Лекция 6.
Амортизационный анализ

Даниил Михайлович Берлизов
Старший преподаватель Кафедры вычислительных систем СибГУТИ
E-mail: sillyhat34@gmail.com

Курс «Структуры и алгоритмы обработки данных»
Осенний семестр, 2021 г.

Анализ вычислительной сложности алгоритмов

1. Определяем параметры алгоритма, от которых зависит время его выполнения

n, m

2. Выражаем количество операций, выполняемых алгоритмом, как функцию от его параметров (для худшего, среднего и лучшего случаев)

$T(n, m) = 2n^2 + 4\log_2 m$

3. Строим асимптотическую оценку вычислительной сложности алгоритма — переходим к асимптотическим обозначениям: O, Θ, Ω

$T(n, m) = O(2n^2 + 4\log_2 m) =$
$= O(\max\{n^2, \log_2 m\})$

Анализ сортировки выбором, худший случай

```
function SelectionSort(v[1..n])
    for i = 1 to n - 1 do
        min = i
        for j = i + 1 to n do
            if v[j] < v[min] then
                min = j
            end if
        end for
        if min != i then
            temp = v[i]
            v[i] = v[min]
            v[min] = temp
        end if
    end for
end function
```

Анализ сортировки выбором, худший случай

```
function SelectionSort(v[1..n])
    for i = 1 to n - 1 do
        min = i                                  ├─ 1 операция
        for j = i + 1 to n do
            if v[j] < v[min] then
                min = j
            end if
        end for
        if min != i then
            temp = v[i]
            v[i] = v[min]
            v[min] = temp
        end if
    end for
end function
```

Анализ сортировки выбором, худший случай

```
function SelectionSort(v[1..n])
    for i = 1 to n - 1 do
        min = i
        for j = i + 1 to n do
            if v[j] < v[min] then
                min = j
            end if
        end for
        if min != i then
            temp = v[i]
            v[i] = v[min]
            v[min] = temp
        end if
    end for
end function
```

1 операция

2 операции
(худший случай)

Анализ сортировки выбором, худший случай

```
function SelectionSort(v[1..n])
    for i = 1 to n - 1 do
        min = i
        for j = i + 1 to n do
            if v[j] < v[min] then
                min = j
            end if
        end for
        if min != i then
            temp = v[i]
            v[i] = v[min]
            v[min] = temp
        end if
    end for
end function
```

1 операция

2 операции
(худший случай)

4 операции

Анализ сортировки выбором, худший случай

```
function SelectionSort(v[1..n])
    for i = 1 to n - 1 do
        min = i
        for j = i + 1 to n do
            if v[j] < v[min] then
                min = j
            end if
        end for
        if min != i then
            temp = v[i]
            v[i] = v[min]
            v[min] = temp
        end if
    end for
end function
```

1 операция

2 операции
(худший случай)

4 операции

$i = 1, j = 2$ to n, $n - 1$ итер.
$i = 2, j = 3$ to n, $n - 2$ итер.
$i = 3, j = 4$ to n, $n - 3$ итер.
…
$i = n - 1, j = n$ to n, 1 итер.

Анализ сортировки выбором, худший случай

```
function SelectionSort(v[1..n])
    for i = 1 to n - 1 do
        min = i
        for j = i + 1 to n do
            temp = v[i]
            v[i] = v[min]
            v[min] = temp
        end if
    end for
end function
```

1 операция

4 операции

$$T(n) = (n-1) + 4(n-1) + 2((n-1) + (n-2) + \ldots + 1) = ?$$

$i = 1, j = 2$ to n, $n - 1$ итер.
$i = 2, j = 3$ to n, $n - 2$ итер.
$i = 3, j = 4$ to n, $n - 3$ итер.
…
$i = n - 1, j = n$ to n, 1 итер.

Анализ сортировки выбором, худший случай

```
function SelectionSort(v[1..n])
    for i = 1 to n - 1 do
        min = i
        for j = i + 1 to n do
            temp = v[i]
            v[i] = v[min]
            v[min] = temp
        end if
    end for
end function
```

1 операция

4 операции

Сумма членов арифметической прогрессии

$$T(n) = (n-1) + 4(n-1) + 2((n-1) + (n-2) + \ldots + 1) = ?$$

$i = 1, j = 2$ to n, $n - 1$ итер.
$i = 2, j = 3$ to n, $n - 2$ итер.
$i = 3, j = 4$ to n, $n - 3$ итер.
…
$i = n - 1, j = n$ to n, 1 итер.

