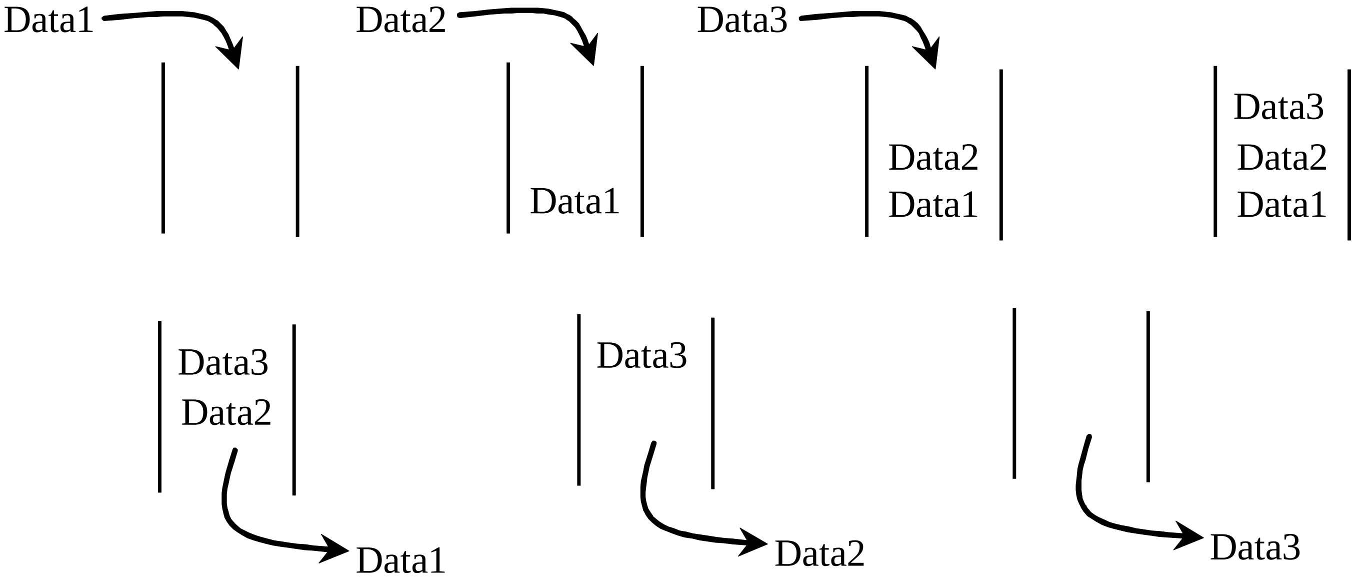
Chapter 20 - Part 4

Queues

Queues

A queue represents a waiting list. A queue can be viewed as a special type of list, where the elements are inserted into the end of the queue, and are accessed/processed/deleted from the beginning of the queue. Queue behavior is knows as First-In, First-Out characteristic (FIFO) .



Queue ADT Specifications

```
public QueueInterface {
public void enqueue(Object item) throws QueueException;
        //Precondition: item is the new item to be added.
        //Postcondition: If insertion is successful, item is at the end of the queue.
        //Postcondition: Throw QueueException if the item cannot be added to the queue.
public Object dequeue( ) throws QueueException;
        //Precondition: queue is not empty
        //Postcondition: If queue is not empty, the front item is retrieved and removed
        //Postcondition: Throws QueueException if the queue is empty.
public Object front( ) throws QueueException;
        //Precondition: queue is not empty
        //Postcondition: If queue is not empty, the front item is returned, queue left unchanged.
        //Postcondition: Throws QueueException if the queue is empty.
public int size( );
        //Precondition: none.
        //Postcondition: return queue size.
public boolean isEmpty();
        //Precondition: none
        //Postcondition: Returns true if the queue is empty, otherwise returns false.
public boolean isFull();
        //Precondition: none
        //Postcondition: Returns true if the queue is full, otherwise returns false.
}
```

Queue Operations

Basic stack operations:

```
enqueu(e);      // add new element to the end of the queue
dequeu();       // process and remove the front element from the queue
front();        // return (don't remove) the front element from the queue
size();         // return number of elements in the queue
isEmpty();      // return true if queue is empty; otherwise return false
isFull();       // return true if queue is full; otherwise return false
```

Basic Variables:

```
 Size;          // hold current size of the queue
 MAX_SIZE;      // hold max size of the queue
```

Queue Implementation Using Array

Since deletions are made at the beginning of the list, it is more efficient to implement a queue using a linked list than an array list. However, array may be used to implement queue, using either static or circular implementation.

