# Ch10. Queues



SNU
IDB Lab.

# Bird's-Eye View (1/2)

- Chapter 9: Stack
  - A kind of Linear list & LIFO(last-in-first-out) structure
  - Insertion and removal from one end
- Chapter 10: Queue
  - A kind of Linear list & FIFO(first-in-first-out) structure
  - Insertion and deletion occur at different ends of the linear list
- Chapter 11: Skip Lists & Hashing
  - Chains augmented with additional forward pointers

# Bird's-Eye View (2/2)

- Representation
  - Array-based class: ArrayQueue
  - Linked class: LinkedQueue

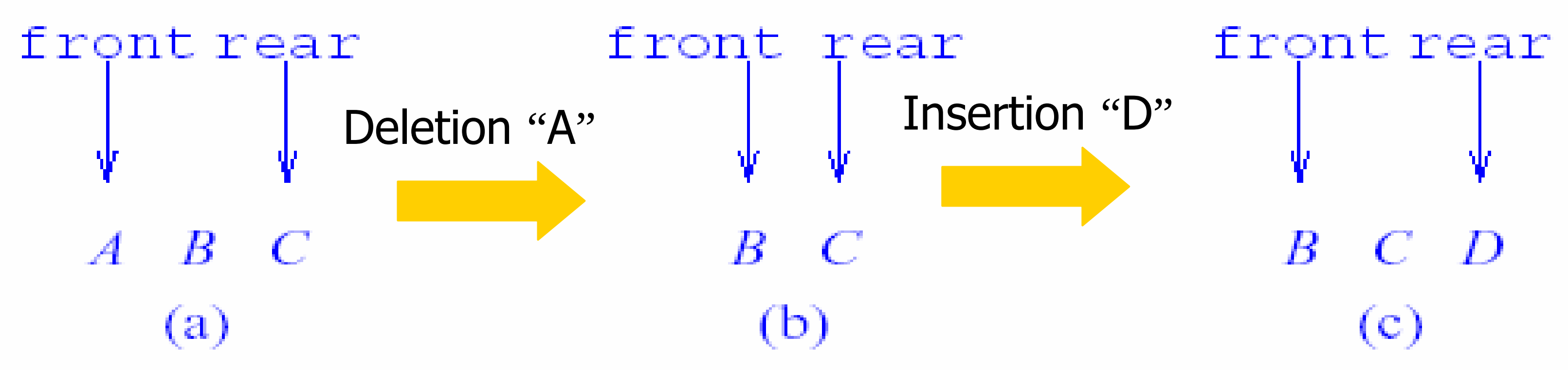
- Queue Applications
  - Railroad Car Rearrangement
    - The shunting track (holding tracks) are FIFO
  - Wire Routing
    - Find the shortest path for a wire
  - Image-Component Labeling
  - Machine Shop Simulation

# Table of Contents

- Definition
- Array Representation
- Linked Representation
- Queue Applications

# Definition

- A Queue is
  - A FIFO (First In First Out) Linear list
  - One end is “front” and the other end is “rear”
  - Additions are done at the rear only
  - Removals are made from the front only



# The Abstract Data Type: Queue

AbstractDataType *Queues* {

Instances

Ordered list of elements; front pointer; rear pointer;

Operations

*isEmpty()* : Return true if queue is empty,
Return false otherwise;

*getFrontElement()* : Return the front element of the queue;

*getRearElement()* : Return the rear element of the queue;

*put(x)* : Add element x at the rear of the queue;

*remove()* : Remove an element from the front of the queue and return it;

}

# The interface Queue

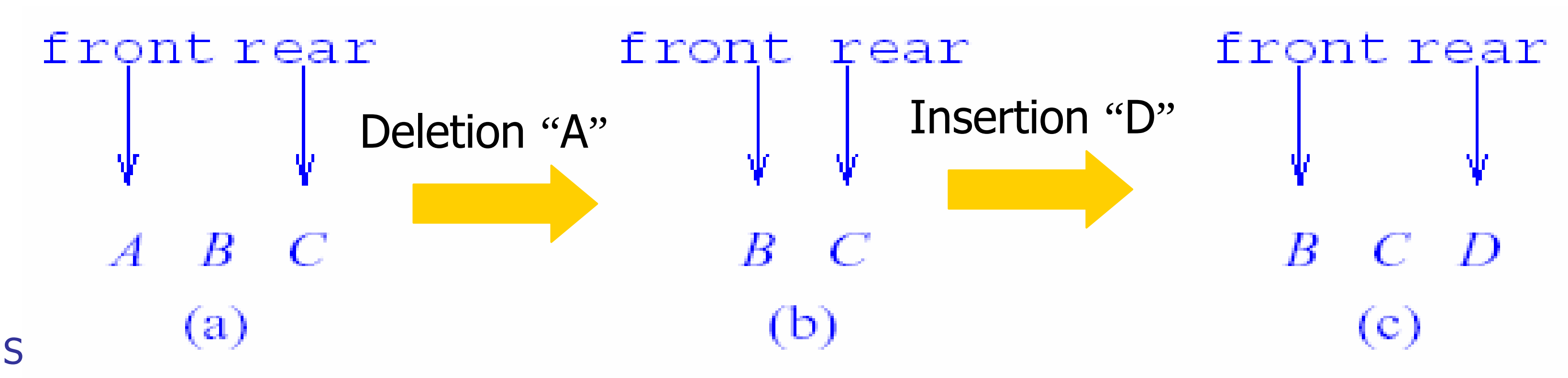
```
public interface Queue
{
    public boolean isEmpty();
    public Object   getFrontEelement();
    public Object   getRearEelement();
    public void     put(Object theObject);
    public Object   remove();
}
```

# Table of Contents

- Definition
- Array Representation
- Linked Representation
- Queue Applications

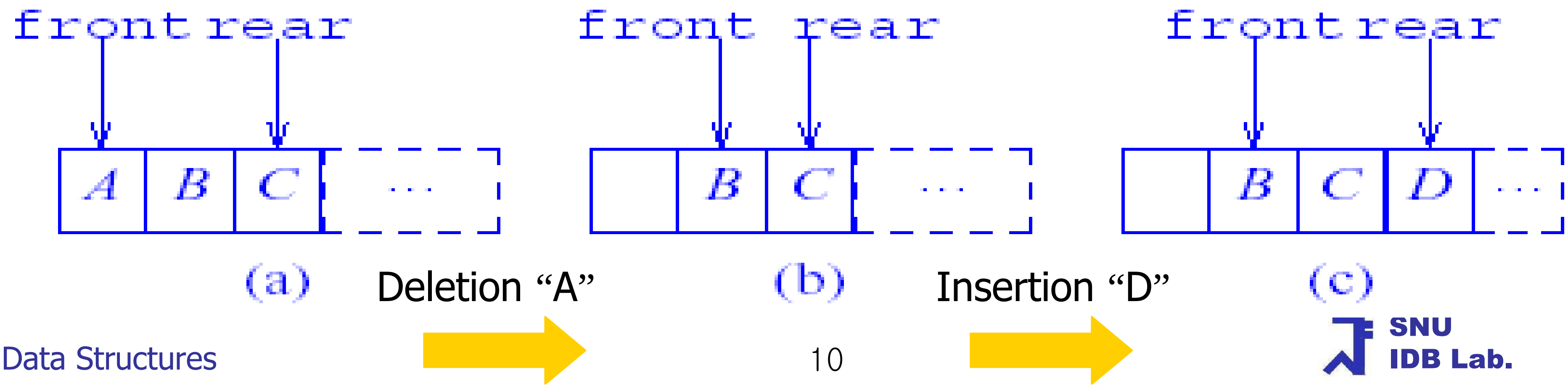
# Queue by Array Representation (1)

