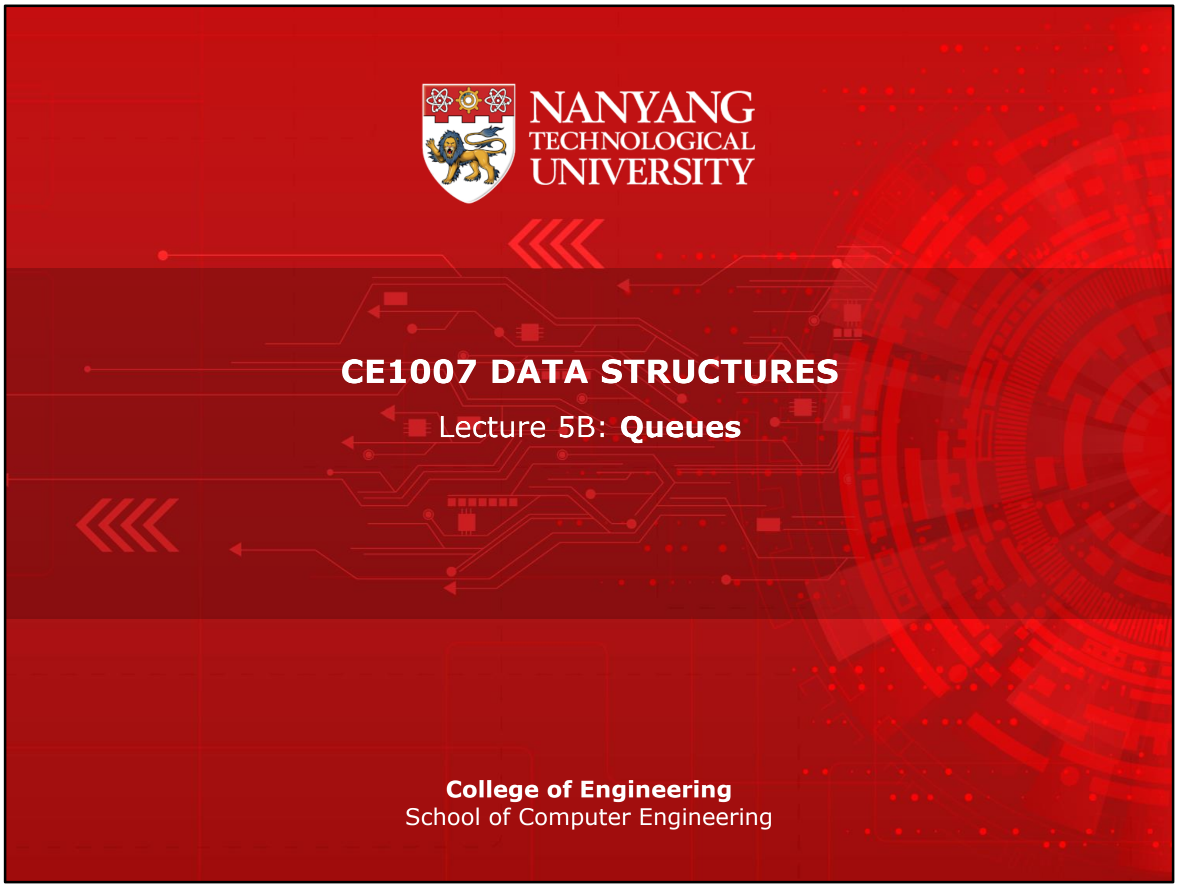


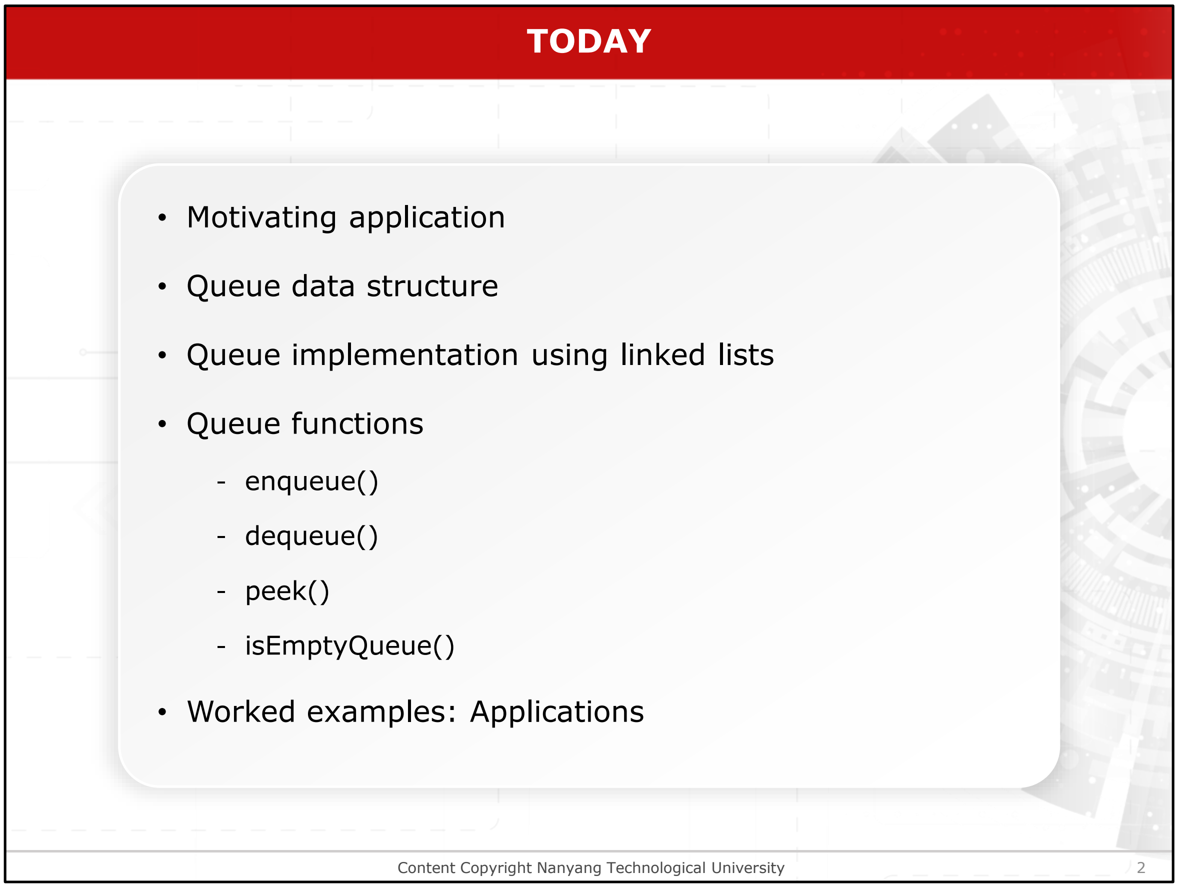
NANYANG TECHNOLOGICAL UNIVERSITY

CE1007 DATA STRUCTURES

Lecture 5B: **Queues**

College of Engineering
School of Computer Engineering

This lecture is on Queue data structure. Queues are not much different from Stacks.

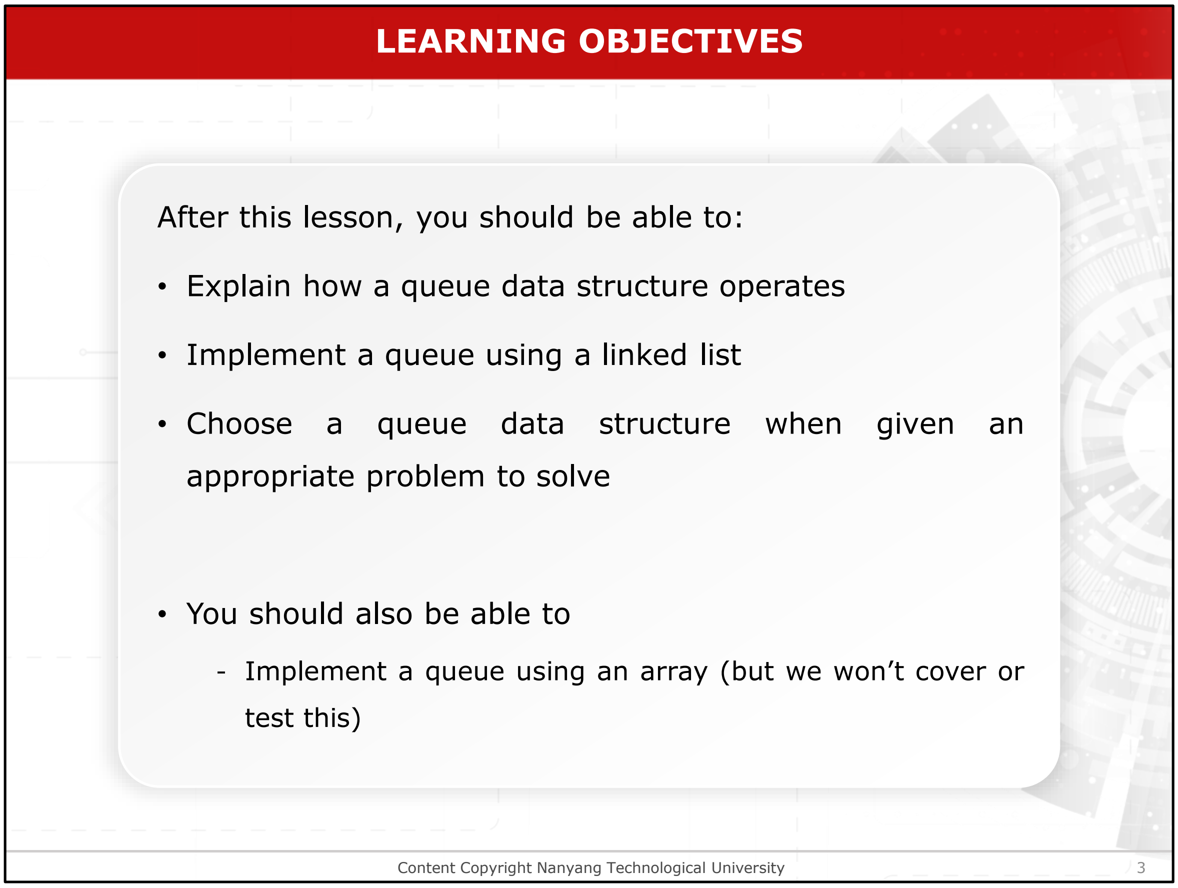


TODAY

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

2

We will discuss about queue data structures, its core functions and their implementation using linked lists. All are very similar to what we have learned in Stacks but for adding and removing nodes we have different names as enqueue and dequeue.



