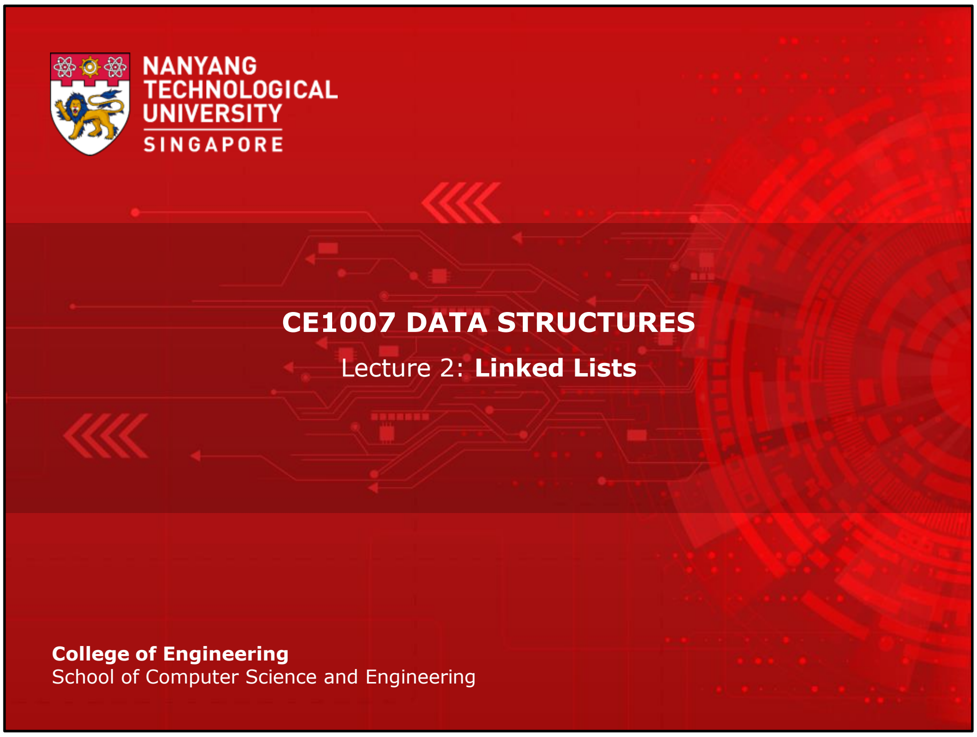


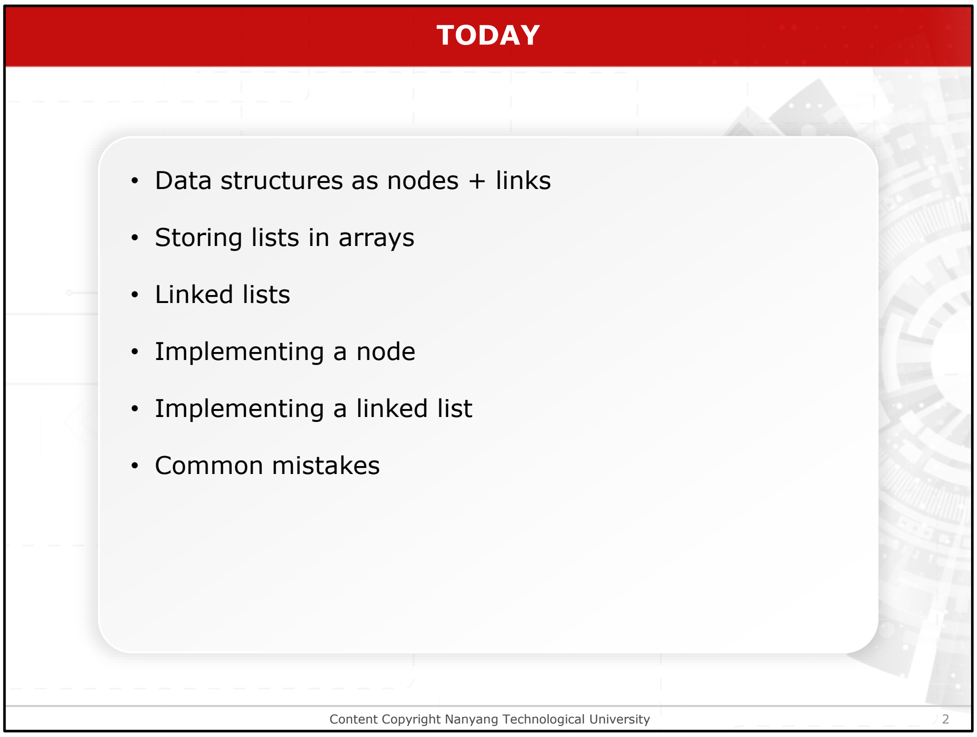
NANYANG TECHNOLOGICAL UNIVERSITY SINGAPORE

CE1007 DATA STRUCTURES

Lecture 2: **Linked Lists**

College of Engineering
School of Computer Science and Engineering

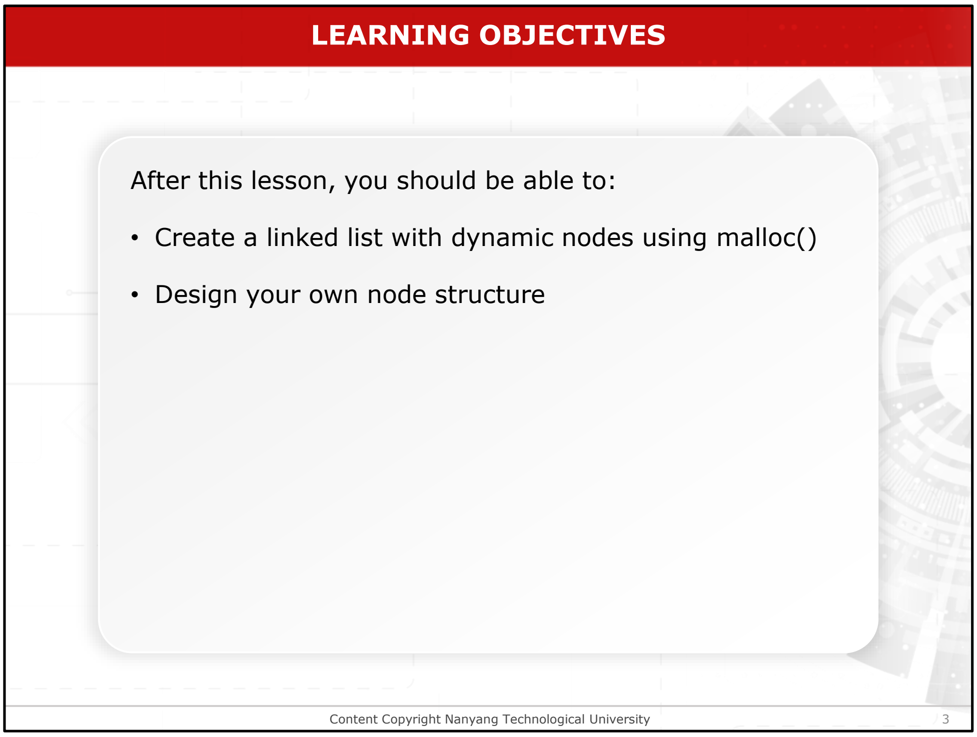
This lecture is on Linked Lists.



TODAY

- Data structures as nodes + links
- Storing lists in arrays
- Linked lists
- Implementing a node
- Implementing a linked list
- Common mistakes

We will discuss about storing information using linear and non-linear data structures. In linear data structures, the data is arranged inline. The pieces of information are lined up without branching out in multiple connections. Instead of arrays, nodes will be connected with each other to form a data structure. Then, we will discuss about the connection between data structures formed by rearranging links between nodes. This will lead us to the idea of linked lists.

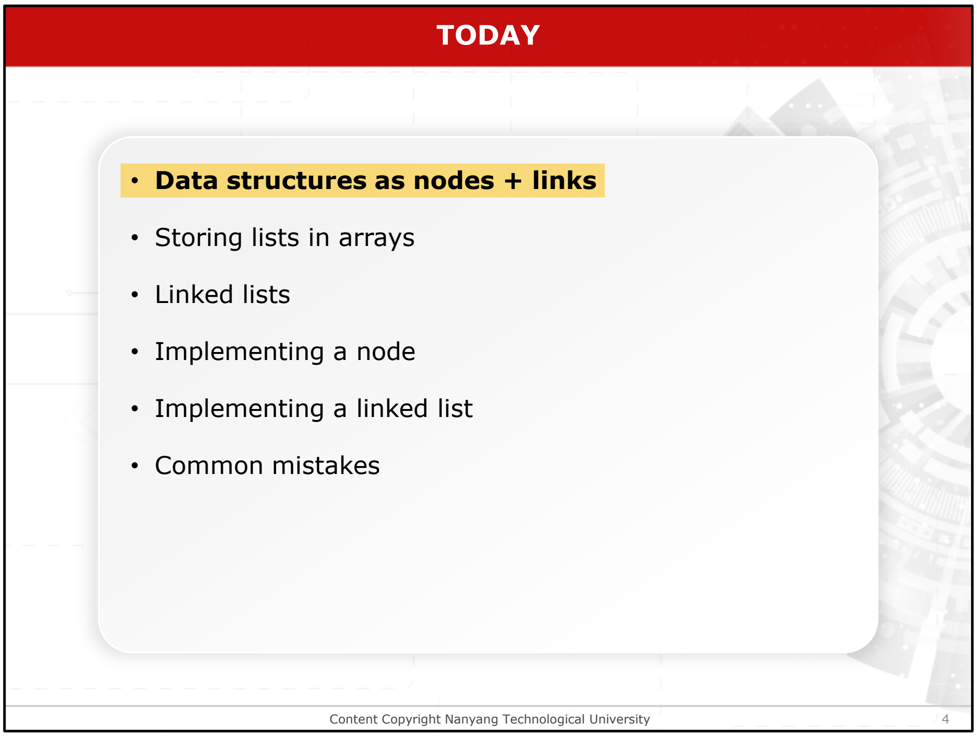


LEARNING OBJECTIVES

After this lesson, you should be able to:

- Create a linked list with dynamic nodes using malloc()
- Design your own node structure

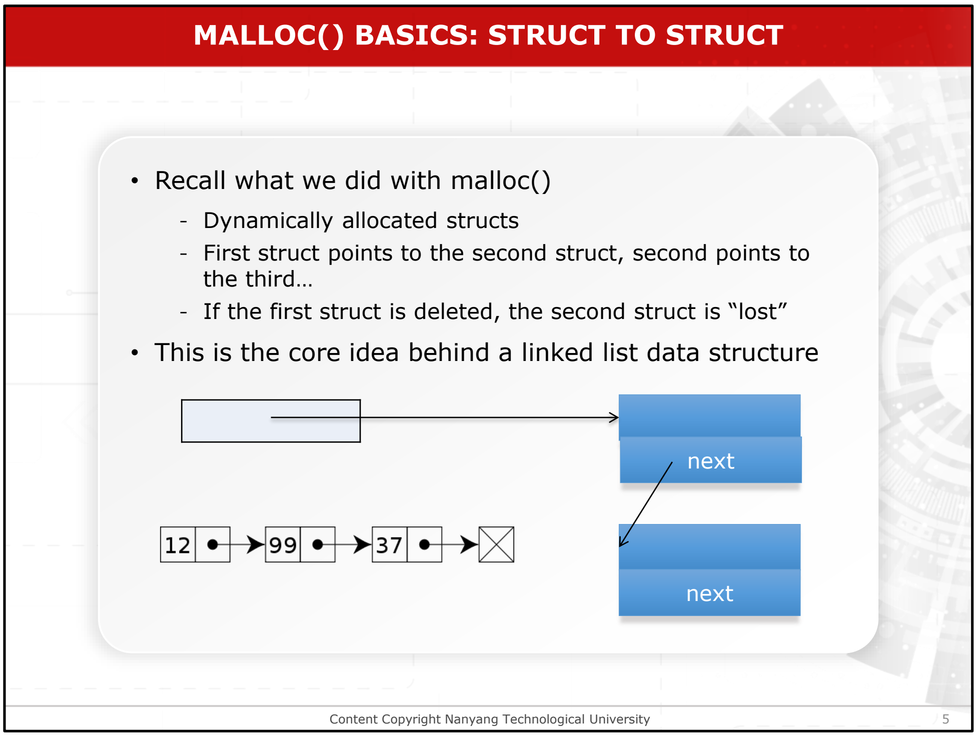
At the end of this lecture, you will be able to create linked lists with dynamic nodes and design node data structures.



TODAY

- **Data structures as nodes + links**
- Storing lists in arrays
- Linked lists
- Implementing a node
- Implementing a linked list
- Common mistakes

Let’s first discuss about data structures as nodes and links.



