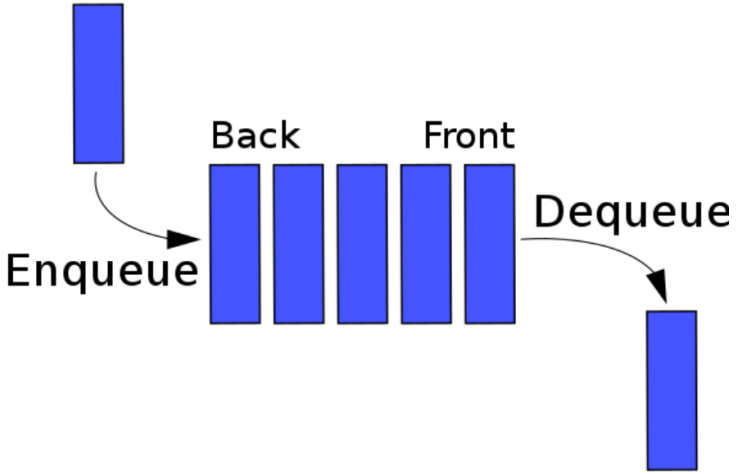
# QUEUES

## OVERVIEW

# OVERVIEW

- **What is a queue?**

Queue at bus stop




Queue data structure

# OVERVIEW

- **With a queue data structure we insert data at the "back" of the queue and remove from the "front" of the queue**

- **Think of a line of people waiting for a bus**
  - People get in line based on their arrival time
  - New arrivals go to back of line and wait their turn
  - People at front of the  line enter bus first

- **This pattern of data usage has two names:**
  - FIFO - first in, first out
  - LILO – last in, last out

# OVERVIEW

- **A wide range of programming problems can be solved using a queue data structure**
  - Queues can be used to simulate human behavior (bus stops, gas stations, banks, ticket sales, etc.)
  - Queues can also be used to provide fair service (print queues, process queues, communication buffers, etc.)
  - Finally, queues can be used in string processing, polygon filling, and breadth first search applications

- **Queues can be implemented using fixed length arrays or using linked lists**
  - Arrays are faster, but linked lists can never become full

# QUEUES

## QUEUE INTERFACE

# QUEUE INTERFACE

- **The queue ADT has the following operations:**
  - Create – Initialize queue data structure
  - Destroy – Delete queue data structure
  - Insert – Insert data at the end of queue
  - Remove – Remove data at the front of queue
  - IsFull – Check if the queue is at max capacity
  - IsEmpty – Check if the queue is has no data

- **The type of data stored in the queue varies by application**
  - Character – string processing
  - Integer – polygon flood fill
  - Object – simulation or scheduling information

# QUEUE INTERFACE

```
class Queue
{
 public:
    // Constructors
    Queue();
    Queue(const Queue& queue);
    ~Queue();

    // Basic methods
    void Insert(const int number);
    void Remove(int & number);
```

# QUEUE INTERFACE

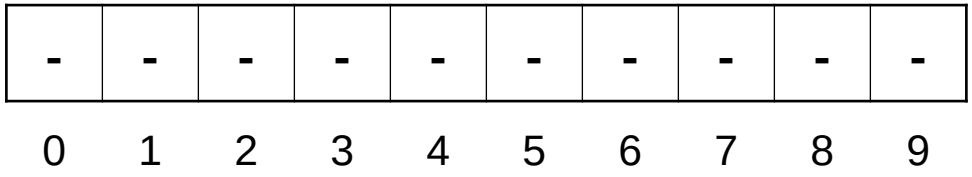
```
...
    // Other methods
    int GetCount();
    int GetFront();
    bool IsFull();
    bool IsEmpty();
    void Print();

private:
    // To be added
};
```

# QUEUES

## QUEUE IMPLEMENTATION

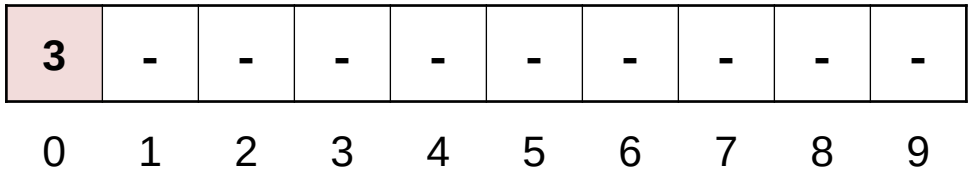
# ARRAY BASED

- **We create an empty queue using an array with size = 10 and a variable count = 0 to store the number of items**




count = 0

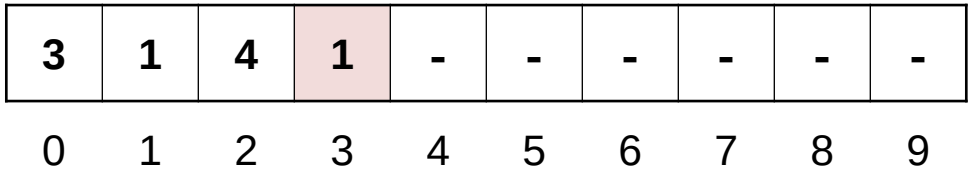
- **When we insert a value 3 on the queue, store the data at array[count] and we increment count**