- Mapping Function (1) : ***location(i) = i***
  - Element i is stored in queue[i], $i \geq 0$
  - front = 0 (always) & rear = the location of the last element
  - Queue size = rear + 1
  - Empty queue: rear = -1
  - To insert an element: $\theta$ (1) time
    - Increase *rear* by 1 and place the new element at *queue[rear]*
  - To delete an element: $\theta$ (n) time
    - Must slide the elements in position 1 through *rear* one position down the array



# Queue by Array Representation (2)

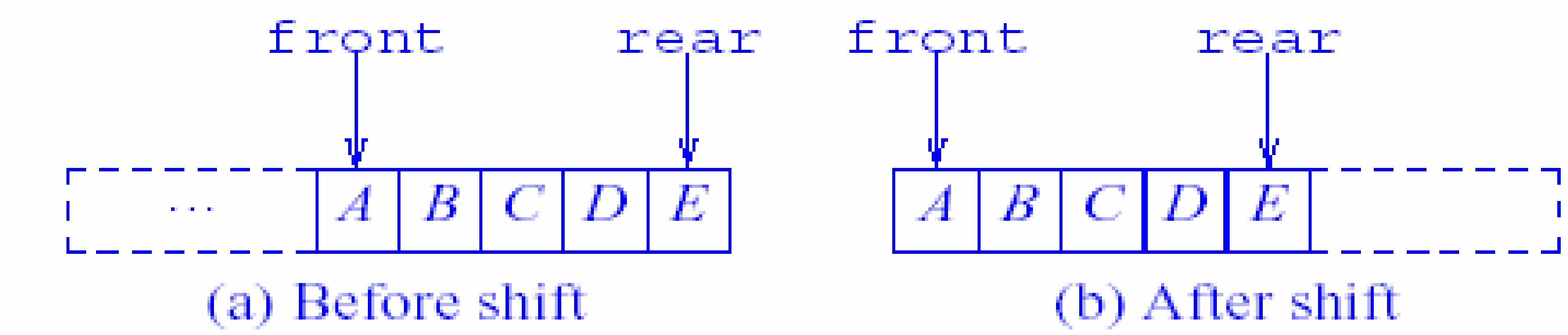
- Mapping Function (2) : **location(i) = location(front) + i**
  - The index i for the front element is 0
  - front = location(front element)
  - rear = location(last element)
  - Empty queue has the condition: rear < front
  - Insert an element
    - The worst-case time from θ (1) to θ (queue.length)
  - Delete an element: θ (1) time
    - Move front to right by 1



# Queue by Array Representation (3)

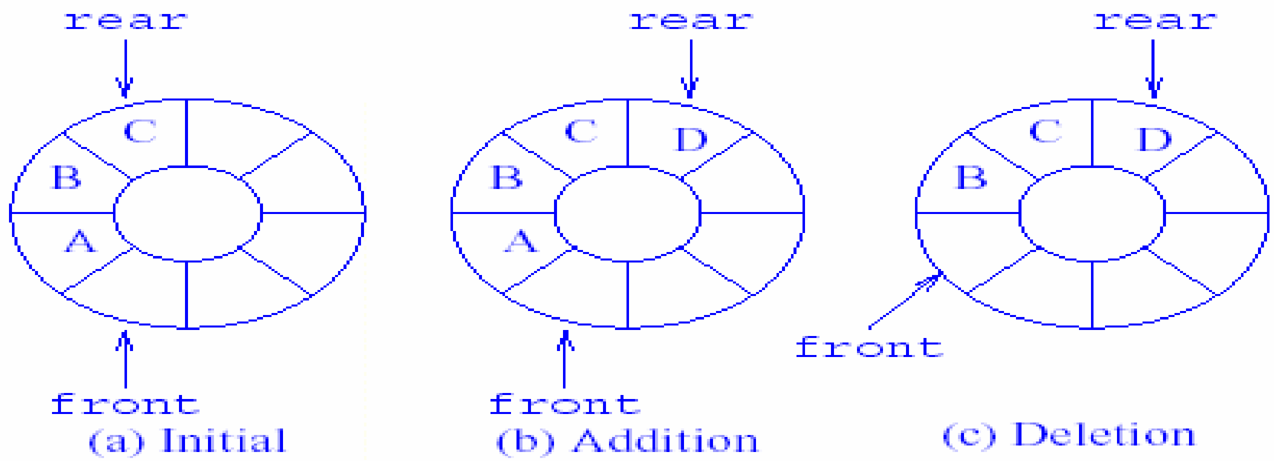
## Mapping Function (2) : **location(i) = location(front) + i**

- When inserting an element
  - If the rear pointer is located in the right end area
    - If some space is available in the left area ➔ move the existing elements to the left rather
    - Else doubling the array



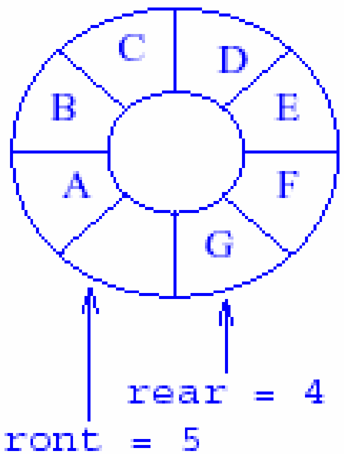
# Queue by Array Representation (4)

- Mapping Function (3) :
  **location(i) = (location(front element) + i) % queue.length**
  - Used the circular array representation of a queue
  - Initial condition : front = rear = 0
  - Empty queue: front = rear
  - When front = rear, it is an empty queue or a full queue
    → Verify whether this insertion will cause the queue to get full
    If so, Double the length of the array queue
  - The circular queue can have at most queue.length -1 elements

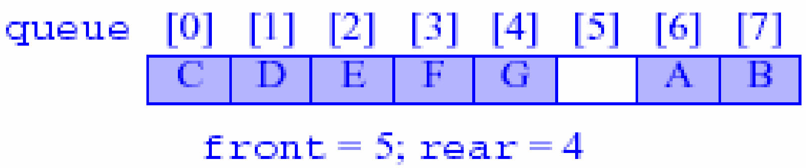


# The class ArrayQueue (1)

- Uses the third mapping function to map a queue into an 1D array queue:
  **location(i) = (location(front element) + i) % queue.length**
- Data members : front, rear, queue
- All methods are similar to those of ArrayStack
- **To visualize array doubling when a circular queue is used, flatten out the array**



