



Структуры данных

Абстрактные типы данных

Структура данных

представляет собой набор некоторым образом сгруппированных данных.

Для каждой структуры данных определяется:

- (1) каким образом данные хранятся в памяти компьютера,
- (2) какие базовые операции можно выполнять над этими данными и за какое время.

Примеры структур данных

Массив (*англ.* array)

Связный список (*англ.* linked list)

Бинарная куча (*англ.* binary heap) –
специализированная древовидная структура данных

Абстрактный тип данных (англ. *abstract data type*)

Для абстрактного типа определяется **интерфейс** — набор операций, которые могут быть выполнены. Пользователь абстрактного типа, используя эти операции, может работать с данными, не вдаваясь во внутренние детали механизма хранения информации.

Если алгоритм работает с данными исключительно через интерфейс, то он продолжит функционировать, если одну реализацию интерфейса заменить на другую. В этом и заключается суть абстракции: реализация скрыта за интерфейсом.

Реализациями абстрактных типов данных являются конкретные структуры данных. Реализация определяет, как именно представлены в памяти данные и как функционирует та или иная операция.

Примеры абстрактных типов данных:

Список (list)

Стек (stack)

Очередь (queue)

Двухсторонняя очередь (deque)

Множество (set)

Ассоциативный массив/отображение/ словарь (associative array/map/dictionar)

Приоритетная очередь (priority queue)

Структуры данных

Массив фиксированного размера (англ. array)

Массив—

это структура данных с **произвольным доступом к элементу** (англ. *random access*), т. е. доступ к любому элементу по индексу осуществляется за время $O(1)$ вне зависимости от того, где в массиве (одномерный или многомерный массив) располагается элемент (в отличие от последовательного доступа, когда время доступа к элементу зависит от места его расположения в структуре).

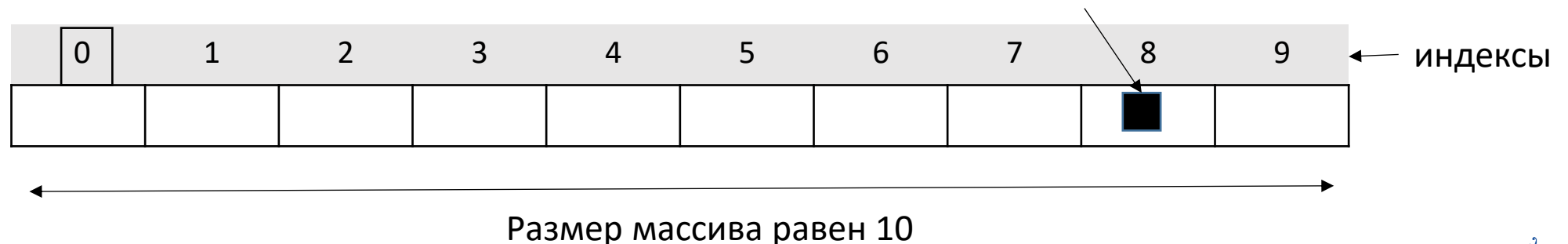
Структура **однородна**, так как все компоненты имеют один и тот же тип.

Под массив в памяти компьютера **выделяется непрерывный блок памяти**.

Элементы массива в памяти располагаются один за другим и являются **равнодоступными**. Индексами массива являются последовательные целые числа.

Начальный индекс

Элемент (с индексом 8)



	произвольный массив	упорядоченный массив
поиск элемента по ключу x	$\Theta(n)$	$O(\log n)$
добавление элемента	$\Theta(n)$	$\Theta(n)$
удаление элемента	$\Theta(n)$	$\Theta(n)$

Динамический массив (англ. dynamic array)

Размер массива в простейшем случае фиксирован и должен быть известен заранее.

На практике часто удобно использовать **динамический массив**, который можно расширять по мере надобности.

Динамический массив -

| структура данных, которая обеспечивает произвольный доступ и позволяет добавлять или удалять элементы.

Пусть изначально массив пуст, затем в него последовательно добавляют n элементов, при этом каждый раз новый элемент добавляется в конец.

**Как можно организовать динамический массив
на базе статического?**

Наивный подход

Первоначально массив состоит из одной свободной ячейки.

Каждый раз при необходимости изменения размера будем делать реаллокацию (англ. *Reallocation*, перемещение), т.е. выделять **новый массив** и перемещать все элементы из старого массива в новый.

