Contents

The Stack collections **1**
Introduction 1
Abstract Data Type 1
Difference with array 1
Difference with lists 2
Possible Implementation 2
Using Cells 2
Using Arrays 4

The Stack collections

Introduction

Abstract Data Type

Described abstractly, a stack is

- a finite collection of elements,
- in a particular order,
- that may contain the same element multiple times.

The fact that it may contain the same element multiple times makes it different from a set, the fact that it is ordered makes it different from a multiset.

Generally, it has operations to…

- … create an empty stack,
- … test for emptiness,
- … obtain the value of the element at "the top" of the stack ("peek"),
- … add an element at "the top" of the stack ("push"),
- … remove an element at "the top" of the stack ("pop").

The fact that only the "top element" can be accessed is the main difference with the list abstract data type. Exactly like a stack of plates, stacks implement a "last in, first out" (LIFO) principle.

Difference with array

Like lists, stacks serve a similar purpose to arrays, but with a few notable differences:

- Stacks do not need to have a number of elements fixed ahead of time,
- Stacks automatically expand when elements are added,

- Stacks automatically shrink when elements are removed.

Difference with lists

However, stacks have difference with lists:

- Only the top element's value can be read (accessed),
- Adding and removing can only be done "on the right side", that is, at the top of the stack.

Possible Implementation

Using Cells

Here is a possible implementation of stacks, using cells:

```
using System;

class CStack<T>
{
  private class Cell
  {
    public T Data { get; set; }
    public Cell Next { get; set; }

    public Cell(T data = default(T), Cell node = null)
    {
      Data = data;
      Next = node;
    }
  }

  private Cell top;

  public CStack()
  {
    top = null;
  }

  public void Clear()
  {
    top = null;
  }

  public bool IsEmpty()
  {
```

```
  return top == null;
}

public void Push(T value)
{
  top = new Cell(value, top);
}

public T Pop()
{
  if (IsEmpty())
    throw new ApplicationException(
      "An empty stack cannot be popped."
    );
  T removedData = top.Data;
  top = top.Next;
  return removedData;
}

public T Peek()
{
  if (IsEmpty())
    throw new ApplicationException(
      "An empty stack cannot be peeked."
    );
  return top.Data;
}

public int Count
{
  get
  {
    int count = 0;
    Cell cCell = top;
    while (cCell != null)
    {
      count++;
      cCell = cCell.Next;
    }
    return count;
  }
}

public override string ToString()
{
  string returned = "";
```

```
    if (!IsEmpty())
    {
      Cell cCell = top;
      while (cCell != null)
      {
        if (returned.Length > 0)
          returned += ":";
        returned += cCell.Data;
        cCell = cCell.Next;
      }
    }
    return returned;
  }
}
```

(Download this code)

Using Arrays

Another implementation, using arrays, requires the stack to have a fixed size:

```
using System;

class CAStack<T>
{
  private T[] mArray;

  // Contains the number of the next empty cell.
  private int top = 0;

  // Our default stack size is 10.
  public CAStack(int sizeP = 10)
  {
    if (sizeP <= 0)
      throw new ApplicationException(
        "Stack size must be strictly greater than 0."
      );
    else
      mArray = new T[sizeP];
  }

  public void Clear()
  {
    top = 0;
  }
```

```
public bool IsEmpty()
{
  return top == 0;
}

public void Push(T value)
{
  if (top < mArray.Length)
  {
    mArray[top] = value;
    top++;
  }
  else
  {
    throw new ApplicationException("Stack is full.");
  }
}

public T Pop()
{
  if (IsEmpty())
    throw new ApplicationException(
      "An empty stack cannot be popped."
    );
  return mArray[--top];
}

public T Peek()
{
  if (IsEmpty())
    throw new ApplicationException(
      "An empty stack cannot be peeked."
    );
  return mArray[top - 1];
}

public int Count
{
  get { return top; }
}

public override string ToString()
{
  string returned = "";
  if (!IsEmpty())
```

```
    {
      int counter = top - 1;
      while (counter >= 0)
      {
        if (returned.Length > 0)
          returned += ":";
        returned += mArray[counter];
        counter--;
      }
    }
    return returned;
  }
}
```

(Download this code)

This technique uses a partially filled array to make sure that non-allocated indices are never accessed.