NANYANG TECHNOLOGICAL UNIVERSITY
SINGAPORE

CE1107/CZ1107: DATA STRUCTURES AND ALGORITHMS

Linked Lists II

College of Engineering
School of Computer Science and Engineering

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

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

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

- 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);
```

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

sizeList() [VERSION 1]

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

```
int sizeList(ListNode *head){

        int count = 0;

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

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

    return count;
}
```

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

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

WORKED EXAMPLE: LINKED LIST APPLICATION

```c
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

- 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

- 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

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

LINKED LIST APPLICATION [VERSION 2]

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

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

    return 0;
}
```

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

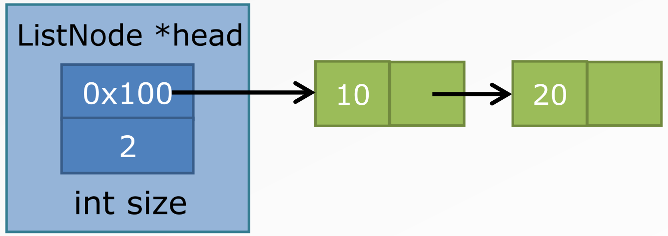
EXISTING LINKED LIST STRUCTURE

- 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

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

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);

CALLING NEW VERSION OF LINKED LIST FUNCTIONS

- Two versions of a small application

```
int main(){

    LinkedList ll;
    LinkedList *ptr_ll;

    insertNode(&ll, 0, 100);
    printList(&ll);
    printf("%d nodes\n", ll.size);
    removeNode(&ll, 0);