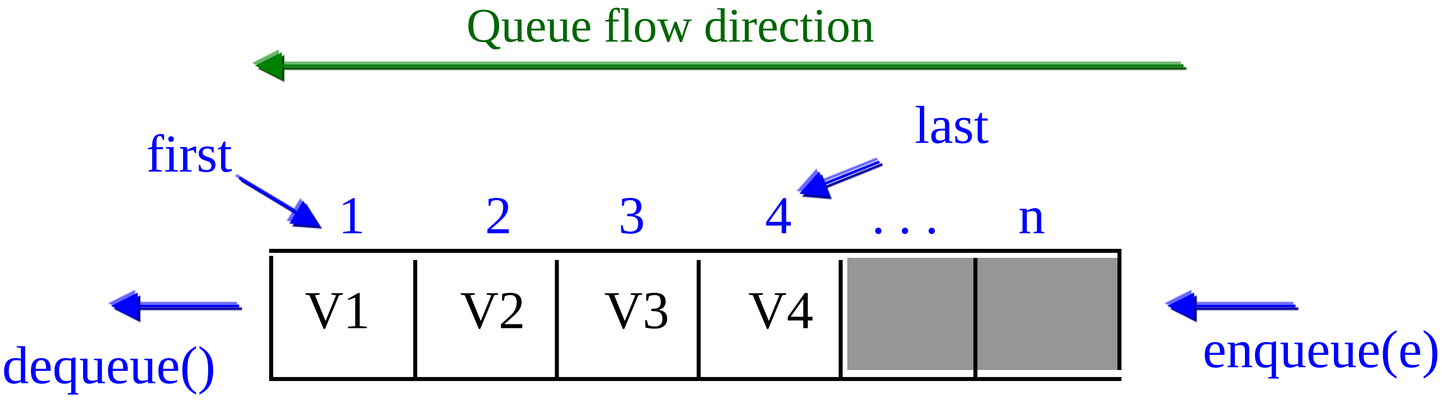
Static is using fixed-sized array and two variables “first” and “last”, which contain the index value of the first item and the last item, respectively. At the beginning “first= -1” and “last= -1” to indicate empty queue. Methods dequeue() requires moving (shifting) elements towards the front of the queue.

Circular is using the array where pointers "first" and "last" advance clockwise to dequeue the first item and add a new item respectively. Here, the values of "first" and "last" are calculated using modulo arithmetic operation (Java: % operator).

Static Implementation

```
enqueue(e):     increment pointer "last";
                add item into queue[last];

dequeue():      remove/serve first item;
                shift all items one position forward;
                adjust pointer "last";
```



Queue Implementation Using Array

```
//Pseudo code for queue Q
enqueue(e)
if Q is not full
    increment last by 1.
    store e in Q[last].
else Queue-Full-Error.

dequeu()
if Q is not empty
   loop for last-first times
      shift element forward.
   adjust pointer last.
else Stack-Empty-Error.

front()
if Q is not empty
   return first element.
else Stack-Empty-Error.
```

```
//Pseudo code for queue Q
size()
return size.

isFull()
if last >= MAX_SIZE
   return True.
else Return False.

isEmpty()
if size < 0
   return True.
else return False.
```

