NANYANG TECHNOLOGICAL UNIVERSITY

CE1007/CZ1007 DATA STRUCTURES

Lecture 04: **Linked Lists II**

Dr. Owen Noel Newton Fernando

College of Engineering

School of Computer Science and Engineering

OUTLINE

- 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
- Summary: Linked lists

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

PREVIOUSLY...

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

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

OUTLINE

- **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
- Summary: Linked lists

sizeList() FUNCTION

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

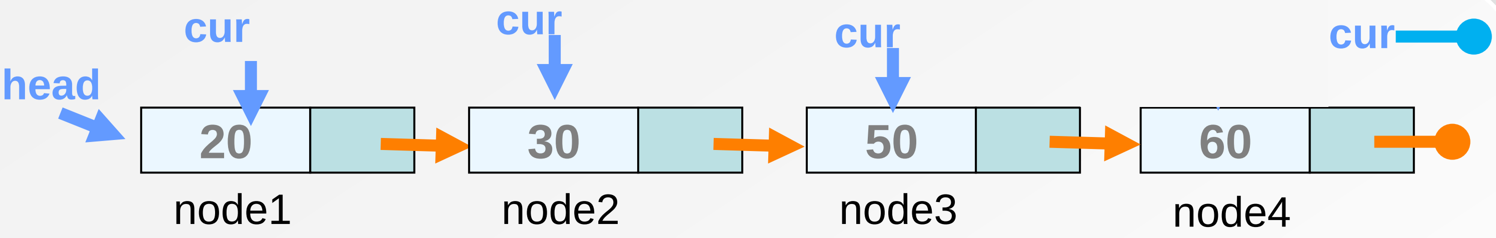
    int sizeList(ListNode *head);
- Use the head pointer to track first node
- Keep following the next pointer until next == NULL
- Increment counter
- Return the counter

sizeList()

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

```
int sizeList(ListNode *head){

    int count = 0;

    if (head == NULL){
        return 0;
    }

    while (head != NULL){
        count++;
        head = head->next;
    }

    return count;
}
```

sizeList()



```
 int sizeList(ListNode *head){
    ListNode *cur;
    cur=head;

    int count = 0;
    if (cur == NULL){
        return 0;
    }
    while (cur != NULL){
        count++;
        cur = cur ->next;
    }

    return count;
}
```

Counter 4

return 4;

OUTLINE

- 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
- Summary: Linked lists

WORKED EXAMPLE: LINKED LIST APPLICATION

- We consider the following problem:

> Generate a list of M (10) numbers by inserting random numbers (0--99) into the front of the list until it has M (10) nodes, then remove all nodes.

- Use sizeList(), insertNode(), printList() and removeNode() functions

LINKED LIST APPLICATION (1)

- Use the sizeList(), insertNode() and printList() functions
- Generate a list of 10 numbers by inserting random numbers (0-99) into the front of the list until it has 10 nodes.

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

```
int main(){

    ListNode *head = NULL;

    srand(time(NULL));
    while (sizeList(head) < 10){
        insertNode(&head, 0, rand() % 100);
        printf("List: ");
        printList(head);
        printf("\n");
    }
    printf("%d nodes\n", sizeList(head));

    while (sizeList(head) > 0){
        removeNode(&head, sizeList(head)-1);
        printf("List: ");
        printList(head);
        printf("\n");
    }
    printf("%d nodes\n", sizeList(head));

    return 0;
}
```

The srand() function sets its argument as the seed for a new sequence of pseudo-random integers to be returned by rand().

LINKED LIST APPLICATION (1)

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

```
int main(){

    ListNode *head = NULL;

    srand(time(NULL));
    while (sizeList(head) < 10){
        insertNode(&head, 0, rand() % 100);
        printf("List: ");
        printList(head);
        printf("\n");
    }
    printf("%d nodes\n", sizeList(head));

    while (sizeList(head) > 0){
        removeNode(&head, sizeList(head)-1);
        printf("List: ");
        printList(head);
        printf("\n");
    }
    printf("%d nodes\n", sizeList(head));

    return 0;
}
```

LINKED LIST APPLICATION (1)

- Very inefficient!
- How often does 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

LINKED LIST APPLICATION (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

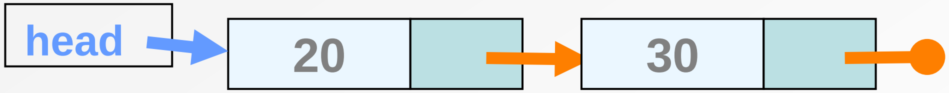
```
int main(){
    ListNode *head = NULL;
    int listsize = 0;
    srand(time(NULL));
    while (listsize < 10){
        insertNode(&head, 0, rand() % 100);
        listsize++;
        printf("List: ");
        printList(head);
        printf("\n");
    }
    printf("%d nodes\n", listsize);

    while (size > 0){
        removeNode(&head, listsize-1);
        listsize--;
        printf("List: ");
        printList(head);
        printf("\n");
    }
    printf("%d nodes\n", listsize);

    return 0;
}
```

