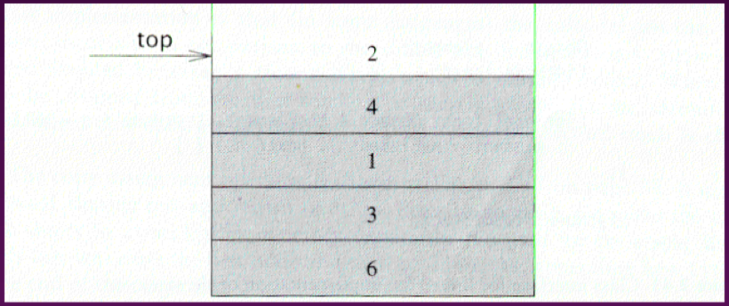
Stack and Queue

Stack Overview

- Stack ADT
- Basic operations of stack
  - Pushing, popping etc.
- Implementations of stacks using
  - array
  - linked list

The Stack ADT

- A stack is a list with the restriction
  - that insertions and deletions can only be performed at the *top* of the list



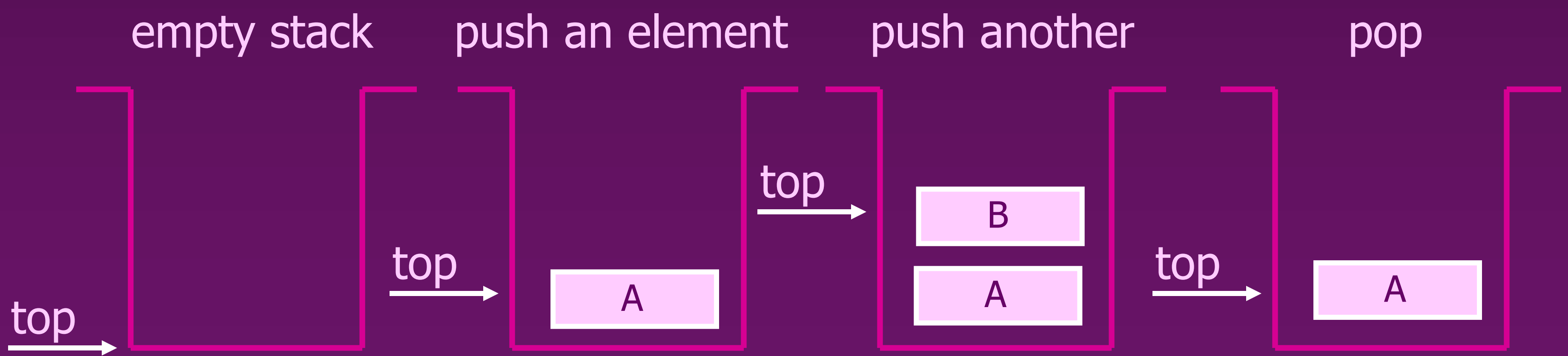
  - The other end is called bottom
- Fundamental operations:
  - Push: Equivalent to an insert
  - Pop: Deletes the most recently inserted element
  - Top: Examines the most recently inserted element

Stack ADT

- Stacks are less flexible
  - ✓ but are more efficient and easy to implement
- Stacks are known as LIFO (Last In, First Out) lists.
  - The last element inserted will be the first to be retrieved

Push and Pop

- Primary operations: Push and Pop
- Push
  - Add an element to the top of the stack
- Pop
  - Remove the element at the top of the stack



Implementation of Stacks

- Any list implementation could be used to implement a stack
  - Arrays (static: the size of stack is given initially)
  - Linked lists (dynamic: never become full)
- We will explore implementations based on array and linked list
- Let’s see how to use an array to implement a stack first

Array Implementation

- Need to declare an array size ahead of time
- Associated with each stack is TopOfStack
  - for an empty stack, set TopOfStack to -1
- Push
  - (1) Increment TopOfStack by 1.
  - (2) Set Stack[TopOfStack] = X
- Pop
  - (1) Set return value to Stack[TopOfStack]
  - (2) Decrement TopOfStack by 1
- These operations are performed in very fast constant time

Stack class

```
class Stack {
public:
      Stack(int size = 10);                    // constructor
      ~Stack() { delete [] values; }           // destructor
      bool IsEmpty() { return top == -1; }
      bool IsFull() { return top == maxTop; }
      double Top();
      void Push(const double x);
      double Pop();
      void DisplayStack();
private:
      int maxTop;          // max stack size = size - 1
      int top;             // current top of stack
      double* values;      // element array
};
```