Подсчитаем общее число «лишних» операций по перемещению данных.

$$0 + 1 + 2 + \dots + (n - 1) = \frac{n \cdot (n - 1)}{2}$$

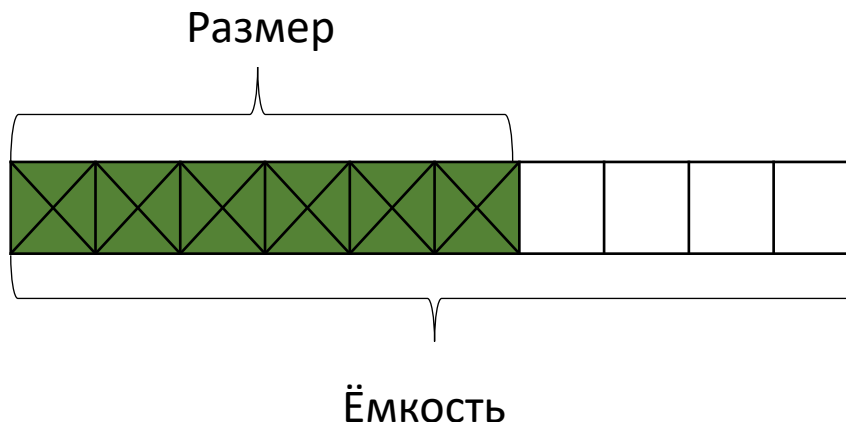
	память	«лишние операции»
1-й элемент	—	0
2-й элемент	<u>+</u> → + —	1
3-й элемент	<u>++</u> → ++ —	2
4-й элемент	<u>+++</u> → +++ —	3
n-й элемент	<u>+...+</u> → +...+ —	n – 1

Расширение с запасом

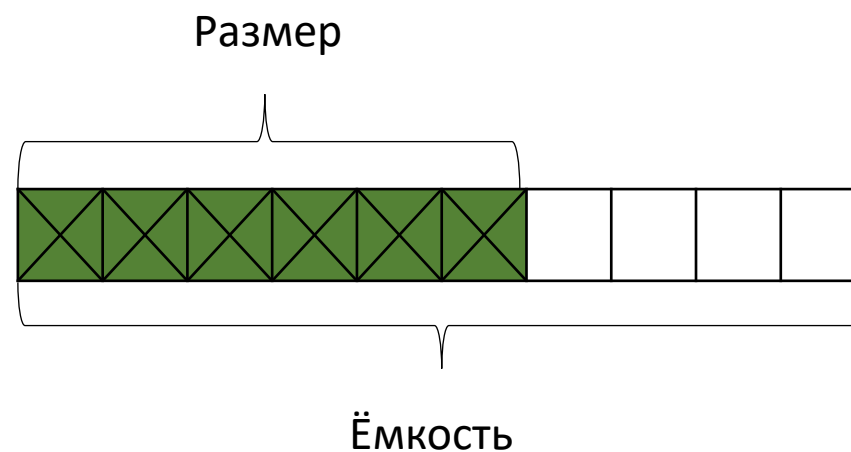
Для уменьшения числа реаллокаций будем расширять массив «с запасом», оставляя пустые ячейки, которые можно будет использовать на следующих шагах.

Число реально занятых ячеек памяти будем называть **логическим размером** (*size*) динамического массива.

Общее число зарезервированных ячеек будем называть **ёмкостью** (*capacity*).



Расширение с запасом: на сколько или во сколько раз?



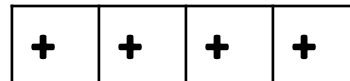
Расширение на Δ :

будем каждый раз расширять массив не на один элемент, а сразу на Δ элементов ($\Delta > 1$).

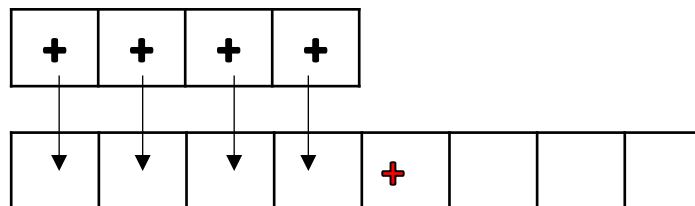
При добавлении первого элемента сразу будет выделен массив ёмкости Δ и в него будет занесён первый элемент.



Последующие (2-й, 3-й, ... , $\Delta-1$, Δ) элементы будут добавлены легко и быстро, так как не потребуется выполнять операции пере выделения памяти.

