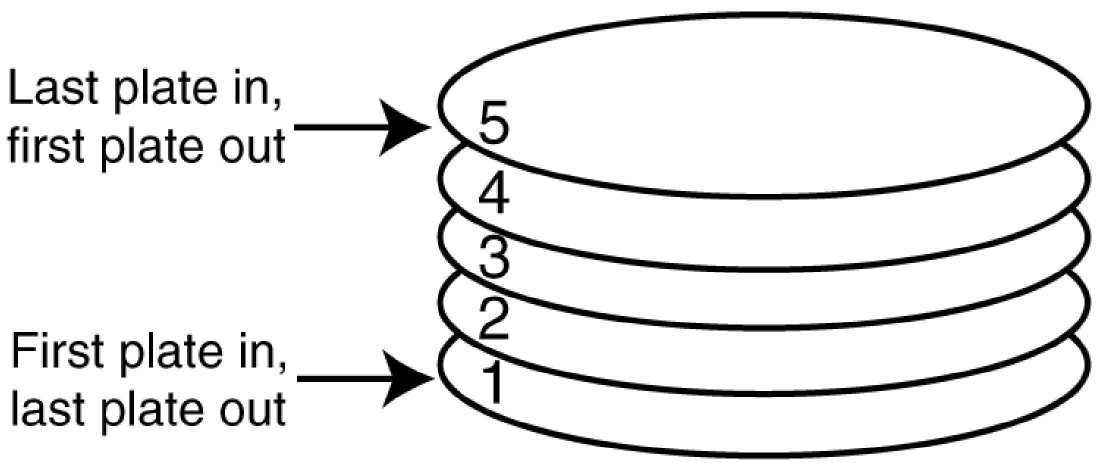
CS 2133
Stacks & Queues

Introduction to the Stack

- A stack is a data structure that stores and retrieves items in a last-in-first-out (LIFO) manner.



Applications of Stacks

- Computer systems use stacks during a program’s execution to store function return addresses, local variables, etc.
- Some calculators use stacks for performing mathematical operations.

Static and Dynamic Stacks

- Static Stacks
  - Fixed size
  - Can be implemented with an array
- Dynamic Stacks
  - Grow in size as needed
  - Can be implemented with a linked list

Stack Operations

- Push
  - causes a value to be stored in (pushed onto) top of the stack
- Pop
  - retrieves and removes a value from the top of the stack

