Понятия алгоритма и структуры данных

Алгоритм - это точное предписание, определяющее вычислительный процесс, ведущий от варьируемых начальных данных к искомому результату.

ЭВМ в настоящее время приходится не только считывать и выполнять определенные алгоритмы, но и хранить значительные объемы информации, к которой нужно быстро обращаться.

Эта информация в некотором смысле представляет собой абстракцию того или иного фрагмента реального мира и состоит из определенного множества данных, относящихся к какой-либо проблеме.

Независимо от содержания и сложности любые данные в памяти ЭВМ представляются последовательностью двоичных разрядов, или битов, а их значениями являются соответствующие двоичны

Данные, рассматриваемые в виде последовательности битов, имеют очень простую организацию или, другими словами, слабо структурированы.

Для человека описывать и исследовать сколько-нибудь сложные данные в терминах последовательностей битов весьма неудобно.

Более крупные и содержательные чем бит, «строительные блоки» для организации произвольных данных получаются на основе

***Структура данных - это* множество элементов данных и множество связей между ними.**

Физическая структура данных (структура хранения, внутренняя структура или структура памяти) –это способ физического представления данных в памяти машины.

Логическая (абстрактная**) структура** – это структура данных без учета ее представления в машинной памяти.

Между логической и соответствующей ей физической структурами существует различие, степень которого зависит от самой структуры и особенностей той среды, в которой она должна быть отражена.

Вследствие этого различия существуют процедуры, осуществляющие отображение логической структуры в физическую и, наоборот - физической структуры в логическую.

Эти процедуры обеспечивают, кроме того, доступ к физическим структурам и выполнение над ними различных операций, причем каждая операция рассматриваема применительно к логической или физической структуре данных.

В зависимости от размещения физических структур, а соответственно, и доступа к ним, различают **внутренние** (находящиеся в оперативной памяти) и **внешние** (на внешних устройствах) **структуры данных**.

Внутренние структуры данных рассматривают как:

- **элементарные** (или простые, или базовые, или примитивные) структуры данных ; это такие структуры данных, которые **не могут быть разделенены на составные части, большие, чем биты**. С точки зрения физической структуры важно то, что в конкретной машинной архитектуре, в конкретной системе программирования всегда можно заранее сказать, каков будет размер элементарного данного и каково его размещение в памяти. С логической точки зрения элементарные данные являются неделимыми единицами.
- **составные** (или интегрированные, композитные, сложные); это такие структуры данных, **составными частями которых являются другие структуры данных** - элементарные или составные. Составные структуры данных конструируются программистом с использованием средств интеграции данных, предоставляемых языками программирования.

Важный признак составной структуры данных - характер упорядоченности ее частей. По этому признаку структуры можно делить на **линейные** и **нелинейные** структуры.

Весьма важный **признак структуры данных** – ее **изменчивость**, т. е. **изменение числа элементов и/или связей между составными частями структуры.** В определении изменчивости структуры не отражен факт изменения значений элементов данных, поскольку в этом случае все структуры данных имели бы свойство изменчивости.

По признаку изменчивости различают структуры **статические** и **динамические**.

Структуры данных

ВНУТРЕННИЕ (в оперативной памяти)

ВНЕШНИЕ (на внешних устройствах)

Элементарные

Файл

Булевый

База д-х

Числовой

.....

Символьный

Указатель

.....

Составные

Линейные

Массив

Запись

Множество

Таблица

Линейный список

Стек

Нелинейные

Слоеный список

Мультисписок

Дерево

Граф

.....

В языках программирования понятие «структуры данных» тесно связано с понятием «типы данных». Любые данные, т. е. константы, переменные, значения функции или выражения, характеризуются своими типами.

Информация но каждому типу однозначно определяет:

- структуру хранения данных указанного типа, т.е. выделение памяти, представление данных в ней и метод доступа к данным;
- множество допустимых значений, которые может иметь тот или иной объект описываемого типа;
- набор допустимых операций, которые применимы к обьекту описываемого типа.

Анализ сложности и эффективности алгоритмов и структур данных

В процессе решения прикладных задач выбор подходящего алгоритма вызывает определенные трудности.

Алгоритм должен удовлетворять следующим противоречащим друг другу требованиям:

- быть простым для понимания, перевода в программный код и отладки;
- эффективно использовать вычислительные ресурсы и выполняться по возможности быстро.

