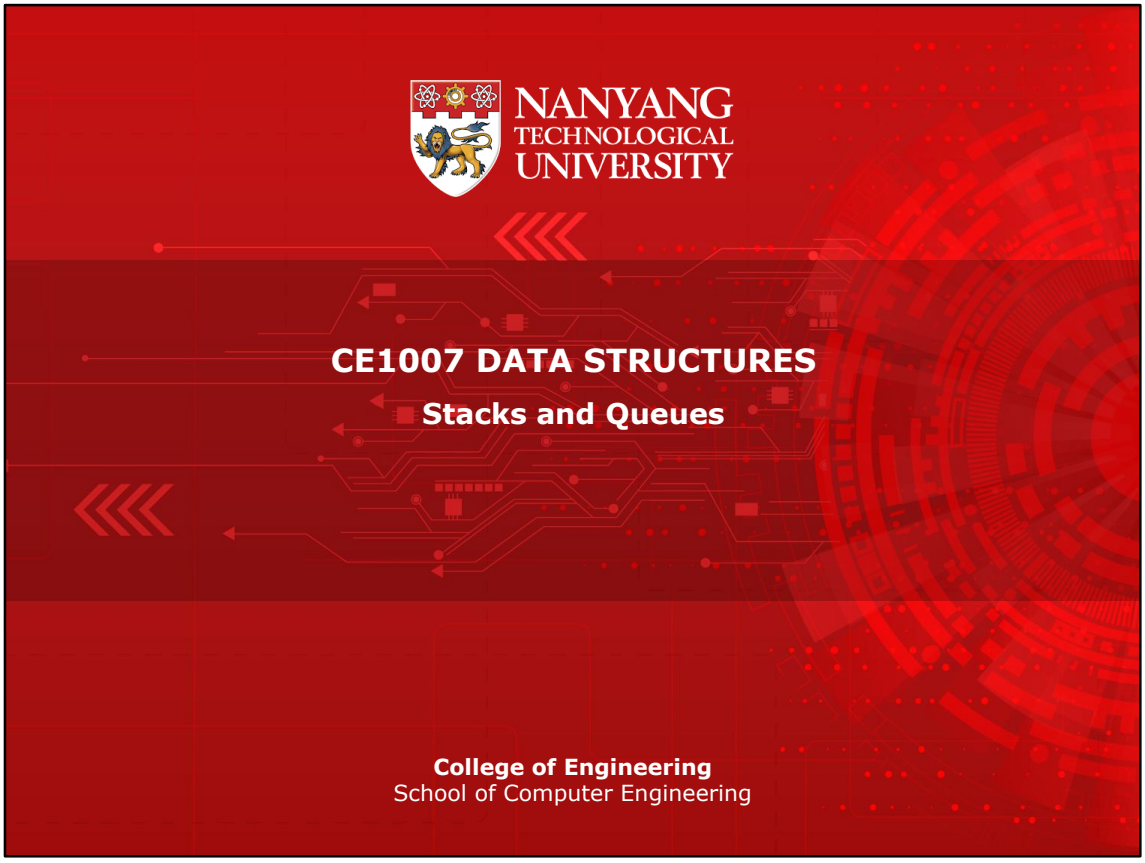


Today, we will cover two additional linear data structures that are built on top of the foundation of linked list. I have separated the materials as stacks and queues but you will notice that most of the slides are repeated or look very similar.

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

At the end of this lecture, you should be able to:

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

You also should be able to implement stack using an array but we will not cover or test this.

STACK DATA STRUCTURE

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

We will run through a simple motivating application which will help us to learn additional data structures. We will learn the concept and the implementation of stack data structures. The basic functions of stack will be discussed first and I will explain how we use stacks and why we might need to use stack in the first place by using a sample application.

STACK DATA STRUCTURE

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

Before we move to the implementation, we will learn what stack data structure is because it is quite intuitive.

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

In previous lectures, we talked about linked list structures and compared linked lists with arrays. An array is a sequence of contiguous elements that are squished together. It is a random access data structure. Therefore, once you access the array and when you know what exactly you want, you can straightaway jump into that element. But linked list does not have random access property. Because of the way linked list is created, I cannot straightaway jump to the desired node.

In linked lists, we have to use sequential access method to get to the desired node. That is why we discussed about inefficient operations because the longer the linked list gets, the worse the efficiency is.