Thus, we need to keep the index of or have a pointer to the top element

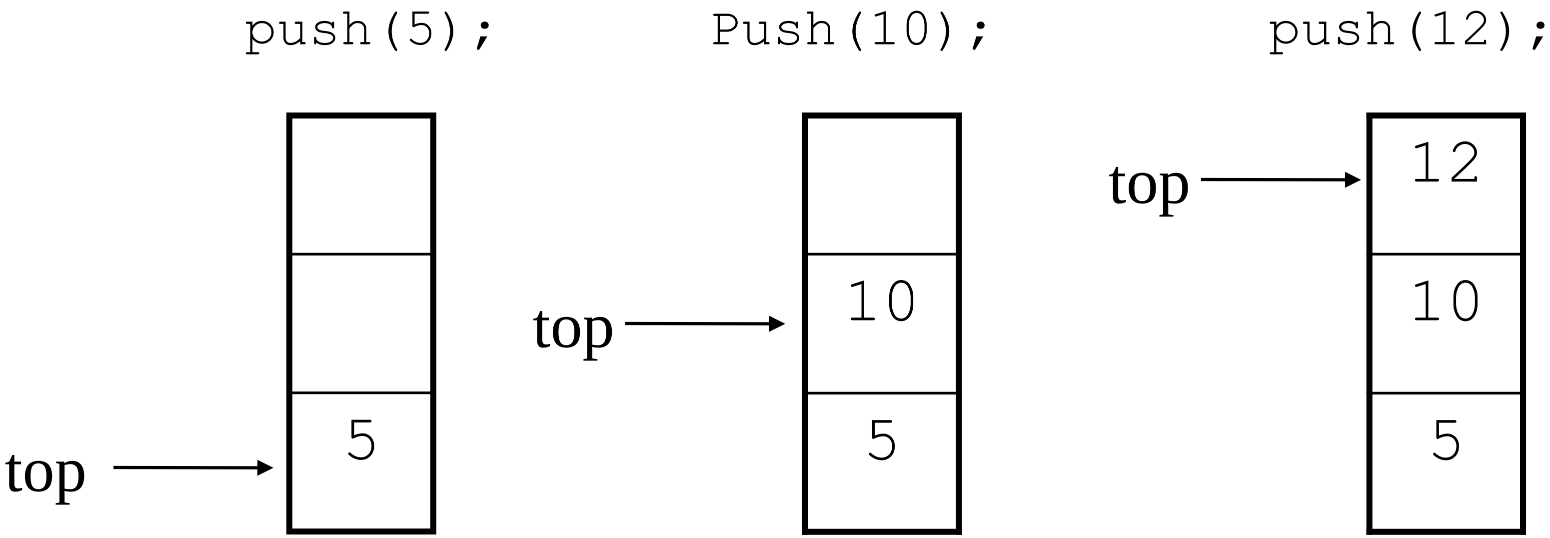
The Push Operation

- Suppose we have an empty integer stack that is capable of holding a maximum of three values. With that stack we execute the following push operations.

```
push(5);
push(10);
push(12);
```

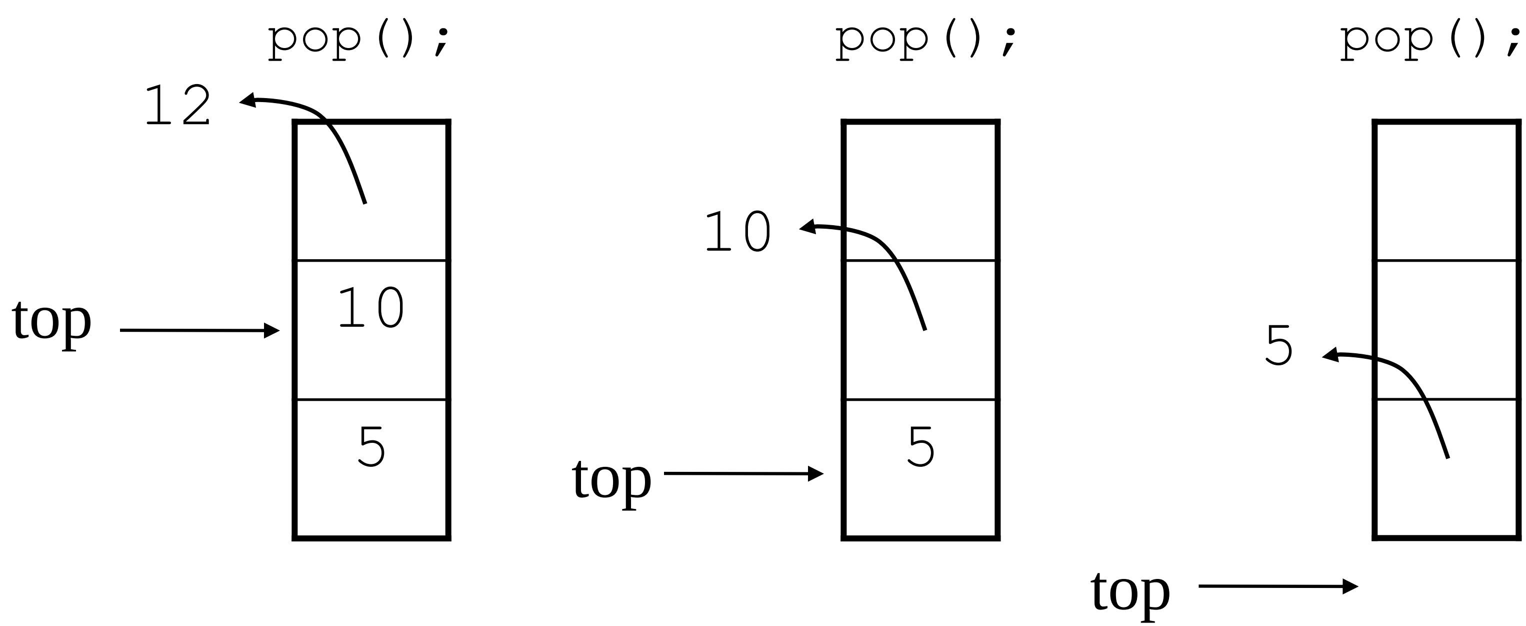
The Push Operation

The state of the stack after each of the `push` operations:



The Pop Operation

- Now, suppose we execute three consecutive pop operations on the same stack:



Other Useful Stack Functions

- `isFull`: A Boolean function needed for static stacks. Returns true if the stack is full. Otherwise, returns false.
- `isEmpty`: A Boolean operation needed for all stacks. Returns true if the stack is empty. Otherwise, returns false.