`Stack` class

- Attributes of `Stack`
  - `maxTop`: the max size of stack
  - `top`: the index of the top element of stack
  - `values`: point to an array which stores elements of stack
- Operations of `Stack`
  - `IsEmpty`: return true if stack is empty, return false otherwise
  - `IsFull`: return true if stack is full, return false otherwise
  - `Top`: return the element at the top of stack
  - `Push`: add an element to the top of stack
  - `Pop`: delete the element at the top of stack
  - `DisplayStack`: print all the data in the stack

Create Stack

- The constructor of `Stack`
  - Allocate a stack array of `size`. By default, `size = 10`.
  - When the stack is full, `top` will have its maximum value, i.e. `size - 1`.
  - Initially `top` is set to -1. It means the stack is empty.

```
Stack::Stack(int size /*= 10*/) {
      maxTop      =     size - 1;
      values      =     new double[size];
      top         =     -1;
}
```

Although the constructor dynamically allocates the stack array, the stack is still static. The size is fixed after the initialization.

Push Stack

- `void Push(const double x);`
  - Push an element onto the stack
  - If the stack is full, print the error information.
  - Note `top` always represents the index of the top element. After pushing an element, increment `top`.

```
void Stack::Push(const double x) {
      if (IsFull())
            cout << "Error: the stack is full." << endl;
      else
            values[++top]     =     x;
}
```

Pop Stack

- `double Pop()`
  - Pop and return the element at the top of the stack
  - If the stack is empty, print the error information. (In this case, the return value is useless.)
  - Don’t forgot to decrement `top`

```
double Stack::Pop() {
	if (IsEmpty()) {
		cout << "Error: the stack is empty." << endl;
		return -1;
	}
	else {
		return values[top--];
	}
}
```

Stack Top

- `double Top()`
  - Return the top element of the stack
  - Unlike `Pop`, this function does not remove the top element

```
double Stack::Top() {
      if (IsEmpty()) {
            cout << "Error: the stack is empty." << endl;
            return -1;
      }
      else
            return values[top];
}
```

Printing all the elements

- `void DisplayStack()`
  - Print all the elements

```
void Stack::DisplayStack() {
      cout << "top -->";
      for (int i = top; i >= 0; i--)
            cout << "\t|\t" << values[i] << "\t|" << endl;
      cout << "\t|-----------------|" << endl;
}
```

```
top -->  |        -8        |
         |        -3        |
         |        6.5       |
         |        5         |
         -------------------
```

Using Stack

```
int main(void) {
        Stack stack(5);
        stack.Push(5.0);
        stack.Push(6.5);
        stack.Push(-3.0);
        stack.Push(-8.0);
        stack.DisplayStack();
        cout << "Top: " << stack.Top() << endl;

        stack.Pop();
        cout << "Top: " << stack.Top() << endl;
        while (!stack.IsEmpty()) stack.Pop();
        stack.DisplayStack();
        return 0;
}
```

result

```
top --> |      -8      |
        |      -3      |
        |      6.5     |
        |      5       |
        |--------------|
Top: -8
Top: -3
top --> |--------------|
```

Implementation based on Linked List

- Now let us implement a stack based on a linked list
- To make the best out of the code of `List`, we implement `Stack` by inheriting `List`
  - To let `Stack` access private member `head`, we make `Stack` as a friend of `List`

```
class List {
public:
        List(void) { head = NULL; }          // constructor
        ~List(void);                         // destructor
        bool IsEmpty() { return head == NULL; }
        Node* InsertNode(int index, double x);
        int FindNode(double x);
        int DeleteNode(double x);
        void DisplayList(void);
private:
        Node* head;
        friend class Stack;
};
```

Implementation based on Linked List

```
-8
-3
6.5
5
Number of nodes in the list: 4
Top: -8
Top: -3
Number of nodes in the list: 0
```

```
class Stack : public List {
public:
	Stack() {}		// constructor
	~Stack() {}		// destructor
	double Top() {
		if (head == NULL) {
			cout << "Error: the stack is empty." << endl;
			return -1;
		}
		else
			return head->data;
	}
	void Push(const double x) { InsertNode(0, x); }
	double Pop() {
		if (head == NULL) {
			cout << "Error: the stack is empty." << endl;
			return -1;
		}
		else {
			double val = head->data;
			DeleteNode(val);
			return val;
		}
	}
	void DisplayStack() { DisplayList(); }
};
```

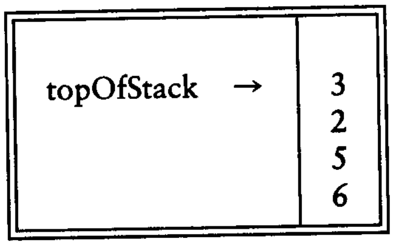
Note: the stack implementation based on a linked list will never be full.