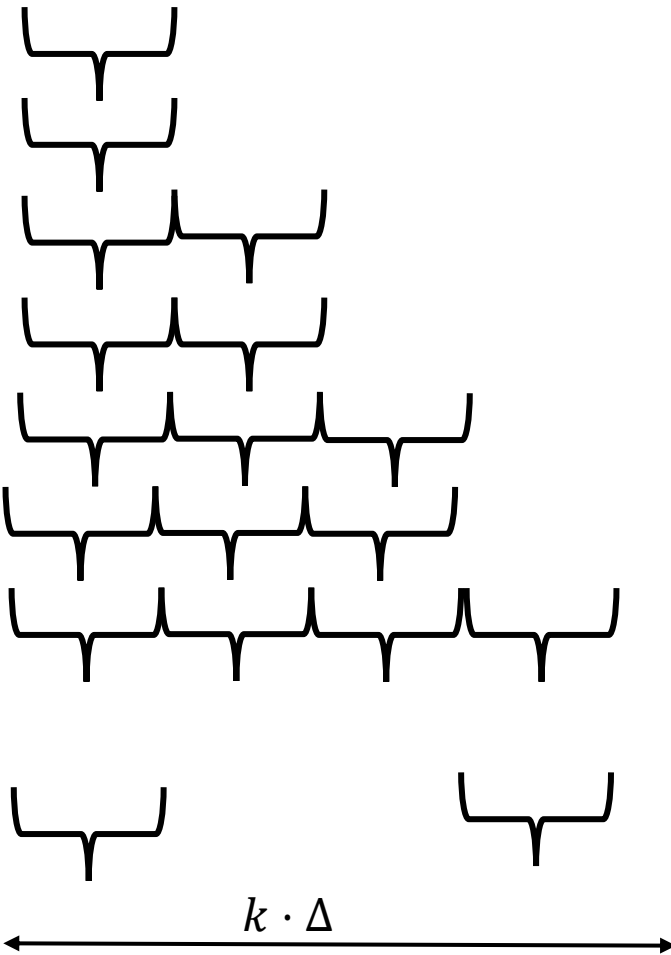


При поступлении ($\Delta+1$) элемента потребуется создать новый массив ёмкости $=[\Delta+ \Delta]$ и перенести все данные в него, затем уже сохранить новый элемент.



$(k - 1) \cdot \Delta < n \leq k \cdot \Delta$

элементы	«лишние операции»	ёмкость массива
начальный этап		Δ
$1, 2, \dots, \Delta$	—	Δ
$\Delta + 1$	Δ	$2 \cdot \Delta$
$\Delta + 2, \dots, 2 \cdot \Delta$	—	$2 \cdot \Delta$
$2 \cdot \Delta + 1$	$2 \cdot \Delta$	$3 \cdot \Delta$
$2 \cdot \Delta + 2, \dots, 3 \cdot \Delta$	—	$3 \cdot \Delta$
$3 \cdot \Delta + 1$	$3 \cdot \Delta$	$4 \cdot \Delta$
...
$(k - 1) \cdot \Delta + 1$	$(k - 1) \cdot \Delta$	$k \cdot \Delta$
$(k - 1) \cdot \Delta + 2, \dots, k \cdot \Delta$	—	$k \cdot \Delta$



$$(k - 1) \cdot \Delta < n \leq k \cdot \Delta$$

элементы	«лишние операции»	ёмкость
		Δ
1, 2,..., Δ	0	Δ
$\Delta+1$	Δ	2Δ
$\Delta+2, \dots, 2\Delta$	0	2Δ
$2\Delta+1$	2Δ	3Δ
$2\Delta+2, \dots, 3\Delta$	0	3Δ
$3\Delta+1$	3Δ	4Δ
...	...	
$(k-1) \Delta+1$	$(k-1) \Delta$	$k\Delta$
$(k-1) \Delta+2, \dots, k\Delta$	0	$k\Delta$

«Лишние» операции

оценка снизу

$$\Delta + 2\Delta + 3\Delta + \dots + (k - 1)\Delta = \Delta \cdot (1 + 2 + \dots + (k - 1)) = \Delta \cdot \frac{k(k - 1)}{2} =$$

$$= \frac{1}{2} \cdot \Delta \cdot k \cdot (k - 1) = \left[\begin{array}{l} (k - 1)\Delta < n \leq k\Delta \\ k \geq \frac{n}{\Delta} \end{array} \right] \geq \frac{1}{2} \cdot \Delta \cdot \frac{n}{\Delta} \cdot \left(\frac{n}{\Delta} - 1 \right) = \frac{n^2}{2\Delta} - \frac{n}{2}$$

оценка сверху

$$\Delta + 2\Delta + 3\Delta + \dots + (k - 1)\Delta = \frac{1}{2} \cdot \Delta \cdot k \cdot (k - 1) = \left[\begin{array}{l} (k - 1)\Delta < n \leq k\Delta \\ k < \frac{n}{\Delta} + 1 \end{array} \right] \leq$$

$$\leq \frac{1}{2} \cdot \Delta \cdot \left(\frac{n}{\Delta} + 1 \right) \cdot \frac{n}{\Delta} = \frac{n^2}{2\Delta} + \frac{n}{2}$$

Расширение в $\lfloor \alpha \rfloor$ раз :

«расширяем не на Δ единиц», а «расширяем в $\lfloor \alpha \rfloor$ раз» ($\alpha > 1$)

Предположим, что $\lfloor \alpha^{k-1} \rfloor < n \leq \lfloor \alpha^k \rfloor$

Начинаем работу с массива ёмкости 1.

Оценка сверху на число «лишних операций»:

$$x - 1 < \lfloor x \rfloor \leq x \leq \lceil x \rceil < x + 1$$



$$\begin{aligned} \lfloor \alpha^0 \rfloor + \lfloor \alpha^1 \rfloor + \lfloor \alpha^2 \rfloor + \dots + \lfloor \alpha^{k-1} \rfloor &\leq \alpha^0 + \alpha^1 + \dots + \alpha^{k-1} = \frac{\alpha^k - 1}{\alpha - 1} = \left[\begin{array}{l} \text{так как } n > \lfloor \alpha^{k-1} \rfloor > \alpha^{k-1} - 1, \\ \text{то } \alpha^k < \alpha \cdot (n + 1) \end{array} \right] \leq \\ &< \frac{\alpha \cdot (n + 1) - 1}{\alpha - 1} = \frac{\alpha \cdot n + (\alpha - 1)}{\alpha - 1} = \frac{\alpha}{\alpha - 1} \cdot n + 1 \end{aligned}$$

Таким образом, можно сделать вывод, что при фиксированной константе $\alpha > 1$ общее число операций по перемещению данных в памяти, которые выполняются при последовательном добавлении n элементов, растёт линейно с ростом n .

