NANYANG TECHNOLOGICAL UNIVERSITY

CE1107/CZ1107: DATA STRUCTURES AND ALGORITHMS

Stacks and Queues

College of Engineering
School of Computer Engineering

LEARNING OBJECTIVES

After this lesson, you should be able to:

- Explain how a stack data structure operates
- Implement a stack using a linked list
- Explain how a queue data structure operates
- Implement a queue using a linked list
- Choose stack or queue data structure when given an appropriate problem to solve

STACK DATA STRUCTURE

- Stack data structure
- Stack implementation using linked lists
- Stack functions
  - push()
  - pop()
  - peek()
  - isEmptyStack()
- Working examples: Applications

STACK DATA STRUCTURE

- **Stack data structure**
- Stack implementation using linked lists
- Stack functions
  - push()
  - pop()
  - peek()
  - isEmptyStack()
- Working examples: Applications

PREVIOUSLY

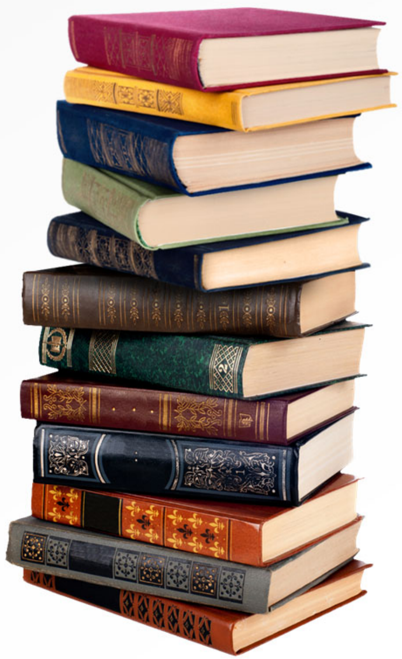
- **Arrays**
  - Random access data structure
  - Access any element directly
    - array[index]
- **Linked lists**
  - Sequential access data structure
  - Have to go through a particular sequence when accessing elements
    - temp->next until you find the right node
- Today, consider one example of limited-access sequential data structures

STACK DATA STRUCTURE

- A **stack** is a data structure that operates like a physical stack of things
  - Stack of books, for example
  - Elements can only be added or removed at the top

- Key: Last-In, First-Out (LIFO) principle
  - Or, First-In, Last-Out (FILO)

- Often built on top of some other data structure
  - Arrays, Linked lists, etc.
  - We'll focus on a linked-list based implementation



STACK DATA STRUCTURE

- **Core operations**
  - Push: Add an item to the top of the stack
  - Pop: Remove an item from the top of the stack
- **Common helpful operations**
  - Peek: Inspect the item at the top of the stack without removing it
  - IsEmptyStack: Check if the stack has no more items remaining
- **Corresponding functions**
  - push()
  - pop()
  - peek()
  - isEmptyStack()
- We'll build a stack assuming that it only deals with integers
  - But as with linked lists, can deal with any contents depending on how you define the functions and the underlying implementation

TODAY

- Stack data structure
- **Stack implementation using linked lists**
- Stack functions
  - push()
  - pop()
  - peek()
  - isEmptyStack()
- Working examples: Applications

STACK IMPLEMENTATION USING LINKED LISTS

- Recall that we defined a LinkedList structure
  - Encapsulates all required variables inside a single object
  - Conceptually neater to deal with

- Similarly, define a Stack structure.
  - We're going to build our stack on top of a linked list

```
typedef struct _stack{
    LinkedList ll;
} Stack;
```