Анализ сортировки выбором, худший случай

```
function SelectionSort(v[1..n])
    for i = 1 to n - 1 do
        min = i
        for j = i + 1 to n do
            temp = v[i]
            v[i] = v[min]
            v[min] = temp
        end if
    end for
end function
```

1 операция

$$T(n) = 5n - 5 + 2((n^2 - n) / 2) = n^2 + 4n - 5$$

4 операции

$i = 1, j = 2$ to n, $n - 1$ итер.
$i = 2, j = 3$ to n, $n - 2$ итер.
$i = 3, j = 4$ to n, $n - 3$ итер.
…
$i = n - 1, j = n$ to n, 1 итер.

Анализ сортировки выбором, худший случай

```
function SelectionSort(v[1..n])
    for i = 1 to n - 1 do
        min = i
        for j = i + 1 to n do
            temp = v[i]
            v[i] = v[min]
            v[min] = temp
        end if
    end for
end function
```

1 операция

$$T(n) = 5n - 5 + 2((n^2 - n) / 2) = n^2 + 4n - 5 = O(n^2)$$

4 операции

$i = 1, j = 2$ to n, $n - 1$ итер.
$i = 2, j = 3$ to n, $n - 2$ итер.
$i = 3, j = 4$ to n, $n - 3$ итер.
…
$i = n - 1, j = n$ to n, 1 итер.

Бинарный счётчик

- Счётчик имеет длину L бит и может принимать 2^L значений
- Поддерживает операцию *Increment*, которая увеличивает его значение на единицу
- Начальное значение счётчика — 0

- Пример 5-разрядного счётчика:

Разряд	4	3	2	1	0
Вес	16	8	4	2	1
Значение	**0**	**0**	**0**	**0**	**0**

Бинарный счётчик

Increment 0 → 1

Разряд	4	3	2	1	0
Вес	16	8	4	2	1
Значение	**0**	**0**	**0**	**0**	**1**

Increment 1 → 2

Разряд	4	3	2	1	0
Вес	16	8	4	2	1
Значение	**0**	**0**	**0**	**1**	**0**

Бинарный счётчик

Increment 2 → 3

Разряд	4	3	2	1	0
Вес	16	8	4	2	1
Значение	**0**	**0**	**0**	**1**	**1**

Increment 3 → 4

Разряд	4	3	2	1	0
Вес	16	8	4	2	1
Значение	**0**	**0**	**1**	**0**	**0**

Бинарный счётчик

Increment 4 → 5

Разряд	4	3	2	1	0
Вес	16	8	4	2	1
Значение	**0**	**0**	**1**	**0**	**1**

Increment 5 → 6

Разряд	4	3	2	1	0
Вес	16	8	4	2	1
Значение	**0**	**0**	**1**	**1**	**0**

Бинарный счётчик

```
function Increment(A)
    i = 0
    while i < L and A[i] = 1 do
        A[i] = 0
        i = i + 1
    end while
    if i < L then
        A[i] = 1
    end if
end function
```

Перенос в старший разряд — замена начальной последовательности единиц нулями

Разряд	3	2	1	0
Вес	8	4	2	1
Значение	1	0	1	1

$= 11_{10}$

Бинарный счётчик

```
function Increment(A)
    i = 0
    while i < L and A[i] = 1 do
        A[i] = 0
        i = i + 1
    end while
    if i < L then
        A[i] = 1
    end if
end function
```

- При каждом вызове функции *Increment* время её работы различается
- Время зависит от внутреннего состояния счётчика — значений $A[1..L]$

Разряд	3	2	1	0
Вес	8	4	2	1
Значение	**1**	**0**	**1**	**1**

$= 11_{10}$

Бинарный счётчик

```
function Increment(A)
    i = 0
    while i < L and A[i] = 1 do
        A[i] = 0
        i = i + 1
    end while
    if i < L then
        A[i] = 1
    end if
end function
```

- При каждом вызове функции *Increment* время её работы различается
- Время зависит от внутреннего состояния счётчика — значений $A[1..L]$

Вычислительная сложность функции Increment?

