



**NANYANG
TECHNOLOGICAL
UNIVERSITY**
SINGAPORE

CE1107/CZ1107: DATA STRUCTURES AND ALGORITHMS

Linked Lists II

College of Engineering
School of Computer Science and Engineering

This lecture is on linked list structures.

TODAY

- `sizeList()` function
- Worked example: Using a linked list
- Linked list C struct
- More complex linked lists
 - Doubly-linked lists
 - Circular linked lists
 - Circular doubly-linked lists
- Array-based list storage
- Summary: Linked lists

Today we will learn about `LinkedList` C struct. This is the core of today's lecture.

LEARNING OBJECTIVES

After this lesson, you should be able to:

- Understand (conceptually) and use (C implementation) a LinkedList struct
- Choose between an array and a linked list for data storage
- Describe (and implement) more complex linked list variants

Content Copyright Nanyang Technological University

3

At the end of today's lecture, you should be able to:

- Understand (conceptually) and use (C implementation) a LinkedList struct
- Choose between an array and a linked list for data storage
- Describe (and implement) more complex linked list variants

TODAY**• sizeList() function**

- Worked example: Using a linked list
- Linked list C struct
- More complex linked lists
 - Doubly-linked lists
 - Circular linked lists
 - Circular doubly-linked lists
- Array-based list storage
- Summary: Linked lists

sizeList function prints the size of a linked list.

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);
```

Content Copyright Nanyang Technological University

5

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

Content Copyright Nanyang Technological University

6

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

```
1 int sizeList(ListNode *head){
2
3 int count = 0;
4
5 if (head == NULL){
6 return 0;
7 }
8
9 while (head != NULL){
10 head = head->next;
11 count++;
12 }
13
14 return count;
15 }
```

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.

**TODAY**

- `sizeList()` function
- **Worked example: Using a linked list**
- Linked list C struct
- More complex linked lists
  - Doubly-linked lists
  - Circular linked lists
  - Circular doubly-linked lists
- Array-based list storage
- Summary: Linked lists

Let's look at how we actually put everything together.



**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

Content Copyright Nanyang Technological University

9

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

```

1 int main() {
2
3 ListNode *head = NULL;
4
5 srand(time(NULL));
6 while (sizeList(head) < 10) {
7 insertNode(&head, 0, rand() % 100);
8 printf("List: ");
9 printList(head);
10 printf("\n");
11 }
12 printf("%d nodes\n", sizeList(head));
13
14 while (sizeList(head) > 0) {
15 removeNode(&head, sizeList(head)-1);
16 printf("List: ");
17 printList(head);
18 printf("\n");
19 }
20 printf("%d nodes\n", sizeList(head));
21
22 return 0;
23 }

```

Content Copyright Nanyang Technological University

10

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

Content Copyright Nanyang Technological University

11

Now, what I want you to think about is how many times does the `sizeList` 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

```

1 int main() {
2
3 ListNode *head = NULL;
4
5 srand(time(NULL));
6 while ((sizeList(head) < 10) {
7 insertNode(&head, 0, rand() % 100);
8 printf("List: ");
9 printList(head);
10 printf("\n");
11 }
12 printf("%d nodes\n", sizeList(head));
13
14 while ((sizeList(head) > 0) {
15 removeNode(&head, sizeList(head)-1);
16 printf("List: ");
17 printList(head);
18 printf("\n");
19 }
20 printf("%d nodes\n", sizeList(head));
21 return 0;
22 }

```

Content Copyright Nanyang Technological University

12

Now, what I want you to think about is how many times does the 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

Content Copyright Nanyang Technological University

13

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

14

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]