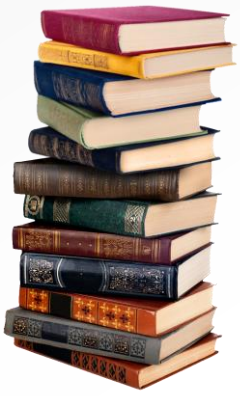
In the previous lecture, we have discussed how the difference between sequential access and random access leads to the strengths and weaknesses of arrays and linked lists, and at what situations you should choose them. Today, we will discuss about sequential access data structures and how we can constrain it further.

In previous lectures, we learned about insert node and remove node

functions. But, instead of removing a node from anywhere, sometimes you may want to constrain it in such a way that we insert or remove only in certain positions. We call this limited access linear sequential data structure.

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



A stack is a data structure which operates like a physical stack of things. For an example, if you have a stack of books, there are only certain places to where you can add a new item. You can only add a book to the top or remove from the top of the stack. Though there are multiple ways to add or remove a book from anywhere of a book stack, if I want to easily remove or insert one item, it should be from the top of the stack.

The key thing of this constrain is that we can be ended up with last in first out principle sometimes. This is because the first item you put in to the stack is at the bottom of the stack. Therefore, to remove items from the stack I need to start with the top most item.

We got the first in last out or last in first out because we constrain the position where we can access in this stack. Then we get to the implementation. Usually, it builds on top of other data structures. Therefore, instead of worrying about things like maintaining the order of the items, or special cases such as when the stack is empty and we need to insert an item.

You have already learned insert node and remove node functions that

covered all these special cases. Therefore, we are going to use these functions to implement a stack and we will use all linked list functions to do things like, add items, remove items, find the number of items, etc.

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

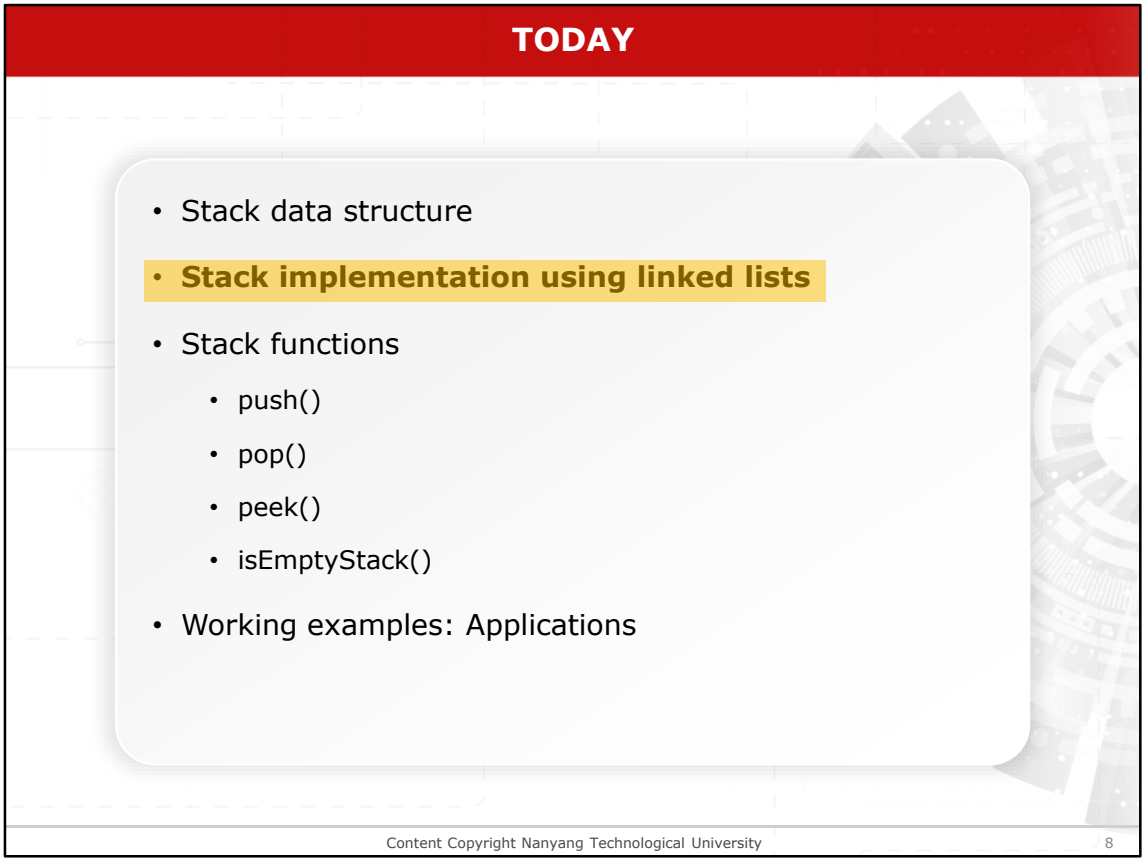
Same as in linked lists, we will insert and remove items from a stack but since we are restricting the position where the insert and remove items are placed, we will use a special name for it.