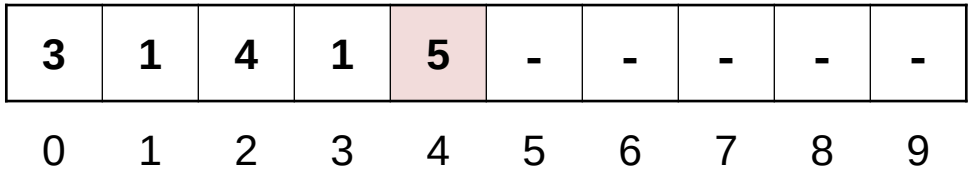

count = 1

# ARRAY BASED

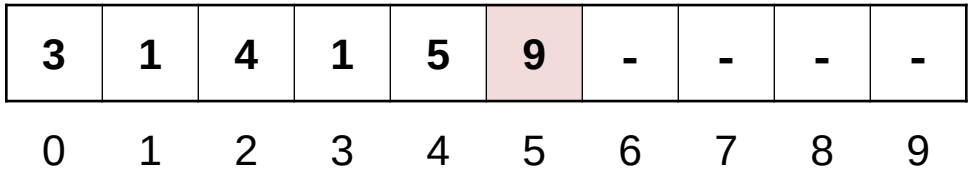
- **As we insert more data into the queue, the array fills in from left to right and count increases**




insert 1, count = 4



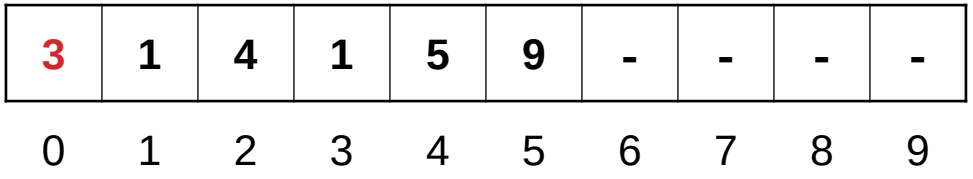

insert 5, count = 5



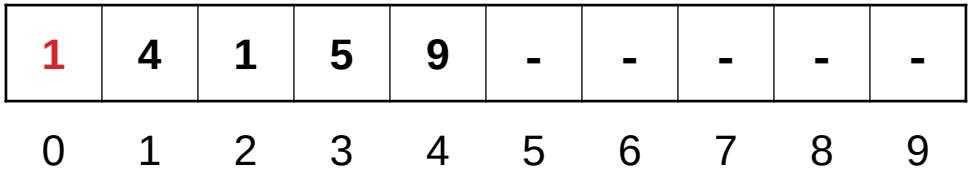

insert 9, count = 6

# ARRAY BASED

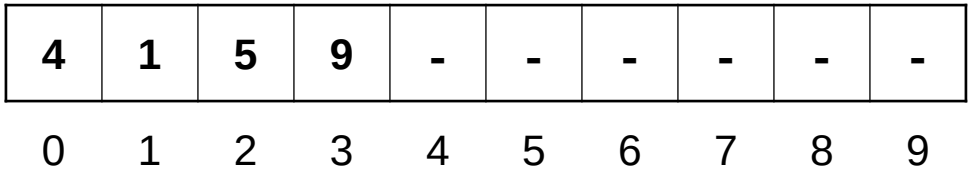
- **When we remove a value from the front of the queue, we shift data left in the array and decrement count by one**




count = 6



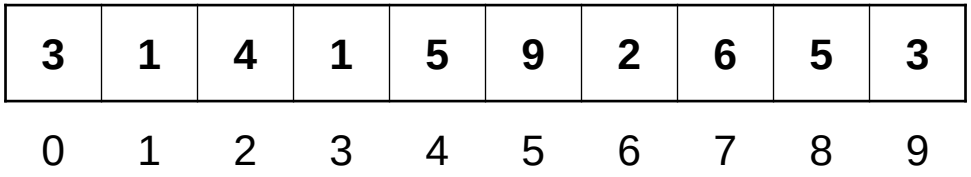

remove 3, count = 5



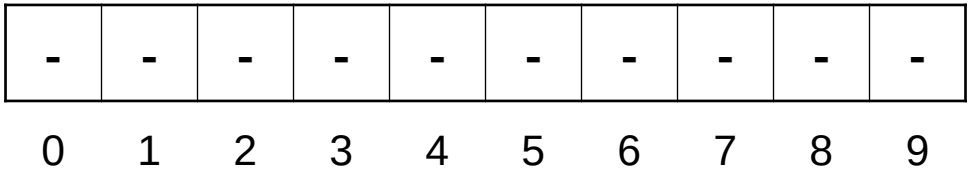

remove 1, count = 4

# ARRAY BASED

- A queue is full when count = size




- A queue is empty when count = 0



# ARRAY BASED

```
class Queue
{
 public:
    // Constructors
    Queue();
    Queue(const Queue& queue);
    ~Queue();

    // Basic methods
    void Insert(const int number);
    void Remove(int & number);
```

# ARRAY BASED

```
...
    // Other methods
    int GetCount();
    int GetFront();
    bool IsFull();
    bool IsEmpty();
    void Print();

private:
    static const int MAX_SIZE = 100;
    int data[MAX_SIZE];
    int count;
};
```

# ARRAY BASED

```
// Constructor function
Queue::Queue ()
{
    for (int index=0; index<MAX_SIZE; index++)
        data[index] = 0;
    count = 0;
}
```

# ARRAY BASED

```
// Copy constructor
Queue::Queue (const Queue & queue)
{
    for (int index=0; index<MAX_SIZE; index++)
        data[index] = queue.data[index];
    count = queue.count;
}
```

# ARRAY BASED

```
// Destructor function
Queue::~Queue()
{
    // Empty
}
```

# ARRAY BASED

```
// Insert method
void Queue::Insert(const int number)
{
    // Check for full queue
    if (IsFull())
        return;

    // Save data in queue
    data[count++] = number;
}
```

This method ignores insert if the queue is already full

This increments count after using its value to access array

# ARRAY BASED

```
// Insert method
void Queue::Insert(const int number)
{
    // Check for full queue
    if (IsFull())
        return;

    // Save data in queue
    data[count] = number;
    count++;
}
```

This method ignores insert if the queue is already full

This increments count after using its value to access array

# ARRAY BASED

```
// Remove method
void Queue::Remove(int & number)
{
    // Check for empty queue
    if (IsEmpty()) return;

    // Remove front value from queue
    number = data[0];
    count--;
    for (int i = 0; i < count; i++)
       data[i] = data[i + 1];
}
```

This method returns if the queue is empty

Shifting data in array is simple but very slow

# ARRAY BASED

```
// GetLength method
int Queue::GetCount()
{
    return count;
}

// GetFront method
int Queue::GetFront()
{
    return data[0];
}
```