A Static Integer Stack Implementation

- Using classes

Member Variable	Description
`stackArray`	A pointer to `int`. When the constructor is executed, it uses `stackArray` to dynamically allocate an array for storage.
`stackSize`	An integer that holds the size of the stack.
`top`	An integer that is used to mark the top of the stack.

The `IntStack` Class

- Member Functions

Member Function	Description
constructor	The class constructor accepts an integer argument, which specifies the size of the stack. An integer array of this size is dynamically allocated, and assigned to `stackArray`. Also, the variable `top` is initialized to `-1`.
`push`	The `push` function accepts an integer argument, which is pushed onto the top of the stack.
`pop`	The `pop` function uses an integer reference parameter. The value at the top of the stack is removed, and copied into the reference parameter.

The `IntStack` Class

- Member Functions (continued)

Member Function	Description
`isFull`	Returns `true` if the stack is full and `false` otherwise. The stack is full when `top` is equal to `stackSize - 1`.
`isEmpty`	Returns `true` if the stack is empty, and `false` otherwise. The stack is empty when `top` is set to `-1`.

Contents of `IntStack.h`

```
#ifndef INTSTACK_H
#define INTSTACK_H

class IntStack
{
private:
       int *stackArray;
       int stackSize;
       int top;

public:
       IntStack(int);
       void push(int);
       void pop(int &);
       bool isFull(void);
       bool isEmpty(void);
};

#endif
```

Contents of `IntStack.cpp`

```
//*******************
// Constructor      *
//*******************

IntStack::IntStack(int size)
{
      stackArray = new int[size];
      stackSize = size;
      top = -1;
}
```

Contents of `IntStack.cpp`

```
//**************************************************
// Member function push pushes the argument onto  *
// the stack.                                      *
//**************************************************

void IntStack::push(int num)
{
	if (isFull())
	{
		cout << "The stack is full.\n";
	}
	else
	{
		top++;
		stackArray[top] = num;
	}
}
```

Contents of `IntStack.cpp`

```
//*****************************************************
// Member function pop pops the value at the top     *
// of the stack off, and copies it into the variable *
// passed as an argument.                            *
//*****************************************************

void IntStack::pop(int &num)
{
	if (isEmpty())
	{
		cout << "The stack is empty.\n";
	}
	else
	{
		num = stackArray[top];
		top--;
	}
}
```

Contents of `IntStack.cpp`

```
//*******************************************************
// Member function isFull returns true if the stack *
// is full, or false otherwise.                      *
//*******************************************************

bool IntStack::isFull(void)
{
	bool status;

	if (top == stackSize - 1)
		status = true;
	else
		status = false;

	return status;
}
```

Contents of `IntStack.cpp`

```
//*******************************************************
// Member funciton isEmpty returns true if the stack *
// is empty, or false otherwise.                     *
//*******************************************************

bool IntStack::isEmpty(void)
{
	bool status;

	if (top == -1)
		status = true;
	else
		status = false;

	return status;
}
```

Demo Program (`IntStackDemo.cpp`)

```
#include "intstack.h"
#include <iostream>
using namespace std;

int main()
{
    IntStack  stack(5);
    int num;
        cout << "Created an empty stack with capacity 5, trying to pop. \n";
        stack.pop(num);

    int values[] = {2, 7, 10, 5, 3, 8, 11};
    cout << "\nPushing...\n";
    for (int k = 0; k < 7; k++)
    {
        cout << values[k] << "   ";
        stack.push(values[k]);
         cout << endl;
    }
    cout << "\nPopping...\n";
    while (!stack.isEmpty())
    {
        stack.pop(num);
        cout << num << endl;
    }
    cout << endl;
    return 0;
}
```

```
Created an empty stack with
capacity 5, trying to pop.
The stack is empty.

Pushing...
2
7
10
5
3
8  The stack is full.

11  The stack is full.


Popping...
3
5
10
7
2
```

