Абстрактные типы данных. Классы коллекций .NET

Лекции по курсу

Языки программирования

семестр 2

Класс как конкретный и абстрактный тип данных

Тип данных характеризуется:

- Хранение в памяти
- Множество значений
- Набор операций
- Реализация операций

Абстрактный тип данных (АТД) характеризуется:

- ~~Хранение в памяти~~
- Множество значений
- Набор операций
- ~~Реализация операций~~
- Асимптотическая сложность операций

Класс – можно рассматривать:

- и как конкретный тип данных (для того, кто реализует класс)
- и как абстрактный тип данных (АТД) – для тех, кто пользуется объектами класса (конструирует их, вызывает операции, методы)

Модификаторы защиты доступа позволяют скрыть доступ к внутреннему представлению, которое не хочет открывать разработчик класса

Помещение класса в откомпилированную библиотеку .dll позволяет скрыть реализацию (в некотором смысле)

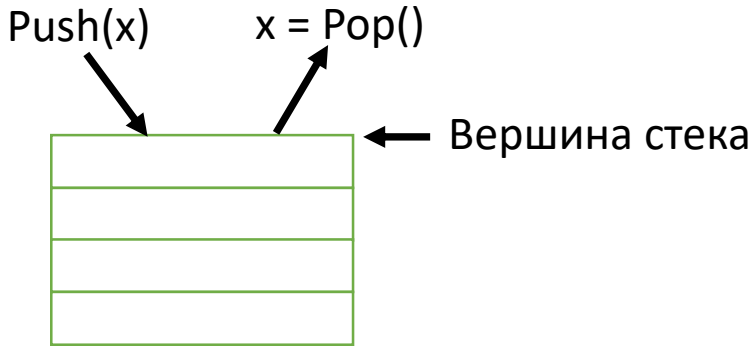
Принцип отделения интерфейса класса от реализации позволяет пользователю абстрагироваться от внутреннего устройства класса

АТД Стек и его интерфейс

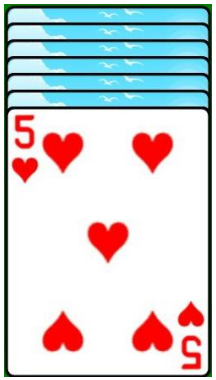
Стек (магазин) – набор данных, устроенный по принципу **LIFO:**
Last In – First Out (последним пришёл – первым ушёл)

Интерфейс класса Стек:

```
class Stack<T>
{
  public int Count { get; }
  public void Push(T item);
  public T Pop();
  public T Peek();
}
```



Примеры стеков:



Выражение в ПОЛИЗ

ПОЛИЗ – польская бесскобочная инверсная запись выражения (1957 г.)

- На стек кладутся только операнды.
- Операция всегда делается над двумя операндами на вершине стека.
- Результат операции снова кладется на стек.
- В конце на стеке должно остаться одно значение. Это – результат.

Выражение в обычной записи

2 + 3 * 4

Выражение в ПОЛИЗ

2 **3** **4** ***** +

3 4 * 2 +

Выражение в обычной записи

(2 + 3) * 4

Выражение в ПОЛИЗ

2 3 + 4 *

Вычисление выражения в ПОЛИЗ с помощью стека: алгоритм и пример работы

Выражение в обычной записи

```
(1 + 2) * 4 + 3
```

Выражение в ПОЛИЗ

```
1 2 + 4 * 3 +
```

Словесное описание алгоритма

Пока не конец ввода
  Считываем лексему
  Если это число, то кладём на стек
  Если это – знак операции, то
    снимаем со стека два последних операнда
    применяем к ним указанную операцию
    кладём результат на стек
Снимаем со стека результат
Assert(стек пуст)

Пример работы

Ввод	Операция	Стек
1	поместить в стек	1
2	поместить в стек	1, 2
+	сложение	3
4	поместить в стек	3, 4
*	умножение	12
3	поместить в стек	12, 3
+	сложение	15

Реализация алгоритма

```
var expr = "1 2 + 4 * 3 +";
var ss = expr.Split();
var st = new Stack<int>();

foreach (var x in ss)
    if (char.IsDigit(x[0]))
        st.Push(int.Parse(x));
    else if (x == "+")
        st.Push(st.Pop() + st.Pop());
    else if (x == "*")
        st.Push(st.Pop() * st.Pop());

WriteLine(st.Pop());

System.Diagnostics.Debug.Assert(st.Count == 0);
// срабатывает только в режиме Debug
```

Риторический вопрос

Мешало ли нам при написании алгоритма вычисления выражения в ПОЛИЗ то, что мы не знали внутреннее устройство стека?