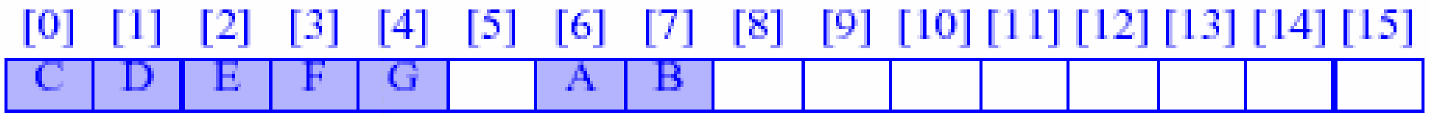
(a) A full circular queue



(b) Flattened view of full circular queue

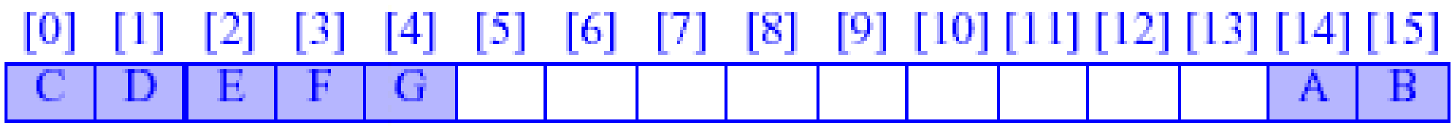
# The class ArrayQueue (2)

- To get a proper circular queue configuration




front = 5; rear = 4
(c) After array doubling

Slide the element the right segment to the right end of array

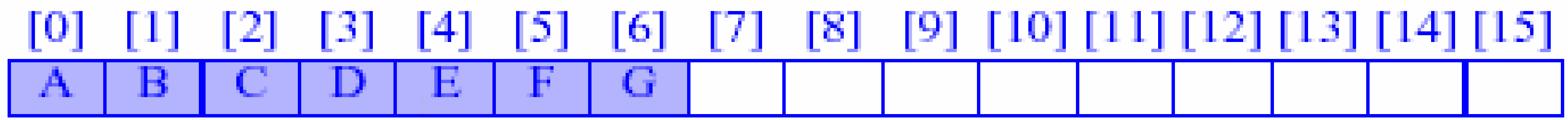



front = 13; rear = 4
(d) After shifting right segment

- **When the second segment is slid to the right, up to *queue.length –2* "additional" element references are copied (worst case: if there are 6 elements (G [] A B C D E F G) in the second segment, A to G will have to be copied into the extended array)**

# The class ArrayQueue (3)

- To limit the number of references copied to *queue.length –1*




front = 15; rear = 6

(e) Alternative configuration

  - Create a new array *newQueue* of twice the length
  - Copy the second segment to positions in *newQueue* beginning at 0
  - Copy the first segment to position in newQueue beginnig at *queue.length – front – 1* (here 3)

# The class ArrayQueue (4)

```
public void put (Object theElement) { //increase array length if necessary
    if ((rear + 1) % queue.length == front) {
        Object[] newQueue = new Object[2*queue.length];
        int start = (front +1) % queue.length;
        if(start < 2)   //no wrap around ([] A B .. G   or A B .. G [])
            System.arraycopy(queue,start,newQueue, 0,queue.length-1);
        else { //queue wraps around
            System.arraycopy(queue,start,newQueue, 0,queue.length-start);
             System.arraycopy(queue,0,newQueue, queue.length-start,rear+1); }
         front = newQueue.length  – 1; //switch to newQueue and set front and rear
          rear = queue.length – 2; //queue size is queue.length - 1
          queue = newQueue; }
    //put the Element at the rear of the queue
    rear = (rear + 1) % queue.length;
    queue[rear] = the Element;
}
```

# The class ArrayQueue (5)

```
public Object remove()
{
  if(isEmpty())
        return null;
  front = (front + 1) % queue.length;
  Object frontElement = queue[front];
  queue[front] = null;   //enable garbage collection
  return frontElement;
}
```

# The class ArrayQueue (6)

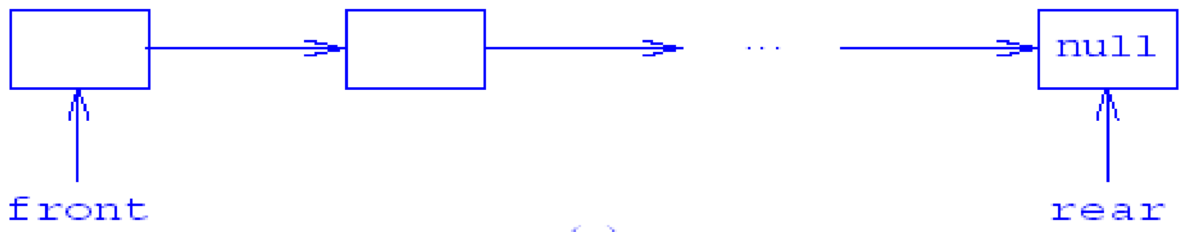
- Complexity
  - constructor(): O(initialCapacity)
  - isEmpty(): θ(1) time
  - getFrontElement(): θ(1) time
  - getRearElement(): θ(1) time
  - remove(): θ(1) time
  - put() :
    - If array doubling is done: θ(queue size)
    - Else: θ(1) time

# Table of Contents

- Definition
- Array Representation
- Linked Representation
- Queue Applications

# Queue by Linked Representation (1)

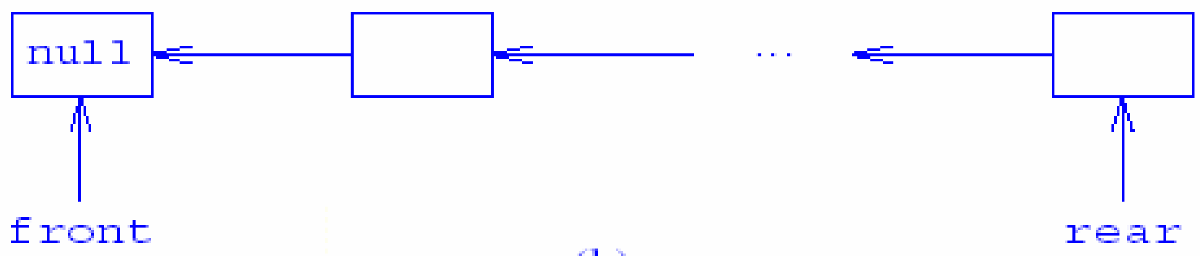
- can be represented as a chain with *front, rear* variables
  - Initial value: front = rear = null
  - Empty queue: front = rear
- Two possibilities for pointers: "from front to rear" or "from rear to front"