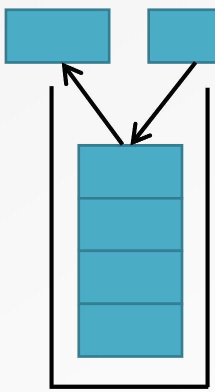
STACK IMPLEMENTATION USING LINKED LISTS

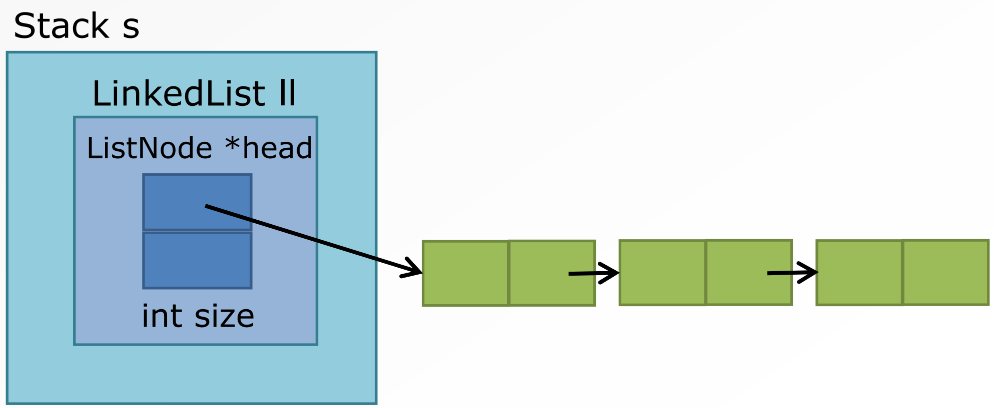
- Stack structure

```
typedef struct _stack{
    LinkedList ll;
} Stack;
```

Notice this is a LinkedList, not a LinkedList *

- Basically wrap up a linked list and use it for the actual data storage
- Just need to ensure we control where elements are added/removed
- Notice that the LinkedList already takes care of little things like keeping track of number of nodes, etc.



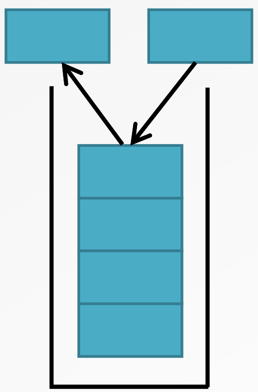


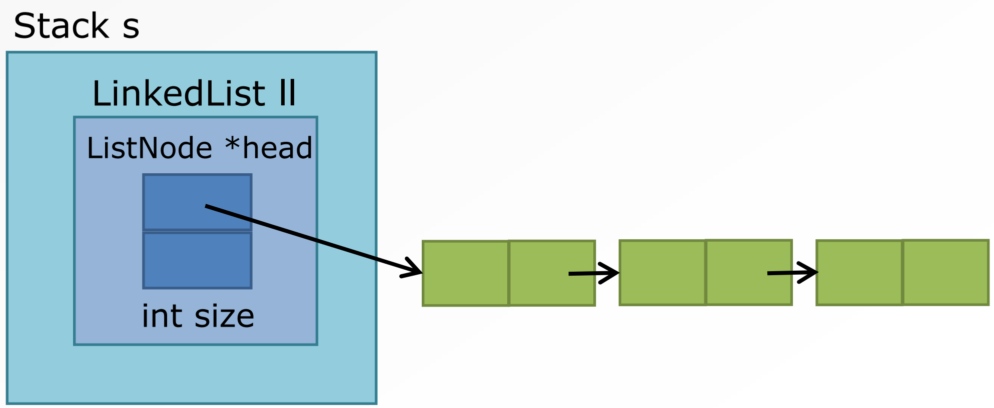
TODAY

- Motivating application
- Stack data structure
- Stack implementation using linked lists
- **Stack functions**
    - push()
    - pop()
    - peek()
    - isEmptyStack()
- Working examples: Applications

STACK FUNCTIONS: push()

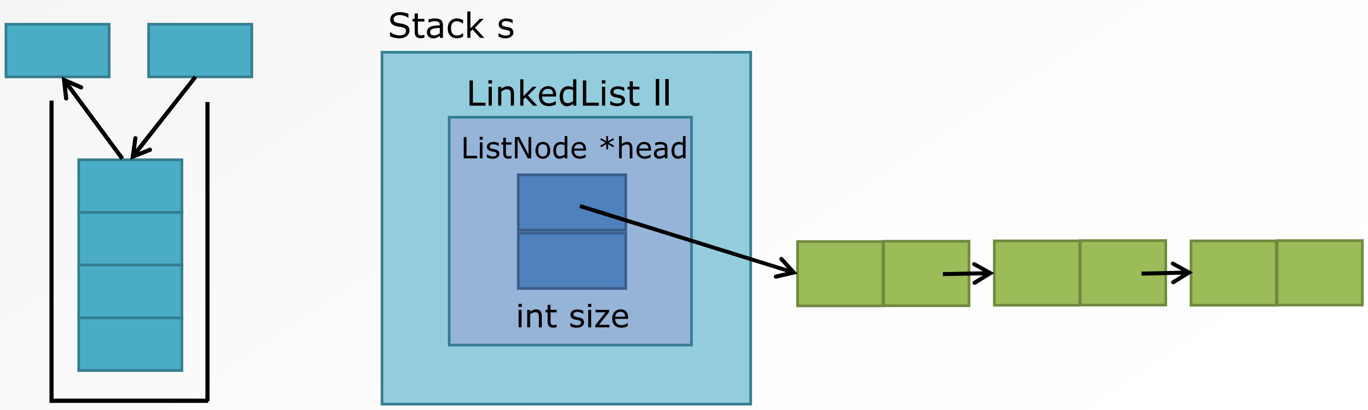
- Push() function is the only way to add an element to the stack data structure
- Only allowed to push() onto the top of the stack
- Question:
  - Using a linked list as the underlying data storage, does the first linked list node represent the top of the bottom of the stack?