# ARRAY BASED

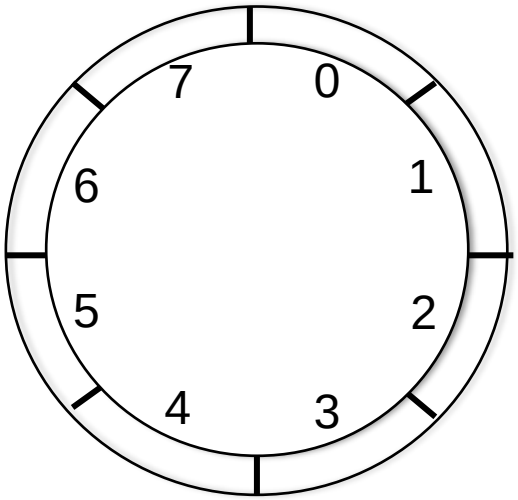
```
// True if queue is full
bool Queue::IsFull()
{
    return (count == MAX_SIZE);
}

// True if queue is empty
bool Queue::IsEmpty()
{
    return (count == 0);
}
```

# ARRAY BASED

```
// Print method
void Queue::Print()
{
    cout << ”queue: ";
    for (int index=0; index<count; index++)
        cout << data[index] << ' ';
    cout << endl;
}
```

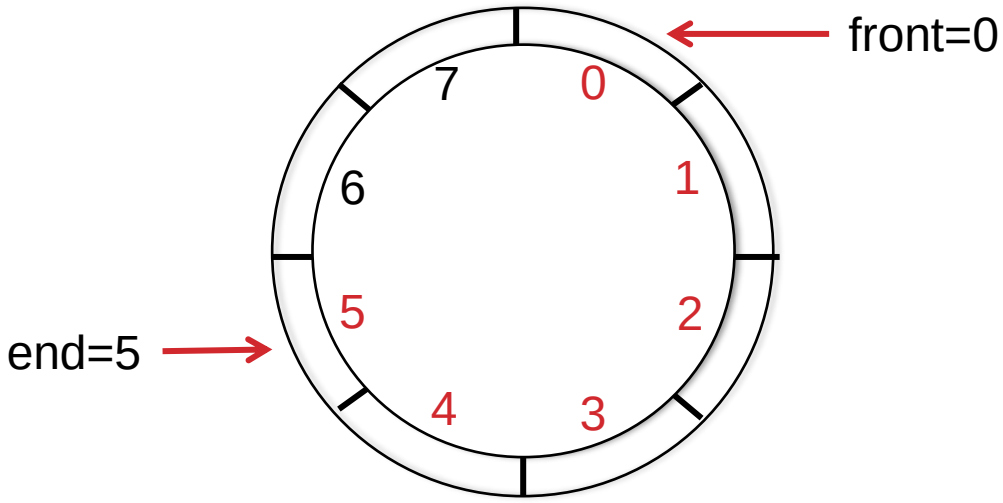
# CIRCULAR QUEUE

- **Moving data in the array after every remove operation is too slow for practical applications**
  - The solution is to create a circular queue by “bending” an array back onto itself



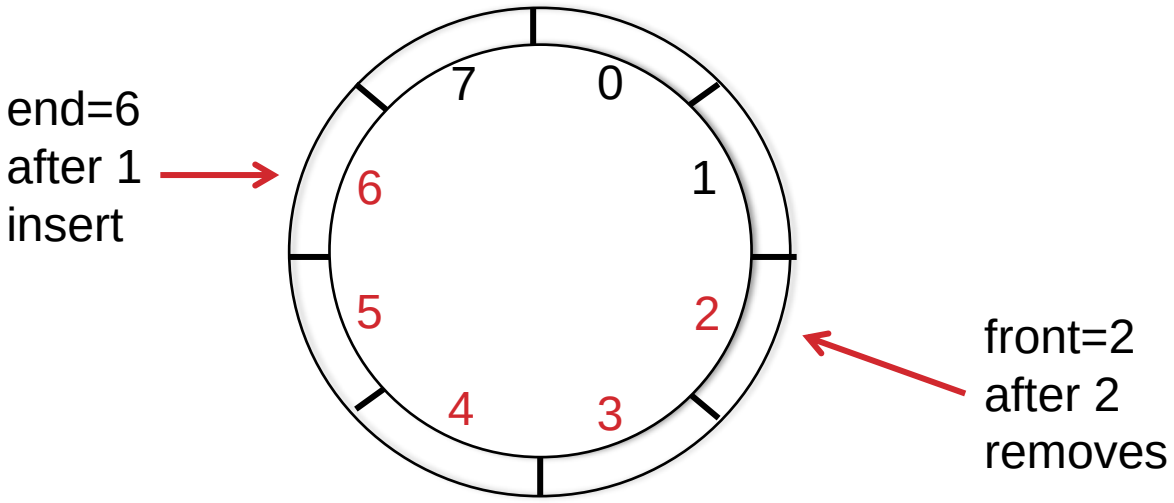
# CIRCULAR QUEUE

- **Use two integer variables keep track of the locations of front and end of the queue**
  - After inserting 6 values we would have the following



