Queues

This lesson is borrowed from the following:

Reference

CS 367 – Introduction to Data Structures

http://pages.cs.wisc.edu/~mattmcc/cs367/notes/Queues.ppt

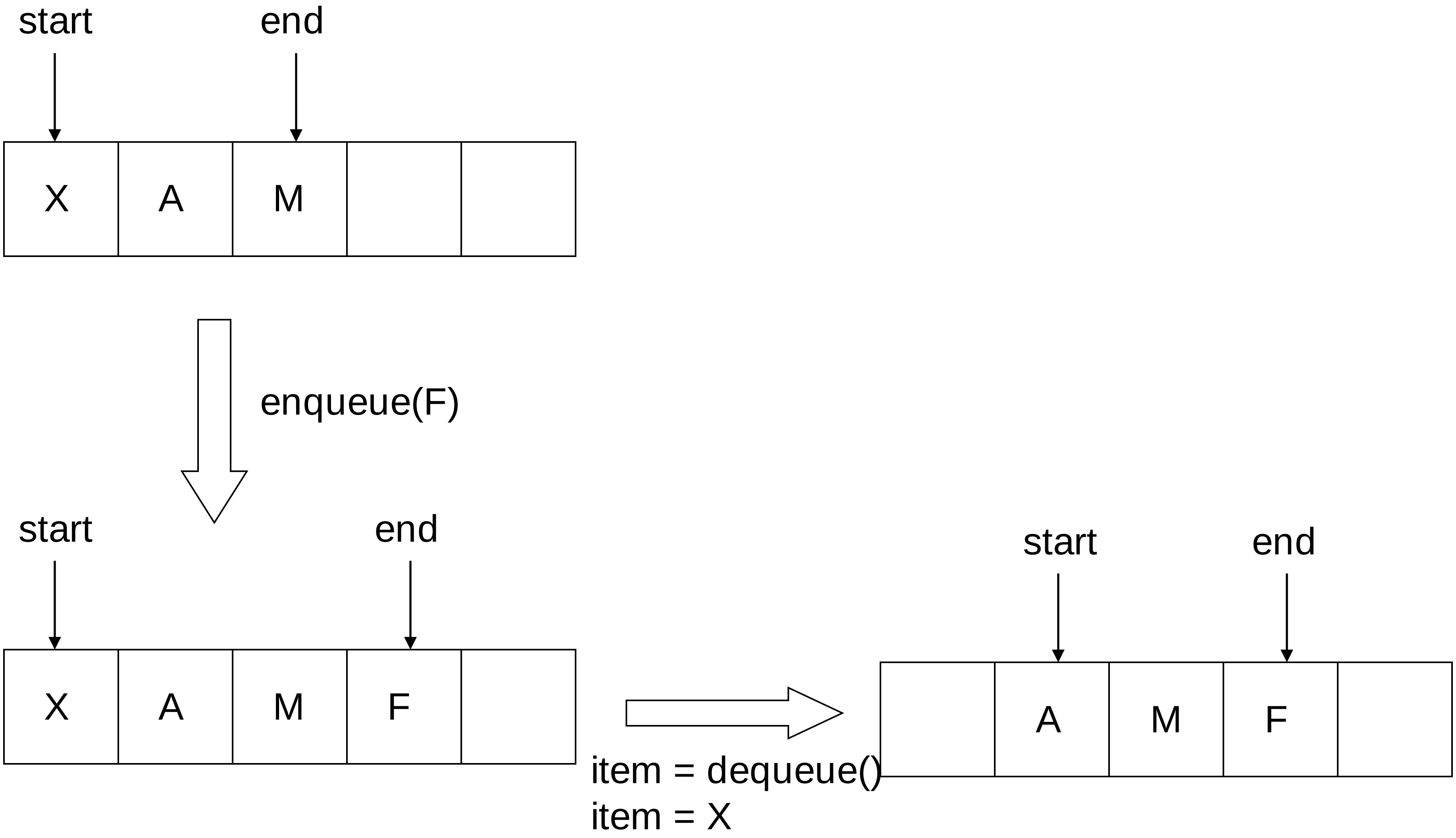
Queue

- A queue is a data structure that stores data in such a way that the last piece of data stored, is the last one retrieved
  - also called First-In, First-Out (FIFO)
- Only access to the stack is the first and last element
  - consider people standing in line
    - they get service in the order that they arrive

