1. **ТИПЫ ДАННЫХ**

**1.1 Типы данных, абстрактные типы и структуры данных**

В языках программирования тип данных переменной обозначает множество значений, которые может принимать эта переменная.

Типы данных включают:

**-натуральные и целые числа,**

**-вещественные (действительные) числа (в виде приближенных десятичных дробей),**

**-литеры,**

**-строки и др.**

В некоторых языках программирования **тип каждой константы или переменной определяется компилятором** по записи присваиваемого значения.

*В зависимости от назначения языка программирования защита типов, осуществляемая на этапе компиляции, может быть более или менее жесткой.*

Так, например,

- *язык Pascal*, изначально созданный как инструмент для иллюстрирования структур данных и алгоритмов, *сохраняет весьма строгую защиту типов.* Pascal-компилятор в большинстве случаев *расценивает смешение в одном выражении данных разных типов или применение к типу данных несвойственных ему операций как фатальную ошибку*.

- Напротив, *язык C,* предназначенный, прежде всего, для системного программирования, *является языком со слабой защитой типов и в случае смешения типов данных* *C-компиляторы выдают предупреждения.* **Отсутствие жесткой защиты типов дает системному программисту дополнительные возможности, но при этом человек сам отвечает за правильность своих действий.**

**Абстрактный тип данных (АТД)** – это математическая модель и набор операторов, определенных в рамках данной модели.

**Базовым строительным блоком структуры данных является ячейка, которая предназначена для хранения значения определенного базового или составного типа данных.**

**1. 2 Классификация структур данных**

*Понятие физическая структура данных отражает способ физического представления данных в памяти машины и называется еще структурой хранения, внутренней структурой или структурой памяти.*

*Структура данных без учета ее представления в машинной памяти называется абстрактной или логической структурой*. В общем случае между логической и соответствующей ей физической структурами существует различие. Известны процедуры, выполняющие отображение логической структуры в физическую и, наоборот. Кроме того, эти процедуры обеспечивают доступ к физическим структурам и выполнение над ними различных операций.

Различают:

**- простые (базовые, примитивные) структуры (типы) данных,**

**- интегрированные (структурированные, композитные, сложные).**

*Интегрированными называются такие структуры данных, составными частями которых являются другие структуры данных: простые или интегрированные. Интегрированные структуры данных конструируются программистом с использованием средств интеграции данных, предоставляемых языками программирования.*

**В зависимости от отсутствия или наличия явно заданных связей** между элементами данных следует различать:

- **несвязные структуры** (векторы, массивы, строки, стеки, очереди)

- **связные структуры** (связные списки).

**По признаку изменчивости** различают структуры:

**- статические,**

**- полустатические,**

**- динамические.**

**В зависимости от упорядоченности элементов** структуры данных делят:

**- на линейные,**

**- нелинейные.**

**По признаку взаимного расположения элементов в памяти** линейные структуры можно разделить:

**- на структуры с последовательным распределением элементов в памяти (векторы, строки, массивы, стеки, очереди)**

**- структуры с произвольным связным распределением элементов в памяти (односвязные, двусвязные списки).**

Информация по каждому типу однозначно определяет:

- структуру хранения данных указанного типа, т.е. *выделение памяти и представление данных в ней, с одной стороны, и интерпретирование двоичного представления, с другой;*

*- множество допустимых значений, которые может иметь тот или иной объект описываемого типа;*

*- множество допустимых операций, которые применимы к объекту описываемого типа.*

**Статические структуры относятся к разряду непримитивных структур, которые, фактически, представляют собой структурированное множество примитивных, базовых, структур.** *Например, вектор может быть представлен упорядоченным множеством чисел.*

*Поскольку по определению* ***статические структуры отличаются отсутствием изменчивости,******память для них выделяется автоматически,*** *как правило, на этапе компиляции или при выполнении в момент активизации того программного блока, в котором они описаны.*

*Базовые структуры данных, статические, полустатические и динамические характерны для оперативной памяти и часто называются оперативными структурами.* *Файловые структуры соответствуют структурам данных для внешней памяти. На рис. 1 приведена классификация структур данных.*



