Course: Object Oriented Programming - Abstract Data Types

The ADT Stack

Definition

The **ADT Stack** is a linear sequence of an arbitrary number of items, together with:

- access procedures that permit **insertions** and **deletion** of items only at one end of the sequence: the "**top**".
  - The stack is a **last-in-first-out** (or LIFO) sequence of elements
  - The **depth** of a stack is the number of elements it contains,
  - The **empty** stack has depth zero.

Uni3: ADT Stack

Slide Number 1

The term "stack" is intended to conjure up visions of things encountered in daily life, such as a stack of dishes in a college cafeteria or a stack of books on your desk. In common English usage, "stack of" and "pile of" are synonymous. To computer scientists, however, a stack is not just any old pile. A stack has the property that the last element placed on the stack will be the first item to be removed. This property is commonly referred to as "last-in-first-out" or simply **LIFO**. The last element placed in the stack is called the "**top**" element of the stack. Access procedures for this type of Abstract Data Type can therefore only examine the top element. The depth of a stack is given by the number of elements in the stack. So an empty stack is a stack with depth zero.

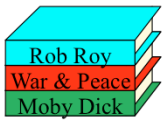
The LIFO property of a stack seems inherently unfair. How would you like to be the first person to arrive on the "stack" for a movie (as opposed to a line for a movie). You would be the last person to be allowed in! Stack are not prevalent in everyday life. The property that we usually desire in our daily life is "first-in-first-out" (or FIFO). A **queue** is the Abstract Data Type with the FIFO property, as we will see later on in this lecture. Most people would much prefer to wait in a movie queue (or in a line) than in a movie stack! However, while the LIFO property is not very appropriate for many day-to-day situations, it is very much needed for a large number of problems that arise in computer science. We will illustrate some during this lecture and others in the Tutorial 2.

The access procedures for a stack include operations such as examining whether the stack is empty (but not how many items are in the stack), inspecting the top element but not others, placing an element at top of the stack, but at no other position, and removing an element from the top of the stack, but from no other position. Stacks can therefore special lists where the accessor and mutator access procedures are restricted to just the top element.

Illustrations

- Stack of books:

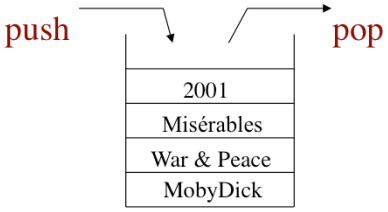
Initially:



After removing a book:

After adding "Misérables":

After adding "2001":

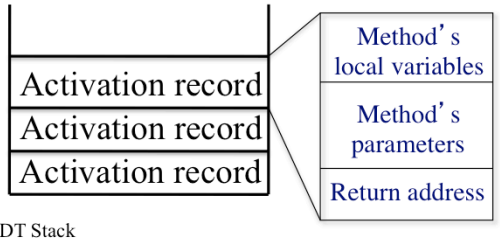


An empty stack

An Application of Stacks

- **Interpreter** (e.g., the **Java Virtual Machine**)
  - maintains a stack containing intermediate results during evaluation of complicated expressions
  - also containing arguments and return addresses for method calls and returns,
  - in particular when executing recursive methods…

Program Stack during recursion

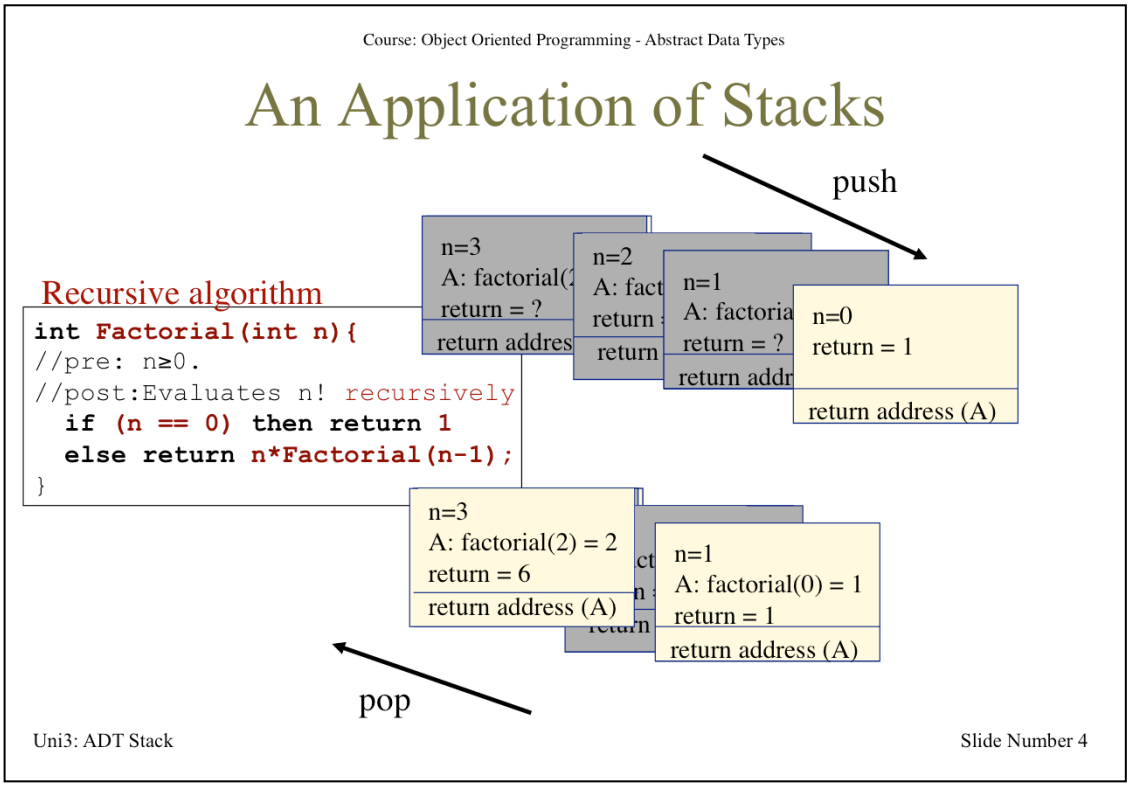


- A representation of variables declared within a method is created on activation of that method, and destroyed on exit from it.
- A distinct representation of variables is created on each re-activation of a method.

One of the typical applications of an ADT Stack is the use of a "Program Stack" by the same Java Virtual Machine. During the execution of a Java program, the interpreter needs to keep track of the intermediate results computed during the evaluation of complex expressions (e.g. concatenation of methods calls with return values), as well as the arguments and return point in the program where to go back once a method call has been executed. This is particularly important during the execution of recursive calls.

Java keeps track of the call of a method (say) A, in the following way. When a method A is called (whether recursive or not) the run-time environment temporarily stops the execution of the current method (this could well be a main program). Before actually executing the method A, some information is saved that will allow the run-time environment to return to the correct instruction after the method A is completed. The run-time environment also provides memory for the parameters and local variables that the method A uses. This memory block is called "activation record" and essentially provides all the important information that the method A needs in order to work. It also provides information as to where the method should return when it is done with the called computation. Once this block is created, the method A is then executed. If the execution of method A should encounter a call to another method, recursive or otherwise, then the first method's computation is temporarily stopped. This is because the second method must be executed first before the first method can continue. Information is saved that indicates precisely where the first method should resume when the second method is completed. An activation record for the second method is then created and placed on top of the other existing activation records. The execution then proceeds to the second method. When this second method is completed, its activation record provides the information as to where the computation should continue. The execution record of this second method is then removed from the top of the collection of existing activation records. All the activation records so created are stored in the "program stack". Each newly created activation record is pushed on top of the program stack and, at the end of execution of a method its activation record is popped from the program stack.

This mechanism is used for both recursive and non-recursive method calls. We will now see, in particular, examples of recursive methods.



This is an example run of recursive call. Note this slide is animated. At each method call a new activation record is created and pushed onto the program stack (the current activation record becomes grey, as it is temporarily stopped). At the end, on the last recursive call, the operation is a simple return instruction. The last activation record is then destroyed or popped off the stuck. And the previous activation record becomes activated (in the animation here it becomes yellow again)... until we reach the activation record at the bottom of the stack and the program is then completed.

Example use of a stack

Text-file reversal

- A text *file* is a sequence of (zero or more) lines.
- To reverse the order of these lines, we must store them in a *first-in-last-out* sequence.
- Text-file reversal algorithm: output the lines of *file* in reverse order:

```
1. Make line-stack empty.
2. For each line read from file, repeat:
       Add line to the top of line-stack.
3. While line-stack is not empty, repeat:
     Remove a line from the top of line-stack into line
     Output line.
```


This is another example use of an ADT stack. Because of its special LIFO access to its elements, a Stack can be used to reverse the order of elements in a given list, or collection. For instance, we can consider the example of a text file, as a sequence of lines. If we want to reverse the order of these lines, so that the last line becomes the first we can think of copying all the lines of the file into a stack and then read the elements off from the so constructed stack. Because of the LIFO access, we will be reading the last line of the file first, since this is the last element we would have pushed into the stack. A pseudo code of such an example is given in this slide.

Example use of a stack

Bracket matching

An expression is **well-bracketed** if:

- for every left bracket, there is a later matching right bracket
- for every right bracket, there is an earlier matching left bracket
- the sub-expression between a pair of matching brackets is itself well-bracketed.

- Examples and counter-examples (math expressions):

$s \times (s - a) \times (s - b) \times (s - c)$ — well-bracketed

$(-b + \sqrt{[b^2 - 4ac]}) / 2a$ — well-bracketed

$s \times (s - a) \times (s - b \times (s - c)$ — ill-bracketed

Another typical application or use of a stack is for checking bracket matching or any symbol matching (you could think for instance to begin and end tags in a special tagged file, etc.). Matching brackets is, for instance, one of the key functions that a compiler has to perform in a given program in order to check that expressions are written syntactically correct.