Расширение в $[\alpha]$ раз :

«расширяем не на Δ единиц», а «расширяем в $\lfloor \alpha \rfloor$ раз» ($\alpha > 1$)

$$\frac{\alpha}{\alpha - 1} \cdot n + 1$$

α	c (capacity, ёмкость)	лишние операции
1,125	$c = c + \frac{c}{2^3}$	$9 \cdot n + 1$
1,25	$c = c + \frac{c}{2^2}$	$5 \cdot n + 1$
1,5	$c = c + \frac{c}{2^1}$	$3 \cdot n + 1$
2	$c = c + c$	$2 \cdot n + 1$
3	$c = c + 2 \cdot c$	$\frac{3}{2} \cdot n + 1$

Пример реализации динамического массива на базе статического с использованием стратегии удвоения

```
class DynamicArray:
    def __init__(self):
        self.data = array(1)
        self.size = 0
        self.capacity = 1

    def append(self, x):
        if self.size == self.capacity:
            new_capacity = self.capacity * 2
            new_data = array(new_capacity)

            for i in range(self.capacity):
                new_data[i] = self.data[i]

            self.data = new_data
            self.capacity = new_capacity

        self.data[self.size] = x
        self.size += 1
```

Конкретная операция вставки каждого элемента осуществляется:

- ✓ или за константное время, когда в массиве есть свободная ёмкость;
- ✓ или за линейное, когда свободного места нет и выполняется реаллокация.

Для получения **усреднённой оценки некоторой операции** выполняют некоторое число раз эту операцию, считают суммарное затраченное время (в худшем случае) и делят это время на число выполненных операций.

Если следовать стратегии удвоения размера, то на добавление в динамический массив k элементов требуется затратить время $O(k)$. Тогда усреднённая оценка трудоёмкости добавления одного элемента в динамический массив:

$$\frac{O(k)}{k} = O(1)$$

В этом случае говорят, что $O(1)$ — амортизированная оценка для операции вставки.

Усреднённо время вставки одного элемента в динамический массив **константное**.

Применение динамических массивов на практике

Динамические массивы очень удобны и широко используются на практике в прикладных задачах. С точки зрения скорости доступа к элементам они эквивалентны статическим массивам.

Готовые реализации динамических массивов предоставляются стандартными библиотеками всех основных современных языков программирования.

C++	Java	Python
<p>Динамический массив реализован в классе std::vector</p> <p>Значение множителя роста не зафиксировано стандартом языка и различается в зависимости от конкретной реализации:</p> <p>в libc++ — число 2</p> <p>в версии от Microsoft — число 1.5.</p>	<p>Класс ArrayList</p> <p>Множитель роста число 1.5.</p>	<p>Тип list</p> <p>В реализации CPython 3.7 при расширении действует следующая оригинальная стратегия: старый размер умножается на 1.125, затем к нему прибавляется константа 3 или 6.</p> <p>Последовательность ёмкостей: 0, 4, 8, 16, 25, 35, 46, 58, 72, 88, . . .</p>

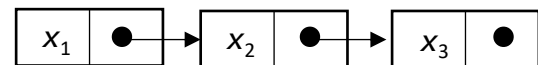
СВЯЗНЫЙ СПИСОК (англ. *Linked List*)

Связный список — некоторая последовательность элементов, которые связаны друг с другом **логически**. Логический порядок прохождения элементов определяется с помощью ссылок, при этом он может не совпадать с физическим порядком размещения элементов в памяти компьютера.

Доступ к элементам списка осуществляется **последовательно**, т. е. чем дальше в структуре расположен элемент, тем дольше к нему по времени будет осуществляться доступ.

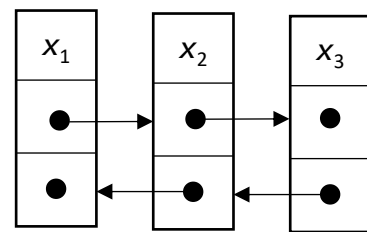
Список состоит из узлов (англ. *nodes*). Каждый узел включает две части: информационную (непосредственные данные, принадлежащие элементу) и ссылочную (указатель/ссылка на следующий и/или предыдущий узел).

В **односвязном, или однонаправленном связном, списке** (англ. *singly linked list*) каждый узел содержит ссылку на следующий узел. Для последнего узла эта ссылка обычно является нулевой.



По односвязному списку можно передвигаться только в сторону конца списка. Узнать адрес предыдущего элемента, опираясь на содержимое текущего узла, невозможно.

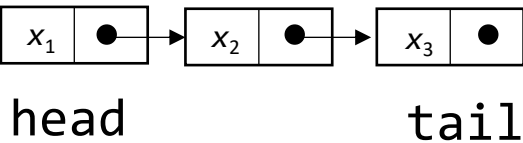
В **двусвязном, или двунаправленном связном, списке** (англ. *doubly linked list*) ссылки в каждом узле указывают на предыдущий и на последующий узел.