Если разрабатываемая программа, реализующая некоторый алгоритм, должна выполняться только несколько раз, то первое требование наиболее важно. В этом случае стоимость программы оптимизируется по стоимости написания (а не по стоимости выполнения) программы.

Если решение задачи требует значительных вычислительных затрат, то стоимость выполнения программы может превысить стоимость написания программы, особенно если программа выполняется многократно.

Более предпочтительным может стать сложный комплексный алгоритм (в надежде, что результирующая программа будет выполняться существенно быстрее).

Прежде чем принимать решение об использовании того или иного алгоритма, необходимо оценить сложность и эффективность этого алгоритма.

Сложность алгоритма – это величина, отражающая порядок величины требуемого ресурса (времени или дополнительной памяти) в зависимости от размерности задачи.

Для рассмотрения оценок сложности алгоритма обычно используют:

- Временну’ю сложность алгоритма **T(n)**
- Простра’нственную сложность алгоритма **V(n)**

Самый простой способ оценки - **экспериментальный**, т. е. запрограммировать алгоритм и выполнить полученную программу на нескольких задачах, оценивая время выполнения программы.

Однако, этот способ имеет ряд недостатков:

- экспериментальное программирование –это, возможно, дорогостоящий процесс;
- на время выполнения программы влияют следующие факторы:
  - временная сложность алгоритма программы;
  - качество скомпилированного кода исполняемой программы;
  - машинные инструкции, используемые для выполнения программы.

Временна’я сложность алгоритма и качество скомпилированного кода не позволяют применять типовые единицы измерения временной сложности алгоритма (секунды, миллисекунды и т. д.), т. к. можно получить самые различные оценки для одного и того же алгоритма, если использовать

- ✔ разных программистов (которые программируют алгоритм каждый по-своему),
- ✔ разные компиляторы и
- ✔ разные вычислительные машины.

Рассмотрим **теоретический метод** оценки сложности алгоритма.

Часто, временна’я сложность алгоритма зависит от количества входных данных. Обычно говорят, что временна’я сложность алгоритма имеет порядок **T(n)** от входных данных размера **n**.

Точно определить величину **T(n)** на практике представляется довольно трудно. Поэтому прибегают к асимптотическим[1)] отношениям с использованием **O-символики.**

[1)] *Математическое поведение функции в особых точках, чаще всего при стремлении аргумента или функции к бесконечности.*

Если число тактов (действий), необходимое для работы алгоритма, выражается как

$$11*n^2 + 19*n*\log(n) + 3*n + 4,$$

то это алгоритм, для которого **T(n)** имеет порядок $O(n^2)$.

n^2 Из всех слагаемых оставляется только то, что вносит наибольший вклад при больших **n** (в этом случае остальными слагаемыми можно пренебречь), и игнорируется коэффициент перед ним.

Когда используют обозначение **O()**, имеют в виду не точное время исполнения, а только его предел сверху, причем с точностью до постоянного множителя.

Когда говорят, например, что алгоритму требуется время порядка **$O(n^2)$,** имеют в виду, что время исполнения задачи растет не быстрее, чем **квадрат количества элементов**.

Пример чисел, иллюстрирующих скорость роста для нескольких функций, которые часто используются при оценке временно’й сложности алгоритмов

n	$\log n$	$n * \log n$	n^2
1	0	0	1
16	4	64	256
256	8	2 048	65 536
4 096	12	49 152	16 777 216
65 536	16	1 048 565	4 294 967 296
1 048 476	20	20 969 520	1 099 301 922 576
16 775 616	24	402 614 784	281 421 292 179 456

Если считать, что числа соответствуют микросекундам, то для задачи с **1048476** количеством элементов

- алгоритму со временем работы **T(log n)** потребуется **20** микросекунд, а
- алгоритму со временем работы **$T(n^2)$** - более 12 дней.

Если операция выполняется за *фиксированное число шагов*, не зависящее от количества данных, то принято писать **O** (1).

Время выполнения алгоритма зависит не только от количества входных данных, но и от их значений, например, время работы некоторых алгоритмов сортировки значительно сокращается, если первоначально данные частично упорядочены, тогда как другие методы оказываются нечувствительными к этому свойству.