    ptr_ll = malloc(sizeof(LinkedList));
    insertNode(ptr_ll, 0, 100);
    printList(ptr_ll);
    printf("%d nodes\n", ptr_ll->size);
    removeNode(ptr_ll, 0);
}
```

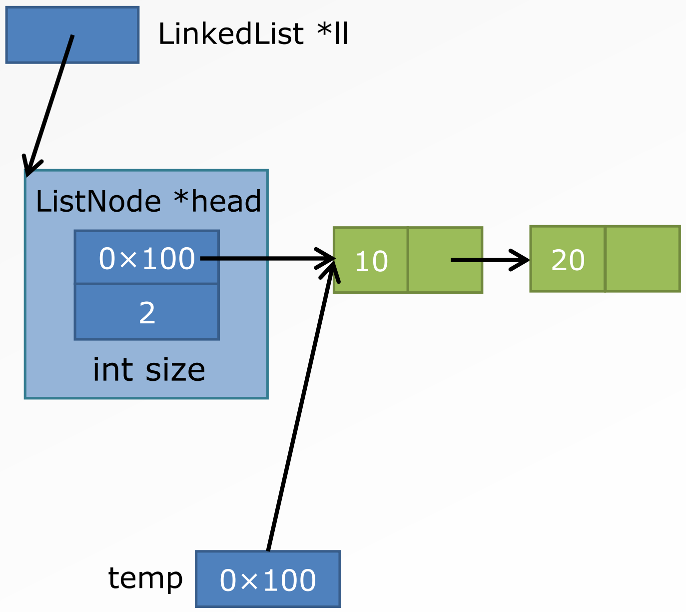
printList() USING LinkedList STRUCT

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

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

    if (temp == NULL)
        return;

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

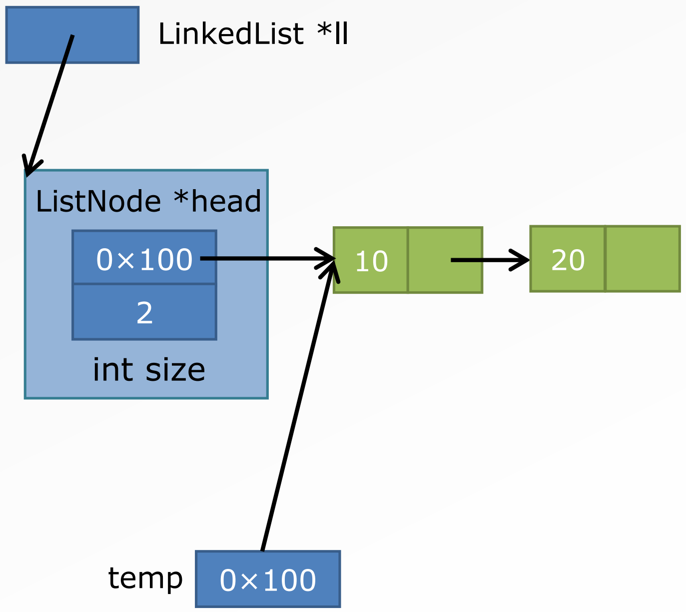


findNode() USING LinkedList STRUCT

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

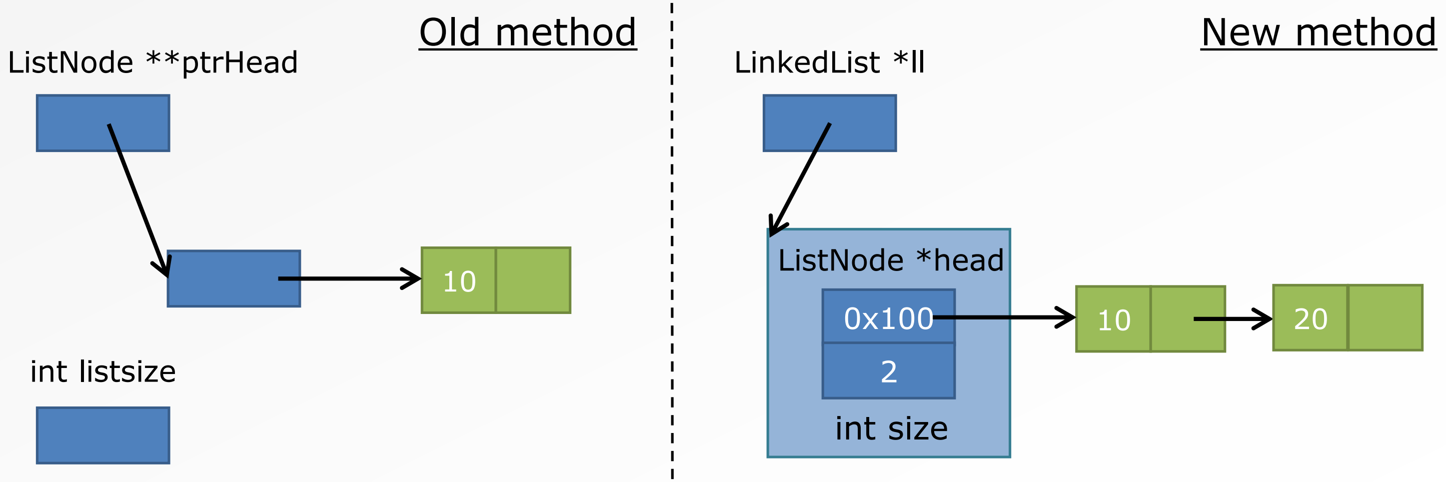
```
ListNode * findNode(
    LinkedList *ll, int index){
    ListNode *temp = ll->head;
    if (temp == NULL || index < 0)
        return NULL;