LEARNING OBJECTIVES

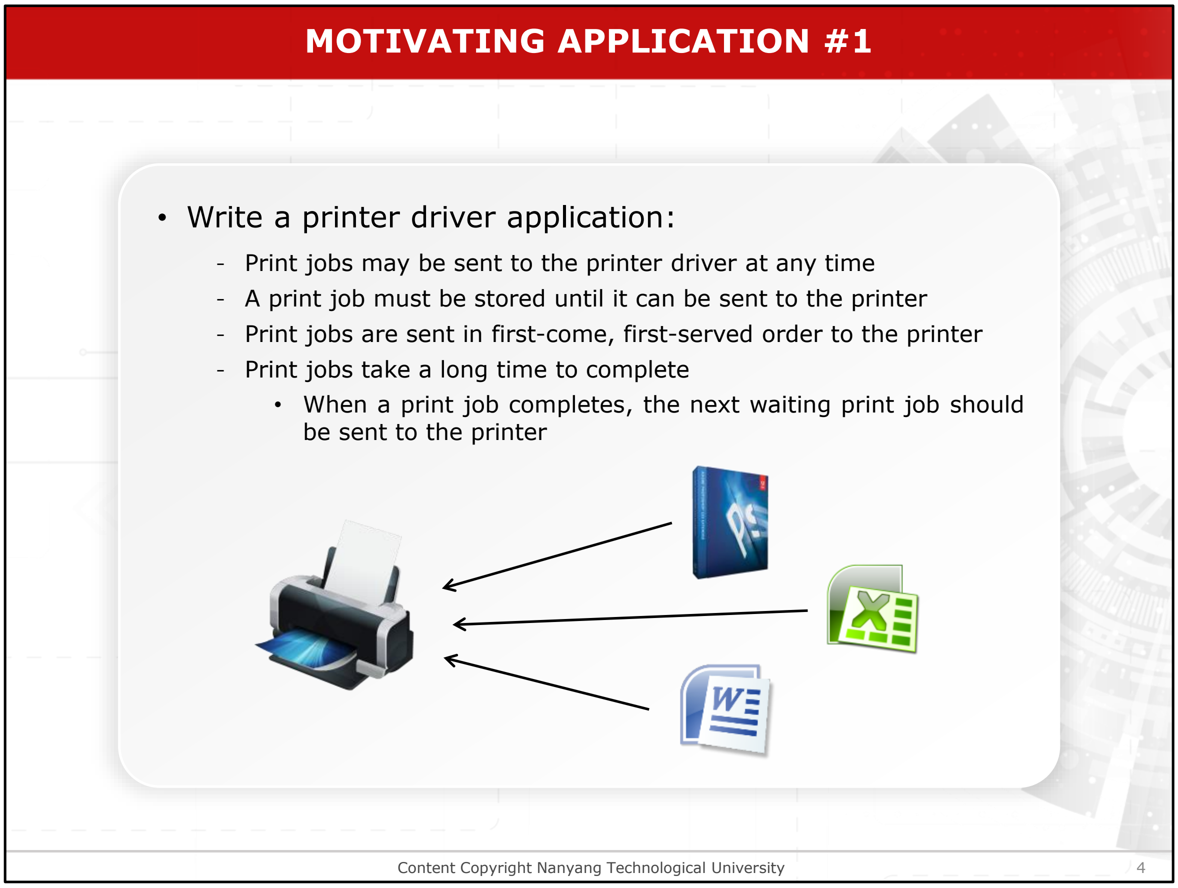
After this lesson, you should be able to:

- Explain how a queue data structure operates
- Implement a queue using a linked list
- Choose a queue data structure when given an appropriate problem to solve
- You should also be able to
  - Implement a queue using an array (but we won't cover or test this)

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

Explain how a queue works
Why it is different from stacks

How the core functions implement in C using linked lists



MOTIVATING APPLICATION #1

- Write a printer driver application:
  - Print jobs may be sent to the printer driver at any time
  - A print job must be stored until it can be sent to the printer
  - Print jobs are sent in first-come, first-served order to the printer
  - Print jobs take a long time to complete
    - When a print job completes, the next waiting print job should be sent to the printer

Content Copyright Nanyang Technological University 4

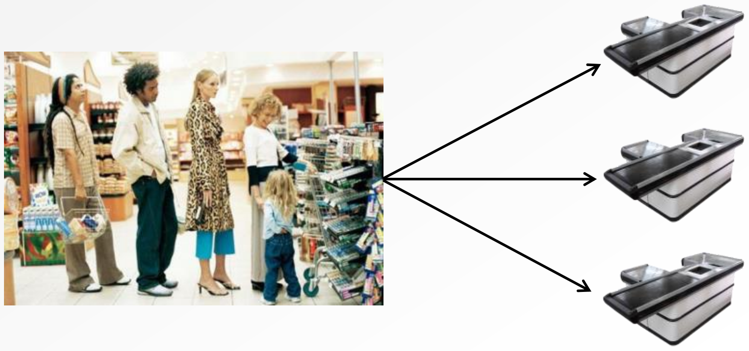
Consider a scenario where you have a printer which is shared by multiple applications on your computer. So, all the applications in your computer are linked with this printer and therefore, you can print only one at a time.

The printing order is a first come first serve order. For example, at first you send a Word document, then send an Excel document and finally send a Powerpoint document. Then the at first, the printer prints the Word document, then the Excel document and finally the Powerpoint document. This is what we call a first in first out data structure. This is different from a Stack because a stack has a first in last out data structure.

In queues, if the first task takes a while to complete, the following tasks also will be held in the queue, waiting for their turn.

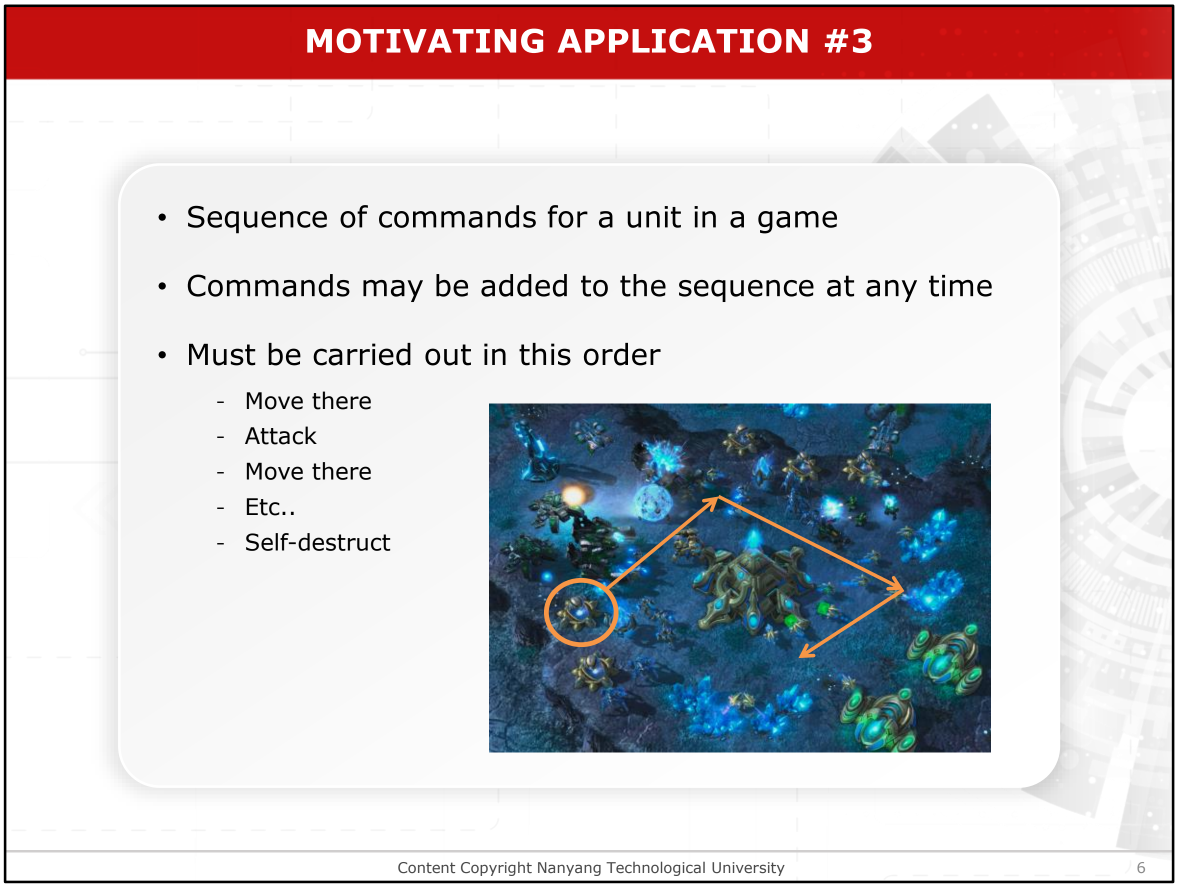
MOTIVATING APPLICATION #2

- Supermarket checkout counter assignment
  - 1 checkout counter OR N checkout counters
  - Single queue of customers
  - First-come, first-served bases
    - Join the back of the queue and wait for your turn



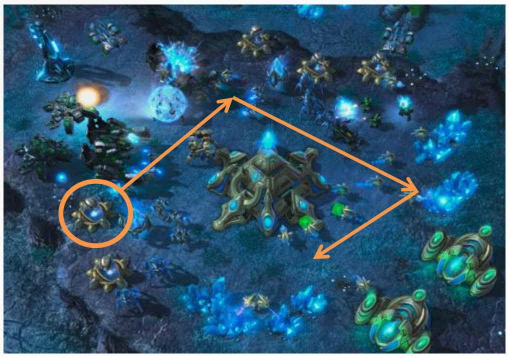
In the previous example, the tasks are coming from different sources and running into one printer. Now, imagine an a queue in a supermarket. People are being queued to different checkout counters.

The first person in the queue will be served first. The queueing allows to make sure that serving is done fairly.



MOTIVATING APPLICATION #3

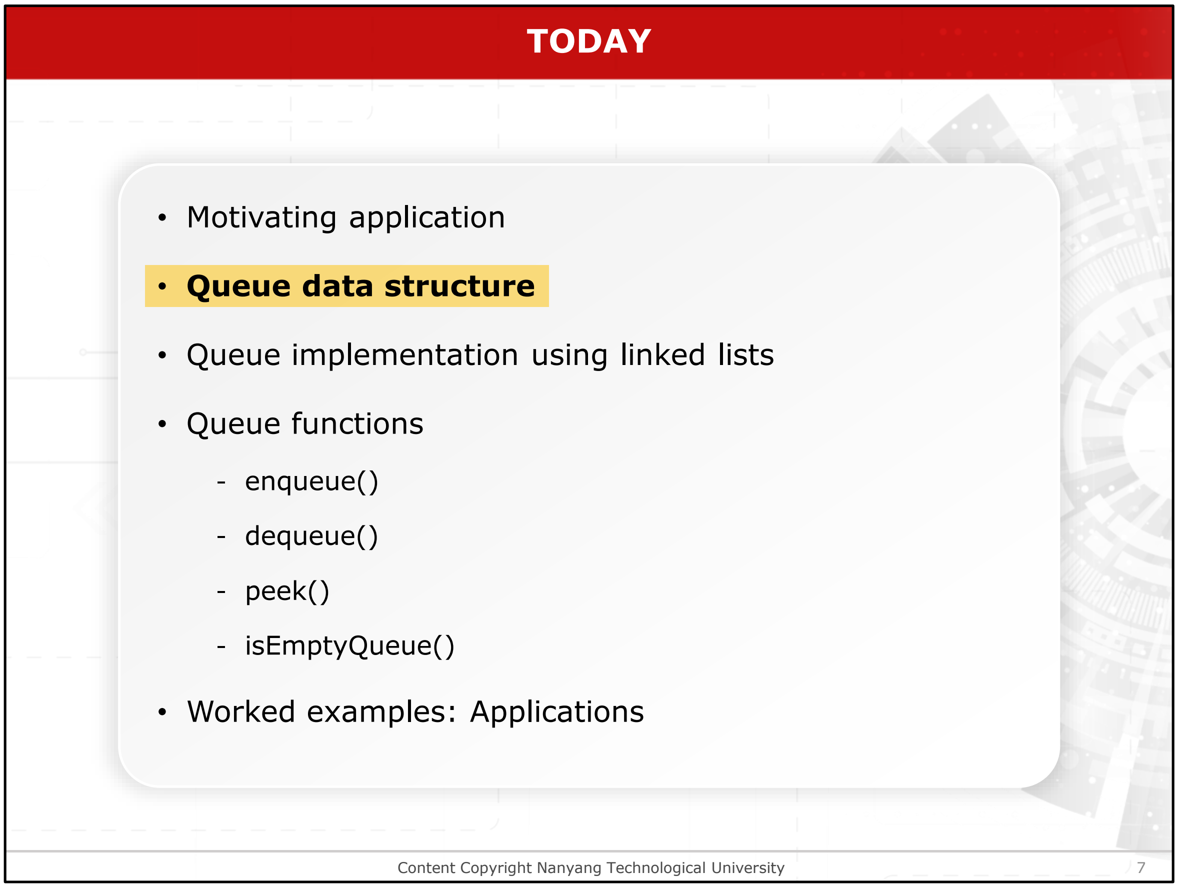
- Sequence of commands for a unit in a game
- Commands may be added to the sequence at any time
- Must be carried out in this order
  - Move there
  - Attack
  - Move there
  - Etc..
  - Self-destruct



6

If you are playing a game, you have a sequence of commands such as move a unit from position X to position Y, attack a unit, build something, etc. You obviously don't want the sequence of commands to be overridden once you add a new command to the list. They should be executed in that order.