(a)

chainNode[rear].next → NULL

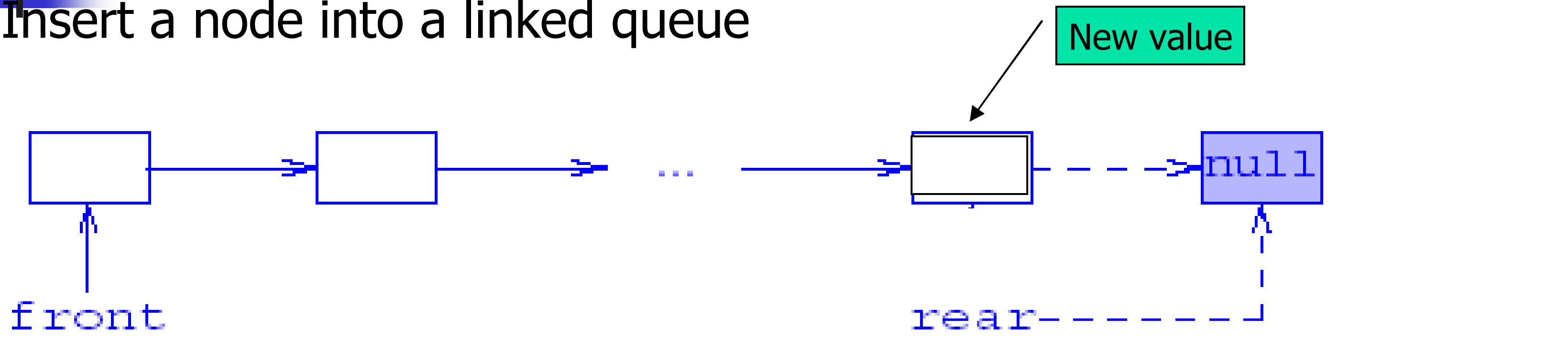
chainNode[front].next → NULL



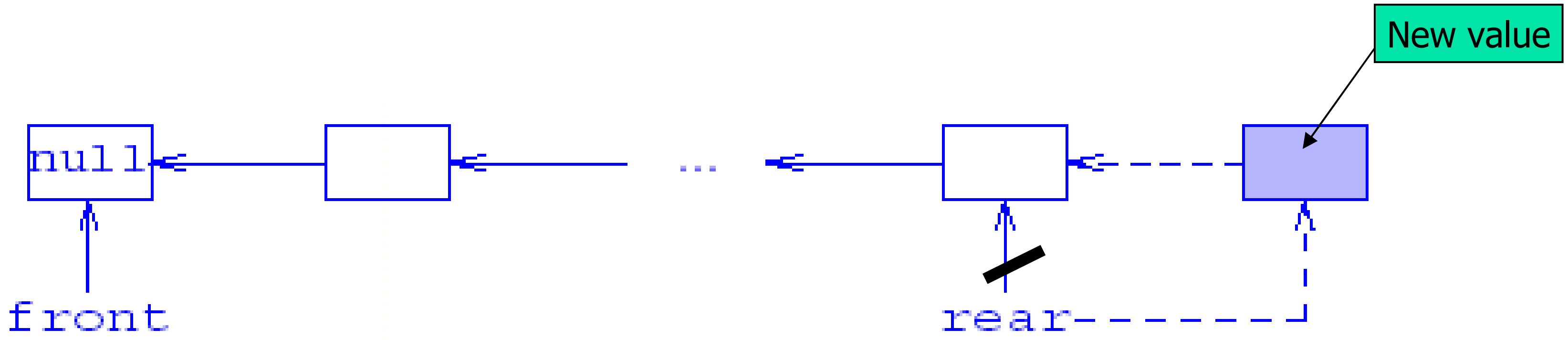

(b)

# Queue by Linked Representation (2)

- Insert a node into a linked queue



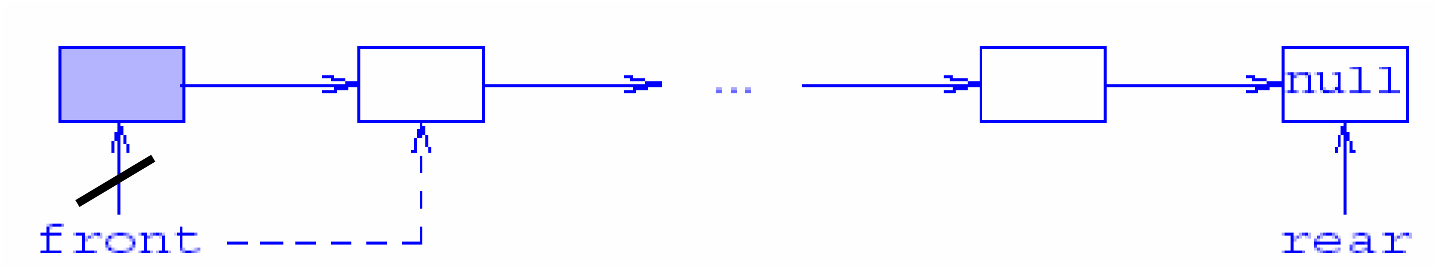

(a) Insert into front-to-rear linkage: create a node & update one pointer




(b) Insert into rear-to-front linkage: create a node & update one pointer

# Queue by Linked Representation (3)

- Remove a node from a linked queue




(a) Remove from front-to-rear linkage: update one pointer


(b) Remove from rear-to-front linkage: update two pointers

- Front-to-rear linkage is more efficient

# Queue by Linked Representation (4)

- Front-to-rear linkage & The complexity of each of methods : θ (1) time

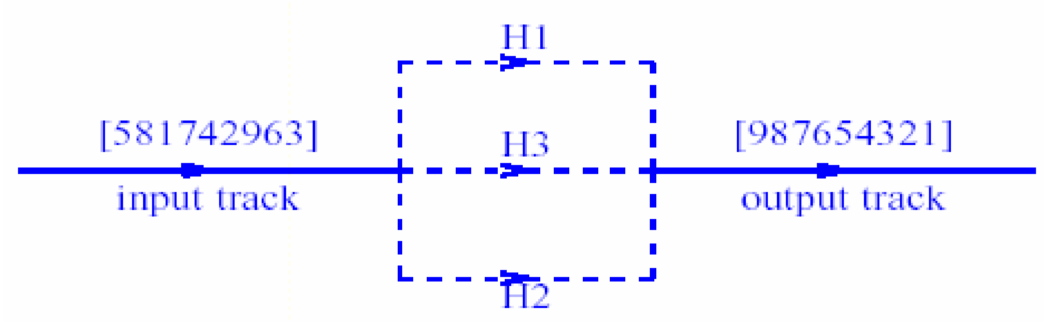
```
public void put (Object theElement){
 ChainNode p = new ChainNode(theElement, null); // create a node
 //append p to the chain
 if (front == null)   front = p;        //empty queue
 else                 rear.next = p;   //nonempty queue
 rear = p;
}
public Object remove() {
  if (isEmpty())     return null;
     Object frontElement = front.element;
  front = front.next;
  if (isEmpty())    rear = null; //enable garbage collection
  return frontElement;
}
```

# Table of Contents

- Definition
- Array Representation
- Linked Representation
- Queue Applications
  - Railroad Car Rearrangement
  - Wire Routing
  - Image-Component Labeling
  - Machine Shop Simulation

# Railroad Car Rearrangement (1)

- Problem: A three-track example




  - Reserve track Hk (the central track) for moving cars directly from the input track to the output track
  - Only k-1 tracks are available for holding cars
    - These tracks operate in a FIFO manner (QUEUE)

- If the next car is not the one that is expected to be output
  - Move car *c* to a holding track that contains only cars with a smaller label
    - If several such tracks exist, select one with the largest label at its left end
    - Otherwise, select any empty track