Бинарный счётчик

- В худшем случае массив $A[1..L]$ состоит только из единиц, для выполнения операции Increment требуется время $O(L)$
- **Это пессимистическая оценка!**
- При каждом вызове функции Increment время её работы различается

Как анализировать вычислительную сложность алгоритмов,
время выполнения которых зависит от их внутреннего состояния?

Амортизационный анализ

- **Амортизационный анализ** (*amortized analysis*) — метод анализа алгоритмов, позволяющий осуществлять оценку времени выполнения последовательности из *n* операций над некоторой структурой данных
- Время выполнения усредняется по всем *n* операциям, и оценивается **среднее время одной операции в худшем случае**

- Амортизационный анализ возник из группового анализа (*aggregate analysis*)
- Введён в практику Робертом Тарьяном (Robert Tarjan) в 1985 году

Tarjan R. **Amortized Computational Complexity** // SIAM. J. on Algebraic and Discrete Methods, 6(2), 1985. — P. 306-318.

Амортизационный анализ

- Некоторые операции структуры данных могут иметь высокую вычислительную сложность, другие — низкую
- Например, некоторая операция может подготавливать структуру данных для быстрого выполнения других операций
- Такие «тяжёлые» операции выполняются редко и могут оказывать незначительное влияние на суммарное время последовательности из n операций

Методы амортизационного анализа

- **Групповой анализ** (*aggregate analysis*)
- **Метод бухгалтерского учёта** (*accounting method*)
- **Метод потенциалов** (*potential method*)

Все методы позволяют получить одну и ту же оценку, но разными способами

[CLRS 3ed., C. 487]

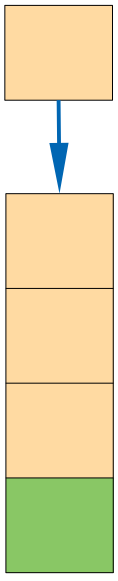
Групповой анализ

- **Групповой анализ** (*aggregate analysis*) — метод амортизационного анализа, позволяющий оценивать верхнюю границу времени $T(n)$ выполнения последовательности из n операций в худшем случае
- Амортизированная стоимость (учётная стоимость, *amortized cost*) выполнения одной операции определяется как

$$T(n) / n$$

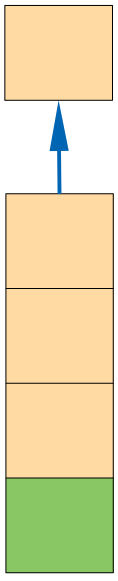
- Амортизированная стоимость операций — это оценка сверху *среднего времени выполнения операций в худшем случае*

Стековые операции (Last In — First Out)



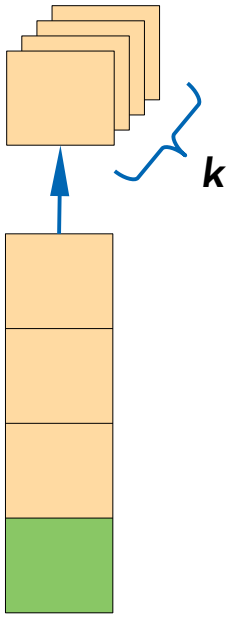
Push(*S*, *x*)

$T_{Push} = O(1)$



Pop(*S*)

$T_{Pop} = O(1)$



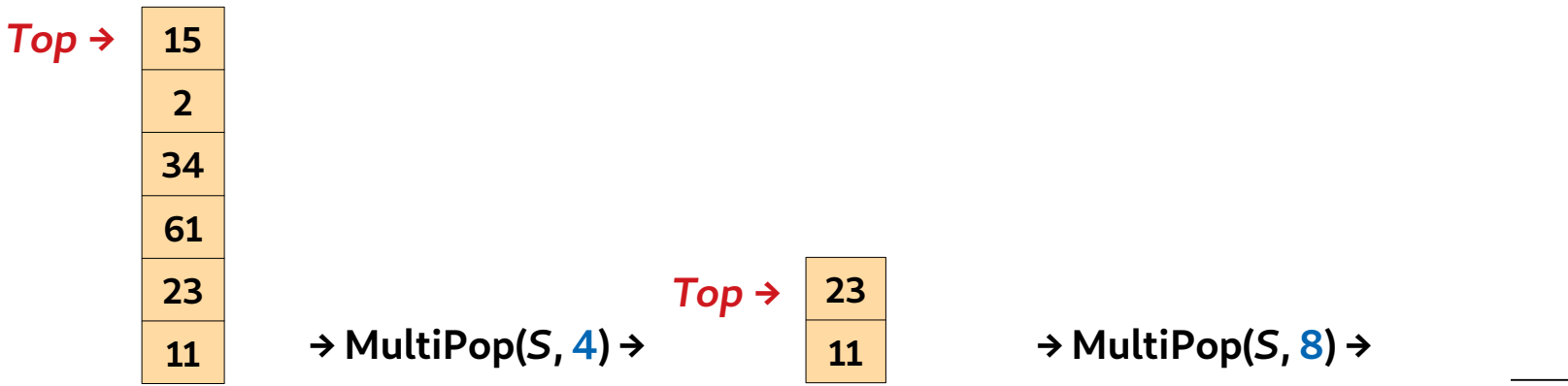
MultiPop(*S*, *k*)