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



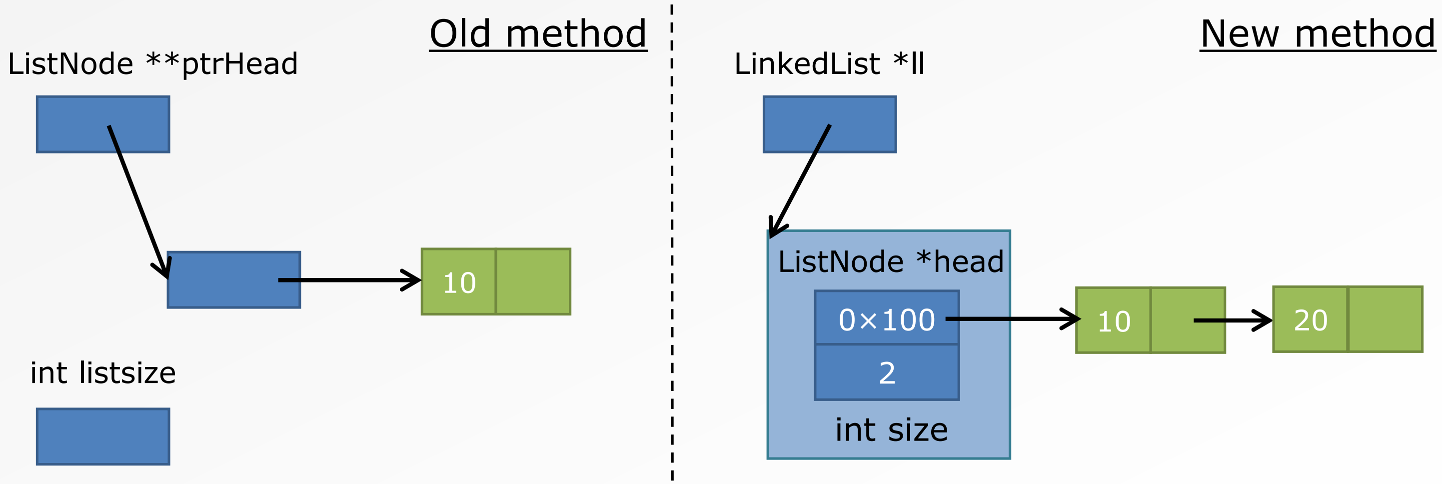
insertNode() USING LinkedList STRUCT

- 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



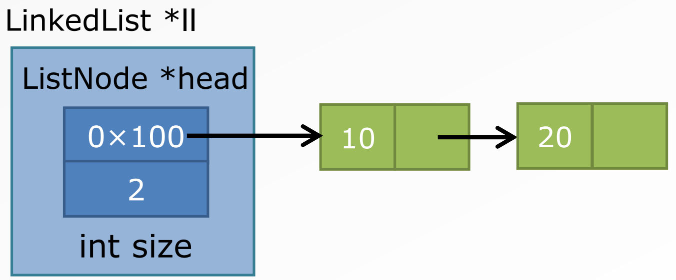
insertNode() USING LinkedList STRUCT

- 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 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
- 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