OUTLINE

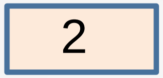
- sizeList() function
- Worked example: Using a linked list
- **Linked list C struct**
- More complex linked lists
  - Doubly-linked lists
  - Circular linked lists
  - Circular doubly-linked lists
- Summary: Linked lists

EXISTING LINKED LIST STRUCTURE

- Consider the “big picture” structure of our linked list
- Head pointer



- int listsize

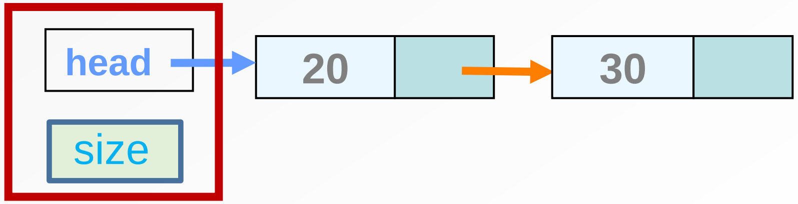


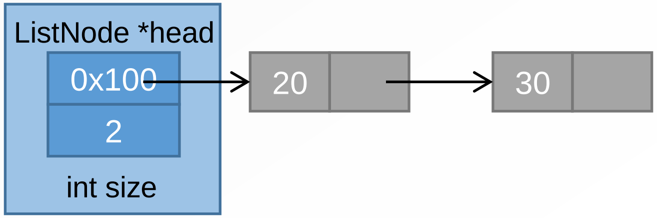
- Problems:
  - Multiple things to manage
    - Now have to pass listsize variable into functions?
  - Functions that modify linked list structure need to be given pointer to head pointer

LinkedList C STRUCT

- Solution:
  - Define another C struct, LinkedList
  - Wrap up all elements that are required to implement the Linked List data structure

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





- Why is this useful?

  Consider the rewritten Linked List functions

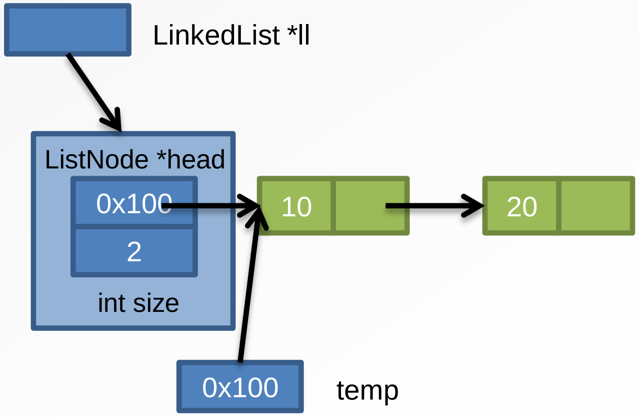
LINKED LIST FUNCTIONS USING LinkedList STRUCT

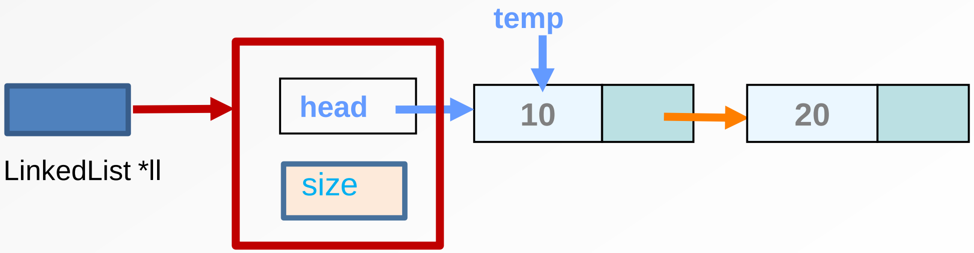
- Original function prototypes:
  - void printList(ListNode *head);
  - ListNode *findNode(ListNode *head);
  - int insertNode(ListNode **ptrHead, int index, int value);
  - int removeNode(ListNode **ptrHead, int index);
- New function prototypes:
  - **void printList(LinkedList *ll);**
  - **ListNode *findNode(LinkedList *ll, int index);**
  - **int insertNode(LinkedList *ll, int index, int value);**
  - **int removeNode(LinkedList *ll, int index);**

printList() USING LINKEDLIST STRUCT

- Have to declare a temp pointer instead of using head (it’s no longer a local variable, it’s the actual head pointer)

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

    if (temp == NULL)
        return;