Чтобы учитывать этот факт, полностью сохраняя при этом возможность анализировать алгоритмы независимо от данных, различают:

- **максимальную сложность** $T_{max}(n)$, или сложность наиболее неблагоприятного случая, когда алгоритм работает дольше всего;
- **среднюю сложность** $T_{mid}(n)$ - сложность алгоритма в среднем;
- **минимальную сложность** $T_{min}(n)$ – сложность в наиболее благоприятном случае, когда алгоритм справляется быстрее всего.

Теоретическая оценка, временно’й сложности алгоритма осуществляется с использованием следующих **базовых принципов**:

1. Время выполнения операций присваивания, чтения, записи обычно имеют порядок **0(1)**. Исключением являются операторы присваивания, в которых операнды представляют собой массивы или вызовы функций;
2. Время выполнения последовательности операции совпадает с наибольшим временем выполнения операции в данной последовательности (правило сумм: если $T_1(n)$ имеет порядок $0(f(n))$, а $T_2(n)$ - порядок $0(g(n))$, то $T_1(n) + T_2(n)$ имеет порядок

$$T_1(n) + T_2(n) \quad 0(\max(f(n), g(n)));$$

3. Время выполнения конструкции ветвления (if-then-else) состоит из времени вычисления логического выражения (обычно имеет порядок O(1) и наибольшего из времени, необходимого для выполнения операций, исполняемых при истинном значении логического выражения и при ложном значении логического выражения;

4. Время выполнения цикла состоит из времени вычисления условия прекращения цикла (обычно имеет порядок O(1) и произведения количества выполненных итераций цикла на наибольшее возможное время выполнения операций тела цикла.

5. Время выполнения операции вызова процедур определяется как время выполнения вызываемой процедуры;

6. При наличии в алгоритме операции безусловного перехода, необходимо учитывать изменения последовательности операции, осуществляемых с использованием этих операций безусловного перехода.

Структуры данных

Элементарные данные

Данные элементарных типов представляют собой единое и неделимое целое. В каждый момент времени они могут принимать только одно значение. Набор элементарных типов в разных языках программирования несколько различаются, однако есть типы, которые поддерживаются практически везде.

```
int  i,j;          //целочисленного типа
float x;           //вещественного типа
char s;            //символьного типа
bool b;            //логического  типа
int *p;            //типизированный
   указатель
void *p1;          //безтиповый указатель
```

Данные целочисленного типа

С помощью целых чисел может быть представлено количество объектов, являющихся дискретными по своей природе (т. е. счетное число объектов).

Диапазон возможных значений целых типов зависит от их внутреннего представления, которое может занимать 1,2 или 4 байта.

Тип	Диапазон значений	Машинное представление
short int	-128...127	8 бит со знаком
int	-32768... .32767	16/32 бит со знаком
longi nt	-214748 3648. ...2147483 647	32 бита со знаком
char	0...255	8 бит без знака
unsigned int	0... 655 35	16 бит без знака

Данные вещественного типа

Значение вещественных типов определяет число с некоторой конечной точностью, зависящей от внутреннего формата вещественного числа.

Тип	Диапазон значений	Значащие цифры	Размер в байтах
real	$2.9*10^{-39}$... $1.7*10^{38}$	11 - 12	6
double	$4.9*10^{-324}$…$1.8*10^{308}$	15 - 16	8

Операции над данными числовых типов

Над числовыми типами, как и над всеми другими возможны прежде всего, четыре основных операции:

создание,
уничтожение,
выбор,
обновление.

Специфическими операциями над числовыми типами являются арифметические операции:

сложение,
вычитание,
умножение,
деление.

Операция возведения в степень в некоторых языках также является базовой и обозначается специальным символом (в Паскале это символ ^) или комбинацией символов, в других - выполняется встроенными функциями.

Операция деления по-разному выполняется для целых и вещественных чисел. При делении целых чисел дробная часть результата отбрасывается, как бы близка к 1 она ни была.

Для целых операндов возможна еще одна операция - остаток от деления (в языке Паскаль операция mod, а в C++ - %).

Операции сравнения >, <, <=, >=, <>/!=

Существенно, что хотя операндами этих операций являются данные числовых чипов, результат их имеет логический тип - «истина» или «ложь».