$T_{MultiPop} = O(\min\{|S|, k\})$

Стековые операции (Last In — First Out)

```
function MultiPop(S, k)
    while StackEmpty(S) = False and k > 0 do
        StackPop(S)
        k = k - 1
    end while
end function
```

$T_{MultiPop} = O(\min\{|S|, k\})$



Групповой анализ стековых операций

- Методом группового анализа оценим верхнюю границу времени $T(n)$ выполнения произвольной последовательности из n стековых операций (Push, Pop, MultiPop)

  1. Стоимость операции Pop равна $O(1)$
  2. Стоимость операции Push равна $O(1)$
  3. Стоимость операции MultiPop в худшем случае $O(n)$, так как в ходе выполнения n операций в стеке не может находиться более n объектов

- В худшем случае последовательность из n операций может содержать только операции MultiPop
- Тогда, суммарное время $T(n)$ выполнения n операций есть $\boldsymbol{O(n^2)}$, а амортизированная стоимость одной операции

$$\boldsymbol{O(n^2) / n = O(n)}$$

Грубая оценка сверху

Групповой анализ стековых операций

- Построим более точную оценку сверху времени $T(n)$ выполнения произвольной последовательности из n стековых операций

  1. Количество операций Pop (включая вызовы из MultiPop) не превышает количества операций Push. В свою очередь, число операций Push не превышает n (операция MultiPop реализована на базе Pop)
  2. Таким образом, для выполнения произвольной последовательности из n операций Push, Pop, MultiPop требуется время $O(n)$

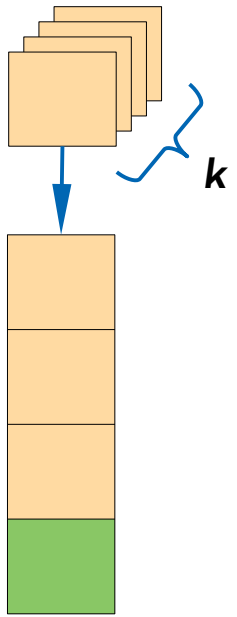
- Суммарное время выполнения n операций в худшем случае есть $\boldsymbol{O(n)}$, тогда *амортизированная стоимость* (средняя стоимость) одной операции над стеком есть

$$\boldsymbol{O(n) / n = O(1)}$$

Групповой анализ стековых операций

Вопрос:

- Останется ли справедливой оценка амортизированной стоимости стековых операций, равная $O(1)$, если включить в множество стековых операций операцию MultiPush(S, k), помещающую в стек k элементов?



MultiPush(S, k)

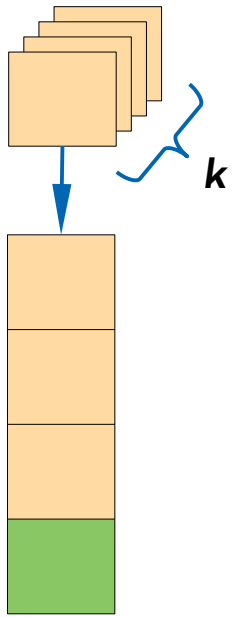
$T_{MultiPush} = O(k)$

Групповой анализ стековых операций

Ответ: нет

Оценка амортизированной стоимости одной стековой операции станет ***O(k)***

- В последовательности из *n* стековых операций может быть *n* операций MultiPush, что требует времени *O*(*nk*)
- Следовательно, амортизированная (средняя) стоимость одной стековой операции *T*(*n*) / *n* = *O*(*nk*) / *n* = *O*(*k*)