Рис. 1 – Классификация структур данных

**1.3 Виды памяти, указатели и работа с ними в языке Pascal (Delphi)**

Различают два способа распределения памяти:

**- статическое – во время трансляции программы, что эффективно, поскольку в ходе выполнения программы на управление памятью не расходуется ни время, ни память;**

**– динамическое управление памятью, которое осуществляется во время выполнения программы, а переменные, которые создаются и уничтожаются в этом процессе, называются динамическими. Для организации динамической памяти используется тип данных, называемый указателем или ссылочным типом данных.**

*Динамическая память широко используется при работе с графическими и звуковыми средствами компьютера.*

*Значением указателя является адрес области памяти, содержащей переменную заранее определенного типа. В этом случае указатели называются типизированными.*

**Нетипизированный указатель** служит для представления адреса, по которому содержатся данные неизвестного типа.

**Для указателей область памяти выделяется статически, а для переменных, на которые они указывают, – динамически***. Адреса задаются совокупностью двух 16-разрядных слов, которые называются сегментом и смещением. Сегмент – это участок памяти, имеющий длину 64 Кбайт и начинающийся с адреса, кратного 16. Смещение указывает, сколько байт от начала сегмента необходимо пропустить, чтобы обратиться к нужному адресу. В результате, абсолютный адрес образуется: сегмент\*16+смещение.*

*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 в программе вводится несколько указателей, которые содержат адрес одной и той же области памяти, но имеют разные типы. При обращении к этой области памяти по определенному указателю ее содержимое обрабатывается как данные того или иного типа.*

**2) При работе с динамическими структурами данных.** *Память под них выделяется в ходе выполнения программы, стандартные процедуры/функции выделения памяти возвращают адрес выделенной области памяти, т.е. указатель на нее. К содержимому динамически выделенной области памяти можно обращаться только через такой указатель.*

*При объявлении типизированного указателя определяется и тип объекта в памяти, адресуемого этим указателем. Для объявления переменных используется символ « ^ », после которого указывается тип динамической (базовой) переменной.*

**Type <имя\_типа>=^ <базовый тип>;**

**Var <имя\_переменной>: <имя\_типа>; или**

**<имя\_переменной>: ^ <базовый тип>:**

**Например:**

**Type ss = ^ Integer;**

**Var x, yt: ss; {\*Указатели на переменные целого типа \*}**

**z : ^Real; {\*Указатель на переменную вещественного типа \*}**

Здесь переменные***:***

***x*** *и* ***y*** *представляют собой адреса областей памяти, в которых хранятся целые число*,

**z** – *адрес области памяти, в которой хранится вещественное число.* *Хотя физическая структура адреса не зависит от типа и значения данных, хранящихся по этому адресу, компилятор считает указатели x, y и z, имеющими разный тип, и в Pascal оператор:* ***x:= z;*** *будет расценен компилятором как ошибочный.*

**Выделение оперативной памяти для динамической переменной базового типа осуществляется с помощью процедуры New(x), где x определен как соответствующий указатель.**

*Обращение к динамическим переменным выполняется по правилу:* **<имя переменной> ^.**

Например, x^:=15. Здесь в область памяти (два байта), адрес которой является значением указателя x, записывается число 15.

Процедура Dispose(x) освобождает память, занятую динамической переменной. При этом значение указателя x становится неопределенным.

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

**Присваивание** – это двухместная операция, оба операнда которой – указатели. Как и для других типов, операция присваивания копирует значение одного указателя в другой, в результате чего оба указателя будут содержать один и тот же адрес памяти. Типизированные указатели должны ссылаться на объекты 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 объекта-операнда.

**Операция выборки** – одноместная, ее операндом является типизированный указатель, результат – данные, выбранные из памяти по адресу, заданному операндом. Тип результата определяется типом указателя-операнда.