Queues

- *Enque*
  - operation to place a new item at the tail of the queue
- *Dequeue*
  - operation to remove the next item from the head of the queue

Queue


start
end
X
A
M
enqueue(F)
start
end
X
A
M
F
item = dequeue()
item = X
start
end
A
M
F

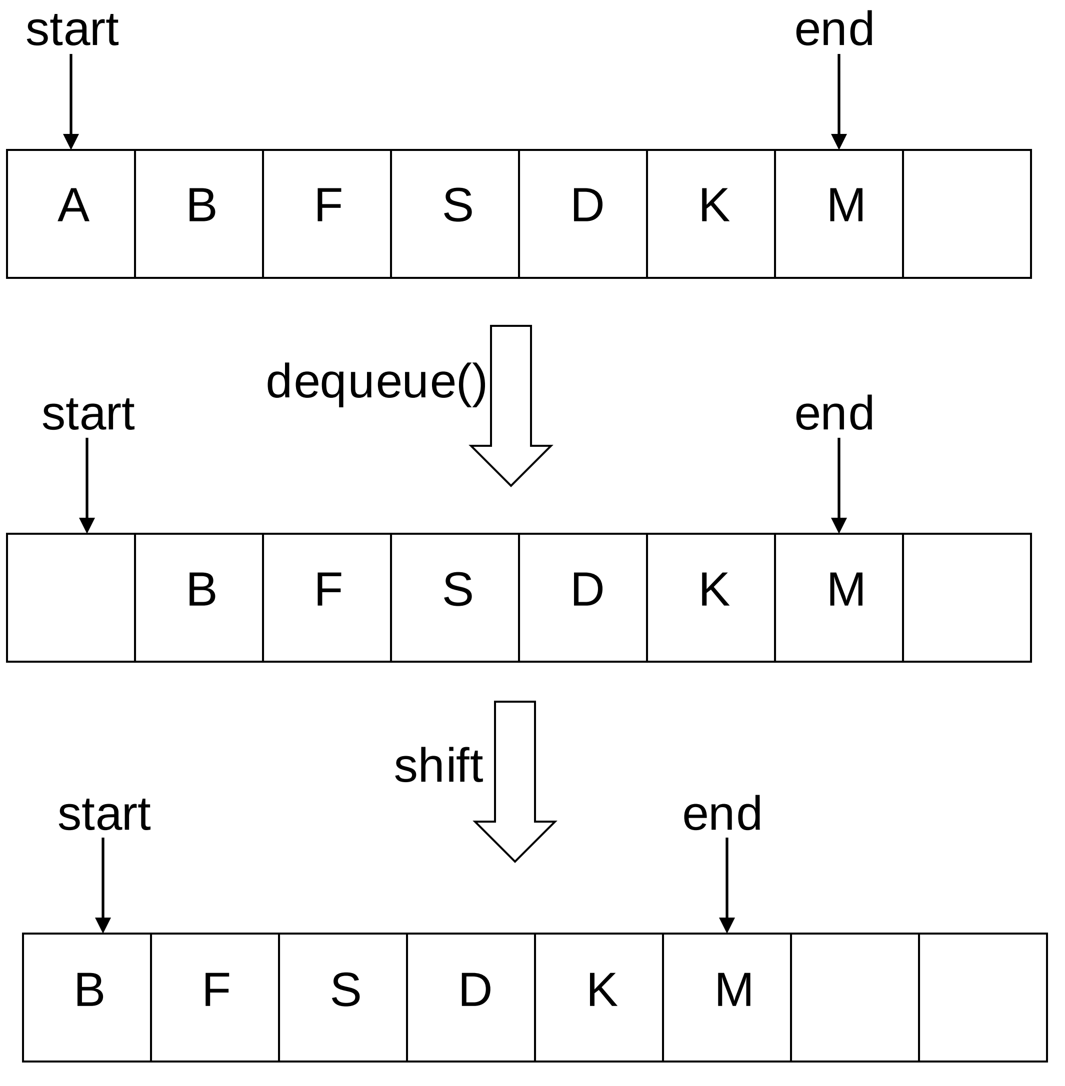
Implementing a Queue

- At least two common methods for implementing a queue
  - array
  - linked list
- Which method to use depends on the application
  - advantages? disadvantages?

Regular Linear Array

- In a standard linear array
  - 0 is the first element
  - array.length – 1 is the last element
- All objects removed would come from element 0
- All objects added would go one past the last currently occupied slot
- To implement this, when an object is removed, all elements must be shifted down by one spot