This is where you need a queue. The first item in should be the first command to process.

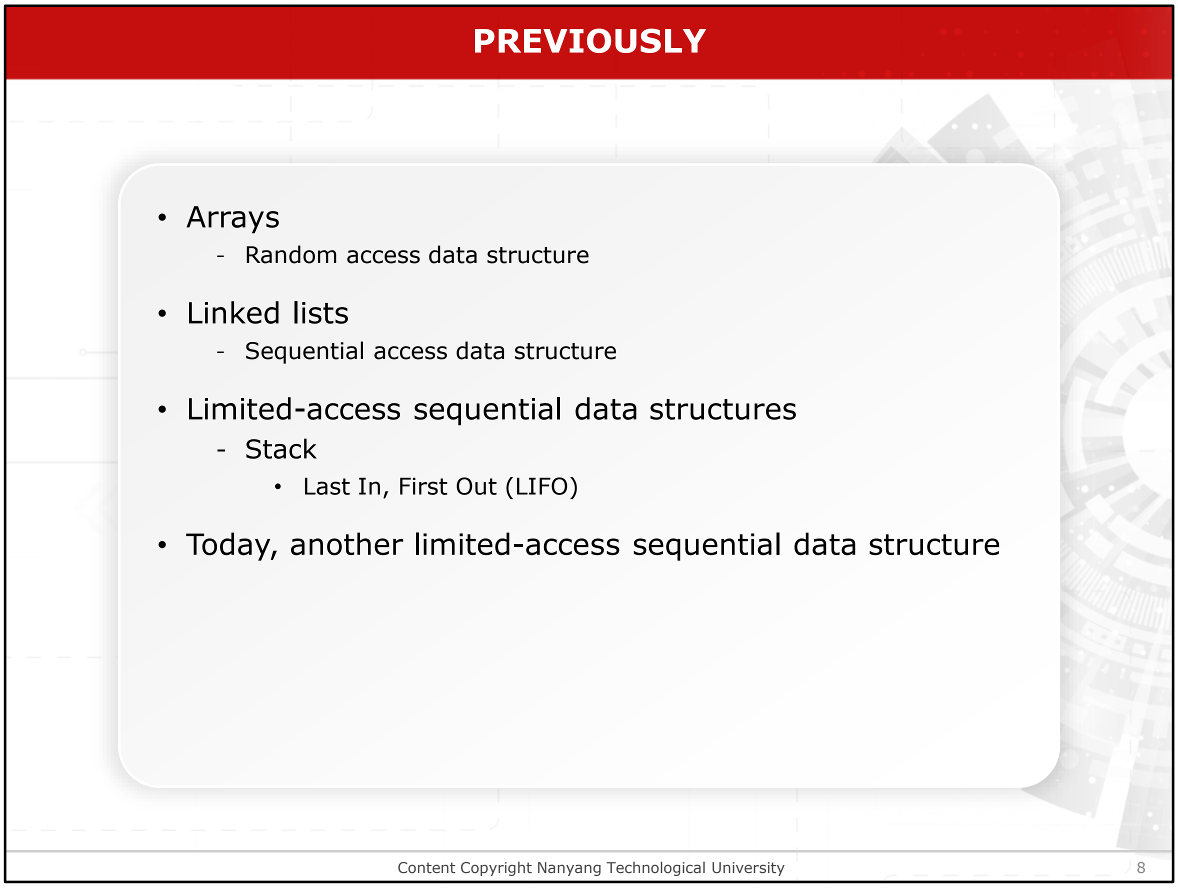


TODAY

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

7

Let’s discuss about Queue data structure.

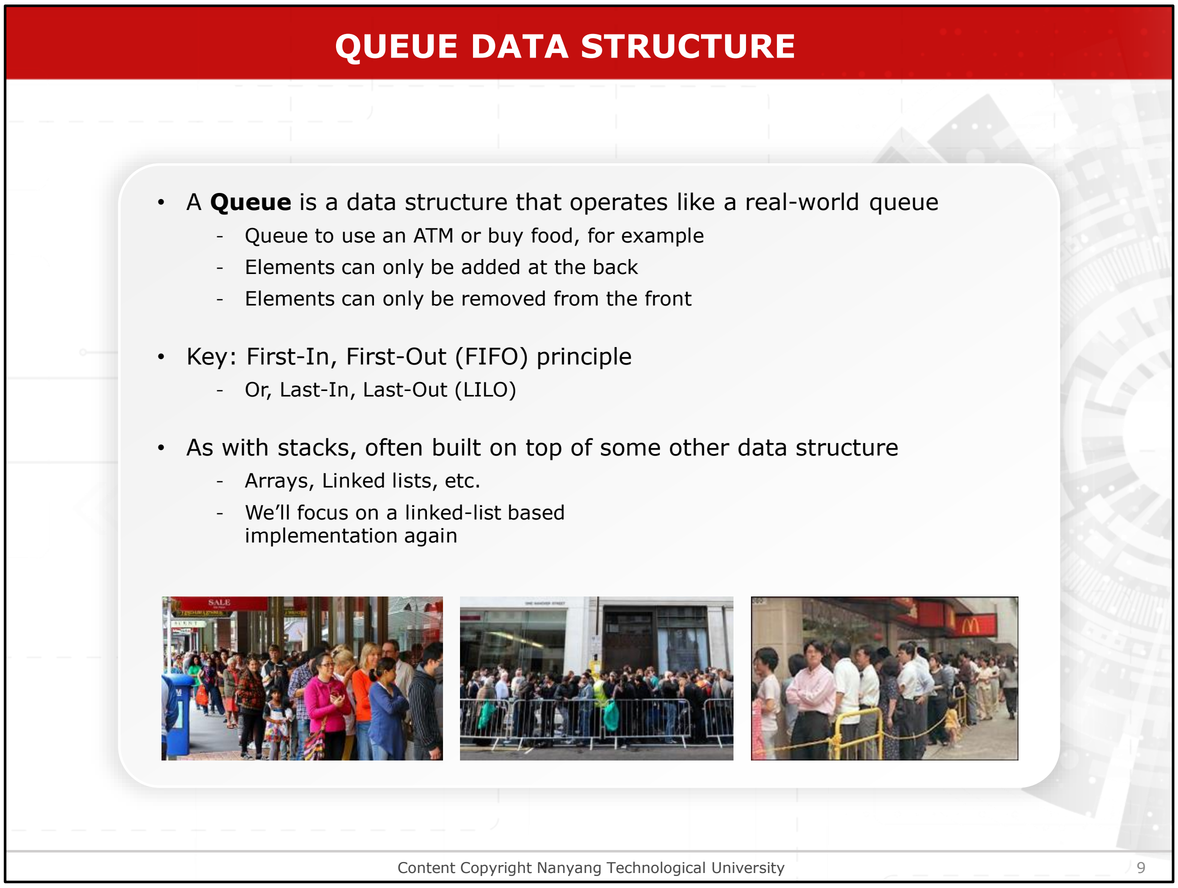


PREVIOUSLY

- 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

8

In previous lectures, we learned that arrays are random access data structures. Once you have the index position, you can access any array element. Linked list is a sequential access data structure. We need to traverse from the first node.



QUEUE DATA STRUCTURE

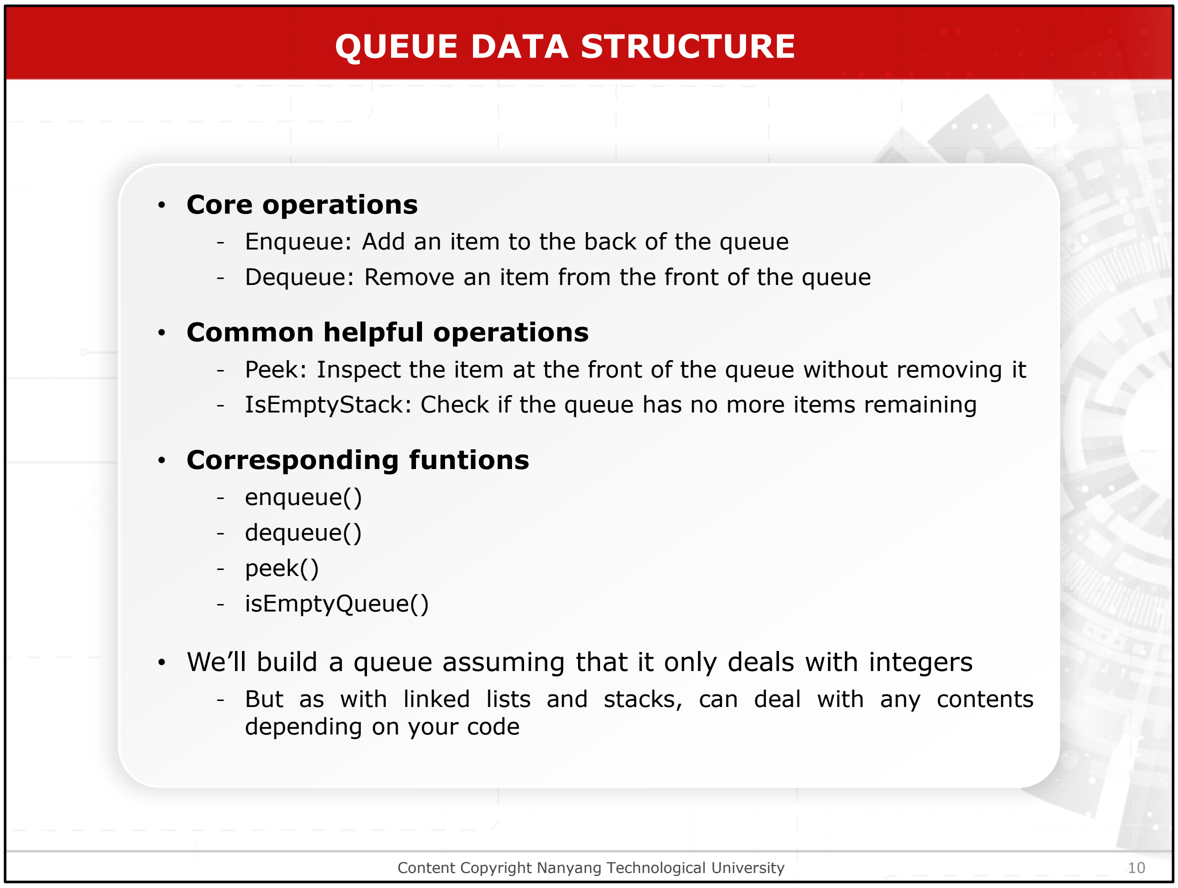
- A **Queue** is a data structure that operates like a real-world queue
  - Queue to use an ATM or buy food, for example
  - Elements can only be added at the back
  - Elements can only be removed from the front
- Key: First-In, First-Out (FIFO) principle
  - Or, Last-In, Last-Out (LILO)
- As with stacks, often built on top of some other data structure
  - Arrays, Linked lists, etc.
  - We'll focus on a linked-list based implementation again

9

Limited access sequential data structures are like taking linked list and constraining it some more such allowing only access the front. This is stack data structures which is last in first out or first in last out. In this lecture we discuss about queue data structures which is first in first out or last in last out.

We add elements at the back of the queue and remove the elements from the front. This is what gives us the first in first out order.

We can build the queue structure just like we built the stacks. We will use the linked lists because we do not want to focus on the management of the nodes.