But what does matching brackets mean?

It means that for every left bracket encountered when parsing a given expression, there has to be a right bracket, later in the expression, matching it. Analogously for every right bracket there has to be an earlier matching left bracket. In this slide I have given some examples of expressions that are well bracketed and ones that are not (i.e. the last one).

But how can we check pairs of matching brackets as we parse a given expression?

Example use of a stack

Bracket matching algorithm:

```
1. Make bracket-stack empty.
2. For each symbol sym in phrase (scanning from left to right),
   repeat:
        If sym is a left bracket:
              Add sym to the top of bracket-stack.
        If sym is a right bracket:
              If bracket-stack is empty, terminate with false.
              Remove a bracket left from the top of bracket-stack.
              If left and sym are not matched brackets, terminate
              with false.
3. Terminate with true if bracket-stack is empty, or false
otherwise.
```


We can use a bracket stack. Each right bracket we encountered is supposed to be matching the last encountered left bracket. So we can use a stack to store all the left brackets as we encounter them in the expressions. For each right bracket, we remove a left bracket stored in the stack since this has now be paired (matched). Of course if the stack result to be empty the expression is not well bracketed since there is some right bracket that cannot be paired with a left bracket. So the second property of matching brackets defined in the previous slide is violated.

Analogously, if at the end of the expression we are left with a bracket stack that is not empty, than the expression is also ill bracketed but, in this case because there are more left brackets than right brackets. So the first property of bracketing matching defined in the previous slide would be violated. If none of these two happens, we can conclude that the given expression is well bracketed. This type of idea is application in other similar contexts, such as computing infix mathematical expression. Tutorial 2 will include an example use of a stack in this context.

Specifying operations of a Stack

createStack( )
// post: Create an empty stack.

push(newElem)
// post: Inserts newElem at the top of the stack, if there is no violation of capacity.
// post: Throws StackException if the stack is full.

isEmpty( )
// post: Determines if a stack is empty

peek( )
// post: Returns the top element of the stack without removing it from the stack.
// post: Returns *null* if the stack is empty.

pop( )
// post: Retrieves and removes the top element from the stack.
// post: Returns *null* if the stack is empty

We have now seen some classes of applications problems where the use of a stack is particular relevant. Let's see how we can define this notion of stack as an ADT. We have already said that it is a linear sequence of elements with a specific type of access (LIFO). But to fully specify the ADT we need to define its access procedures. In this slide I have included the core access procedures for a stack. In the current Java 6 version an ADT Stack is defined through the notion of double-ended queue, which we will be present later in this lecture. The double-ended interface (i.e. **Deque**) includes more access procedures (e.g. more than one type of peek method). The old legacy Stack interface defined in the previous versions of Java is now been replaced by the deque interface.

The main access procedures for a stack include the operation push(newElem) that adds an element to the stack. Because of the LIFOP access policy to the data structure this element is added to the "top" of the stack. Violations can occur only in the case of static implementation of a stack, where the memory allocated has become full. The second operation is peek(). This access procedure is an accessor method as it does not change the content or structure of the ADT. It returns the top element in the stack without modifying the content of the stack. If the stack is empty, the access procedure could either throw an exception or just return a null reference. In this slide we have followed the second case, as this is also the case for JAVA API definition of peek() method in a deque. Finally the other key access procedure is **pop().** This operation retrieves the top element of the stack (as it is the same for the peek() operation) but it also deletes it from the stack. Again it returns null if the stack is empty. Alternatively it could be defined so to throw a stack exception.

Axioms for ADT Stacks

The access procedures must satisfy the following axioms, where Item is an element in the stack:

1. (aStack.createStack( )).isEmpty = true
2. (aStack.push(Item)).isEmpty() = false
3. (aStack.createStack( )).peek( ) = null
4. (aStack.push(Item)).peek( ) = Item
5. (aStack.createStack( )).pop( ) = null
6. (aStack.push(Item)).pop( ) = Item

A stack is like a list with peek( ), pop( ) and push(item) equivalent to get 1st item, remove 1st item and add item at the beginning of a list.

In this slide I have listed the main axioms that the access procedures for an ADT stack have to satisfy. In this case, we have two axioms for each operation.

Note that the access procedures for a stack are somewhat equivalent to the access procedures for a list described in the previous lecture (Unit2). In particular, the peek( ) procedure for a stack can be seen as equivalent to the get(1) procedure for a list, which takes the first element of a given list. In the same way, the procedure pop( ) for a stack is equivalent to the procedure remove(1), which removes the first element in a list, whereas the procedure push(item) is equivalent to the procedure add(1,item), which adds the given item to the first position in a list.

In the next slides we' ll look at the two different ways of implementing a stack, one based on array and the other based on linked lists. The definition of data structure for a stack in the case of a dynamic implementation will further illustrate the fact that stacks are essentially lists with restricted use of their access procedures.

Stack Interface for the ADT Stack

```
public interface Stack<T>{
    public boolean isEmpty();
    //Pre: none
    //Post: Returns true if the stack is empty, otherwise returns false.

    public void push(T elem) throws StackException;
    //Pre: elem is the new item to be added
    //Post: Inserts elem at the top of the stack, if there is no violation of capacity.
    //Post: Throws StackException if the stack is full.

    public T peek()
    //Pre: none
    //Post: Returns the top element of the stack without removing it from the stack.
    //Post: Returns null if the stack is empty.

    public T pop()
    //Pre: none
     //Post: Retrieves and removes the top element from the stack.
     //Post: Returns null if the stack is empty.
```

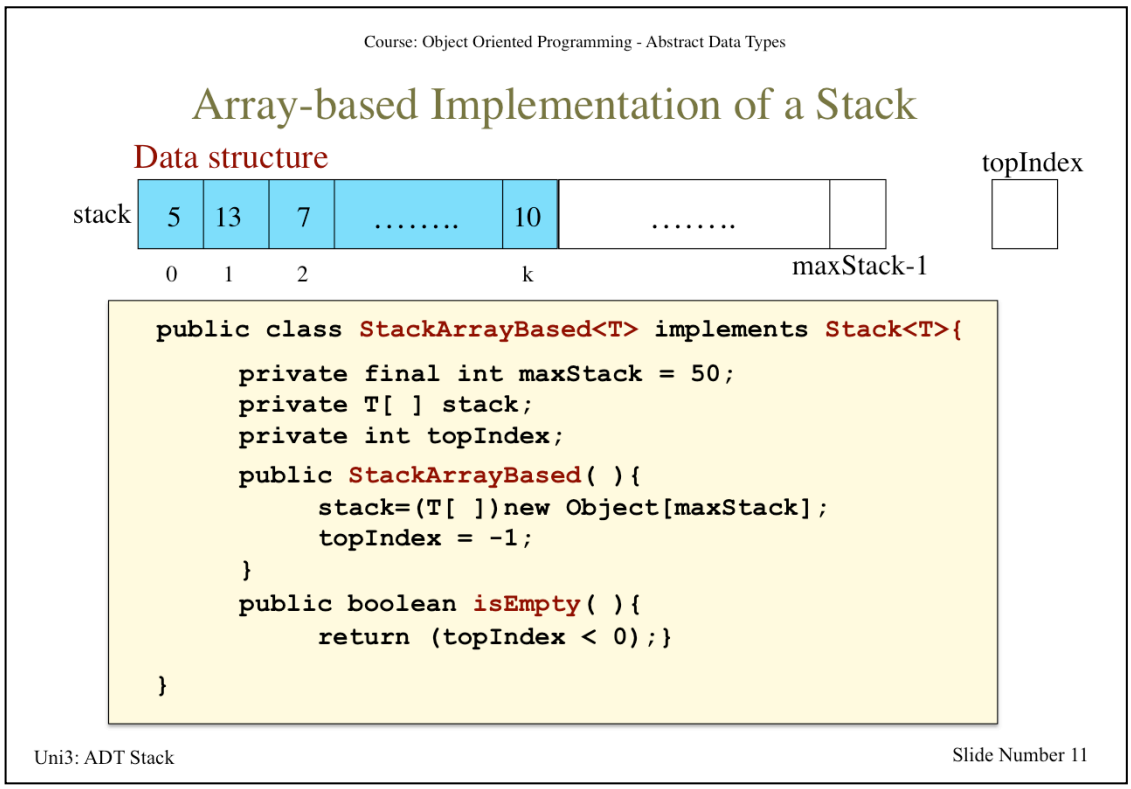

The interface Stack<T> given in this slide provides the definition and specification of the main access procedure for the ADT stack. This is a generic interface where the type of the element stored in the stack is given by the generic type T. A non generic interface, i.e. where the items of a stack are of type Objects would look very much like this interface but with the parameter T removed from the declaration and replaced with the type Object in the rest of the interface.
As in the case of the ADT list, the constructor createStack( ) is not included here as it is normally given by the constructor of the particular Java class that implements this interface.

The two main types of implementation (array-based, reference-based) for a stack are classes that implement this interface.

An example of the StackException for a stack can be the following:

```
public class StackException extends java.lang.RuntimeException {
        public StackException(String s){
                super(s);
        }// end constructor
}// end StackException
```

Note that this exception could be avoided in the case of the push(item) access procedure. For instance, for a static implementation, the method push could check whether the array is full before trying to push a new entry in the stack. A private method, *isFull*, can be defined: if the array is full you could use the same approach we have seen for the array-based implementation of lists, that extends the array. In a reference-based implementation, such exception is not necessary since the memory is dynamically allocated. The methods peek() and pop() are declared here to return a null reference when applied to an empty stack. This is the same definition as that used in the Java API. Alternatively we could have also defined these two methods so to throw an exception when applied to an empty stack.