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

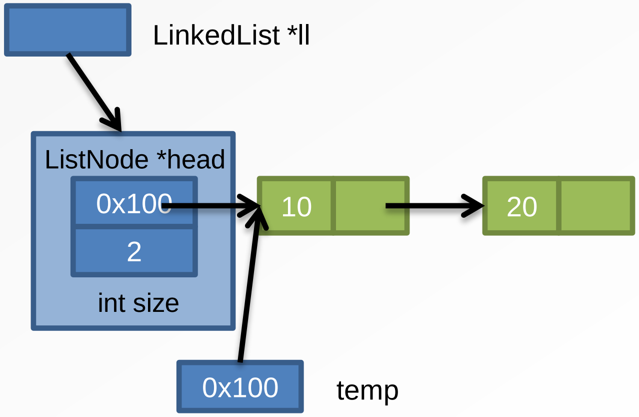


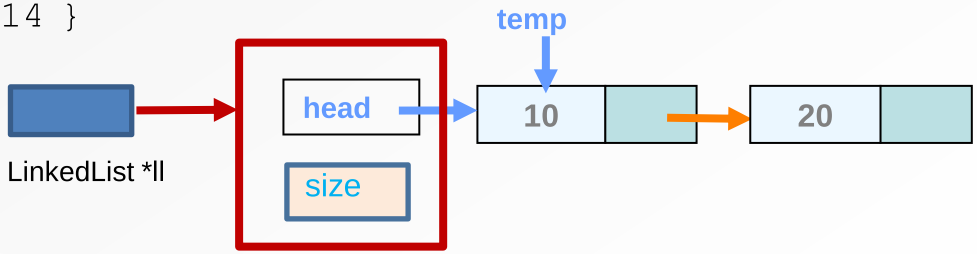


findNode() USING LinkedList STRUCT

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

```
ListNode * findNode(LinkedList *ll, int index){

    ListNode *temp = ll->head;
    if (temp == NULL || index < 0)
        return NULL;

    while (index > 0){
        temp = temp->next;
        if (temp == NULL)
            return NULL;
        index--;
    }
    return temp;
}
```

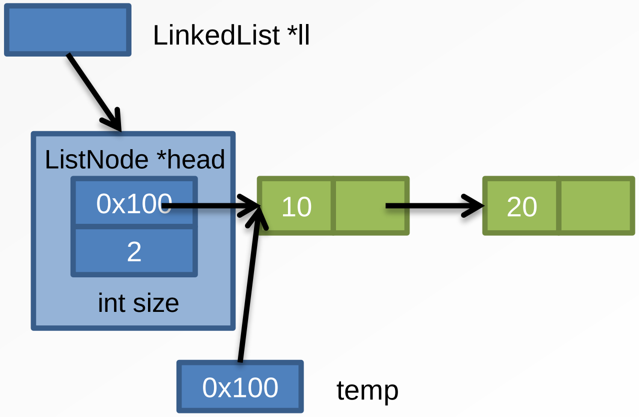


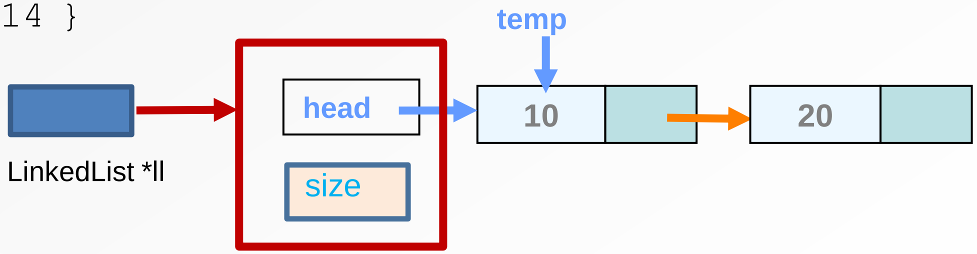


findNode() USING LinkedList STRUCT

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

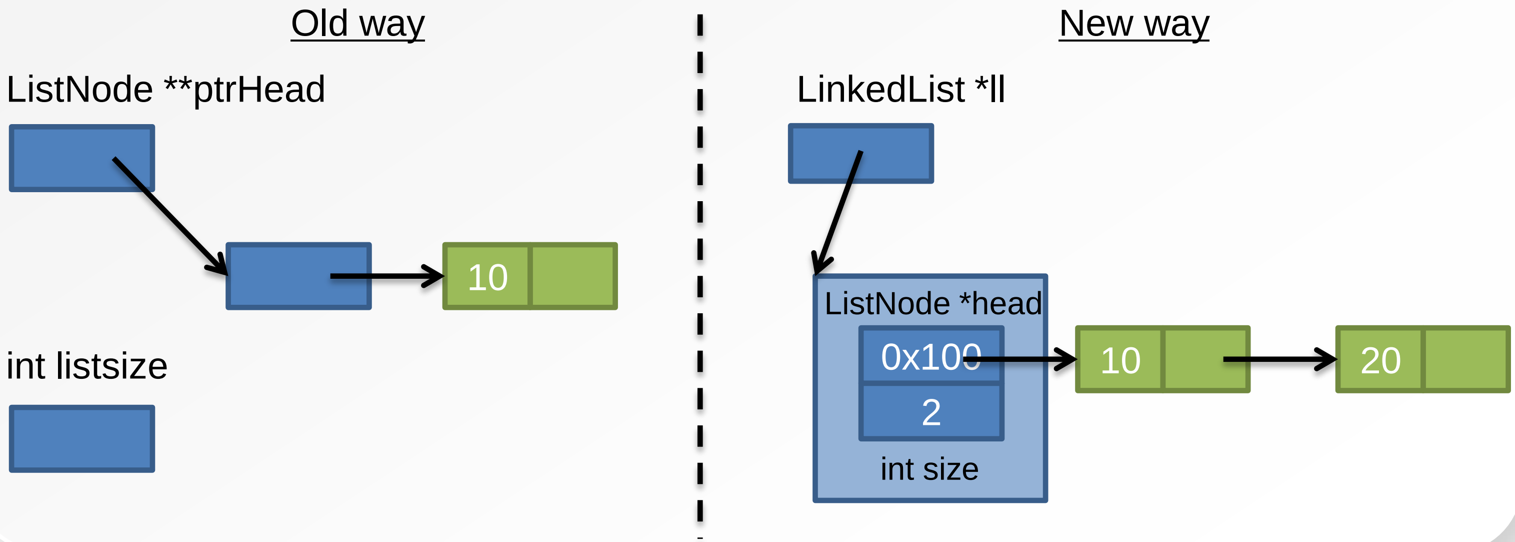
```
ListNode * findNode(LinkedList *ll, int index){
    ListNode *temp
    temp = ll->head;
    if (temp == NULL || index < 0)
        return NULL;

    while (index > 0){
        temp = temp->next;
        if (temp == NULL)
            return NULL;
        index--;
    }
    return temp;
}
```