Dynamic Stacks

- A dynamic stack is built on a linked list instead of an array.
- A linked list-based stack offers two advantages over an array-based stack.
  - No need to specify the size of the stack.
    - A dynamic stack simply starts as an empty linked list, and then expands by one node each time a value is pushed.
      - When a value is popped, its memory is freed.
  - A dynamic stack will never be full, as long as the system has enough free memory.
- Next couple of slides give an implementation of a dynamic integer stack using classes (see `DynIntStack.h` and `DynIntStack.cpp`)

Contents of DynIntStack.h

```
struct StackNode
{
	int value;
	StackNode *next;
};

class DynIntStack
{
private:
	StackNode *top;

public:
	DynIntStack(void);
	void push(int);
	void pop(int &);
	bool isEmpty(void);
};
```

Contents of `DynIntStack.cpp`

```
//**********************************************
// Constructor to generate an empty stack.     *
//**********************************************

DynIntStack::DynIntStack()
{
       top = NULL;
}
```

Contents of `DynIntStack.cpp`

```
//**********************************************
// Member function push pushes the argument onto *
// the stack.                                    *
//**********************************************
void DynIntStack::push(int num)
{
        StackNode *newNode;
        // Allocate a new node & store Num
        newNode = new StackNode;
        newNode->value = num;

        // If there are no nodes in the list
        // make newNode the first node
        if (isEmpty())
        {
                top = newNode;
                newNode->next = NULL;
        }
        else    // Otherwise, insert NewNode before top
        {
                newNode->next = top;
                top = newNode;
        }
}
```

Contents of `DynIntStack.cpp`

```
/*******************************************************
// Member function pop pops the value at the top     *
// of the stack off, and copies it into the variable *
// passed as an argument.                            *
//*******************************************************
void DynIntStack::pop(int &num)
{
	StackNode *temp;

	if (isEmpty())
	{
		cout << "The stack is empty.\n";
	}
	else	// pop value off top of stack
	{
		num = top->value;
		temp = top->next;
		delete top;
		top = temp;
	}
}
```

Contents of `DynIntStack.cpp`

```
//*****************************************************
// Member funciton isEmpty returns true if the stack *
// is empty, or false otherwise.                     *
//*****************************************************

bool DynIntStack::isEmpty(void)
{
	bool status;

	if (top == NULL)
		status = true;
	else
		status = false;

	return status;
}
```

Demo Program (`DynIntStackDemo.cpp`)

```cpp
// This program demonstrates the dynamic stack class DynIntStack.
#include <iostream>
#include "DynIntStack.h"
using namespace std;

int main()
{
    DynIntStack stack;
    int catchVar;
    // Push values 5, 10, and 15 on the stack
    for (int value = 5; value <=15; value = value + 5)
    {
       cout << "Push: " << value << "\n";
       stack.push(value);
    }
    cout << "\n";

    //Pop three values, then attempt a fourth pop
    for (int k = 1; k <= 3; k++)
    {
       cout << "Pop: ";
       stack.pop(catchVar);
       cout << catchVar << endl;
    }

    cout << "\nAttempting to pop again... ";
    stack.pop(catchVar);
    return 0;
}
```

```
Push: 5
Push: 10
Push: 15

Pop: 15
Pop: 10
Pop: 5

Attempting to pop again...
The stack is empty
```

A Case Study: Postfix Expression Evaluator

- Infix expressions
  - Operator is between the operands
  - e.g. 4 + 7
- Postfix expressions
  - Operator is after the operands
  - e.g. 4 7 +
  - In more complex expressions
    - When an operator is seen, it is applied to the previous two operands and the result is replaced.
  - **<u>No</u>** need to use parantheses; **<u>no</u>** operator precedence to imply the order of evaluation in complex expressions (examples in the next slide).