# 1st Code: RRC with Queue (1)

```
// Can reuse the code in Program 9.9 through Program 9.12
// The method railroad() of Program 9.10 needs to be modified
            //  --   Decrease the number of tracks by 1
            //  --   Change the type of track to ArrayQueue
/* Output the smallest car from the holding tracks */
private static void outputFromHoldingTrack()  {
track[itsTrack].remove(); // remove smallestCar from itsTrack'th track
// find new smallestCar and itsTrack by checking all queue fronts
smallestCar = numberOfCars + 2;
for (int i = 1; i <= numberOfTracks; i++)
  if (!track[i].isEmpty() && ((Integer) track[i].getFrontElement()).intValue() < smallestCar)
   {   smallestCar = ((Integer) track[i].getFrontElement()).intValue();
        itsTrack = i;  }
}
```

# 1st Code: RRC with Queue (2)

```
private static boolean putInHoldingTrack (int c) {  /* put car c into a holding track */
    int bestTrack = 0, bestLast =  0;
    for (int i = 1; i <= numberOfTracks; i++){ // scan tracks
      if  (!track[i].isEmpty())  { // track i not empty
         int lastCar = ((Integer) track[i].getRearElement()).intValue();
         if (c > lastCar && lastCar > bestLast) { // track i has bigger car at its rear
              bestLast = lastCar; bestTrack = i; }
      }  else // track i empty   if (bestTrack == 0)   bestTrack = i;
    } //end of for
    if (bestTrack == 0)   return false; // no feasible track
    track[bestTrack].put(new Integer(c)); // add c to bestTrack
    if (c < smallestCar) { // update smallestCar and itsTrack if needed
        smallestCar = c;
        itsTrack = bestTrack; }
    return true;
  }       // Complexity : O(numberOfCars * k)
```

# $2^{nd}$ Code: RRC with Queue

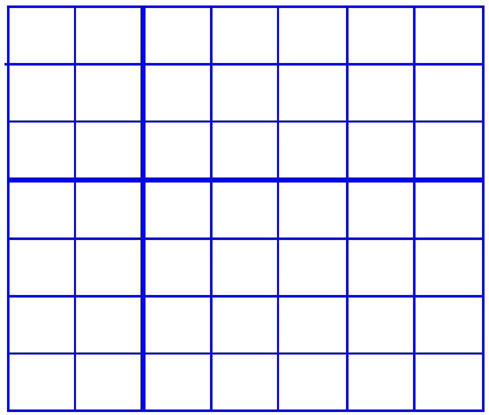
- To simplify the step for outputting the sequence of moves, use below variables and no queues
  - Initially, lastCar[i] = 0, $1 \leq i \leq k$
    whichTrack[i] = 0, $1 \leq i \leq n$
  - If holding track i is empty, lastCar[i] = 0
    Else lastCar[i] = the label no of the last car in track i
  - If car i is in the input track, whichTrack[i] = 0
    Else whichTrack[i] = the hold track that car i was in
- The no-queue implementation is in the website as the class application.RailroaqdWithNoQueues

# Table of Contents

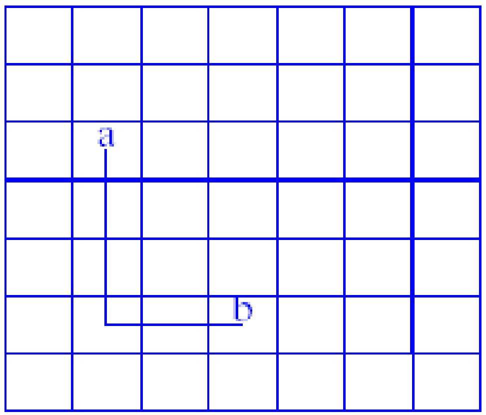
- Queue Applications
  - Railroad Car Rearrangement
  - Wire Routing
  - Image-Component Labeling
  - Machine Shop Simulation

# Wire Routing (1)

- Impose a grid over the wire-routing region
- The grid divides the routing region an n x m array of squares
- Grid squares that already have a wire through them are blocked
- To minimize signal delay, a shortest path is used



(a) A 7 × 7 grid



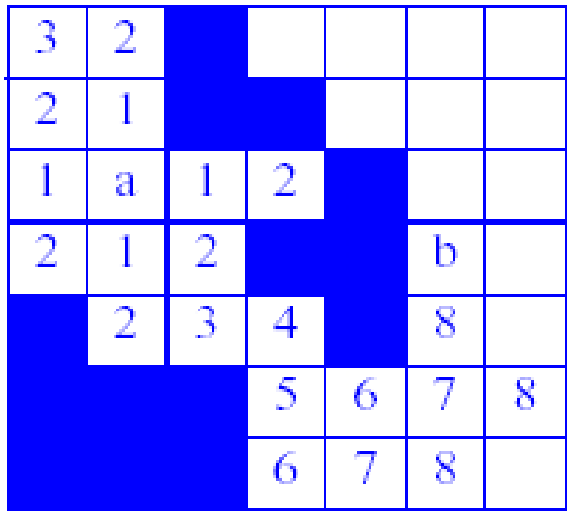

(b) A wire between a and b

- The distance-labeling pass
- The path-identification pass

# Wire Routing (2)

- The distance-labeling pass
  - Begin at “a” and label its reachable neighbors “1”
  - Next, the reachable neighbors of square labeled “1” are labeled “2”
  - Continue until reach “b” or have no neighbors

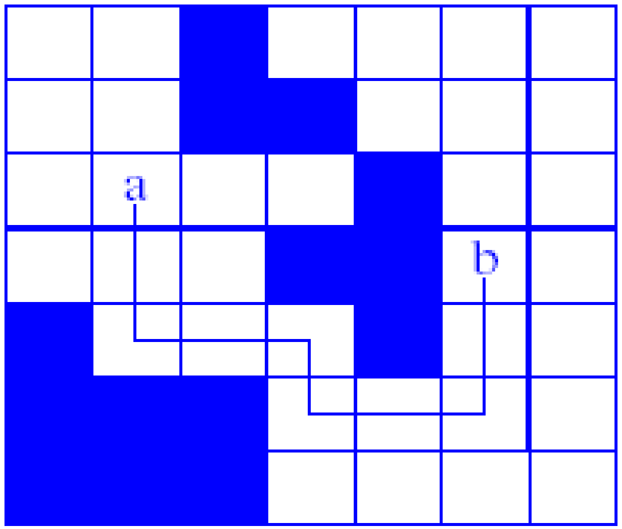
- **The case: a = (3,2) and b = (4,6)**
- **The shaded squares are blocked squares**




(a) Distance labeling

# Wire Routing (3)

- The path-identification pass
  - Reverse traversal from "b"
    - Begin at b
    - Move to any one its neighbors labeled 1 less than b's label
    - **The shortest path between a and b is not unique**




(b) Wire path

# Representation: Wire Routing

- m x m grid : 2D array *grid*
  - 0 : an open position
  - 1 : a blocked position

- To move from a position to its neighbors
  - use array *offsets*

- To keep track of labeled grid positions whose neighbors have not been examined
  - use a *queue*

- Need to overload the array grid: blocking vs. distance
- Conflict the usage of the label "1"
  - To resolve, increase all distance labels by 2