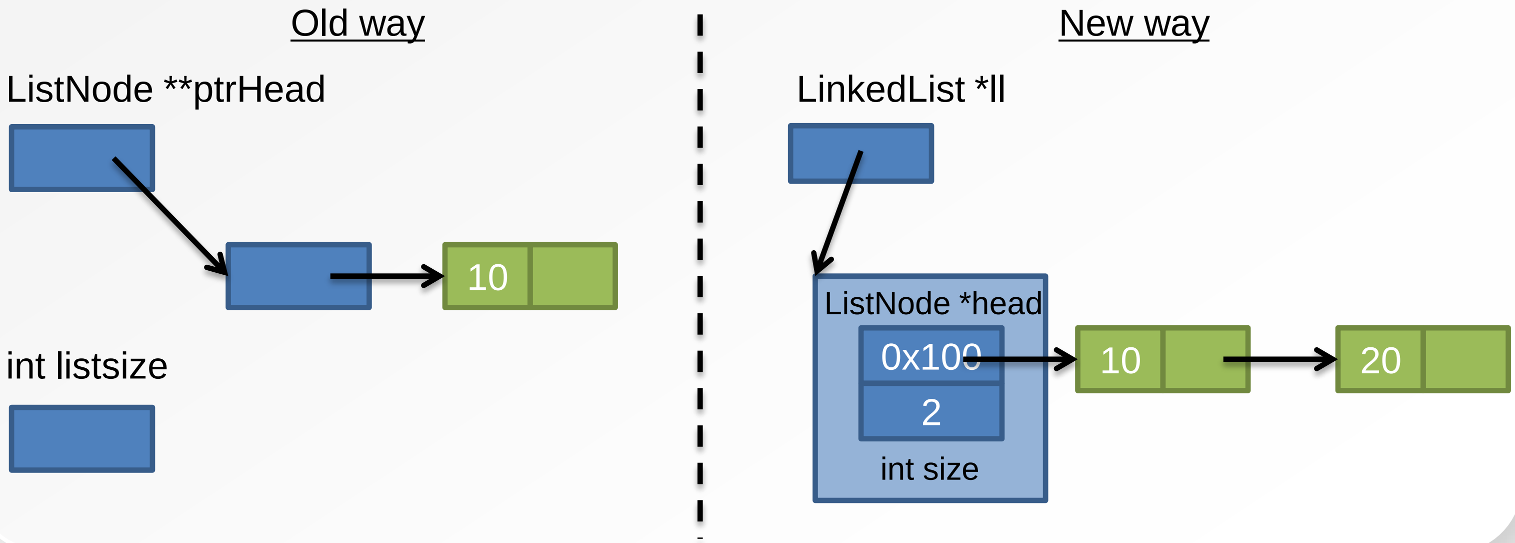
insertNode() USING LinkedList STRUCT

- Pass in pointer to LinkedList struct
- Function has full access to read and write address in head pointer
- Function can also update the # of nodes in the size variable, no need to pass in &listsize
- No need to think about double dereferencing



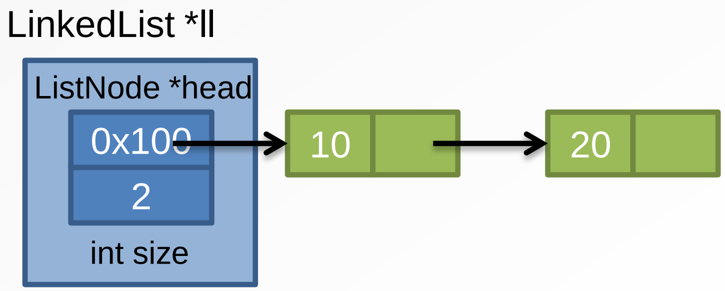
insertNode() USING LinkedList STRUCT

- 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



LinkedList C 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



NEW sizeList() FUNCTION

- sizeList() just became a trivial function!