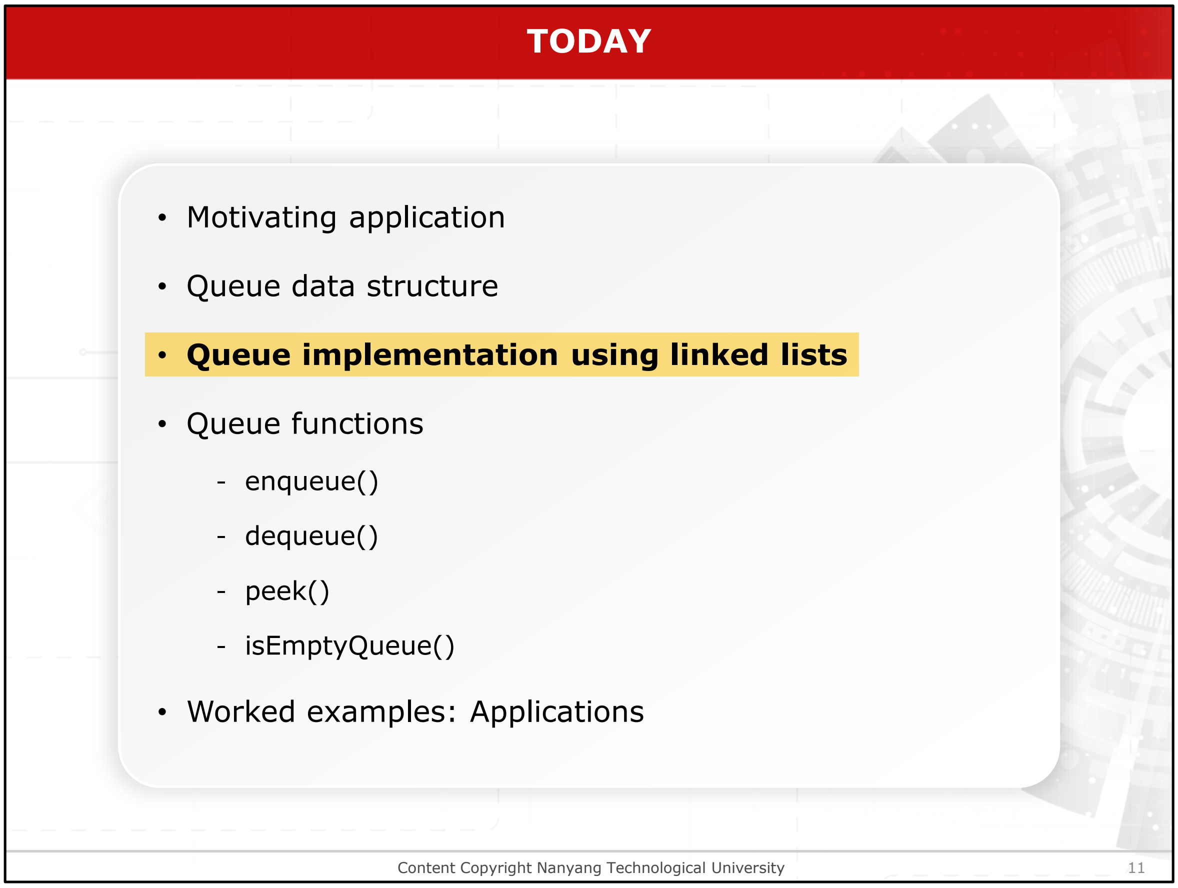
QUEUE DATA STRUCTURE

- **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

10

The core operations are enqueue and dequeue. Enqueue adds items to the back of the queue and dequeue allows you to remove items from the front. The peek function allows to look at the first item before we remove it and isEmptyQueue function determines if there is anything left inside the queue.

We will build these four functions. Only the enqueue and dequeue functions are going to be completely different.

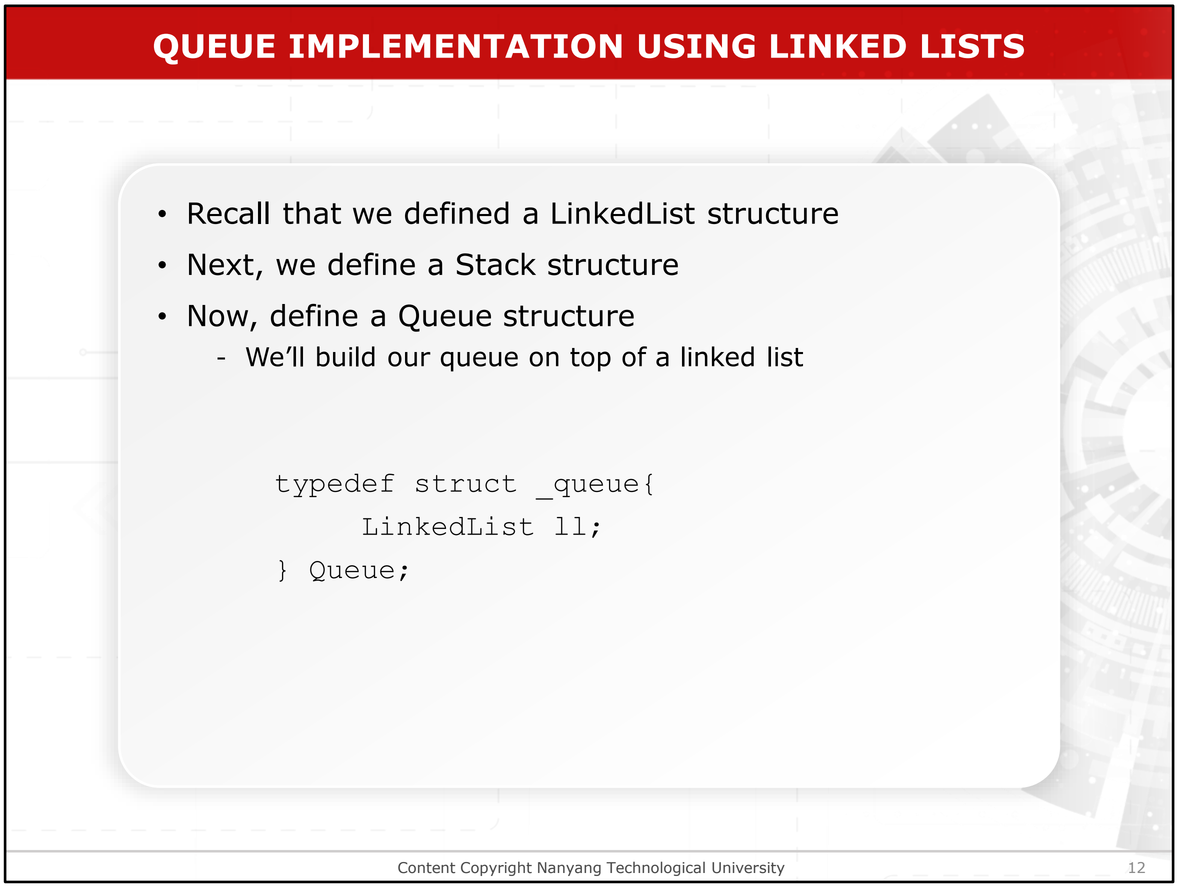


TODAY

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

11

As in the stack, we define queue with a linked list struct.

