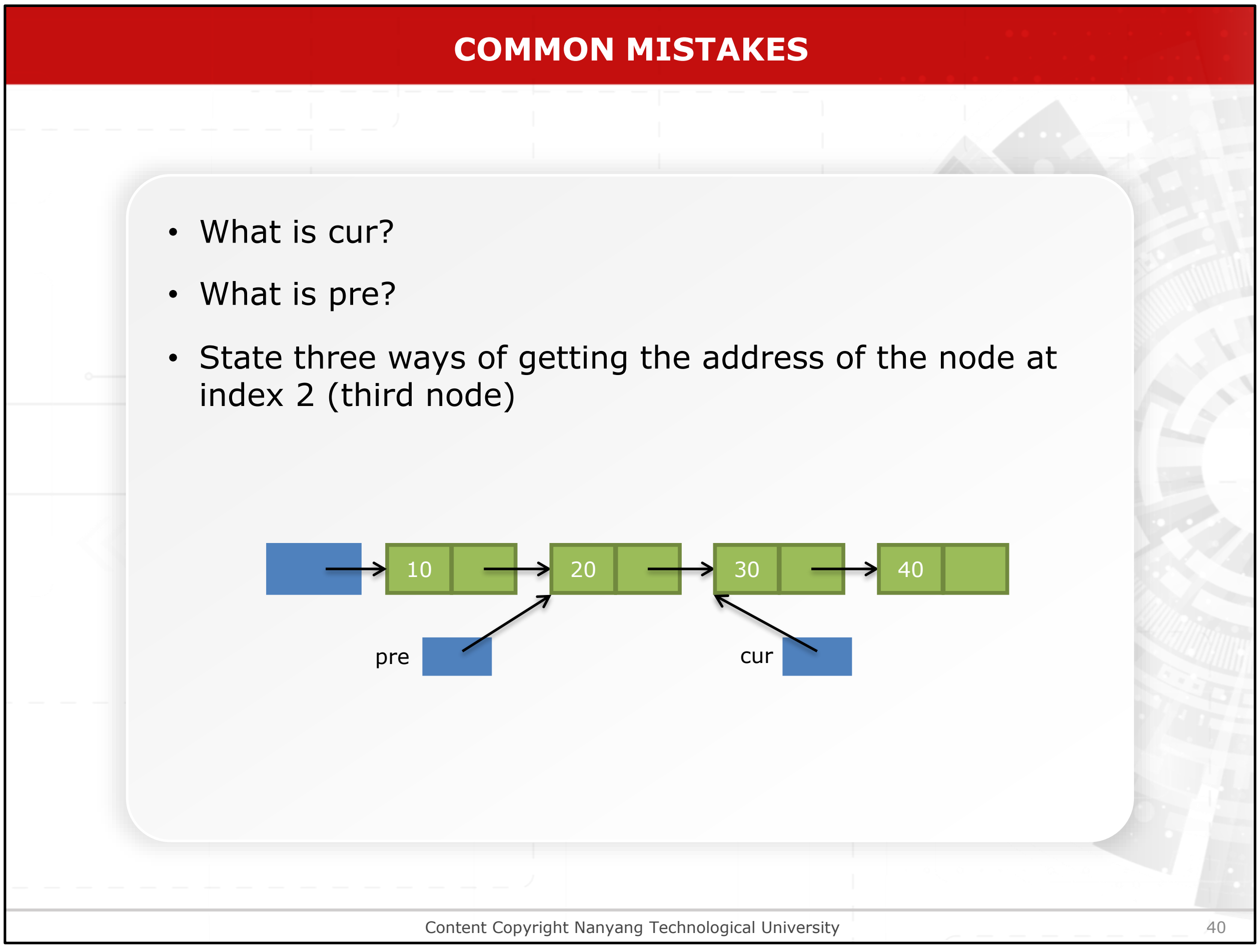


TODAY

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

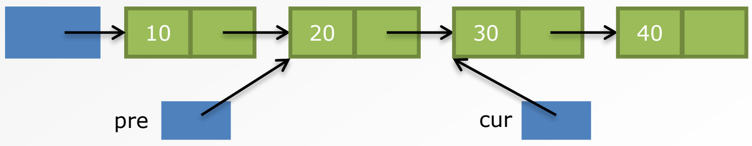
39

Common mistakes

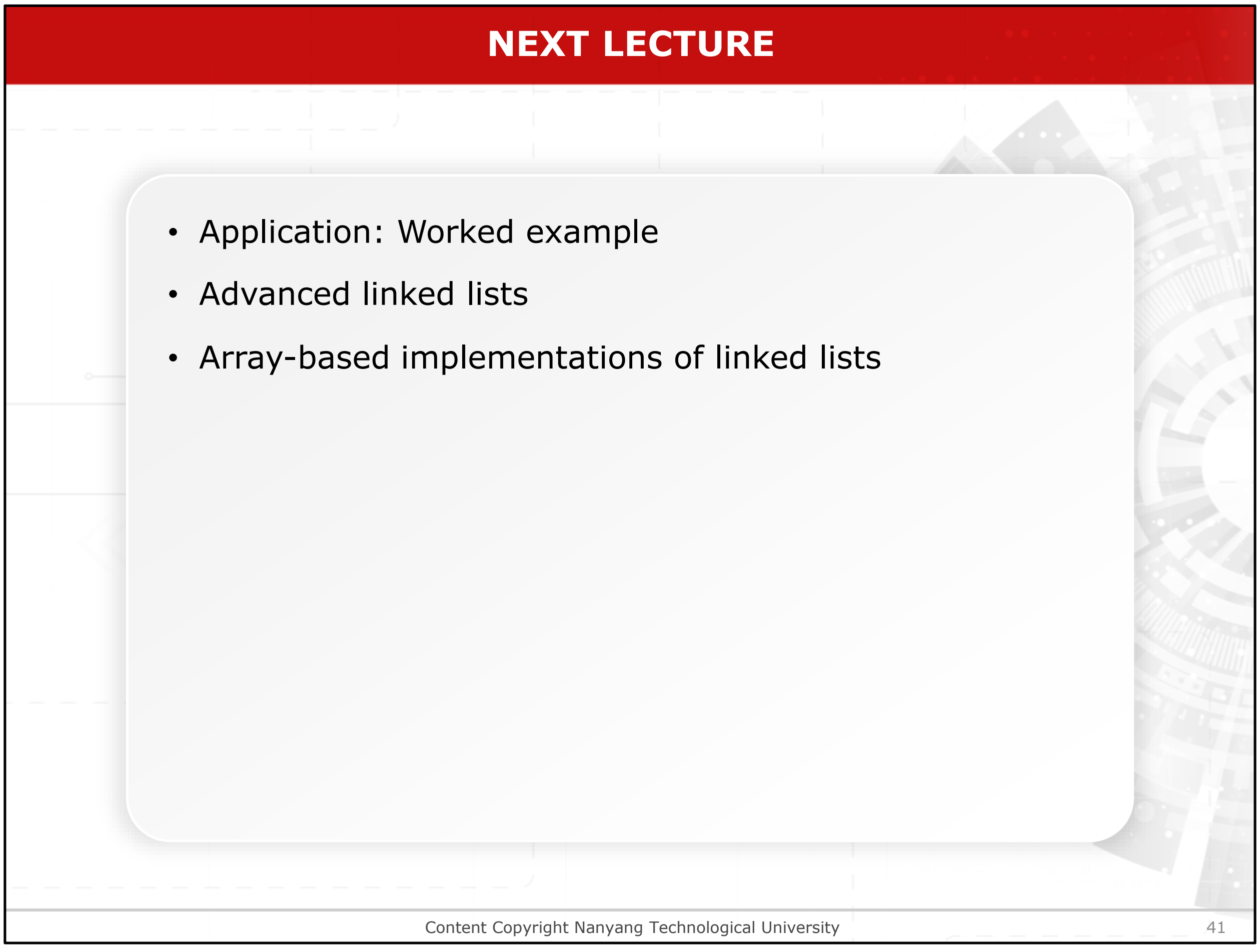


COMMON MISTAKES

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



One of the common mistakes, as we discussed in the previous lecture, is that the head is a pointer. It is not a node. The cur and pre are also pointers which allow storing of addresses of the list node structure.
In the given list, state three ways to get the address of the node at index 2.
The first way is that cur node is already pointing at the node at index 2.
The second way is to get it using pre->next.
The third way is by starting from the head pointer and traversing all the way down.



NEXT LECTURE

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

41

We will discuss the application, advanced linked lists and array-based implementation of linked lists in the next lecture.