Introduction to the Stack

- **Stack**: a data structure that holds a collection of elements of the same type.
  - The elements are accessed according to LIFO order: last in, first out
  - No random access to other elements

- **Examples**:
  - plates in a cafeteria
  - bangles . . .

Stack Operations

- **Operations**:
  - **push**: add a value onto the top of the stack
    - make sure it’s not full first.
  - **pop**: remove (and return) the value from the top of the stack
    - make sure it’s not empty first.
  - **isFull**: true if the stack is currently full, i.e., has no more space to hold additional elements
  - **isEmpty**: true if the stack currently contains no elements

```cpp
int item;
stack.push(2);
stack.push(3);
stack.push(5);
item = stack.pop(); //item is 5
item = stack.pop(); //item is 3
stack.push(10);
```
Stack Applications

- Easily reverse the order of a list of items.
  - push all the items, then pop while not empty.
- Evaluate an expression in postfix notation.
  - 4 5 + 7 2 - * is equivalent to (4+5)*(7-2)
  - push numbers, when operator encountered, pop top two values, apply operator, push result.
- Matching brackets in a text file
  - if (x==list.getCurrent()) { z[i] = x; count++; }
- Implement nested function calls and returns.

Implementing a Stack Class

- Array implementations:
  - fixed size (static) arrays: size doesn’t change
  - dynamic arrays: can resize as needed in push
- Linked List
  - grow and shrink in size as needed
- Templates
  - any of the above can be implemented using templates

A static stack class

class IntStack
{
private:
  const int STACKSIZE = 100;   // The stack size
  int stackArray[STACKSIZE];  // The stack array
  int top;                     // Index to the top of the stack
public:
  // Constructor
  IntStack();

  // Stack operations
  void push(int);
  int pop();
  bool isFull() const;
  bool isEmpty() const;
};

A static stack class: functions

//***********************************************
// Constructor                                  *
// This constructor creates an empty stack.     *
//***********************************************
IntStack::IntStack()
{
  top = -1;                                   // empty
}

//no need to initialize the static array stackArray
A static stack class: push

```cpp
void IntStack::push(int num)
{
    // assert(!isFull());
    assert(!isFull());
    top++;
    stackArray[top] = num;
}
```

A static stack class: pop

```cpp
int IntStack::pop()
{
    // assert(!isEmpty());
    assert(!isEmpty());
    int num = stackArray[top];
    top--;
    return num;
}
```

A static stack class: functions

```cpp
bool IntStack::isFull() const
{
    return (top == stackSize - 1);
}
```

```cpp
bool IntStack::isEmpty() const
{
    return (top == -1);
}
```

A Dynamic Stack Class: Linked List implementation

- **Push and pop from the head of the list:**

```cpp
void IntStack2::push(int num)
{
    // assert(!isFull());
    assert(!isFull());
    Node *temp=new Node;
    temp->data = num;
    // insert at head of list
    temp->next = head;
    head = temp;
}
```
A Dynamic Stack Class:
Linked List implementation

• Push and pop from the head of the list:

```cpp
int IntStack2::pop()
{
    assert(!isEmpty());
    int result = head->data;
    Node* temp = head;
    head = head->next;
    delete temp;
    return result;
}
```

Introduction to the Queue

• **Queue**: a data structure that holds a collection of elements of the same type.
  - The elements are accessed according to FIFO order: first in, first out
  - No random access to other elements

• Examples:
  - people in line at a theatre box office
  - restocking perishable inventory

Queue Operations

• Operations:
  - **enqueue**: add a value onto the rear of the queue (the end of the line)
    - make sure it’s not full first.
  - **dequeue**: remove a value from the front of the queue (the front of the line) “Next!”
    - make sure it’s not empty first.
  - **isFull**: true if the queue is currently full, i.e., has no more space to hold additional elements
  - **isEmpty**: true if the queue currently contains no elements

Queue illustrated

- enqueue: add a value onto the rear of the queue (the end of the line)
  - make sure it’s not full first.
- dequeue: remove a value from the front of the queue (the front of the line) “Next!”
  - make sure it’s not empty first.
- isFull: true if the queue is currently full, i.e., has no more space to hold additional elements
- isEmpty: true if the queue currently contains no elements
Queue implemented

- Just like stacks, queues can be implemented using arrays (fixed size, or resizing dynamic arrays) or linked lists (dynamic queues) or templates.
- The previous illustration assumed we were using an array to implement the queue.
- When an item was dequeued, the items were NOT shifted up to fill the slot vacated by dequeued item.
- Instead, both front and rear indices move in the array. Why?