Postfix Expressions

Postfix Expression	Infix Expression	Value
4 7 *	4 * 7	28
4 7 2 + *	4 * (7 +2)	36
4 7 * 9 -	4 * 7 - 9	19
3 4 7 * 2 / +	3+ 4 * 7 / 2	17

Postfix Expression Evaluator – Algorithm

- Scan the expression from left to right
  - If next token is an operand
    - Push it to the stack
  - If next token is an operator
    - Pop from the stack – this is the rhs operand
    - Pop from the stack – this is the lhs operand
    - Apply operator on these operands
    - Push the result to the stack
- When expression is finished
  - Pop from the stack, this is the result of expression
- Let's trace this algorithm on 4 7 2 + * 9 -

Postfix Expression Evaluator – Implementation

- We use integer stack to store the operands and intermediate results
- For the sake on simplicity in expression parsing
  - We assume that expression is a valid postfix expression
  - We assume single digit operands in the expression so that we can read char by char
    - of course, intermediate results may be larger
- See next slide for the code (PostfixEvaluator.cpp)

```
DynIntStack stack;
char token;
int rhs, lhs, result;

cout << "Please enter postfix expression:  ";
while (cin >> token)  //as long as there is input
{
   if (token >= '0' && token <= '9')   //if digit
   {
      stack.push(token - '0');  //push it to stack
   }
   else     //if operator
   {
      stack.pop(rhs);
      stack.pop(lhs);  // pop two operands
      //and apply the operations. Result is pushed to the stack
      if (token == '+')
         stack.push(lhs + rhs);
      else if (token == '-')
         stack.push(lhs - rhs);
      else if (token == '*')
         stack.push(lhs * rhs);
      else if (token == '/')
         stack.push(lhs / rhs);
   }
}
//after while, the stack contains only the final result, pop it and display
stack.pop(result);
cout << "result is: " << result << endl;
```

Introduction to the Queue

- Like a stack, a queue is a data structure that holds a sequence of elements.
- A queue, however, provides access to its elements in *first-in, first-out (FIFO)* order.
- The elements in a queue are processed like customers standing in a grocery check-out line: the first customer in line is the first one served.

Example Applications of Queues

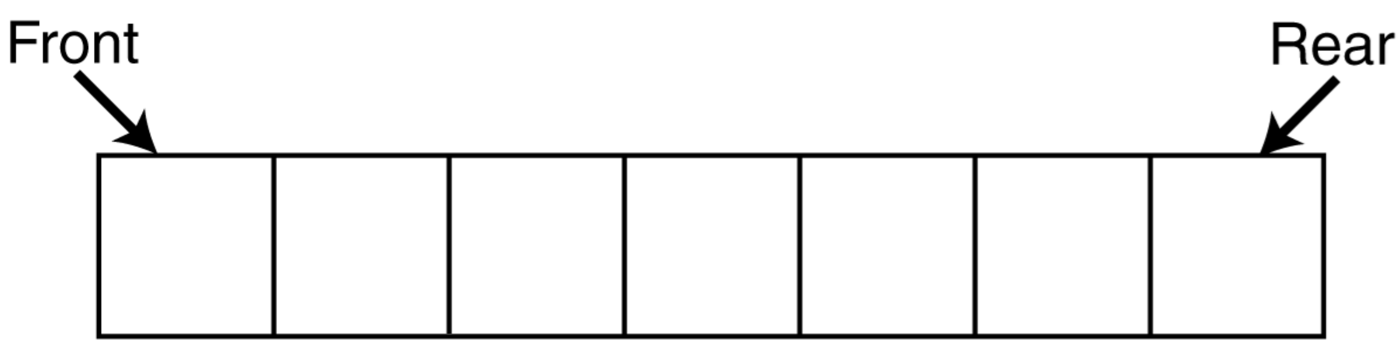
- In a multi-user system, a queue is used to hold print jobs submitted by users, while the printer services those jobs one at a time.
- Communications software also uses queues to hold information received over networks. Sometimes information is transmitted to a system faster than it can be processed, so it is placed in a queue when it is received.

Static and Dynamic Queues

- Just as stacks, queues are implemented as arrays or linked lists.
- Dynamic queues offer the same advantages over static queues that dynamic stacks offer over static stacks.

Queue Operations

- Think of queues as having a front and a rear.
  - rear: position where elements are added
  - front: position from which elements are removed



Queue Operations

- The two primary queue operations are *enqueuing* and *dequeuing*.
- To *enqueue* means to insert an element at the rear of a queue.
- To *dequeue* means to remove an element from the front of a queue.

