Линейные структуры

Массив

Когда вам нужен один объект, создаём один объект. Когда нужно несколько объектов, тогда есть несколько вариантов на выбор.

Нерационально создавать несколько отдельных однотипных объектов . Можно делать и так, пока не потребуется большое количество элементов. Тогда использовать массив.

```
int scores [50];
или
int number=50;
int *scores=new int [number];
```

Индекс массива начинается с нуля.

В простом массиве количество элементов фиксировано. Нельзя добавлять новые элементы или удалять старые. Можно менять только содержимое уже существующих ячеек.

Динамический массив

Динамический массив — это массив, который может менять свой размер. Основные языки программирования в своих стандартных библиотеках поддерживают динамические массивы. В C++ это *vector*. В Java это *ArrayList*. В C# это *List*. Все они являются динамическими массивами. В своей сути динамический массив — это простой массив, однако имеющий ещё два дополнительных блока данных. В них хранятся действительный размер простого массива и объём данных, который может на самом деле храниться в простом массиве. Динамический массив может выглядеть примерно так:

```
// Внутреннее устройство класса динамического массива
sometype *internalArray;              //динамически размещаемый буфер
unsigned int currentLength;         //действительный массив
unsigned int maxCapacity;          //количество используемых элементов
```

Добавление к динамическому массиву

При добавлении объекта к динамическому массиву происходит несколько действий. Класс массива проверяет, достаточно ли в нём места.
Если *currentLength* < *maxCapacity*, то в массиве есть место для добавления. Если места недостаточно, то размещается больший внутренний массив, и всё

копируется в новый внутренний массив. Значение maxCapacity увеличивается до нового расширенного значения. Если места достаточно, то добавляется новый элемент. Каждый элемент после точки вставки должен быть скопирован на соседнее место во внутреннем массиве, и после завершения копирования пустота заполняется новым объектом, а значение *currentLength* увеличивается на единицу.

Поскольку необходимо перемещать каждый объект после точки вставки, то наилучшим случаем будет добавление элемента к концу. При этом нужно перемещать ноль элементов (однако внутренний массив всё равно требует расширения). Динамический массив лучше всего работает при добавлении элемента в конец, а не в середину.

При добавлении объекта к динамическому массиву каждый объект может переместиться в памяти. В таких языках, как C и C++, добавление к динамическому массиву означает, что ВСЕ указатели на объекты массива становятся недействительными.

Удаление из динамического массива

Удаление объектов требует меньше работы, чем добавление. Во-первых, уничтожается сам объект. Во-вторых, каждый объект после этой точки сдвигается на один элемент. Наконец, currentLength уменьшается на единицу.

Как и при добавлении к концу массива, удаление из конца массива является наилучшим случаем, потому что при этом нужно перемещать ноль объектов. Также стоит заметить, что нам не нужно изменять размер внутреннего массива, чтобы сделать его меньше. Выделенное место может оставаться таким же, на случай, если мы позже будем добавлять объекты.

Удаление объекта из динамического массива приводит к смещению в памяти всего после удалённого элемента. В таких языках, как C и C++, удаление из динамического массива означает, что указатели на всё после удалённого массива становятся недействительными.

Но в большом динамическом массиве при частом удалении или добавлении элемента объекты могут часто копироваться в другие места, а многие указатели становиться недействительными

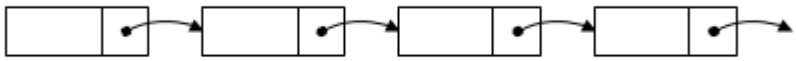
Связные списки

Массив — это непрерывный блок памяти, и каждый элемент его расположен после другого. Связанный список — это цепочка объектов. Связанные списки тоже присутствуют в стандартных библиотеках основных языков программирования. В C++ они называются *list*. В Java и C# это *LinkedList*. Связанный список состоит из

серии узлов. Каждый узел выглядит примерно так:

```
// Узел связанного списка
sometype data;
Node* next;
```

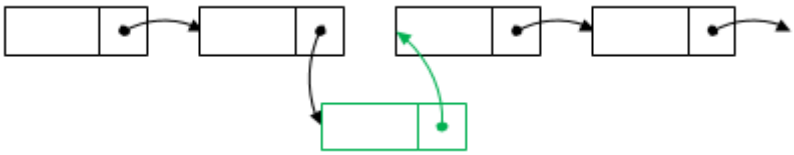
Он создаёт структуру такого типа:



Каждый узел соединяется со следующим.