*Перечисленных операций, как правило, достаточно для решения задач прикладного программирования, поэтому набор операций над указателями, допустимых в языке Pascal, этим и ограничивается.*

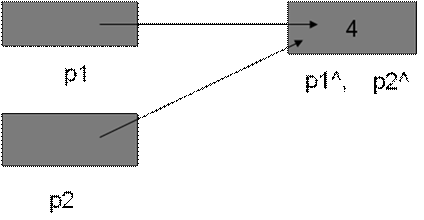
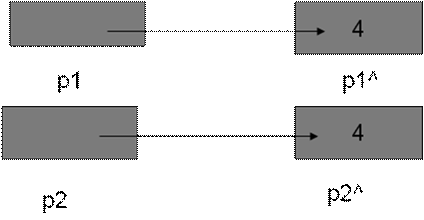
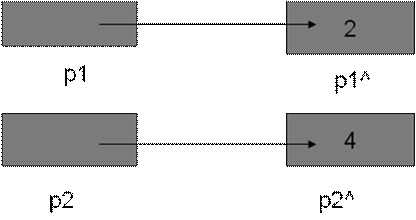
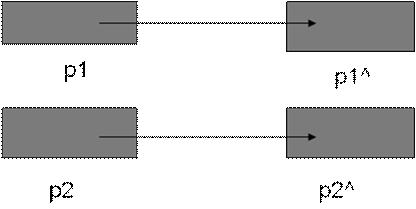
*К указателю можно прибавить целое число или вычесть из него целое число. Поскольку память имеет линейную структуру, прибавление к адресу числа даст адрес области памяти, смещенной на это число байт (или других единиц измерения) относительно исходного адреса. Результат операций «указатель + целое», «указатель – целое» имеет тип «указатель».*

*Можно вычесть один указатель из другого (оба указателя-операнда при этом должны иметь одинаковый тип). Результат такого вычитания будет иметь тип целого числа со знаком. Его значение показывает, на сколько байт (или других единиц измерения) один адрес отстоит от другого в памяти.*

*Необходимо отметить, что сложение указателей не имеет смысла. Поскольку программа разрабатывается в относительных адресах и при разных своих выполнениях может размещаться в разных областях памяти, сумма двух адресов в программе будет давать разные результаты при разных выполнениях. Смещение же объектов внутри программы относительно друг друга не зависит от адреса загрузки программы, поэтому результат операции вычитания указателей будет постоянным, и такая операция является допустимой.*

*Операции адресной арифметики выполняются только над типизированными указателями. Единицей измерения в адресной арифметике является размер объекта, который указателем адресуется. Так, если переменная x определена как указатель на целое число, то выражение x+1 даст адрес, больший не на 1, а на количество байт в целом числе. Вычитание указателей также дает в результате не количество байт, а количество объектов данного типа, помещающихся в памяти между двумя адресами. Это справедливо как для указателей на простые типы, так и для указателей на сложные объекты, размеры которых составляют десятки, сотни и более байт.*

Ниже приведено несколько схем, иллюстрирующих работу с указателями.



**2. ХЕШИРОВАНИЕ ДАННЫХ**

**2.1. Открытое хеширование**

На рис. 1 показана базовая структура данных при открытом хешировании.

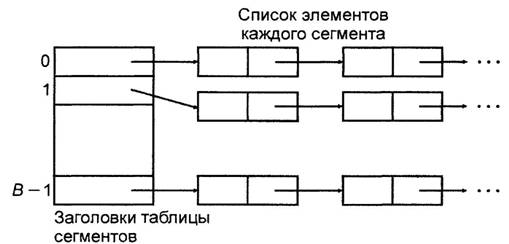


Рис. 1 – Организация данных при открытом хешировании

*Основная идея метода  заключается в том, что* ***множество данных*** *разбивается на* ***конечное число классов.***

*Для В классов, пронумерованных от 0 до В-1, строится хеш-функция  h такая, что для любого элемента x исходного множества функция h(x) принимает целочисленное значение из интервала 0, …, В-1, которое соответствует классу, которому принадлежит элемент x.*