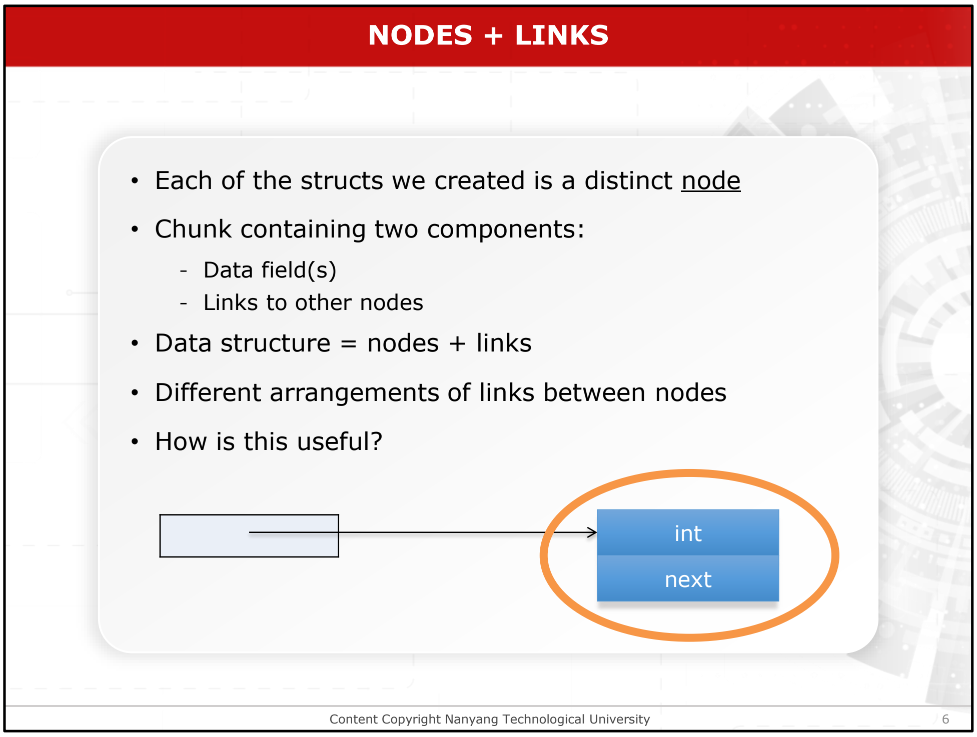
MALLOC() BASICS: STRUCT TO STRUCT

- Recall what we did with malloc()
  - Dynamically allocated structs
  - First struct points to the second struct, second points to the third…
  - If the first struct is deleted, the second struct is "lost"
- This is the core idea behind a linked list data structure

5

We have already discussed the basic idea behind linked lists. As discussed, we can connect structs using pointers and use the same concept when we add a new node; we can link it with the existing linked items using a pointer. Therefore, by setting the pointer values appropriately, we can get a nice linear structure which is called **linked list**.

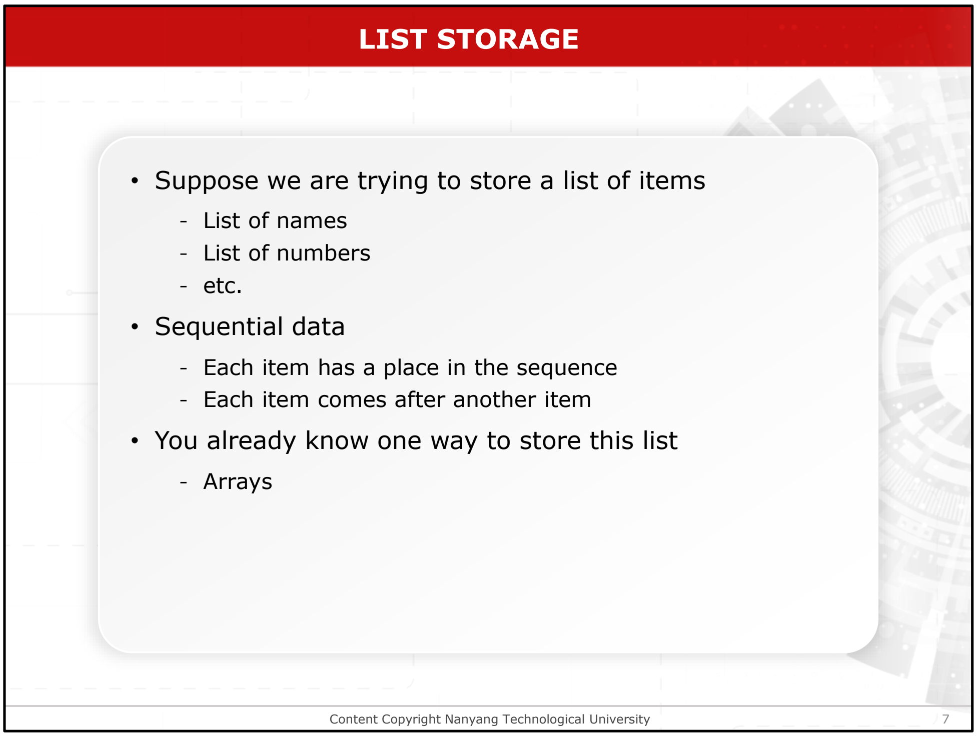
Yet, there are some weaknesses in this arrangement. For example, if you see the given linked list, the 2nd node is linked to the 1st node, the 3rd node to the 2nd, likewise. Now, if you lose track of the first node, you will lose the whole chain of data. We will discuss more about this problem in the next topic.



NODES + LINKS

- Each of the structs we created is a distinct node
- Chunk containing two components:
  - Data field(s)
  - Links to other nodes
- Data structure = nodes + links
- Different arrangements of links between nodes
- How is this useful?

Each node that you declare does not have to store 1 integer all the time. For example, you declare a node to store an array which stores 10 integers. Each data structure contains nodes and links, and each node contains a data field and the pointer to the next node.

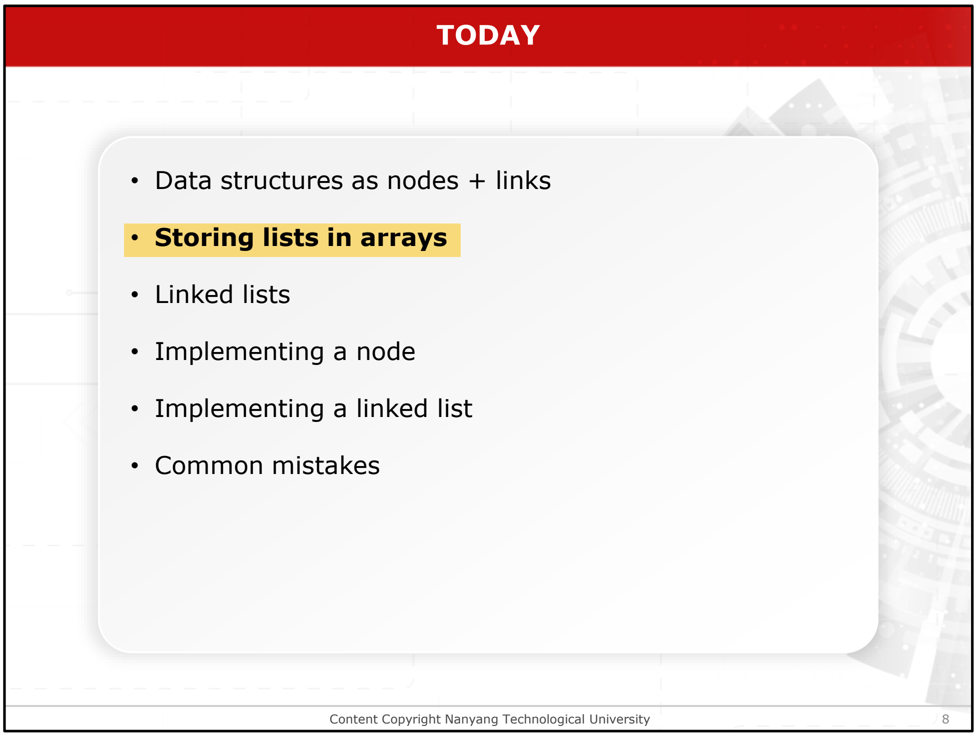


LIST STORAGE

- Suppose we are trying to store a list of items
  - List of names
  - List of numbers
  - etc.
- Sequential data
  - Each item has a place in the sequence
  - Each item comes after another item
- You already know one way to store this list
  - Arrays

7

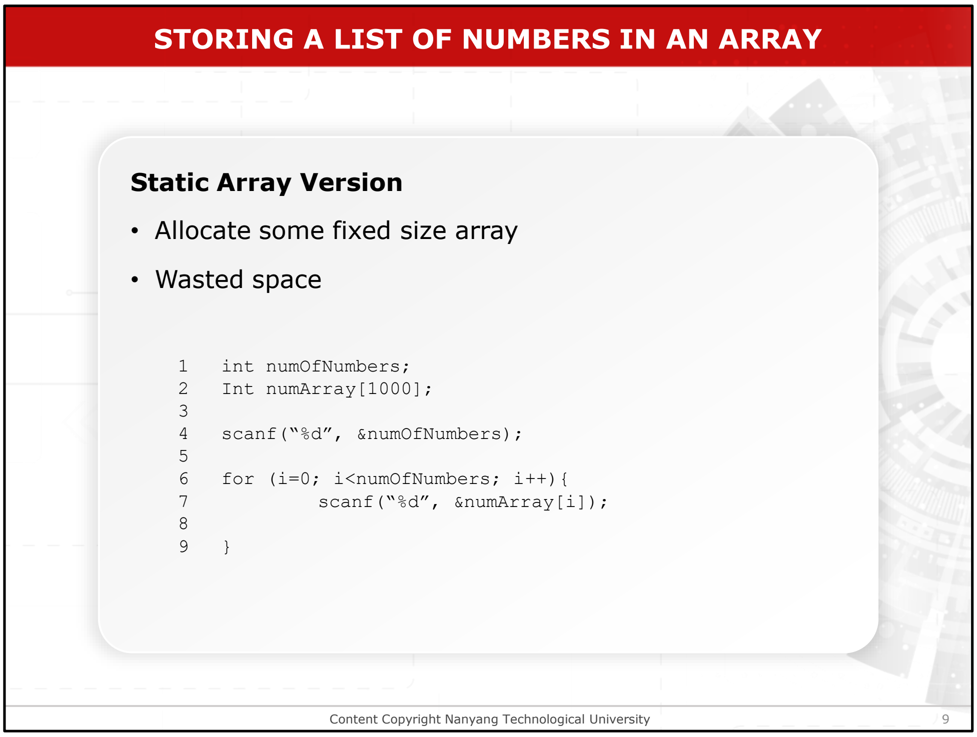
If you are going to store a list of items and the sequence of the items is important, one way to store the lists is arrays.



TODAY

- Data structures as nodes + links
- **Storing lists in arrays**
- Linked lists
- Implementing a node
- Implementing a linked list
- Common mistakes

Let’s discuss how do we store lists in arrays.



STORING A LIST OF NUMBERS IN AN ARRAY

Static Array Version

- Allocate some fixed size array
- Wasted space

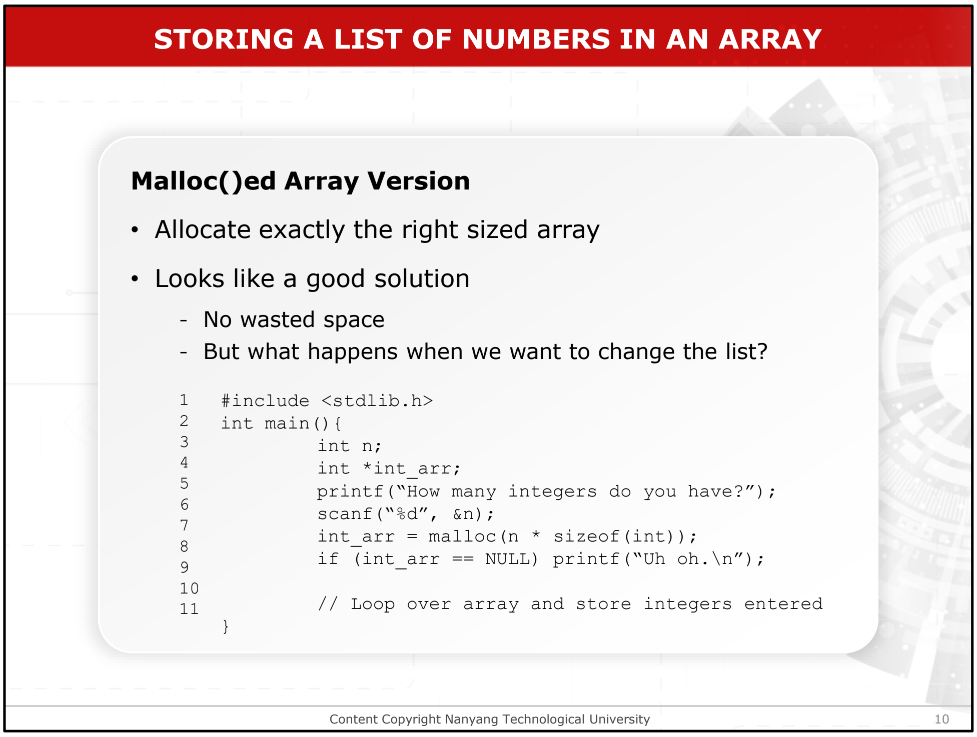
```
int numOfNumbers;
Int numArray[1000];

scanf(“%d”, &numOfNumbers);

for (i=0; i<numOfNumbers; i++){
        scanf(“%d”, &numArray[i]);

}
```

As we have discussed in the previous lecture, arrays are not so optimal because of the problem of wasted space.



