- intro to queues
- simple dynamic implementation
- full implementation
- introduce next lab

Online Midterm exam is next week

- full details in 'About the Midterm exam' at the end of this doc

Next week and homework

- next week: Midterm exam
- homework by next week:
  - download, run and understand this week's example programs
  - to prepare for Midterm exam: re-read all my lecture notes and my example programs
  - review all of the programs you have written
  - re-read Horstmann textbook, Chapters 13, 15, 16, 18

Lab

- about the Midterm exam
- next lab is assigned. See Canvas for due date

Review last week

- finished looking at recursion

Introduction to this week

- cover the next major data structure
  - will begin queues this week
  - assign queue lab, so we can work with queues

Intro to queues
Objective: review queues, see some primitive operations

A queue is an "ordered set of items where we remove from the front and insert at the rear, only"

- e.g. like the line at the cafeteria, bank, cashier's office

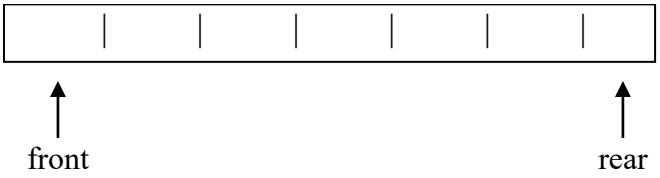


Figure 1 illustration of a queue

- compare against a stack:
  - stack is: Last In First Out (LIFO)
  - queue is: First In First Out (FIFO)
- as an elaboration on the basic idea, can have a priority queue, where "more important items handled more quickly" e.g.
  - student print jobs queued on the networked laser printer – strictly FIFO
  - then an instructor print job comes along – a higher priority, could be moved directly to the front of the queue :)!!

Queue primitives

- some appropriate primitive operations:
  - q.insert(x) inserts item x to rear of queue q
  - x = q.remove() remove item from front of queue q and directly return it
    beware underflow!
  - x = q.query() returns item at front of queue q without changing q
  - q.isEmpty() returns true if q is empty
  - q.clear() remove all items from q

Summary

- a queue is a FIFO data structure that everyone is familiar with from real life
- many applications in real life and in programming
- will now begin to implement

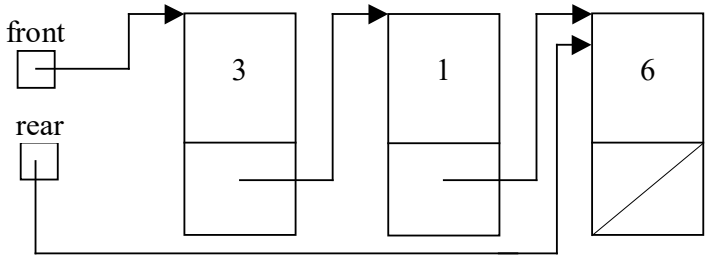
Simple dynamic implementation
Objective: design data structure methods using pictures, then implement in Java!

Draw pictures to design data structures!

- as we already saw with stacks, drawing is the best way to design data structures!
  - so that you focus on what to do and how to do it
  - ignoring trivial syntax
  - more examples here of how to do it…

Begin design of queue implementation

- we're going to implement queues dynamically here, using a single linked list. Is a truly beautiful thing, e.g.



  - maintain 2 references `front` and `rear`
  - `front` is next element to be removed
  - `rear` is last element inserted
- do simplest possible implementation first
  - then make more general later, same as we did for stack
- packaged of course into a `Queue` class
  - will use the familiar `Item` class, to give a queue of items:

```
public class Queue
{
    private Item front;
    private Item rear;

    public Queue()
```

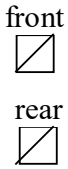
```
{
    front = null;
    rear = null;
}
```

– see how the constructor creates an empty queue

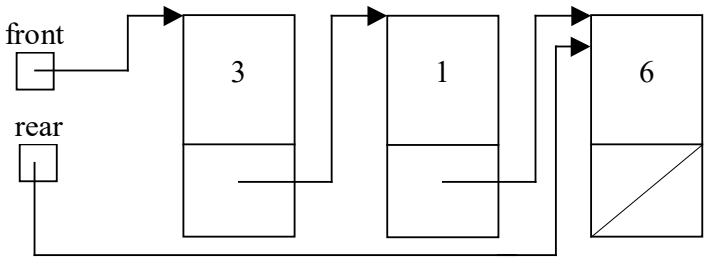
`insert()` adds new item at rear

- design by drawing first. Here it is. Turns out there are two cases to handle

  – if the queue is empty, have to update front and rear:



  – otherwise stick the new item to the end of the queue and update rear



- now use the pictures to figure out steps to get to the desired end state

  – e.g. insert new item containing 9 at the rear of these queues

  – using the pictures, we come up with an algorithm something like this, written as pseudocode. (Update the pictures as you work through the steps):

    create a new item
    initialize it
    update reference to the new item
    update rear reference

- now ready to turn the design into Java. So something like:

```
public void insert(int i)
```

```
{
    Item item = new Item(i);

    //queue is empty
    if (isEmpty())
        front = item;
    //queue is not empty
    else
        rear.next = item;
    rear = item;
}
```

- will do `isEmpty()` shortly

`remove()` removes item from front

- draw your pictures. Two cases again:
  - if the queue is empty, will just report an error and quit for now
  - otherwise remove the item from the front of the queue and update front
- from the pictures we come up with something like this, in pseudocode:

  if queue is empty
    report error and exit
  retrieve info from front of queue
  update front reference

- then turn the design into Java. Something like:

```
public int remove()
{
    int temp = -999;    //initialize to unusual value
    if (isEmpty()) {
        System.out.println("Error in remove() - queue is empty");
        System.err.println("Error in remove() - queue is empty");
        System.exit(1);
    }
    else {
        temp = front.info;
        front = front.next;
    }
    return temp;
}
```

  - as usual, the compiler requires a guaranteed return value. So I just hard coded to an unusual value

`isEmpty()` returns true if the queue is empty

- there's a subtle problem here! Turns out that the best definition of empty is when `front` is `null`
  - subtle reason for this is that `remove()` updates only `front`, since we remove from the front of a queue
  - so when we remove the last item from a queue, `front` is guaranteed to be set to `null`…
  - whereas `rear` is actually left still pointing to the last item and is not `null`, even though the queue has just been emptied
- so this gives simply:

```
public Boolean isEmpty()
{
    return front == null;
}
```

Summary

- begin here with a very simple dynamic implementation of a queue, to show clearly all the most important design then programming details
- saw the important programming technique to design data structures using pictures!
  - essential, particularly when data structures will get much more complex
- from Canvas, 'Queues' module, Example programs, download, read, run and understand my `Simple queue` example program

Full implementation
Objective: build on this simple beginning, to a full dynamic implementation

Add features to the simple beginning

- as we did for stacks, will now add many features to this simple beginning, to make a more realistic queue
  - use interface
  - make generic
  - add exceptions
- followed along from when we added these features to the simple stack back in 'Dynamic implementation of stack' module
  - very straightforward, nothing surprising

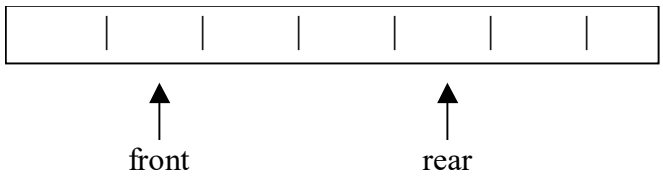
Summary

- from Canvas, 'Queues' module, Example programs, download, read, run and understand my `Queue` example program
  - see in `Tester::main()` that I tested the generic feature by using a queue of `Character` for a change
  - tested underflow exception handling, by deliberately removing from an empty queue
  - is excellent!

Introduce next lab
Objective: in the next lab you will implement queue using a 'circular array'!

Begin implementing queue using a simple array

- will now begin to re-implement queue, this time statically using an array. Will start with a very simple array of `int` and 2 indexes:



  - `front` is next element to be removed
  - `rear` is last element inserted

- packaged into a class:

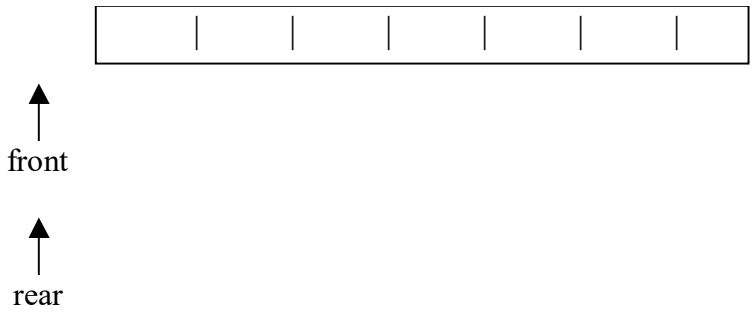
```
public class Queue
{
    public static final int MAX = 10;

    private int elements[];
    private int front;
    private int rear;

    public Queue()
    {
        elements = new int[MAX];
        front = -1;
        rear = -1;
    }
```