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

- This is not a bad thing!
  - No need to recalculate size every time
  - Size only changes when adding/removing nodes

LinkedList C STRUCT: TAIL POINTER

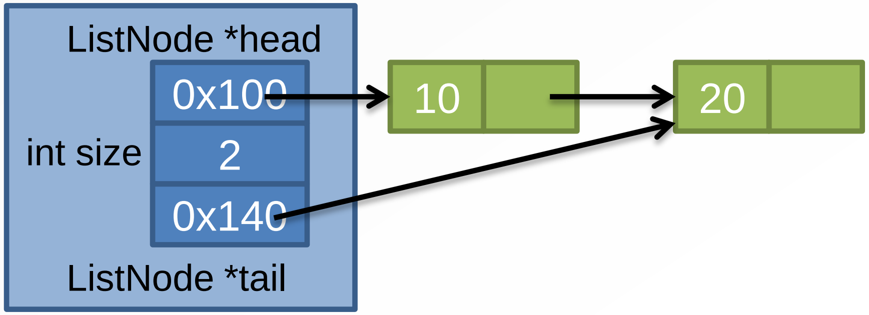
Depends on the problem, you can define new linked list structure

- If you want to insert many nodes to the back of the list

  findNode(…, ptr_ll->size) traverses the whole linked list for every insertNode().

- Tail pointer: always points to the last node of the linked list
- New version of LinkedList struct

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



OUTLINE

- 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**
- Summary: Linked lists

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
    - Can’t go backwards
    - What if we want to start from a given node and search EITHER backwards OR forwards
- Idea: allow two-way traversal of a list
  - Each node now has to connect to the <u>previous</u> node as well

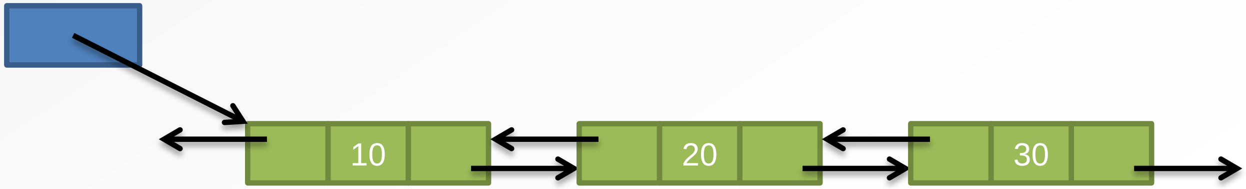
DOUBLY LINKED LIST

- Modify the ListNode struct

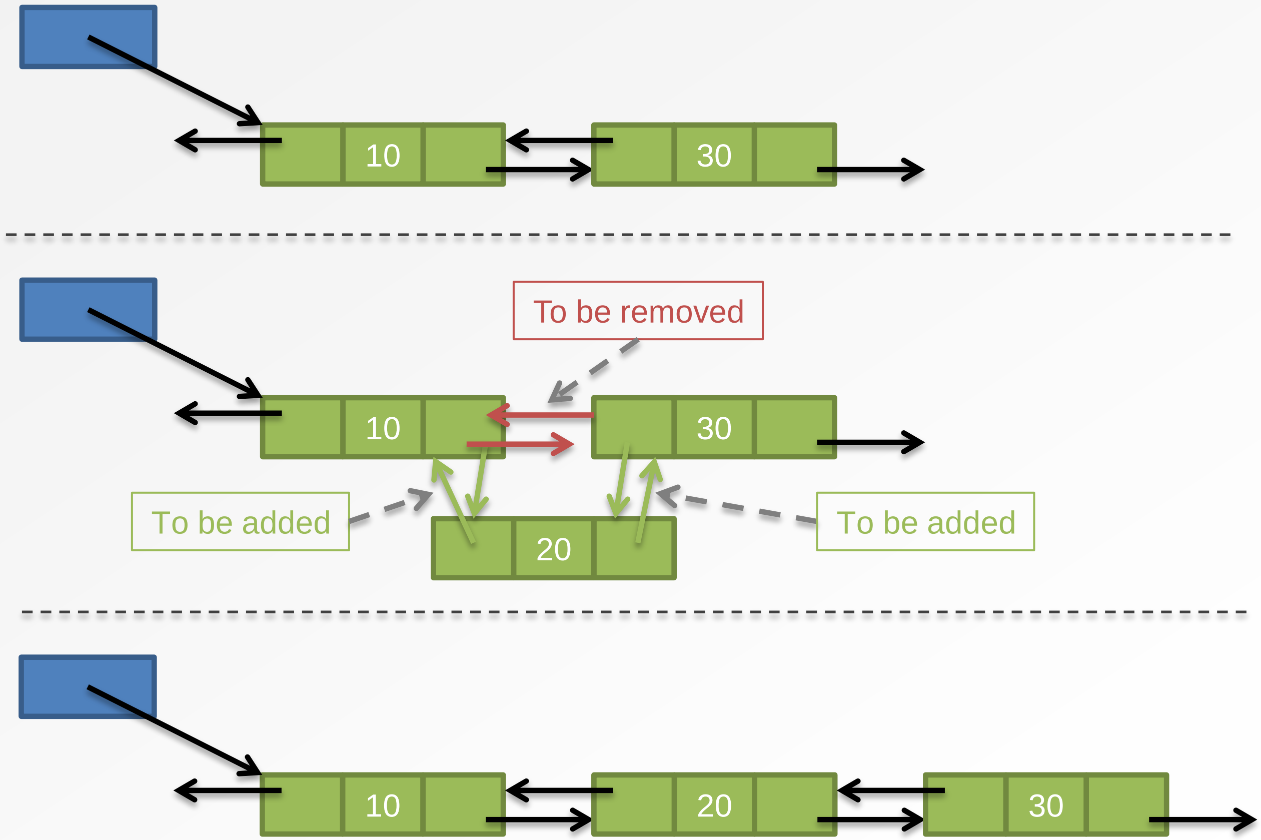
```
typedef struct dbllistnode{
    int item;
    struct _dbllistnode *pre;
    struct _dbllistnode *next;
} DblListNode;
```