***Элемент x****называют ключом,*

***h(x)****– хеш-значением х,*

***классы*** *– сегментами.*

***Массив (таблица сегментов)****, проиндексированный номерами сегментов 0, 1, … В-1, содержит заголовки для В списков.*

***Элемент х i-го списка*** *– это элемент исходного множества, для которого h(x)=i.*

*Идеальной хеш-функцией является такая, которая для любых двух неодинаковых ключей выдает неодинаковые адреса, т.е.*

**

*Однако подобрать такую функцию можно в случае, если все возможные значения ключей известны заранее. Такая организация данных носит название* ***«совершенное хеширование».***

*Рассмотрим хеш-функцию, определенную на символьных строках, которая не является совершенной, однако значения h(x) будут «хорошими».*

***Пример 1. Хеш-функция, определенная на символьных строках***

*Program hesh;*

*Var*

*i,B : integer;*

*s : string[10];*

*{Хеш-функция с  возвращаемым значением типа Integer}*

*Function h(x : string[10]): integer;*

*Var i, sum: integer;*

*Begin*

*sum:=0;*

*Ffor i:=1 to 10 do*

*sum:=sum+ord(x[i]);*

***h:=sum mod B;****{Возвращаемое значение}*

*End;*

*{Вызывающая программа}*

*Begin*

*Writeln ('Введите число классов В');*

*Readln(B);*

*Writeln ('Введите ключ');*

*Readln (S);*

*Writeln (h(s));*

*End.*

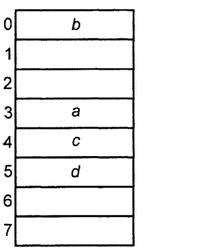
*Идея построения этой функции заключается в том, чтобы представить символы в виде целых чисел, используя для этого машинные коды символов. Если х –  это ключ, тип данных ключей определен как строка символов, тогда можно использовать хеш-функцию, код которой приведен ниже.*

**2.2. Закрытое хеширование**

*При закрытом хешировании в таблице сегментов хранятся непосредственно элементы словаря, а не заголовки списков. Поэтому в каждом сегменте может храниться только один элемент словаря. При закрытом хешировании применяется методика повторного хеширования.*

*Если произойдет попытка поместить элемент х в сегмент с номером h(x), который уже занят другим элементом (такая ситуация называется коллизией), то в соответствии с методикой повторного хеширования выбирается последовательность других номеров сегментов  куда можно поместить элемент х. Каждое из этих местоположений последовательно проверяется, пока не будет найдено свободное. Если свободных сегментов нет, то таблица заполнена, и элемент х вставить в нее нельзя.*

*Рассмотрим пример заполнения хеш-таблицы в случае закрытого хеширования (рис. 2).*

**

*Рис. 2 – Частично заполненная хеш-таблица*

*Здесь В=8 и ключи a, b, c, d имеют  хеш-значения h(a)=3, h(b)=0, h(c)=4, h(d)=3. Для добавления в таблицу элемента d применим методику  линейного хеширования, когда  Тогда, если сегмент 3 уже занят, то проверяются на занятость сегменты 4, 5, 6, 7, 0, 1, 2 (именно в таком порядке). 5-ый сегмент оказался первым пустым сегментом, поэтому элемент d был вставлен в него.*

*При поиске элемента х необходимо просмотреть все адреса   пока не будет найден х, или пока не встретится пустой сегмент.*

*Покажем это, введя условие, что в словаре не допускается удаление элементов. Пусть  – первый пустой сегмент. В такой ситуации невозможно нахождение элемента х в сегментах  и далее, т.к. х вставляется  в первый свободный сегмент, следовательно, он находится до сегмента  .*

**2.3 Методы разрешения коллизий при закрытом хешировании**

*При заполнении таблицы могут возникать коллизии, для борьбы с которыми разработаны специальные методы, которые в основном сводятся к методам "цепочек" и "открытой адресации". Ключи, выдающие одинаковые адреса в таблице, называются ключи-синонимы.*

