Abstract Data Type

- A data type for which:
  - only the properties of the data and the operations to be performed on the data are specific,
  - not concerned with how the data will be represented or how the operations will be implemented.
- In fact, an ADT may be implemented by various specific data types or data structures, in many ways and in many programming languages.
- Examples:
  - ItemInventory (impl’d using array AND linked list)
  - Stacks and Queues

Introduction to the Stack

- **Stack**: an abstract data type 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 a 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
Stack Application: Postfix notation

- Postfix notation is another way of writing arithmetic expressions.
- We normally use infix, the operator is between the operands.
- In postfix notation, the operator is written after the two operands.
  \[ \text{infix: } 2+5 \quad \text{postfix: } 2 \ 5 + \]

- Expressions are evaluated from left to right.
- Precedence rules and parentheses are never needed!!

Postfix notation: using a stack

- evaluation from left to right: push operands
- for operator: pop two values, perform operation, and push the result

Updating the input line is unnecessary
Evaluate Postfix Expression algorithm

• Using a stack:
  WHILE more input items exist
    get next item
    IF item is a number
      stack.push(item)
    ELSE (item is an operator)
      num2 = stack.pop()
      num1 = stack.pop()
      Compute result
      stack.push(result)
  end WHILE
result = stack.pop()

Implementing a Stack Class

• IntStack:
  - contains ints
  - implemented using a dynamically allocated array, but once allocated, the array does not change size

• Alternative implementations of an integer stack:
  - use a regular array of ints
  - use a linked list with nodes that contain ints (see 18.2)
  - std::stack from the C++ library (STL) (see 18.3)

IntStack: A stack class

class IntStack
{
private:
  int *stackArray;  // Pointer to the stack array
  int stackSize;    // The stack size (will not change)
  int top;          // Index to the top of the stack

public:
  // Constructor
  IntStack(int size);
  // Destructor
  ~IntStack();

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

IntStack: functions

//****************************************************
// Constructor
// This constructor creates an empty stack. The size parameter is the size of the stack.
//****************************************************
IntStack::IntStack(int size)
{
  stackArray = new int[size];  // dynamic alloc
  stackSize = size;            // save for reference
  top = -1;                    // empty
}

//****************************************************
// Destructor
//****************************************************
IntStack::~IntStack()
{
  delete [] stackArray;
}
IntStack: push

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

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

IntStack: pop

// Member function pop pops the value at the top of the stack off, and returns it as the result.
// If the stack is empty, outputs a message and returns -1.

void IntStack::pop(int &num) {
    int num = -1;
    if (isEmpty())
        cout << "The stack is empty.\n"; // This is called stack underflow
    else
        num = stackArray[top];
        top--;
    return num;
}

IntStack: test functions

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

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

// Member function isEmpty returns true if the stack is empty, or false otherwise.

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

IntStack: driver

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

int main() {
    // set up the stack
    IntStack stack(50);
    stack.push(2);
    stack.push(3);
    stack.push(5);
    int x;
    x = stack.pop();
    x = stack.pop();
    stack.push(10);
    cout << x << endl;
    return 0;
}
Introduction to the Queue

- **Queue**: an abstract data type 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
  - print jobs sent to a (shared) printer

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 Applications

- The best example applications of queues involve multiple processes.
- For example, imagine the print queue for a computer lab.
- Any computer can add a new print job to the queue (enqueue).
- The printer performs the dequeue operation and starts printing that job.
- While it is printing, more jobs are added to the Q.
- When the printer finishes, it pulls the next job from the Q, continuing until the Q is empty

```cpp
int item;
g.enqueue(2);
g.enqueue(3);
g.enqueue(5);
item = g.dequeue(); //item is 2
item = g.dequeue(); //item is 3
item = g.dequeue(); //item is 3
```
Implementing a Queue Class

- **IntQueue:**
  - contains ints
  - implemented using a dynamically allocated array, but once allocated, the queue does not change size

- Alternative implementations of an integer queue:
  - use a regular array of ints
  - use a linked list with nodes that contain ints (see 18.5)
  - `std::deque` and `std::queue` from the C++ library (STL) (see 18.6)