```

1 int main(){
2 ListNode *head = NULL;
3 int listsize = 0;
4 srand(time(NULL));
5 while (listsize < 10){
6 insertNode(&head, 0, rand() % 100);
7 listsize++;
8 printf("List: ");
9 printList(head);
10 printf("\n");
11 }
12 printf("%d nodes\n", listsize);
13
14 while (size > 0){
15 removeNode(&head, listsize-1);
16 listsize--;
17 printf("List: ");
18 printList(head);
19 printf("\n");
20 }
21 printf("%d nodes\n", listsize);
22
23 return 0;
24 }

```

Content Copyright Nanyang Technological University

15

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.



**TODAY**

- `sizeList()` function
- Worked example: Using a linked list
- **Linked list C struct**
- More complex linked lists
  - Doubly-linked lists
  - Circular linked lists
  - Circular doubly-linked lists
- Array-based list storage
- Summary: Linked lists

Previously, we had list node structs. Each piece of information in our list gets its own struct. Now we will wrap up the entire list into one struct.



## EXISTING LINKED LIST STRUCTURE

- Consider the “big picture” structure of our linked list
- Head pointer
 
- int listsize
 
- Problems
  - Multiple things to manage
    - We now have to pass listsize variable into functions
  - Functions that modify linked list structure need to be given pointer to head pointer

Content Copyright Nanyang Technological University

17

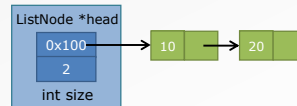
Currently, we have the head pointer and the listsize that are related to our linked list. So, there are multiple things to take care of. Now, if I write a function which separately adds a node and removes a node, I have to rely on that function to increment or decrement the listsize variable correctly. Most importantly, I need to pass the listsize variable into that function to get access to the variable. This makes the parameter list longer. Also, if you have a function that modifies the linked list structure, you need to have a pointer to the head pointer which is complex.

## LinkedList C STRUCT

- Solution

- Define another C struct, LinkedList
- Wrap up (encapsulate) 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

Now, I have wrapped both the head pointer and the listsize variable into a separate structure. This is a linked list structure. If you look at the image, the actual list node structs are not drawn inside the linked list structure because the linked list struct has only two components. The head pointer points to the actual list node structs, but since structs are not inside the linked list structure, you have to be careful.

By looking at the new versions of the functions that we have learned previously, we can see how this linked list structure is important.

## LINKED LIST FUNCTIONS USING LinkedList STRUCT

- Original function prototypes
  - void printList(ListNode \*head);
  - ListNode \* findNode(ListNode \*head);
  - 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);

In the original function prototypes, you pass in the head pointer for printList and pass in the head pointer for findNode. To insert and remove a node, you pass in the pointer to the head pointer. In the new version of the function prototypes, instead of the head pointer, we pass in the linked list struct. We need to pass it in by reference because we want the function to be able to modify the actual linked list object. Now we do not want to worry about different parameter list. We have to pass in the linked list struct regardless of the function.

## 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 }

```

Content Copyright Nanyang Technological University

20

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 insert, print the list, print the size of the list and remove.

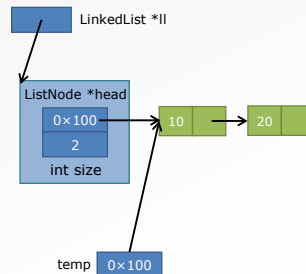
## printList() USING LinkedList STRUCT

- Declare a temp pointer instead of using head (it is no longer a local variable; it is the actual head pointer)

```

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

```



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.

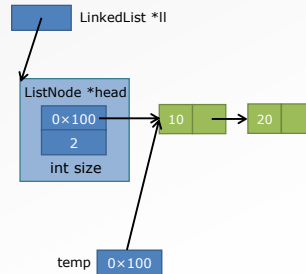
## findNode() USING LinkedList STRUCT

- Again, declare a temp pointer to track the node we are looking at
- Also not much change/improvement in development time here

```

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

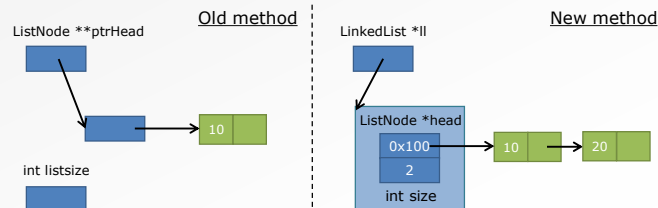
```



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.

## 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



Content Copyright Nanyang Technological University

23

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?

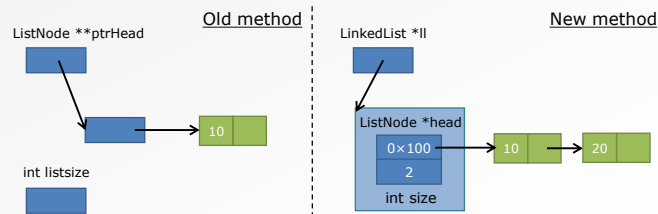
To take care of such a problem, we need to pass in the listsize variable to the insert node function as a parameter.