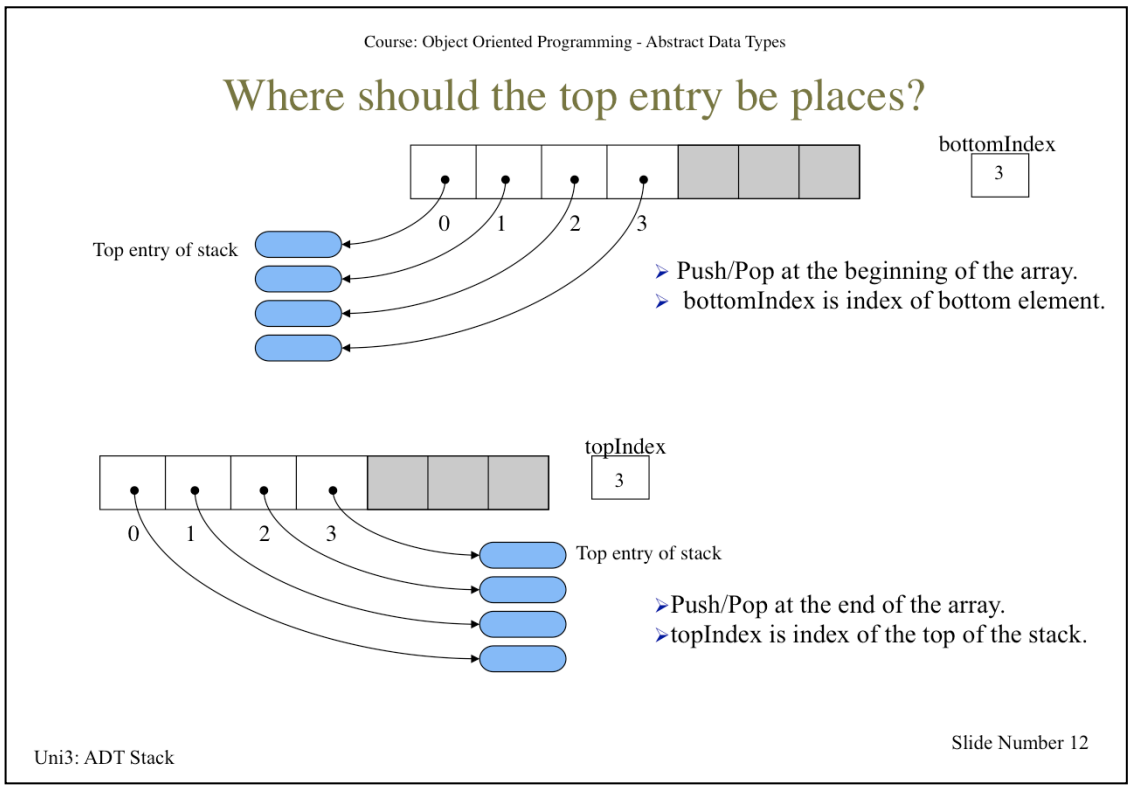
The data structure for a static implementation of a stack uses an array of objects of type T called **stack**, to store the elements in the stack, and an index value **topIndex** such that stack[topIndex] is the top of the stack (i.e. the last element added to the stack).

When a stack is created it does not include any item at all. In this case the value of the index `topIndex` is set to –1. This allows us to test in general when a stack is (or has become) empty by just checking whether the value of `topIndex` is a negative integer.

The partial implementation of `StackArrayBased` given in this slide includes the definition of the default constructor and the implementation of the method `isEmpty`. Note that since we are using in this case a generic class `StackArrayBased` the instantiation of the array is casted to (T[ ]) as it is not allowed to declare new `T[maxStack]` when `T` is a generic type.

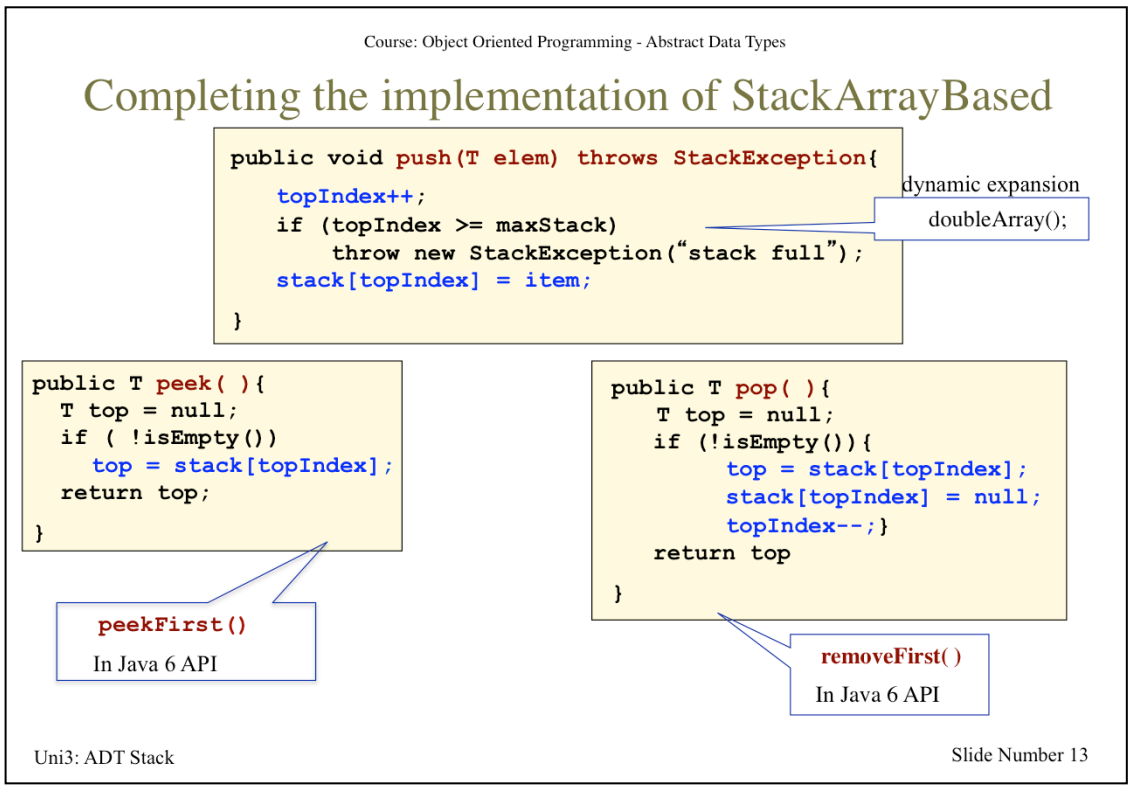
The implementation of a stack of Objects (instead of generic type) would not need the casting (T[ ]) in the constructor.

The implementation of the other methods included in the class `StackArrayBased` is given later.



When we implement a stack using an array data structure what we need to decide is where the top entry should be placed: should it be the first element in the array or the last element? This slide shows the two options. In the first case the class `StackArrayBased` would include the data field `bottomIndex` to store the array index of the bottom element in the stack, i.e. if incremented by one it gives the size of the stack as well. The operations push/pop would then add/delete the first element in the array and increment/decrement the bottomIndex as well. If you recall the evaluation slide given at the end of the previous lecture, the operation of adding and deleting an element at the beginning of an array is time inefficient, as we need to shift right (left) the remaining elements in the array each time we would make a push (pop).

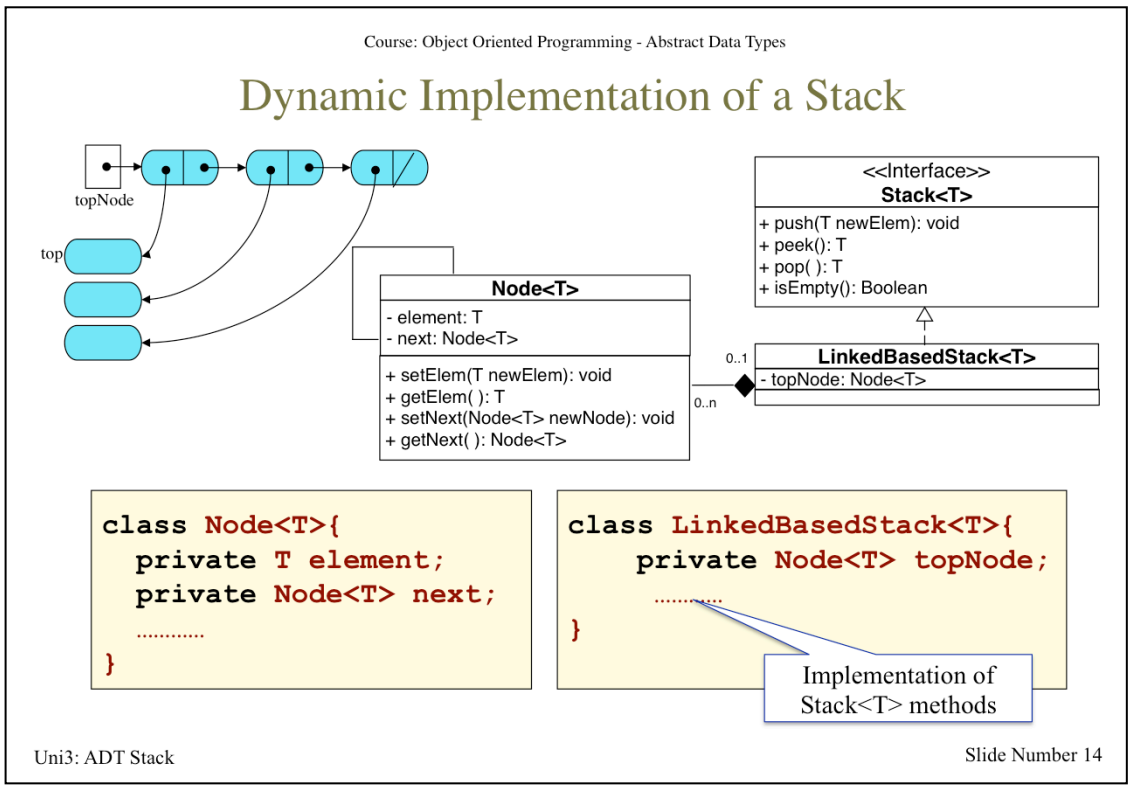
On the other hand, if we assume that the top of the stack is the last element in the array, the class `StackArrayBased` would include the data field `topIndex` (as in the previous slide) to store the index value of the last element inserted (i.e. top of the stack). The push and pop operations would in this case have a time efficiency of the order of at most 1, i.e. O(1). We'll only need to just assign an element at the end of the array when we want to make a push operation to the stack without moving the other elements. The addition of items in the stack would in this case start from position 0 in the array. Each time a push procedure is called, the value of `topIndex` is first incremented by 1 to point to the next top free cell in the array, checked if it is bigger than the size of the array and, if not, assign the new item to this position. Similarly, each time a `pop( )` operation is called, the deletion of the element is simply done by decrementing the value of `topIndex` by 1. The array will still include the element but no longer point to it. The implementation of the access procedures of the class `StackArrayBased` is given in the next slide.