STORING A LIST OF NUMBERS IN AN ARRAY

Malloc()ed Array Version

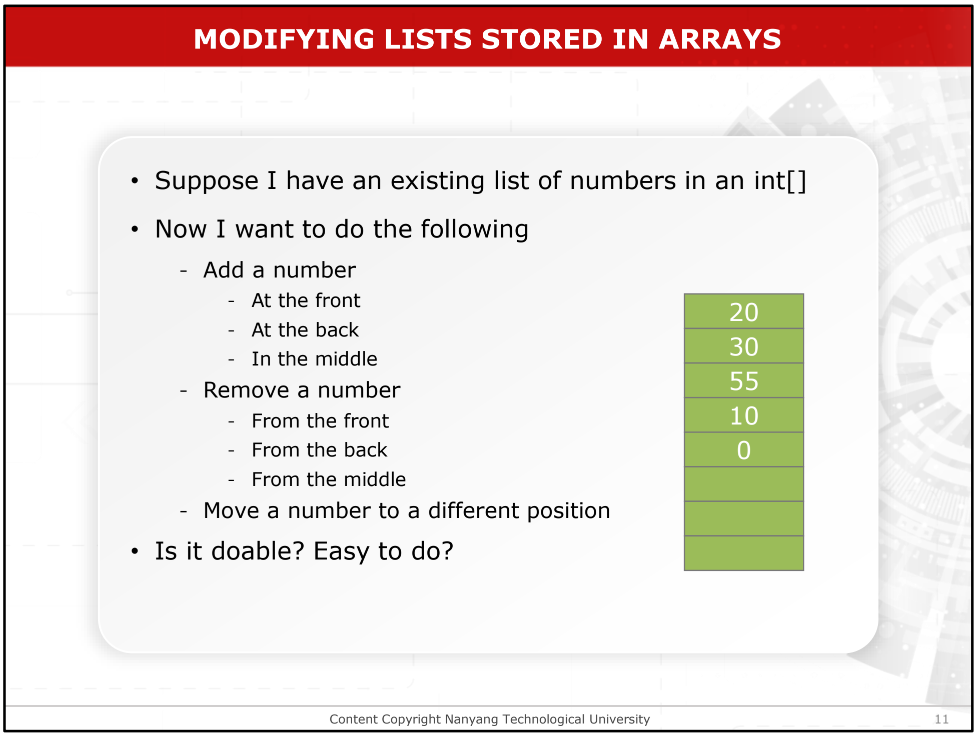
- Allocate exactly the right sized array
- Looks like a good solution
  - No wasted space
  - But what happens when we want to change the list?

```
#include <stdlib.h>
int main(){
        int n;
        int *int_arr;
        printf(“How many integers do you have?”);
        scanf(“%d”, &n);
        int_arr = malloc(n * sizeof(int));
        if (int_arr == NULL) printf(“Uh oh.\n”);

        // Loop over array and store integers entered
}
```

Content Copyright Nanyang Technological University 10

In the previous lecture, you learned to use malloc() function to allocate memory exactly as much as you need. If you have a list of 7 items to store, malloc function allocates enough memory to store the 7 items. But what happens if you want to change the contents of the list?



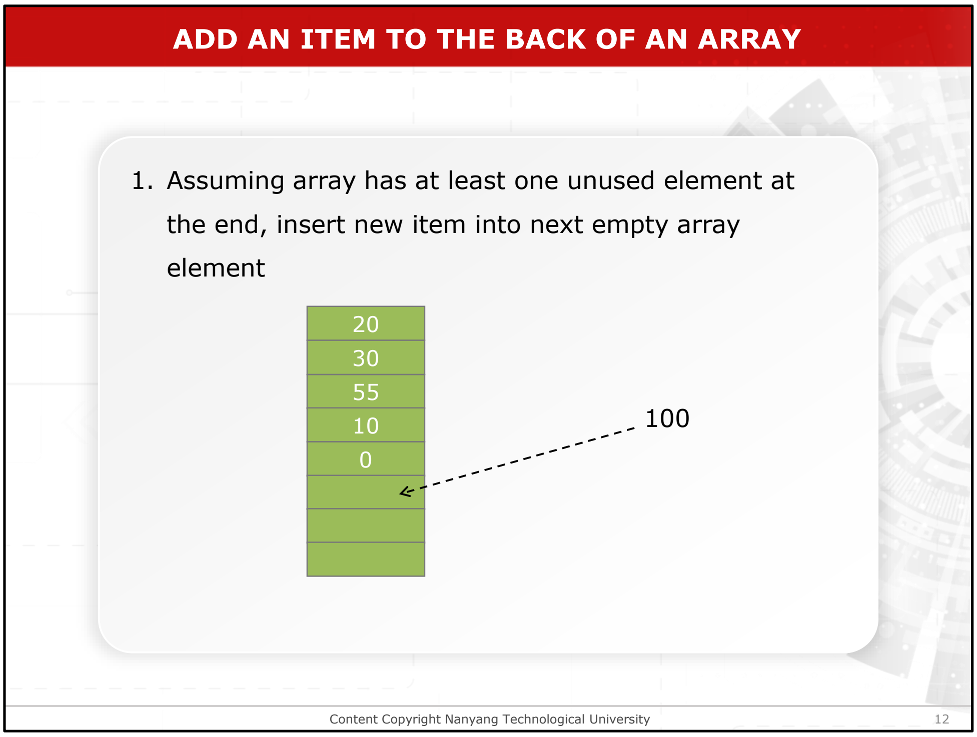
MODIFYING LISTS STORED IN ARRAYS

- Suppose I have an existing list of numbers in an int[]
- Now I want to do the following
  - Add a number
    - At the front
    - At the back
    - In the middle
  - Remove a number
    - From the front
    - From the back
    - From the middle
  - Move a number to a different position
- Is it doable? Easy to do?

20
30
55
10
0

If I have an array of integers, what happens when I try to insert numbers at the front, at the back or in the middle of the array? What if I want to remove a number from the front, from the back or from the middle? What if I want to change the sequence? Can it be done easily?

As an example, the given array has five numbers, 20, 30, 55, 10 and 0 with three empty spaces. Let's use this array to find out how we can add or remove items from an existing list.



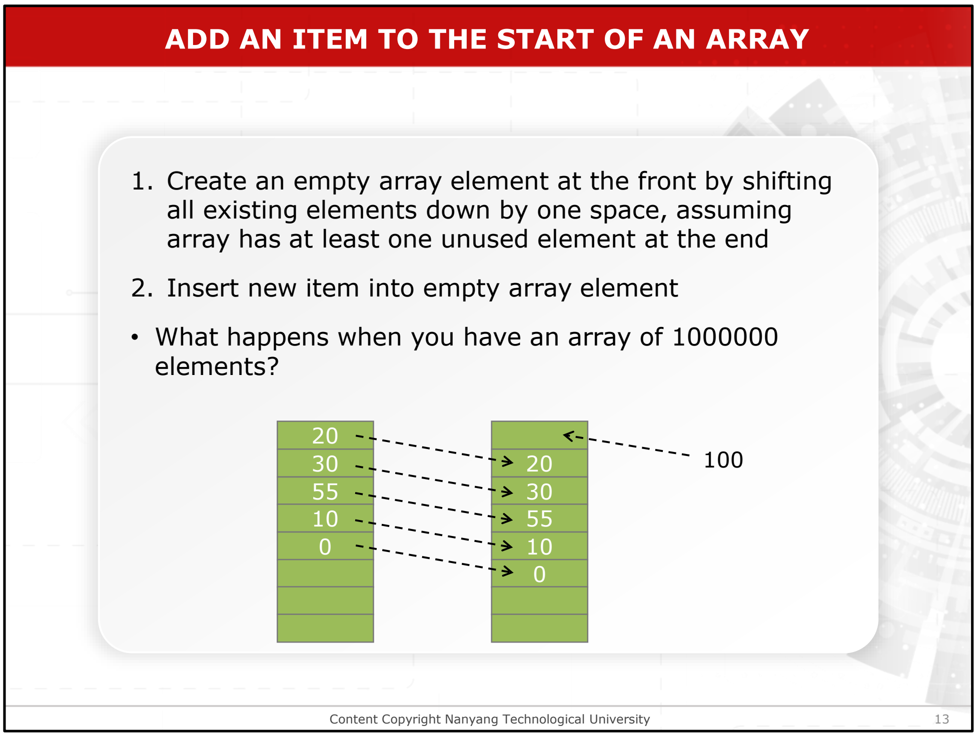
ADD AN ITEM TO THE BACK OF AN ARRAY

1. Assuming array has at least one unused element at the end, insert new item into next empty array element

Content Copyright Nanyang Technological University

12

To add an item to the back of an array, assuming that you have extra space already, we just need to find the next available space and write the item in that space.



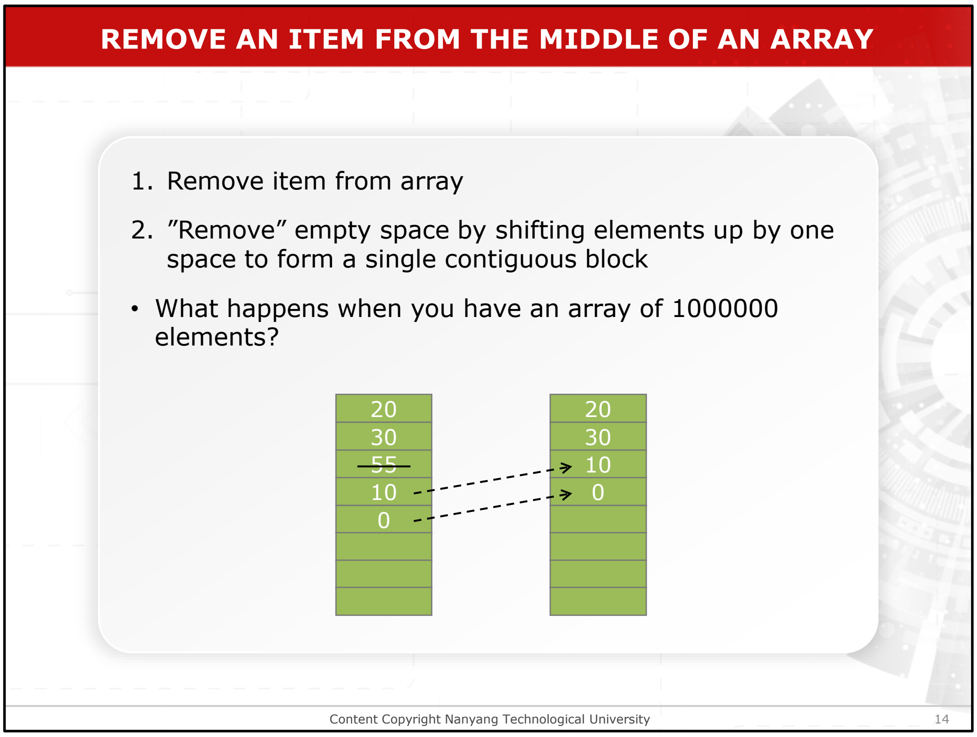
ADD AN ITEM TO THE START OF AN ARRAY

1. Create an empty array element at the front by shifting all existing elements down by one space, assuming array has at least one unused element at the end
2. Insert new item into empty array element

- What happens when you have an array of 1000000 elements?

20
30
55
10
0
20
30
55
10
0
100

Adding an item at the beginning of an array is a little bit complicated than the previous example because now we do not have free space at the front of the array. For example, to add a 100 at the front of the array, I have to move all other items down by one. For five items, moving all of them down by one would seem easy. But, what if I have 1 million numbers in the array? And what if I want to add two numbers to the front of the array, one after the other? Then you need to add one number by moving 1 million numbers down by one, and then adding the other number by moving 1million+1 numbers down by one.



REMOVE AN ITEM FROM THE MIDDLE OF AN ARRAY

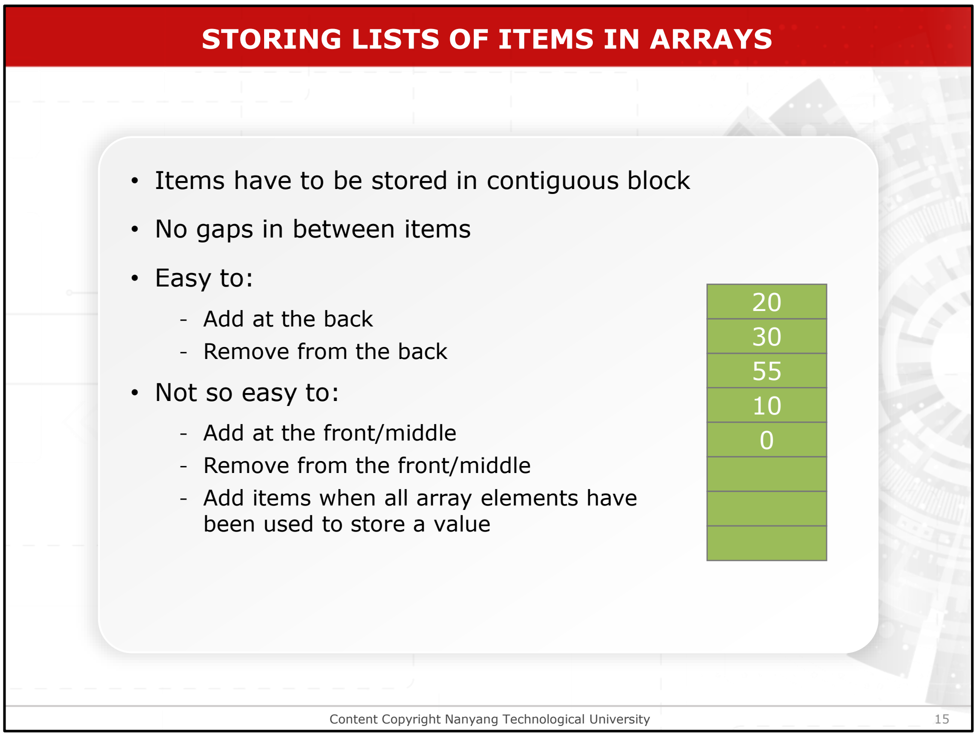
1. Remove item from array
2. "Remove" empty space by shifting elements up by one space to form a single contiguous block

- What happens when you have an array of 1000000 elements?

14

Same as adding, we may need to remove items from the middle of an array. To keep the array contiguous we should not have empty spaces between the list items. Therefore, once we remove an item from the middle of an array, we need to move every item listed below that removed item upwards by one space.

For example, if we have an array with N number of elements, and we removed the 2nd element of the array, we need to move N-2 number of elements upwards by one space.