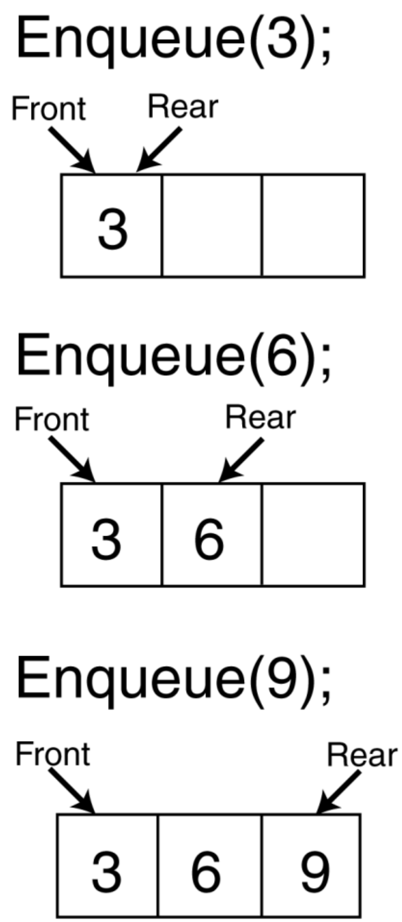
Queue Operations

- Suppose we have an empty static integer queue that is capable of holding a maximum of three values. With that queue we execute the following enqueue operations.

```
Enqueue(3);
Enqueue(6);
Enqueue(9);
```

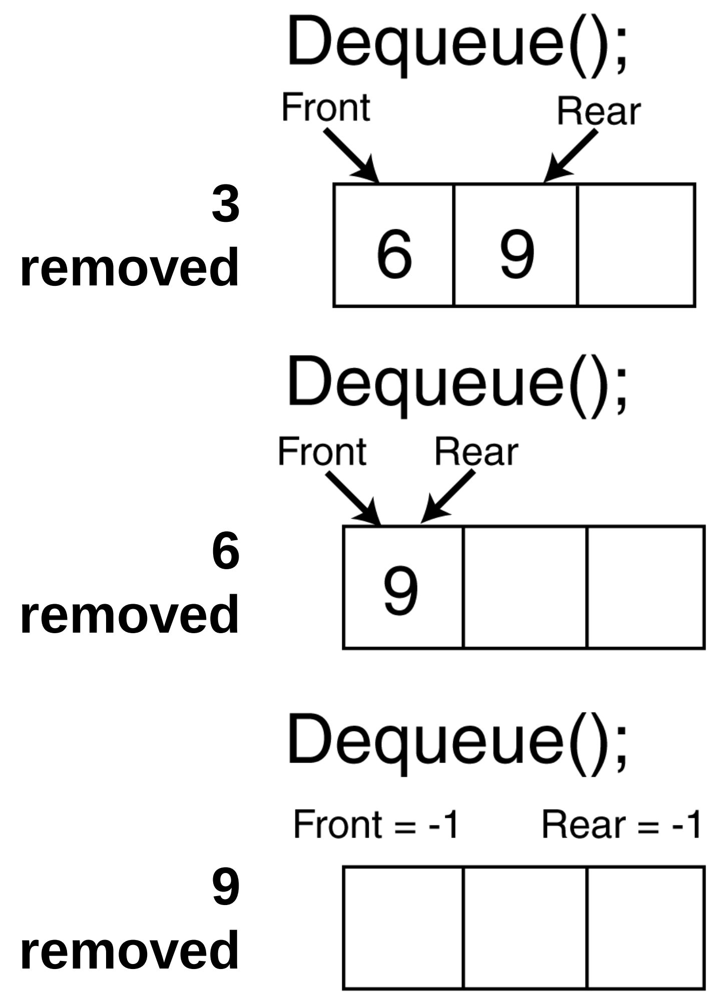
Queue Operations - Enqueue

- The state of the queue after each of the enqueue operations.



Queue Operations - Dequeue

- Now let's see how dequeue operations are performed. The figure on the right shows the state of the queue after each of three consecutive dequeue operations
- An important remark
  - After each dequeue, remaining items shift toward the front of the queue.