Как и односвязный список, двусвязный допускает только последовательный доступ к элементам, но при этом даёт возможность перемещения в обе стороны. В таком списке проще производить удаление и перестановку элементов, так как легко получить доступ ко всем элементам списка, ссылки которых направлены на изменяемый элемент.

При работе со списком вводятся дополнительные ссылки на первый и последний элемент списка. Будем называть их **head** («голова») и **tail** («хвост»).

Чаще всего узлы списка размещают в динамической памяти, при этом в качестве значений ссылок используются адреса узлов.

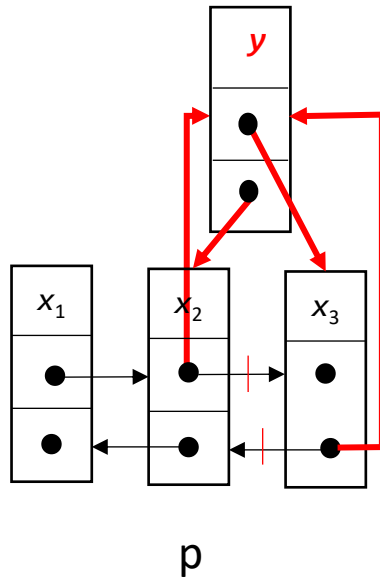
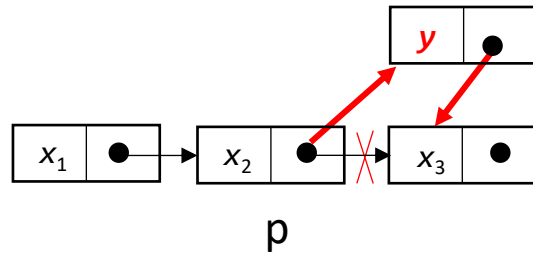


Альтернативный способ — использовать для хранения информации обычные массивы, тогда в качестве значений ссылок будут выступать индексы (порядковые номера элементов массива).

	0	1	2	3	4	
list[i]	A	E	L	N	E	head=1
next[i]	-1	2	4	0	3	tail=0

Добавление элемента

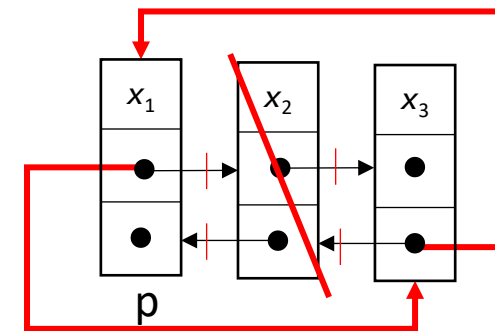
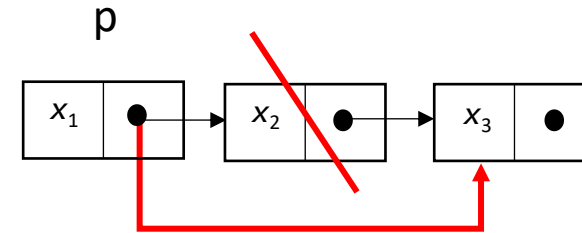
задана ссылка на элемент,
после которого выполняется
добавление



$O(1)$

Удаление элемента

задана ссылка на элемент,
который предшествует
удаляемому



1. Поиск элемента по ключу x :

$$O(n)$$

2. Добавление элемента

задана ссылка на элемент, после
которого выполняется добавление

$$O(1)$$

3. Удаление элемента

задана ссылка на элемент, который
предшествует удаляемому

$$O(1)$$

Рекомендация

В результате выполнения
базовых операций
необходимо придерживаться
правила:

**ранее вставленные
элементы никуда не
перемещаются, их
адреса в памяти не
меняются.**

Преимущества связных списков

Быстрая вставка и удаление

Операции вставки в конкретное место списка и удаления определённого элемента списка выполняются за $O(1)$ при условии, что на вход даётся ссылка на узел (идущий перед точкой вставки или предшествующий узлу, который будет удалён). Если такая ссылка не предоставлена, то операции работают за $O(n)$.

В то же время вставка в произвольное место динамического массива требует перемещения в среднем половины элементов, а в худшем случае — всех элементов. Хотя можно «удалить» элемент из массива за константное время, пометив его ячейку как «свободную», это вызовет фрагментацию, которая будет негативно влиять на скорость прохода по массиву.

Нет реаллокаций

В связный список может быть вставлено произвольное количество элементов, ограниченное только доступной памятью. Ранее вставленные элементы никуда не перемещаются, их адреса в памяти не меняются.

В динамических массивах при вставке иногда происходит реаллокация; это дорогостоящая операция, которая может оказаться невозможной при высокой фрагментированности памяти (не удастся найти непрерывный блок памяти нужного размера, хотя небольшие свободные блоки будут доступны в достаточном количестве).

Недостатки СВЯЗНЫХ СПИСКОВ

Нет произвольного доступа

Динамические массивы обеспечивают произвольный доступ к любому элементу по индексу за константное время, в то время как связанные списки допускают лишь последовательный доступ к элементам. По односвязному списку можно пройти только в одном направлении. Это делает связанные списки непригодными для алгоритмов, в которых нужно быстро получать элемент по его индексу (например, к такому типу относятся многие алгоритмы сортировки).