*В методе цепочек для разрешения коллизий во все записи вводятся указатели, используемые для организации списков – "цепочек переполнения". В случае возникновения коллизии при заполнении таблицы в список для требуемого адреса хеш-таблицы добавляется еще один элемент.*

*Поиск в хеш-таблице с цепочками переполнения осуществляется следующим образом. Сначала вычисляется адрес по значению ключа. Затем осуществляется последовательный поиск в списке, связанном с вычисленным адресом. Процедура удаления из таблицы сводится к поиску элемента и его удалению из цепочки переполнения.*

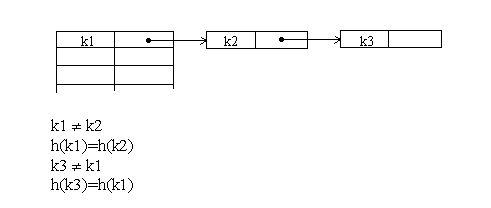


Рис. 3 –  Разрешение коллизий при добавлении элементов

*Метод открытой адресации состоит в том, чтобы, пользуясь каким-либо алгоритмом, обеспечивающим перебор элементов таблицы, просматривать их в поисках свободного места для новой записи.*

**Различают:**

а)Линейное пробирование

б)Квадратичное пробирование

в)Двойное хеширование

*Линейное пробирование сводится к последовательному перебору элементов таблицы с некоторым фиксированным шагом*

*a=h(key) + c\*i ,*

*где i – номер попытки разрешить коллизию. При шаге равном единице происходит последовательный перебор всех элементов после текущего.*

*Квадратичное пробирования отличается от линейного тем, что шаг перебора элементов нелинейно зависит от номера попытки найти свободный элемент*

*a = h(key2) + c\*****i****+ d\*****i****2.*

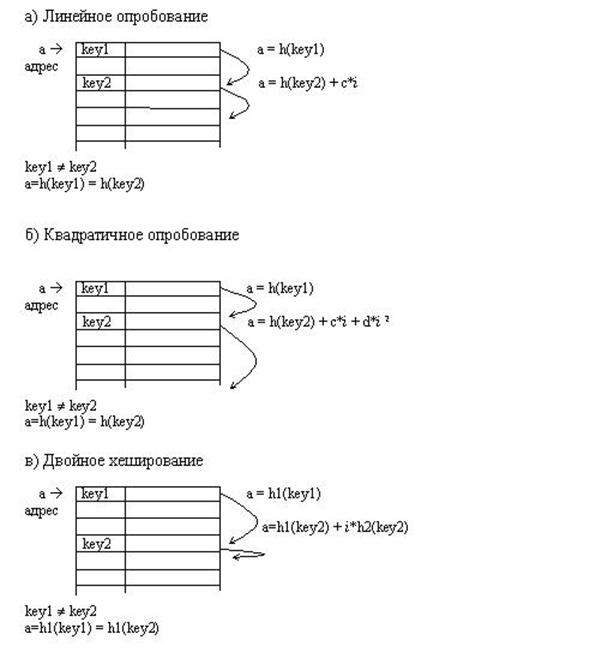


Рис. 4  –  Разрешение коллизий при добавлении элементов методами

открытой адресации

Интервал между ячейками фиксирован, как при **линейном пробировании**, но, в отличие от него, размер интервала вычисляется второй, вспомогательной хеш-функцией, а значит может быть различным для разных ключей.

Значения этой хеш-функции должны быть ненулевыми и взаимно-простыми с размером хеш-таблицы, что проще всего достичь, взяв простое число в качестве размера, и потребовав, чтобы вспомогательная хеш-функция принимала значения от 1 до N — 1.

**2.4. Алгоритмы работы с хеш-таблицами методами открытой адресации**

Опишем алгоритмы вставки и поиска элементов для метода **линейного опробования.** Здесь  t(a) – строка (элемент) в хеш-таблице.