In stacks, for these operations we call push and pop. In push, we add an item to the top of the stack. And in pop, we remove an item from the top. These are the only two operations that allow you to modify data in a stack.

The peek function allows you to look at the top most item in the stack without actually removing it. If you think of a book stack, you may recall that you can look at the entire stack of books because by looking at the spines you can see the titles.

But since this is a computer program, you can only see the value by accessing the variable. You will see why peek function is helpful in future.

Empty stack is also a simple function which checks whether the stack is empty. It is helpful for special cases where we do not care how many items are left inside a stack, but care if it is empty or not.



In this part we will look at the actual implementation of stacks. We will use linked list struct to implement the stacks.

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;
```


If you remember how painful it was to understand the pass in pointer to the head pointer concept, you may recall that we learned why it is better to go with linked list structure that encapsulates all the parts of the structure.

Earlier, we used linked list structure to wrap up all the linked list items including the head pointer and size integer variable. Now, we use the linked list structure and warp it again inside a stack data structure.

We are going to take a copy of the linked list struct and throw it into a stack struct. This is a very simple struct. Inside the stack struct it has an actual linked list struct. This is not a pointer to a linked list.

This is useful, because we have all the functions and variables we need inside this linked list struct. Therefore, we do not need to rewrite the codes.

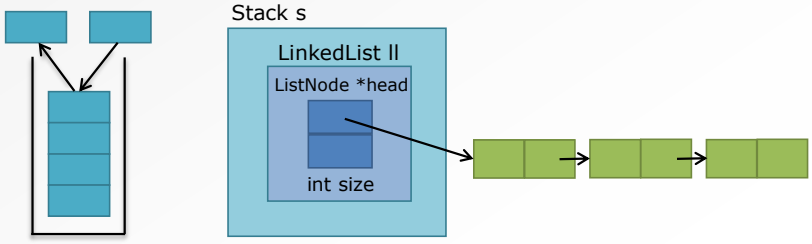
STACK IMPLEMENTATION USING LINKED LISTS

- 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.



Now the previous diagram has changed. We have an outer box for stack that covers the linked list struct. We are trying to build the concept that is visualized in the left diagram.

A stack has the values that are come in and go out only at the top position of the stack. But if we look at the memory layout, at the end we will get a stack structure. Inside the stack struct we have the linked list struct and inside the linked list struct we have two items as the head pointer and the size variable.

This is the simplest form of linked list we can have which does not include doubly linked nodes, or tail pointers. We have the nodes outside the struct and connected to the struct via the head pointer. The nodes are defined as external blocks of memory. Therefore, now we have a stack struct which wraps up the linked list structure which warps up a head pointer and a size variable.

In this way we do not need to worry about rewriting the code. For an example, if we want to add an item to the stack we have to call the insert node function in the linked list.

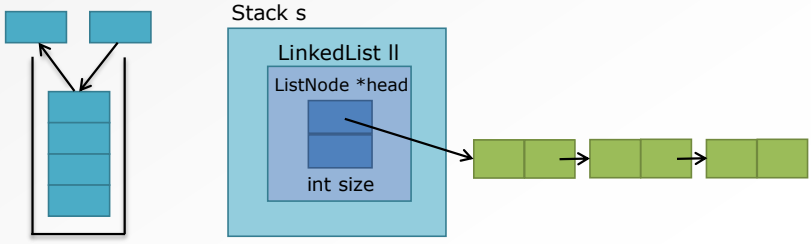
TODAY

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

Stack functions.

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?



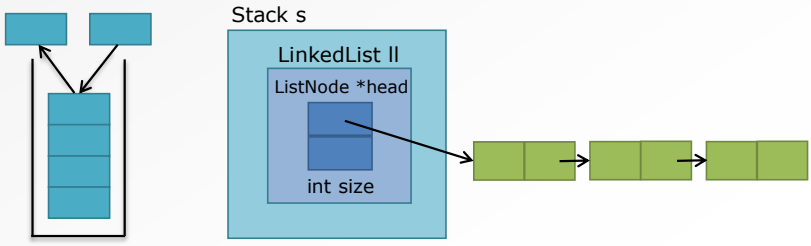
We have two core functions in stacks as push and pop. The push function is the only way you can use to add an item to the stack.

The first thing you need to figure out is how your linked list is arranged. Does the first linked list node correspond to the top position or the bottom position of the stack? This is important because the linked list has certain operations that are efficient and certain operations that are inefficient when the size of the linked list increases.

Therefore, we need to choose a mapping where adding an item to the stack is as efficient as possible.

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?)

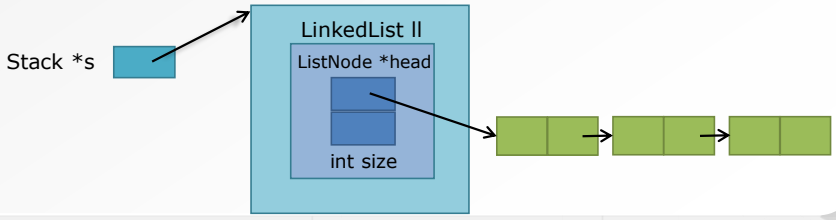


Here, I want to first try to write the push function by yourself. You are going to use the functions that you have learned so far. I have already given you a partial function prototype.

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);
}
```