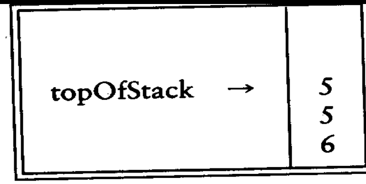
Balancing Symbols

- To check that every right brace, bracket, and parentheses must correspond to its left counterpart
  - e.g. [( )] is legal, but [( ]) is illegal
- Algorithm
  (1) Make an empty stack.
  (2) Read characters until end of file
    i. If the character is an opening symbol, push it onto the stack
    ii. If it is a closing symbol, then if the stack is empty, report an error
    iii. Otherwise, pop the stack. If the symbol popped is not the corresponding opening symbol, then report an error
  (3) At end of file, if the stack is not empty, report an error

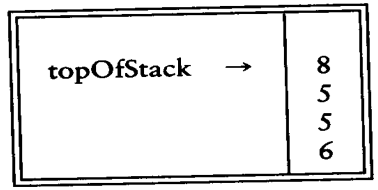
Postfix Expressions

- Calculate 4.99 * 1.06 + 5.99 + 6.99 * 1.06
  - Need to know the precedence rules
- Postfix (reverse Polish) expression
  - 4.99 1.06 * 5.99 + 6.99 1.06 * +
- Use stack to evaluate postfix expressions
  - When a number is seen, it is pushed onto the stack
  - When an operator is seen, the operator is applied to the 2 numbers that are popped from the stack. The result is pushed onto the stack
- Example
  - evaluate 6 5 2 3 + 8 * + 3 + *
- The time to evaluate a postfix expression is O(N)
  - processing each element in the input consists of stack operations and thus takes constant time

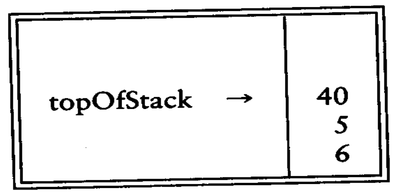




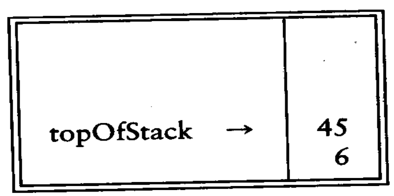
Next 8 is pushed.



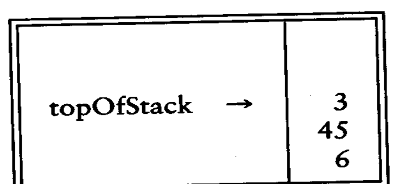
Now a '*' is seen, so 8 and 5 are popped and $5 * 8 = 40$ is pushed.



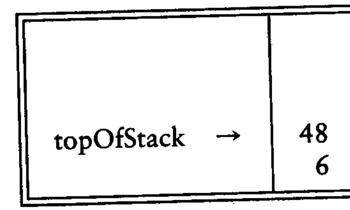
Next a '+' is seen, so 40 and 5 are popped and $5 + 40 = 45$ is pushe



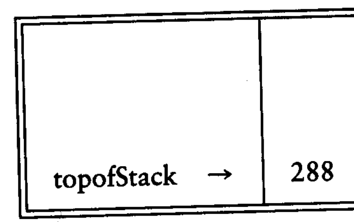
Now, 3 is pushed.



Next '+' pops 3 and 45 and pushes $45 + 3 = 48$.