Regular Linear Array



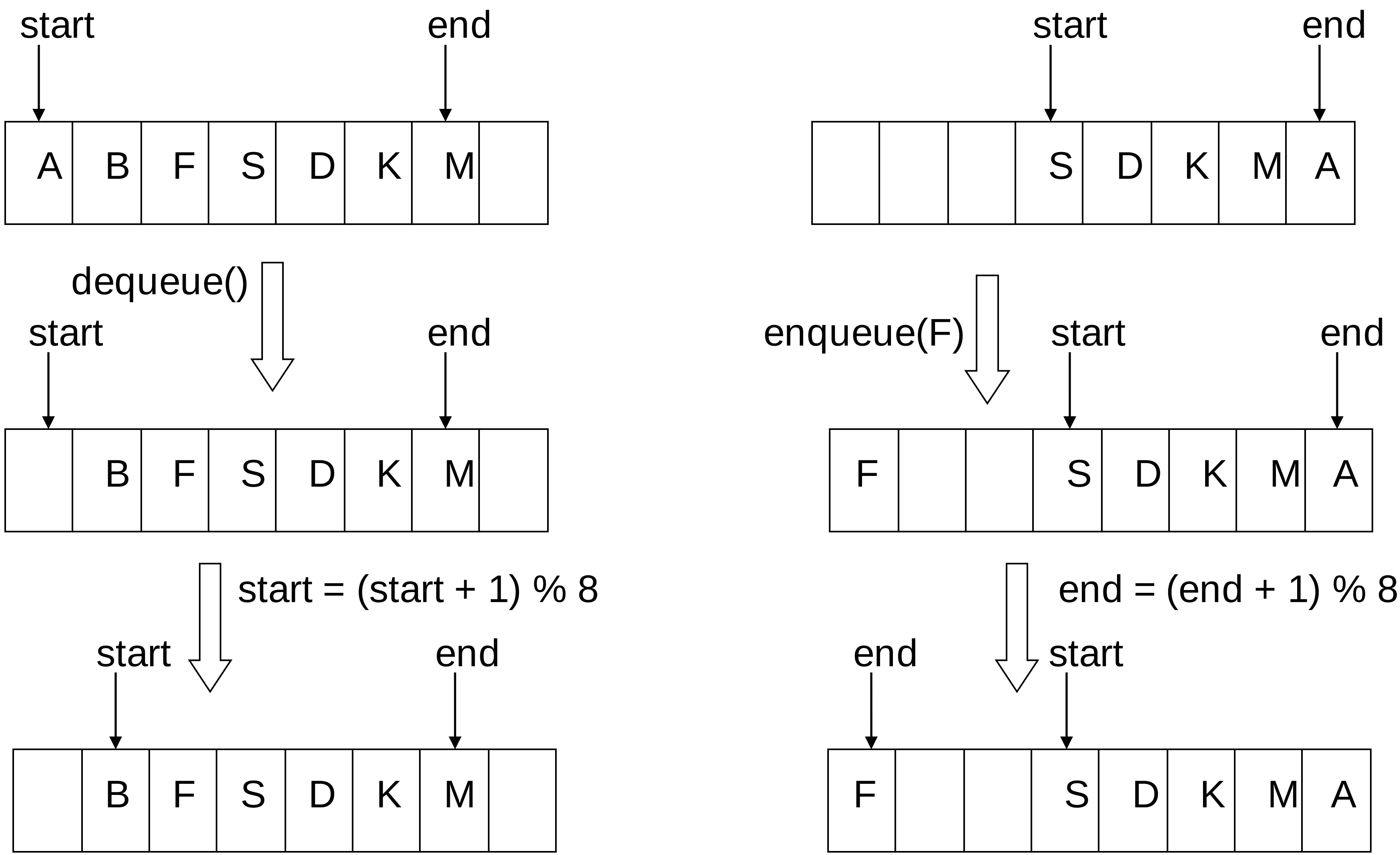
Regular Linear Array

- Very expensive structure to use for a queue
  - shifting all the data down by one is very time consuming
- Would prefer to let the data “wrap-around”
  - the start would not always be zero
    - it would be the first occupied cell
  - the last item may actually appear in the array before the first item
  - this is called a *circular array*

Circular Array

- Need to keep track of the index that holds the first item
- Need to keep track of the index that holds the last item
- The “wrap-around” is accomplished through the use of the *mod* operator (%)
  - index = (end + 1) % array.length
    - assume array.length = 5 and end = 4
    - then: index = (4 + 1) % 5 = 0

Circular Array



Circular List

- If the list is empty, *start* and *end* would refer to the same spot
- If the list is full, *start* and *end* would refer to the same spot
- How do you tell the difference between an empty and a full list?
  - if the list is empty, make *start* and *end* refer to -1
  - then if *start* and *end* both refer to an element greater than -1, the list is full

Implementing Queues: Array

- Advantage:
  - best performance
- Disadvantage:
  - fixed size
- Basic implementation
  - initially empty circular array
  - two fields: *start* and *end*
    - where the next data goes in, and the next data comes out
  - if array is full, *enqueu()* returns false
    - otherwise the data is added to the queue
  - if array is empty, *dequeue()* returns null
    - otherwise removes the start and returns it

Queue Class (array based)

```
class QueueArray {
    private Object[ ] queue;
    private int start, end;
    public QueueArray(int size) {
        queue = new Object[size];
        start = end = -1;
    }
    public boolean enqueue(Object data);
    public Object dequeue();