In this slide, I have given example implementation of the access procedures of a stack for the case of a static implementation that uses an array as underlying data structure. In the method push, the case of throwing an exception can be replaced with dynamic expansion of the array when full, as we have seen in the case of array dynamic expansion for static implementation of a List.

These three main access procedures correspond to the methods `peekFirst()`, `removeFirst()` and `addFirst()` of a deque data type of the current Java 6 version. This differs from the current example, as it uses the dynamic implementation for which the addition at the beginning of a linked structure is of the O(1).

In our current static implementation, the operation push for an array data structure that does not expand would also be of the order O(1). If we use the dynamic expansion of the array or a vector the operation push can degrade to an O(n) operation since the complexity of doubling an array is of the order O(n), where n is the number of elements in the array. However, the push operations after a doubleArray has been performed are again of the order O(1). So, averaging out the cost of doubleArray operation over subsequent n push operations (where n is the initial size of the array) we would get that the cost of the push operation is almost of the order O(1) also when we use dynamic expansion. The operations peek() and pop() are of the order O(1).



In the dynamic implementation of a stack, the underlying data structure is a linked list. In this case the top of the stack should be the head of the linked list as the operation of adding a node to or deleting a node from the head of a chain is the most time-efficient operation. The first node in the chain should therefore reference the top entry in the stack. A diagrammatic representation of the dynamic data structure for a stack is given in this slide, together with an example of partial implementation of the class LinkedBasedStack<T>. This, in particular, uses the same Node<T> data structure that we have defined in the previous lecture for linked lists.

We give below the implementation of the other access procedures. Note that in the dynamic implementation of a stack, the operations push, peek and pop are all of the order O(1). Hence when you need a stack, think always of using a linked data structure!

```
public void push(T newElem){
        topNode = new Node<T>(newElem, topNode);
}
public T peek( ) {
         T top =  null;
         if (!isEmpty) top = topNode.getElem( );
         return top;
}
public T pop( ) {
        T top =  null;
        if (!isEmpty) {
            top = topNode.getElem( );
            topNode = topNode.getNext();
        }
        return top;
```

Implementation of Text-file reversal

```
public static void reverse (BufferedReader input, BufferedWriter
                                    output)throws IOException {
      Stack<String> lineStack = new LinkedBaseStack<String>();
      for (;;) {
                  String line = input.readLine();
                  if (line == null)  break;  // end of input
                  lineStack.push(line);
      }
      input.close();
      while (! lineStack.isEmpty()) {
         String line = lineStack.pop();
         System.out.println(line);
      }
      output.close();
}
```


As it is the case for an ADT list, once the operations of the ADT stack have been satisfactorily specified, applications can be designed that access and manipulate the ADT's data, by using only the access procedures without accessing directly to the underlying data structure.
We revise here the example of reverse of a text-file described at the beginning of this lecture. We have used here the methods defined in this lecture and not the JAVA 6 API methods of the deque<E> interface that are applicable to an ADT stacks.

Stack as part of Java Collection

Java provides the interface `Deque<E>` that includes a comprehensive set of operations, also for Stacks:

Stack Methods	Equivalent Deque Methods
push(elem)	addFirst(elem)
pop()	removeFirst(elem)
peek()	peekFirst(elem)

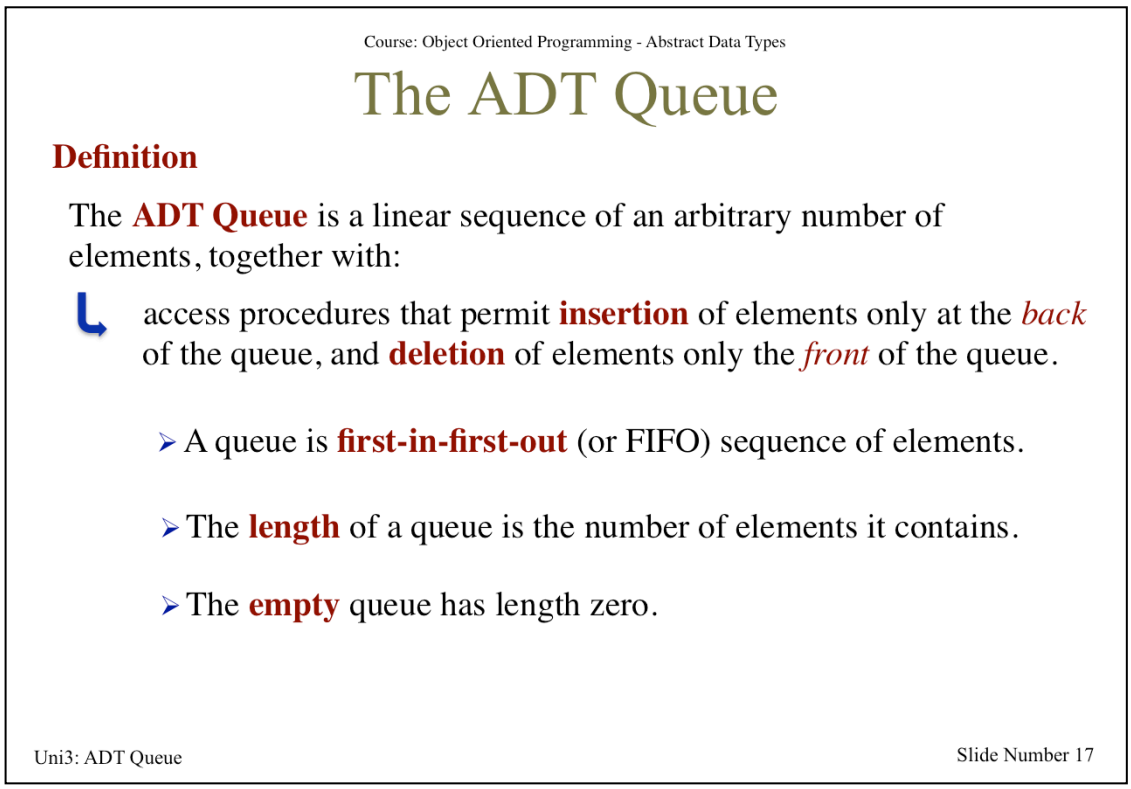
```
Deque<Integer> stack = new ArrayDeque<Integer>();
```

This is preferred to the legacy class Stack<E> that extends the class vector with the operations that allow a vector to be treated as a stack.

This is just a note regarding the Stack ADT already provided by the Java API. In past versions of Java, the Stack has been defined as an extended class of the vector class. The extended class included the additional access procedures that allow a vector to be treated as a stack.

In Java Version 5, it is recommended to the operations provided by the Deque interface. This is the ADT doubly-ended queue, which we will see in the next lectures. In Java 6, the deque interface provides a variety of access procedures for manipulating structures as stacks and as queues. In this slide I have given you the access procedure of a deque interface that are equivalent in behavior to the access procedures for a stack that we have seen in this unit. When using Java API you could for instance create a stack using the above command (in the case of a stack of integers for instance):

```
Deque<Integer>  stack = new ArrayDeque<Integer>
```



A **queue** is like a line of people where the first person to join a line is the first person served and is thus the first to leave the line. Queues are much more appropriate for many real-world situations than stacks. Whereas a stack can be seen as a list with only one end, because all the operations are performed at the top of the stack, the queue can be seen, in contrast, as a list with two ends, the **front** and the **back**.

Queues have the main property of allowing new elements to be added only at the back of the sequence, and the element first inserted in the sequence (i.e. the item at the front of the queue) is the first to be removed. This behaviour is commonly referred to as "first-in-first-out" or simply **FIFO**. So the first element in a queue is the "front" element, and the last element in a queue is the "back" element.

The access procedures for a queue include operations such as examining whether the queue is empty, inspecting the element at the front of the queue but not others, placing an element at the back of the queue, but at no other position, and removing an element from the front of the queue, but from no other position. We will see in the rest of this lecture uses of queues, example application, definition of the ADT and both static and dynamic implementation of such an ADT.

We will also briefly introduce variations of queues: sometime, more flexible access is needed than what a queue permits. For instance, **double-ended queues** allow to organise the data like a queue but enables you to operate on both its oldest and newest entries. **Priority queues** also organize data as queue but with priorities defined on the entries which depends on criteria other than their arrival time. Access is allowed according to their priorities instead of chronologically. Some examples of these queue variations will be given in Tutorial 2.