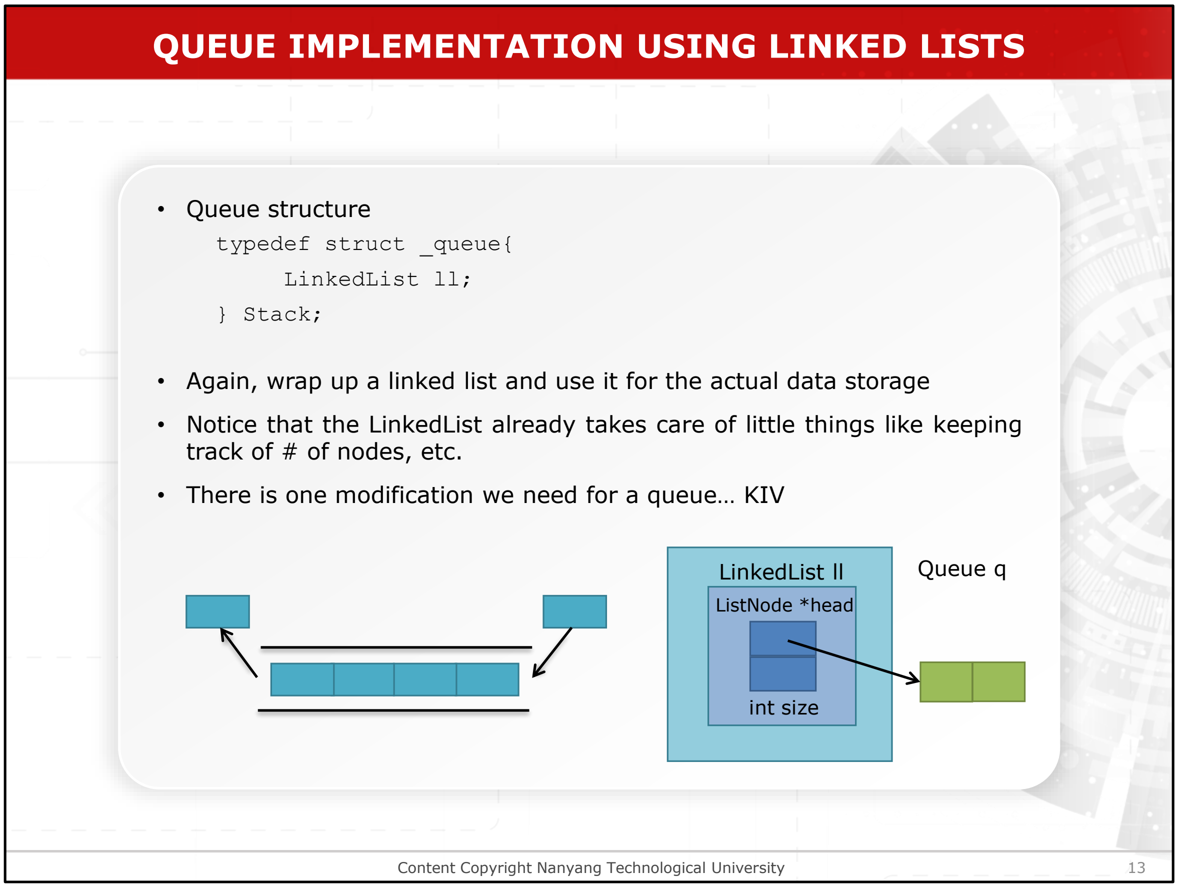


QUEUE IMPLEMENTATION USING LINKED LISTS

- 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;
```

So, do we want the front of the queue to correspond to the first node of the linked list? Or, do we need the back of the queue to correspond to the first node of the linked list? When we asked these questions when we learned stack, the answer was that it doesn't matter.

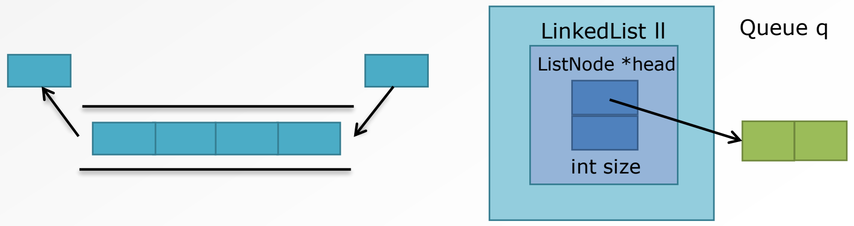


QUEUE IMPLEMENTATION USING LINKED LISTS

- Queue structure

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

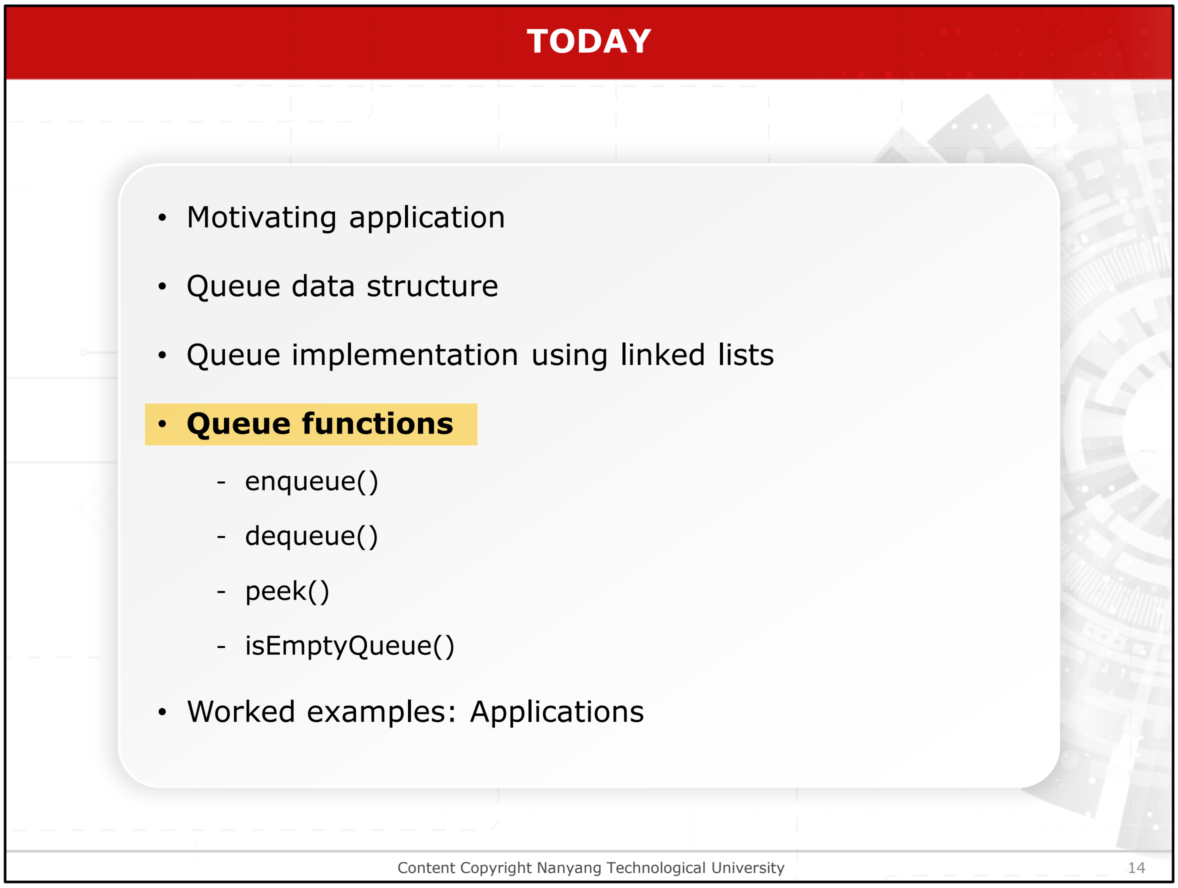
- Again, wrap up a linked list and use it for the actual data storage
- Notice that the LinkedList already takes care of little things like keeping track of # of nodes, etc.
- There is one modification we need for a queue… KIV



13

Now, is it same with queue data structure? Actually, it shouldn't matter if the linked list has a tail pointer. So, the implementation of queue using linked list is different from stacks because here we use the tail pointer to access the both ends of the structure. With stack, we only access items at one position.

Therefore, with queue data structure, to have it properly implemented and running efficiently, I actually need to make sure that my linked list has a tail pointer inside.

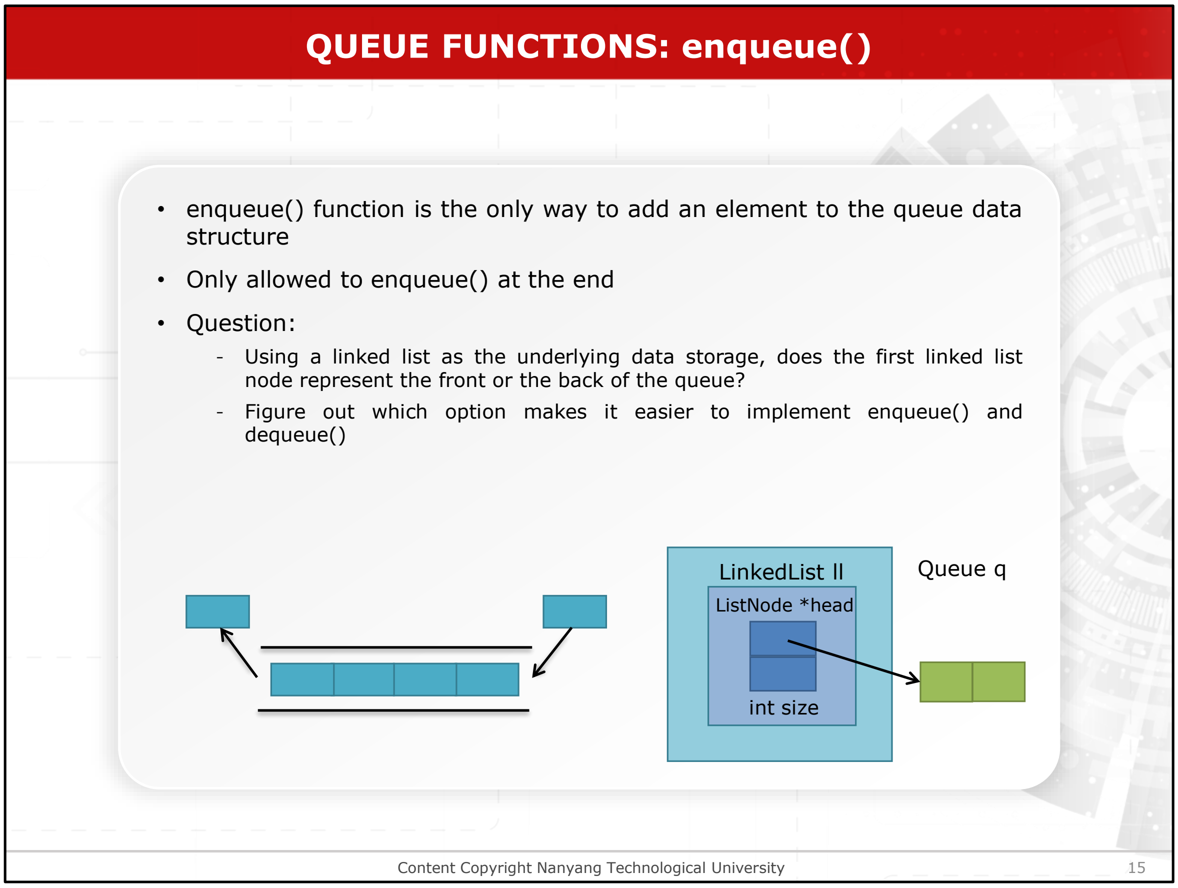


TODAY

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

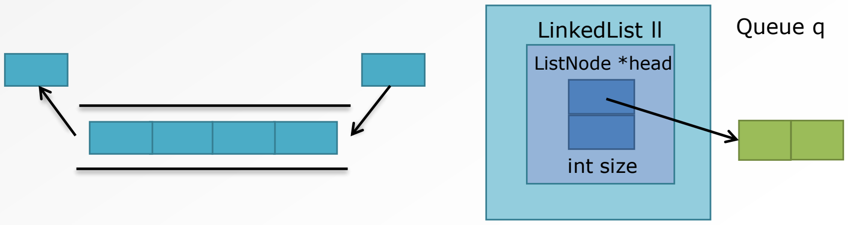
14

Let's look at the enqueue and dequeue functions in detail. The enqueue function supposed to add items to the back of the linked list and the dequeuer functions supposed to remove items from the front of the linked list.

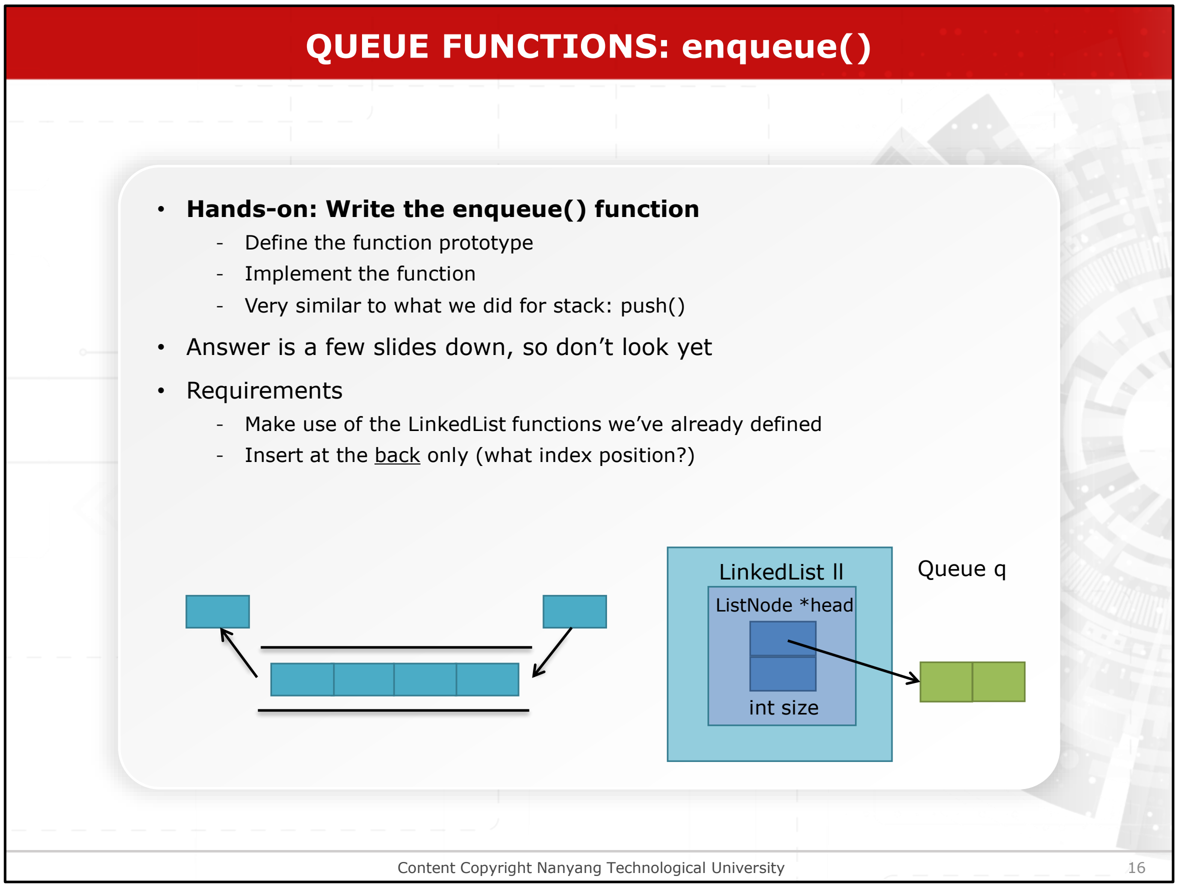


QUEUE FUNCTIONS: enqueue()

- 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()

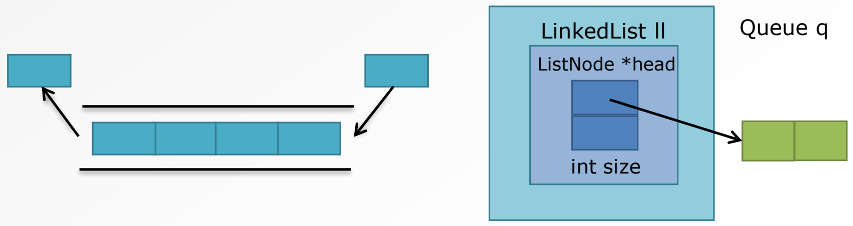


So if you think about which index positions you are going to be accessing, you will realize that you need to access either position zero and the last index, or if you flip it around, it's still going to be position zero and the last index position.



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?)



We will implement the enqueue function now.

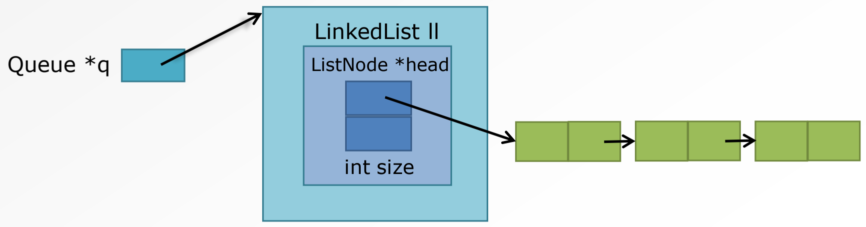
QUEUE FUNCTIONS: enqueue()

- **Hands-on: Write the enqueue() function**
  - Before looking at the code on the next slide, try writing the code for yourself

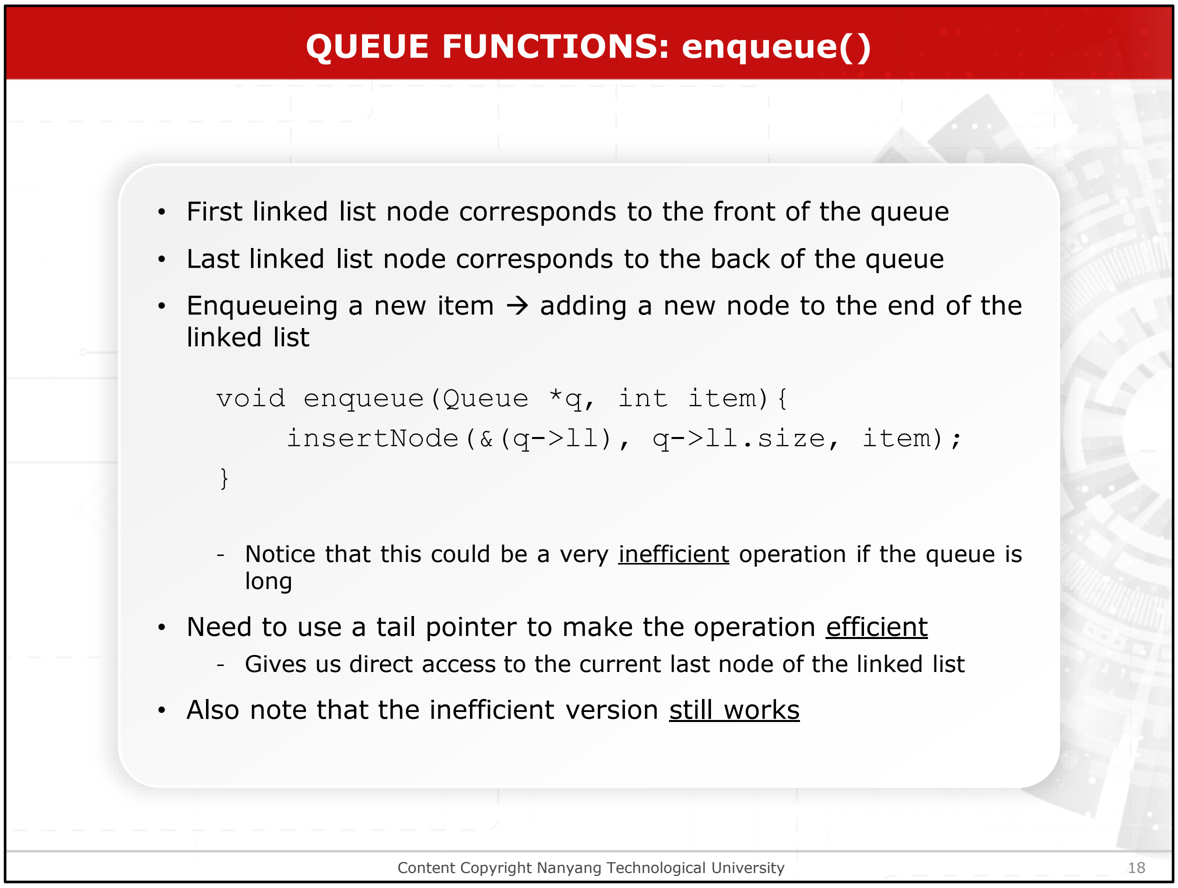
```
void enqueue(Queue *q,                        ){