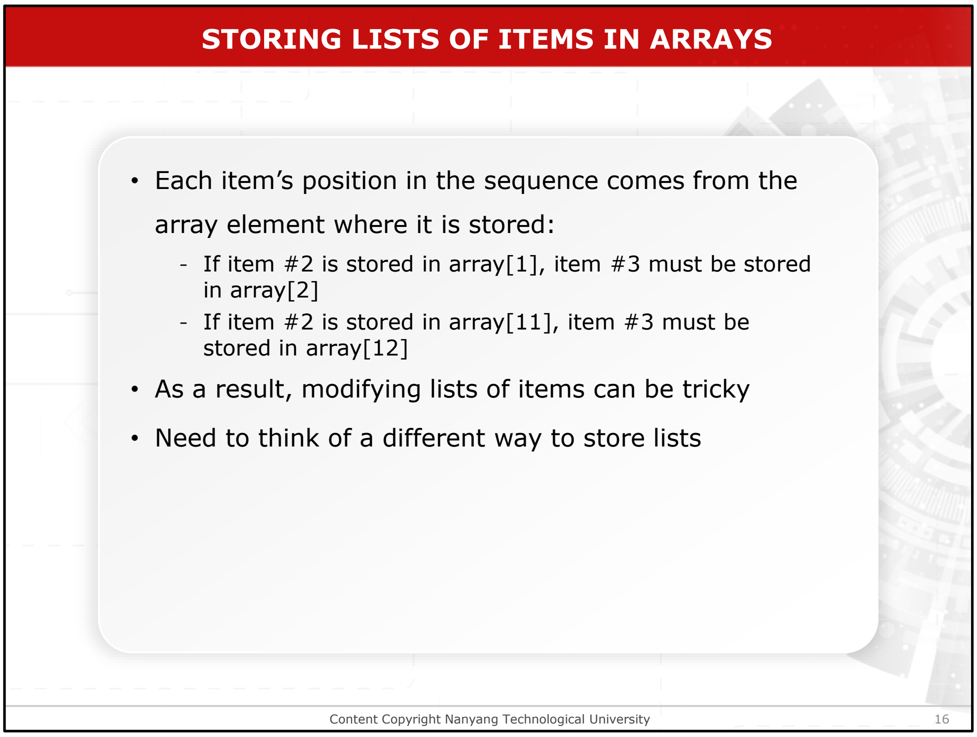


STORING LISTS OF ITEMS IN ARRAYS

- Items have to be stored in contiguous block
- No gaps in between items
- Easy to:
  - Add at the back
  - Remove from the back
- Not so easy to:
  - Add at the front/middle
  - Remove from the front/middle
  - Add items when all array elements have been used to store a value

20
30
55
10
0

Storing list of items in an array can result in a number of problems because the items should be contiguous. Adding and removing list items from the back is easy. But, adding and removing items from the front and middle is a little bit complex. Also, adding items to a list where there is no space is also troublesome.



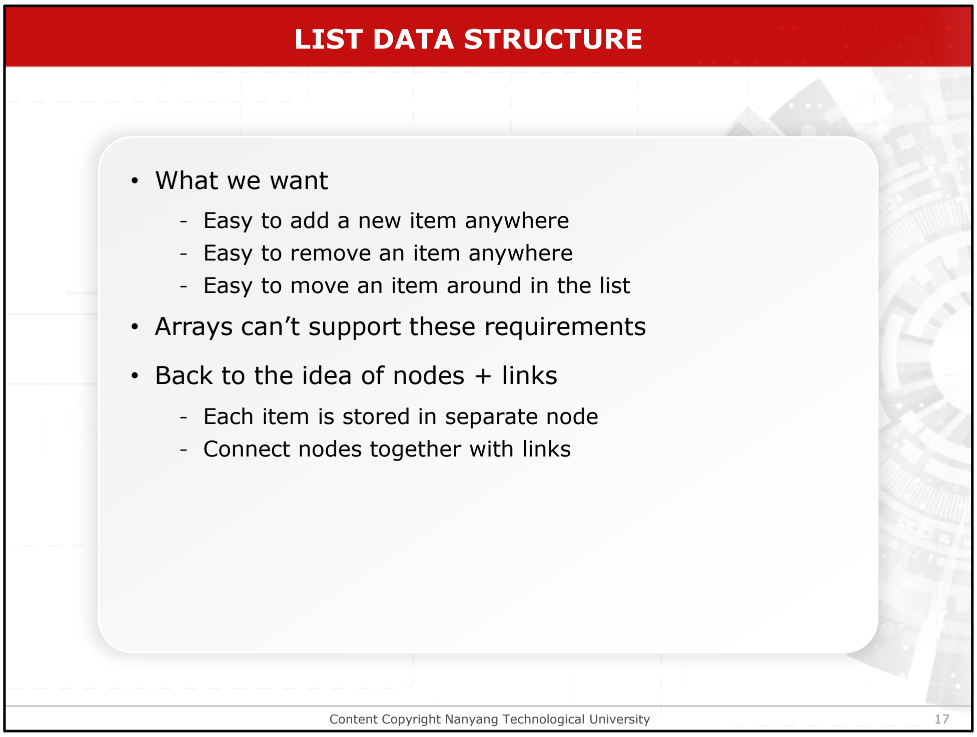
STORING LISTS OF ITEMS IN ARRAYS

- Each item's position in the sequence comes from the array element where it is stored:
  - If item #2 is stored in array[1], item #3 must be stored in array[2]
  - If item #2 is stored in array[11], item #3 must be stored in array[12]
- As a result, modifying lists of items can be tricky
- Need to think of a different way to store lists

Content Copyright Nanyang Technological University

16

Because of the discussed problems, we need to think about a different way of storing lists.

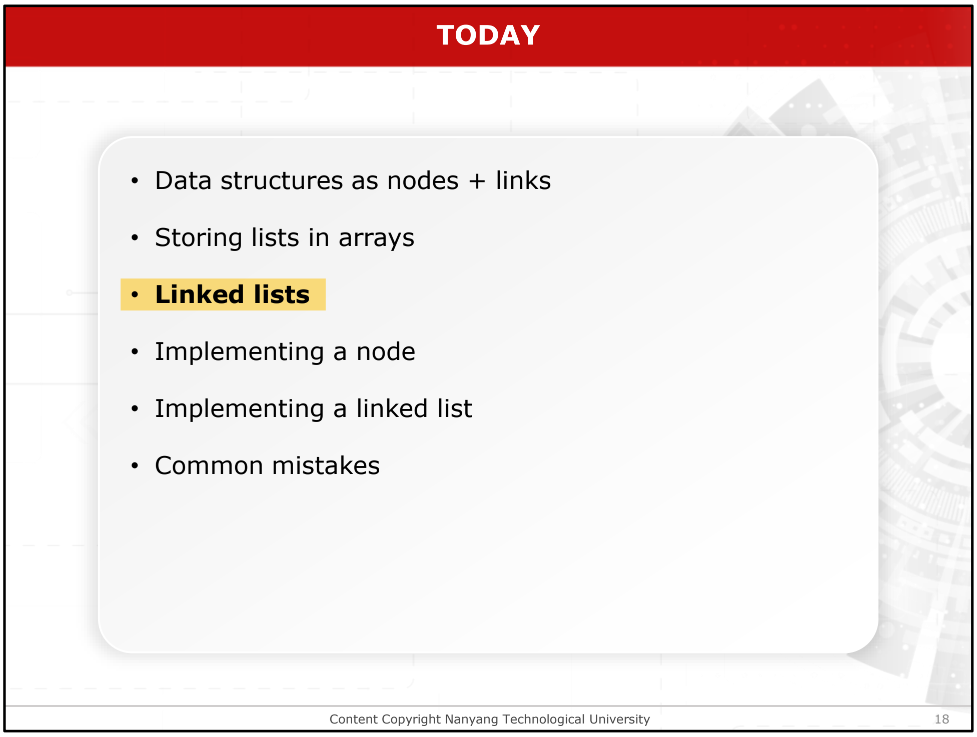


LIST DATA STRUCTURE

- What we want
  - Easy to add a new item anywhere
  - Easy to remove an item anywhere
  - Easy to move an item around in the list
- Arrays can't support these requirements
- Back to the idea of nodes + links
  - Each item is stored in separate node
  - Connect nodes together with links

To solve the previously discussed problems, we need a data structure which requires the same number of steps when adding, removing or moving items in a list, from the front, back and the middle of the list. The number of items in the list should not matter, and therefore, as an example, the number of steps should be similar to both the list with 5 items and the list with 1 million items.

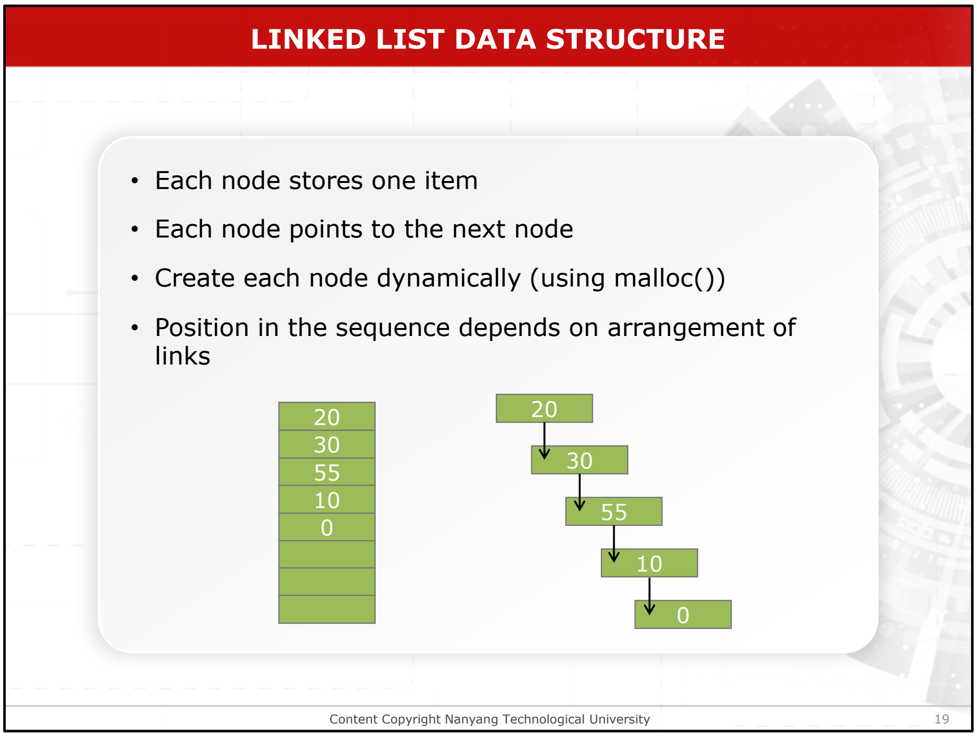
As we have discussed in the previous slides, arrays do not support these requirements, and because of that, we come back to the idea of nodes and links.



TODAY

- Data structures as nodes + links
- Storing lists in arrays
- **Linked lists**
- Implementing a node
- Implementing a linked list
- Common mistakes

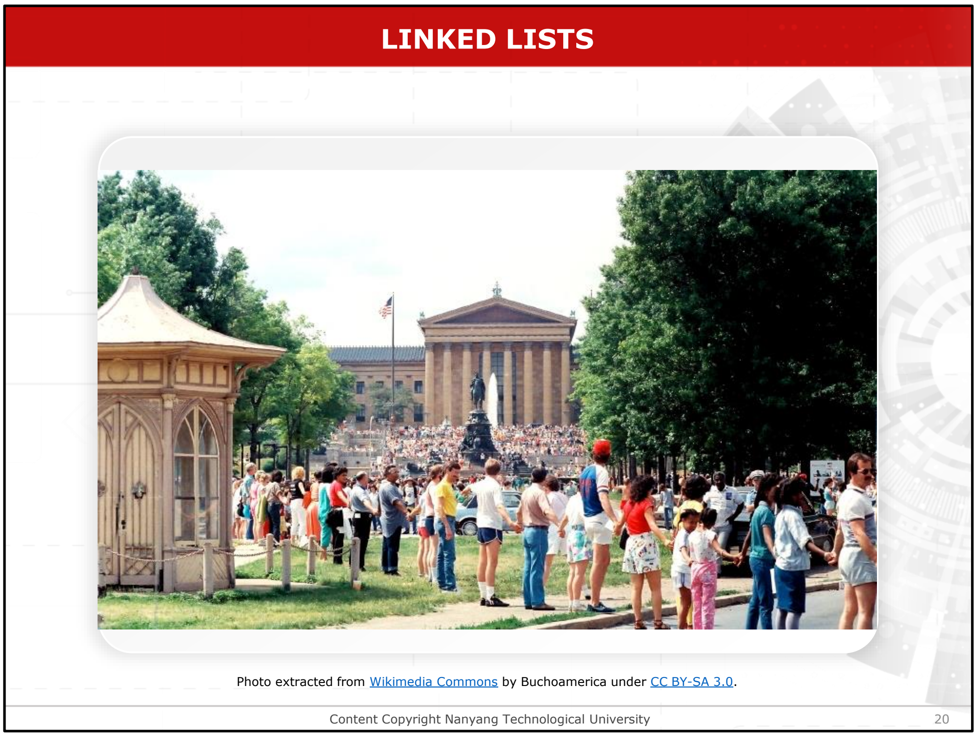
Let’s discuss about linked lists.



LINKED LIST DATA STRUCTURE

- Each node stores one item
- Each node points to the next node
- Create each node dynamically (using malloc())
- Position in the sequence depends on arrangement of links

The image on the left is an array while on the right side, there is a linked list. The power of a linked list or node based data structures is that the links are easy to break by storing the correct address in the pointer variable which connects to the next node. Therefore, we do not need to worry about the specific space on which we are going to store the new item. The nodes will be placed in the same place while we rearrange the pointers. As long as the arrows are moved appropriately, we can recreate the arrangement of the items in a list.