Try to avoid looking at the next slide because that contains the answer. Try to figure out what parameters you need and how build the code so that it can add a new item to the stack.

If you are wondering what is stack *S, remember we are passing the stack structure by reference. Which means we have a pointer so that the actual stack can be accessed from inside of the push function.

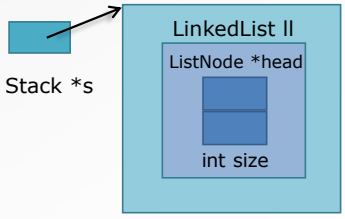
Now, the first question is, what corresponds with the top of the stack? Is it the first node or the last node of the linked list? We need to avoid traversing the linked list all the way to the end. Therefore, it is obvious that the first node corresponds with the top of the stack. Every time we access the stack, we access the top most position of the stack. Therefore, now all we need to do is get the head pointer which takes us to the first node every single time.

If you chose the opposite direction where the last node corresponding with the top of the stack, every time we access the top of the stack, we need to jump to the very end of the linked list, which is not efficient.

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

Pop function is a bit more complicated. Here, we use the remove node function. The only difference from the push function is that we need to return a value.

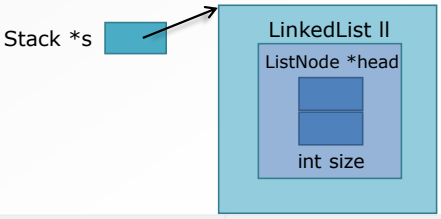
We actually need the removing value to be stored unless somebody else can figure out a way to simultaneously remove and throw it back. Because the remove node function does not give back the value and that is one of the deficiencies of that function.

So, we will store it inside a temporary integer variable and return it. One way to do this is rewrite the remove node function and return the value of the node as we remove the node. But, we are not going to do that.

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;
}
```



Peek function is also a really simple code. All we have to do is to get access to the head pointer. S.ll gets us to the actual struct, not the pointer of the struct. Therefore, using the dot notation, we can access the head pointer and follow it to get the item.

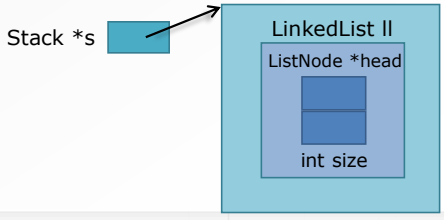
All of these parentheses are not strictly necessary. I will leave it for you to mess around by deleting them and figure out which one is necessary.

Be careful of when you use the dot notation and when you use the arrow notation, depends on whether it's a pointer to a struct or the actual struct itself.

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;
}
```



Empty stack is really trivial. Here, we will use the built in size variable and check the value of the size variable.