Медленный последовательный доступ

Линейный проход по элементам массива на реальных машинах выполняется гораздо быстрее, чем по элементам связного списка. Это связано с тем, что элементы массива хранятся в памяти один за другим, поэтому не требуется выполнять на каждом шаге переход по указателю. За счёт локальности хранения данных в массиве эффективно работает кеширование на уровне процессора.

Перерасход памяти

На хранение ссылок в узлах связного списка расходуется дополнительная память. Эта проблема особенно актуальна, если полезные данные имеют небольшой размер. Накладные расходы на хранение ссылок могут превышать размер данных в восемь или более раз.

Применение на практике связанных списков

В реальной практике прикладного программирования связанные списки в чистом виде используются крайне редко.

Динамические массивы обычно оказываются удобнее и эффективнее.

Однако есть ряд алгоритмов, при разработке которых не обойтись без классических связанных списков (например, к ним относятся многие механизмы кэширования).

Связные списки находят применение в системном программировании: в ядре операционной системы в связанных списках хранятся активные процессы, потоки и другие динамические объекты, в менеджерах памяти (аллокаторах) в связанных списках хранятся готовые к использованию блоки свободной памяти, и т. д.

В современных языках программирования двусвязный список представлен:

C++	Java	Python
Класс std::list	Класс LinkedList	<i>нет встроенной реализации</i>

Абстрактные типы данных

Абстрактные типы данных

Для абстрактного типа определяется **интерфейс** — набор операций, которые могут быть выполнены. Пользователь абстрактного типа, используя эти операции, может работать с данными, не вдаваясь во внутренние детали механизма хранения информации.

Список (*list*)

Стек (*stack*)

Очередь (*queue*)

Двухсторонняя очередь (*deque*)

Множество (*set*)

Ассоциативный массив/словарь (*associative array/ map/ dictionary*)

Список (англ. *List*)

Список –

абстрактный тип данных, представляющий собой набор элементов, которые следуют в определённом порядке.

Список является компьютерной реализацией математического понятия конечной последовательности.

Список (англ. *list*)

Базовые операции:

1. создание пустого списка;
2. проверка, является ли список пустым;
3. операцию по добавлению объекта в начало или конец списка;
4. операцию по получению ссылки на первый элемент («голову») или последний элемент («хвост») списка;
5. операцию перехода от одного элемента к следующему или предыдущему элементу;
6. операцию для доступа к элементу по заданному индексу.

Реализация интерфейса:

Абстрактный тип данных «список» обычно реализуется на практике либо как **массив** (чаще всего **динамический**), либо как **связный список** (односвязный или двусвязный).

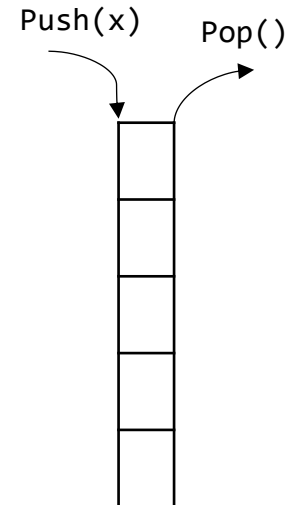
C++	Java	Python
<p>в стандартной библиотеке C++ нет специального контейнера, представляющего абстрактный список</p> <p>Следует использовать std::vector (<u>динамический массив</u>), либо std::list (<u>связный список</u>).</p>	<p>существует интерфейс List</p> <p>реализациями которого являются классы</p> <p>ArrayList (<u>динамический массив</u>)</p> <p>LinkedList (<u>связный список</u>)</p>	<p>широко используется тип данных list</p> <p>внутри являющийся <u>динамическим массивом</u></p>

Стек (англ. *stack*)

Если при добавлении и исключении элементов реализуется принцип «**последним пришёл — первым вышел**» (англ. **LIFO** — *last in first out*), то такой абстрактный тип данных называют **стеком**.

Базовые операции:

1. `Init()` — создание пустого стека;
2. `IsEmpty()` — проверка стека на пустоту; возвращается значение «истина», если стек пуст, и «ложь» в противном случае;
3. `Push(x)` — добавление элемента `x`; заданный элемент добавляется на вершину стека
4. `Pop()` — удаление элемента из стека; выполняется при условии, что стек не пуст, поэтому сначала надо убедиться в этом, а затем — извлечь с вершины стека последний занесённый в него элемент.



Реализация интерфейса:

Моделирование стека выполняется на **динамическом массиве** и на **связном списке**. Если наибольшее число элементов, которые будут одновременно находиться в стеке, заранее известно, то можно использовать и **статический массив**.