Говоря об операциях сравнения, следует обратить внимание на особенность выполнения сравнений на равенство или неравенство вещественных чисел. Поскольку эти числа представляются в памяти с некоторой (не абсолютной) точностью, сравнения их не всегда могут быть абсолютно достоверны.

Поскольку одни и те же операции допустимы для разных числовых типов, возникает проблема арифметических выражений со смешением типов. В реальных задачах выражения со смешанными типами встречаются довольно часто. Поэтому большинство языков допускает выражения, операнды которых имеют разные числовые типы, но обрабатываются такие выражения в разных языках по-разному.

В одних языках все операнды выражения приводятся к одному типу, а именно к типу той переменной, в которую будет записан результат, а затем уже выражение вычисляется.

В других (например, язык Си) преобразование типов выполняется в процессе вычисления выражения, при выполнении каждой отдельной операции, без учета других операций; каждая операция вычисляется с точностью самого точного участвующего в ней операнда.

Данные символьного типа

Значением символьного типа **char** являются символы из некоторого предопределенного множества. В качестве примеров этих множеств можно назвать **ASCII** (American Standard Code for Information Interchange). Это множество состоит из 256 разных символов, упорядоченных определенным образом, и содержит символы заглавных и строчных букв, цифр и других символов, включая специальные управляющие символы.

Значение символьного типа **char** занимает в памяти 1 байт. Код от 0 до 255 в этом байте задает один из 256 возможных символов **ASCII** таблицы.

ASCII включает в себя буквенные символы только латинского алфавита**.**

Символы национальных алфавитов занимают «свободные места» в таблице кодов и, таким образом, одна таблица может поддерживать только один национальный алфавит.

Этот недостаток преодолен во множестве **UNICODE**. В этом множестве каждый символ кодируется двумя байтами, что обеспечивает более 2^{16} возможных кодовых комбинаций и дает возможность иметь единую таблицу кодов, включающую в себя все национальные алфавиты**. UNICODE** является перспективным, однако, повсеместный переход к двухбайтным кодам символов может вызвать необходимость переделки значительной части существующего программного обеспечения.

Специфические операции над символьными типами - только операции сравнения. При сравнении коды символов рассматриваются как целые числа без знака. Кодовые таблицы строятся так, что результаты сравнения подчиняются лексикографическим правилам: **символы, занимающие в алфавите места с меньшими номерами, имеют меньшие коды, чем символы, занимающие места с большими номерами.**

Данные логического типа

Значениями логического типа может быть одна из предварительно объявленных констант false (ложь) или true (истина).

Данные логического типа занимают один байт памяти. При этом значению false соответствует нулевое значение байта, а значению true - любое ненулевое значение байта.

Над логическими типами возможны операции булевой алгебры – НЕ (not), ИЛИ (or), И (and), ИСКЛЮЧАЮЩЕЕ ИЛИ (xor).

Последняя операция реализована для логическою типа не во всех языках.

Кроме того, следует помнить, что результаты логического типа получаются при сравнении данных любых типов.

Данные типа указатель

Тип указателя представляет собой адрес ячейки памяти. Физическое представление адреса существенно зависит от аппаратной архитектуры вычислительной системы.

В языках программирования высокою уровня определена специальная константа nil, которая означает пустой указатель или указатель, не содержащий какой-либо конкретный адрес.

При решении прикладных задач с использованием языков высокого уровня наиболее частые случаи, когда могут понадобиться указатели, следующие:

1) при необходимости представить одну и ту же область памяти, а следовательно, одни и те же физические данные как данные разной логической структуры. В этом случае вводятся два или более указателей, которые содержат адрес одной и той же области памяти, но имеют разный тип. Обращаясь к этой области памяти по тому или иному указателю, можно обрабатывать ее содержимое как данные -того или иного типа;

2) при работе с динамическими структурами данных. Память под такие структуры выделяется в ходе выполнения программы, стандартные функции выделения памяти возвращают адрес выделенной области памяти - указатель на неё. К содержимому динамически выделенной области памяти можно обращаться только через такой указатель.

```
int     *a_ptr;
float   * b_ptr;
void    *ptr;
```

В языках высокого уровня указатели могут быть типизированными и нетипизированными.