WORKING EXAMPLES: APPLICATIONS

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

Let me show you this worked example.

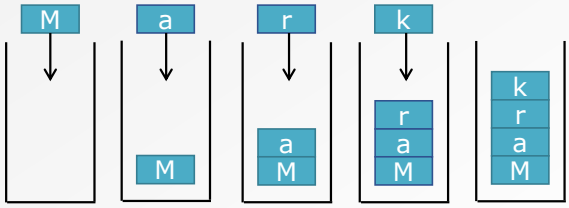
SIMPLE APPLICATION #1: REVERSE STRING

- 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

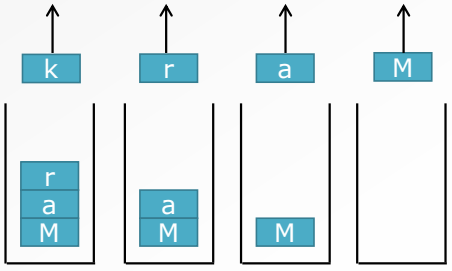
Stacks are very useful for reversing lists of items. Let's say we have a string and we want to reverse the string. We will use the first in last out property of the stacks. So, we will push one by one to the stack and when we are done, we will pop them off. Likewise, the end letter comes out first, so we will get the string or the list of items in reverse order from the origin order.

SIMPLE APPLICATION #1: REVERSE STRING

- Step 1
  - Push onto stack until no more letters



- Step 2
  - Pop from stack until stack is empty



Let's say the original string is **M. A. R. K.** In the images you see how the contents build up when we push them into the stack. But, because of the first in last out property, it returns the original string in reverse. Now, why can't we just take the existing character array and start at the end of the string and work our way backwards?

Why can't we write a for loop that goes from n to 0 and we use i--? We cannot do this because you may have to do this procedure not knowing the exact size of the string.

Therefore, what we have to do is keep buffering in all the inputs until the string is done, and then start reversing it.

So, the stack structure allows you to deal with the uncertainty. We do not have to worry about the number of characters up to some extent because the memory blocks are allocated in heap. Therefore, we read them when we need.

At the end of the input, we start to return the input.

SIMPLE APPLICATION #2: REVERSE LIST OF INTEGERS

```
1  int main(){
2      int i = 0;
3      Stack s;
4      s.ll.head = NULL;
5
6      printf("Enter a number: ");
7      scanf("%d", &i);
8      while (i != -1){
9          push(&s, i);
10         printf("Enter a number: ");
11         scanf("%d", &i);
12     }
13     printf("Popping stack: ");
14     while (!isEmptyStack(&s))
15         printf("%d ", pop(&s));
16     return 0;
17 }
```

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


The second example will leave to you to look at. It is just like the previous example but with integers this time.

QUEUE DATA STRUCTURE

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

Today we learned about stacks and its functions. The next day we'll talk about the queue data structures.

QUEUE DATA STRUCTURE

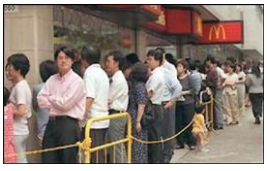
- **Queue data structure**
- Queue implementation using linked lists
- Queue functions
  - enqueue()
  - dequeue()
  - peek()
  - isEmptyQueue()
- Worked examples: Applications