STACK FUNCTIONS: push()

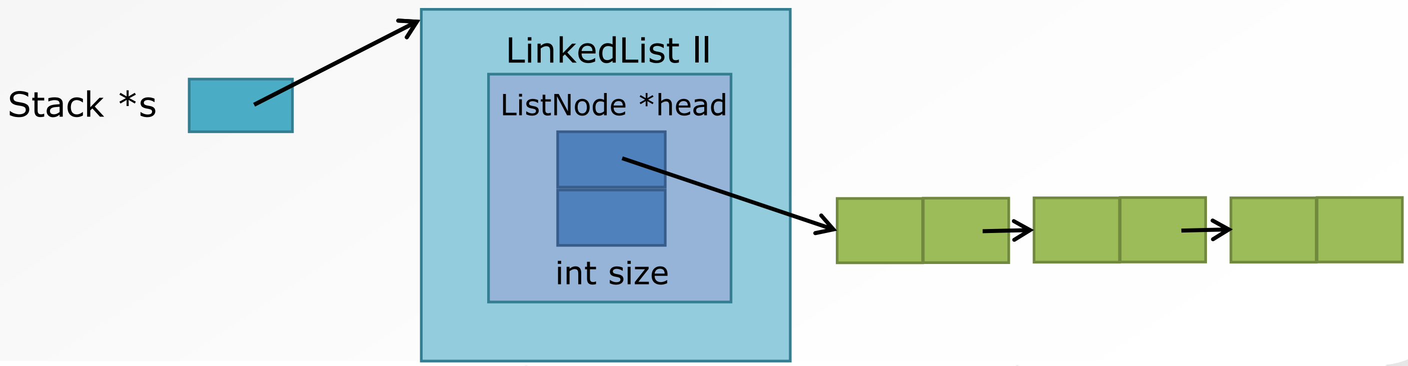
- **Hands-on: Write the push() function**
  - Define the function prototype
  - Implement the function
- Answer is a few slides down, so don't look yet
- Requirements
  - Make use of the LinkedList functions we've already defined
  - Insert a new integer (what data type for the "item"?)
  - Insert at the top only (what index position?)



STACK FUNCTIONS: push()

- First linked list node corresponds to the top of the stack
- Last linked list node corresponds to the bottom of the stack
- Pushing a new node onto the stack → adding a new node to the front of the linked list

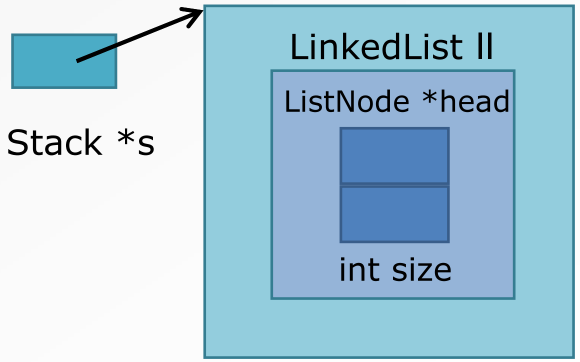
```
void push(Stack *s, int item){
    insertNode(&(s->ll), 0, item);
}
```



STACK FUNCTIONS: pop()

- Popping a value off the top of the stack is a two-step process
  - Get the value of the node at the front of the linked list
  - Removing that node from the linked list

```
int pop(Stack *s){
    int item;
    item = ((s->ll).head->item;
    removeNode(&(s->ll), 0);
    return item;
}
```

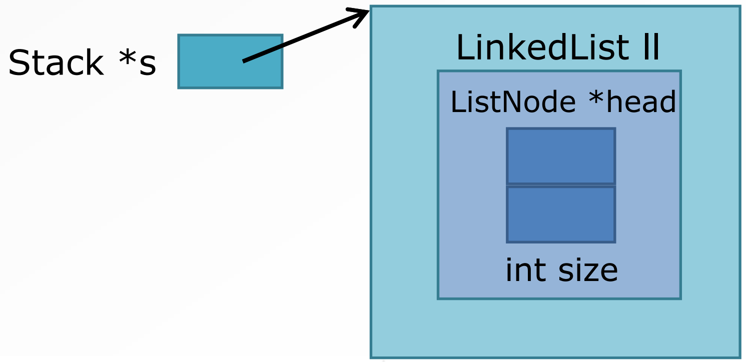


- Need a temporary int variable to hold the stored value because I can’t get it after I remove the top node