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



Content Copyright Nanyang Technological University

24

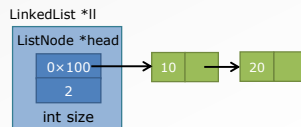
Now you can rewrite the functions with the linked list struct.

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



Content Copyright Nanyang Technological University

25

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.

## NEW sizeList() FUNCTION

- sizeList() just became a trivial function!

```
1 int sizeList(LinkedList *ll){
2 return ll->size;
3 }
```

- This is not a bad thing!
  - No need to recalculate size every time
  - Size only changes when adding/removing nodes
- There is a tradeoff here
  - Sometimes it is better to use some memory to store a value
  - While other times, it is better to use some computation time to calculate it
  - Again, you will encounter this in Algorithms

Content Copyright Nanyang Technological University

26

Now we can rewrite the sizeList function. The size variable is a part of the `LinkedList` struct. The size of the linked list is the value of the size variable. If we want to write a function for it, it returns `ll->size`.

Now, every time I call insert node function, it will automatically increase the value of the size variable and decrease it when I call remove node function. By having an additional variable to store the size of the linked list, we will only use 4 bytes from memory. Is that important?

The expected size of a linked list is a lot more than 4 bytes, and therefore, 4bytes is trivial. But, if we are running a system with only 16 bytes of memory, 4 bytes can make a big difference. And, in such a situation, traversing the linked list to calculate the size of the list every time is better.

So, you have to know that sometimes it makes more sense to use memory to pre-store a value and sometimes it is better to recalculate it whenever it is needed.

**TODAY**

- `sizeList()` function
- Worked example: Using a linked list
- Linked list C struct
- **More complex linked lists**
  - Doubly-linked lists
  - Circular linked lists
  - Circular doubly-linked lists
- Array-based list storage
- Summary: Linked lists

Let's spend about five minutes talking about more complex linked list before we end off with a summary of what's good and bad about linked list and when you will use them.

## MORE COMPLEX LINKED LISTS

- So far, singly-linked list
  - Each ListNode is linked to at most one other ListNode
  - Traversal of the list is one-way only
    - Cannot go backwards
- Idea: Allow two-way traversal of a list
  - Maybe we want to start from a given node and search EITHER backwards OR forward
  - Each node now has to connect to the previous node as well

Content Copyright Nanyang Technological University

28

So far we've talked about singly linked lists. In singly linked lists, we store one integer value inside each node, and we have only one next pointer for each node. This means, when traversing a singly linked list, we can only go one way, starting from the first node using the head pointer and go all the way down of the linked list.

We cannot start traversing the linked list from the last node and traverse to the first node of the linked list.

If we want to traverse backwards as well, we need to connect each of the linked list nodes to the node after and the node before.

## DOUBLY LINKED LIST

- Modify the ListNode struct

```
typedef struct _dblListNode{
 int item;
 struct _dblListNode *prev;
 struct _dblListNode *next;
} DblListNode;
```

- Note that first node has prev == NULL
- Inserting a node
  - Have to set the prev and next pointers accordingly for all nodes involved
  - Included in practice questions for Lab-Tutorial 9



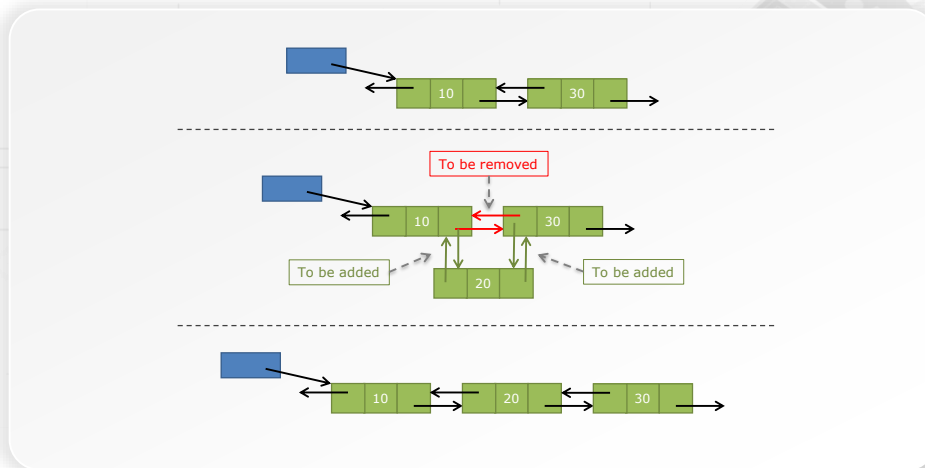
Content Copyright Nanyang Technological University

29

Now we have a new version of list nodes called doubly linked list nodes. These nodes link with the node after as well as the node before. Therefore, the struct type also has to be changed to double list node.