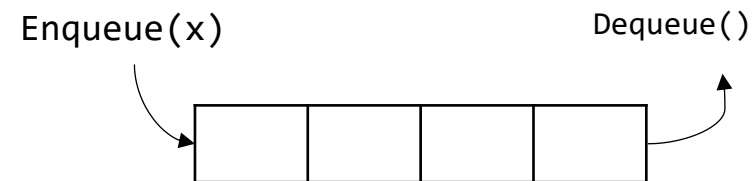
C++	Java	Python
<p>В стандартной библиотеке C++ доступен контейнер-адаптер std::stack, реализующий интерфейс стека</p> <p>по умолчанию std::stack функционирует на базе контейнера std::deque</p>	<p>класс Stack</p>	<p>нет специального класса для создания стека, предлагается использовать тип данных list (методы <code>append</code> и <code>pop</code>)</p>

Очередь (англ. *queue*)

Если при добавлении и исключении элементов реализуется принцип «**первым пришёл — первым вышел**» (англ. **FIFO** — *first in first out*), то такой абстрактный тип данных называют **очередью**.

Базовые операции:

1. `Init()` — создание пустой очереди;
2. `IsEmpty()` — проверка очереди на пустоту;
3. `Enqueue(x)` — добавление элемента `x`; заданный элемент добавляется в конец очереди
4. `Dequeue()` — удаление элемента из очереди; элемент удаляется из начала очереди; операция выполняется при условии, что очередь не пуста, поэтому сначала надо убедиться в этом, а затем — извлечь элемент.



Реализация интерфейса:

Наиболее простые способы моделирования очереди:
на **статическом массиве** (кольцевая очередь) и на **связном списке**.

C++	Java	Python
В стандартной библиотеке C++ доступен контейнер-адаптер std::queue , реализующий интерфейс очереди (по умолчанию он работает на основе контейнера std::deque).	Интерфейс Queue	Более общий контейнер типа collections.deque

Двухсторонняя очередь (англ. *double-ended queue*, или *deque*)

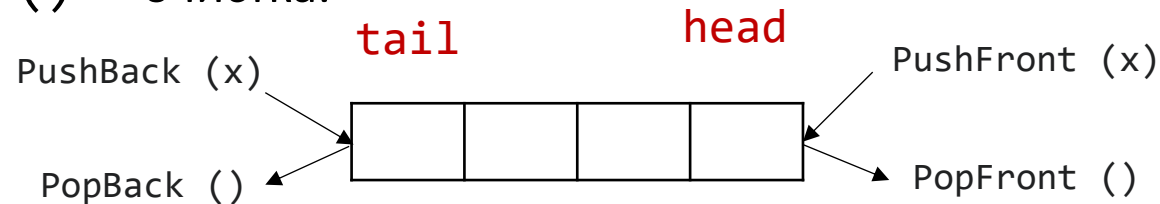
Двухсторонняя очередь —

обобщение очереди, где добавление и удаление элементов возможно с обоих концов.

Таким образом, интерфейсы стека и очереди являются частным случаем интерфейса двухсторонней очереди.

Базовые операции:

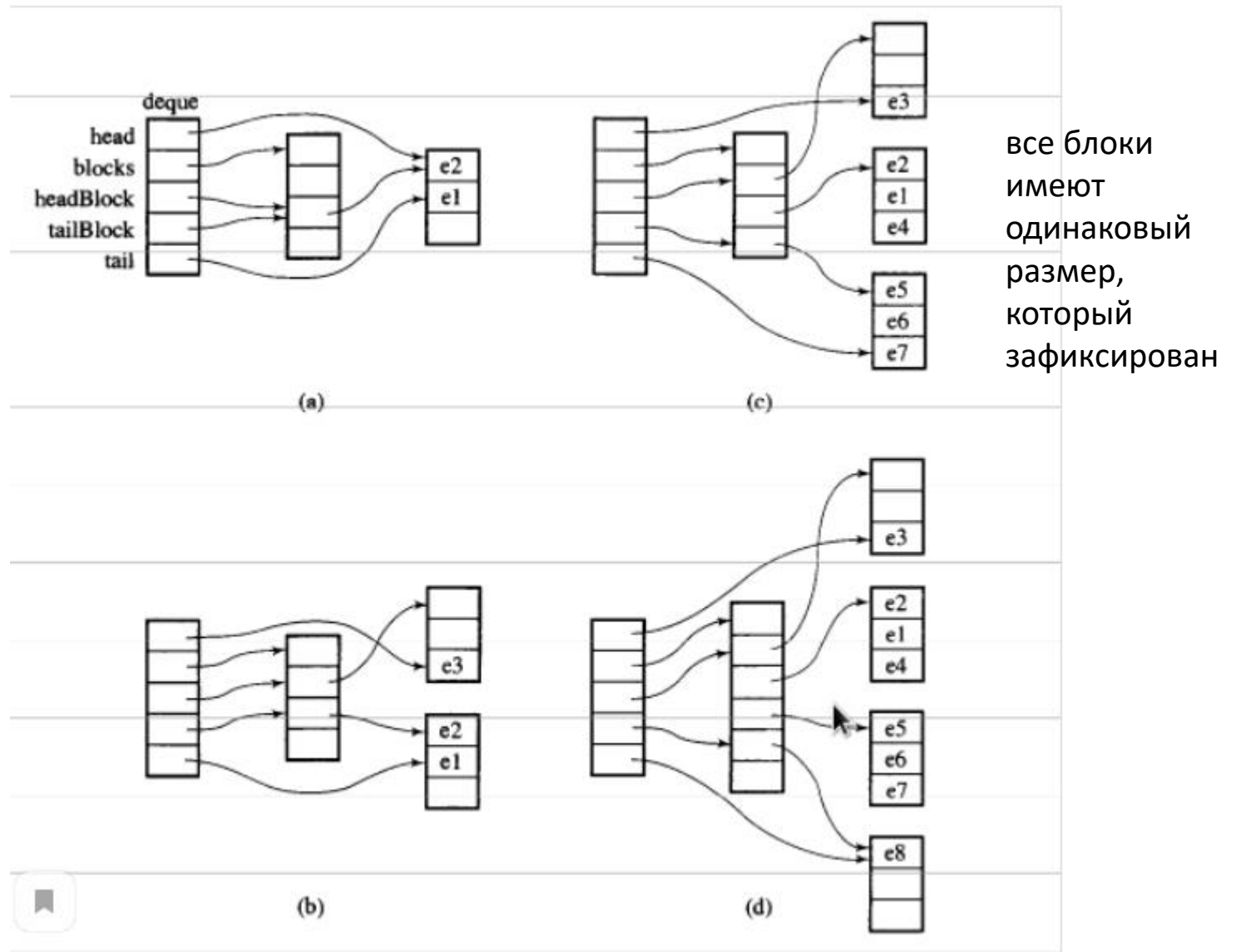
1. `PushBack (x)` — заданный элемент `x` добавляется в конец очереди;
2. `PushFront (x)` — заданный элемент `x` добавляется в начало очереди;
3. `PopBack ()` — удаление элемента из конца очереди;
4. `PopFront ()` — удаление элемента из начала очереди;
5. `IsEmpty ()` — проверка наличия элементов.
6. `Clear ()` — очистка.