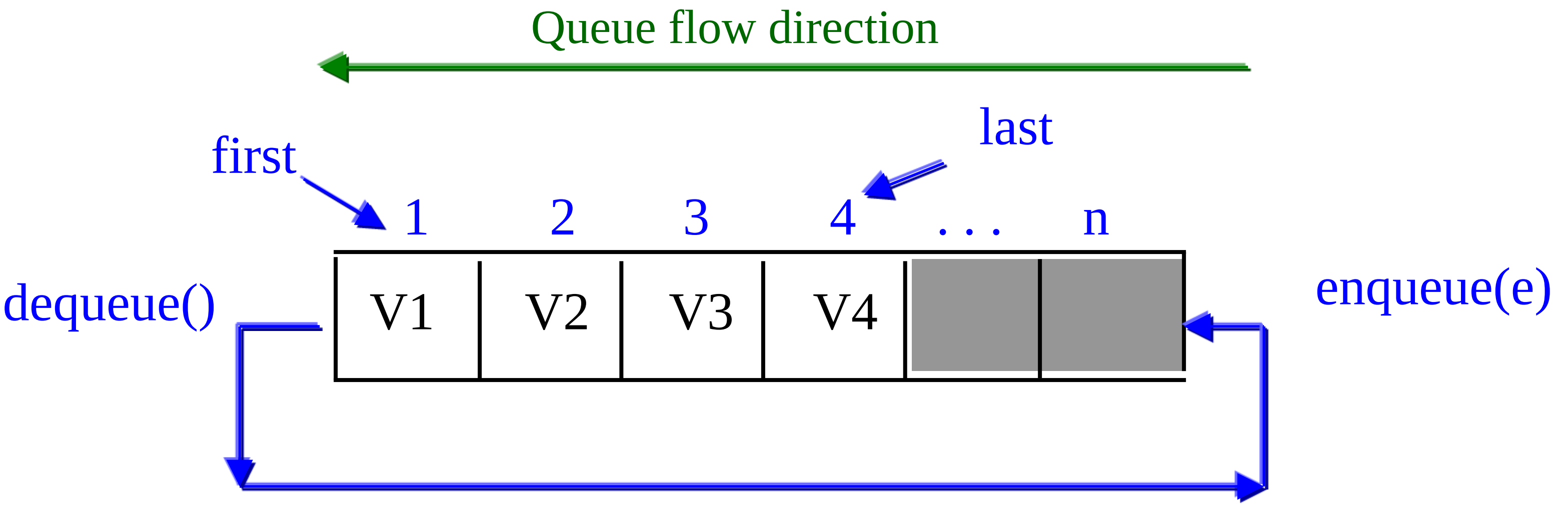
Circular Implementation

enqueue(e):

```
last = (last + 1) % MAX_SIZE;
queue[last] = newItem;
size++;
```

dequeue():

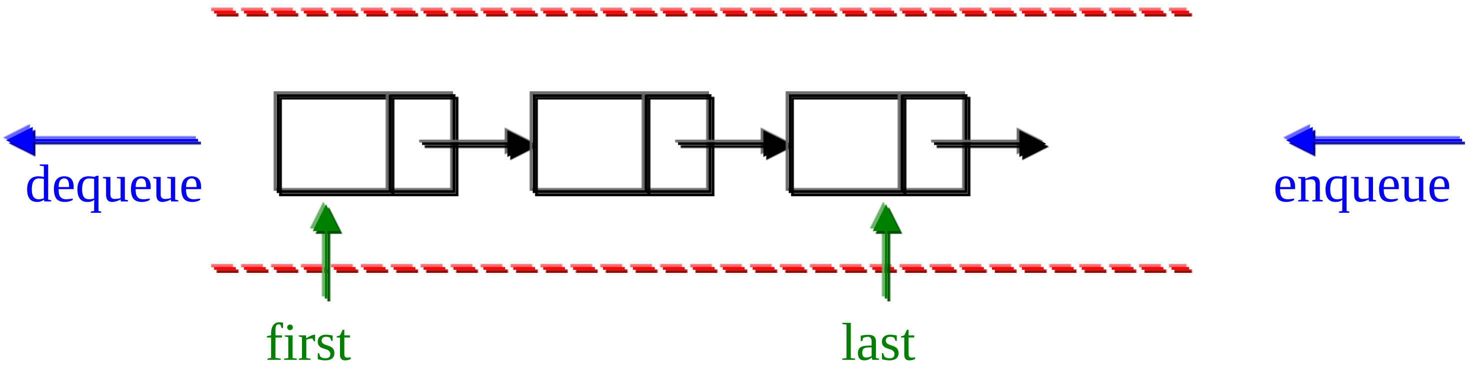
```
first = (first + 1) % MAX_SIZE;
size--;
```