MultiPush(*S*, *k*)

Бинарный счётчик

- Счётчик имеет длину L бит и может принимать 2^L значений
- Поддерживает операцию *Increment*, которая увеличивает его значение на единицу
- Пример: 5-разрядный счётчик
  - Значение счётчика: 1, битовая последовательность: **00001**
  - Значение счётчика: 4, битовая последовательность: **00100**
  - Значение счётчика: 5, битовая последовательность: **00101**
  - Значение счётчика: 10, битовая последовательность: **01010**

Разряд	4	3	2	1	0
Вес	16	8	4	2	1
Значение	**1**	**0**	**1**	**1**	**0**

Бинарный счётчик

```
function Increment(A)
    i = 0
    while i < L and A[i] = 1 do
        A[i] = 0
        i = i + 1
    end while
    if i < L then
        A[i] = 1
    end if
end function
```

- При каждом вызове функции *Increment* время её работы различается
- Время зависит от внутреннего состояния счётчика — значений $A[1..L]$

Разряд	4	3	2	1	0
Вес	16	8	4	2	1
Значение	**1**	**0**	**1**	**1**	**0**

Бинарный счётчик

Значение	A[L]	...	A[4]	A[3]	A[2]	A[1]	A[0]	Стоимость (число операций)	Суммарная стоимость
1	0		0	0	0	0	1	1	1
2	0		0	0	0	1	0	2	3
3	0		0	0	0	1	1	1	4
4	0		0	0	1	0	0	3	7
5	0		0	0	1	0	1	1	8
6	0		0	0	1	1	0	2	10
7	0	...	0	0	1	1	1	1	11
8	0		0	1	0	0	0	4	15
9	0		0	1	0	0	1	1	16
10	0		0	1	0	1	0	2	18
11	0		0	1	0	1	1	1	19
12	0		0	1	1	0	0	3	22

Бинарный счётчик

Значение	A[L]	...	A[4]	A[3]	A[2]	A[1]	A[0]	Стоимость (число операций)	Суммарная стоимость
1	0		0	0	0	0	1	1	1
2	0		0	0	0	1	0	2	3
3	0		0	0	0	1	1	1	4
8	0		0	1	0	0	0	4	15
9	0		0	1	0	0	1	1	16
10	0		0	1	0	1	0	2	18
11	0		0	1	0	1	1	1	19
12	0		0	1	1	0	0	3	22

Какова амортизированная стоимость выполнения функции Increment (среднее время в худшем случае)?

Бинарный счётчик

Значение	$A[L]$	...	$A[4]$	$A[3]$	$A[2]$	$A[1]$	$A[0]$	Стоимость (число операций)	Суммарная стоимость
1	0		0	0	0	0	1	1	1
2	0		0	0	0	1	0	2	3
10	0		0	1	0	1	0	2	18
11	0		0	1	0	1	1	1	19
12	0		0	1	1	0	0	3	22

- Оценим сверху время $T(n)$ выполнения n операций Increment
- Тогда амортизированная стоимость операции Increment (среднее время выполнения) будет

$$T(n) / n$$

Бинарный счётчик

Значение	$A[L]$	...	$A[4]$	$A[3]$	$A[2]$	$A[1]$	$A[0]$	Стоимость (число операций)	Суммарная стоимость
1	0		0	0	0	0	1	1	1
2	0		0	0	0	1	0	2	3
3	0								4
	0								
n	0								$T(n)$
6	0		0	0	1	1	0	2	10
7	0	...	0	0	1	1	1	1	11
8	0		0	1	0	0	0	4	15
9	0		0	1	0	0	1	1	16
10	0		0	1	0	1	0	2	18
11	0		0	1	0	1	1	1	19
12	0		0	1	1	0	0	3	22

Можно заметить, что время выполнения n операций

$2n \geq T(n)$

Бинарный счётчик

Значение	A[L]	...	A[4]	A[3]	A[2]	A[1]	A[0]	Стоимость (число операций)	Суммарная стоимость
1	0								1
2	0								3
3	0								4
4	0								7
5	0								8
6	0								10
7	0								11
8	0								15
9	0								16
10	0								18
11	0		0	1	0	1	1	1	19
12	0		0	1	1	0	0	3	22