    public void clear();
    public boolean isEmpty();
    public boolean isFull();
}
```

enqueue() Method (array based)

```
public boolean enqueue(Object data) {
    if(((end + 1) % queue.length) == start)
            return false;   // queue is full

    // move the end of the queue and add the element
    end = (end + 1) % queue.length;
    queue[end] = data;
    if(start == -1) { start = 0; }
    return true;
}
```

dequeue() Method (array based)

```
public Object dequeue() {
    if(start == -1)
        return null;   // empty list

    // get the object, update the start, and return the object
    Object tmp = queue[start];
    if(start == end)
        start = end = -1;
    else
        start = (start + 1) % queue.length;
    return tmp;
}
```

Notes on *enqueu()* and *dequeue()*

- Just implementing a circular list
  - if start and end equal -1, the list is empty
  - if start and end are the same and not equal to -1, there is only one item in the list
  - if the end of the list is one spot "behind" the start of the list, the list is full
    - if(((end + 1) % queue.length) == start) { … }
  - always remove from the start and add to the end
    - make sure to move *end* before adding
    - make sure to move *start* after removing

Remaining Methods (array based)

```
public void clear() {
    start = end = -1;
}

public boolean isEmpty() {
    return start == -1;
}

public boolean isFull() {
    return ((end + 1) % queue.length) == start;
}
```

Implementing a Stack: Linked List

- Advantages:
  - can grow to an infinite size
  - lists are well suited for implementing a queue
    - makes things very easy
- Disadvantage
  - potentially slower than an array based queue
  - can grow to an infinite size
- Basic implementation
  - add new node to the tail of the list
  - remove a node from the head of the list