Основными операциями, в которых участвуют указатели, являются **присваивание, получение адреса, выборка.**

Присваивание является двухместной операцией, оба операнда которой - указатели. Как и для других типов, операция присваивания копирует значение одного указателя в другой, в результате оба указателя будут содержать один и тот же адрес памяти. Если оба указателя, участвующие в операции присваивания типизированные, то оба они должны указывать на данные одного и того же типа.

Операция **получения адреса** – одноместная, ее операнд может иметь любой тип. Результатом является типизированный (в соответствии с типом операнда) указатель, содержащий адрес операнда.

Операция **выборки** - одноместная, ее операндом является типизированный указатель. Результатом этой операции являются данные, выбранные из памяти по адресу, заданному операндом. Тип результата определяется типом указателя.

Системное программирование требует более гибкой работы с адресами, поэтому, например, в языке Си доступны также операции адресной арифметики.

Линейные структуры данных

Рассмотрим *статические структуры данных:* массивы и структуры.

Цель описания типа данных и определения некоторых переменных, относящихся к статическим типам - зафиксировать диапазон значений, присваиваемых этим переменным, и соответственно размер выделяемой для них памяти.

Такие переменные называются **статическими**.

Массив

Массив - это поименованная совокупность однотипных элементов, упорядоченных по индексам, определяющих положение элемента в массиве.

Следующее объявление задает имя для массива, количество элементов и тип элементов массива:

Тип массива **Имя** массива [**Количество** элементов];

Тип индекса, в общем случае, может быть любым порядковым, но некоторые языки программирования поддерживают в качестве индексов массивов только последовательности целых чисел.

Количество используемых индексов определяет размерность массива. Массив может быть одномерным (вектор), двумерным (матрица), - трехмерным (куб) и т. д.:

```
int  Vector [100];
int  Matrix[100] [100];
int Cube[100] [100] [100];
```

Можно выполнять операции над отдельными элементами массива. Перечень таких операций определяется типом элементов. Доступ к отдельным элементам массива осуществляется через имя массива и индекс (индексы) элемента;

```
Cube[0,0,10]   = 25;
Matrix[1][0][3]= Cube[1][0][3]  + 1;
```

В **памяти** ЭВМ элементы массива обычно раеполагаются непрерывно, в **соседних ячейках.** Размер памяти, занимаемой массивом, есть суммарный размер элементов массива.

Строка

Строка - это последовательность символов (элементов символьного типа).

```
char TTxt [255];
char stroka [80];
char Name [25];
```

Каждый символ строки имеет свой индекс, принимающий значение от 1 до заданной длины строки.

Благодаря индексам, строки очень похожи на одномерные массивы символов, и доступ к отдельным элементам строки можно получать с использованием этих индексов, выполняя операции, определенные для символьного типа данных.

В памяти ЭВМ символы строки располагаются непрерывно, в соседних ячейках. Размер памяти, занимаемой строкой, есть суммарный размер элементов массива.

Структуры

Структура - это **агрегат,** составляющие которого (поля) **имеют** имя и могут быть различного типа.

Рассмотрим пример простейшей структуры:

```
Struct TPerson
{    char Name[15];
     char Address[30];
     long int Index;
}
Tperson : Person1;
```

Структура объединяет 3 поля: первые два из них символьного типа, а третье - целочисленного.

Можно также выполнять операции над отдельным полем записи. Перечень таких операций определяется типом поля.

Доступ к полям отдельной записи осуществляется через имя записи и имя поля:

Person1.Index = 190 000;

Person1.Name = ' Иванов' ;

Person1.Adress = ' Санкт-Петербург, ул. Б. Морская, д.67';

В памяти ЭВМ поля записи обычно располагаются непрерывно, в соседних ячейках. Размер памяти, занимаемой объектом, есть суммарный размер полей, составляющих структуру.

Линейные списки

Список - это структура данных, представляющая собой логически связанную последовательность элементов списка.

Иногда бывают ситуации, когда невозможно на этапе разработки алгоритма определить диапазон значений переменной. В этом случае применяют динамические структуры данных.

Динамическая структура данных-это **структура данных, определяющие характеристики которой могут изменяться на протяжении ее существования.**

Обеспечиваемая такими структурами способность к адаптации часто достигается меньшей эффективностью доступа к их элементам.