Efficiency Problem of Dequeue & Solution

- Shifting after each dequeue operation causes ineffiency.
- Solution
  - Let front index move as elements are removed
  - let rear index "wrap around" to the beginning of array, treating array as circular
    - Similarly, the front index as well
  - Yields more complex enqueue, dequeue code, but more efficient

```
Enqueue(3);
Enqueue(6);
Enqueue(9);
Dequeue();
Dequeue();
Enqueue(12);
Dequeue();
```

Implementation of a Static Queue

- The previous discussion was about static arrays
  - Container is an array
- Class Implementation for a static integer queue
  - Member functions
    - `enqueue`
    - `dequeue`
    - `isEmpty`
    - `isFull`
    - `clear`

Contents of `IntQueue.h`

```
#ifndef INTQUEUE_H
#define INTQUEUE_H

class IntQueue
{
private:
   int *queueArray;
   int queueSize;  //capacity of queue
   int front;
   int rear;
   int numItems;  //# of elements currently in the queue
public:
   IntQueue(int);  //constructor, parameter is capacity
   void enqueue(int);
   void dequeue(int &);
   bool isEmpty() const;
   bool isFull() const;
   void clear();  //removes all elements
};
#endif
```

Contents of `IntQueue.cpp`

```
#include <iostream>
#include "IntQueue.h"
using namespace std;

//*****************************************
// Constructor - creates an empty queue   *
// with given number of elements          *
//*****************************************
IntQueue::IntQueue(int s)
{
   queueArray = new int[s];
   queueSize = s;
   front = -1;
   rear = -1;
   numItems = 0;
   #ifdef _DEBUG
      cout << "A queue with " << s << " elements created\n";
   #endif
}
```

Contents of IntQueue.cpp

```
//********************************************
// Function enqueue inserts the value in num *
// at the rear of the queue.                 *
//********************************************
void IntQueue::enqueue(int num)
{
   if (isFull())
    {
      cout << "The queue is full. " << num << " not enqueued\n";
    }
   else
   {
      // Calculate the new rear position circularly.
      rear = (rear + 1) % queueSize;
      // Insert new item.
      queueArray[rear] = num;
      // Update item count.
      numItems++;
      #ifdef _DEBUG
         cout << num << " enqueued\n";
      #endif
   }
}
```

Contents of `IntQueue.cpp`

```
//*********************************************
// Function dequeue removes the value at the   *
// front of the queue, and copies it into num. *
//*********************************************
void IntQueue::dequeue(int &num)
{
   if (isEmpty())
   {
      cout << "Attempting to dequeue on empty queue, exiting program...\n";
      exit(1);
   }
   else
   {
      // Move front.
      front = (front + 1) % queueSize;
      // Retrieve the front item.
      num = queueArray[front];
      // Update item count.
      numItems--;
   }
}
```

Contents of `IntQueue.cpp`

```
//*********************************************
// Function isEmpty returns true if the queue *
// is empty, and false otherwise.             *
//*********************************************
bool IntQueue::isEmpty() const
{
   if (numItems > 0)
      return false;
   else
      return true;
}
```

Open Question: How would we understand that queue is empty without using `numItems`?

Contents of `IntQueue.cpp`

```
//***************************************
// Function isFull returns true if the queue *
// is full, and false otherwise.             *
//***************************************
bool IntQueue::isFull() const
{
   if (numItems < queueSize)
      return false;
   else
      return true;
}
```

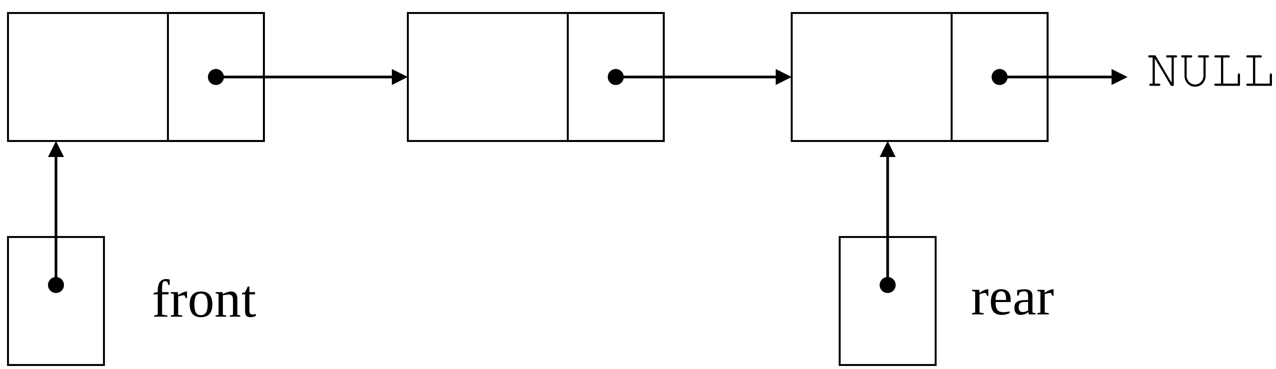
Open Question: How would we understand that queue is full without using `numItems`?