C++

В стандартной библиотеке C++ роль двухсторонней очереди играет контейнер `std::deque`.

Этот контейнер обеспечивает доступ к любому элементу по индексу за $O(1)$, как и вектор, но не гарантирует, что все элементы будут лежать в памяти последовательно.



<https://coderoad.ru/6292332/Что-же-такое-на-самом-деле-дек-B-STL>

C++	Java	Python
<p>Контейнер std::deque</p> <p>контейнер обеспечивает доступ к любому элементу по индексу за $O(1)$, как и вектор, но не гарантирует, что все элементы будут лежать в памяти последовательно</p>	<p>Интерфейс Deque</p> <p>реализуется, в частности, классами ArrayDeque и LinkedList</p>	<p>В модуле collections контейнер deque</p>

Множество (англ. *set*)

Множество —

абстрактная структура данных, которая хранит набор попарно различных объектов без определённого порядка.

Базовые операции:

1. `Insert(x)` — добавить в множество ключ x ;
2. `Contains(x)` — проверить, содержится ли в множестве ключ x ;
3. `Remove(x)` — удалить ключ x из множества.

Отличия множества от списка:

1. В множестве все элементы уникальны (в списке одинаковые элементы могут храниться несколько раз).
2. В множестве порядок следования элементов не сохраняется (в списке — сохраняется).

C++	Java	Python
<p>В стандартной библиотеке C++</p> <p>контейнер std::set реализуется на основе сбалансированного поискового дерева (красно-чёрное бинарное поисковое дерево);</p> <p>контейнер unordered_set построен на базе хеш-таблицы.</p>	<p>Интерфейс Set реализуется</p> <p>класс TreeSet - сбалансированное поисковое дерево (красно-чёрное бинарное поисковое дерево)</p> <p>класс HashSet - построен на базе хеш-таблицы.</p>	<p>встроенный тип set</p> <p>нет готового класса, построенного на сбалансированных деревьях</p> <p>построен на базе хеш-таблицы.</p>

Ассоциативный массив /отображение/словарь (англ. associative array/ map/ dictionary)

Ассоциативный массив или **отображение**, или **словарь**, —

абстрактная структура данных, которая хранит пары вида (ключ, значение),
при этом каждый ключ встречается не более одного раза.

Базовые
операции:

1. `Insert(k, v)` — добавить пару, состоящую из ключа `k` и значения `v`;
2. `Find(k)` — найти значение, ассоциированное с ключом `k`, или сообщить, что значения, связанного с заданным ключом, нет;;
3. `Remove(k)` — удалить пару, ключ в которой равен `k`.

Реализация ассоциативного массива технически немного сложнее, чем множества (`set`), но использует те же идеи.

C++	Java	Python
<p>В стандартной библиотеке C++</p> <p>контейнер std::map реализуется на основе сбалансированного поискового дерева (обычно красно-чёрное бинарное поисковое дерево);</p> <p>контейнер unordered_map построен на базе хеш-таблицы.</p>	<p>Интерфейс Map реализуется</p> <p>класс TreeMap - сбалансированное поисковое дерево (красно-чёрное бинарное поисковое дерево)</p> <p>класс HashMap - построен на базе хеш-таблицы.</p>	<p>встроенный тип dict</p> <p>-</p> <p>построен на базе хеш-таблицы.</p>



Спасибо за внимание!