}
```



The enqueue function is going to accept a pointer to the queue. Also, we need to pass the value that we need to add to the queue.



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

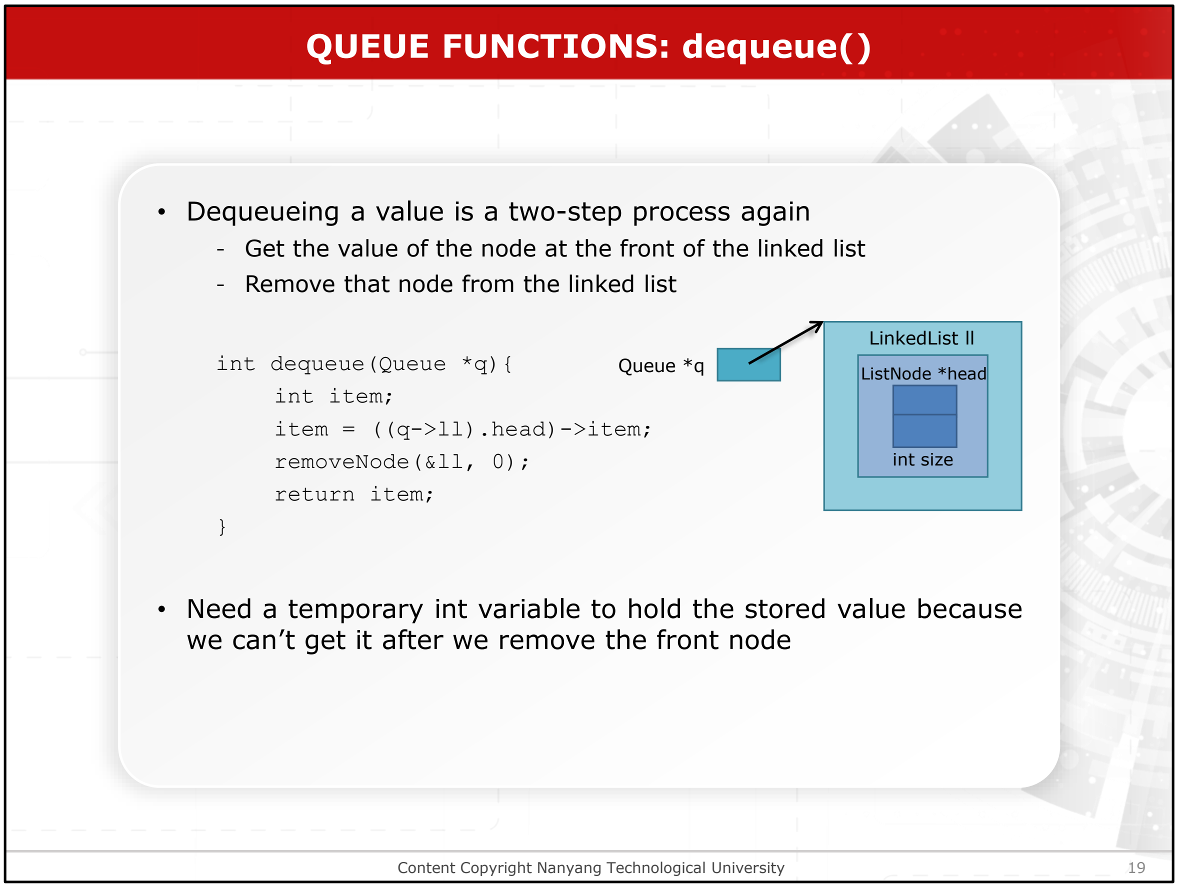
```
void enqueue(Queue *q, int item){
    insertNode(&(q->ll), q->ll.size, item);
}
```

  - 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

Content Copyright Nanyang Technological University 18

Just like the stack push function, this is a simple insert node call. For enqueue, we are going to add items to the last position of the linked list. Therefore, the correct index position is ll.size because the index position of the last item in the linked list is size-1.

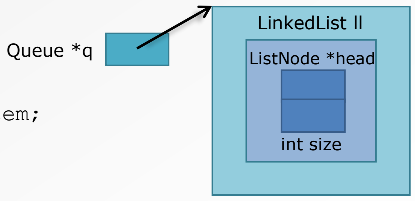
In order for this to be efficient, we need to give direct access to the last node of the linked list. So, we need a tail pointer. It will still work without a tail pointer, but then it will be very inefficient because every time we add an item, we have to start from the front and travers all the way down.



QUEUE FUNCTIONS: dequeue()

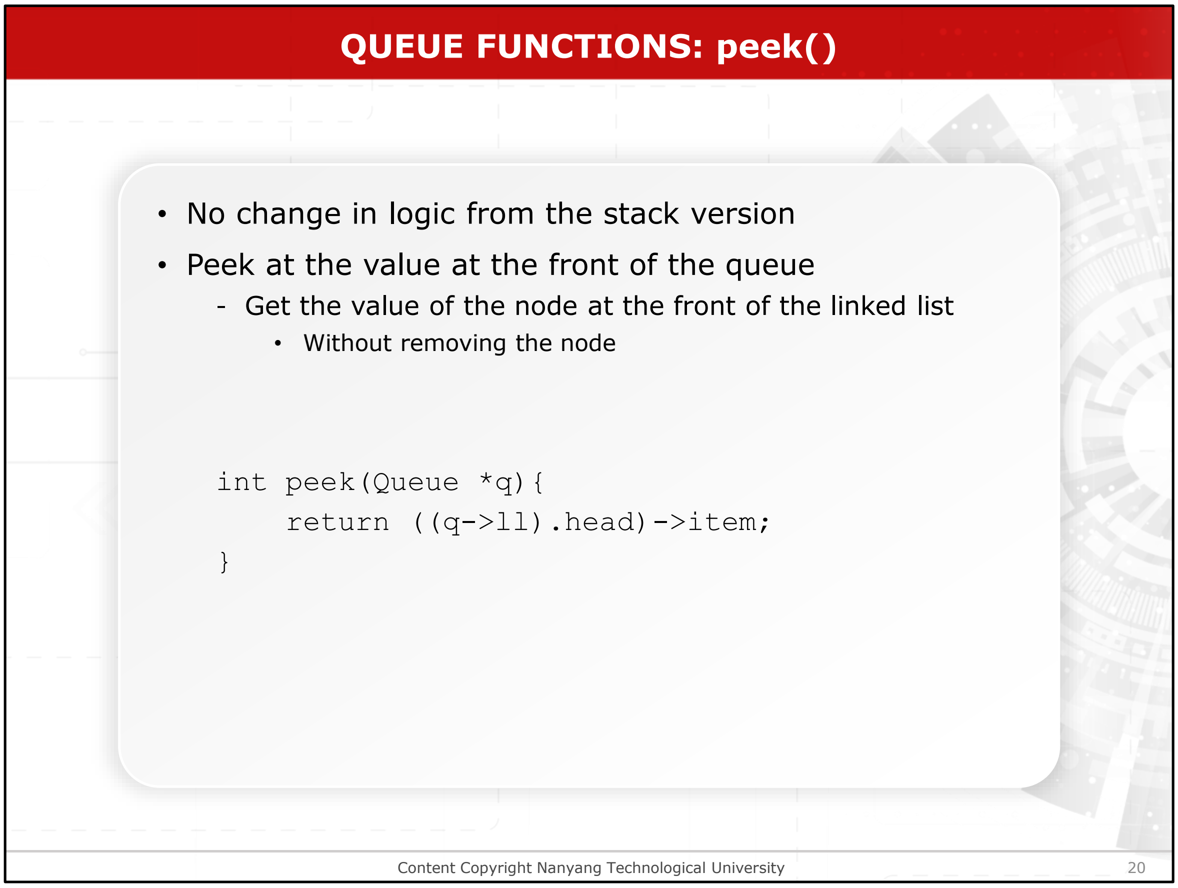
- Dequeueing a value is a two-step process again
  - Get the value of the node at the front of the linked list
  - Remove that node from the linked list

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



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

Dequeue is a two-step process because we need to hold the value before deleting the node from the linked list. So, just as in stack data structure, we call remove node from the index position 0. The only difference of this with stack data structure is that enqueue is dealing with the last position rather than the 1st position as in the stack.



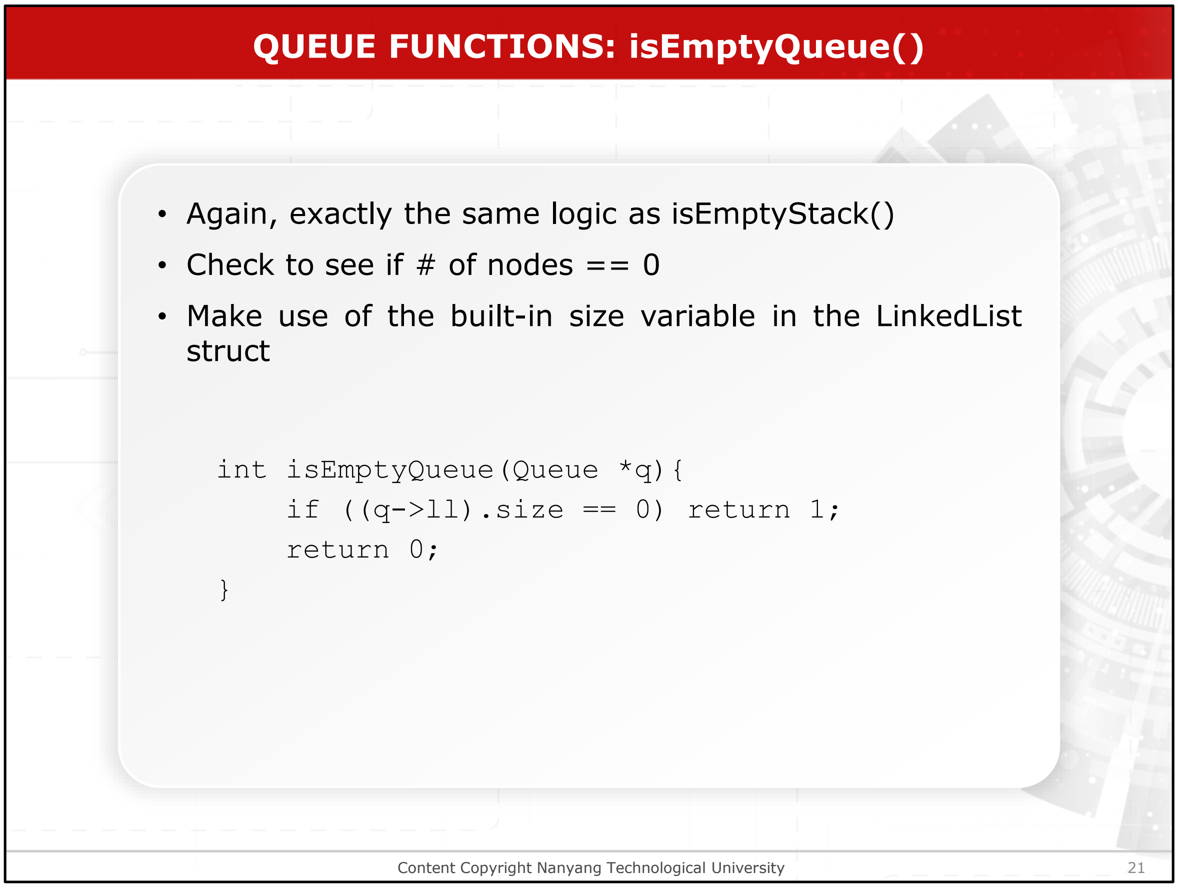
QUEUE FUNCTIONS: peek()

- 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;
}
```