Риторический вопрос

Мешало ли нам при написании алгоритма вычисления выражения в ПОЛИЗ то, что мы не знали внутреннее устройство стека?

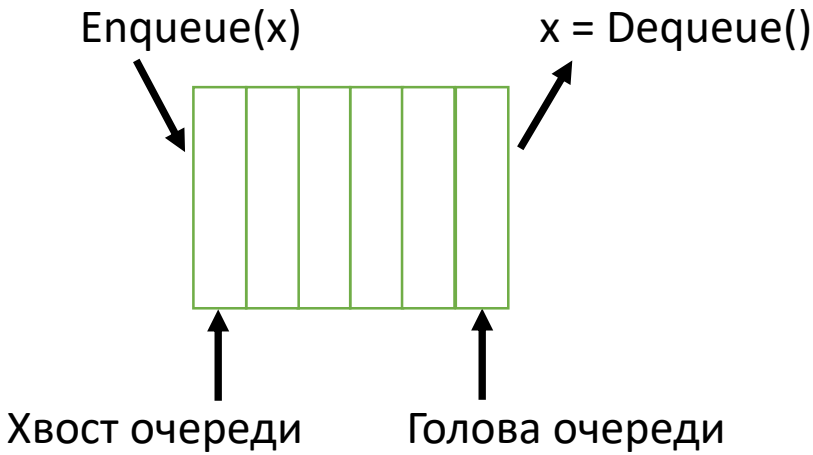
Правильный ответ: **помогало**

АТД Очередь и его интерфейс

Очередь – набор данных, устроенный по принципу **FIFO:**
First In – First Out (первым пришёл – первым ушёл)

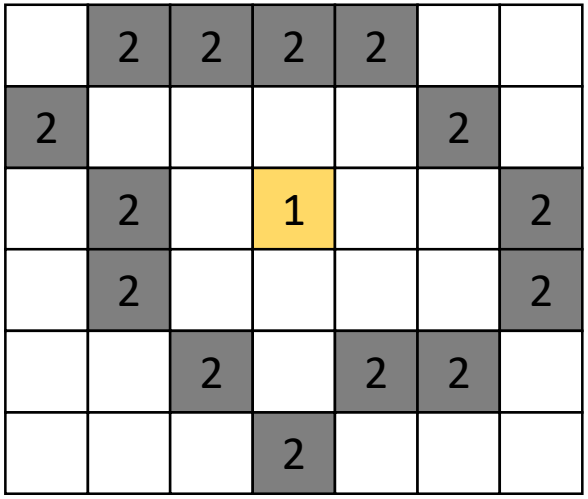
Интерфейс класса Очередь:

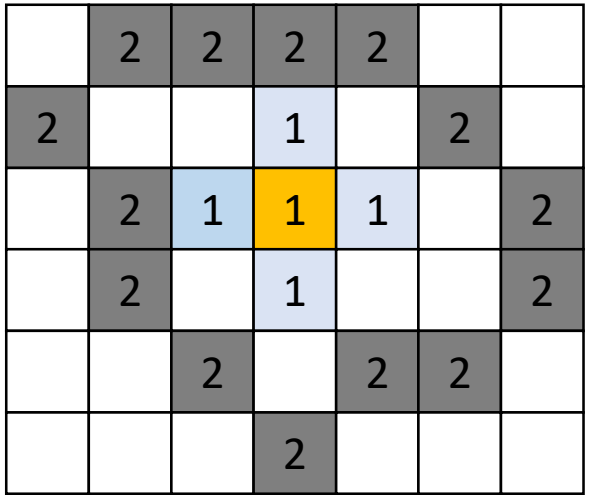
```
class Queue<T>
{
  public int Count { get; }
  public void Enqueue(T item);
  public T Dequeue();
  public T Peek();
}
```