  - note that this simple array implementation does not use the `Item` class. Instead all the information is stored in the array, named `elements` here, and hardcoded for `int`
  - unlike the linked list implementation, an array has a fixed, limited size. Named `MAX` here
  - see how the constructor is careful to allocate memory for the array
  - then initializes `front` and `rear` to `-1`

- let's make a first attempt at `insert()` and `remove()`, to discover a big problem
  - here's a new `Queue` object, `front` and `rear` initialized to `-1`



`insert()` adds new element at rear

- `rear` is last element inserted, so we must increment `rear` first, e.g.

```
public void insert(int x)
{
   if (isFull())
         error("Queue overflow");
   rear = rear + 1;
   elements[rear] = x;
}
```

  - NOTE: will look at `isFull()` shortly – assume we have one that works
  - also, some `error()` method, to report a problem appropriately then quit

`remove()` removes element from front

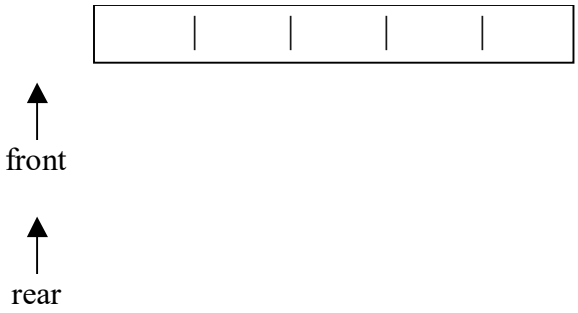
- `front` is the next element to be removed, so increment `front` after the removal, e.g.

```
public int remove()
{
   if (isEmpty())
         error("Queue underflow");
   int temp = elements[front];
   front = front + 1;
   return temp;
}
```

  - NOTE: minor problem here on first removal, because `front` was initialized to –1
  - could fix by instead initializing `front` to 0, or by adding

```
if (front == -1)
      front = 0;
```

- then this pair of insert() and remove() kind of works…
  - e.g., assuming array of only 5 elements. Draw this starting picture, then update it as you carefully work through these lines of code:

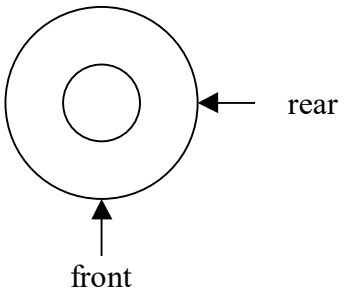



```
Queue q = new Queue();
q.insert(10);
q.insert(20);
q.insert(30);
int x = q.remove();
x = q.remove();
q.insert(40);
q.insert(50);
```

- …and see the big problem here is that the queue slowly moves through the array!
  - so if the next operation is:

```
q.insert(60);
```
←fails now, because at end of array

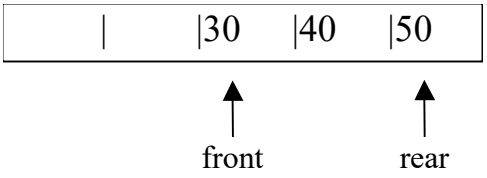
  - but we have elements at beginning of array we could use!!!
  - the fix is conceptually to make the array <u>circular</u>, so the queue can crawl around it happily!



- will look next at how to implement this idea of a circular array…

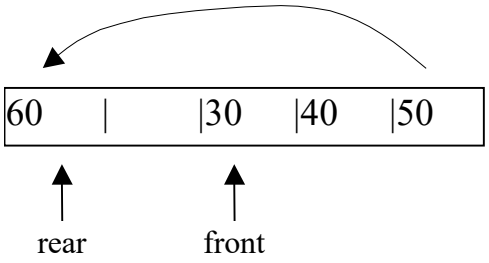
Now implement queue using a circular array

- here's the problem again with our first implementation of queue, above:



  - now do `q.insert(60)`
  - fails, with queue overflow error

- but we have unused elements before `front`
  - so a good solution is to wrap the queue around inside the array:



- sketch how we would change `insert()` to wrap around, but there's a problem with this:
  - insert `x` into the queue, wrapping around in the circular array

```
…
if (rear == MAX - 1)
   rear = 0;  //wrap queue round to beginning of array
else
   rear = rear + 1;
elements[rear] = x;
…
```

  - (would also have to change `remove()` along similar lines)

- but the problem with this wrapping is that it is now difficult to tell when a queue is empty or full!