20

So the peek function is also the same thing. Only difference is that I'm passing in a queue instead of a stack.



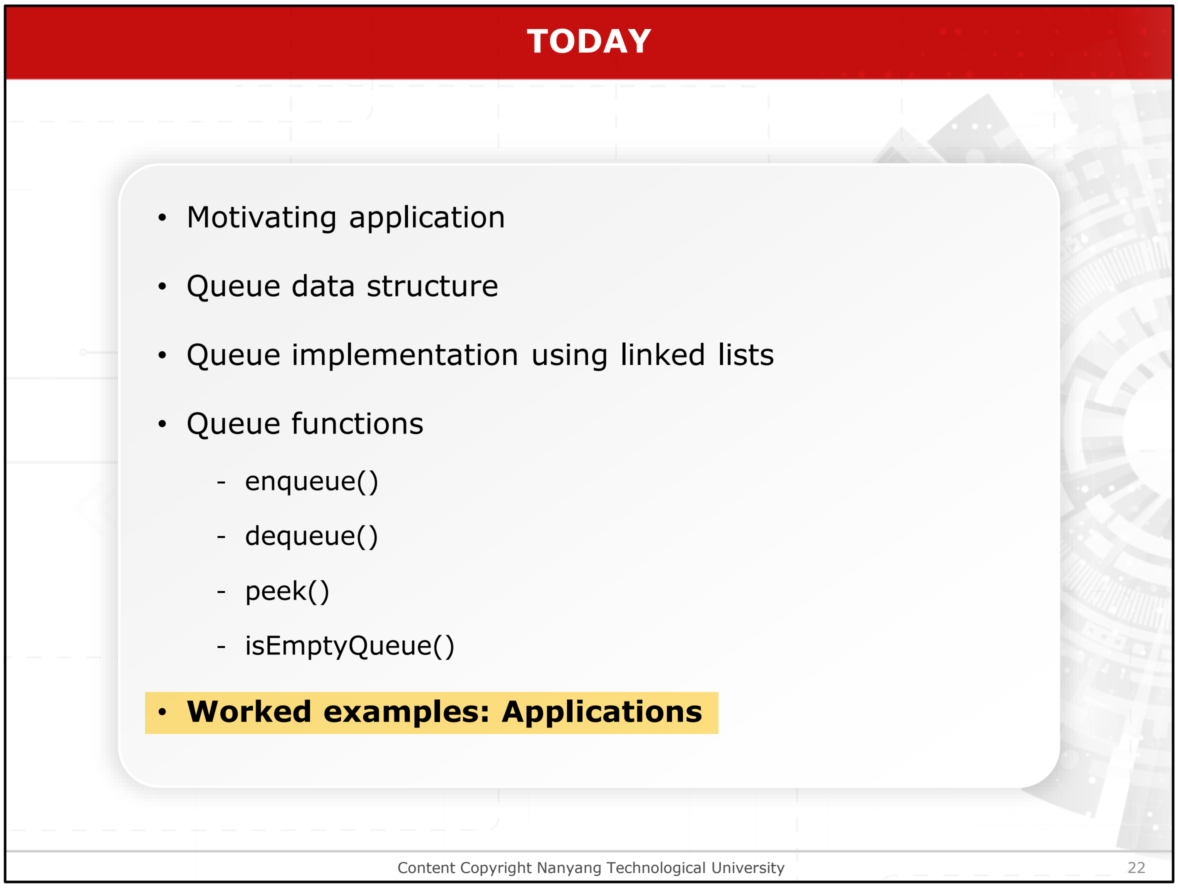
QUEUE FUNCTIONS: isEmptyQueue()

- 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;
}
```

21

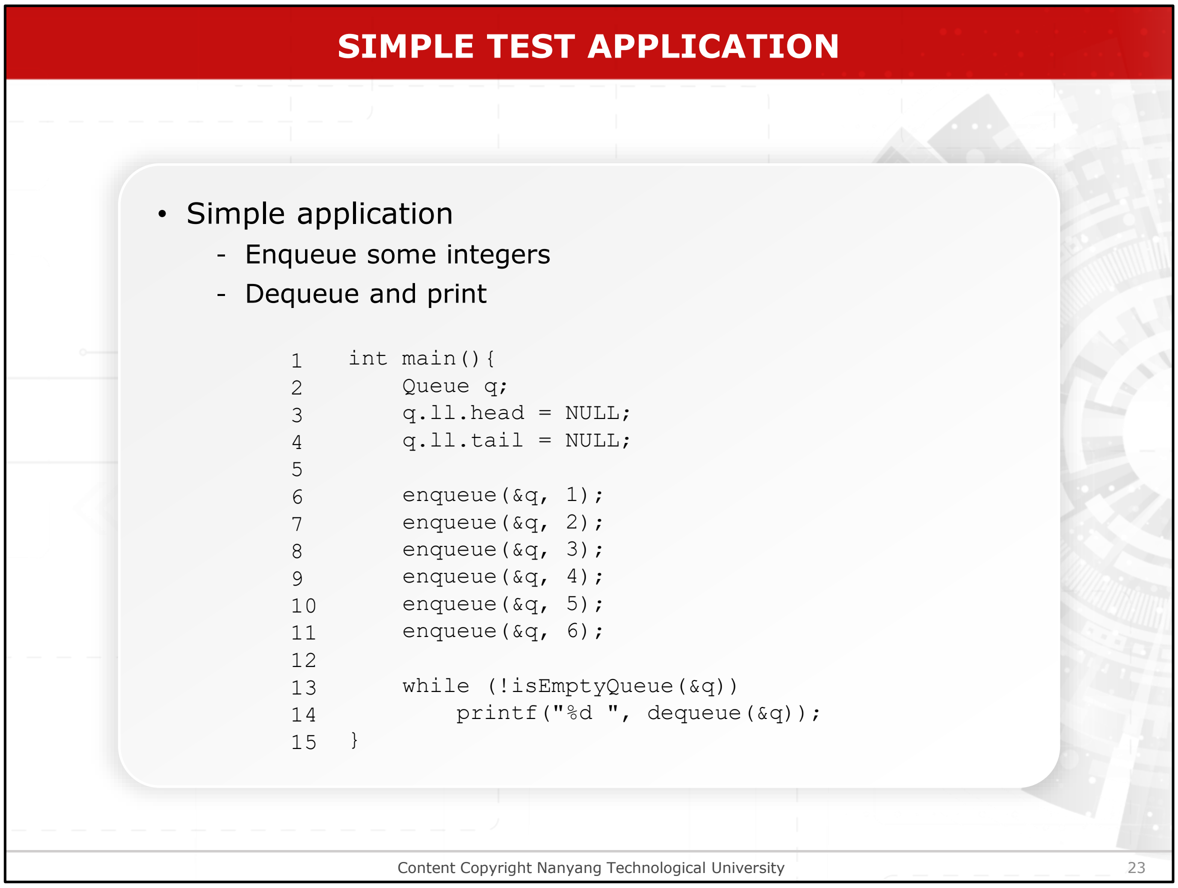
`isEmptyQueue functions` is also doing the same thing, Here, we will use the built in size variable and check the value of the size variable.



TODAY

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

Let’s see simple example that uses enqueue(), dequeue(), and isEmptyQueue() functions.



SIMPLE TEST APPLICATION

- Simple application
  - Enqueue some integers
  - Dequeue and print

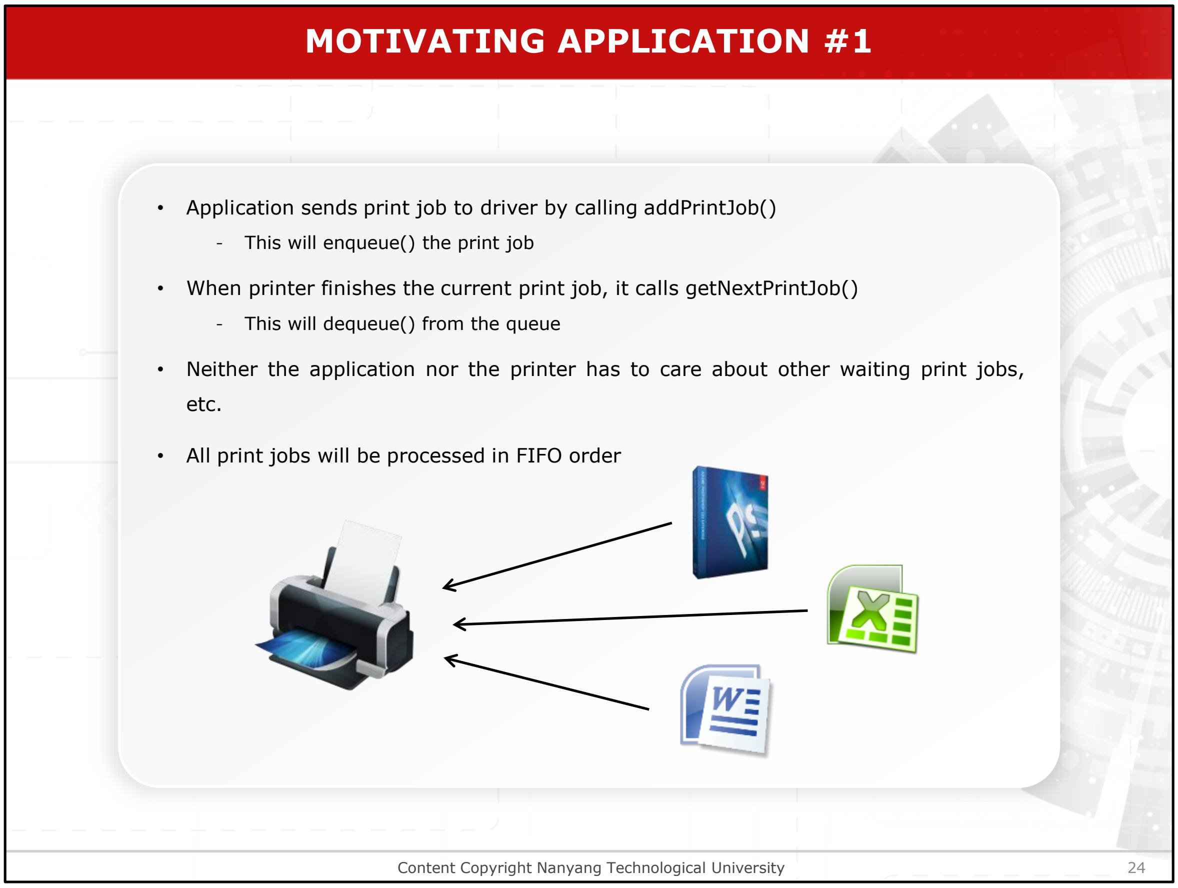
```
int main(){
    Queue q;
    q.ll.head = NULL;
    q.ll.tail = NULL;