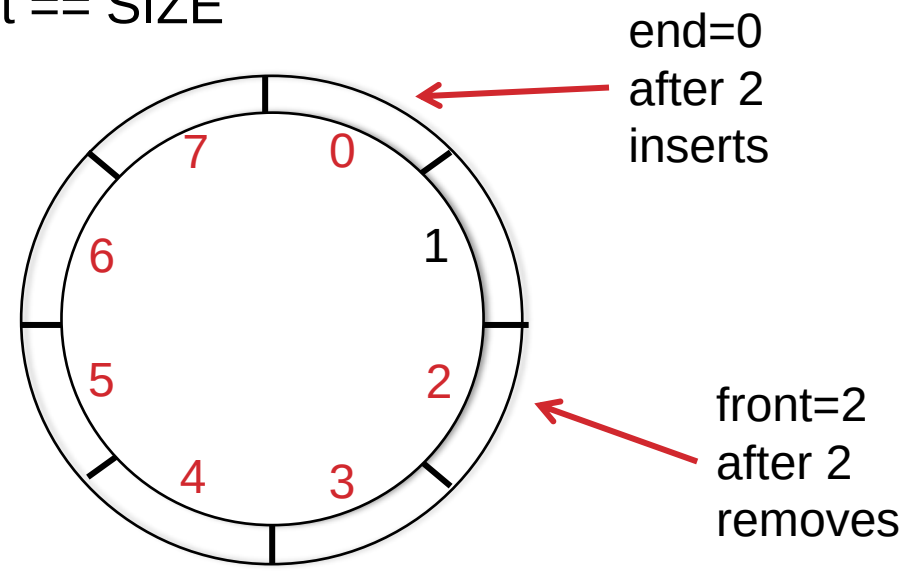
# CIRCULAR QUEUE

- **Update values of front and end as we insert and remove**
  - Insert:  end = (end + 1) % SIZE
  - Remove:  front = (front + 1) % SIZE



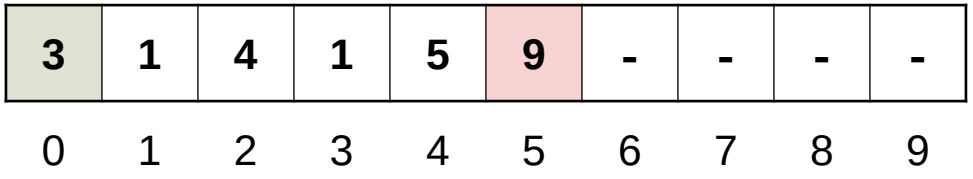
# CIRCULAR QUEUE

- **Array front and end locations will wrap around when they reach the end of the array**
  - Queue is full if ((end + 1) % SIZE == front)
  - Or when count == SIZE

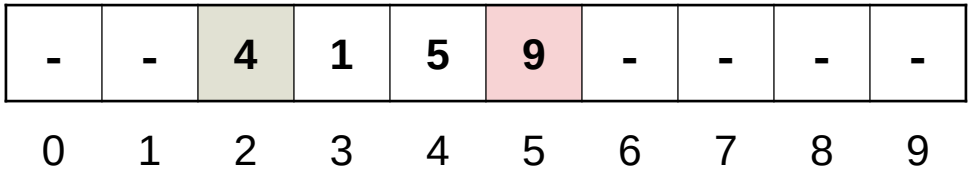


# CIRCULAR QUEUE

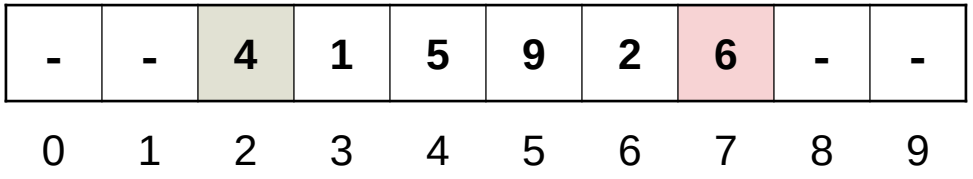
- **Example:**




Initial queue
front=0, end=5,
count=6



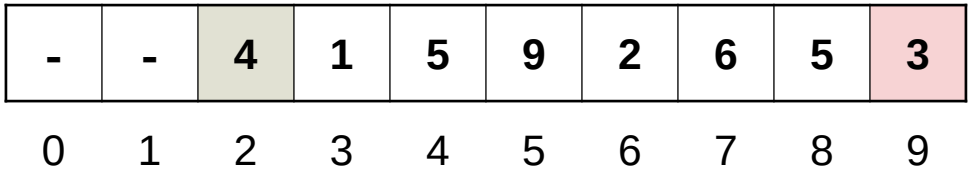

After 2 removes
front=2, end=5,
count=4



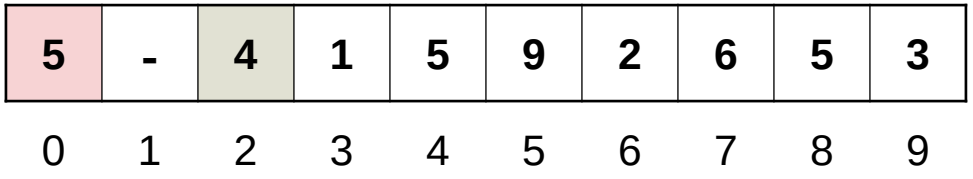

After 2 inserts
front=2, end=7,
count=6

# CIRCULAR QUEUE

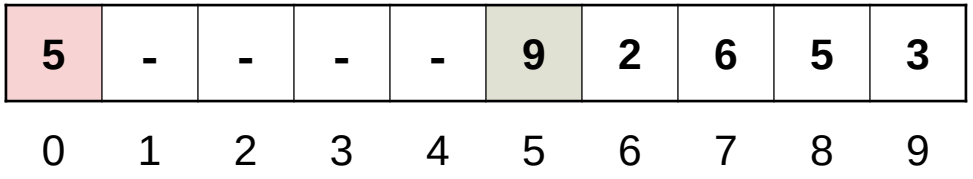
- **Example:**




After 2 inserts
front=2, end=9,
count=8




After 1 insert
front=2, end=0,
count=9




After 3 removes
front=5, end=0,
count=6

# CIRCULAR QUEUE

```
// Insert method
void Queue::Insert(const int number)
{
    // Check for full queue
    if (IsFull()) return;

    // Save data in queue
    end = (end + 1) % SIZE;
    data[end] = number;
    count++;
}
```

This method ignores insert if the queue is already full

This updates the end location and then stores the data

# CIRCULAR QUEUE

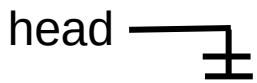
```
// Remove method
void Queue::Remove(int & number)
{
    // Check for empty queue
    if (IsEmpty()) return;

    // Remove front value from queue
    number = data[front];
    front = (front + 1) % SIZE;
    count--;
}
```