- Note that first node has **pre == NULL**

- Inserting a node
  - Have to set the **pre** and next pointers accordingly for all nodes involved



DOUBLY LINKED LIST: INSERT



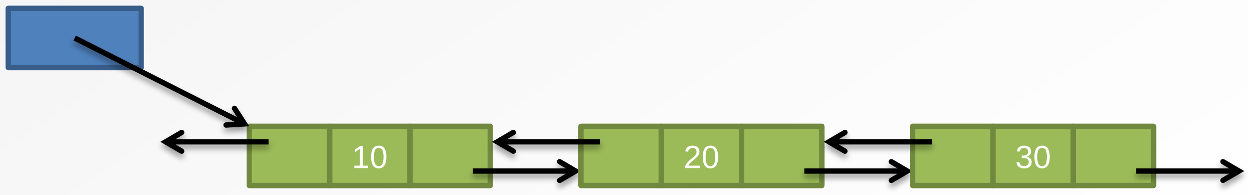
DOUBLY LINKED LIST

- Traversing a doubly linked list in forward direction

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

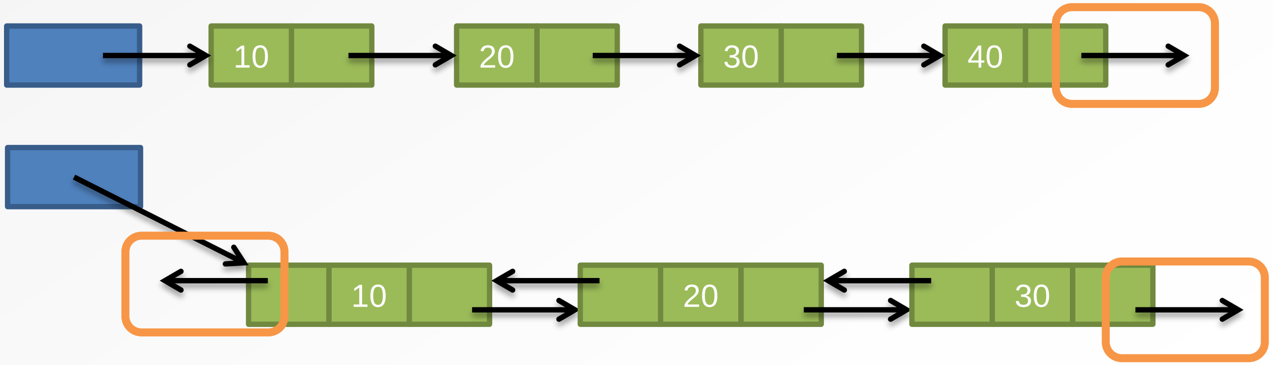
- Traversing a doubly linked list in backward direction

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



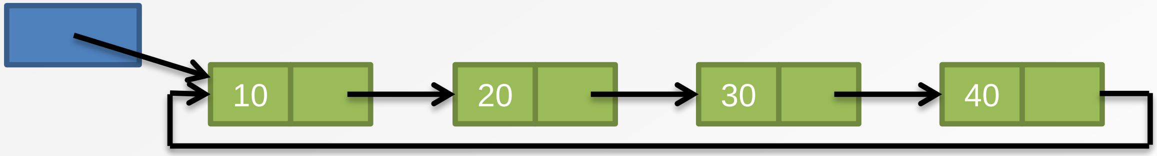
MORE COMPLEX LINKED LISTS - CIRCULAR LINKED LISTS

- So far, linked list has a fixed end
- 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