  – and any implementation must prevent:

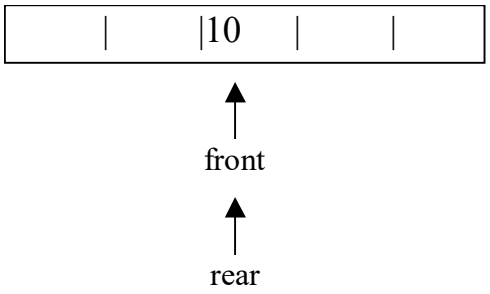
`remove()` from an empty queue ← this is an underflow error

`insert()` to a full queue ← this is an overflow error

Simplest definitions of empty and full

- in the simplest case of no wrapping, here's a definition of an empty queue

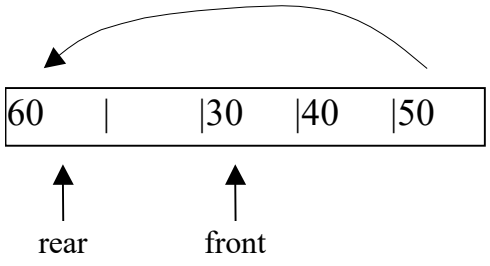
  – start with a queue containing 1 element, which is to be removed to leave the queue empty, e.g.:




```
int x = q.remove();
```

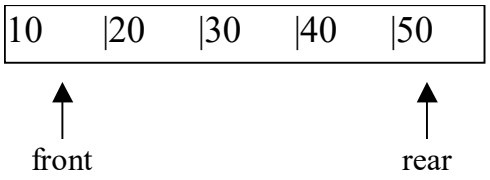
- this increments `front`, so maybe queue is empty when `front > rear`?…

  – …but this is not correct with wrapping, e.g. this wrapped queue is definitely not empty



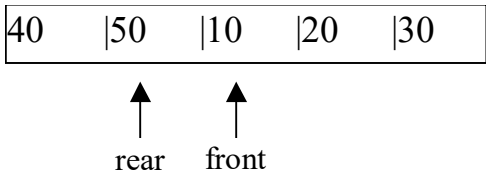
- same sort of problem when defining a full queue for the simplest case of no wrapping

  – e.g. here's a full queue, to which we try to insert a new element:



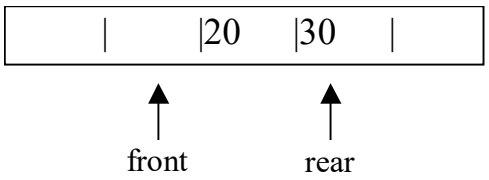

```
q.insert(60);
```

- so maybe queue is full when `rear == MAXQUEUE - 1`?…
  - …again, not correct with wrapping e.g. here's an example of a full queue with wrapping:



Solving the problem of empty and full for circular arrays

- to get a nice definition of empty for a circular array, change our definition of front:
  - front now points to "location before next item to be removed" e.g.



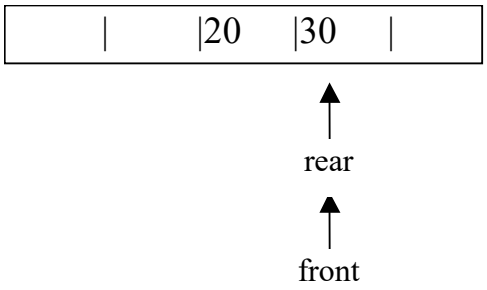
- must now change `remove()` with this new definition of front. In pseudocode:

  check first for empty, report any underflow and quit
  update front, with wrapping
  do removal

- so remove the 2 elements from the queue above:

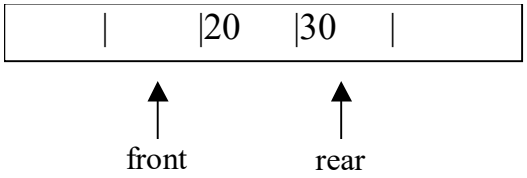
```
q.remove();
q.remove();
```

  - would give an empty queue that looks like this:



  - (BTW, notice here there is no requirement to re-initialize an element in the array after a removal)

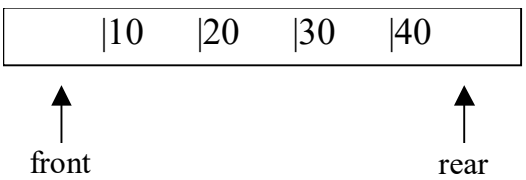
- this gives the new definition of empty, that works with wrapping:
  - "queue is empty if `front == rear`"
- however, with wrapping, this creates a problem when defining full…
  - here's an example wrapped queue containing 2 elements:



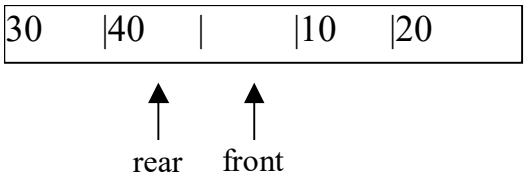
  - now insert 3 more elements to deliberately make the queue full:

```
q.insert(40);
q.insert(50);
q.insert(60);
```
←here `front == rear`, yet queue is full, not empty!!!