and 48 and 6 are popped; the result, $6 * 4$



Queue Overview

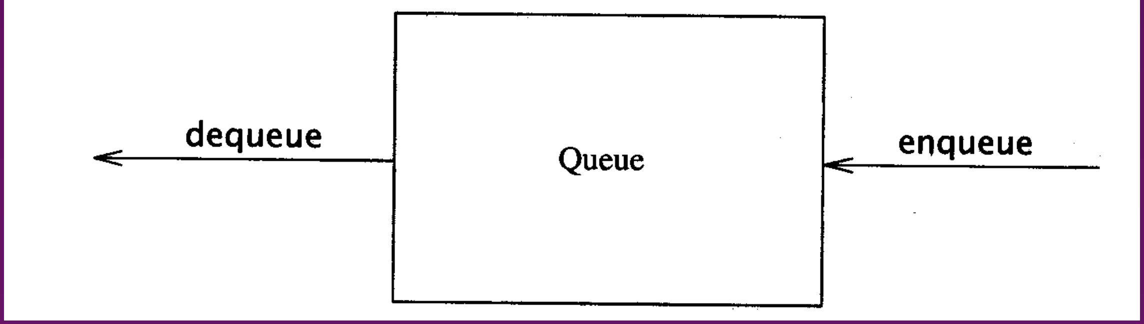
- Queue ADT
- Basic operations of queue
  - Enqueuing, dequeuing etc.
- Implementation of queue
  - Array
  - Linked list

Queue ADT

- Like a stack, a *queue* is also a list. However, with a queue, insertion is done at one end, while deletion is performed at the other end.
- Accessing the elements of queues follows a First In, First Out (FIFO) order.
  - Like customers standing in a check-out line in a store, the first customer in is the first customer served.

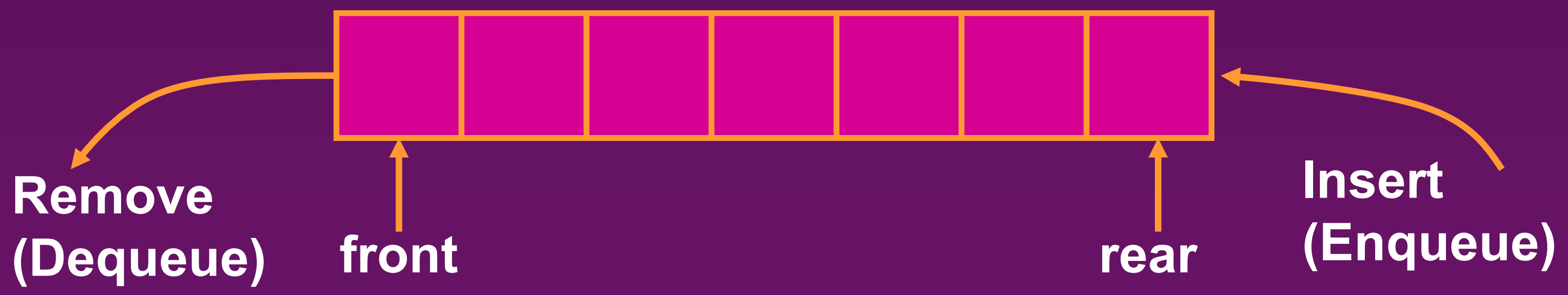
The Queue ADT

- Another form of restricted list
  - Insertion is done at one end, whereas deletion is performed at the other end
- Basic operations:
  - enqueue: insert an element at the rear of the list
  - dequeue: delete the element at the front of the list



Enqueue and Dequeue

- Primary queue operations: Enqueue and Dequeue
- Like check-out lines in a store, a queue has a front and a rear.
- Enqueue
  - Insert an element at the rear of the queue
- Dequeue
  - Remove an element from the front of the queue