Applications of Queues

- Scheduler (e.g., in an operating system) for controlling access to shared resources
  - maintains queues of printers' jobs
  - maintains queues of disk input/output requests
- Simulation of real word situations
  - use queues to simulate waiting queues in real scenarios (e.g. find out a good number of tellers, by simulating typical customer transactions)
- Telephone operators
  - queuing system for handling calls to toll-free numbers.

Queue structures are normally used whenever it is needed to capture "waiting lines". This slide shows few examples of where queues are needed in both computer science applications and real-world applications. Within the first category, typical applications is for controlling access to shared system resources such as printers, files, disks, etc.. A specific example of print queues is when multiple users (in a network) need to share a printer. Print requests are added to a print queue by the system and when the request reaches the front of the queue than it is printed. This ensures that only one print job is dealt at the time (underlying notion of shared resources) and requests are served on a first-come first-served basis.

In real-world applications, simulations are typical examples where queues are useful. Suppose that as a bank manager, you need to determine how many tellers to install or keep, in order to serve each customers within a "reasonable wait time", but at the same time not have too many tellers (to reduce the costs). To find out a good number you will probability need to run a simulation of typical customer transactions, (as events) and using queues to represent the waiting customers.

Another real-world application is that of telephone operators. Calling for instance a toll-free number of a company/bank/etc. it is often the case that your call gets put into a queue until an operator is free to answer the call. This is again dealt using a queuing system.

Example use of a Queue

Demerging algorithm

Consider a file of person records, each of which contains a person's name, gender, date-of-birth, etc. The records are sorted by date-of-birth.

Task:

Rearrange the records such that females precede males but they remain sorted by date-of-birth within each gender group.

Sorting algorithm ✗

Time complexity
$O(n \log n)$

Demerging algorithm ✓

Time complexity
$O(n)$

Queue structures can also be used in order to support more efficient algorithms, not just to store and handle data in a queue manner when requested by the application problem. An example is the "demerging algorithm". Support that you have a large collection of data ordered with respect to a certain key value. For instance a demographic survey produces a large file of people records ordered according to the date of birth. Suppose that a secondary order is needed, i.e. put all the female people first and all the male people after but preserving their ordering on age. If we were going to apply in this case some sorting algorithm, we would have at least required a time complexity of the order O(n log n) in the best case scenario.

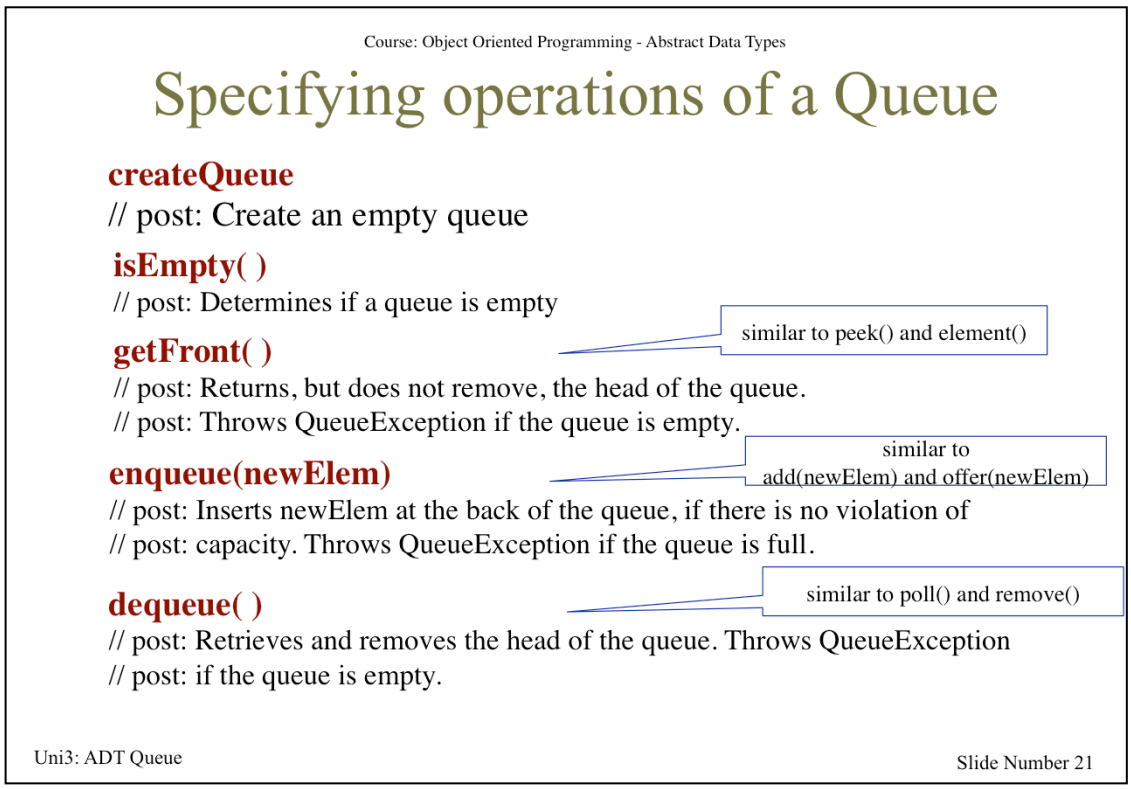
Queues can be used to obtain a more efficient sorting algorithm. We can create as many queues as many sorted classes we need. For instance in the case above we only need two sorted classes (males and females). The queue access of first-in-first-out will guarantee that records will be read from the queue in the same order as they are read from the initial file. The time complexity in this case is just of the order O(n). Detail of the algorithm is given in the next slide.

Demerging Algorithm

Rearrange a file of person records such that females precede males but their order is otherwise unchanged.

```
1. Make females queue and males queue.
2. For each record in the input file, repeat:
      Let p be the next person read from the file.
      If p is female, adds p to the females queue
      If p is male, adds p to the males queue
3. While females is not empty, repeat:
      Remove the person from the females queue and
      write it out to a file
4. While males is not empty, repeat:
      Remove the person from the males queue and
      write it out to the same file
5. Output the file.
```


To re-arrange the records according to a secondary ordering (females and males) we can create two queues, one for the females and the other for the males. As record are read for the file, they are added at the back at to the female queues (for female records) and male queues (for male records). Within each queue the access to the records preserves the same ordering as in the initial file (i.e. age ordering). So reading from these two queues in order to create the new sorted file, we will be able to create a file that includes first all the female (from the females queue) in the same ordering as they were entered in the queue and then all the male records (from the males queue) again in the same ordering as they were entered in the initial file.



This slide provides the list of the main access procedures and operations for an ADT queue. The operation ***enqueue(newElem)*** adds a new entry at the back of a queue. In the specification given here, the operation throws a ***QueueException*** if the queue capacity is violated. This operation corresponds to the operation ***add(newElem)*** in the Java 6 API. You could also define an alternative operation that does not throw an exception but allows dynamic expansion of the queue. This corresponds to the operation ***offer(newElem)*** in the Java 6 API.

The operation ***dequeue()*** retrieves and removes an item from the front of the queue operation. Again this operation can be defined so that it returns *null* when the queue is empty, as it is the case of ***poll()***, in the Java 6 API. The operation specified here corresponds to the operation ***remove()*** in the Java 6 API for queues.

Similar to the operation *dequeue()* is the operation ***getFront().*** This, however, retrieves the element at the front of a queue, but leaves the queue unchanged. It throws an exception if the queue is empty, as it is the case for the corresponding operation ***element()*** in the Java 6 API. It can be defined so to return *null* if the queue is empty, as it is the case for the operation ***peek()*** in Java 6 API.

Since elements in a queue can be looked at and/or removed from the front of a queue, one way to look at an entry that is not at the front of a queue is to remove repeatedly all the elements from the queue until the desired one is reached at the front. The queue does not have therefore a search operation as part of its interface. But you can thinking of making the queue interface extend iterable and provide the implementation of an ***iterator*** that will scan a queue without removing elements from it.

Axioms for ADT Queues

The access procedures must satisfy the following axioms, where Elem is an element and aQueue is a given queue:

1. (aQueue.createQueue( )).isEmpty = true
2. (aQueue.enqueue(Elem)).isEmpty() = false
3. (aQueue.createQueue( )).getFront( ) = error
4. pre: aQueue.isEmpty() = true:
   (aQueue.enqueue(Elem)).getFront( ) = Elem
5. (aQueue.createQueue( )).dequeue( ) = error
6. pre: aQueue.isEmpty( )=false:
   (aQueue.enqueue(Elem)).dequeue( )=(aQueue. dequeue( )). enqueue(Elem)

A queue is like a list with getFront( ), dequeue( ) and enqueue(elem) equivalent to get 1st elem, get and remove 1st elem, and add elem at the end of a list respectively.

In this slide I have listed the main axioms that the access procedures of an ADT queue have to satisfy. Note that axioms 3 and 5 could have defined differently if the operations return *null* instead of throwing an exceptions.

In the next slides we'll look at the two different types of implementations of a queue, one based on arrays and the other based on linked lists. The definition of the data structure for a queue in the case of a dynamic implementation will further illustrate the fact that queues are essentially lists with restricted use of their access procedures.