Queue Class (list based)

```
class QueueList {
    private LinkedList queue;
    public QueueList() {
        queue = new LinkedList();
    }
    public boolean enqueue(Object data)   { list.addTail(data); }
    public Object dequeue()   { return list.deleteHead(); }
    public void clear()   { list.clear(); }
    public boolean isEmpty()   { return list.isEmpty(); }
}
```

Additional Notes

- It should appear obvious that linked lists are very well suited for queues
  - *addTail()* and *deleteHead()* are basically the *enque()* and *dequeue()* methods, respectively
- Our original list implementation did not have a *clear()* method
  - all it has to do is set the *head* and *tail* to null
- Again, no need for the *isFull()* method
  - list can grow to an infinite size

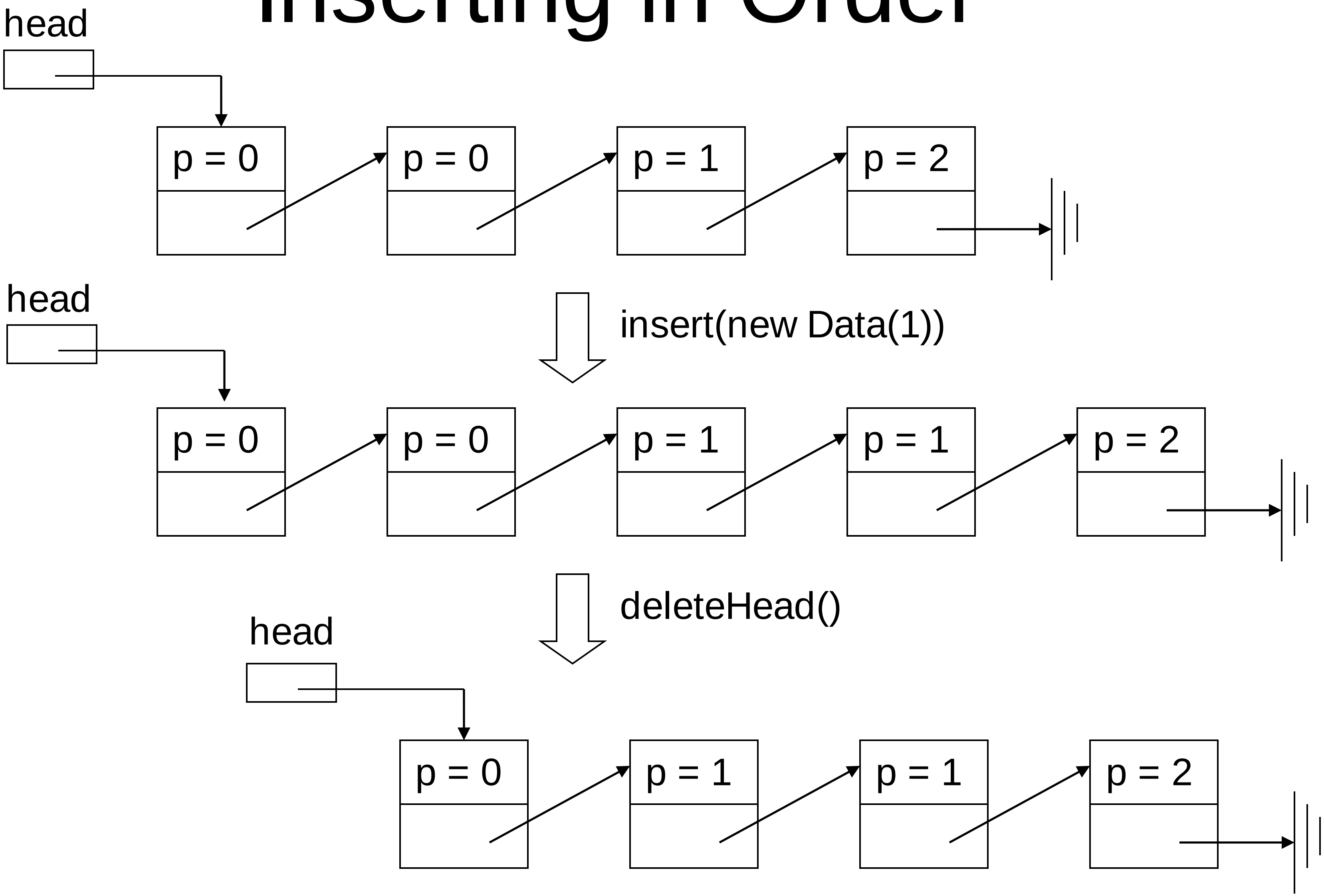
Priority Queue

- Sometimes it is not enough just do FIFO ordering
  - may want to give some items a higher priority than other items
    - these should be serviced before lower priority even if they arrived later
- Two major ways to implement a priority queue
  - insert items in a sorted order
    - always remove the head
  - insert in unordered order and search list on remove
    - always add to the tail
  - either way, time is O(n)
    - either adding data takes time and removing is quick, or
    - adding data is quick and removing takes time

Inserting in Order

- Use the very first linked list class shown
  - only need to use the *add()* and *removeHead()* methods
    - *add()* method puts things in the list in order
  - the *compareTo()* method (implemented by your data class) should return a value based on priority
    - usually consider lower number a higher priority
  - Performance
    - O(n) to add
    - O(1) to remove

Inserting in Order



Queue Applications

- As with stacks, queues are very common
  - networking
    - routers queue packets before sending them out
  - operating systems
    - disk scheduling, pipes, sockets, etc.
  - system modeling and simulation
    - queueing theory
- Any of these queues can be done as a priority queue
  - consider a disk scheduler
    - higher priority is given to a job closer to the current position of the disk head
    - next request done is that closest to the current position

Disk Scheduler