Динамические структуры данных отличаются от статических двумя основными свойствами:

- в них нельзя обеспечить хранение в заголовке всей информации о структуре, поскольку каждый элемент должен содержать информацию, логически связывающую его с другими элементами структуры;
- для них зачастую неудобно использовать единый массив смежных элементов памяти, поэтому необходимо предусматривать ту или иную схему динамического управления памятью.

Для обращения к динамическим данным применяют указатели.

Созданием динамических данных должна заниматься сама программа во время своего исполнения. В языке программирования «С» для этого существует специальная функция new:

<Tüüp> *p;

P=new <Tüüp>;

После выполнения данной функции в оперативной памяти ЭВМ создается динамическая переменная, тип которой определяемся типом указателя **p**.

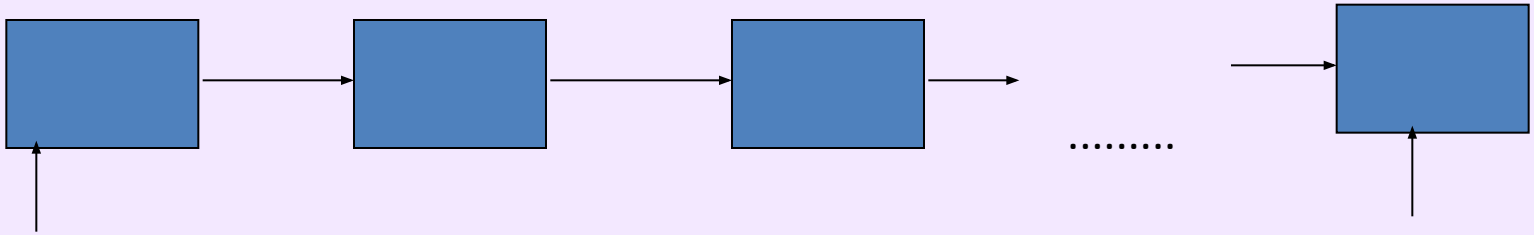
После использования дипамического данного и при отсутствии необходимости его дальнейшего использования необходимо освободить оперативную память ЭВМ от этого данного с помощью соответствующей функции:

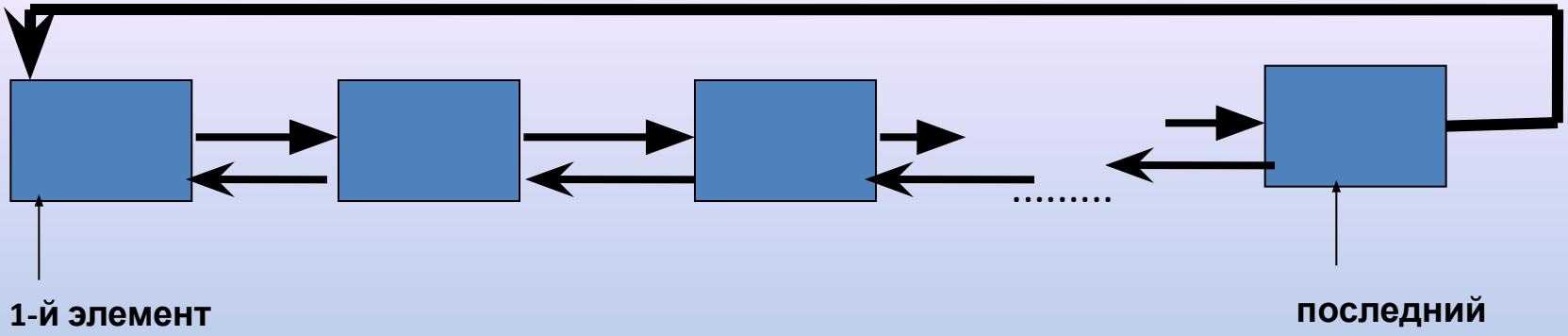
```
delete p;
```

Наиболее простой способ организовать структуру данных, состоящую из некоторого множества элементов - это организовать линейный список.

При такой организации элементы некоторою типа образуют цепочку.

Для связывания элементов в списке используют систему указателей, и в зависимости от их количества в элементах различают однонаправленные и двунаправленные линейные списки.


.....
1-й элемент
последний


.....
1-й элемент
последний

Tänan tähelepanu eest!