Circular Implementation

```
//Java cod for circular array queue
public class QueueArray {
 private final int max_queue = 50;
 private Object items[]; //an array of list items
 private int first, last, count; //number of items in the list
 public QueueArrayBased(){ //default constructor
 items = new Object[max_queue];
 first = -1; count = 0; last = -1;}
 public void enqueue(Object newItem) throws QueueException {
 if (!isFull()){
    last = (last+1) % (max_queue);
    items[last]=newItem;
    count++;}
 else throw new QueueException(“Queue is full”); }
 public Object dequeue() throws QueueException {
 if (!isEmpty()){
    Object queuefront = item[first];
    first=(first+1)%(max_queue);
    count--;
    return queuefront;}
 else {throw new QueueException(“Queue is empty”);}
 public isFull(){
 return count == max_queue};
}
```

Queue Implementation as Linked List



dequeue

enqueue

first

last

```
enqueue(e);    // add element e to the queue
dequeue();     // remove first element from the queue
front();       // return (don't remove) the first element in the queue
size();        // return number of elements in the queue
isEmpty();     // return true if queue is empty
size;          // variable to hold current size of the queue
```

Method enqueue(e)

```
//Pseudo code - Method enqueue(e)
Method enqueue(E e)
{
  // 1. create and initialize a new bode
  Node<E> newNode = new Node<E>(e);

  // 2. check if empty queue
  if (first == NULL) {
       first = newNode;
       last  = newNode;
     }

  //3. Generic case, add to end of queue
  last.next = newNode;
  last = last.next;
}
```

Method dequeue()

```
//Pseudo code - Method dequeue()
Method dequeue()
{
// 1. check if empty queue
   if (size == 0) return null;

// 2. Generic case 1 or more nodes queue
    Node<E> current = first; //declare current and set to first
    first = first.next; //make first point to next node
    current.next = null; //set current.next to null
    size--; //decrease list size by one element
}
```

Methods front(), size(), and isEmpty()

```
//Pseudo code - Method fron()
Method front()
{
    return first.data;
}

//Pseudo code - Method size()
Method size()
{
    return size;
}

//Pseudo code - Method isEmpty()
Method isEmpty()
{
    return (size == 0);
}
```

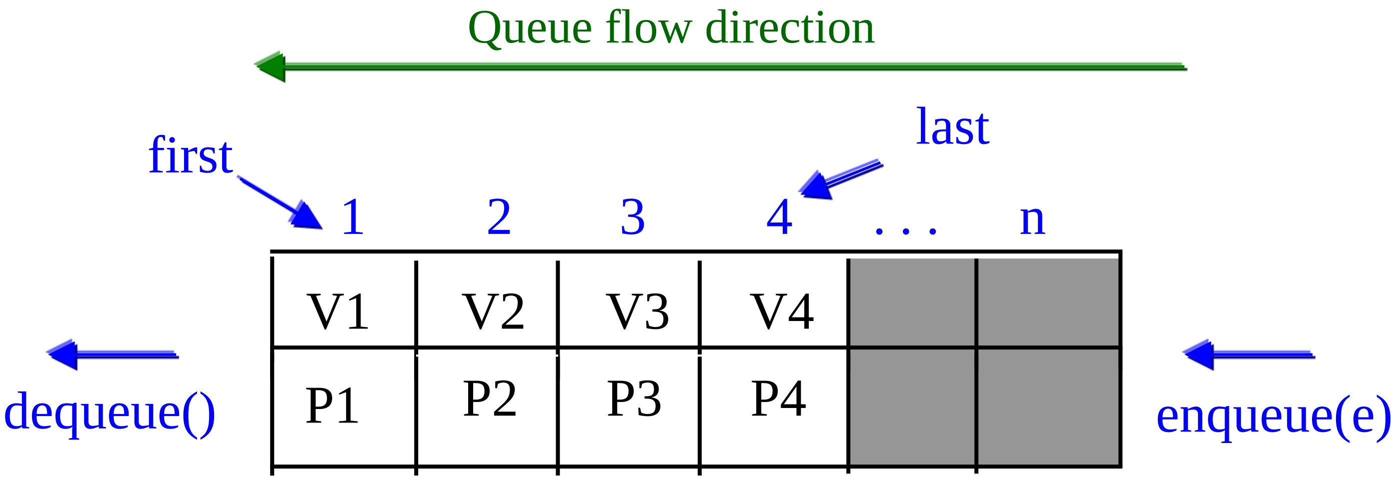
Special Queues

Special queue include:

- Priority Queue
- Double-Ended Queue
- Balking Queue

Priority Queue

In a priority queue, elements are assigned priority values. When accessing elements, the element with the highest priority is removed/processed first.



Queue flow direction

first

last

dequeue()

enqueue(e)

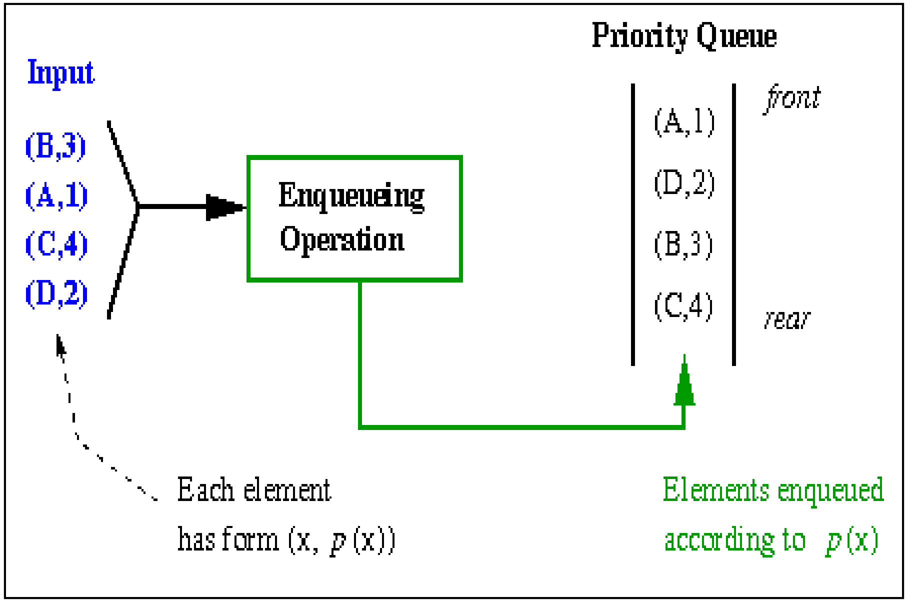
1	2	3	4	. . .	n
V1	V2	V3	V4		
P1	P2	P3	P4		

For example, a doctor's office or emergency room in a hospital would assign patients priority numbers based the severity of the patient's condition.

Priority Queue

Implementation: keep the queue sorted by the priority value.

Array Implementation:
To keep the array sorted by priority value, it requires shifting when adding a new element to the queue. Not efficient.



Linked-List Implementation:
Keep the element with the highest priority at the head of the queue and with the lowest priority at the end of the queue. Requires no shifting, but searching for the correct position.

Double-Ended Queue (Dqueue)

A Dequeue (not dequeue) is a queue structure that allows elements to be added and removed from both ends of the structure.

For example, in a print queue, small printing jobs are kept on one side of the queue while large printing jobs are kept on the other side of the queue.

Note: the specification of dequeue and enqueue operations require additional argument to specify the **end** at which the operations to be applied.

Balking Queue

A Balking queue is a queue such that each data element has a unique key value. The key is used for searching and manipulating the queue content.

A Balking queue has additional operations that allow removing an element from a given valid position in the queue.

Balking queues are used with database applications where the key is important data element.

A Balking queue is best implement using Linked List.

FYI (from Queuing Theory):

balking: customers deciding not to join the queue if it is too long.

reneging: customers leave the queue if they have waited too long for service.

Jockeying: customers switch between queues if they think they will get served faster by so doing.

End of Slides