Queue Interface for the ADT Queue

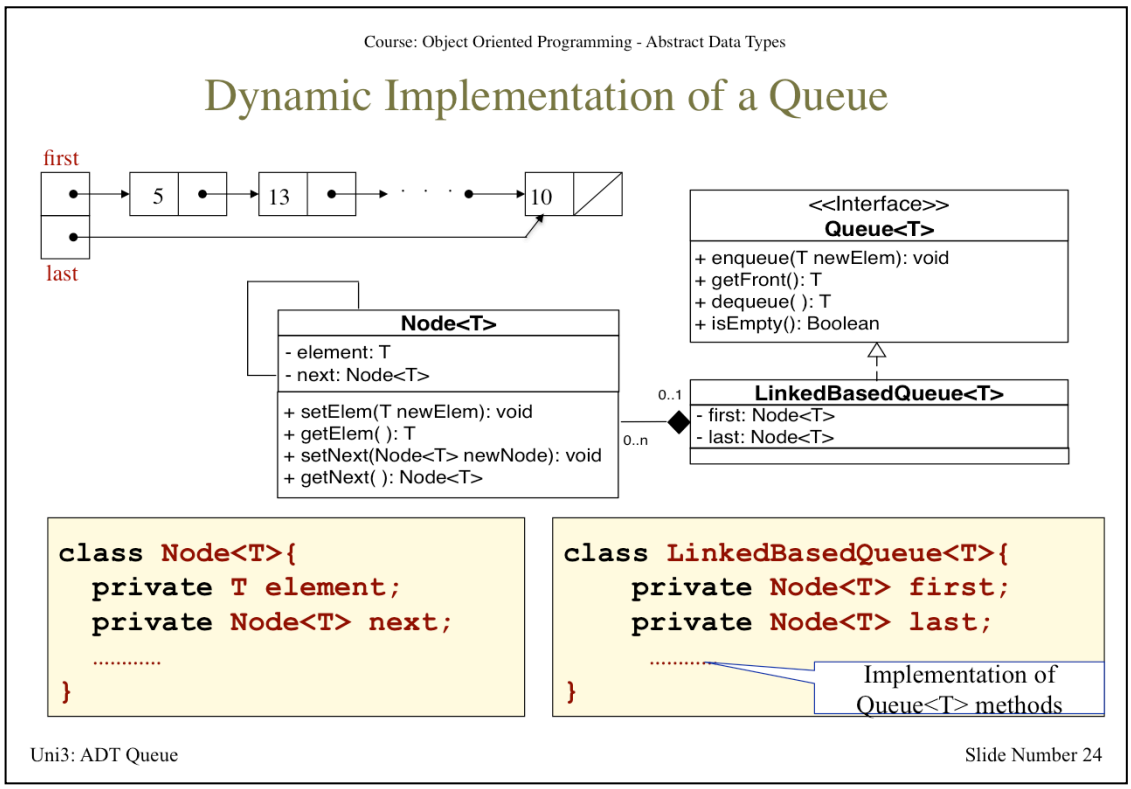
```
public interface Queue<T>{
    public boolean isEmpty();
    //Pre: none
    //Post: Returns true if the queue is empty, otherwise returns false.
    public void enqueue(T elem) throws QueueException;
    //Pre: item is the new item to be added
    //Post: If insertion is successful, item is at the end of the queue.
    //Post: Throws QueueException if the item cannot be added to the queue.
    public T getFront( ) throws QueueException;
    //Pre: none
    //Post: If queue is not empty, the item at the front of a queue is returned, and the queue
    //Post: is left unchanged. Throws QueueException if the queue is empty.
    public T dequeue( ) throws QueueException;
    //Pre: none
    //Post: If queue is not empty, the item at the front of the queue is retrieved and removed
    //Post:from the queue. Throws QueueException if the queue is empty.
}//end of interface
```


The generic interface Queue<T> given in this slide provides the main definition of each access procedure for the ADT queue. As in the case of the ADT list, the constructor createQueue( ) is not included here as it is normally given as constructor of the particular Java class that implements the interface Queue.

The two main types of implementations (array-based, linked-based) for a queue are classes that implement this interface.

An example of the *QueueException* for a queue can be the following:

```
public class QueueException extends java.lang.RuntimeException {
        public QueueException(String s){
                super(s);
        }// end constructor
}// end QueueException
```



Like any other ADT, queues can have an array-based or a linked-based implementation. For a queue, the linked-based implementation is more straightforward than the array-based one, so we start with this first.

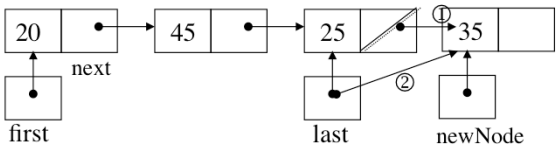
A linked-based implementation of a queue can use the same data structure as that used by a linked list, but with two external references, one to refer to the front of the queue, i.e. the first element in the queue, and one to refer to the back of the queue, i.e. the last element in the queue. We call these two reference variables "`first`" and "`last`" respectively. If we had only the reference "`first`", as in a basic linked list, the `enqueue` operation would be very inefficient as it would require scanning the entire queue in order to add an element at the back. Having two references for, respectively, the first and last node in the chain, makes the access procedures `enqueue` and `dequeue` more efficient. It is also more efficient to have the first reference variable to reference the front of the queue and the last reference variable to reference the last node in the queue. If it was the other way around (i.e. against the linking direction of the list) the `dequeue` operation would be more inefficient as it would require searching for the preceding node in order to delete last node in the chain. The entire queue would be needed to be traversed at each add operation.

When we first create an empty queue (no item is inserted), the two reference variables `first` and `last` still exist and their reference value is `null`. This is shown in the main constructor of the class `LinkedBasedQueue` given in the next slide.

The implementation of the operation `isEmpty( )` is therefore pretty simple, and it consists of checking that the reference variable last is equal to null, as shown below. Of course, it is actually sufficient to check that either of the two variables `last` or `first` is equal to `null`. If the queue were including at least one element both `first` and `last` reference variables would be different from `null`, as shown in the sample implementation given in the next slides.

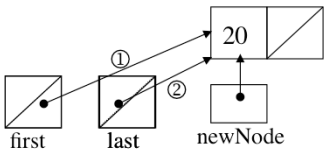
Dynamic Implementation of a Queue (continued)

Inserting a new Node in a queue



1. last.setNext(newNode);
2. last = newNode;

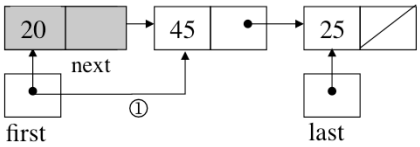
For non-empty queues



1. first = newNode;
2. last = newNode;

For empty queues

Deleting a Node in a queue



1. first = first.getNext( );

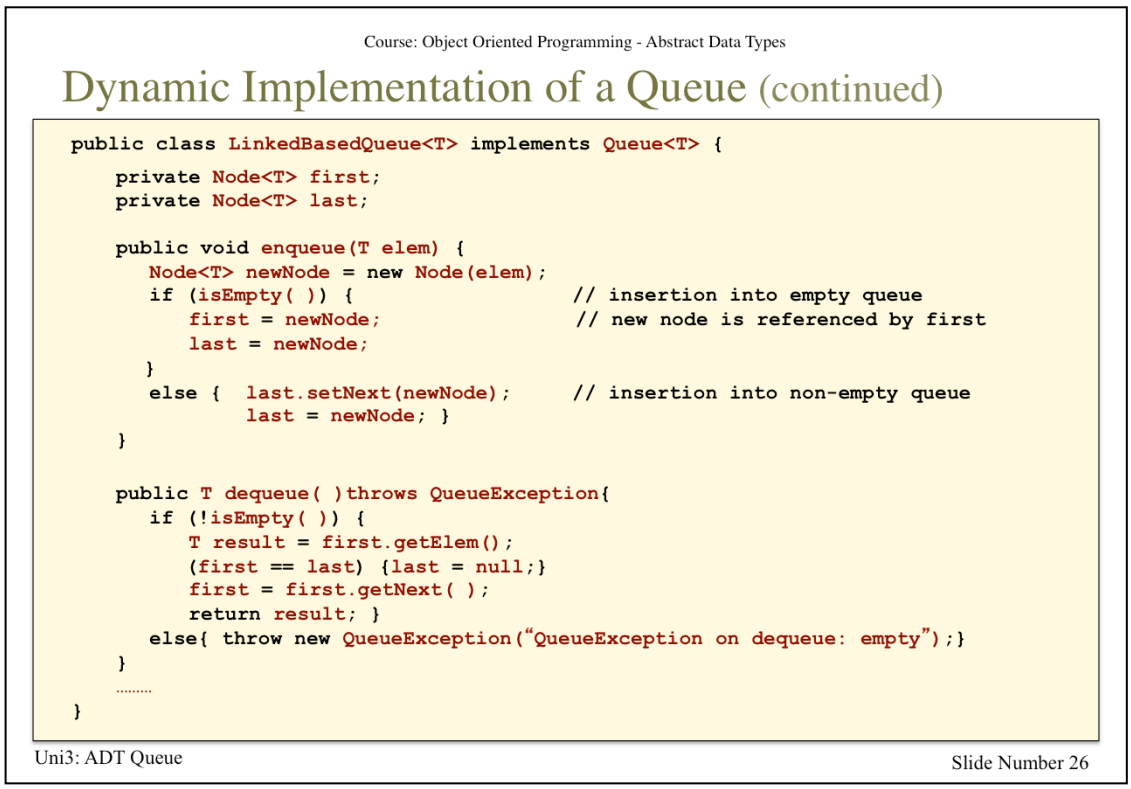
For non-empty queues

Insertion at the back of a not-empty queue is straightforward. A diagrammatical representation is given in this slide, where the dashed lines indicate reference values before the operation. Addition of an item at the back of a non-empty queue requires two reference changes: the "`next`" reference of last node needs to be changed so as to reference `newNode` instead of `null` (this is denoted with ① in the diagram), and the external reference `last` needs to be changed so as to reference now `newNode` (this is denoted with ② in the diagram).

When the queue is empty both instance variables `first` and `last` have value equal to `null`. The addition of one new element requires the variables `first` and `last` to reference the node referenced by `newNode` since the single item in the queue is both the first and the last element.