- Бит 0 изменяется n раз (при каждом вызове Increment)
- Бит 1: $\lfloor n / 2 \rfloor$ раз
- Бит 2: $\lfloor n / 4 \rfloor$ раз
- ...
- Бит $L - 1$ изменяется $\lfloor n / 2^{L-1} \rfloor$ раз

$$T(n) = n + \frac{n}{2} + \frac{n}{4} + \ldots + \frac{n}{2^{L-1}} = n + n = 2n$$

Бинарный счётчик

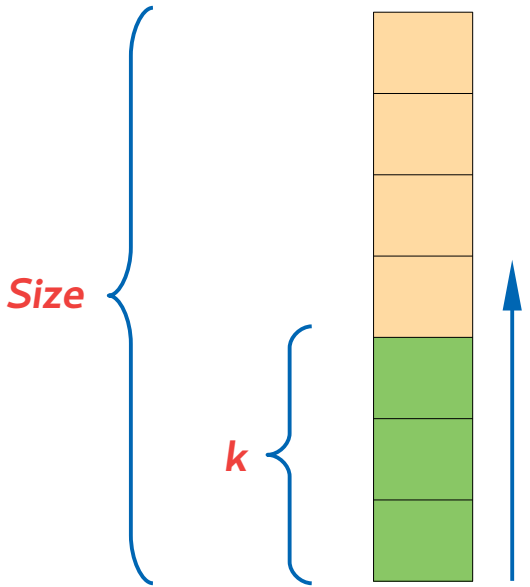
Значение	$A[L]$	...	$A[4]$	$A[3]$	$A[2]$	$A[1]$	$A[0]$	Стоимость (число операций)	Суммарная стоимость
1	0		0	0	0	0	1	1	1
2	0		0	0	0	1	0	2	3
3	0		0	0	0	1	1	1	4
8	0		0	1	0	0	0	4	15
9	0		0	1	0	0	1	1	16
10	0		0	1	0	1	0	2	18
11	0		0	1	0	1	1	1	19
12	0		0	1	1	0	0	3	22

Средняя (амортизированная) стоимость одной операции Increment

$O(n) / n = O(1)$

Динамические таблицы

- **Динамическая таблица** (*dynamic table*, *dynamic array*, *growable array*) — это массив, поддерживающий вставку и удаление элементов и динамически изменяющий свой размер до необходимого значения
- Поддерживаемые операции:
  - ***Insert***(***T***, ***x***)
  - ***Delete***(***T***, ***x***)
  - ***Size*** — количество свободных ячеек
  - ***k*** — количество элементов, добавленных в массив



Динамические таблицы

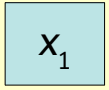
- При выполнении операции Insert размер массива увеличивается
- Как увеличивать размер массива?
- **Аддитивная схема** — размер массива увеличивается на *k* ячеек (арифметическая прогрессия)
- **Мультипликативная схема** — размер массива увеличивается в *k* раз (геометрическая прогрессия)
- **Примеры реализации:**
  - **C++:** std::vector
  - **Java:** ArrayList
  - **.NET 2.0:** List<>
  - **Python:** list

Динамические таблицы

```
function Insert(x)
    if Size = 0 then
        Size = 1
        Table = AllocateMemory(1)
        k = 0
    else if k = Size then
        Size = Size * 2
        NewTable = AllocateMemory(Size)
        for i = 0 to k - 1 do
            NewTable[i] = Table[i]
        end for
        FreeMemory(Table)
        Table = NewTable
    end if
    Table[k] = x
    k = k + 1
end function
```

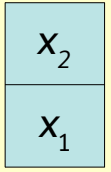
Динамические таблицы

```
function Insert(x)
    if Size = 0 then
        Size = 1
        Table = AllocateMemory(1)
        k = 0
    else if k = Size then
        Size = Size * 2
        NewTable = AllocateMemory(Size)
        for i = 0 to k - 1 do
            NewTable[i] = Table[i]
        end for
        FreeMemory(Table)
        Table = NewTable
    end if
    Table[k] = x
    k = k + 1
end function
```



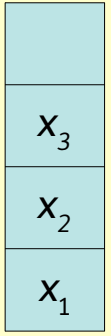
Динамические таблицы

```
function Insert(x)
    if Size = 0 then
        Size = 1
        Table = AllocateMemory(1)
        k = 0
    else if k = Size then
        Size = Size * 2
        NewTable = AllocateMemory(Size)
        for i = 0 to k - 1 do
            NewTable[i] = Table[i]
        end for
        FreeMemory(Table)
        Table = NewTable
    end if
    Table[k] = x
    k = k + 1
end function
```