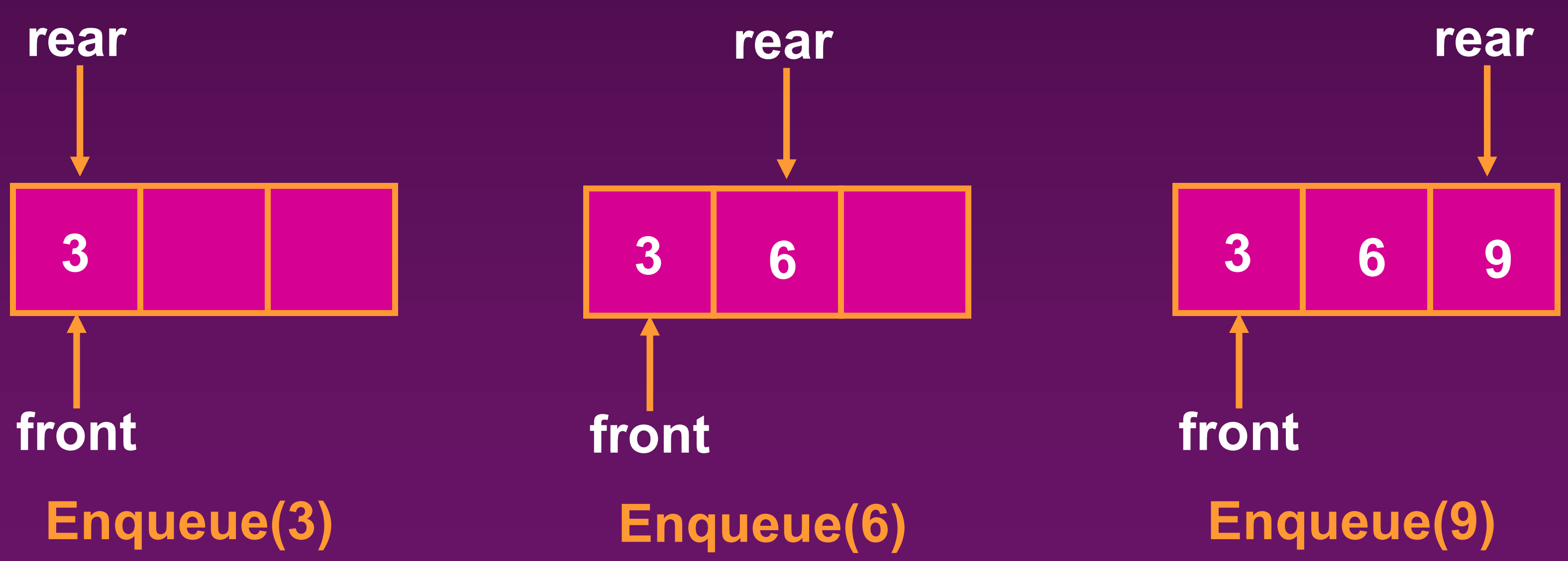


Implementation of Queue

- Just as stacks can be implemented as arrays or linked lists, so with queues.
- Dynamic queues have the same advantages over static queues as dynamic stacks have over static stacks

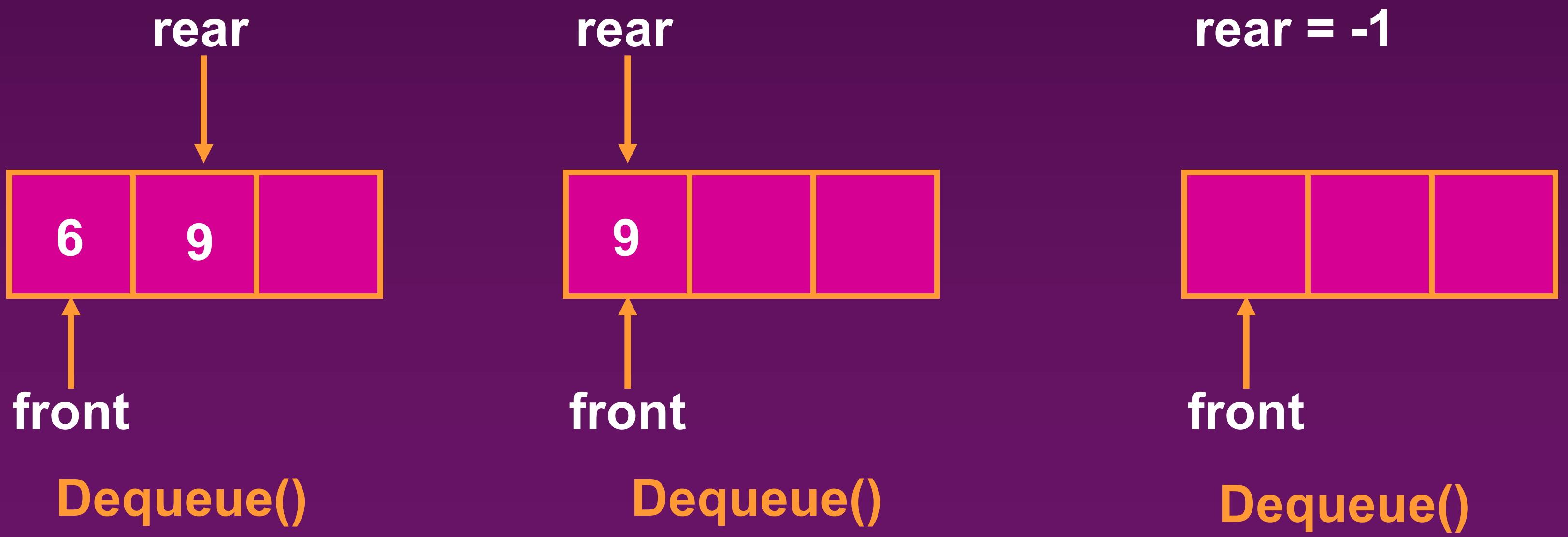
Queue Implementation of Array

- There are several different algorithms to implement Enqueue and Dequeue
- Naïve way
  - When enqueuing, the front index is always fixed and the rear index moves forward in the array.