Вставка

i = 0

a = h(key) + i\*c

Если t(a) = свободно, то t(a) = key, записать элемент, ***стоп элемент добавлен***

i = i + 1, перейти к шагу 2

Поиск

i = 0

a = h(key) + i\*c

Если t(a) = key, то ***стоп элемент найден***

Если t(a) = свободно, то ***стоп элемент не найден***

  i = i + 1, перейти к шагу 2

 *Аналогичным образом можно было бы сформулировать алгоритмы добавления и поиска элементов для любой схемы открытой адресации. Отличия будут только в выражении, используемом для вычисления адреса.*

*Процедура удаления в данном случае не будет являться обратной процедуре вставки. Элементы таблицы находятся в двух состояниях:****свободно****и****занято****.*

*Если удалить элемент, переведя его в состояние свободно, то после такого удаления алгоритм поиска будет работать некорректно.*

Рассмотрим алгоритмы вставки, поиска и удаления для хеш-таблицы, имеющей три состояния элементов (свободно, занято удалено).

Вставка

i = 0

a = h(key) + i\*c

Если t(a) = свободно или t(a) = удалено, то t(a) = key, записать элемент, ***стоп элемент добавлен***

i = i + 1, перейти к шагу 2

 Удаление

i = 0

a = h(key) + i\*c

Если t(a) = key, то t(a) =удалено, ***стоп элемент удален***

Если t(a) = свободно, то ***стоп элемент не найден***

i = i + 1, перейти к шагу 2

 Поиск

i = 0

a = h(key) + i\*c

Если t(a) = key, то ***стоп элемент найден***

Если t(a) = свободно, то ***стоп элемент не найден***

i = i + 1, перейти к шагу 2

**При использовании хеш-таблиц необходимо стараться избегать очень плотного ее заполнения. Обычно длину таблицы выбирают из расчета двукратного превышения предполагаемого максимального числа записей. В случае большой заполненности таблицы может понадобиться рехеширование. В этом случае увеличивают длину таблицы, изменяют хеш-функцию и переупорядочивают данные.**

Рассмотрим алгоритм вставки, реализующий предлагаемый подход.

 Вставка

i = 0

a = ((h(key) + c\*i) div n + (h(key) + c\*i) mod n) mod n

Если t(a) = свободно или t(a) = удалено, то t(a) = key, записать элемент, ***стоп элемент добавлен***

Если i > m , то ***стоп требуется рехеширование***

i = i + 1, перейти к шагу 2

 Удаление

i = 0

a = ((h(key) + c\*i) div n + (h(key) + c\*i) mod n) mod n

Если t(a) = key, то t(a) =удалено, ***стоп элемент удален***

Если t(a) = свободно или i>m, то ***стоп элемент не найден***

i = i + 1, перейти к шагу 2

Поиск

i = 0

a = ((h(key) + c\*i) div n + (h(key) + c\*i) mod n) mod n

Если t(a) = key, то ***стоп элемент найден***

Если t(a) = свободно или i>m, то ***стоп элемент не найден***

i = i + 1, перейти к шагу 2

 В данном алгоритме номер итерации сравнивается с пороговым числом *m*. Следует отметить, что алгоритмы вставки, поиска и удаления должны использовать  одинаковое образование адреса очередной записи.

**3. СТРУКТУРЫ ДАННЫХ**

**3.1 Полустатические и динамические структуры данных**

**Полустатические структуры данных характеризуются следующими признаками:**

**- они имеют переменную длину и простые процедуры ее изменения;**

**- изменение длины структуры происходит в определенных пределах, не превышая какого-то максимального (предельного) значения.**

Если полустатическую структуру рассматривать на логическом уровне, то о ней можно сказать, что это последовательность данных, связанная отношениями линейного списка.