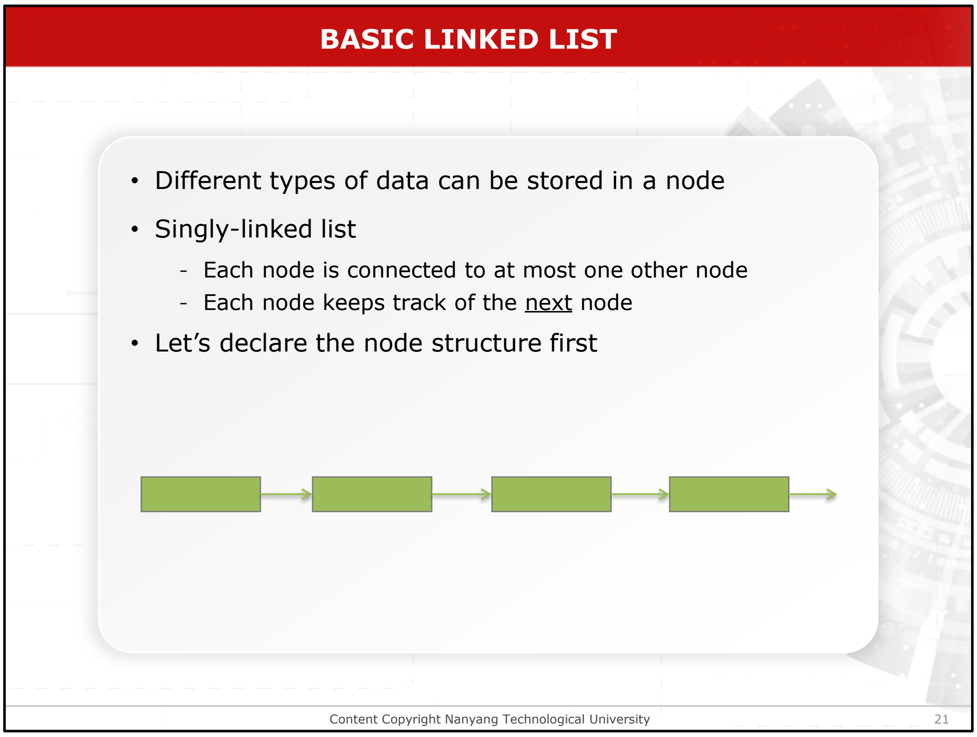


LINKED LISTS

Photo extracted from Wikimedia Commons by Buchoamerica under CC BY-SA 3.0.

Consider a linked list using the analogy of people standing in line and holding hands. If a person wants to join the middle part of the line, how does that person join in?

The person should not consider the number of people in the line or where he should join in. He just needs to take two people who are holding hands, break them apart and grab their hands with his. The connection to the left and right side of the person is all that matters, regardless of the number of people standing in the line or to where he tries to join.

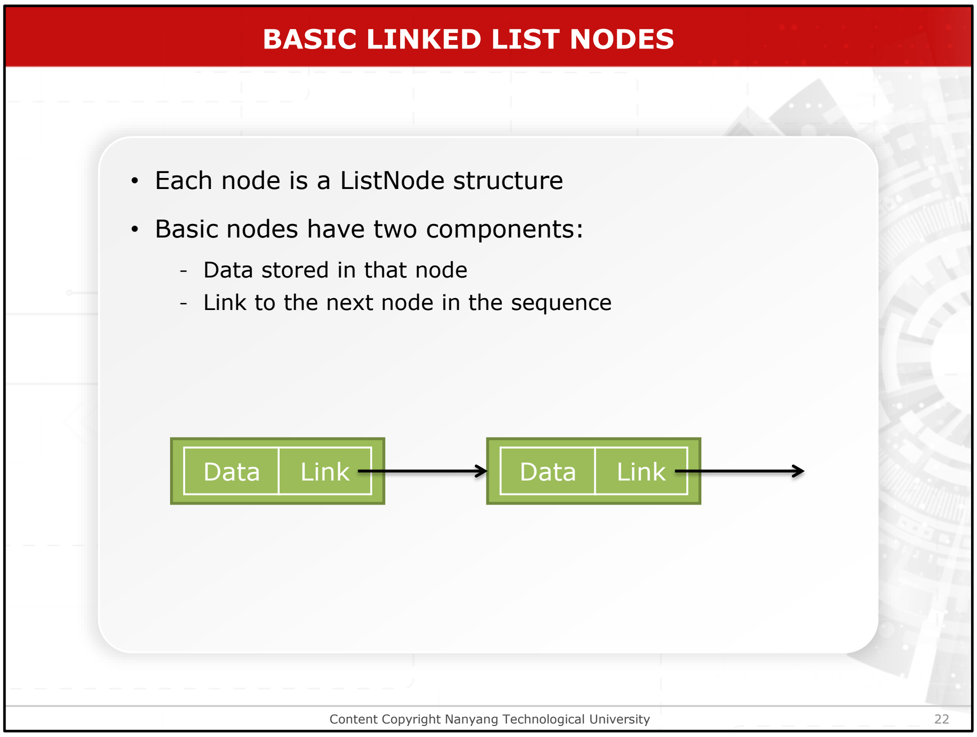


BASIC LINKED LIST

- 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
- Let’s declare the node structure first

Content Copyright Nanyang Technological University 21

We will start by dealing with simple lists which stores integers. Also, we will restrict ourselves to use a singly-linked list where each node is connected to at most one other node. Each node keeps track of the next node which comes after it as the first node keeps track of the second node, and that keeps track of the third node and so on. Before focusing on building linked list structure, we need to focus on individual node creation because it stores the data and the address to the next node.



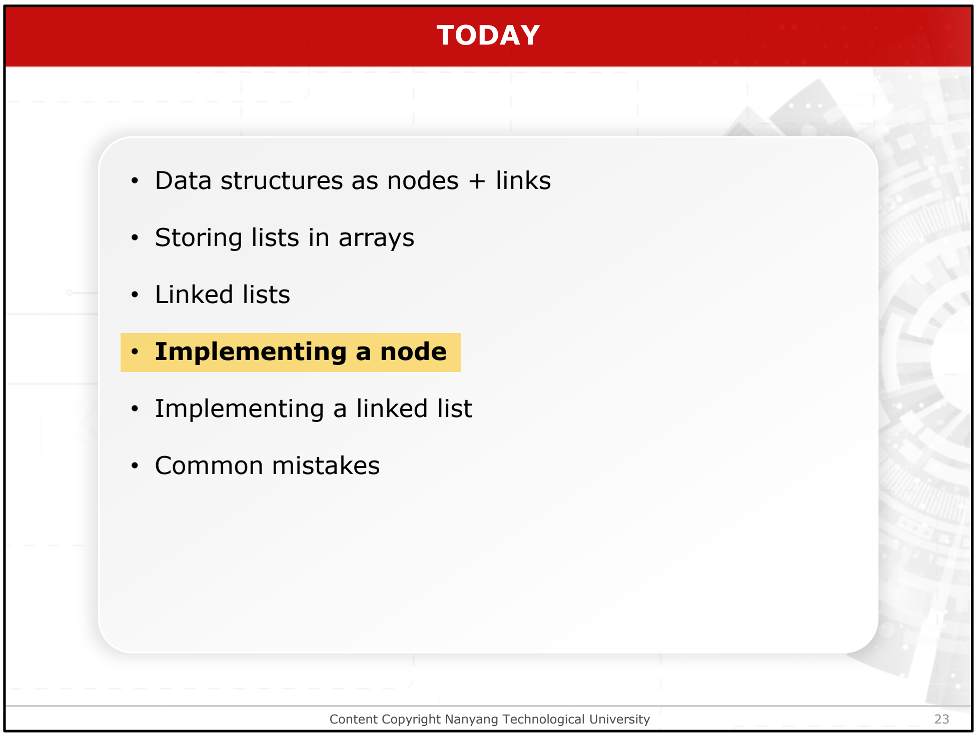
BASIC LINKED LIST NODES

- Each node is a ListNode structure
- Basic nodes have two components:
  - Data stored in that node
  - Link to the next node in the sequence

Data
Link
Data
Link

22

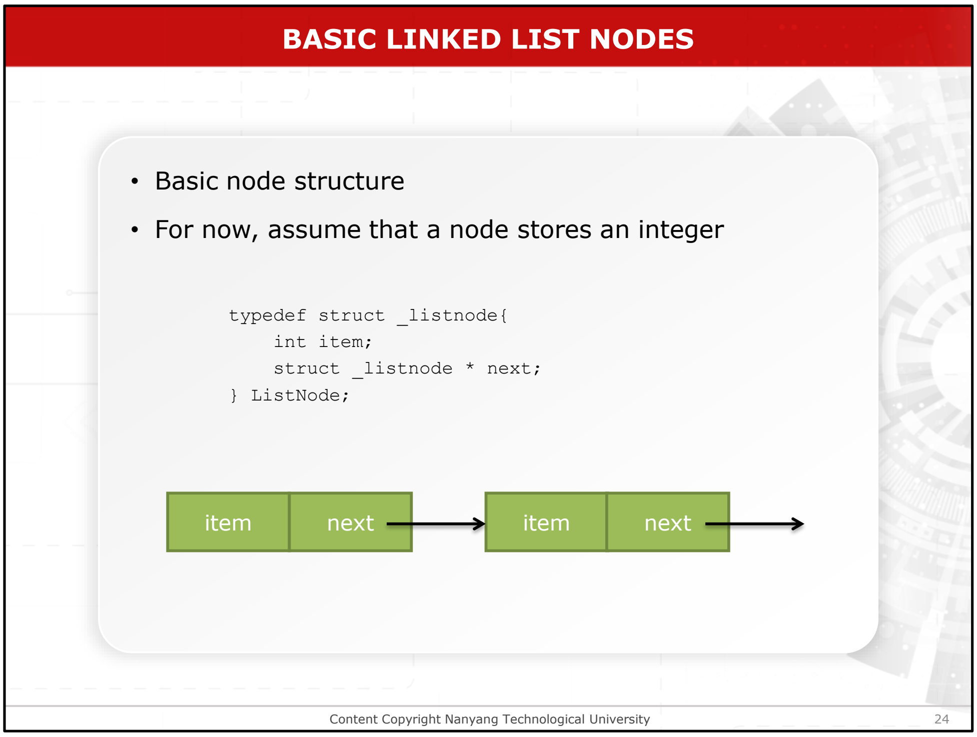
Each of these nodes is a struct which we call ListNode struct. Inside each node, there are two components as the data and the link to the next node. Since this is a ListNode struct, the pointer to the next ListNode is ListNode *. As int * is used to point to an integer, the ListNode * is used to point to the next ListNode.



TODAY

- Data structures as nodes + links
- Storing lists in arrays
- Linked lists
- **Implementing a node**
- Implementing a linked list
- Common mistakes

Let’s learn on implementing a node.



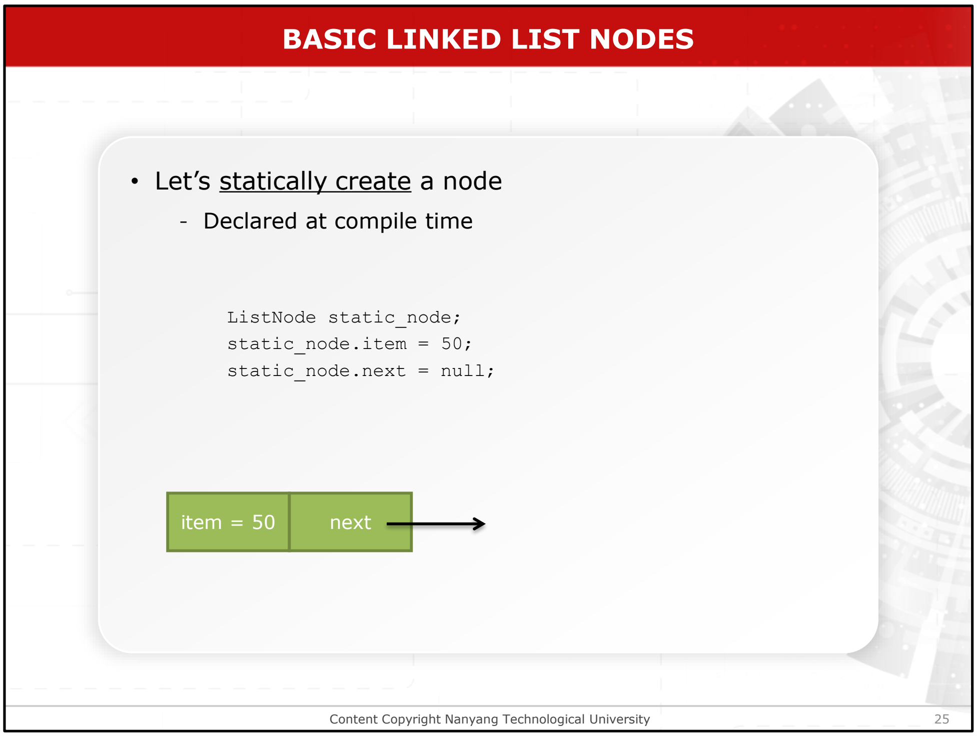
BASIC LINKED LIST NODES

- Basic node structure
- For now, assume that a node stores an integer

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

item
next
item
next

As we discussed in the previous slides, the List Node structure has two components. Since all of these nodes need to be created dynamically, we will use malloc(). But before that, we will discuss on how to create a struct traditionally.