So, in addition to the item and the next pointer, we have the previous pointer. This is powerful because when we insert a node to a linked list now, we would need to check whether four pointers are set correctly while earlier it was just two.

In a doubly linked list node, if the next pointer is NULL, it indicates the last node, and if the previous pointer is NULL, it indicates the first node.

**INSERTING A NODE INTO A DOUBLY LINKED LIST**

Content Copyright Nanyang Technological University

30

In the doubly linked list, when inserting a node, you have to make sure that you have set and remove pointers correctly.

## DOUBLY LINKED LIST

- Traversing a doubly linked list in forward direction

```
temp = temp->next;
```

- Traversing a doubly linked list in backward direction

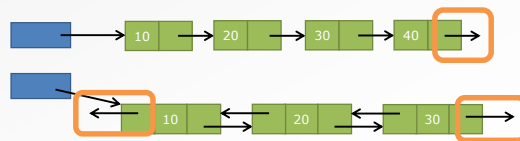
```
temp = temp->prev;
```



Traversing a doubly linked list is very simple. `temp=temp->next` gets the address of the next pointer and stores in the temporary pointer. If we traverse backwards, we should copy the address of the previous node to the temporary pointer.

## MORE COMPLEX LINKED LISTS (PART II)

- So far, linked list has a fixed end (or ends)
- No way to loop around
- Might be useful to allow looping traversal
  - Circular linked lists
- No extra variables needed in the ListNode struct
  - Just have to add connections



Content Copyright Nanyang Technological University

32

Now, let's make things a bit more complicated. In the doubly-linked list we have learned, it has fixed ends where both next pointer of the last node and previous pointer of the first node are set to NULL. What if we want to allow looping?

What if we want the linked list to become a circular list? Do we need to have additional variables? NO! The doubly linked list has all the variable to turn it into a circular doubly linked list. Even singly linked list has all the variables to turn it into a circular singly-linked list.

To turn a singly-linked list to a circular singly-linked list, we need to take the next pointer of the last node and link it back to the first node.

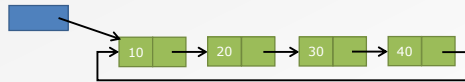
To turn a doubly-linked list to a circular doubly-linked list, we need to take the next pointer of the last node and link it to the first node, and take the previous pointer of the first node and link it to the last node.



## CIRCULAR LINKED LISTS

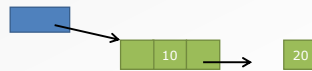
- Circular singly-linked lists

- Last node has next pointer pointing to first node



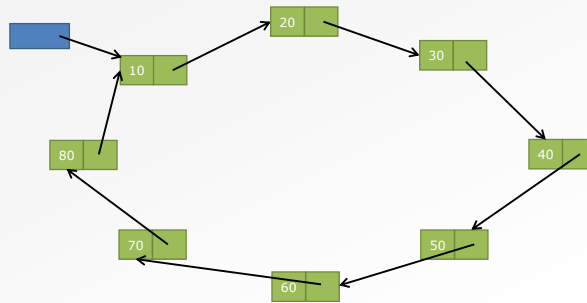
- Circular doubly-linked lists

- Last node has next pointer pointing to first node
- First node has prev pointer pointing to last node



## CIRCULAR LINKED LISTS

- Effectively we will have this (singly-linked version)

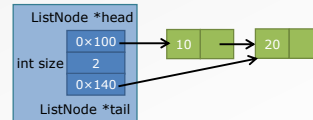


This is how the circular singly-linked list looks once we expand the image.

## LinkedList C STRUCT: ONE MORE THING

- Alternative version of our LinkedList struct

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



- Tail pointer
  - Always points to the last node of the linked list
- Why is this useful?

Since we can add anything to the linked list struct, I have added a tail pointer to the struct which always points to the last node of the linked list.

If you can recall what you have learned so far, you will remember that we started with a small program that adds a new item to the back of the existing list each time. This is inefficient as we have to traverse the linked list each time we add an item. Now that we have a tail pointer, we can straightaway jump to the last node and attach the new node to it.