- to fix this, redefine full:
  - "queue is full if, were rear to be updated, front == rear"
  - then there are 2 cases to handle – when the queue is full and not wrapped e.g.



  - and when the queue is full and wrapped e.g.

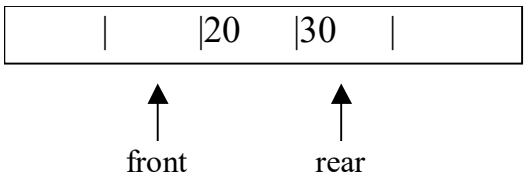


– see how the new definition of full works correctly for both these cases

- we must check for full first during every `insert()`. In pseudocode:

  check first for full, report any overflow and quit
  update rear, with wrapping
  insert element

- now try the example again:




```
q.insert(40);
q.insert(50);
q.insert(60); ←can't do this, because queue is full by new definition
```

– NOTE: this implementation sacrifices a location in the array to give good definitions of full and empty that work with wrapping

- finally, this changes the initialization of `front` and `rear` in the constructor to `MAX - 1`:

```
public Queue()
{
    elements = new int[MAX];
    front = MAX - 1;
    rear = MAX - 1;
}
```

– this has `front == rear`, to fit the definition of empty

– and nicely forces the first `insert()` into index 0

Summary

- will implement queue using a circular array for the next lab

Next week and homework

- next week: Midterm exam

- homework by next week:

  - download, run and understand this week's example programs

  - to prepare for Midterm exam: re-read all my lecture notes and my example programs

  - review all of the programs you have written

  - re-read Horstmann textbook, Chapters 13, 15, 16, 18

Lab

- about the Midterm exam
- next lab assigned. See Canvas for due date

About the Midterm exam
Objective: what to expect, how to prepare, to do super-well on the Midterm exam please!

Have a DRC Accommodations Request Form?

- YOU MUST email me ASAP (awsmith@palomar.edu) if you have a DRC Accommodations Request Form

Format of the online Midterm

- I will post the 'Midterm' module on Friday, then email everyone as usual. You take the Midterm exam online through Canvas, any time during the week. Due date will be given in Canvas. You must submit the exam before the end of that day
- will be 25 multiple-choice questions
- strict 1 hour time limit once you start, cannot be exceeded
- you have to finish the exam once you've started. Cannot stop part way through and come back later
- must be your own independent work – do not collaborate or confer in any way
- the exam has to be taken inside a secure browser. Set-up instructions are given when you start the exam. Any set-up time does not count against the exam time limit
- is 15% of your overall course grade

Preparation required so far

- every week you are required to print out then work through my written pdf lecture document, running and understanding my example programs where indicated
- you have done the assigned textbook reading
- you have completed the programming exercises and homeworks
- you have completed the graded programming labs
- if you have not done all this, you are not likely to do well on the exams
- if necessary, review the withdrawal deadline given in the 'Syllabus' doc in 'Course information'

How to prepare

- the questions cover everything up to and including this week's material
- to prepare, first review my lectures, my example programs and your labs in detail
  - every week, you have to print out my lecture, and study all of my example programs...
  - make sure you re-read all these lectures
  - make sure you understand every example program
- if you have time, then re-read Horstmann textbook, Chapters 13, 15, 16, 18
  - review Horstmann's 'Self Check' questions
  - (answers are given at the end of each Chapter)

Practice Midterm now posted

- I've posted a Practice Midterm in the 'Queues' module...
  - is intended to show you how the real Midterm will work...
  - contains only 5 questions – the real Midterm has 25
  - also, enables you to set-up and use the secure browser environment before the real Midterm. (BTW, the use of cameras is not required)
  - see Canvas for due date. Submit the Practice Midterm before the end of that day, to familiarize yourself with the process
- the scores on the Practice Midterm will be available in Grades by the end of the day after the deadline
- importantly, the Practice Midterm has absolutely no effect on your overall course grade!