This method returns if the queue is empty

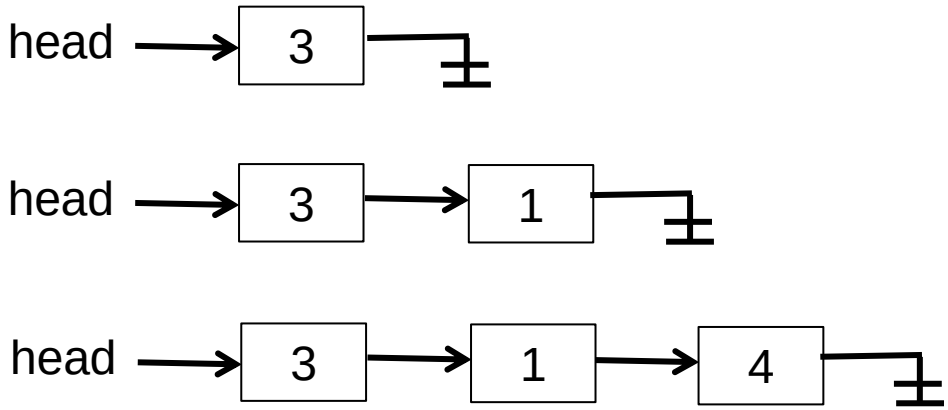
This removes the data and then updates front location

# LINKED LIST BASED

- **We create an empty queue by creating an empty linked list**

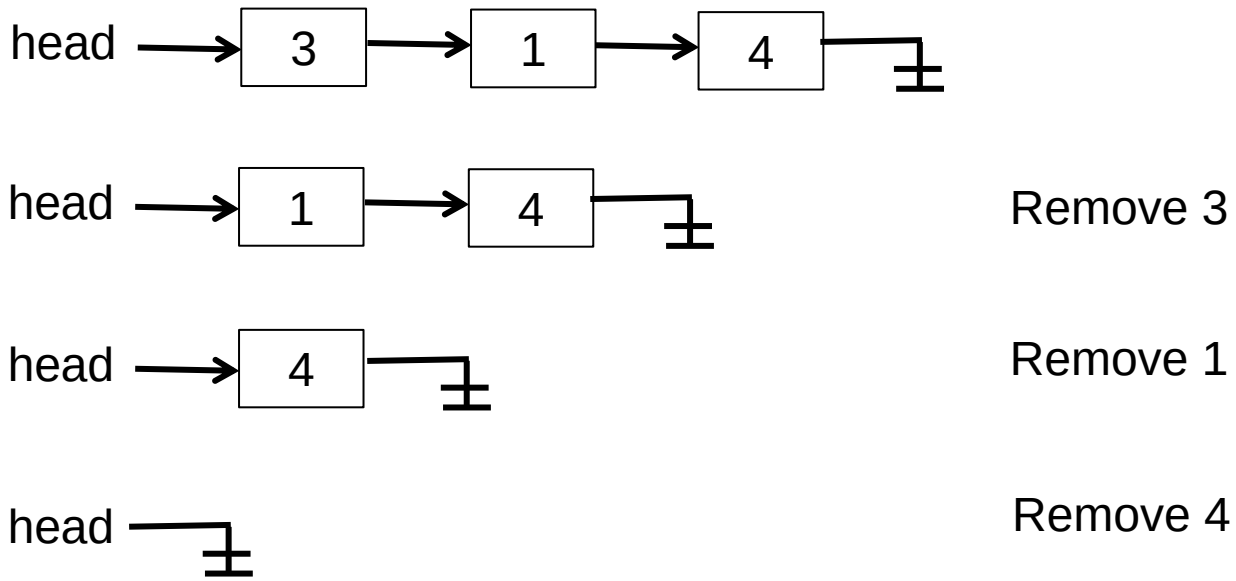



- **When we insert values on the queue we insert new nodes at the tail of the linked list**



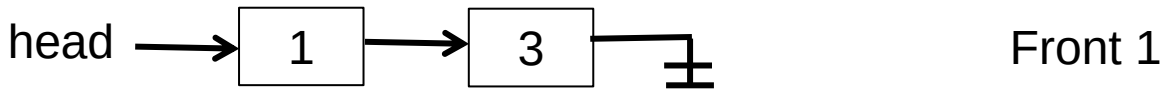
# LINKED LIST BASED

- **When we remove values from the queue we delete nodes from the head of the linked list**



# LINKED LIST BASED

- **To get the front of the queue, we return the first value in the linked list, without removing it from the list**




- **A linked list queue can not become full unless our program runs out of memory on the heap**
- **A linked list queue is empty when the head pointer is null**

# LINKED LIST BASED

```
class Queue
{
 public:
    // Constructors
    Queue();
    Queue(const Queue& queue);
    ~Queue();

    // Basic methods
    void Insert(const int number);
    void Remove(int & number);
```

# LINKED LIST BASED

```
...
    // Other methods
    int GetCount();
    int GetFront();
    bool IsFull();
    bool IsEmpty();
    void Print();

private:
    QueueNode *head;
    QueueNode *tail;
};
```

# LINKED LIST BASED

```
Class QueueNode
{
public:
    int Number;
    QueueNode *Next;
};
```

This class “breaks” the information hiding principle of OOP, but we are only going to use it in the Queue class

# LINKED LIST BASED

```
Queue::Queue ()
{
    head = NULL;
    tail = NULL;
}
```

# LINKED LIST BASED

```
Queue::Queue(const Queue & queue)
{
    // Create first node
    QueueNode *copy = new QueueNode();
    Head = copy;

    // Walk list to copy nodes
    QueueNode *ptr = queue.Head;
```

# LINKED LIST BASED

```
    while (ptr != NULL)
    {
        copy->Next = new QueueNode();
        copy = copy->Next;
        copy->Number = ptr->Number;
        copy->Next = NULL;
        Tail = copy;
        ptr = ptr->Next;
    }

    // Tidy first node
    copy = Head;
    Head = copy->Next;
    delete copy;
}
```

# LINKED LIST BASED

```
Queue::~Queue()
{
    // Delete nodes from queue
    while (Head != NULL)
    {
        QueueNode *Temp = Head;
        Head = Head->Next;
        delete Temp;
    }
    Head = NULL;
    Tail = NULL;
}
```