# findPath() in WireRouter (1)

- Start(a) & finish(b) : static data members

```
private static boolean findPath () {
    if  ( (start.row == finish.row) && (start.col == finish.col) ) {
        pathLength = 0;    return true; }
     Position [] offset = new Position [4]; // initialize offsets
    offset[0] = new Position(0, 1);   // right   offset[1] = new Position(1, 0);   // down
    offset[2] = new Position(0, -1);  // left      offset[3] = new Position(-1, 0);  // up
     for (int i = 0; i <= size + 1; i++){ // initialize wall of blocks around the grid
        grid[0][i] = grid[size + 1][i] = 1; // bottom and top
        grid[i][0] = grid[i][size + 1] = 1; // left and right
     }
    Position here = new Position(start.row, start.col);
    grid[start.row][start.col] = 2;   // block
    int numOfNbrs = 4;    // neighbors of a grid position
     // label reachable grid positions
    ArrayQueue q = new ArrayQueue();
    Position nbr = new Position(0, 0);
```

# findPath() in WireRouter (2) -- distance labeling pass

```
do { // label neighbors of here
        for (int i = 0; i < numOfNbrs; i++){  // check out neighbors of here
            nbr.row = here.row + offset[i].row;
            nbr.col = here.col + offset[i].col;
           if (grid[nbr.row][nbr.col] == 0) { // unlabeled nbr, label it
                grid[nbr.row][nbr.col] = grid[here.row][here.col] + 1;
                 if ((nbr.row == finish.row) && (nbr.col == finish.col))  break;
                 q.put(new Position(nbr.row, nbr.col)); // put on queue for later expansion }
         }//end of for
       if ((nbr.row == finish.row) && (nbr.col == finish.col))  break; // are we done?
       if (q.isEmpty())  return false;      // no path
        here = (Position) q.remove();  // get next position
} while(true); // end of do loop
     pathLength = grid[finish.row][finish.col] - 2; // construct path
     path = new Position [pathLength];
```

# findPath() in WireRouter (3) -- path identification pass

```
here = finish; // trace backwards from finish
for (int j = pathLength - 1; j >= 0; j--) {
    path[j] = here;
    // find predecessor position
    for (int i = 0; i < numOfNbrs; i++){
        nbr.row = here.row + offset[i].row;
        nbr.col = here.col + offset[i].col;
        if (grid[nbr.row][nbr.col] == j + 2) break;
     }
     here = new Position(nbr.row, nbr.col); // move to predecessor
}
return true;
} //end of function findPath()
```

# Complexity: Wire Routing

- The distance-labeling phase
  - $O(m^2)$ time (for an m x m grid)
- The path-constructing phase
  - O(length of the shortest path)
- The overall complexity for findPath()
  - $O(m^2)$

# Table of Contents

- Queue Applications
  - Railroad Car Rearrangement
  - Wire Routing
  - Image-Component Labeling
  - Machine Shop Simulation

# Image-component Labeling (1/3)

- A digitized image: m x m matrix of pixels
- In a binary image
  - 0 pixel : image background
  - 1 pixel : a point on an image component
- Two pixels are adjacent
  - if one is to the left, above, right, or below the other
  - If two pixels are adjacent, they are called component pixels
  - Component pixels get the same label
- Two pixels get the same label iff they are pixels of the same image component

# Image-Component Labeling (2/3)

| | | | | | | |
|---|---|---|---|---|---|---|
| | | 1 | | | | |
| | | 1 | 1 | | | |
| | | | | 1 | | |
| | | | 1 | 1 | | |
| | 1 | | | 1 | | 1 |
| 1 | 1 | 1 | | | | 1 |
| 1 | 1 | 1 | | | 1 | 1 |

(a) A 7 × 7 image

- ✓ Four components
  - (1,3),(2,3),(2,4)
  - (3,5),(4,4),(4,5),(5,5)
  - (5,2),(6,1),(6,2),(6,3),(7,1),(7,3)
  - (5,7),(7,6),(7,7)

| | | | | | | |
|---|---|---|---|---|---|---|
| | | 2 | | | | |
| | | 2 | 2 | | | |
| | | | | 3 | | |
| | | | 3 | 3 | | |
| | 4 | | | 3 | | 5 |
| 4 | 4 | 4 | | | | 5 |
| 4 | 4 | 4 | | | 5 | 5 |

(b) Labeled components

- ✓ 2,3,4,…. as component identifiers
- ✓ 1 designated an unlabeled component pixel

# Image-component Labeling (3/3)

- Solution strategy
  - By scanning the pixels by rows and within rows by columns
    - Determine components
  - If unlabeled component is encountered
    - Give a component identifier/label
  - By identifying and labeling all component pixels that are adjacent to the seed
    - Determine the remaining pixels in the component

# Code: ICL (1)

- To move around the image easily, surround the image with a wall of blank
- To determine the pixels adjacent to a given pixel, use the *offset* array

```
private static void labelComponents() {
    // initialize offsets
    …….
   // initialize wall of 0 pixels
    ……
   int numOfNbrs = 4; // neighbors of a pixel position
   ArrayQueue q = new ArrayQueue();
   Position nbr = new Position(0, 0);
   int id = 1;  // component id
   // scan all pixels labeling components
    for (int r = 1; r <= size; r++)      // row r of image
      for (int c = 1; c <= size; c++)  // column c of image
```

# labelComponents() : ICL (2)

```
if (pixel[r][c] == 1){      // new component
   pixel[r][c] = ++id;      // get next id
   Position here = new Position(r, c);
  do{ // find rest of component
       for (int i = 0; i < numOfNbrs; i++) { // check all neighbors
          nbr.row = here.row + offset[i].row;
          nbr.col = here.col + offset[i].col;
          if (pixel[nbr.row][nbr.col] == 1){ //current component
             pixel[nbr.row][nbr.col] = id;
             q.put(new Position(nbr.row, nbr.col)); }
        } //end of for
       here = (Position) q.remove(); // a component pixel if any unexplored pixels
    } while (here != null);
  } // end of if, for c, and for r
} //end of function labelComponent()
```

# Complexity: labelComponents()

- To initialize the wall : $\theta$(m) time
- To initialize offsets : $\theta$(1) time
- For each component, 0(num of pixels in component N) time is spent for identifying and labeling where N is at most $m^2$

- Overall time complexity : $O(m^2)$ time

# Table of Contents

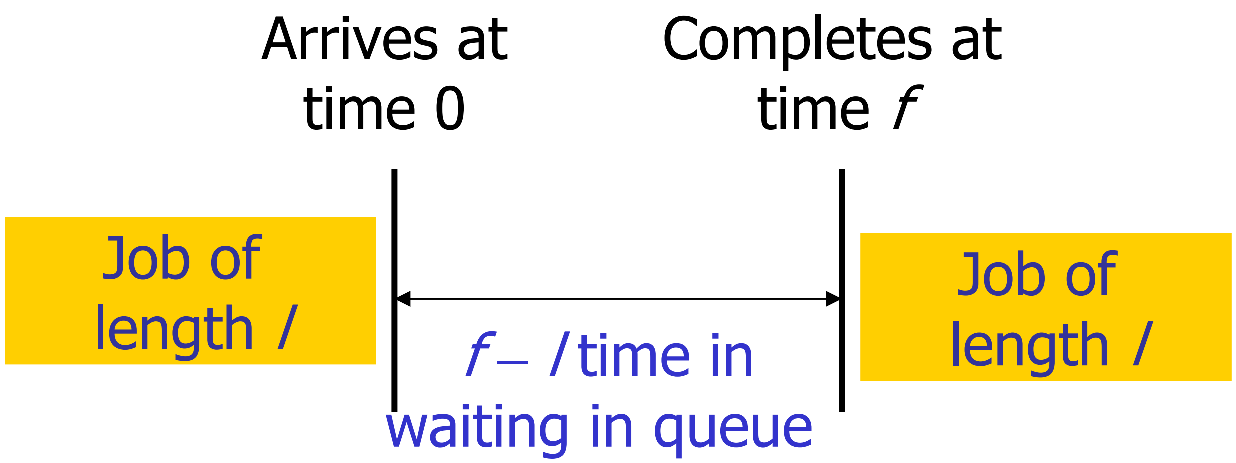
- Queue Applications
  - Railroad Car Rearrangement
  - Wire Routing
  - Image-Component Labeling
  - Machine Shop Simulation