Queue Implementation of Array

- Naïve way
  - When enqueuing, the front index is always fixed and the rear index moves forward in the array.
  - When dequeuing, the element at the front the queue is removed. Move all the elements after it by one position. (Inefficient!!!)



Queue Implementation of Array

- Better way
  - When an item is enqueued, make the rear index move forward.
  - When an item is dequeued, the front index moves by one element towards the back of the queue (thus removing the front item, so no copying to neighboring elements is needed).

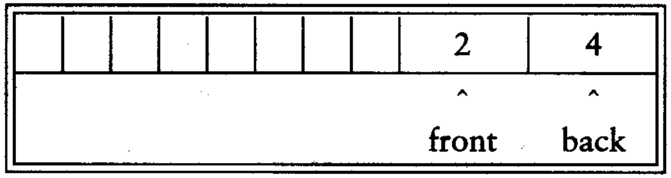
(front) XXXXOOOOO (rear)
OXXXXOOOO (after 1 dequeue, and 1 enqueue)
OOXXXXXOO (after another dequeue, and 2 enqueues)
OOOOXXXXX (after 2 more dequeues, and 2 enqueues)

The problem here is that the rear index cannot move beyond the last element in the array.

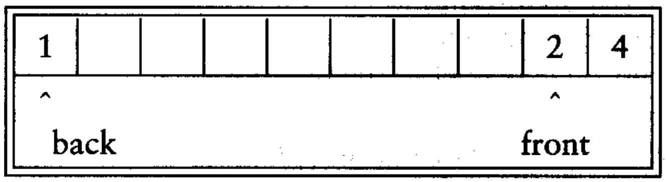
Implementation using Circular Array

- Using a circular array
- When an element moves past the end of a circular array, it wraps around to the beginning, e.g.
  - OOOOO7963 → 4OOOO7963 (after Enqueue(4))
  - After Enqueue(4), the rear index moves from 3 to 4.

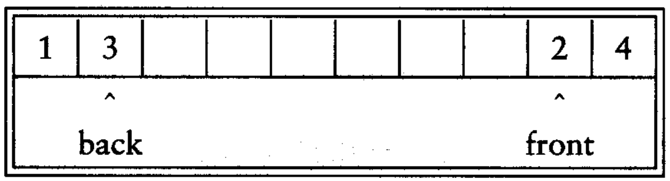
Initial State



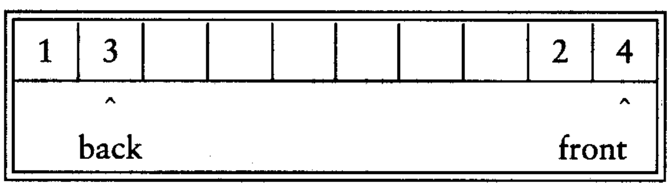
After enqueue(1)



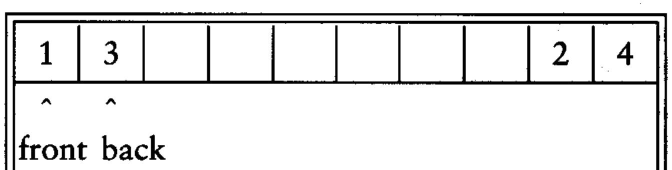
After enqueue(3)



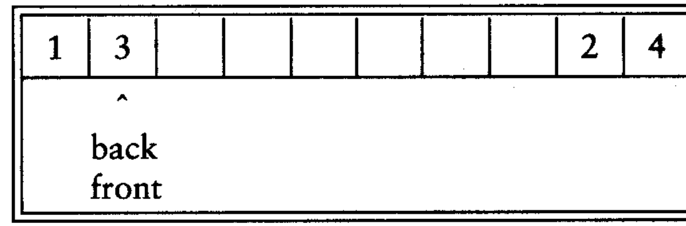
After dequeue, Which Returns 2



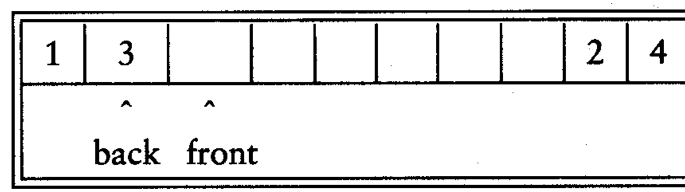
After dequeue, Which Returns 4



After dequeue, Which Returns 1



After dequeue, Which Returns 3 and Makes the Queue Empty



Empty or Full?