The case of deleting an element from the front of a non-empty queue requires just one reference change: `first` has now to reference the second element in the chain.

Note that if the queue includes only one single element, then the command "`first = first.getNext( )`" will automatically set the reference variable `first` equal to `null`, but we will still need to change the value of the reference variable `last` to be equal to `null`.

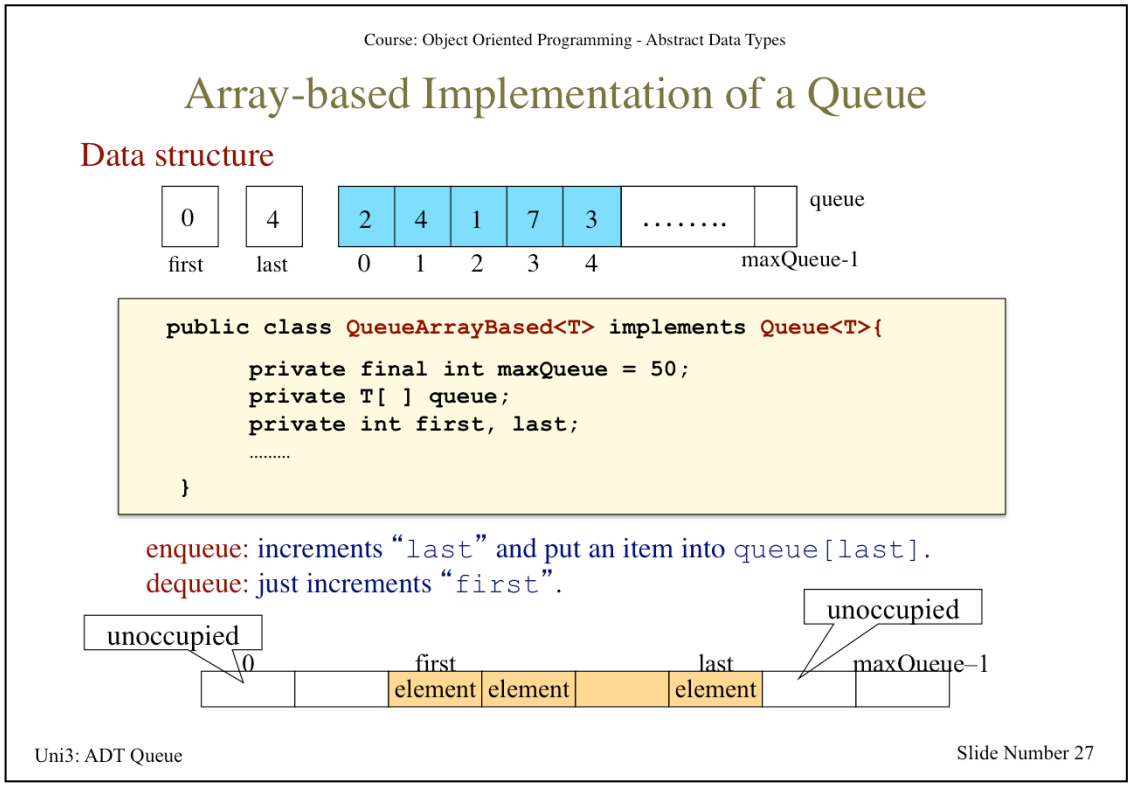


This slide gives the basic implementation of the operations "`enqueue`" and "`dequeue`". Note that a slightly different implementation would be required if instead of using just a linked list as data structure, we had chosen to use a circular linked list.

What is missing is an example implementation of the access procedure "`getfront ( )`". The code is given below:

```
public T getFront( ){
    if (!isEmpty( )) {
       // queue is not empty. Retrieve the front element;
       return first.getElem( );
    }
    else {throw new QueueException("QueueException on
                                    front: queue empty");}
}
```

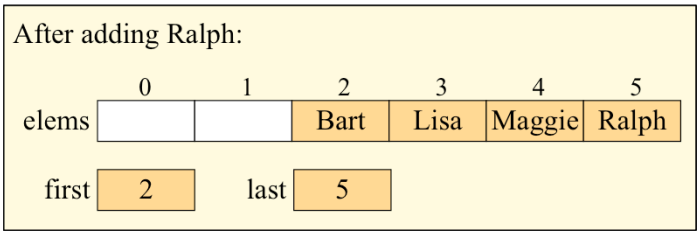
Note that a different data structure could have been chosen to implement the queue dynamically. An example could be to use a circular linked list and just a single reference variable referring to the last node in the circular list. This would allow us to get the reference to the first item in a queue by just getting the next value included in the last item of the queue. In Tutorial 2, you are asked to give a dynamic implementation of a queue using a circular linked list.



For applications where a fixed-sized queue does not present a problem, we could also use an array data structure to implement a queue. A naïve array-based implementation of a queue is to consider an array, whose size gives the maximum size of the queue, and two variables "`first`" and "`last`", which contain, respectively, the index value of the first element and the last element in the queue. In the slide we have named this array "`queue`". At the beginning "`first`= 0" and "`last`= -1" to indicate that the queue is empty. To add an element to the queue, we would in this case increment the value of "`last`" and place the new element in `queue[last]`. To `dequeue` an element we would just need to increment the variable "`first`". To check if the queue is empty we would just need to check that `last<first`. We can check that the queue is full by checking if `last=maxQueue-1`. One of the problem with this implementation is that after a number of `enqueue` and `dequeue` operations, the items in the queue will drift toward the end of the array, and `last` could become equal to `maxQueue-1` without the queue being actually full. This is illustrated through an animation in the next slide.

Array-based Implementation of a Queue

Animation (with *maxQueue* = 6):



What happens when last reaches the end of the array and the queue is not full?

We would like to re-use released space between the start of the array and first.

- We can shift elements along the array, BUT operations `enqueue` and `dequeue` will have time complexity $O(n)$.
- We can avoid this if we use a "**circular array**" instead of ordinary array.

To re-use the empty spaces at the front of the array, we could think of implementing the `enqueue` and `dequeue` operation in a way that they shift the elements in the array to the left after each `dequeue` operation or whenever `last` becomes equal to `maxQueue-1` and we would like to `enqueue` new elements. But this is not an efficient solution. In the worse case scenario the two operations will be of the order O(n).

A more elegant solution is to view the array as circular, as shown in the next slide.

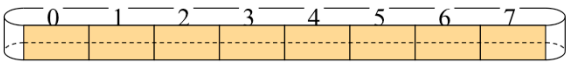
Course: Object Oriented Programming - Abstract Data Types

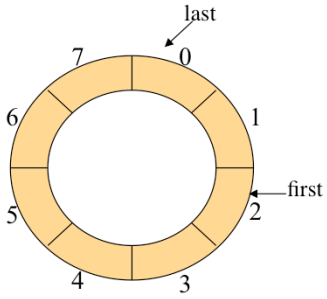
Array-based Implementation of a Queue

Using a circular array

In a **circular array** *a* of length *n*, every component has both a successor and a predecessor. In particular:

- the successor of *a*[*n*–1] is *a*[0]
- the predecessor of *a*[0] is *a*[*n*–1].





Enqueue:
```
last=(last+1)%maxQueue;
queue[last] = newElem;
  ++count;
```

Dequeue:
```
first = (first+1)%maxQueue;
--count;
```

Uni3: ADT Queue

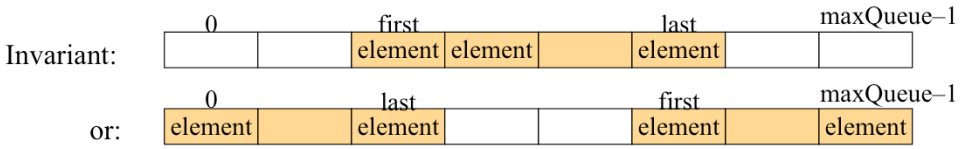
Slide Number 29

In this case `first` and `last` advance clockwise when `dequeue` the first element and `enqueue` a new element respectively. At the beginning, the empty queue is given with `first`=0 and `last`=`maxQueue-1`. To obtain the effect of circular array, the values of first and last are calculated using modulo arithmatic (that is the Java % operator) as illustrated in the slide. The main question is now: what conditions define the queue to be full and what conditions define the queue to be empty? We can think of the queue to be empty whenever first passes the value of last. However, this condition would also be possible when the queue becomes full. Because the array is circular consecutive `enqueue` operations will just increase the value of `last` until it will eventually catch up with `first` (i.e. when the queue is full). We therefore need to have a way to distinguish between these two different situations. This can be implemented using a "counter" variable, called here "`count`", which is incremented at each `enqueue` operation and decremented at each `dequeue` operation. `count`=0 will indicate an empty queue, whereas `count`= `maxQueue` will indicate a full queue. And of course in this static implementation approach when the queue is full we can double the array first and then add a new entry to the queue.

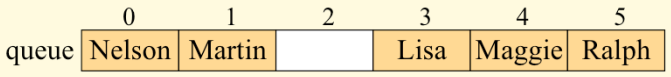
Array-based Implementation of a Queue

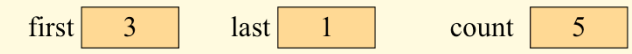
Using a circular array:

- a **circular** array *queue* of length *maxQueue*, contains the queued elements **either** in *queue*[*first*…*last*] **or** in *queue*[*first*…*maxQueue*–1] and *queue*[0…*last*].



Removing the front element:





A circular array implementation of a Queue

```
public class QueueArrayBased<T> implements Queue<T>{
    private final int maxQueue = 50;
    private T[ ] queue;
    private int first, last, count;

    public QueueArrayBased( ){
        queue = (T[ ]) new Object[maxQueue];
        first = 0; count = 0; last = maxQueue-1;
    }

    public void enqueue(T newElem) throws QueueException {
        if (!isFull( )){
           last = (last+1) % maxQueue;
           queue[last] = newElem;
           count++;
        }
        else {throw new QueueException("Queue is full");}    }
```


Example implementations for the main access procedures of a queue in the case of circular array implementation of the queue, are given here and in the next slide.