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

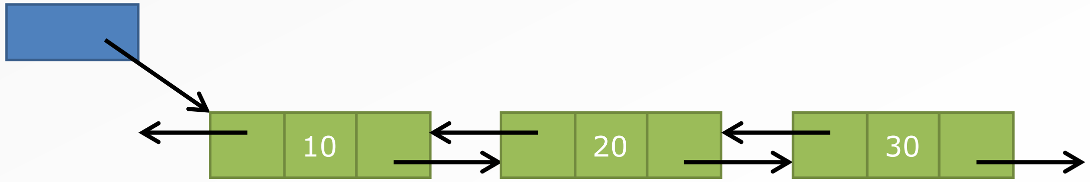
MORE COMPLEX LINKED LISTS

- 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

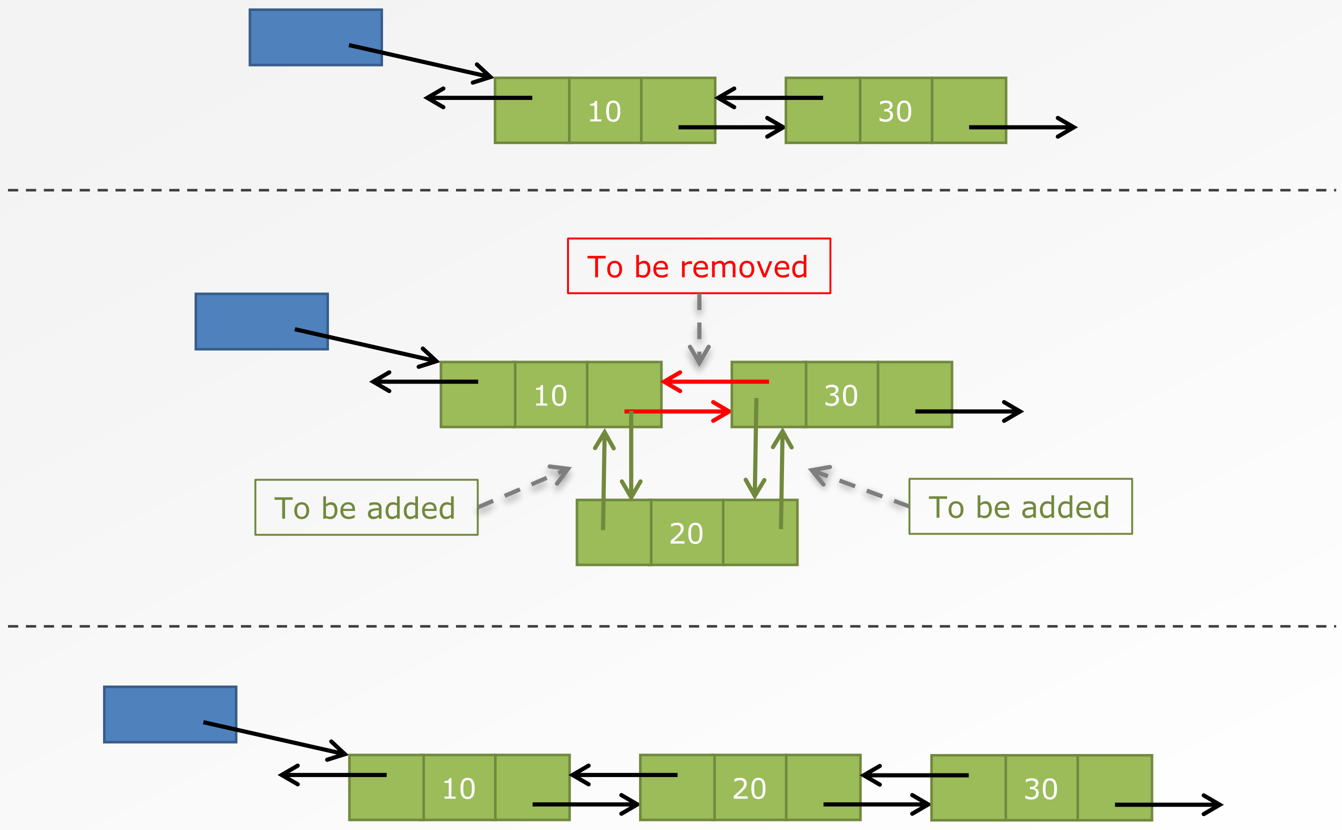
- Modify the ListNode struct

```
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



INSERTING A NODE INTO A DOUBLY LINKED LIST

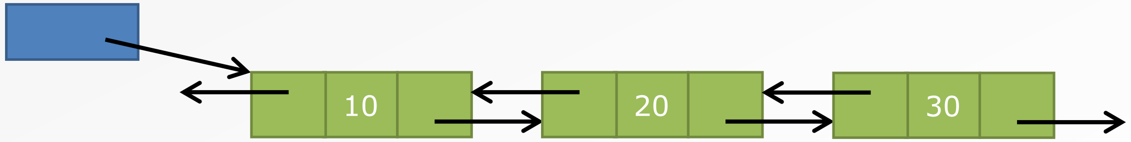


- Traversing a doubly linked list in forward direction

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

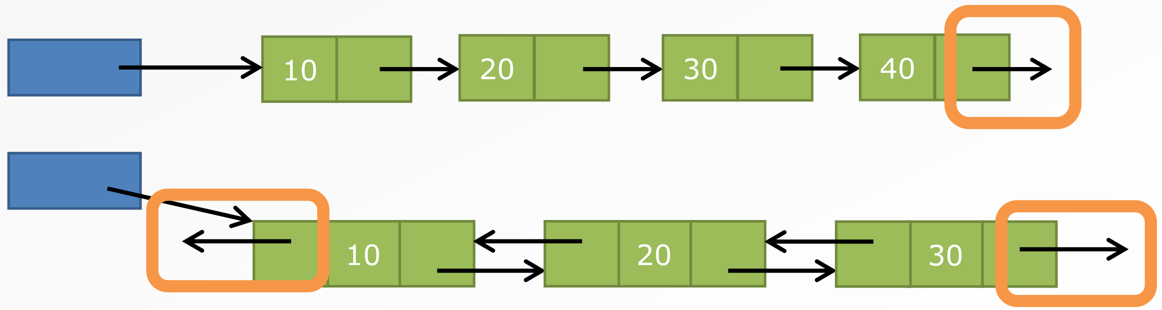
- Traversing a doubly linked list in backward direction