- Requests for disk sectors arrive randomly
- Disk requests are completed at a much slower rate than disk requests arrive
  - need to place waiting jobs in a queue
- All requests should be placed in a priority queue
  - jobs closest to the current position get placed closer to the front of the queue
- When the current job finishes, the next job is removed from the head of the queue

Code for a Priority Queue Class

```
class QueuePriority {
    private LinkedList queue;
    public Queue() { queue = new LinkedList(); }
    public void enqueue(Object data) { queue.add(data); }
    public Object dequeue() { return queue.deleteHead(); }
    public void clear() { queue.clear(); }
    public boolean isEmpty() { return queue.isEmpty(); }
}
```

Code for a Disk Request Class

```
class DiskRequest implements Comparable{
    private int cylinder;
    private int head;
    private int sector;
    public DiskRequest(int cylinder, int head, int sector) {
        this.cylinder = cylinder;
        this. head = head;
        this.sector = sector;
    }
    public int getCylinder() { return cylinder; }
    public int getHead() { return head; }
    public int getSector() { return sector; }
    public int compareTo(Object obj) {
        DiskRequest req = (DiskRequest)obj;
        return cylinder – req.getCylinder();
    }
    public String toString() {
        String msg = new String("Cylinder(" + cylinder + ")\tHead(" +
                                        head + ")\tSector(" + sector + ")");
        return msg;
    }
}
```

```java
public static void main(String[] args) {
    QueuePriority diskQueue = new QueuePriority();
    int option = getOption();
    DiskRequest req;
    while(option != 3) {
      switch(option) {
        case 1:
            req = getRequest();
            diskQueue.enqueue(req);
            break;
        case 2:
           if(diskQueue.isEmpty())
              System.out.println("Disk queue is empty.");
           else {
              req = (DiskRequest)diskQueue.dequeue();
              System.out.println("Removed request: " + req.toString());
           }
           break;
        case 3:
           break;
        default:
           System.out.println("Error: invalid entry.");
      }
      option = getOption();
    }
}
```