BASIC LINKED LIST NODES

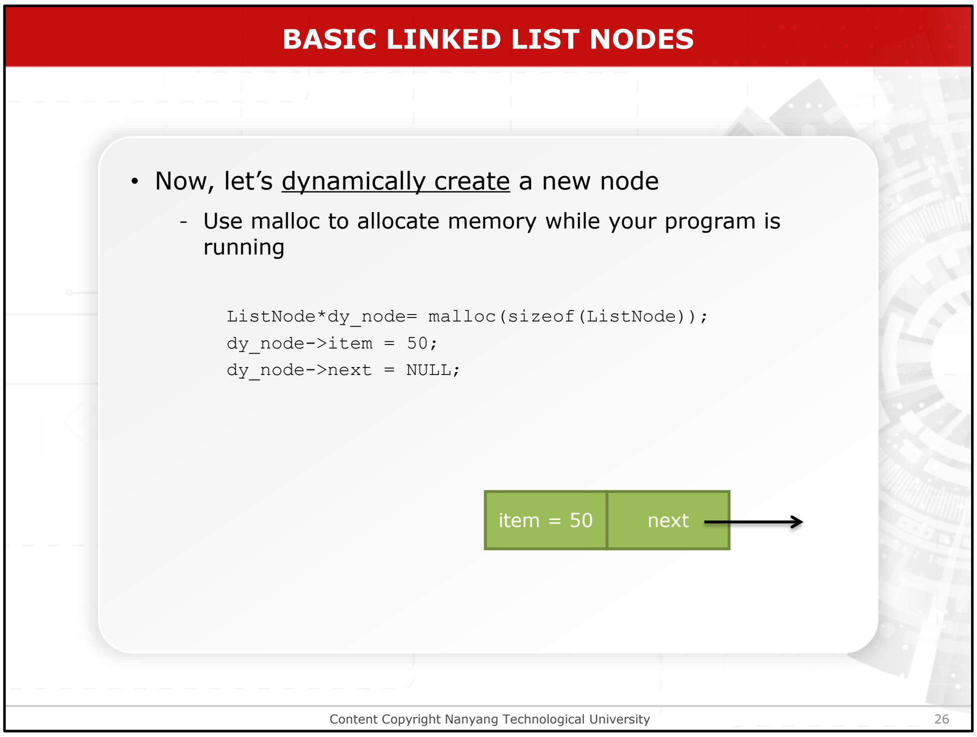
- Let’s statically create a node
  - Declared at compile time

```
ListNode static_node;
static_node.item = 50;
static_node.next = null;
```

item = 50
next

25

In the slide, you can find a simple static declaration. The ListNode static_node creates a single node struct which is used to store number 50. Since it is a single node structure, the pointer to the next node is NULL. This is a basic list node. Now let’s dynamically create a new node using malloc().



BASIC LINKED LIST NODES

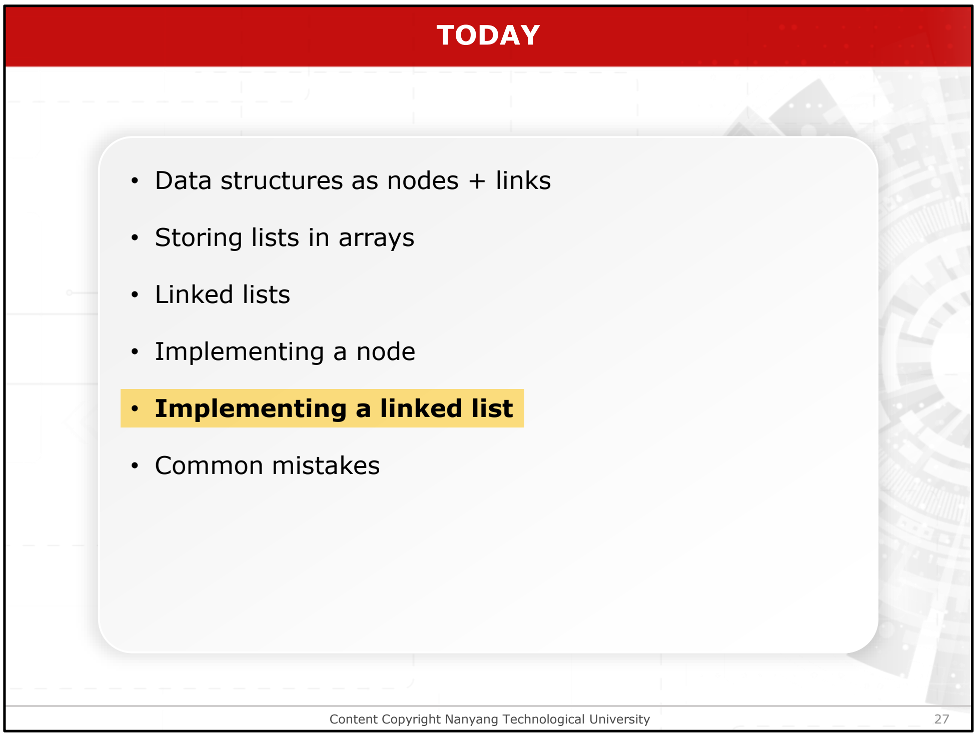
- Now, let’s dynamically create a new node
  - Use malloc to allocate memory while your program is running

```
ListNode*dy_node= malloc(sizeof(ListNode));
dy_node->item = 50;
dy_node->next = NULL;
```

item = 50 | next

Content Copyright Nanyang Technological University 26

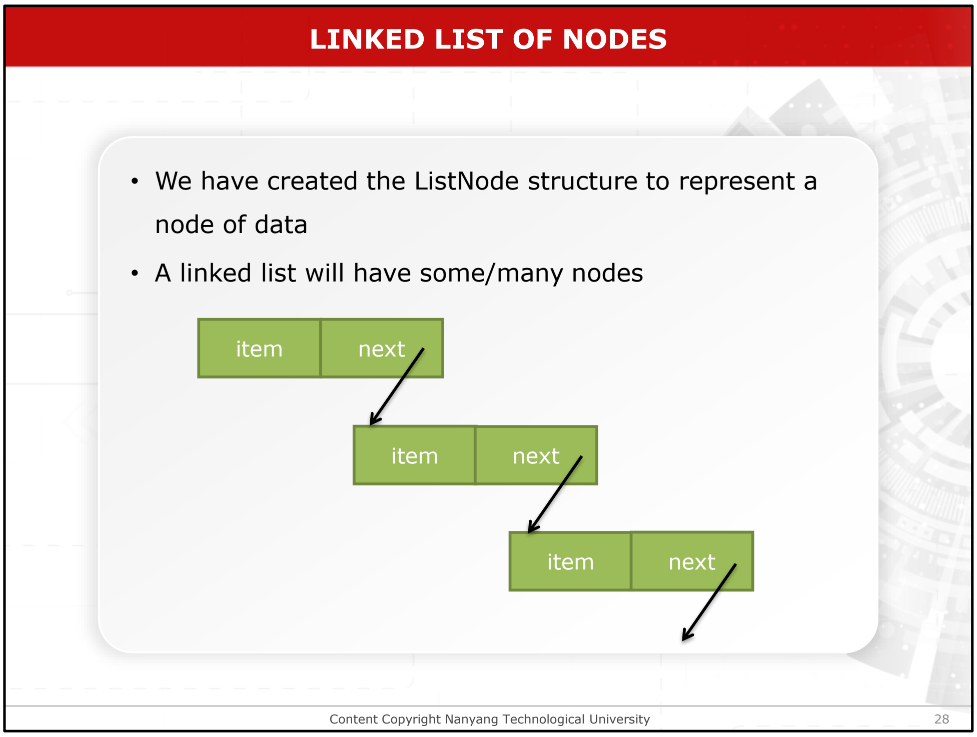
We call malloc to request enough memory to store the structure which is supposed to wrap up two things as the integer variable and the pointer to the next node. Therefore, instead of worrying about the number of bytes we need, we can use sizeof(ListNode) to pass the correct number of bytes automatically to malloc. After malloc() returns the address in the heap, it will be assigned to the ListNode*. This is how you dynamically create a new node.



TODAY

- Data structures as nodes + links
- Storing lists in arrays
- Linked lists
- Implementing a node
- **Implementing a linked list**
- Common mistakes

Now, let’s learn about implementing a linked list.



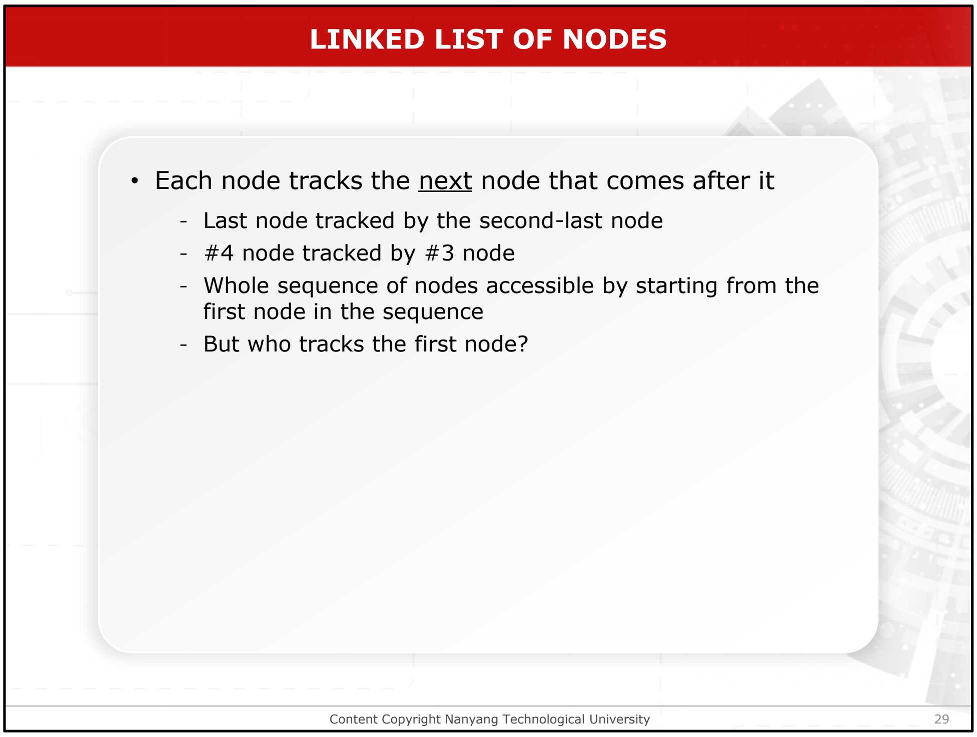
LINKED LIST OF NODES

- We have created the ListNode structure to represent a node of data
- A linked list will have some/many nodes

Content Copyright Nanyang Technological University

28

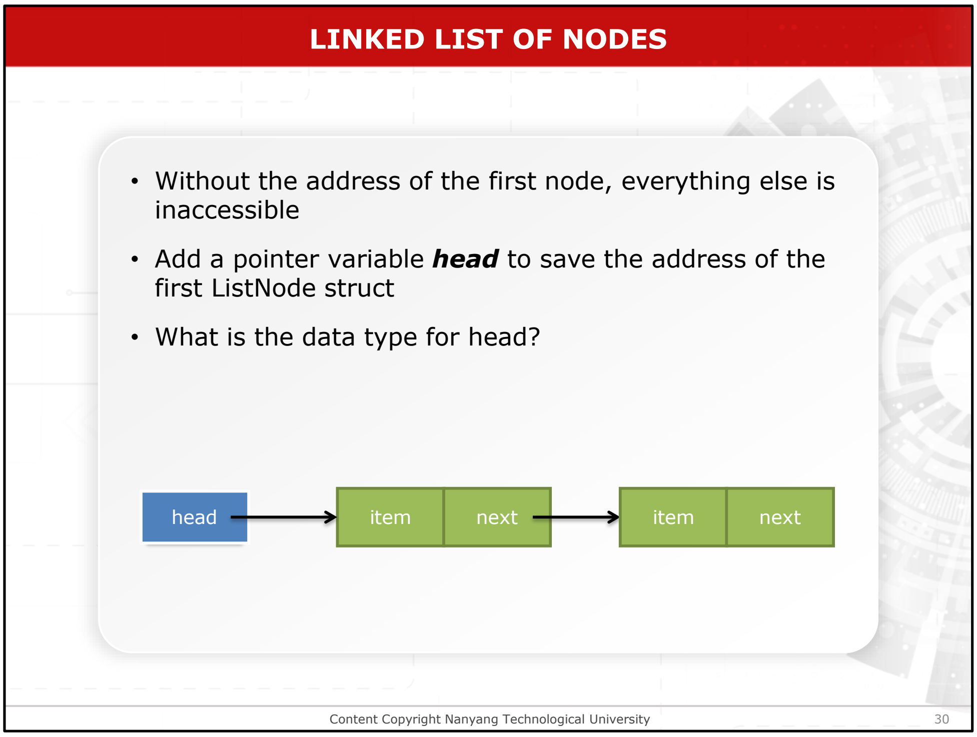
We can connect the ListNodes to form a linked list by adding nodes into different positions in the list.



LINKED LIST OF NODES

- Each node tracks the next node that comes after it
  - Last node tracked by the second-last node
  - #4 node tracked by #3 node
  - Whole sequence of nodes accessible by starting from the first node in the sequence
  - But who tracks the first node?

If you can remember, each node keeps track of the node which comes after it since it is a singly linked list. It does not keep track of the node that comes before; it only keeps track of the node that comes after it. But since we are discussing about sequenced data, if we lose track of one node, the information of the nodes that comes after it also will be lost.