# Machine Shop Simulation (1)

- Problem Description
  - A machine shop
    - Comprises *m* machines
    - Works on jobs
    - Each job comprises several tasks
  - For each task of a job
    - A task time
    - Machine on which it is to be performed
    - Have to be performed in a specified order
  - Each machine is in one of three states
    - active state : working on a task
    - idle state : doing nothing
    - changing-over state : has completed a task and be preparing for a new task
  - Assume
    - Serves waiting jobs in a FIFO manner: QUEUE

# Machine Shop Simulation (2)

- Finish time : the time at which a job's last task completes
- The length of a job : the sum of its task times
- Objective : Minimize the time a job spends waiting in queues



# How the Simulation Works (1)

When has event occurred?

- A task completes, then a new job enters the shop
- Start-event initiates the simulation

```
{ Input the data;
  Create the job queues at each machine;
  Schedule first job in each machine queue;
  While (an unfinished job remains) { // do the simulation
     determine the next event;
     if (the next event is the completion of a machine change over)
        schedule the next job(if any) from this machine's queue;
     else { //a job task has completed
          put the machine that finished the job task
          into its change-over states;
          move the job whose task has finished
          to the machine for its next task; }
  }
}
```

# How the Simulation Works (2)

- For example, Consider a machine shop that has m=3 machines and n=4 jobs

| Job# | #Tasks | Tasks | Length |
|---|---|---|---|
| 1 | 3 | (1,2) (2,4) (1,1) | 7 |
| 2 | 2 | (3,4) (1,2) | 6 |
| 3 | 2 | (1,4) (2,4) | 8 |
| 4 | 2 | (3,1) (2,3) | 4 |

(a) Job characteristics

✓ Job 1 has three tasks

✓ (1,2) of job1 means that it is to be done on M1 and takes tow time units

# How the Simulation Works (3)

- Show the machine shop simulation

| Time | Machine Queues | | | Active Jobs | | | Finish Times | | |
|---|---|---|---|---|---|---|---|---|---|
| | $M1$ | $M2$ | $M3$ | $M1$ | $M2$ | $M3$ | $M1$ | $M2$ | $M3$ |
| Init | 1,3 | – | 2,4 | I | I | I | L | L | L |
| 0 | 3 | – | 4 | 1 | I | 2 | 2 | L | 4 |
| 2 | 3 | – | 4 | C | 1 | 2 | 4 | 6 | 4 |
| 4 | 2 | – | 4 | 3 | 1 | C | 8 | 6 | 5 |
| 5 | 2 | – | – | 3 | 1 | 4 | 8 | 6 | 6 |
| 6 | 2,1 | 4 | – | 3 | C | C | 8 | 9 | 7 |
| 7 | 2,1 | 4 | – | 3 | C | I | 8 | 9 | L |
| 8 | 2,1 | 4,3 | – | C | C | I | 10 | 9 | L |
| 9 | 2,1 | 3 | – | C | 4 | I | 10 | 12 | L |
| 10 | 1 | 3 | – | 2 | 4 | I | 12 | 12 | L |
| 12 | 1 | 3 | – | C | C | I | 14 | 15 | L |
| 14 | – | 3 | – | 1 | C | I | 15 | 15 | L |
| 15 | – | – | – | C | 3 | I | 17 | 19 | L |
| 16 | – | – | – | C | 3 | I | L | 19 | L |
| 17 | – | – | – | I | 3 | I | L | 19 | L |

(b) Simulation

✓ C : change over,  L : large time (i.e., the finish time is undefined)

# How the Simulation Works (4)

- Show the machine shop simulation

| Job# | Finish Time | Wait Time |
|---|---|---|
| 1 | 15 | 8 |
| 2 | 12 | 6 |
| 3 | 19 | 11 |
| 4 | 12 | 8 |
| Total | 58 | 33 |

(c) Finish and wait times

✓ Example, job2 : The length = 6, the finish time = 12. So, wait time = 6

- Benefit
  - Can identify bottleneck machines and bottleneck stations
  - Can be used to help make expansion /modernization decisions at the factory
  - May be obtained an accurate estimate

# High-Level Simulator Design

- Assume: All jobs are available initially
  - Until all jobs are completed, keep running
- Be implemented as *MachineShopSimulator* class

- The data objects : tasks, jobs, machines, an event list
  - Be defined in a class
  - Top- level nested class of MachineShopSimulator

- 5 modules
  - Input the data
  - Put the jobs into the queues
  - Perform the start event
  - Run through all the events
  - Output the Machine wait times

# The main() method in MSS

- LargeTime : class data member of MachineShopSimulator
  - Larger than any permissible simulated time

```
/** entry point for machine shop simulator */
public static void main(String [] args)
{
  largeTime = Integer.MAX_VALUE;
  inputData(); // get machine and job data
  startShop(); // initial machine loading
  simulate();  // run all jobs through shop
  outputStatistics(); // output machine wait times
}
```

# The class Task in MSS

- Assume that all times are integral

```
private static class Task{
    // data members
    private int machine;
    private int time;
    // constructor
    private Task (int theMachine, int theTime)
    { machine = theMachine;
      time = theTime;
    }
  }
```

# The class Job in MSS (1)

- The class Job has a list of associated tasks that are performed in list order which are represented as a queue

```
private static class Job{
    // data members
    private LinkedQueue taskQ;  // this job's tasks
    private int length;         // sum of scheduled task times
    private int arrivalTime;  // arrival time at current queue
    private int id;             // job identifier
    // constructor
    private Job(int theId){
        id = theId; //only used when outputting the total wait time
        taskQ = new LinkedQueue(); // length and arrivalTime have default value 0
    }
```

# The class Job in MSS (2)

```
// only be used during data input
private void addTask (int theMachine, int theTime)
{  taskQ.put(new Task(theMachine, theTime));
}

/* * only be used when a job is moved from a machine
   * queue to active status
   * remove next task of job and return its time
   * also update length */
private int removeNextTask() {
  int theTime = ((Task) taskQ.remove()).time;
  length += theTime;
  return theTime; }
```

# The class Machine in MSS

By using linked queues, limit the space required for machine queues

```
private static class Machine {
  // data members
  LinkedQueue jobQ;
  int changeTime; // machine change-over time
  int totalWait;    // total delay at this machine
  // number of tasks processed on this machine
  int numTasks;
  Job activeJob; // job currently active on this machine
  private Machine() // constructor
  { jobQ = new LinkedQueue();}
}
```