The performance of `enqueue` is O(1). However, if we use the dynamic expansion of the array when the queue is full, his performance will degrade to O(n) when the array is full because the doubleArray operation is an O(n) operation.

A circular array implementation of a Queue

```
public T dequeue( ) throws QueueException {
     if (!isEmpty( ))
        { T queuefront = queue[first];
          first =(first+1)% maxQueue;
          count--;
          return queuefront;
         }
     else {throw new QueueException("Queue is empty");}
}


private boolean isFull( )
{
      return count = = maxQueue};
}
```

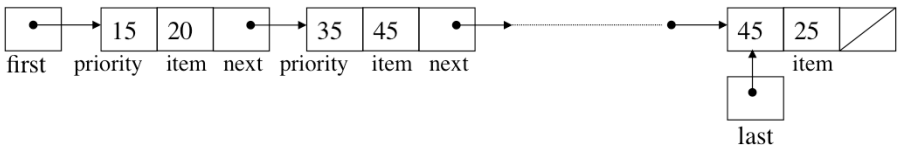

What would the implementation of the access procedures `isEmpty( )` and `getfront( )` be? These are left as exercise because they are really very simple.

Course: Object Oriented Programming - Abstract Data Types

The ADT Priority Queue

Definition

- The **ADT Priority Queue** is a linear sequence of an arbitrary number of items, **which are ordered**, together with access procedures.
- The access procedures permit **addition** of elements into the queue in priority order, not at the back, and **deletion** of elements with highest priority first. If several elements have same priority, the one that was placed first in the priority queue is removed first.



- We might think of implementing it as a queue. The class Node has to include also information about priority. However the `enqueue` and `dequeue` operations will need to be modified to take into account the priorities.

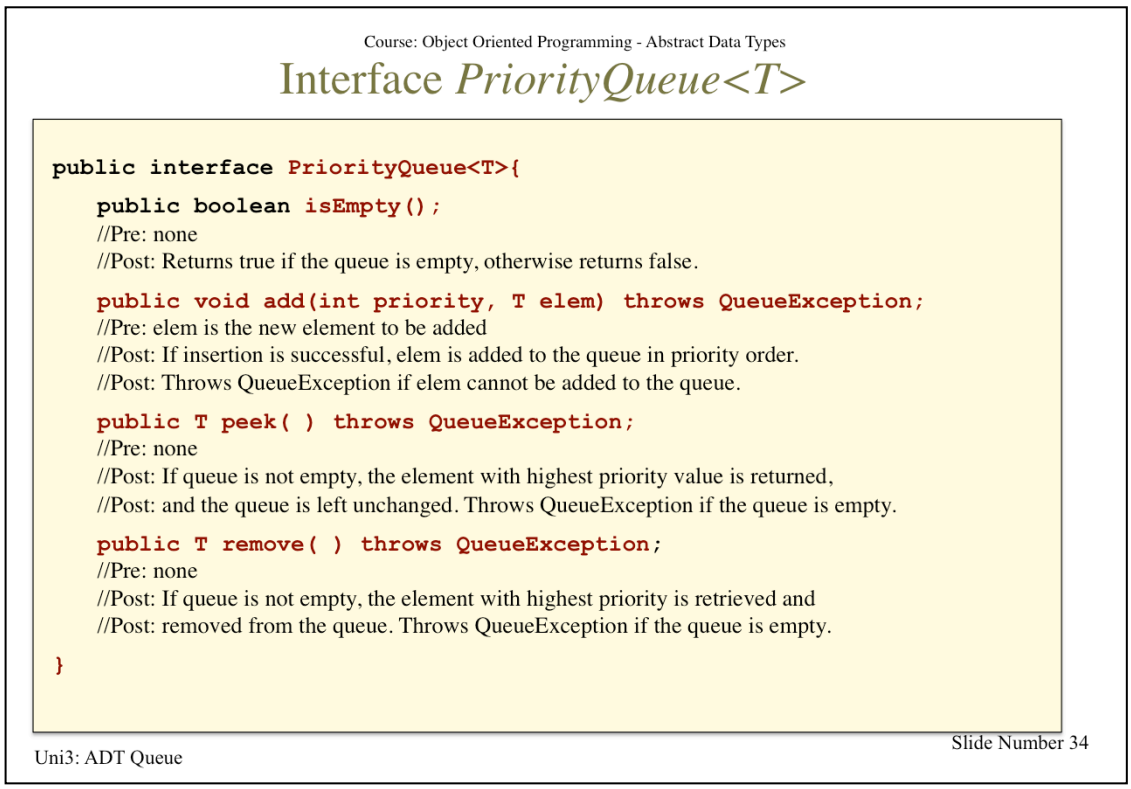
Uni3: ADT Queue

Slide Number 33

Using a queue ensures that customers are served in the exact order in which they arrive. However, we often want to assign priorities to customers and serve the higher priority customers before those with lower priority. For instance, a computer operating system that keeps a queue of programs waiting to use a printer, may give priority to short programs and lower priority to longer programs. The idea of prioritised services can be applied to any shared resource.

A "priority queue" is an ADT that stores items along with a priority for each item. Items are removed in order of priority.

We might think of implementing this specific ADT using similar technique to that used for the standard ADT queues. The class Node will have to be changed so as to include information about the priority of the individual item in the queue. In the same way the implementation of the operations "`enqueue`" and "`dequeue`" will have to take into account the priority value of the item to be inserted or deleted from the priority queue. This would however be very inefficient. We will see a better implementation of a priority queue when we will introduce the notion of Heaps.



This is an example interface for a Priority Queue ADT. Note that the queue access procedures `getfront()` and `dequeue()` have, here, be renamed `peek()` and `remove()` respectively, to distinguish them from the standard `dequeue()` and `getfront()` where no priority is considered.

We can have several options for implementing a class that implements the interface `PriorityQueue<T>`. Here we list some of these options.

1)We could use, as array-based implementation, a sorted array in order of priority value. The add operation in this case will have to first use a binary search to find the correct position for insertion in the array, and then shift the elements in the array to make room for the new item. On the other hand, the `peek()` and `remove()` access procedures would require only one access to the array as the element with highest priority will be at the beginning of the array.

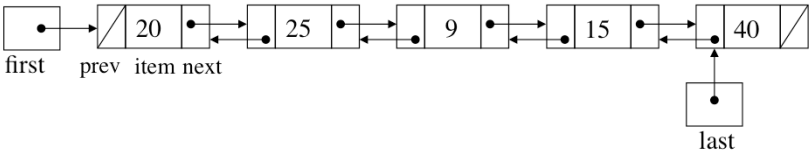
2) We can use an Ordered Linked List where the ordering in the LinkedList is on the priority level. Again the `peek()` method will consist of returning the head of the Linked List. The `remove()` access procedure will also require a constant time execution, as it will consist of just removing the head of the LinkedList. The `add()` method will instead be the `add` method of the LinkedList by priority level instead of by position. In the case of adding a new element to a priority queue this has to be `enqueued` at the end of any existing items with the same priority-level. To achieve this behaviour, the method add for OrderedLinkedList might need to be slightly modified.

3) There is a third better option for implementing Priority Queue. This makes use of a Heap ADT structure and we will see it later after we have defined the concept of an ADT Heaps.

The ADT Double-Ended Queue

Definition

- The **ADT Double-ended Queue** is a queue.
- The access procedures permit **addition** of items and **deletion** of items at both ends of the sequence.



- A possible dynamic implementation is to have a doubly linked list with two additional reference variables to refer to the first and last item in the queue.

Another particular form of queue is a Double-Ended queue, also called DEQueue. This is similar to a queue with the particular feature that any item can be added or deleted from the queue at either end. A diagrammatical representation is given in this slide.

A possible dynamic implementation is to use a double-linked list with the addition of two extra reference variables, called first and last, which refer to the first item and the last item in the queue. The list of access procedure for this special type of queues is given in the next slide.

Interface DEQueue<T>

```
public interface DEQueue<T>{
    public boolean isEmpty();
    //Post: Returns true if the queue is empty, otherwise returns false.
    public void addToBack(T elem) throws QueueException;
    //Post: Item is added at the end of the queue.  Throws QueueException if the item cannot be added.
    public void addToFront(T elem) throws QueueException;
    //Post: Item is added at the front of the queue. Throws QueueException if the item cannot be added.
    public T removeFront( ) throws QueueException;
    //Post: If queue is not empty, the item at the front of a queue is retrieved and removed
    //        from the queue. Throws QueueException if the queue is empty.
    public T removeBack( ) throws QueueException;
    //Post: If queue is not empty, the item at the back of a queue is retrieved and removed
    //        from the queue. Throws QueueException if the queue is empty.
    public T getFront( ) throws QueueException;
    //Post: If queue is not empty, the item at the front of the queue is retrieved without changing the
    //       queue. Throws QueueException if the queue is empty.
    public T getBack( ) throws QueueException;
    //Post: If queue is not empty, the item at the back of the queue is retrieved and removed
    //        from the queue. Throws QueueException if the queue is empty.  }
```


In Tutorial 3 you will be asked to implement this interface. Note that the underlying data structure is that of a double linked list of nodes with two references for the first and the last node in the linked structure respectively.

Queue as part of Java Collection

Java provides the interface `Queue<E>` that includes:

	Throws Exception	**Return Special Value**
getFront()	element()	peek()
enqueue(elem)	add(elem)	offer(elem)
dequeue()	remove()	poll()

Java provides many more access procedures as part of the interface `Deque<E>`.