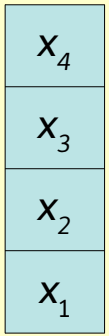
Динамические таблицы

```
function Insert(x)
    if Size = 0 then
        Size = 1
        Table = AllocateMemory(1)
        k = 0
    else if k = Size then
        Size = Size * 2
        NewTable = AllocateMemory(Size)
        for i = 0 to k - 1 do
            NewTable[i] = Table[i]
        end for
        FreeMemory(Table)
        Table = NewTable
    end if
    Table[k] = x
    k = k + 1
end function
```



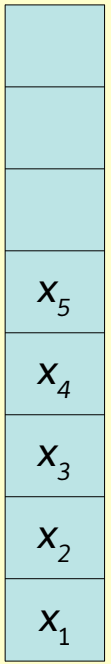
Динамические таблицы

```
function Insert(x)
    if Size = 0 then
        Size = 1
        Table = AllocateMemory(1)
        k = 0
    else if k = Size then
        Size = Size * 2
        NewTable = AllocateMemory(Size)
        for i = 0 to k - 1 do
            NewTable[i] = Table[i]
        end for
        FreeMemory(Table)
        Table = NewTable
    end if
    Table[k] = x
    k = k + 1
end function
```



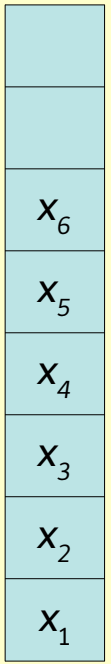
Динамические таблицы

```
function Insert(x)
    if Size = 0 then
        Size = 1
        Table = AllocateMemory(1)
        k = 0
    else if k = Size then
        Size = Size * 2
        NewTable = AllocateMemory(Size)
        for i = 0 to k - 1 do
            NewTable[i] = Table[i]
        end for
        FreeMemory(Table)
        Table = NewTable
    end if
    Table[k] = x
    k = k + 1
end function
```



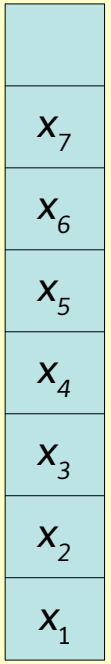
Динамические таблицы

```
function Insert(x)
    if Size = 0 then
        Size = 1
        Table = AllocateMemory(1)
        k = 0
    else if k = Size then
        Size = Size * 2
        NewTable = AllocateMemory(Size)
        for i = 0 to k - 1 do
            NewTable[i] = Table[i]
        end for
        FreeMemory(Table)
        Table = NewTable
    end if
    Table[k] = x
    k = k + 1
end function
```



Динамические таблицы

```
function Insert(x)
    if Size = 0 then
        Size = 1
        Table = AllocateMemory(1)
        k = 0
    else if k = Size then
        Size = Size * 2
        NewTable = AllocateMemory(Size)
        for i = 0 to k - 1 do
            NewTable[i] = Table[i]
        end for
        FreeMemory(Table)
        Table = NewTable
    end if
    Table[k] = x
    k = k + 1
end function
```

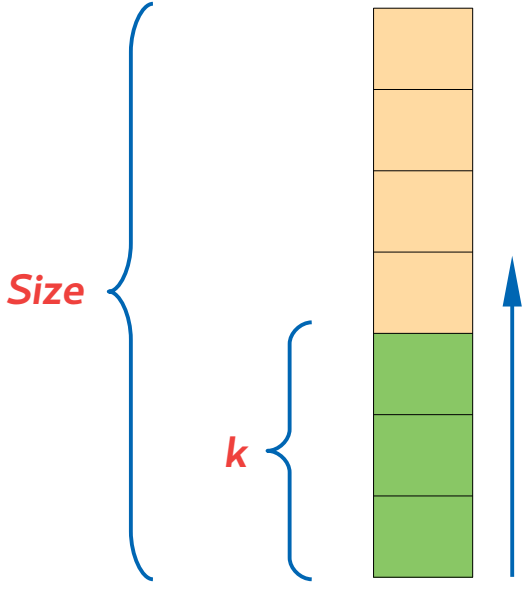


Динамические таблицы

Проведём амортизационный анализ времени $T(n)$ выполнения последовательности из n операций Insert