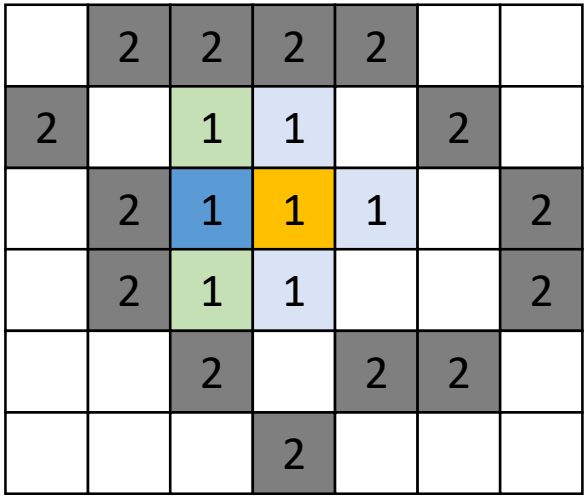


Примеры очередей:

Алгоритм FloodFill заливки области







Словесное описание алгоритма

Красим начальную точку и добавляем ее в конец очереди

Пока очередь не пуста

- Считываем точку из очереди
- Определяем ее соседей, которые еще не раскрашены и не входят в границу
- Красим каждого соседа из предыдущего пункта и добавляем его в конец очереди

Конец цикла Пока

```
a[y,x] = 0 – не залито
a[y,x] = 1 – залито
a[y,x] = 2 – цвет границы
```

– начальная точка

Почему алгоритм не зацикливается?

Реализация алгоритма FloodFill

```
static void AddPixel(int[,] a, int x, int y, Queue<(int, int)> q)
{
  if (a[y,x] == 0)
  {
    a[y, x] = 1;
    q.Enqueue((x,y));
  }
}
static void FloodFill(int[,] a, int x, int y)
{
  var q = new Queue<(int,int)>();
  AddPixel(a, x, y, q);
  while (q.Count > 0)
  {
    (var xx, var yy) = q.Dequeue(); // распаковка кортежа в переменные
    AddPixel(a, xx-1, yy, q);
    AddPixel(a, xx+1, yy, q);
    AddPixel(a, xx, yy-1, q);
    AddPixel(a, xx, yy+1, q);
  }
}
```

Риторический вопрос

Мешало ли нам при написании алгоритма FloodFill то, что мы не знали внутреннее устройство очереди?

Стандартные классы коллекций .NET

Стандартные классы коллекций находятся в пространстве имён System.Collections.Generic. **Нам известны следующие:**

- Stack<T>
- Queue<T>
- List<T> - динамический массив с возможностью расширения
- HashSet<T> - неупорядоченное множество (операции Add, Remove, Contains выполняются за время Θ(1))
- SortedSet<T> - упорядоченное множество (операции Add, Remove, Contains выполняются за время Θ(log(n)))
- Dictionary<Key,Value> - неупорядоченный словарь пар (Ключ, Значение) (все операции выполняются за время Θ(1))
- SortedDictionary<Key,Value> - упорядоченный словарь пар (Ключ, Значение) (все операции выполняются за время Θ(log(n)))

Коллекции как последовательности

- Все коллекции являются **последовательностями**
- Это значит, что по любой коллекции возможен цикл foreach, любую коллекцию можно присваивать переменной IEnumerable<T> и применить к ней все методы последовательностей

Примеры

```
var hs = new HashSet<int> {1, 2, 5, 8, 3, 15 };
foreach (var x in hs)
  Write($"{x} ");
WriteLine();

var st = new Stack<double>();
st.Push(2); st.Push(5); st.Push(3);
WriteLine(string.Join(" ", st)); // 3 5 2

static void Println<T>(IEnumerable<T> en)
{
    WriteLine(string.Join(" ", en));
}
...
Println(st); // принцип подстановки Барбары Лискоу
Println(hs);
Println(new SortedSet<int>(hs));
Println(new List<int> {4, 6, 5});
WriteLine(hs.Sum());
```

Словарь Dictionary<Key,Value> (ассоциативный массив)

- **Словарь** – последовательность пар (Ключ, Значение) с быстрым поиском значения по ключу
- Пример: англо-русский словарь – это набор пар вида ("cat", "кошка"), ("dog", "собака"), ("pig", "свинья")
- Чтобы найти значение по ключу, используется запись dict["dog"], напоминающая обращение к элементу массива, отсюда название "**ассоциативный массив**".