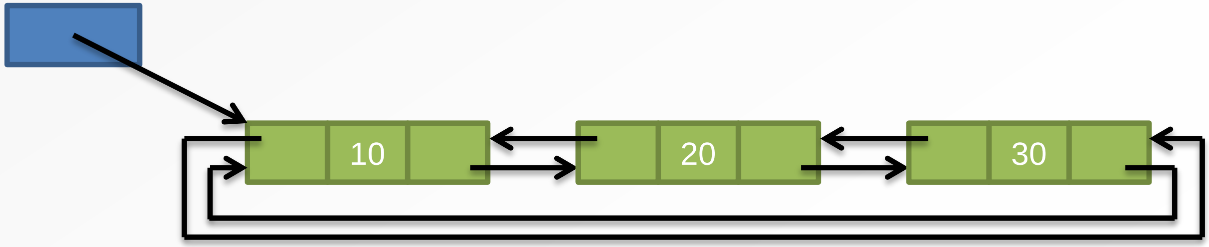


CIRCULAR LINKED LISTS

- Circular singly-linked lists
  - Last node has next pointer pointing to first node

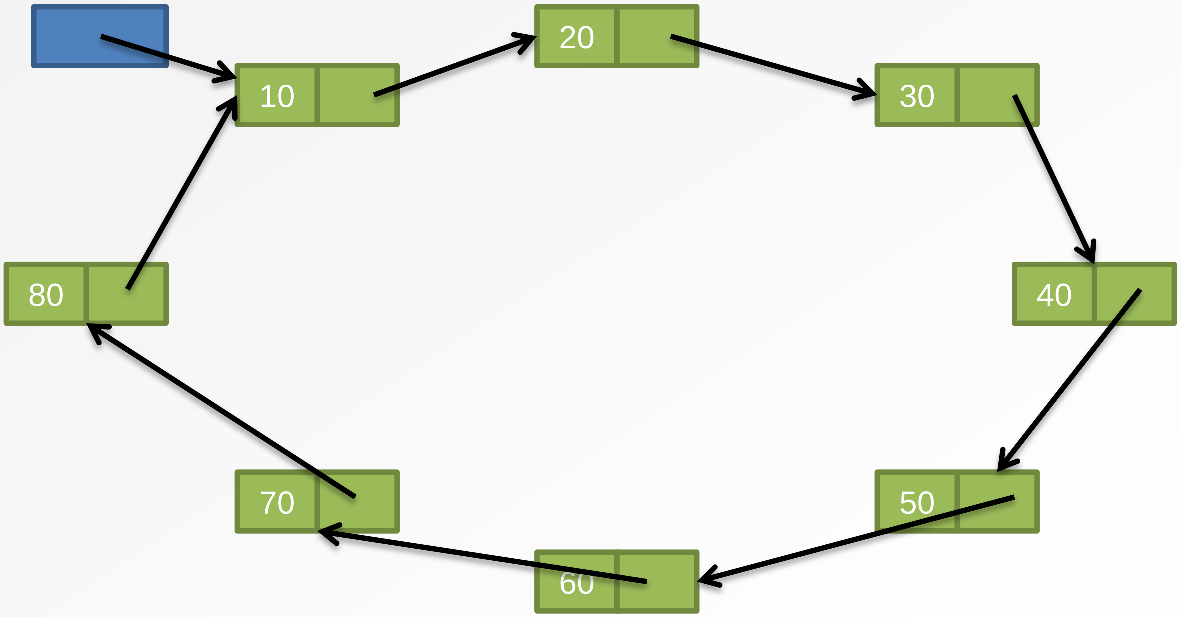


- Circular doubly-linked lists
  - Last node has next pointer pointing to first node
  - First node has pre pointer pointing to last node



CIRCULAR LINKED LISTS

- Effectively have this (singly-linked version)