Contents of `IntQueue.cpp`

```
//****************************************
// Function clear resets the front and rear *
// indices, and sets numItems to 0.         *
//****************************************
void IntQueue::clear()
{
   front =  - 1;
   rear =  - 1;
   numItems = 0;
   #ifdef _DEBUG
      cout << "queue cleared\n";
   #endif
}
```

Dynamic Queues

- Like a stack, a queue can be implemented using a linked list
- Allows dynamic sizing, avoids issue of shifting elements or wrapping indices



Dynamic Queues

- A dynamic queue starts as an empty linked list.
- With the first enqueue operation, a node is added, which is pointed to by `front` and `rear` pointers.
- As each new item is added to the queue, a new node is added to the rear of the list, and the `rear` pointer is updated to point to the new node.
- As each item is dequeued, the node pointed to by the `front` pointer is deleted, and `front` is made to point to the next node in the list.

Contents of `DynIntQueue.h`

```
struct QueueNode
{
     int value;
     QueueNode *next;
     QueueNode(int num, QueueNode *ptr = NULL)
     {
          value = num;
          next = ptr;
     }
};

class DynIntQueue
{
private:
     // These track the front and rear of the queue.
     QueueNode *front;
     QueueNode *rear;
public:
     DynIntQueue(); // Constructor.
     // Member functions.
     void enqueue(int);
     void dequeue(int &);
     bool isEmpty() const;
     void clear();
};
```

Contents of `DynIntQueue.cpp`

```
#include <iostream>
#include "DynIntQueue.h"
using namespace std;

//************************************************
// Constructor. Generates an empty queue         *
//************************************************
DynIntQueue::DynIntQueue()
{
    front = NULL;
    rear = NULL;
    #ifdef _DEBUG
        cout << "An emmpty queue has been created\n";
    #endif
}
```

Contents of DynIntQueue.cpp

```
//****************************************
// Function enqueue inserts the value in num *
// at the rear of the queue.                 *
//****************************************
void DynIntQueue::enqueue(int num)
{
    if (isEmpty())   //if the queue is empty
    {    //make it the first element
         front = new QueueNode(num);
         rear = front;
    }
    else  //if the queue is not empty
    {    //add it after rear
         rear->next = new QueueNode(num);
         rear = rear->next;
    }
    #ifdef _DEBUG
         cout << num << " enqueued\n";
    #endif
}
```

Contents of `DynIntQueue.cpp`

```
//*******************************************************
// Function dequeue removes the value at the   *
// front of the queue, and copies it into num. *
//*******************************************************
void DynIntQueue::dequeue(int &num)
{
   QueueNode *temp;
   if (isEmpty())
   {
      cout << "Attempting to dequeue on empty queue, exiting program...\n";
      exit(1);
   }
   else //if the queue is not empty
   {  //return front's value, advance front and delete old front
      num = front->value;
      temp = front;
      front = front->next;
      delete temp;
   }
}
```

Contents of DynIntQueue.cpp

```
//********************************************
// Function isEmpty returns true if the queue *
// is empty, and false otherwise.             *
//********************************************
bool DynIntQueue::isEmpty() const
{
    if (front == NULL)
        return true;
    else
        return false;
}
```

Contents of `DynIntQueue.cpp`

```
//****************************************
// Function clear dequeues all the elements  *
// in the queue.                             *
//****************************************
void DynIntQueue::clear()
{
    int value;   // Dummy variable for dequeue

    while(!isEmpty())
        dequeue(value); //delete all elements
    #ifdef _DEBUG
        cout << "queue cleared\n";
    #endif
}
```