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



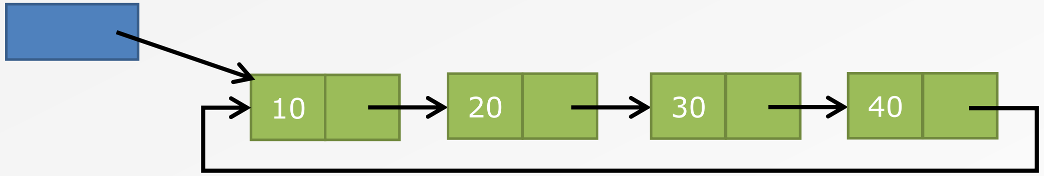
MORE COMPLEX LINKED LISTS (PART II)

- 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

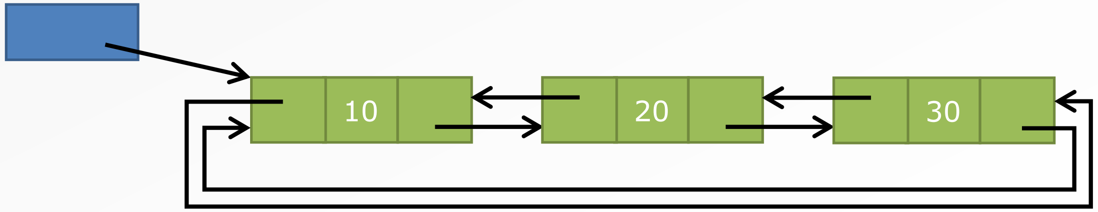


CIRCULAR LINKED LISTS

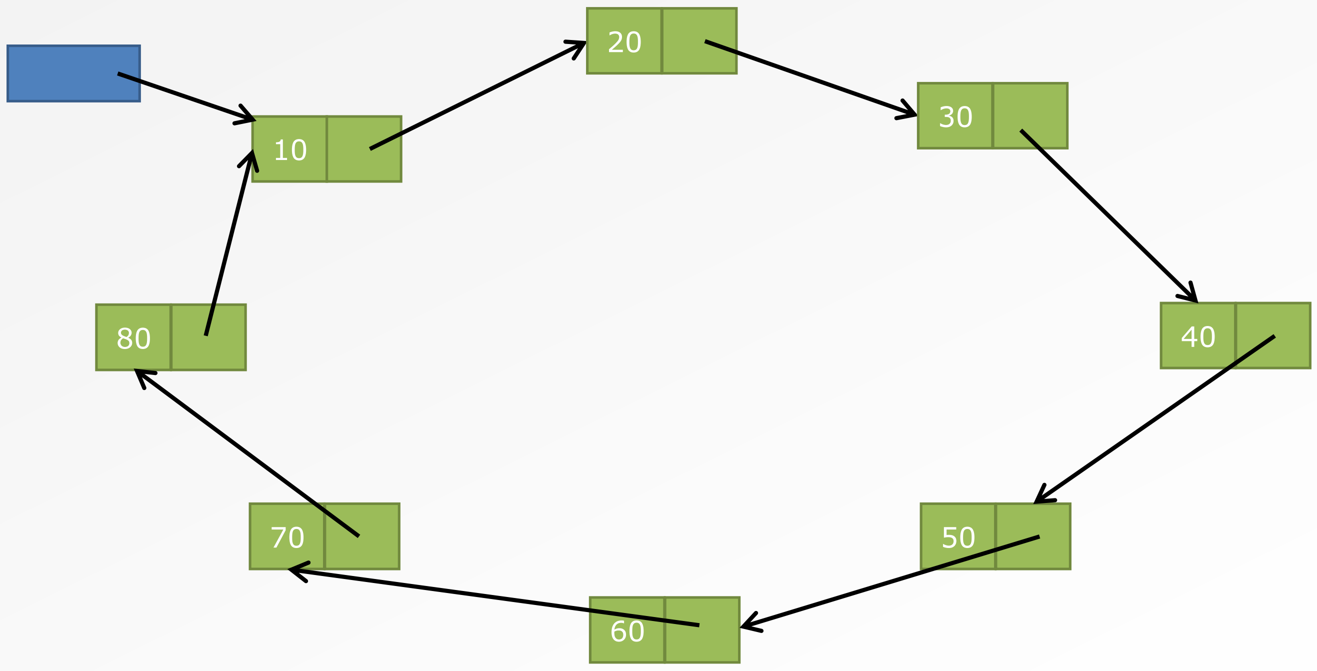
- Circular singly-linked lists
  - Last node has next pointer pointing to first node



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



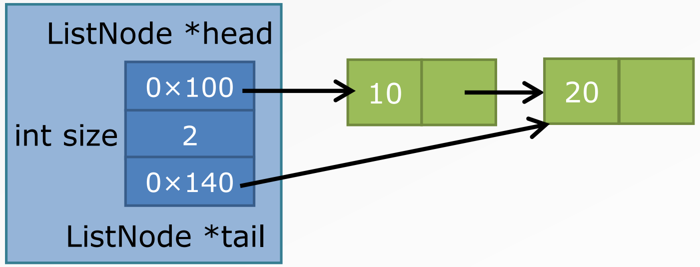
- Effectively we will have this (singly-linked version)