    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));
}
```

23

To make this work properly, we need to set the head and tail pointers. To add an item, we just need to call the enqueue function and pass in the value. So, we add the values, 1, 2, 3, 4, 5 and 6. And then we print whatever is returned from the dequeue function. Because it is first in first out, when we print, it will the same order as they inserted which is 1, 2, 3, 4, 5 and 6.

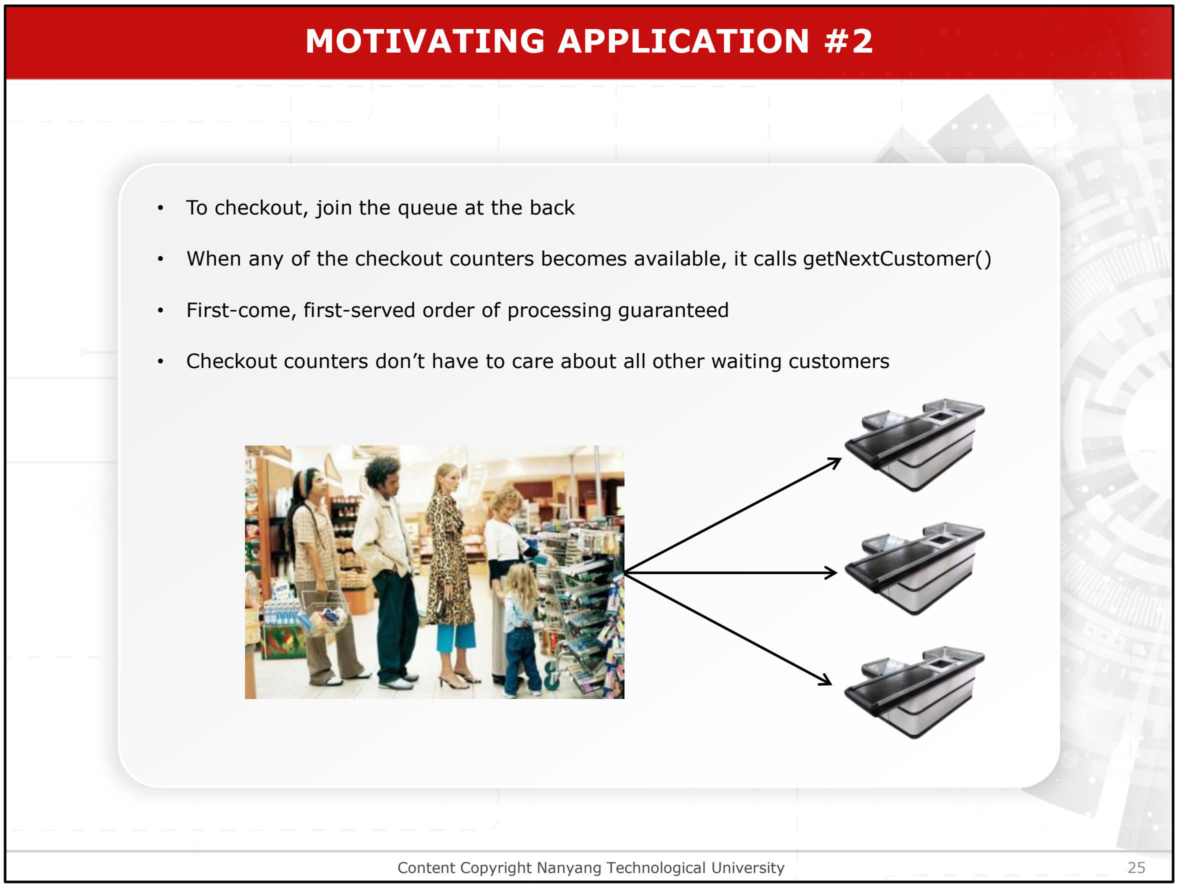


MOTIVATING APPLICATION #1

- Application sends print job to driver by calling addPrintJob()
  - This will enqueue() the print job
- When printer finishes the current print job, it calls getNextPrintJob()
  - This will dequeue() from the queue
- Neither the application nor the printer has to care about other waiting print jobs, etc.
- All print jobs will be processed in FIFO order

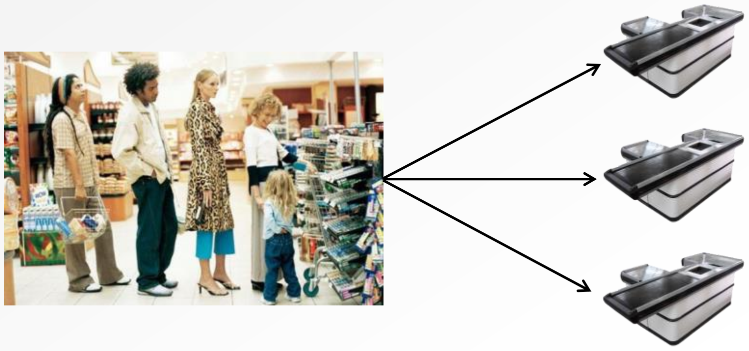
24

If we go back to this motivating application, because queue structure takes care of keeping everything in the right order, the printer does not have to care about the order. The next print job will call the dequeue and gets the next item to process.



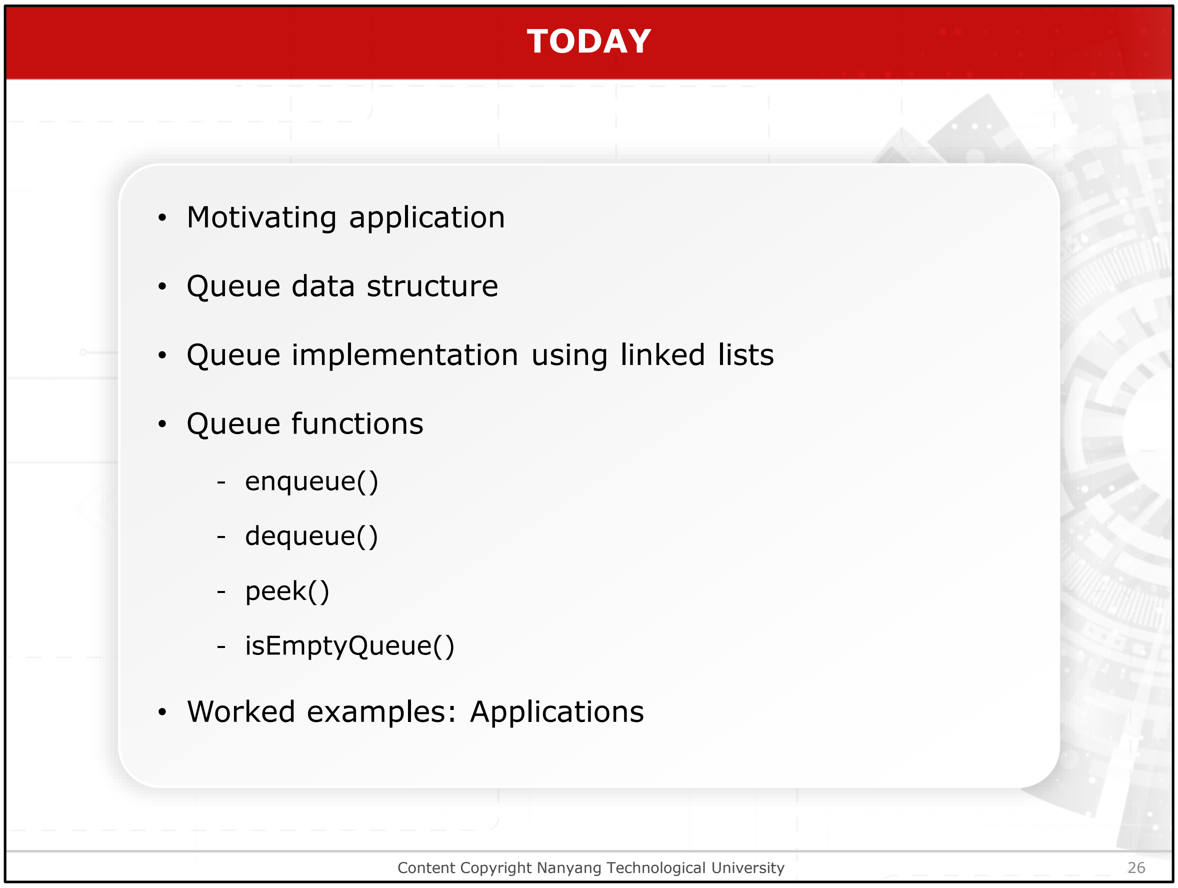
MOTIVATING APPLICATION #2

- To checkout, join the queue at the back
- When any of the checkout counters becomes available, it calls getNextCustomer()
- First-come, first-served order of processing guaranteed
- Checkout counters don’t have to care about all other waiting customers



25

And likewise over here, with a single queue of people and multiple checkout counters, the people just stay in their position in the queue, they are not allowed to move around, they are not allowed to say, person at the back come forward. And all of these individual checkout lines don't care about who is third forth fifth sixth seventh, they only care about who is next, who is at the front of the queue.



TODAY

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

Today we learned about queues and its functions.