Пример

```
var zoo = new Dictionary<string, int>();
zoo["Крокодил"] = 3;
zoo["Какаду"] = 2;
zoo["Какаду"] = zoo["Какаду"] + 1;

foreach (var x in zoo)
    WriteLine(x);

foreach (var x in zoo)
    WriteLine($"{x.Key} {x.Value}");
```

Вывод:
[Крокодил, 3]
[Какаду, 3]

Имеет тип **KeyValuePair -**
структура со свойствами **Key** и **Value**

Инициализатор словаря

В современном C# с версии 6.0 словари можно инициализировать с помощью инициализаторов словарей:

```
var zoo = new Dictionary<string, int>
{
    ["Крокодил"] = 3,
    ["Какаду"] = 2,
};
```

Индексное свойство

- Как реализуется обращение по индексу в словаре? С помощью **индексного свойства**
- Индексное свойство реализуется в виде пары методов: getter для доступа на чтение и setter для доступа на запись

Анатомия индексного свойства для Dictionary<K,V>

```
class Dictionary<Key,Value>
{
  public Value this[Key k]
  {
    get { }  // генерируется Value GetValue(Key k);
    set { }  // генерируется void SetValue(Key k, Value value);
  }
}
```

Использование:

```
var zoo = new Dictionary<string, int>();
zoo["Крокодил"] = 3; // заменяется на zoo.SetValue("Крокодил",3)
var n = zoo["Крокодил"]; // заменяется на n = zoo.GetValue("Крокодил")
```

Реализация словаря

- Ниже дана простейшая реализация словаря на базе списка KeyValuePair
- Данная реализация имеет асимптотическую сложность Θ(n) (это плохо)
- Реальная реализация сложнее и имеет лучшую асимптотическую сложность

Возможная реализация индексного свойства для Dictionary<K,V>

```
class Dictionary<Key,Value>
{
  private List<KeyValuePair> l = new List<KeyValuePair>();
  public Value this[Key k]
  {
    get
    {
      var i = l.FindIndex(x => x.Key = k);
      if (i>=0)
        return l[i].Value;
      else throw new KeyNotFoundException();
    }
    set
    {
      var i = l.FindIndex(x => x.Key = k);
      if (i>=0)
        l[i].Value = value;
      else l.Add(new KeyValuePair<Key,Value>(k,value);
    }
  }
}
```

Асимптотическая сложность Θ(n)

АТД "Словарь" реализован здесь на базе другого АТД "Список"

Исключение KeyNotFoundException и проверка на существование ключа

Если в словаре обратиться к несуществующему ключу на чтение, то возникнет исключение KeyNotFoundException:

```
var zoo = new Dictionary<string, int>();
WriteLine(zoo["Бегемот"]);
```

Чтобы этого не произошло, необходимо предварительно проверить, существует ли такой ключ в словаре:

```
if (zoo.ContainsKey("Бегемот"))
  WriteLine(zoo["Бегемот"]);
```

Как безопасно увеличить количество бегемотов на 1?

```
if (zoo.ContainsKey("Бегемот"))
  zoo["Бегемот"] += 1;
else zoo["Бегемот"] = 1;
```

Другие методы словаря

Очистка словаря:

```
zoo.Clear();
```

Удаление значения с указанным ключом (возвращает true если удалено):

```
bool b = zoo.Remove("Бегемот");
```

Последовательность всех ключей словаря:

```
foreach (var k in zoo.Keys)
  WriteLine(zoo[k]);
```

Осторожно проверить, сколько бегемотов (если нет, вернётся false)

```
var b = zoo.TryGetValue("Бегемот", out int v);
```

Пример: словарь количества встречаемости слов в файле

```
var text = File.ReadAllText(@"..\..\Program.cs");

var wordcount = new Dictionary<string, int>();
foreach (Match x in Regex.Matches(text, @"[a-zA-Z]{2,}"))
    if (wordcount.ContainsKey(x.Value))
        wordcount[x.Value] += 1;
    else wordcount[x.Value] = 1;

foreach (var x in wordcount.OrderByDescending(kv => kv.Value).
                  Select(kv => kv.Value+" - " + kv.Key).Take(10))
    WriteLine(x);
```

Вывод:

```
12 - var
9 - zoo
8 - using
8 - System
7 - WriteLine
7 - int
6 - foreach
6 - in
6 - Value
6 - wordcount
```

Q & A