# LINKED LIST BASED

```
void Queue::Insert(const int Number)
{
    // Allocate space for data
    QueueNode *Temp = new QueueNode;
    if (Temp == NULL) return;
    Temp->Number = Number;
    Temp->Next = NULL;

    // Insert data at tail of list
    if (IsEmpty()) Head = Temp;
    else Tail->Next = Temp;
    Tail = Temp;
}
```

This ignores insert operation if we run out of memory

We insert node at the tail of linked list

# LINKED LIST BASED

```
void Queue::Remove(int & Number)
{
    // Extract information from node
    if (IsEmpty()) return;
    int Number = Head->Number;

    // Delete first node from linked list
    QueueNode *Temp = Head;
    Head = Head->Next;
    if (IsEmpty())
       Tail = NULL;
    delete Temp;
}
```

This returns 0 is queue is empty

We delete node after updating pointers

# LINKED LIST BASED

```
int Queue::GetFront()
{
    // Extract information from node
    if (IsEmpty()) return 0;
    int Number = Head->Number;

    // Return front value
    return Number;
}
```

This returns 0 is queue is empty

# LINKED LIST BASED

```
// True if queue is full
bool Queue::IsFull()
{
   return false;
}

// True if queue is empty
bool Queue::IsEmpty()
{
   return (Head == NULL);
}
```

# LINKED LIST BASED

```
void Queue::Print()
{
    cout << ”queue: ";
    QueueNode *Temp = Head;
    while (Temp != NULL)
    {
        cout << Temp->Number << " ";
        Temp = Temp->Next;
    }
    cout << endl;
}
```

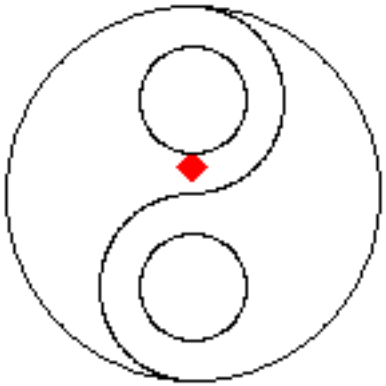
# QUEUES

## APPLICATION:

## POLYGON FLOOD FILL

# POLYGON FLOOD FILL

- **Flood fill is an algorithm used in most paint packages to fill in the interior of a line drawing**
  - User draws the object outline
  - User selects a seed point inside the object
  - User selects the desired color
  - Algorithm simulates “flooding” to fill region



Queue based flood fill demo from Wikipedia

# POLYGON FLOOD FILL

- **Flood fill can be implemented recursively as follows:**
  - We start at seed location (x,y) in picture
  - If pixel(x,y) is not already colored, we color this pixel and make four recursive calls to fill in adjacent locations

```
floodfill(x+1, y);
floodfill(x-1, y);
floodfill(x, y+1);
floodfill(x, y-1);
```

  - Recursion terminates if the pixel is already colored (or if the location is outside the boundary of the image)
  - If the flood fill region is large, this could result in millions of recursive calls and crash the program

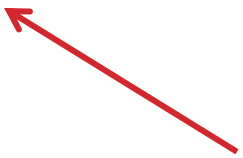
# POLYGON FLOOD FILL

```
void floodfill(int picture[SIZE][SIZE],
    int x, int y, int value)
{
    // Check terminating condition
    if ((x >= 0) && (x < SIZE) &&
        (y >= 0) && (y < SIZE) &&
        (picture[y][x] != value))
    {
        // Paint this pixel
        picture[y][x] = value;
```

Checking we are inside array bounds before checking pixel

# POLYGON FLOOD FILL

```
  ...

        // Visit four neighbors
        floodfill(picture, x+1, y, value);
        floodfill(picture, x-1, y, value);
        floodfill(picture, x, y+1, value);
        floodfill(picture, x, y-1, value);
    }
}
```



Four recursive calls to visit the four adjacent locations

# POLYGON FLOOD FILL

- **Flood fill can also be implemented using a queue:**
  - We start by inserting the seed location (x,y) on queue
  - We loop until the queue is empty
  - We remove(x,y) location of current point
  - If pixel(x,y) is not already colored, we color this pixel and save adjacent locations on queue

```
insert(x+1, y);
insert(x-1, y);
insert(x, y+1);
insert(x, y-1);
```

  - We stop filling when the queue is empty
  - This method is faster and safer than recursive flood fill

# POLYGON FLOOD FILL

```
void floodfill(int picture[SIZE][SIZE],
    int startx, int starty, int value)
{
    // Push start point on queue
    Queue q;
    q.Insert(startx); q.Insert(starty);

    // Loop while queue not empty
    while (!q.IsEmpty())
    {
```

We insert two values for (x,y) location

# POLYGON FLOOD FILL

```
// Remove next point from queue
int x = 0;
int y = 0;
q.Remove(x); q.Remove(y);

// Check if pixel is painted
if ((x >= 0) && (x < SIZE) &&
    (y >= 0) && (y < SIZE) &&
    (picture[y][x] != value))
{
```

We remove two values to get next (x,y) location

Checking we are inside array bounds before checking pixel

# POLYGON FLOOD FILL

```
            // Paint this pixel
            picture[y][x] = value;

            // Insert four neighbors
            q.Insert(x+1); q.Insert(y);
            q.Insert(x-1); q.Insert(y);
            q.Insert(x); q.Insert(y+1);
            q.Insert(x); q.Insert(y-1);
        }
    }
}
```

We insert two values for each of the four (x,y) locations

# POLYGON FLOOD FILL

- **We showed how flood fill can be implemented using recursion or using a queue to store pixel locations**
  - In the recursive floodfill code we visited the four adjacent (x,y) locations in RLTB order
  - In the queue based floodfill code we remove the points in FIFO order so the fill looks more like paint spreading out

- **We could reduce the queue size by checking if each (x,y) location is in bounds and colored before pushing**
  - This is a classic space-time tradeoff
  - See full solution on class website