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?

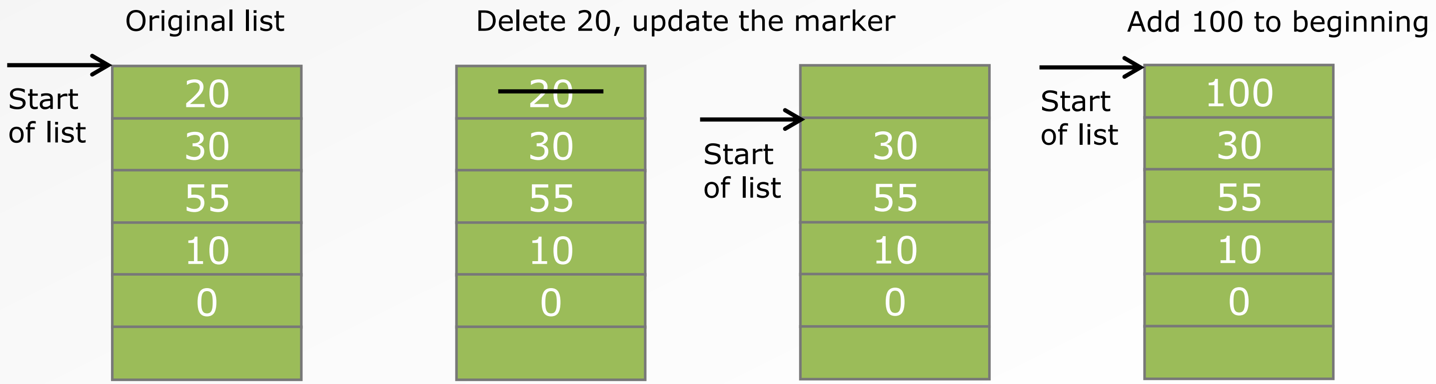
TODAY

- 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

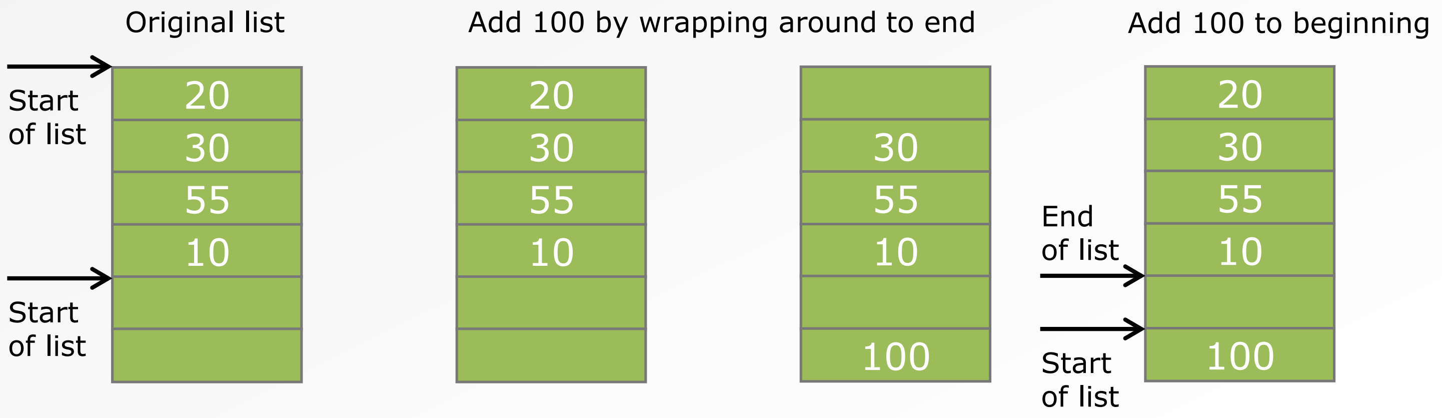
ARRAY-BASED LISTS

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

- 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



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



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**

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

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

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