STACK FUNCTIONS: peek()

- Peek at the value on the top of the stack
  - Get the value of the node at the front of the linked list
    - Without removing the node

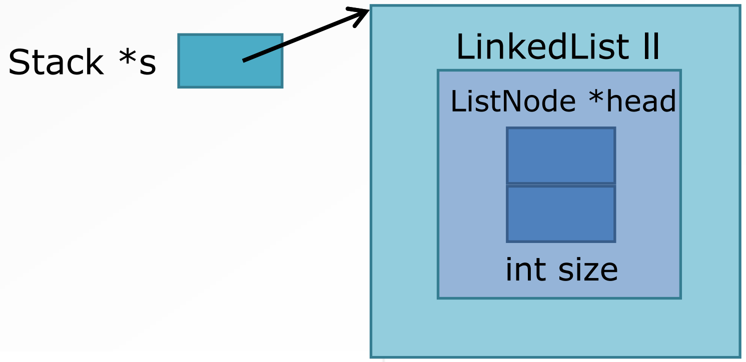
```
int peek(Stack *s){
    return ((s->ll).head)->item;
}
```



STACK FUNCTIONS: isEmptyStack()

- Check to see if number of nodes == 0
- Make use of the built-in size variable in the LinkedList struct

```
int isEmptyStack(Stack *s){
    if ((s->ll).size == 0) return 1;
    return 0;
}
```



WORKING EXAMPLES: APPLICATIONS

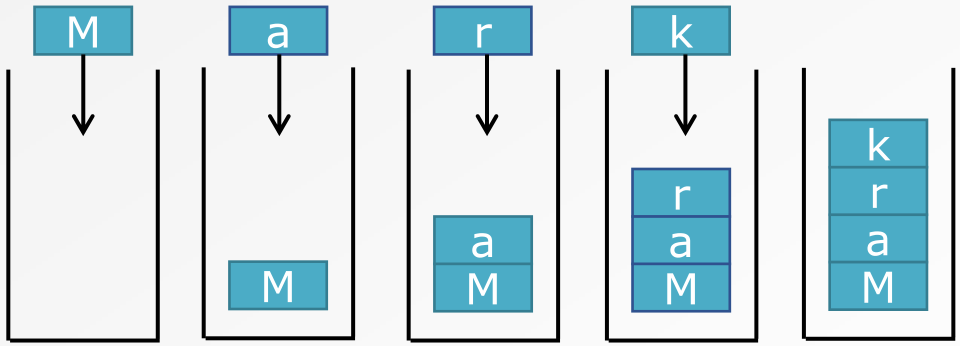
- Motivating application
- Stack data structure
- Stack implementation using linked lists
- Stack functions
  - push()
  - pop()
  - peek()
  - isEmptyStack()
- **Working examples: Applications**

SIMPLE APPLICATION #1: REVERSE STRING

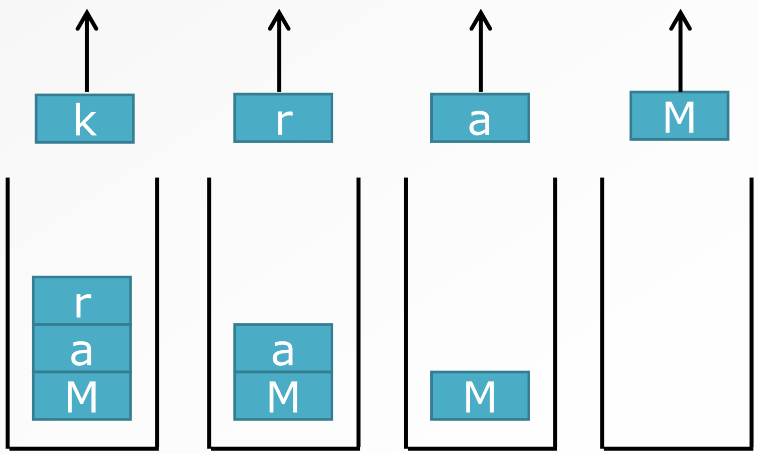
- Stacks are useful for reversing items
- Reverse a string: **Mark**
- Idea:
  - Push each letter on the stack
  - When there are no more letters in the original string, pop one by one from the stack
  - The letters will be popped in reverse order from their original position in the string