OUTLINE

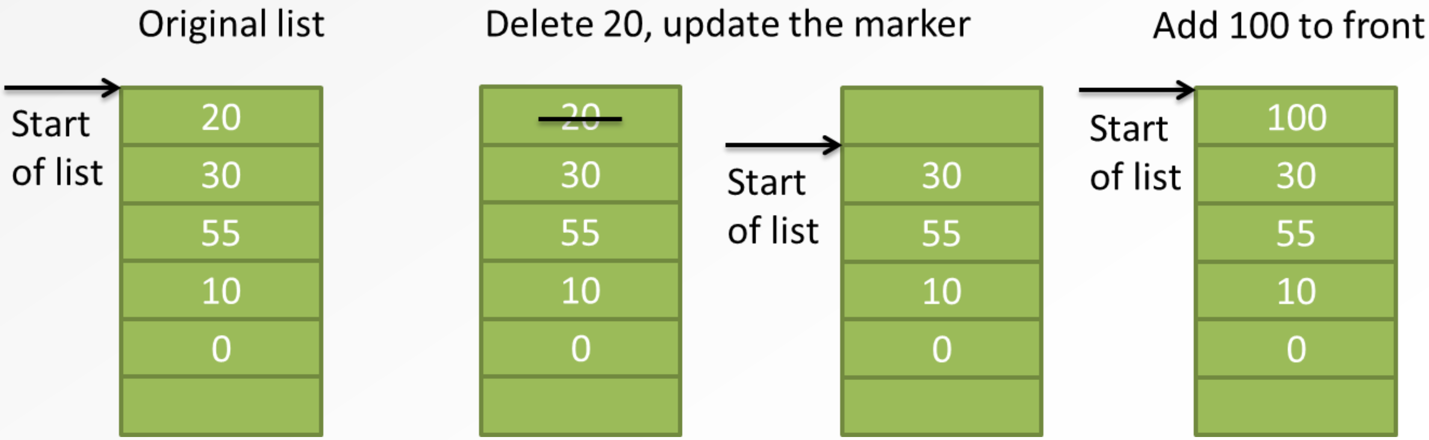
- sizeList() function
- Worked example: Using a linked list
- Linked list C struct
- More complex linked lists
  - Doubly-linked lists
  - Circular linked lists
  - Circular doubly-linked lists
- **Summary: Linked lists**

ARRAY-BASED LISTS

- Back to arrays as list storage
- Try to implement "smarter" array-based list
- Avoid some of the problems we saw earlier using arrays to store lists
  - Key is to **minimize shifting operations**
- Array is statics DS – "**static linked list**".
  - mimic the dynamic linked list.
  - avoid **common mistakes** when using pointers (**buffer overflows**, **memory leaks**).

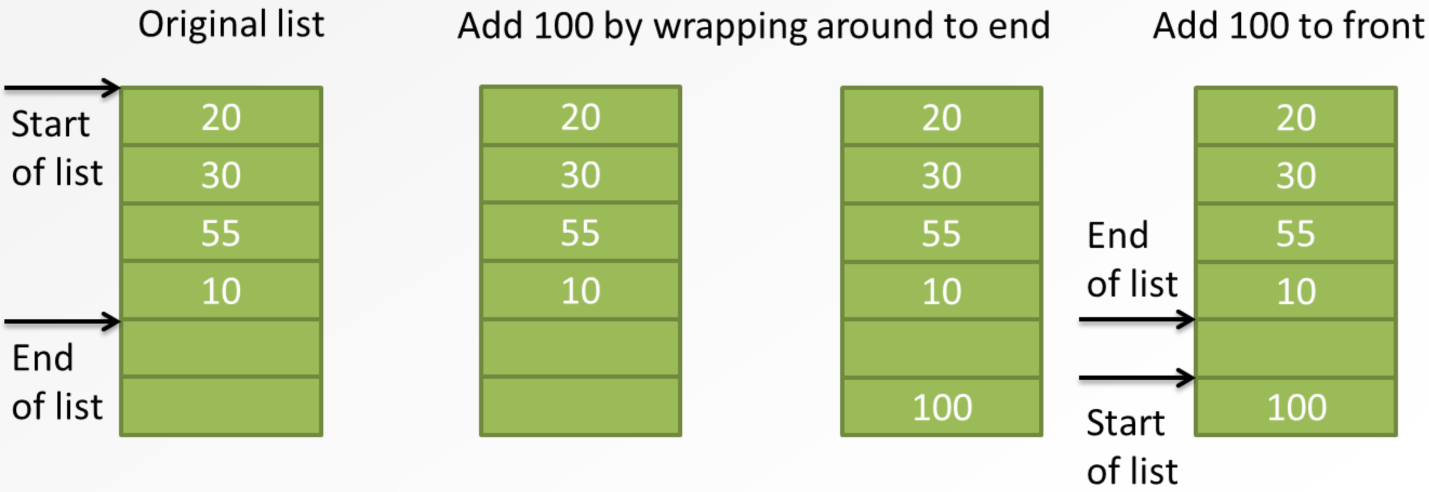
ARRAY-BASED LISTS

- Delete an item from the front of a list
  - Key idea: Leave the empty space, don’t shift everything down
  - In future, if adding to the front, empty space gets used
  - Use a marker (or index #) to store location of first actual item
- Try: Delete 20 from index 0, then add 100 to index 0



ARRAY-BASED LISTS

- Unfortunately, this doesn’t help once you run out of space in front
- Idea: Wrap around to the other end, circular array



ARRAYS VS. LINKED LISTS

- **Arrays**
  - Efficient random access
  - Difficult to expand, re-arrange
  - When inserting/removing items in the middle or at the front, computation time scales with size of list
  - Generally a better choice when data is immutable
- **Linked lists (dynamic-pointer-based and static-array-based)**
  - "Random access" can be implemented, but more inefficient than arrays
  - cost of storing links, only use internally.
  - Easy to shrink, rearrange and expand (but array-based linked list has a fixed size)
  - Insert/remove operations only require fixed number of operations regardless of list size. no shifting
- Know when to choose an array vs a linked list

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
- Summary: Linked lists