# QUEUES

## APPLICATION:

## DISCRETE EVENT SIMULATION

# DISCRETE EVENT SIMULATION

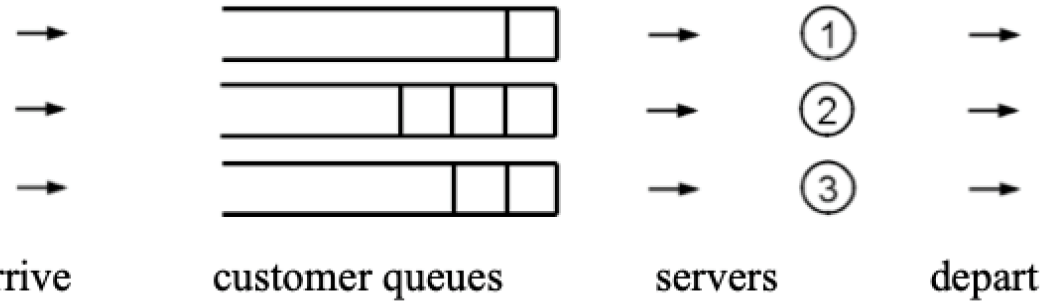
- **Queues can be used to model activities where a customer waits in line for service**
  - Bank teller windows
  - Supermarket checkouts
  - Gas station pumps
  - Amusement park rides

- **We store customer information in a queue when they arrive and remove them when they get service**
  - We can modify simulation to maximize throughput or minimize the number of customers who “give up”

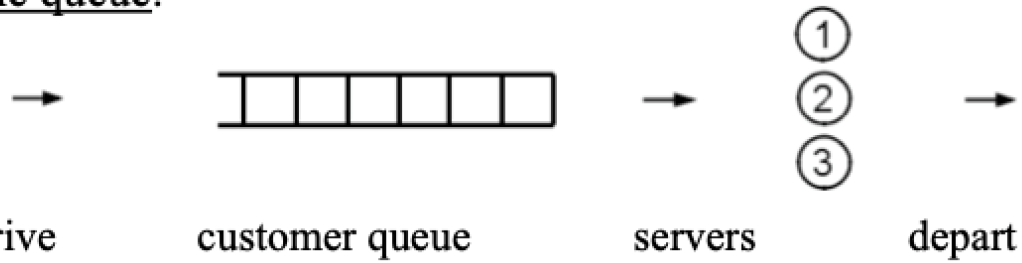
# DISCRETE EVENT SIMULATION

- **Simulation setup**
  - Decide how many queues and servers to use

multiple queues:



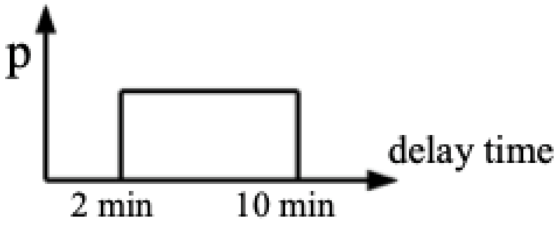

single queue:



# DISCRETE EVENT SIMULATION

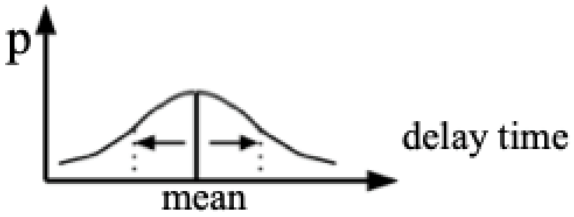
- **Simulation setup**
  - Create models for customer arrivals and service time
  - Use a uniform or normal distribution of times between customer arrivals and times to serve each customer

uniform distribution




- Specify the min and max delay time between customers in seconds
- delay = min+random() % (max-min+1)

normal distribution




- Specify the mean and standard deviation of delay time distribution
- Use methods in <random> to generate delay values

# DISCRETE EVENT SIMULATION

- **Simulation algorithm**
  - Create N empty queues
  - Start virtual clock at zero
  - Loop for C customers (or until time T is reached)
    - Calculate arrival time for next customer
    - Calculate service time for this customer
    - Update the virtual clock
    - Check all queues and remove serviced customers
    - Add new customer to the shortest queue
  - We can calculate min/max/ave customer waiting time
  - We can simulate customers “giving up” if queue is too long
  - We can test several models to improve customer service

# QUEUES

## APPLICATION:
## FAIR SCHEDULING

# FAIR SCHEDULING

- **Queues are used in a number of software applications to provide fair service**
  - Queues store information in first in first out (FIFO) order so they are ideal for “first come first served” applications