SIMPLE APPLICATION #1: REVERSE STRING

- Step 1
  - Push onto stack until no more letters



- Step 2
  - Pop from stack until stack is empty



SIMPLE APPLICATION #2: REVERSE LIST OF INTEGERS

```
int main(){
    int i = 0;
    Stack s;
    s.ll.head = NULL;

    printf("Enter a number: ");
    scanf("%d", &i);
    while (i != -1){
        push(&s, i);
        printf("Enter a number: ");
        scanf("%d", &i);
    }
    printf("Popping stack: ");
    while (!isEmptyStack(&s))
        printf("%d ", pop(&s));
    return 0;
}
```

```
void push(Stack *s, int item);
void push(Stack *s, int item);
int isEmptyStack(Stack *s);
int peek(Stack *s);
```

QUEUE DATA STRUCTURE

- Stack data structure
- Stack implementation using linked lists
- Stack functions
  - push()
  - pop()
  - peek()
  - isEmptyStack()
- Working examples: Applications

QUEUE DATA STRUCTURE

- **Queue data structure**
- Queue implementation using linked lists
- Queue functions
  - enqueue()
  - dequeue()
  - peek()
  - isEmptyQueue()
- Worked examples: Applications

PREVIOUSLY

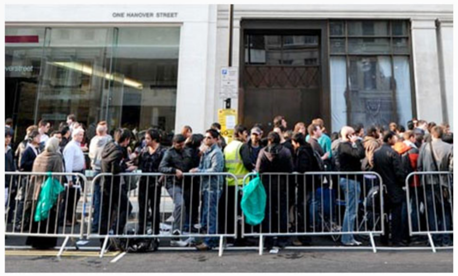
- Arrays
  - Random access data structure
- Linked lists
  - Sequential access data structure
- Limited-access sequential data structures
  - Stack
    - Last In, First Out (LIFO)
- Today, another limited-access sequential data structure

QUEUE DATA STRUCTURE

- A **Queue** is a data structure that operates like a real-world queue
  - Queue to use an ATM or buy food, for example
  - Elements can only be added at the back
  - Elements can only be removed from the front

- Key: First-In, First-Out (FIFO) principle
  - Or, Last-In, Last-Out (LILO)

- As with stacks, often built on top of some other data structure
  - Arrays, Linked lists, etc.
  - We'll focus on a linked-list based implementation again



QUEUE DATA STRUCTURE

- **Core operations**
  - Enqueue: Add an item to the back of the queue
  - Dequeue: Remove an item from the front of the queue
- **Common helpful operations**
  - Peek: Inspect the item at the front of the queue without removing it
  - IsEmptyStack: Check if the queue has no more items remaining
- **Corresponding funtions**
  - enqueue()
  - dequeue()
  - peek()
  - isEmptyQueue()
- We'll build a queue assuming that it only deals with integers
  - But as with linked lists and stacks, can deal with any contents depending on your code

- **Queue implementation using linked lists**
- Queue functions
  - enqueue()
  - dequeue()
  - peek()
  - isEmptyQueue()
- Worked examples: Applications

QUEUE IMPLEMENTATION USING LINKED LISTS

- Recall that we defined a LinkedList structure
- Next, we define a Stack structure
- Now, define a Queue structure
  - We'll build our queue on top of a linked list

```
typedef struct _queue{
    LinkedList ll;
} Queue;
```

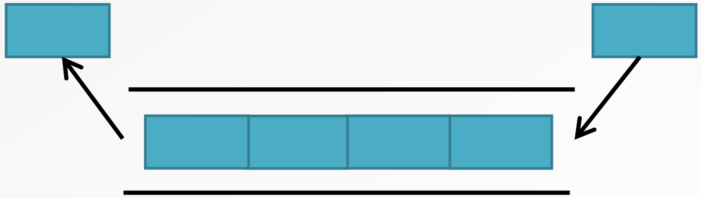
QUEUE IMPLEMENTATION USING LINKED LISTS

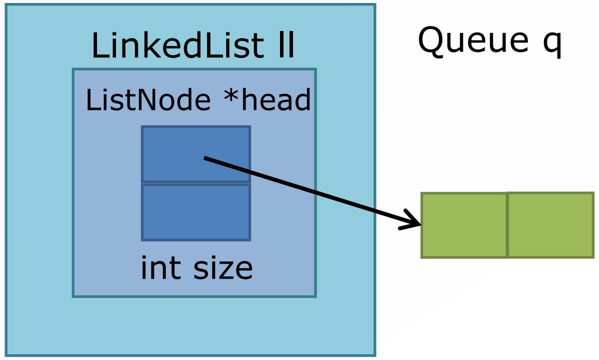
- Queue structure

```
typedef struct _queue{
    LinkedList ll;
} Queue;
```