- Empty queue
  - back = front - 1
- Full queue?
  - the same!
  - Reason: n values to represent n+1 states
- Solutions
  - Use a boolean variable to say explicitly whether the queue is empty or not
  - Make the array of size n+1 and only allow n elements to be stored
  - Use a counter of the number of elements in the queue

Queue Implementation of Linked List

```
class Queue {
public:
        Queue(int size = 10);                   // constructor
        ~Queue() { delete [] values; }          // destructor
        bool IsEmpty(void);
        bool IsFull(void);
        bool Enqueue(double x);
        bool Dequeue(double & x);
        void DisplayQueue(void);
private:
        int front;              // front index
        int rear;               // rear index
        int counter;            // number of elements
        int maxSize;            // size of array queue
        double* values;         // element array
};
```

`Queue` Class

- Attributes of `Queue`
  - `front/rear`: front/rear index
  - `counter`: number of elements in the queue
  - `maxSize`: capacity of the queue
  - `values`: point to an array which stores elements of the queue
- Operations of `Queue`
  - `IsEmpty`: return true if queue is empty, return false otherwise
  - `IsFull`: return true if queue is full, return false otherwise
  - `Enqueue`: add an element to the rear of queue
  - `Dequeue`: delete the element at the front of queue
  - `DisplayQueue`: print all the data

Create Queue

- `Queue(int size = 10)`
  - Allocate a queue array of `size`. By default, `size = 10`.
  - `front` is set to `0`, pointing to the first element of the array
  - `rear` is set to `-1`. The queue is empty initially.

```
Queue::Queue(int size /* = 10 */) {
      values      =     new double[size];
      maxSize     =     size;
      front       =     0;
      rear        =     -1;
      counter     =     0;
}
```

IsEmpty & IsFull

- Since we keep track of the number of elements that are actually in the queue: `counter`, it is easy to check if the queue is empty or full.

```
bool Queue::IsEmpty() {
      if (counter)          return false;
      else                  return true;
}
bool Queue::IsFull() {
      if (counter < maxSize)   return false;
      else                     return true;
}
```

Enqueue

```
bool Queue::Enqueue(double x) {
      if (IsFull()) {
            cout << "Error: the queue is full." << endl;
            return false;
      }
      else {
            // calculate the new rear position (circular)
            rear             = (rear + 1) % maxSize;
            // insert new item
            values[rear]     = x;
            // update counter
            counter++;
            return true;
      }
}
```

Dequeue

```
bool Queue::Dequeue(double & x) {
	if (IsEmpty()) {
		cout << "Error: the queue is empty." << endl;
		return false;
	}
	else {
		// retrieve the front item
		x            = values[front];
		// move front
		front = (front + 1) % maxSize;
		// update counter
		counter--;
		return true;
	}
}
```

Printing the elements

```
front -->       0
                1
                2
                3
                4       <-- rear
```

```
void Queue::DisplayQueue() {
      cout << "front -->";
      for (int i = 0; i < counter; i++) {
            if (i == 0) cout << "\t";
            else        cout << "\t\t";
            cout << values[(front + i) % maxSize];
            if (i != counter - 1)
                  cout << endl;
            else
                  cout << "\t<-- rear" << endl;
      }
}
```

Using Que

```
Enqueue 5 items.
Now attempting to enqueue again...
Error: the queue is full.
front -->       0
                1
                2
                3
                4       <-- rear
Retrieved element = 0
front -->       1
                2
                3
                4       <-- rear
front -->       1
                2
                3
                4
                7       <-- rear
```

