

## Collections of Data

- o Stacks
  - o Like a real stack (of dishes, books, etc.)
  - o Is a LIFO (last-in first-out) collection
  - o Can only be accessed via the top
- o Queues
  - o Like a real queue (grocery line)
  - o A FIFO (first-in first-out) collection
- o Implementations:
  - o Arrays
  - o Linked Lists

## Collections?

- o Both C# and Java have Collections classes
  - o System.Collections in C#
  - o Stack class

## A Simple Stack Application

Stacks can be used to evaluate expressions.

We will do something closely related that's simpler:  
We will look at an expression and check if its brackets match correctly (are balanced).

Our expression will contain two different kinds of brackets: ( ) and { }. Any symbol other than these brackets is ignored.

For example,

1.  $\{(x + (y)) * (z)\}$  and  $\{(x + ((y * z)))\}$  are balanced, and
2.  $\{(x + (y)) * (z)\}$  and  $\{(x + \{y\} * z)\}$  are not balanced.

## The Algorithm

1. Go through the expression from left to right.
2. If you see a left bracket, push it on the stack.
3. If you see a right bracket, and there is a matching left bracket on top of the stack, then pop that left bracket off the stack.
4. If you see a right bracket, and there is not a matching left bracket on top of the stack, then the expression is unbalanced.
5. If you see a non-bracket, ignore it.

If all goes well, and the stack is empty after you have processed the whole expression, the expression is balanced; if there are still symbols on the stack, the expression is unbalanced.

## Use this code as a starting point

```
public static bool isBalanced(string expression) {  
    return false;  
}  
  
public static void Main(string[] args) {  
    string ourParens = args[0];  
    Console.WriteLine(isBalanced(ourParens));  
}
```

## A Stack of Characters

- o We can use an instance of the Stack class to implement the algorithm from the previous page.
- o We can write our own instance of a Stack class.

## Writing the CharStack Class

Let's use an array of char's for storage of the data in the CharStack class.

The class should have the following functions:

- a constructor with no arguments
- public void push(char item) {
- public char pop() {
- public char peek() {
- public bool empty() {

Print an error message when something really bad happens (like pop on an empty stack) or make your own exception. What about push when the underlying array is completely filled? For now, you can just give an error message or throw an exception.

## A Simple Stack Application

o Reversing things:

- o Finding palindromes or words/sentences that read the same forward or backward.

- o Example: A man, a plan, a canal, Panama!

## The Algorithm

1. Go through a string from left to right.
2. If you see an alphabetic character, push it on the stack
3. Go through the string again. If you see an alphabetic character, pop a character off the stack and see if they match.
  - 3a. If they match, continue the search with the next character.
  - 3b. If they don't match stop, because the word isn't a palindrome.

If all goes well, and the stack is empty after you have processed the whole string, then the string is a palindrome. If there are still symbols on the stack, the string is not a palindrome.

## Unbounded Version of Push

```
public void push(char item) {  
  
    // if the stack is full, allocate more space  
    if (numberOfItems >= storage.Length) {  
        char temp[] = new char[2 * storage.Length + 1];  
        for (int i = 0; i < storage.Length; i++) {  
            temp[i] = storage[i];  
        }  
        storage = temp;  
    }  
    storage[numberOfItems] = item;  
    numberOfItems++;  
}
```

## What's wrong with the following code?

```
Stack s = new Stack();  
string name;  
s.push("Armondo");  
name = s.pop();
```

What is newer in C# that avoids this issue? Generics  
Read: <http://www.ondotnet.com/pub/a/dotnet/2004/05/17/liberty.html>

## Characteristics of Queues

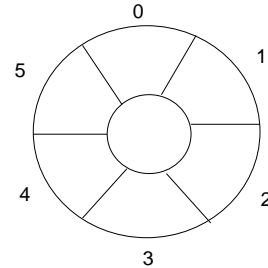
- o Data can only be placed at the back of the queue
- o Data can only be removed from the front of the queue
- o Data can only be removed from the rear of the queue if there is only one item on the queue
- o Data can not be removed from the middle of the queue without first removing all items in front of it.

## Three Main Varieties of Queues

- o Bounded queue
  - o An initial capacity is given and the queue can not grow beyond that capacity
- o Unbounded queue
  - o The queue can continue to grow until you run out of physical resources (such as computer memory)
- o Circular queue
  - o A more efficient way of implementing a bounded queue
  - o enqueue() is used to put elements into the queue
  - o dequeue() is used to take elements out of the queue

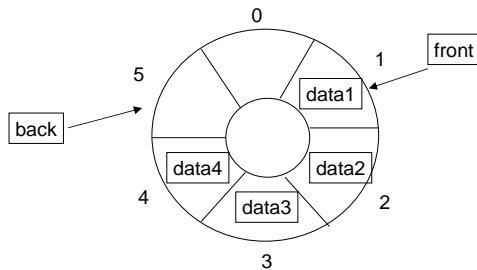
## Circular Queues

- o View an array as a circular structure



## Keeping Track

- o Circular Queues keep track of their elements with a front and a back index



## Not All Data Structures...

- o Use arrays!

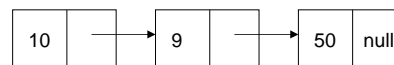
## A Linked List

- o A sequence of elements arranged one after another, with each element connected to the next element by a "link"



## The Head and the Tail

- o A node is a data item containing a connection to the next item (or null if there is no next item)
- o The list's first node is the head
- o The last node is the tail



## Node class

---

```
public class Node {  
    private object data;  
    private Node next;  
  
    ... methods here ...  
}
```

## Nodes

---

- o Single Linked Lists
  - o Nodes contain a link to the next node only
- o Doubly Linked Lists
  - o Nodes contain a link to the next node and the previous node

## Doofie the Programmer

---

- o Why you want to make your data structures private:
  - o <http://www.cs.ucsd.edu/classes/su99/cse8a/lec61.html>

## LinkedList

---

- o [discussed in class]
- o adding to a list
- o removing from a list