Notice this is a LinkedList, not a LinkedList *

- Again, wrap up a linked list and use it for the actual data storage
- Notice that the LinkedList already takes care of little things like keeping track of # of nodes, etc.
- There is one modification we need for a queue… KIV



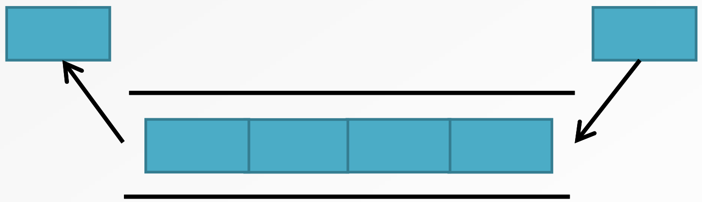


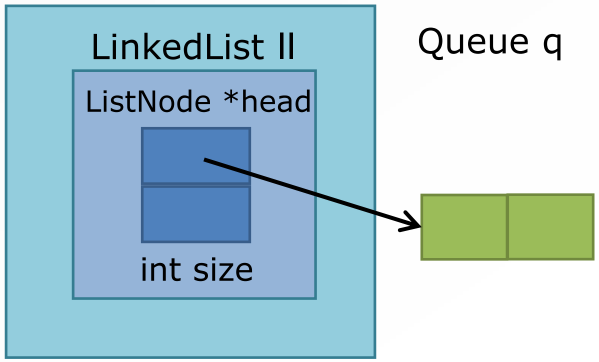
TODAY

- Queue data structure
- Queue implementation using linked lists
- **Queue functions**
  - enqueue()
  - dequeue()
  - peek()
  - isEmptyQueue()
- Worked examples: Applications

QUEUE FUNCTIONS: enqueue()

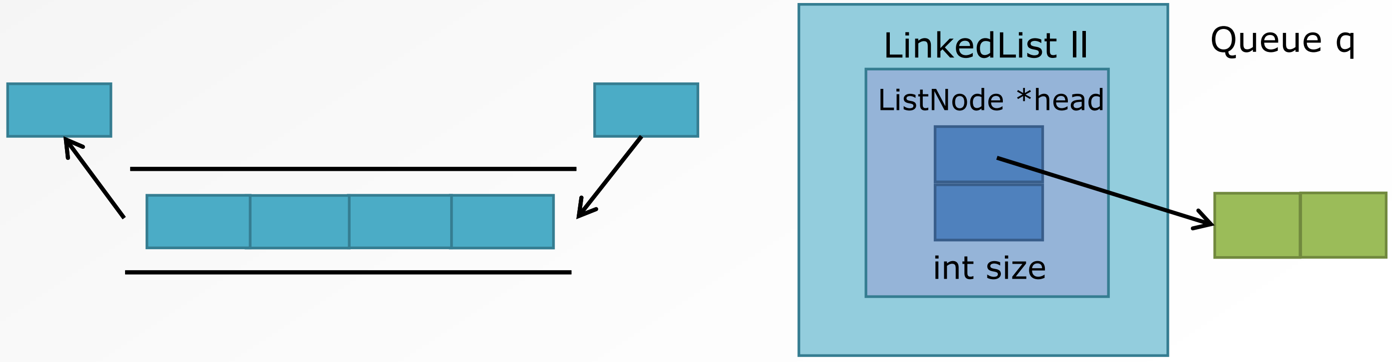
- enqueue() function is the only way to add an element to the queue data structure
- Only allowed to enqueue() at the end
- Question:
  - Using a linked list as the underlying data storage, does the first linked list node represent the front or the back of the queue?
  - Figure out which option makes it easier to implement enqueue() and dequeue()