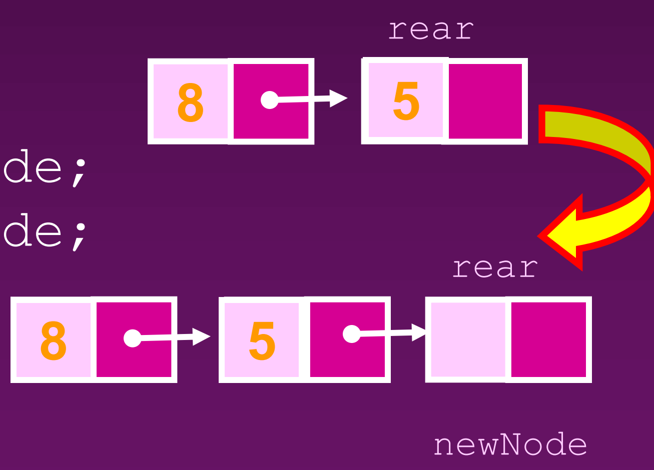
```
int main(void) {
	Queue queue(5);
	cout << "Enqueue 5 items." << endl;
	for (int x = 0; x < 5; x++)
		queue.Enqueue(x);
	cout << "Now attempting to enqueue again..." << endl;
	queue.Enqueue(5);
	queue.DisplayQueue();
	double value;
	queue.Dequeue(value);
	cout << "Retrieved element = " << value << endl;
	queue.DisplayQueue();
	queue.Enqueue(7);
	queue.DisplayQueue();
	return 0;
}
```

Stack Implementation based on Linked List

```
class Queue {
public:
        Queue() {                   // constructor
                front = rear = NULL;
                counter = 0;
        }
        ~Queue() {                  // destructor
                double value;
                while (!IsEmpty()) Dequeue(value);
        }
        bool IsEmpty() {
                if (counter)        return false;
                else                return true;
        }
        void Enqueue(double x);
        bool Dequeue(double & x);
        void DisplayQueue(void);
private:
        Node* front;    // pointer to front node
        Node* rear;     // pointer to last node
        int counter;    // number of elements
};
```

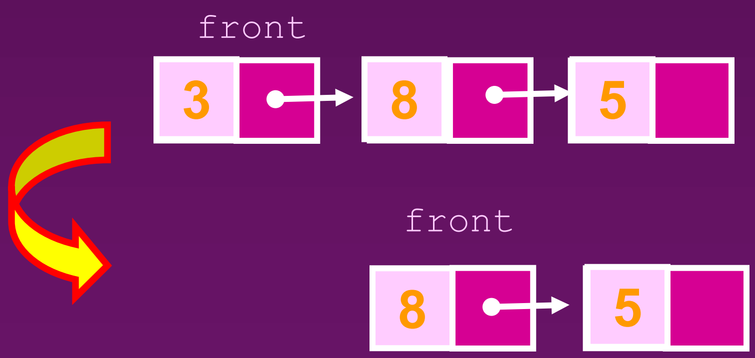
Enqueue

```
void Queue::Enqueue(double x) {
      Node* newNode      =     new Node;
      newNode->data      =     x;
      newNode->next      =     NULL;
      if (IsEmpty()) {
            front        =     newNode;
            rear         =     newNode;
      }
      else {
            rear->next   =     newNode;
            rear         =     newNode;
      }
      counter++;
}
```



Deque

```
bool Queue::Dequeue(double & x) {
	if (IsEmpty()) {
		cout << "Error: the queue is empty." << endl;
		return false;
	}
	else {
		x		=	front->data;
		Node* nextNode	=	front->next;
		delete front;
		front		=	nextNode;
		counter--;
	}
}
```



Printing all the elements

```
void Queue::DisplayQueue() {
      cout << "front -->";
      Node* currNode      =      front;
      for (int i = 0; i < counter; i++) {
            if (i == 0) cout << "\t";
            else            cout << "\t\t";
            cout << currNode->data;
            if (i != counter - 1)
                  cout << endl;
            else
                  cout << "\t<-- rear" << endl;
            currNode      =      currNode->next;
      }
}
```

```
Enqueue 5 items.
Now attempting to enqueue again..
front -->       0
                1
                2
                3
                4
                5       <-- rear
Retrieved element = 0
front -->       1
                2
                3
                4
                5       <-- rear
front -->       1
                2
                3
                4
                5
                7       <-- rear
```

Result

- Queue implemented using linked list will be never full

```
Enqueue 5 items.
Now attempting to enqueue again...
Error: the queue is full.
front -->        0
                 1
                 2
                 3
                 4        <-- rear
Retrieved element = 0
front -->        1
                 2
                 3
                 4        <-- rear
front -->        1
                 2
                 3
                 4
                 7        <-- rear
```

based on array

```
Enqueue 5 items.
Now attempting to enqueue again..
front -->        0
                 1
                 2
                 3
                 4
                 5        <-- rear
Retrieved element = 0
front -->        1
                 2
                 3
                 4
                 5        <-- rear
front -->        1
                 2
                 3
                 4
                 5
                 7        <-- rear
```

based on linked list