You will see that this becomes useful when we make use of our linked list struct to create stack and queue data structures next week.

**TODAY**

- `sizeList()` function
- Worked example: Using a linked list
- Linked list C struct
- More complex linked lists
  - Doubly-linked lists
  - Circular linked lists
  - Circular doubly-linked lists
- **Array-based list storage**
- Summary: Linked lists

Now let's take a look at array-based list storage.

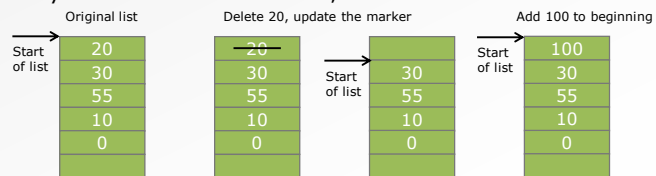
## ARRAY-BASED LISTS

- Back to arrays as list storage
- Try to implement “smarter” array-based list
- Avoid some of the problems we saw earlier with using arrays to store lists
  - Key is to minimize shifting operations

This is an extra knowledge for you. You can go through this to learn to make use of arrays and still get most of the benefits of a linked list.

## ARRAY-BASED LISTS

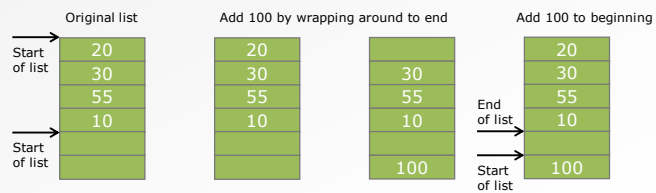
- Delete an item from the beginning of a list
  - Key idea: Leave the empty space, do not shift everything down
  - In future, empty space gets used if we add to the beginning
  - Use a marker (or index number) to store location of first actual item
- Try: Delete 20 from index 0, then add 100 to index 0



Delete an item from the beginning of a list

## ARRAY-BASED LISTS

- Unfortunately, this doesn't help once you run out of space at the beginning
- Idea: Wrap around to the other end; circular array



When you run out of space at the beginning of the list, what should you do?

**TODAY**

- `sizeList()` function
- Worked example: Using a linked list
- Linked list C struct
- More complex linked lists
  - Doubly-linked lists
  - Circular linked lists
  - Circular doubly-linked lists
- Array-based list storage

- **Summary: Linked lists**

Summary of linked lists.



## ARRAYS VS. LINKED LISTS

- Array-based lists allow random access
  - No need to traverse list until you reach the node index that you want
  - Much more efficient lookup compared to linked lists
- Previous slides show how clever tricks can be used to overcome some shortcomings of array-based list storage
- Important to know what arrays and linked lists are good and bad for

Content Copyright Nanyang Technological University

41

We learned about linked lists because arrays cannot efficiently add something to the beginning of the list, add something to the middle or remove something from the middle of the list.

With a linked list, every item is a separate node which you can move anywhere you want. But you cannot do random access to linked lists. With array-based lists, you can perform random access.

Random access is just the ability to get directly to, say, number three or element number five or element number seven, based on the index value.

## ARRAYS VS. LINKED LISTS

- **Arrays**

- Efficient random access
- Difficult to expand, rearrange
- When inserting/removing items in the middle or at the beginning, computation time scales with size of list
- Generally a better choice when data is immutable

- **Linked Lists**

- "Random access" can be implemented, but more inefficient than arrays
- Excellent for dynamic lists
- Easy to expand, shrink, rearrange
- Insert/remove operations only require fixed number of operations regardless of list size

- Know when to choose an array or a linked list

Content Copyright Nanyang Technological University

42

Linked lists are great for any list that is dynamic. Linked lists can handle size changes, arrangement changes, etc. easier than arrays. Linked lists take a fixed number of operations to perform any kind of an arrangement or movement. But, with an array, to insert an item to the beginning, you need to move everything down. And if the size is increased, the steps also get increased.

**TODAY**

- `sizeList()` function
- Worked example: Using a linked list
- Linked list C struct
- More complex linked lists
  - Doubly-linked lists
  - Circular linked lists
  - Circular doubly-linked lists
- Array-based list storage
- Summary: Linked lists

We have covered the first proper linear dynamic data structure. After this, we are going to use the foundation of the linked list to move on the stacks and queues.