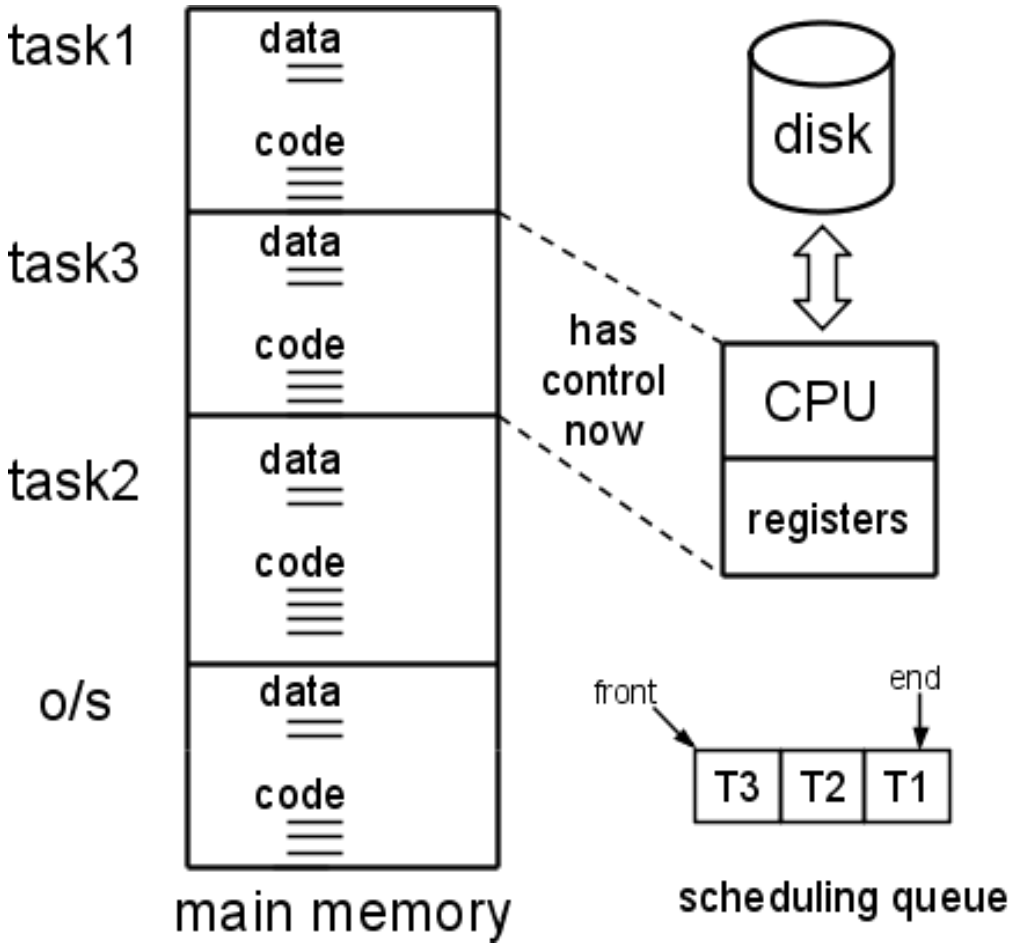
- **Printer queues:**
  - Network printer provides shared service to many users
  - Print requests are added to the end of printer queue
  - Printer removes documents from the printer queue and prints them in FIFO order

# FAIR SCHEDULING

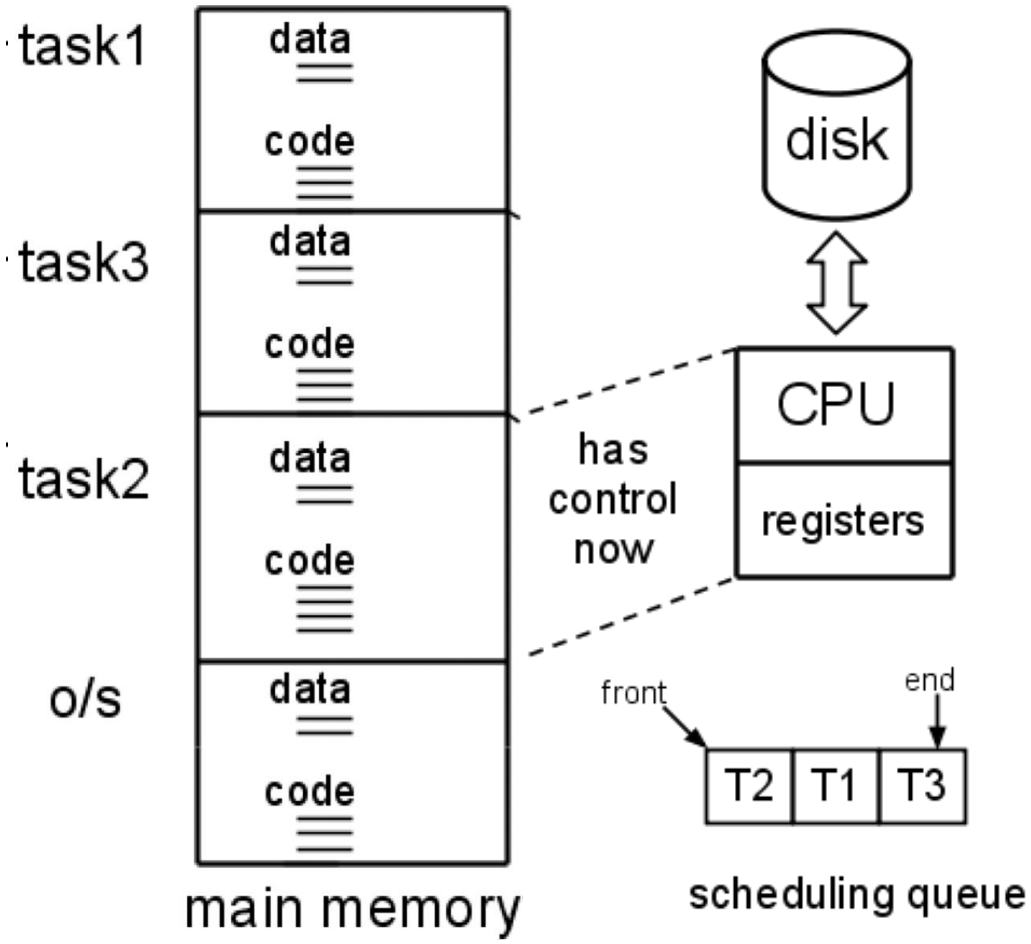
- **Communication buffers:**
  - Hub or switch stores incoming data packets in a queue
  - Packets are processed and transmitted in FIFO order
  - Queue prevents data loss if there is temporary congestion

- **Process scheduling on CPU:**
  - Scheduling queue keeps track of all running tasks on OS
  - Scheduler will remove task at front of scheduling queue
  - This task gets a small “time slice” of the CPU
  - When time slice is up, the task is “paused” and added to the end of the schedule queue
  - Tasks are removed from queue when they terminate

# FAIR SCHEDULING



# FAIR SCHEDULING



# QUEUES

## SUMMARY

# SUMMARY

- **Queues are a very simple abstract data type that store data in a first in first out (FIFO) order**
  - We can only store data using insert
  - We can only access data using remove
- **Queues can be implemented using arrays or linked lists**
  - Array implementation is much faster but can get full
  - Linked list implementation can never get full but is slower
- **Queues can be used to solve wide range of problems**
  - Polygon flood fill, discrete event simulation, fair scheduling