**Физическое представление полустатических структур данных в памяти – это обычно последовательность слотов в памяти, где каждый следующий элемент расположен в следующем слоте. Физическое представление может иметь вид однонаправленного списка, где каждый следующий элемент адресуется указателем, находящимся в текущем элементе.**

В противовес статическим величинам, память под которые выделяется во время компиляции и сохраняется в течение всей работы программы, существуют динамические величины. *Под них память выделяется динамически во время выполнения программы.*

**Использование динамических величин предоставляет программисту ряд дополнительных возможностей.**

* **Во-первых, подключение динамической памяти позволяет увеличить объем обрабатываемых данных.**
* **Во-вторых, если потребность в каких-то данных отпала до окончания программы, то занятую ими память можно освободить для другой информации.**
* **В-третьих, использование динамической памяти позволяет создавать структуры данных переменного размера.**

При этом следует понимать, что работа с динамическими данными замедляет выполнение программы, поскольку доступ к величине происходит в два этапа: сначала ищется **указатель**, а затем по нему –  **величина**.  Работа с динамическими величинами связана с использованием ссылочного типа данных. Указатель содержит адрес поля в динамической памяти, хранящего величину определенного типа. Сам указатель располагается в статической памяти.

**3.2 Тип «Запись»**

***Запись* – это структурированный тип данных, состоящий из фиксированного числа компонентов одного или нескольких типов. Определение типа записи начинается идентификатором *record*  и заканчивается зарезервированным словом *end*. Между ними располагается список компонентов, называемых *полями*, с указанием идентификаторов полей и типа каждого поля.**

***Формат записи:***

***Type***

***<имя типа> = record***

***<идентификатор поля>:< тип компонента>;***

***…***

***<идентификатор поля>:< тип компонента>;***

***End;***

***Var***

***<идентификатор, …> : < имя типа >;***

***Пример:***

***Type***

***Care = record***

***Number : Integer;       { Номер}***

***Marka : String[20];  {Марка автомобиля}***

***FIO : String[40];  {Фамилия, имя, отчество владельца}***

***Address : String[60];  {Адрес владельца}***

***End;***

***Var***

***M,V : Car;***

В данном примере запись *Car* содержит четыре компонента. Доступ к полям записи осуществляется через переменную типа «запись». В нашем случае это переменные *M* и *V* типа *Car.*

        Значения полей записи могут использоваться в выражениях. Имена отдельных полей не применяются по аналогии с идентификаторами переменных, поскольку может быть несколько записей одинакового типа. Обращение к значению поля осуществляется с помощью идентификатора переменной и идентификатора поля, разделенных точкой. Такая комбинация называется *составным именем*. Для получения доступа к полям записи *Car* можно записать*: M.Number, M.Marka, M.FIO, M.Address*.

        Составное имя можно использовать везде, где допустимо применение типа поля. Для присваивания полям значений используется оператор присваивания.

**Пример:**

*M.Number:=1234;*

*M.Marka:=’Mersedes’.*

        (Важно!) Указатели могут использоваться с типом «запись» таким же образом, как и с данными других типов.

**3.3 Абстрактный тип данных «список»**

**В математике список представляет собой последовательность элементов определенного типа: *а*1, *а*2, …..*аn*, где *n*≥0 и все *аi* имеют один тип. Количество, *n* – длина списка, если *n* ≥ 1, то *а*1 – первый элемент, а *аn* – последний, в случае *n* = 0 имеем пустой список.**

Списки являются чрезвычайно гибкой структурой: их легко сделать большими или меньшими; их элементы доступны для вставки или удаления в любой позиции списка. Списки можно объединять или разбивать на меньшие списки.

При реализации списков с помощью массивов их элементы располагаются в смежных ячейках массива. Это представление позволяет легко просматривать содержимое списка и вставлять новые элементы в его конец. Однако вставка нового элемента в середину списка требует перемещения всех последующих элементов на одну позицию к концумассива. Операция удаления также требует перемещения элементов списка с целью освобождения ячейки.

Рассмотрим реализацию однонаправленного связанного списка с использованием указателей, связывающих последовательные элементы списка.

**Однонаправленный связанный список представляет собой динамическую структуру данных, число элементов которой может изменяться по мере того, как данные помещаются в список или удаляются из него.** Эта реализация освобождает от непрерывной области памяти для хранения списка и, следовательно, от необходимости перемещения элементов списка при вставке или удалении элементов. Однако ценой за это удобство становится использование дополнительной памяти для хранения указателей.