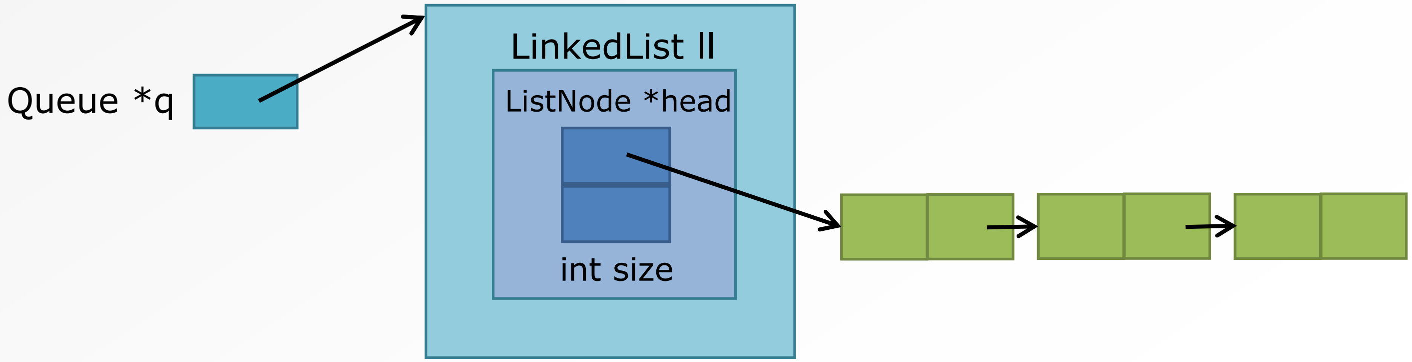
QUEUE FUNCTIONS: enqueue()

- **Hands-on: Write the enqueue() function**
  - Define the function prototype
  - Implement the function
  - Very similar to what we did for stack: push()
- Answer is a few slides down, so don't look yet
- Requirements
  - Make use of the LinkedList functions we've already defined
  - Insert at the <u>back</u> only (what index position?)



QUEUE FUNCTIONS: enqueue()

```
void enqueue(Queue *q, int item){
    insertNode(&(q->ll), q->ll.size, item);
}
```



QUEUE FUNCTIONS: enqueue()

- First linked list node corresponds to the front of the queue
- Last linked list node corresponds to the back of the queue
- Enqueueing a new item → adding a new node to the end of the linked list

```
void enqueue(Queue *q, int item){
    insertNode(&(q->ll), q->ll.size, item);
}
```

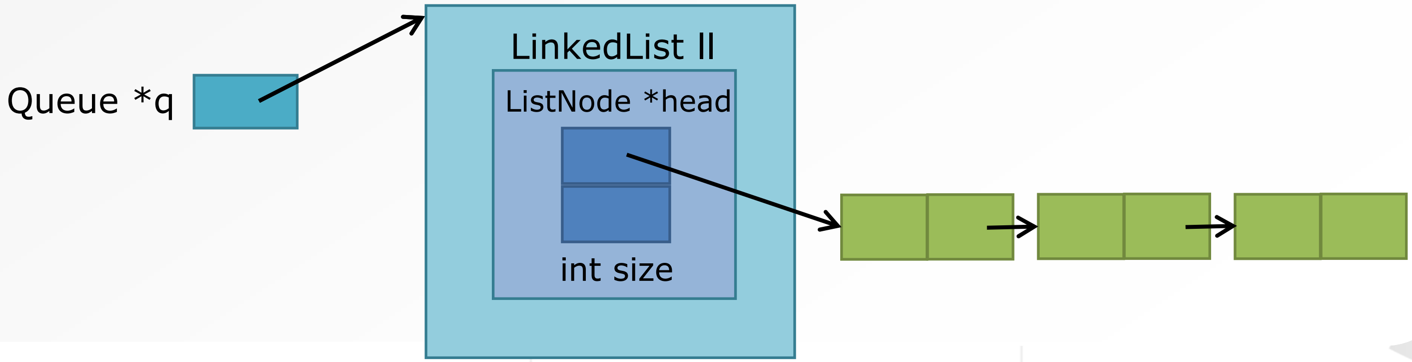
  - Notice that this could be a very <u>inefficient</u> operation if the queue is long
- Need to use a tail pointer to make the operation <u>efficient</u>
  - Gives us direct access to the current last node of the linked list
- Also note that the inefficient version <u>still works</u>

QUEUE FUNCTIONS: dequeue()

- Dequeueing a value is a two-step process again
  - Get the value of the node at the front of the linked list
  - Remove that node from the linked list

- Need a temporary int variable to hold the stored value because we can't get it after we remove the front node