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

<table>
<thead>
<tr>
<th>3</th>
<th>4</th>
<th>7</th>
<th>9</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>rear: q.enqueuer(4)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>front</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Implementing a Queue Class

- To “wrap” the rear index back to the front of the array, you can use this code to increment the rear during enqueue:

  ```cpp
  if (rear = queueSize-1)
    rear = 0;
  else
    rear = rear+1;
  ``

  The following code is equivalent, but shorter (assuming 0 <= rear < queueSize):

  ```cpp
  rear = (rear + 1) % queueSize;
  ```

  Do the same for advancing the front index.
Implementing a Queue Class

• When is it full?

<table>
<thead>
<tr>
<th>3</th>
<th>4</th>
<th>5</th>
<th>2</th>
<th>1</th>
<th>7</th>
<th>9</th>
<th>6</th>
</tr>
</thead>
</table>

(rear+1)%queueSize==front

- Note: enqueue increments rear
- q.enqueue(5);
- q.enqueue(2);
- q.enqueue(1);

• It’s full:

Implementing a Queue Class

• When is it empty?

Note: dequeue increments front

<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>

(rear+1)%queueSize==front

- Note: dequeue increments front
- int x;
- for (int i=0; i<queueSize;i++)
- x = q.dequeue();
- after the first one:

one element left:

<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>

no elements left, front passes rear:

<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>

• It’s empty:

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

Implementing a Queue Class

• In the implementation that follows:
- the queue is a dynamically allocated array, whose size does not change once initialized.
- If the queue is not empty:
  - rear is the index of the last item that was enqueued.
  - front is the index of the next item to be dequeued.
- numItems: how many items are in the queue
- initial values: rear = -1, front = 0, numItems=0;
- queueSize: the size of the array
IntQueue: a queue class

class IntQueue
{
  private:
  int *queueArray;  // Points to the queue array
  int queueSize;    // The queue size
  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(int);
  // Destructor
  ~IntQueue();
  // Queue operations
  void enqueue(int);
  int dequeue();
  bool isEmpty();
  bool isFull();
};

IntQueue: functions

//**************************************************
// Creates an empty queue of a specified size. *
//**************************************************
IntQueue::IntQueue(int s)
{
  queueArray = new int[s];
  queueSize = s;
  front = 0;
  rear = -1;
  numItems = 0;
}

//**************************************************
// Destructor                                          *
//**************************************************
IntQueue::~IntQueue()
{
  delete [] queueArray;
}

IntQueue: enqueue

//**************************************************
// Enqueue inserts a value at the rear of the queue. *
//**************************************************
void IntQueue::enqueue(int num)
{
  if (isFull())
    cout << "The queue is full.\n";
  else
  { // Calculate the new rear position
    rear = (rear + 1) % queueSize;
    // Insert new item
    queueArray[rear] = num;
    // Update item count
    numItems++;
  }
}

IntQueue: dequeue

//**************************************************
// Dequeue removes the value at the front of the   *
// queue and copies it into num.  *
// If the queue is empty, outputs a message and     *
// returns -1                                        *
//**************************************************
int IntQueue::dequeue(int &num)
{
  int num = -1;
  if (isEmpty())
  { // Retrieve the front item
    num = queueArray[front];
    // Move front
    front = (front + 1) % queueSize;
    // Update item count
    numItems--;
  }
  else
  { // Calculate the new front position
    num = queueArray[front];
    // Move front
    front = (front + 1) % queueSize;
    // Update item count
    numItems++;
  }
  return num;
}
IntQueue: test functions

//****************************************************
// isEmpty returns true if the queue is empty, otherwise false.
//****************************************************
bool IntQueue::isEmpty()
{
    return (numItems == 0);
}

//****************************************************
// isFull returns true if the queue is full, otherwise false.
//****************************************************
bool IntQueue::isFull()
{
    return (numItems == queueSize);
}

IntQueue: driver

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

int main()
{
    // set up the queue
    IntQueue q(50);
    int item;
    q.enqueue(2);
    q.enqueue(3);
    q.enqueue(5);
    item = q.dequeue();
    item = q.dequeue();
    q.enqueue(10);
    cout << item << endl;
}

What is output?
What is left on the queue when the driver is done?