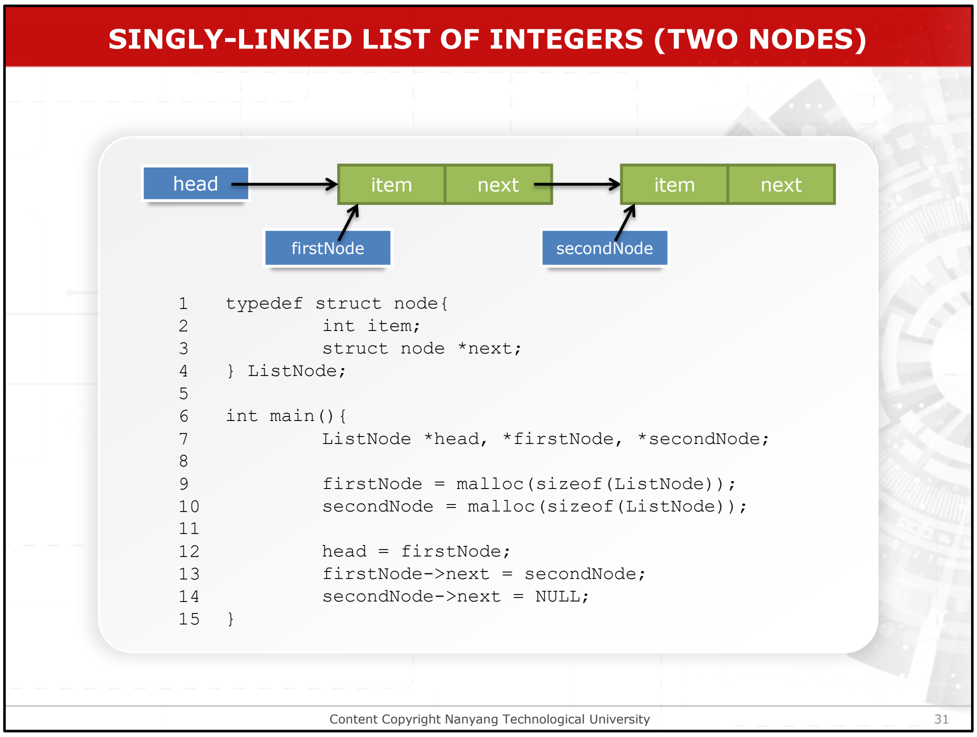


LINKED LIST OF NODES

- Without the address of the first node, everything else is inaccessible
- Add a pointer variable ***head*** to save the address of the first ListNode struct
- What is the data type for head?

head → item | next → item | next

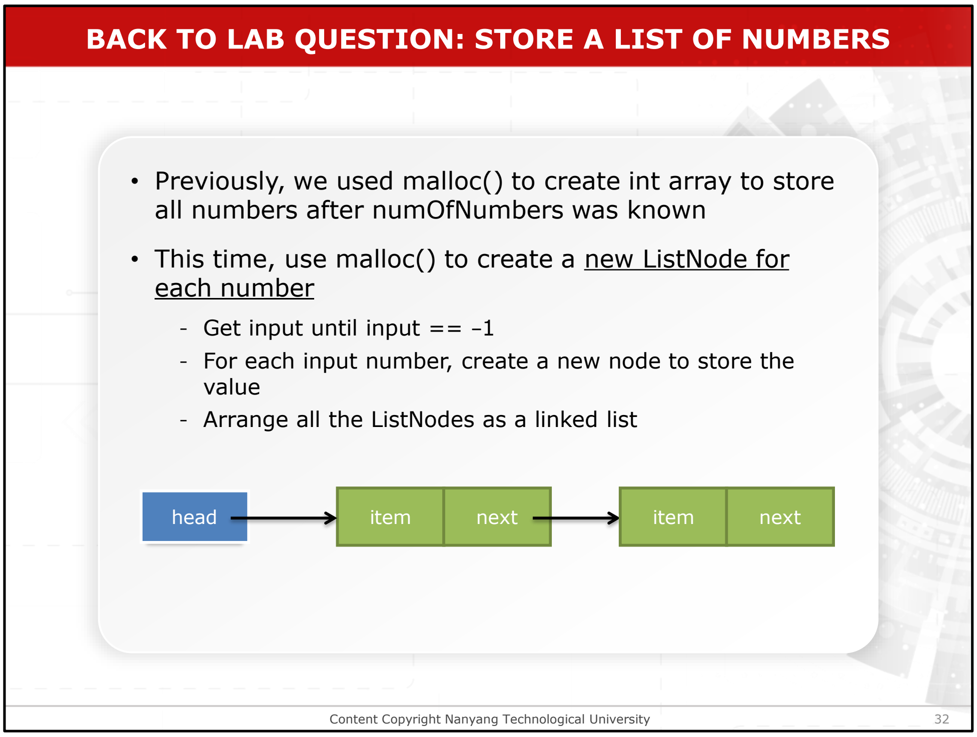
Without the address of the first node, everything else is inaccessible. Therefore, we use the head pointer. Head pointer is a pointer variable. In the slide, the green blocks are structs, and the blue blocks are pointer variable. The head pointer stores the address of the first green block which is first list node struct. Therefore, the head pointer is not a node. It holds the address of the first list node.



SINGLY-LINKED LIST OF INTEGERS (TWO NODES)

```
1    typedef struct node{
2            int item;
3            struct node *next;
4    } ListNode;
5
6    int main(){
7            ListNode *head, *firstNode, *secondNode;
8
9            firstNode = malloc(sizeof(ListNode));
10           secondNode = malloc(sizeof(ListNode));
11
12           head = firstNode;
13           firstNode->next = secondNode;
14           secondNode->next = NULL;
15   }
```

To create a singly linked list of 2 integers, we first create the two nodes individually and then link them. As in line 9 and 10, I first individually create two nodes dynamically using malloc(). Then I need to link everything where the head pointer keeps track of the first node, and the first node keeps track of the second node, in line 12, 13 and 14.

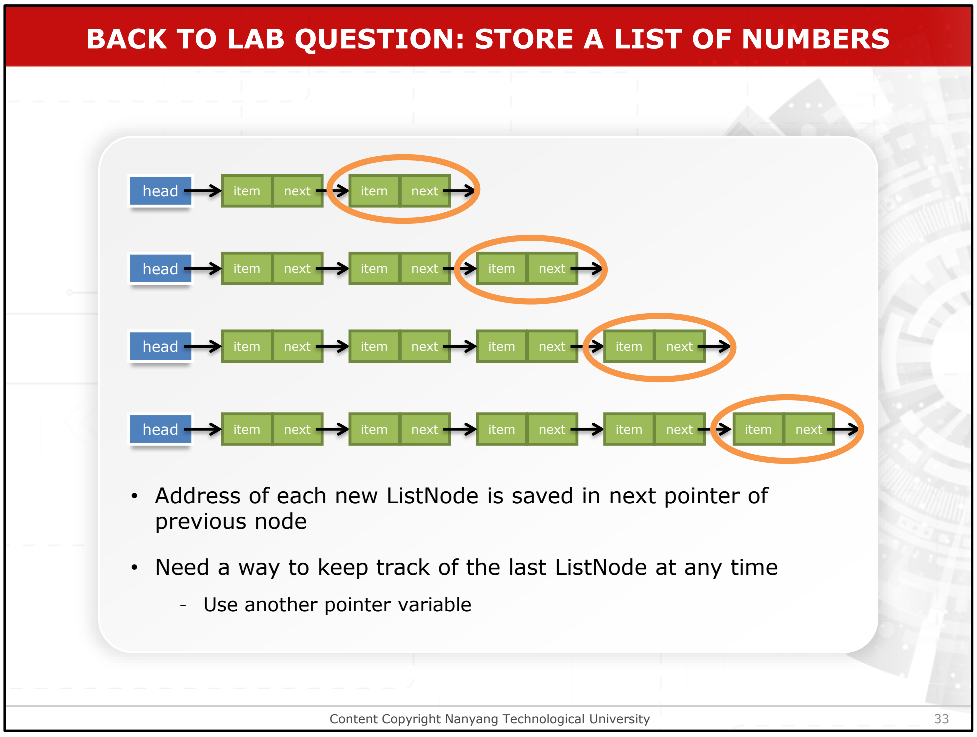


BACK TO LAB QUESTION: STORE A LIST OF NUMBERS

- 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

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 item next
head item next item next item next
head item next item next item next item next
head item next item next item next item next item next

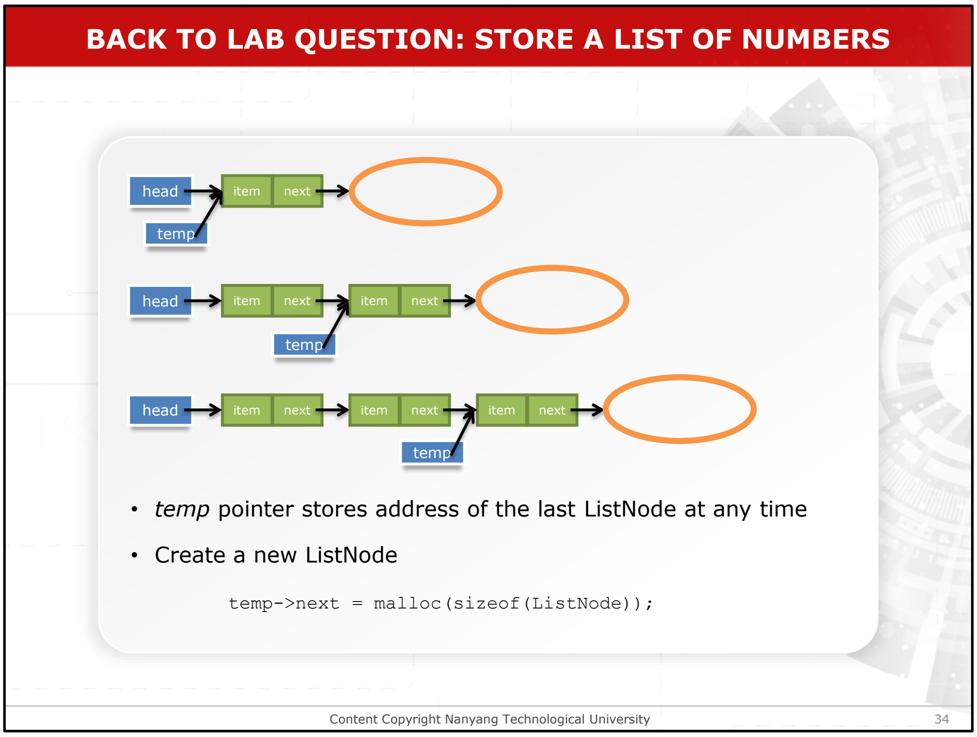
- Address of each new ListNode is saved in next pointer of previous node
- Need a way to keep track of the last ListNode at any time
  - Use another pointer variable

Content Copyright Nanyang Technological University 33

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.

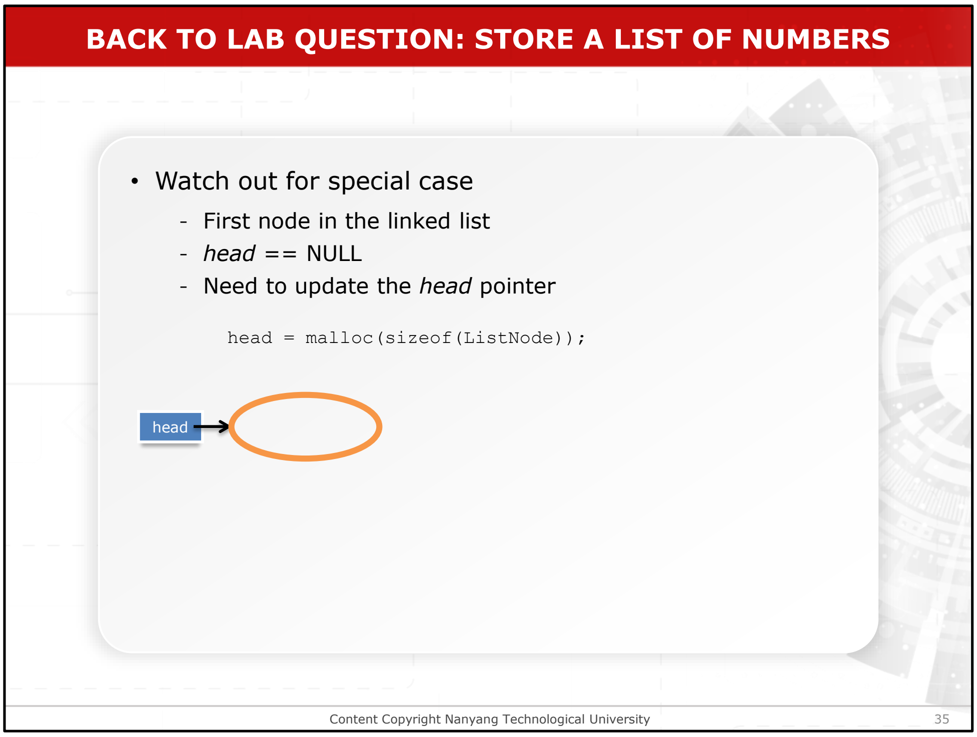


BACK TO LAB QUESTION: STORE A LIST OF NUMBERS

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

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

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

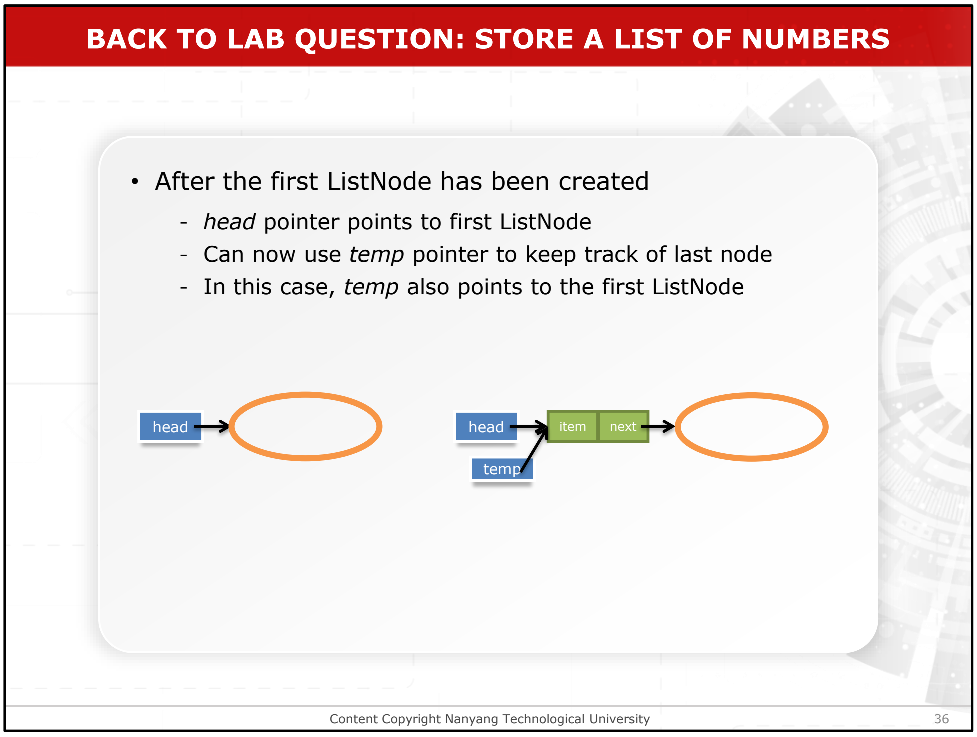
- Watch out for special case
  - First node in the linked list
  - *head* == NULL
  - Need to update the *head* pointer

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

Content Copyright Nanyang Technological University

35

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.