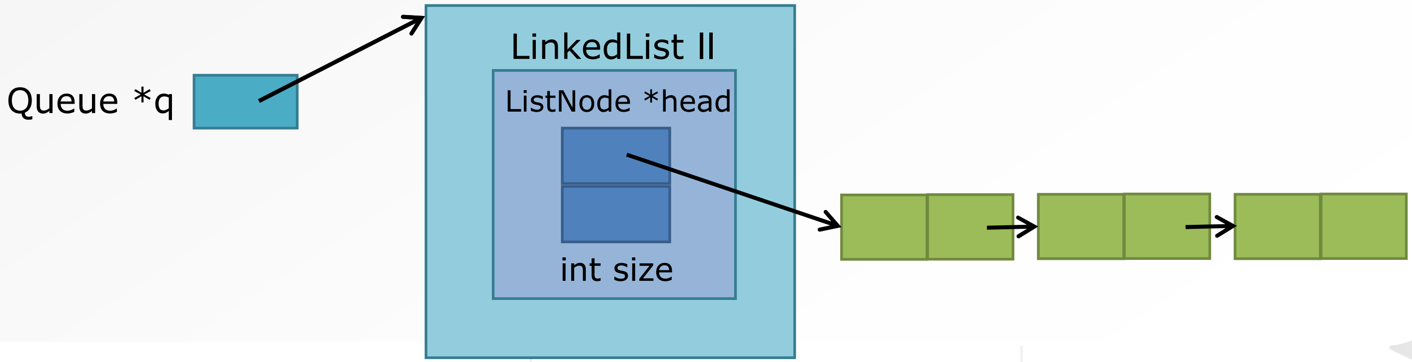
```
int dequeue(Queue *q){
    int item;
    item = ((q->ll).head)->item;
    removeNode(&ll, 0);
    return item;
}
```



QUEUE FUNCTIONS: peek()

- No change in logic from the stack version
- Peek at the value at the front of the queue
  - Get the value of the node at the front of the linked list
    - Without removing the node

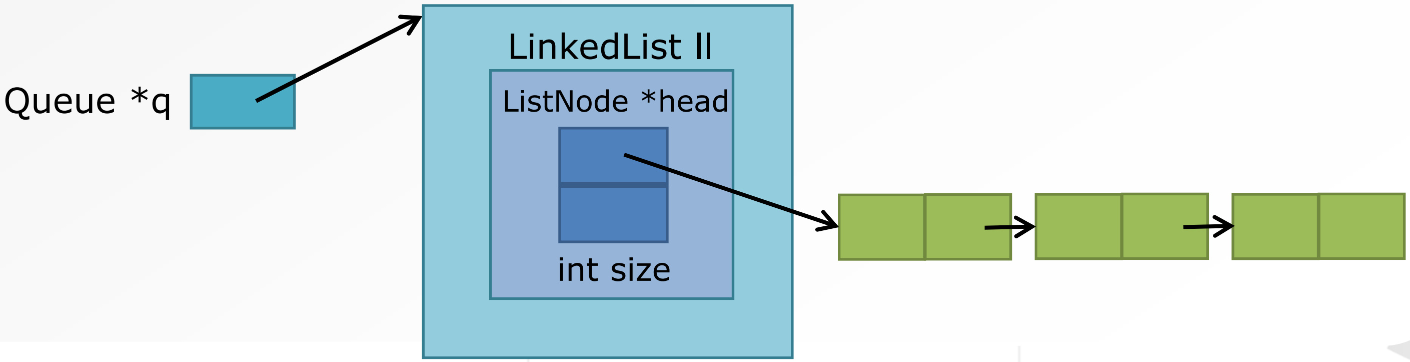
```
int peek(Queue *q){
    return ((q->ll).head)->item;
}
```



QUEUE FUNCTIONS: isEmptyQueue()

- Again, exactly the same logic as isEmptyStack()
- Check to see if # of nodes == 0
- Make use of the built-in size variable in the LinkedList struct

```
int isEmptyQueue(Queue *q){
    if ((q->ll).size == 0) return 1;
    return 0;
}
```



WORKED EXAMPLES: APPLICATIONS

- Motivating application
- Queue data structure
- Queue implementation using linked lists
- Queue functions
  - enqueue()
  - dequeue()
  - peek()
  - isEmptyQueue()
- **Worked examples: Applications**

SIMPLE TEST APPLICATION

```
int main(){
    Queue q;
    q.ll.head = NULL;
    q.ll.tail = NULL;

    enqueue(&q, 1);
    enqueue(&q, 2);
    enqueue(&q, 3);
    enqueue(&q, 4);
    enqueue(&q, 5);
    enqueue(&q, 6);

    while (!isEmptyQueue(&q))
        printf("%d ", dequeue(&q));
}
```

```
void enqueue(Queue *q, int item);
int dequeue(Queue *q);
int peek(Queue *q);
int isEmptyQueue(Queue *q);
```

TODAY

- Stack and Queue data structure
- Stack and Queue implementation using linked lists
- Stack functions
  - push(), pop(), peek(), isEmptyStack()
- Queue functions
  - enqueue(), dequeue(), peek(), isEmptyQueue()
- Worked examples: Applications