Добавление к связанному списку

Добавление объекта к связанному списку начинается с создания нового узла. Данные копируются внутрь узла. Затем находится точка вставки. Указатель нового узла на следующий объект изменяется так, чтобы указывать на следующий за ним узел. Наконец, узел перед новым узлом изменяет свой указатель, чтобы указывать на новый узел.



Удаление из связанного списка

При удалении объекта из связанного списка находится узел перед удаляемым узлом. Он изменяется таким образом, чтобы указывать на следующий после удалённого объекта узел. После этого удалённый объект можно безопасно стереть.

Внесение изменений в середину списка выполняется очень быстро.
Но в связанном списке память соединена в цепочку. Если вам нужно найти пятисотый элемент, то придётся начинать с начала цепочки и следовать по её указателю
Произвольный доступ к связанному списку выполняется очень медленно.

Ещё один серьёзный недостаток связанного списка не особо очевиден. Каждому узлу необходимо небольшое дополнительное место. При использовании операторы добавления и удаления, размещается целая страница памяти, даже

если вам нужно использовать только один байт.

Стек

Стек — это структура LIFO(Last In First Out «последним вошёл, первым вышел»). При добавлении и удалении из стека последний добавленный элемент будет первым удаляемым.

Для стека нужно всего три операции: Push, Pop и Top.

Push добавляет объект в стек. Pop удаляет объект из стека. Top даёт самый последний объект в стеке. Эти контейнеры в большинстве языков являются частью стандартных библиотек. В C++ они называются *stack*. В Java и C# это *Stack*. (Да, единственная разница в названии с заглавной буквы.) Внутри стек часто реализуется как динамический массив. Поскольку стек всегда добавляет и удаляет с конца, обычно push и pop объектов в стеке выполняется невероятно быстро.

Очередь

Очередь — это структура FIFO (First In First Out, «первым зашёл, первым вышел»).

При добавлении и удалении из очереди первый добавляемый элемент будет первым извлекаемым. Очереди нужно только несколько операций: Push_Back, Pop_Front, Front и Back. Push_Back добавляет элемент к концу очереди. Pop_Front удаляет элемент из начала очереди. Front и Back позволяют получить доступ к двум концам очереди.

В двухсторонней очереди можно удалять и добавлять элементы с обоих концов (double ended queue, deque). В этом случае добавляется ещё пара операций: Push_Front и Pop_Back. Эти контейнеры тоже включены в большинство основных языков. В C++ это *queue* и *deque*.
.

Очередь с приоритетом

Это очень распространённая вариация очереди. Очередь с приоритетом очень похожа на обычную очередь.

Программа добавляет элементы с конца и извлекает элементы из начала. Разница в том, что можно задавать приоритеты определённым элементам очереди. Все самые важные элементы обрабатываются в порядке FIFO. Потом в порядке FIFO обрабатываются элементы с более низким приоритетом. И так повторяется, пока не будут обработаны в порядке FIFO элементы с самым низким приоритетом.

При добавлении нового элемента с более высоким приоритетом, чем остальная часть очереди, он сразу же перемещается в начало очереди. В C++ эта структура называется *priority_queue*. В стандартной библиотеке C# очереди с приоритетом нет. Очереди с приоритетом полезны не только для того, чтобы встать первым на очереди к принтеру организации. Их можно использовать для удобной реализации алгоритмов, например, процедуры поиска A*. Наболее вероятным результатам можно отдать более высокий приоритет, менее вероятным — более низкий. Можно создать собственную систему для сортировки и упорядочивания поиска A*, но намного проще использовать встроенную очередь с приоритетом.

Стеки, очереди, двухсторонние очереди и очереди с приоритетом можно реализовать на основе других структур данных. Они очень эффективны, когда нужно работать только с конечными элементами данных, а серединные элементы не важны.