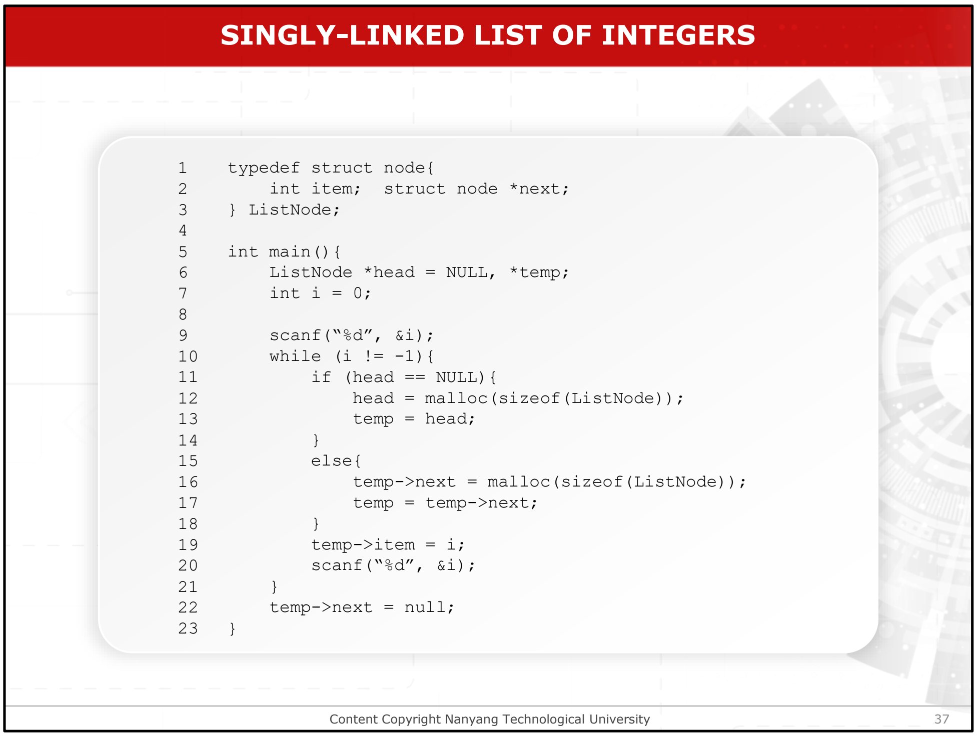


BACK TO LAB QUESTION: STORE A LIST OF NUMBERS

- 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

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

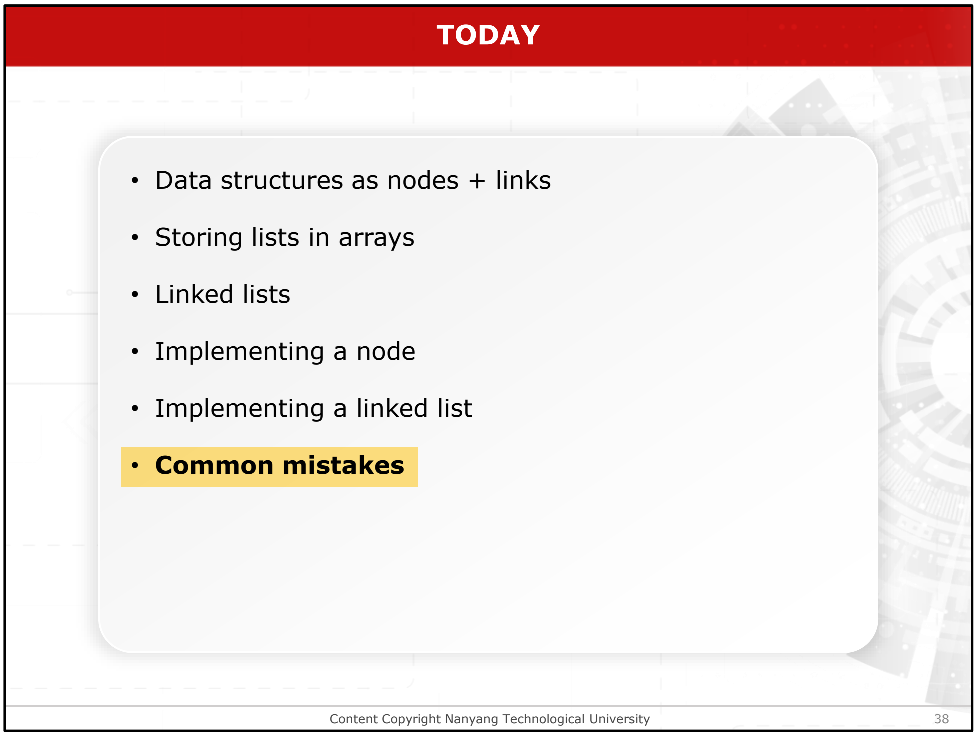
```
typedef struct node{
    int item;  struct node *next;
} ListNode;

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;
}
```

Content Copyright Nanyang Technological University 37

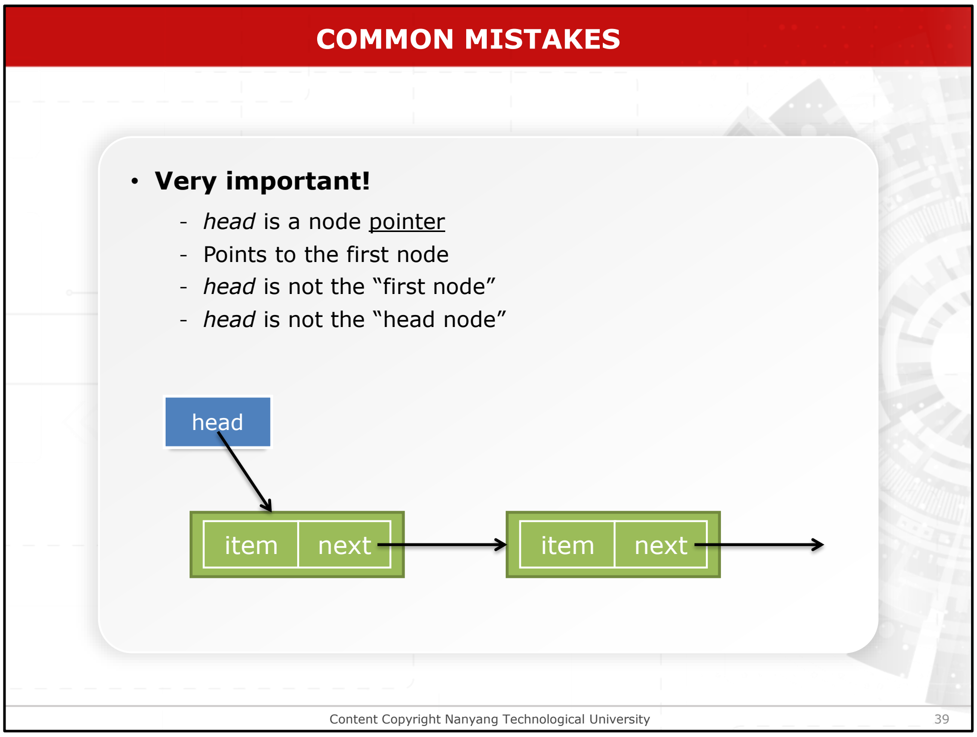
This is the code for the singly-linked list of integers.



TODAY

- Data structures as nodes + links
- Storing lists in arrays
- Linked lists
- Implementing a node
- Implementing a linked list
- **Common mistakes**

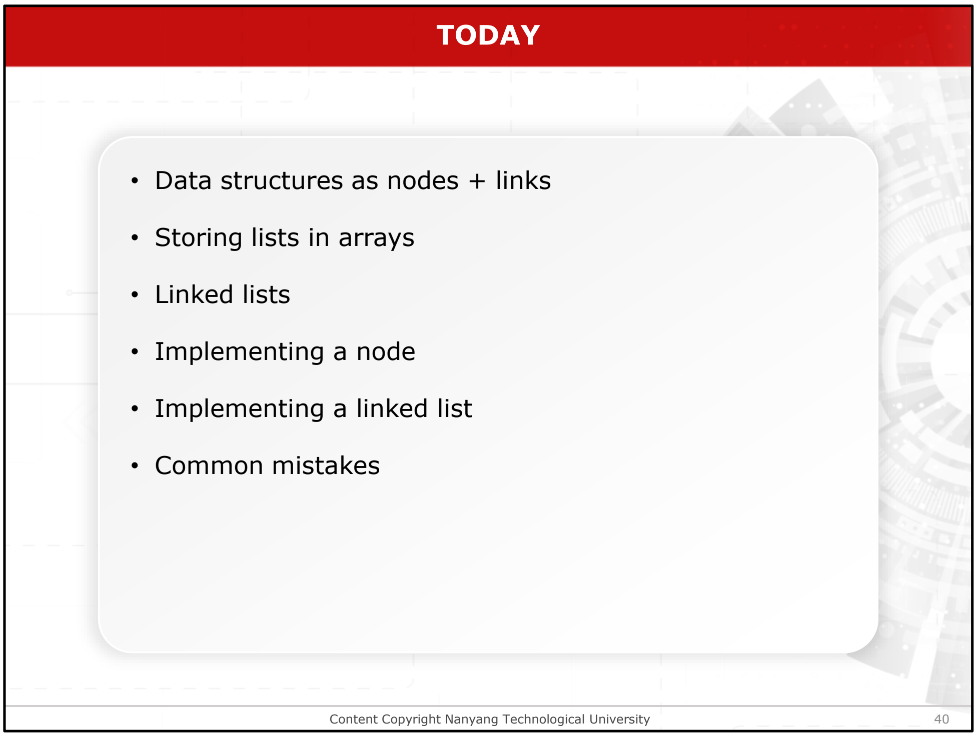
Finally, we will discuss about the common mistakes.



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"

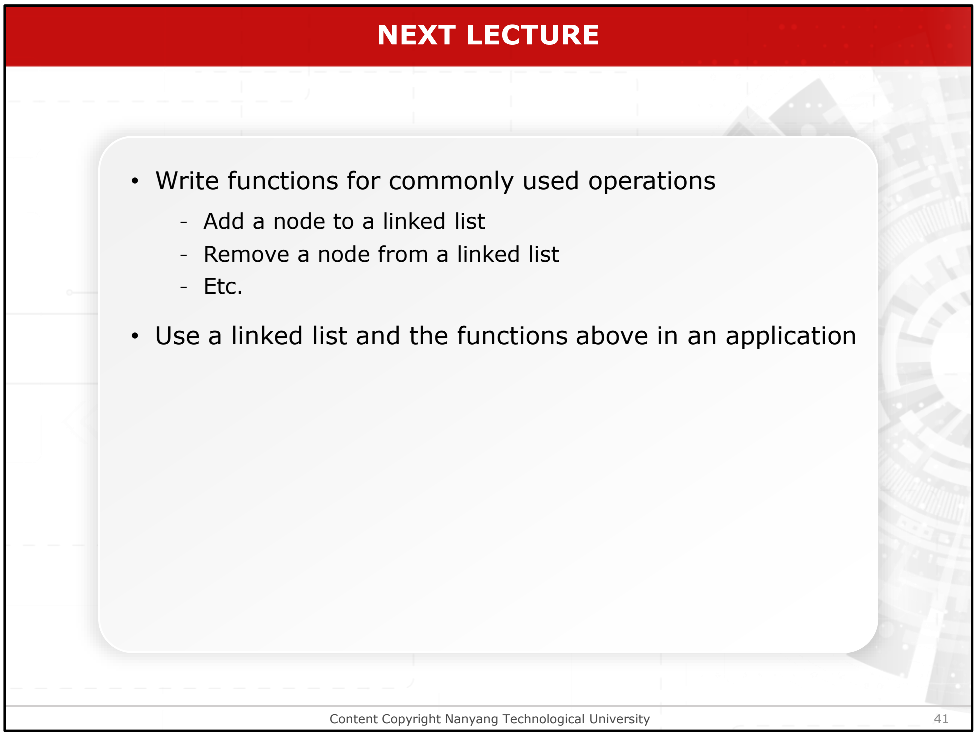
It is essential to realise that the head variable is a pointer which points to the first node of the linked list. It is not a node.



TODAY

- Data structures as nodes + links
- Storing lists in arrays
- Linked lists
- Implementing a node
- Implementing a linked list
- Common mistakes

We have covered all the topics for today’s lecture.



NEXT LECTURE

- Write functions for commonly used operations
  - Add a node to a linked list
  - Remove a node from a linked list
  - Etc.
- Use a linked list and the functions above in an application

41

In the next lecture, we will discuss about the commonly used operations in linked lists such as adding and removing nodes.