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 DATA STRUCTURE

- **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;
```

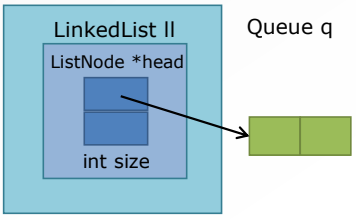
QUEUE IMPLEMENTATION USING LINKED LISTS

- 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

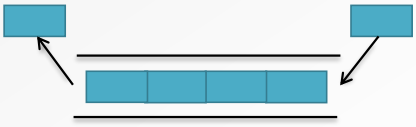


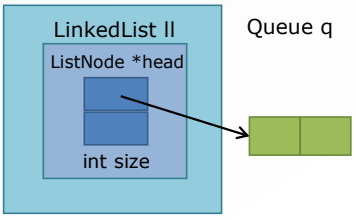
TODAY

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

QUEUE FUNCTIONS: enqueue()

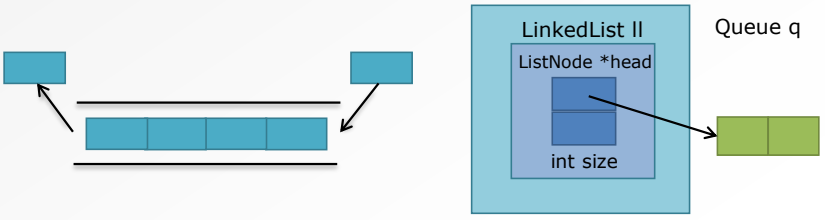
- enqueue() function is the only way to add an element to the queue data structure
- Only allowed to enqueue() at the end
- Question:
  - Using a linked list as the underlying data storage, does the first linked list node represent the front or the back of the queue?
  - Figure out which option makes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





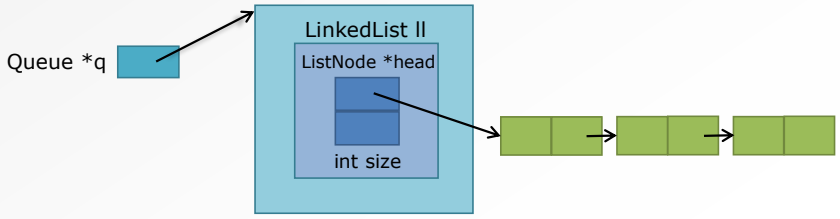
QUEUE FUNCTIONS: enqueue()

- **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 back 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

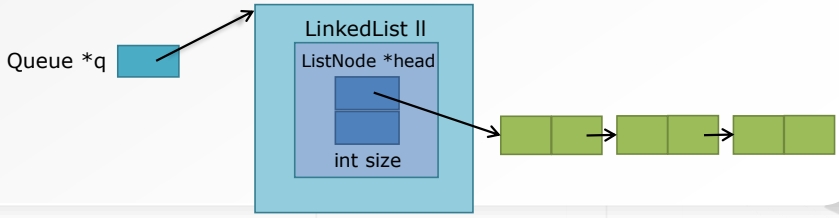
```
void enqueue(Queue *q, int item){
    insertNode(&(q->ll), q->ll.size, item);
}
```

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

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

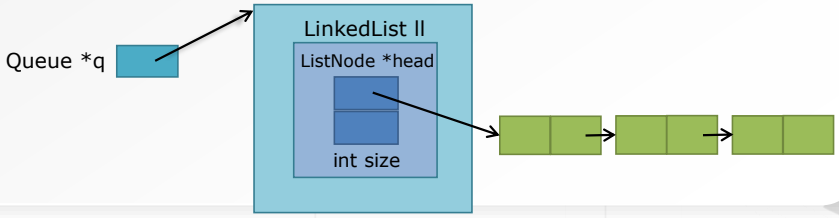
```
int dequeue(Queue *q){
    int item;
    item = ((q->ll).head)->item;
    removeNode(&ll, 0);
    return item;
}
```



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

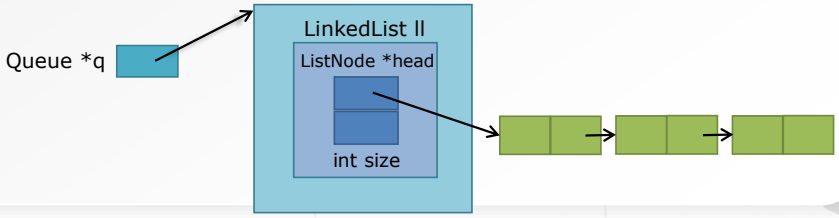
```
int peek(Queue *q){
    return ((q->ll).head)->item;
}
```



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

```
1   int main(){
2       Queue q;
3       q.ll.head = NULL;
4       q.ll.tail = NULL;
5
6       enqueue(&q, 1);
7       enqueue(&q, 2);
8       enqueue(&q, 3);
9       enqueue(&q, 4);
10      enqueue(&q, 5);
11      enqueue(&q, 6);
12
13      while (!isEmptyQueue(&q))
14          printf("%d ", dequeue(&q));
15  }
```

```
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