Implementing a Queue Class

- When front and rear indices move in the array:
  - problem: rear hits end of array quickly
  - solution: wrap index around to front of array

Implementing a Queue Class

- To “wrap” the rear index back to the front of the array, you can use this code to increment rear during enqueue:
  ```c++
  if (rear == queueSize-1)
      rear = 0;
  else
      rear = rear+1;
  ```
- The following code is equivalent, but shorter (assuming 0 <= rear < queueSize):
  ```c++
  rear = (rear + 1) % queueSize;
  ```
- Do the same for advancing the front index.

Implementing a Queue Class

- When is it full?

  ```c++
  if (rear + 1) % queueSize == front
  ```

- It’s full:
Implementing a Queue Class

• When is it empty?

```cpp
int x;
for (int i=0; i<queueSize; i++)
    x = q.dequeue();
```

after the first one:

<table>
<thead>
<tr>
<th>3</th>
<th>4</th>
<th>5</th>
<th>2</th>
<th>1</th>
<th>9</th>
<th>6</th>
</tr>
</thead>
</table>

one element left:

| 1 |
front

no elements left, front passes rear:

|     |
front
rear

• It’s empty:  

(rear+1)%queueSize==front

Implementing a Queue Class

• When is it full?  (rear+1)%queueSize==front
• When is it empty?  (rear+1)%queueSize==front
• How do we define isFull and isEmpty?
  - Use a counter variable, numItems, to keep track of the total number of items in the queue.
  - enqueue: numItems++
  - dequeue: numItems--
  - isEmpty is true when numItems == 0
  - isFull is true when numItems == queueSize

A static queue class

```cpp
class IntQueue
{
    private:
        const int QUEUESIZE = 100;  // capacity of the queue
        int queueArray[QUEUESIZE];  // The queue array
        int front;        // Subscript of the queue front
        int rear;         // Subscript of the queue rear
        int numItems;     // Number of items in the queue
    public:
        // Constructor
        IntQueue();

        // Queue operations
        void enqueue(int);
        int dequeue();
        bool isEmpty() const;
        bool isFull() const;
};
```
A static queue class: enqueue

```c++
void IntQueue::enqueue(int num)
{
    assert(!isFull());
    // Calculate the new rear position
    rear = (rear + 1) % queueSize;
    // Insert new item
    queueArray[rear] = num;
    // Update item count
    numItems++;
}
```

A static queue class: dequeue

```c++
int IntQueue::dequeue()
{
    assert(!isEmpty());
    // save the result to return
    int result = queueArray[front];
    // Advance front
    front = (front + 1) % queueSize;
    // Update item count
    numItems--;
    // Return the front item
    return result;
}
```

A static queue class: functions

```c++
bool IntQueue::isEmpty() const {
    return (numItems == 0);
}
```

```c++
bool IntQueue::isFull() const {
    return (numItems == queueSize);
}
```

A Dynamic Queue Class: Linked List implementation

- Use pointers `front` and `rear` to point to first and last elements of the list:
A Dynamic Queue Class: Linked List implementation

- Enqueue at the rear of the list, dequeue from the front:

```cpp
void IntQueue2::enqueue()
{
    assert(!isFull());
    Node *temp=new Node;
    temp->data = num;
    temp->next = NULL;
    // append to rear of list, reset rear
    if (isEmpty())
        front = rear = temp;
    else {
        rear->next = temp;
        rear = temp;
    }
}
```

```cpp
private:
    struct Node {
        int data;
        Node* next;
    }; Node* front; // ptr to first
    Node* rear; // ptr to last
```

- Dequeue removes the value at the front of the queue and returns the value:

```cpp
int IntQueue2::dequeue()
{
    assert(!isEmpty());
    int value = front->data;
    // remove the first node (front)
    Node *temp = front;
    front = front->next;
    delete temp;
    return value;
}
```

```cpp
private:
    struct Node {
        int data;
        Node* next;
    }; Node* front; // ptr to first
    Node* rear; // ptr to last
```