- Время работы операции Insert зависит от состояния таблицы: количества k элементов и её размера *Size*
- Будем учитывать только операции записи элементов в таблицу

$$Table[k] = x$$



Амортизационный анализ операции Insert

```
function Insert(x)
    if Size = 0 then
        Size = 1
        Table = AllocateMemory(1)
        K = 0
    else if k = Size then
        Size = Size * 2
        NewTable = AllocateMemory(Size)
        for i = 0 to k - 1 do
            NewTable[i] = Table[i]
        end for
        FreeMemory(Table)
        Table = NewTable
    end if
    Table[k] = x
    k = k + 1
end function
```

- Первый вызов Insert — **1 операция**
- Второй вызов — **2 оп.** (Copy 1 + Write 1)
- Третий — **3 оп.** (Copy 2 + Write 1)
- Четвёртый — **1 оп.**
- Пятый — **5 оп.** (Copy 4 + Write 1)
- …

Амортизационный анализ операции Insert

- Обозначим через c_i количество операций, выполняемых на i-ом вызове Insert

$$c_i = \begin{cases} i, & \text{если } i-1 \text{ степень 2} \\ 1, & \text{иначе} \end{cases}$$

- Тогда оценка сверху времени T(n) выполнения n операций Insert есть

$$T(n) = c_1 + c_2 + \ldots + c_n \le n + 2^0 + 2^1 + 2^2 + \ldots + 2^{\lfloor \log_2 n \rfloor}$$

$$T(n) < n + 2n = 3n$$

Амортизационный анализ операции Insert

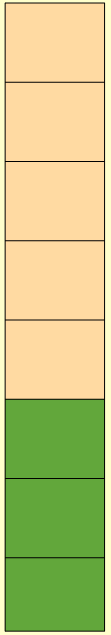
- Оценка сверху амортизированной сложности одной операции Insert есть

$$T_{Insert} = \frac{T(n)}{n} = \frac{3n}{n} = 3 = \boldsymbol{O}(\mathbf{1})$$

Среднее время выполнения (вычислительная сложность) одной операции Insert в худшем случае есть *O*(1)

Обработчик с накопителем

```
function AddToBuffer(value)
    Count = Count + 1
    Buffer[Count] = value
    if Count = H then
        for i = 1 to H do
            Packet[i] = Buffer[i]
        end for
        Count = 0
    end if
end function
```



Обработчик с накопителем

- Построить оценку сверху времени $T(n)$ выполнения n операций AddToBuffer
- Оценить амортизированную сложность функции AddToBuffer

Амортизационный анализ операции AddToBuffer

- Обозначим через c_i количество операций, выполняемых на i-ом вызове AddToBuffer

$$c_i = \begin{cases} H+3, & \text{если } i \% H = 0 \\ 1, & \text{иначе} \end{cases}$$

- Тогда оценка сверху времени T(n) выполнения n операций AddToBuffer есть

$$T(n) = c_1 + c_2 + \ldots + c_n < 2n + \frac{n(3+H)}{H} < 2n + 3n + H = 5n + H$$

$$T(n) = O(5n + H) = O(n + H)$$

Амортизационный анализ операции AddToBuffer

- Оценка сверху времени $T(n)$ выполнения n операций AddToBuffer

$$T(n) = O(5n + H) = O(n + H)$$

- Амортизированная стоимость (сложность) одной операции AddToBuffer

$$\frac{T(n)}{n} = \frac{5n + H}{n} = \boldsymbol{O(1)}$$

Дальнейшее чтение

- Прочитать в **[CLRS 3ed., C. 487]**:
  - 17.2 Метод бухгалтерского учёта
  - 17.3 Метод потенциалов

Кормен Т. Х., Лейзерсон Ч. И., Ривест Р. Л., Штайн К. **Алгоритмы: построение и анализ.** — 3-е изд. — М.: Вильямс, 2014. — 1328 с.

ご清聴ありがとうございました!

Даниил Михайлович Берлизов
Старший преподаватель Кафедры вычислительных систем СибГУТИ
E-mail: sillyhat34@gmail.com

Курс «Структуры и алгоритмы обработки данных»
Осенний семестр, 2021 г.