# The class EventList in MSS (1)

```
/* store the finish times of all machines */
private static class EventList {
   int [] finishTime; // finish time array
   // constructor
   private EventList(int theNumMachines, int theLargeTime)
   { // initialize finish times for m machines
      if (theNumMachines < 1)
      throw new IllegalArgumentException  ("number of machines must be >= 1");
      finishTime = new int [theNumMachines + 1];
       // all machines are idle, initialize with large finish time
       for (int i = 1; i <= theNumMachines; i++)   finishTime[i] = theLargeTime;
   } //end of constructor
```

# The class EventList in MSS (2)

```
private int nextEventMachine(){ // find first machine to finish
  //this is the machine with smallest finish time
      int p = 1; int t = finishTime[1];
      for (int i = 2; i < finishTime.length; i++) {
            if (finishTime[i] < t){ // i finishes earlier
                  p = i;
                  t = finishTime[i]; }
       } //end of for
       return p;
   } //end of nextEventMachine

private int nextEventTime(int theMachine) {  return finishTime[theMachine];}

private void setFinishTime(int theMachine, int theTime) {
          finishTime[theMachine] = theTime; }
}
```

# The class EventList in MSS (3)

- The complexity
  - nextEventMachine
    - θ(m) time with m machines
  - setFinishTime
    - θ(1) time
  - When numTracks is the total number of tasks across all jobs
    - nextEventMachine() : θ(numTracks * m)
    - setFinishTime invocations() : θ(numTracks * m)

# Data Members in MSS

```
private static int timeNow;              // current time
private static int numMachines;          // number of machines
private static int numJobs;              // number of jobs
private static EventList eList;          // pointer to event list
private static Machine [] machine;       // array of machines
private static int largeTime;            // all machines finish before this
```

# inputData() in MSS (1)

```
static void inputData(){ // define the input stream to be the standard input stream
    MyInputStream keyboard = new MyInputStream();
    System.out.println("Enter number of machines and jobs");
    numMachines = keyboard.readInteger();
    numJobs = keyboard.readInteger();
    //Exception processing
     ……..
    // create event and machine queues
    eList = new EventList(numMachines, largeTime);
    machine = new Machine [numMachines + 1];
    for (int i = 1; i <= numMachines; i++)      machine[i] = new Machine();
    for (int j = 1; j <= numMachines; j++) { // input the change-over times
        int ct = keyboard.readInteger();
        if (ct < 0) throw new MyInputException("change-over time must be >= 0");
        machine[j].changeTime = ct;
    }
```

# inputData() in MSS (2)

```
Job theJob; // input the jobs
for (int i = 1; i <= numJobs; i++){
  System.out.println("Enter number of tasks for job " + i);
  int tasks = keyboard.readInteger();  // number of tasks
  int firstMachine = 0;               // machine for first task
  if (tasks < 1) throw new MyInputException("each job must have > 1 task");
  theJob = new Job(i); // create the job
  for (int j = 1; j <= tasks; j++) { // get tasks for job i
      int theMachine = keyboard.readInteger();
      int theTaskTime = keyboard.readInteger();
      //InputException processing …
      if (j == 1)  firstMachine = theMachine;        // job's first machine
      theJob.addTask(theMachine, theTaskTime);  // add to task queue }
   machine[firstMachine].jobQ.put(theJob); } // end of for i
}
```

# startShop() in MSS

```
/** load first jobs onto each machine */
static void startShop()
{
  for (int p = 1; p <= numMachines; p++)
    changeState(p);
}
```

# changeState() in MSS (1)

```
static Job changeState(int theMachine)  { // Task on theMachine has finished, schedule next one.
   Job lastJob;
   if (machine[theMachine].activeJob == null) { lastJob = null;  // in idle or change-over state
       // wait over, ready for new job
       if (machine[theMachine].jobQ.isEmpty()) // no waiting job
             eList.setFinishTime(theMachine, largeTime);
       else{ // take job off the queue and work on it
             machine[theMachine].activeJob =  (Job) machine[theMachine].jobQ.remove();
             machine[theMachine].totalWait += timeNow - machine[theMachine].activeJob.arrivalTime;
             machine[theMachine].numTasks++;
             int t = machine[theMachine].activeJob.removeNextTask();
             eList.setFinishTime(theMachine, timeNow + t); } //end of else
     } //end of if
```

# changeState() in MSS (2)

```
  else
  {  // task has just finished on machine[theMachine]
     // schedule change-over time
     lastJob = machine[theMachine].activeJob;
     machine[theMachine].activeJob = null;
     eList.setFinishTime(theMachine, timeNow +  machine[theMachine].changeTime);
  }
  return lastJob;
} //end of changeState()
```

# simulate() in MSS

- Until the last job completes, cycles through all shop events

```
static void simulate()  {
    while (numJobs > 0) { // at least one job left
      int nextToFinish = eList.nextEventMachine();
      timeNow = eList.nextEventTime(nextToFinish);
      // change job on machine nextToFinish
      Job theJob = changeState(nextToFinish);
      // move theJob to its next machine
      // decrement numJobs if theJob has finished
      if (theJob != null && !moveToNextMachine(theJob))
         numJobs--;
    }
}
```

# moveToNextMachine() in MSS

- move theJob to machine for its next task

```
static boolean moveToNextMachine(Job theJob){
if (theJob.taskQ.isEmpty()) { // no next task, the job has completed and output times
      System.out.println("Job " + theJob.id + " has completed at "
                        + timeNow + " Total wait was "  + (timeNow - theJob.length));
      return false; }
 else { // theJob has a next task     // get machine for next task
        int p = ((Task) theJob.taskQ.getFrontElement()).machine;
        machine[p].jobQ.put(theJob); // put on machine p's wait queue
        theJob.arrivalTime = timeNow;
        // if p idle, schedule immediately
        if (eList.nextEventTime(p) == largeTime) {changeState(p); } // machine is idle
        return true; } //end of else
  } //end of moveToNextMachine()
```

# outputStatistics() in MSS

```
 /* Output the finish time, total wait time and no of tasks processed */
static void outputStatistics()
{
  System.out.println("Finish time = " + timeNow);
  for (int p = 1; p <= numMachines; p++) {
    System.out.println("Machine " + p + " completed "+ machine[p].numTasks + " tasks");
    System.out.println("The total wait time was "+ machine[p].totalWait);
    System.out.println();
    }
}
```

# Summary

- A queue is
  - A kind of Linear list
  - Insertion and deletion occur at different ends of the linear list
  - FIFO structure
- Representation
  - Array-based class
  - Linked class
- Queue Applications
  - Railroad Car Rearrangement
    - The shunting track are FIFO
  - Wire Routing
    - Find the shortest path for a wire
  - Image-Component Labeling
    - If two pixels are part of the same image component, they are the same label
  - Machine Shop Simulation
    - Determine the total time each job spends and the total wait at each machine

# JDK class: java.util.Queue

public interface ***Queue*** extends Collection {

methods

*boolean offer(Object obj)*: Inserts *obj* into the queue;
Returns *true* iff adding is successful

*Object remove()*: Removes and returns the object at the head

}