**В этой реализации список состоит из ячеек, каждая из которых содержит элемент списка и указатель на следующую ячейку списка. Если список состоит из элементов F:\12_116608_1_53138\ЭУМКД_СиАОД\Теория\content\lk3\lk3.files\image001.gif то для *i=1, 2,…, n-1* ячейка, содержащая элемент F:\12_116608_1_53138\ЭУМКД_СиАОД\Теория\content\lk3\lk3.files\image002.gif, имеет также указатель на ячейку, содержащую элемент F:\12_116608_1_53138\ЭУМКД_СиАОД\Теория\content\lk3\lk3.files\image003.gif. Ячейка, содержащая элемент F:\12_116608_1_53138\ЭУМКД_СиАОД\Теория\content\lk3\lk3.files\image004.gif, имеет указатель *nil* (нуль). Имеется также ячейка *header* (заголовок), которая указывает на ячейку, содержащую F:\12_116608_1_53138\ЭУМКД_СиАОД\Теория\content\lk3\lk3.files\image005.gif. Ячейка *header* не содержит элементов списка. В случае пустого списка заголовок имеет указатель *nil*, не указывающий ни на какую ячейку. Список, не содержащий элементов, называется пустым или нулевым.**

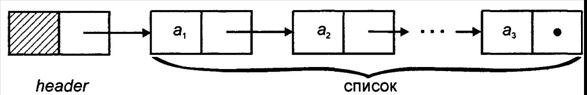
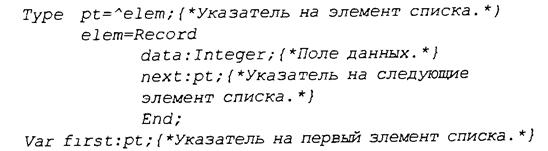


Рис. 1 – Связанный список

Для однонаправленных списков удобно использовать определение позиций элементов, отличное от того определения, которое применялось в реализации списков с помощью массивов. Здесь для *i=2, 3, … , n* позиция *i* определяется как указатель на ячейку, содержащую указатель на  элемент F:\12_116608_1_53138\ЭУМКД_СиАОД\Теория\content\lk3\lk3.files\image002.gif. Позиция 1 – это указатель в ячейке заголовка, а позиция *end*(L) – указатель в последней ячейке списка L.

**Формально структуру связанного списка можно определить следующим образом:**



**Основные операции, выполняемые с элементами списка: просмотр, вставка, удаление.**

**Начнем с  процедуры создания  списка. Ниже приведен ее код. Здесь создается список целых чисел, и признаком завершения построения является введенное нулевое значение в  поле данных.**

***Procedure make(x : pt);***

***Var  y:pt;***

***Begin***

***New(x);***

***Readln (x^.data);***

***While(  x^.data <> 0)   do***

***Begin***

***y:=x;***

***New(x);***

***y^.next:=x;***

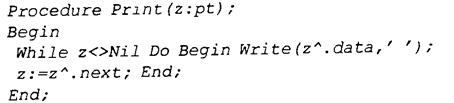
***Readln(x^.data);***

***End;***

***y^.next:=nil;***

***End;***

**Процедура просмотра элементов списка**



**Процедура вставки элемента в заданную позицию списка*.***

***Procedure Insert ( x : integer; p:pt);***

***Var***

***temp: pt;***

***Begin***

***(1)               temp:= p^.next;***

***(2)               New(p^.next);***

***(3)               p^.next^.data:= x;***

***(4)              p^.next^.next:=temp;***

***End;***

**Механизм управления указателями в процедуре *Insert* приведен на рис. 1. На рисунке 1. а) показана ситуация перед выполнением процедуры вставки. Необходимо вставить новый элемент *x* перед элементом *b*, поэтому в строке (1) задается *temp* как указатель на ячейку, содержащую элемент *b*. В строке (2) листинга создается новая ячейка, и в поле *next*ячейки, содержащей элемент *а*, ставится указатель на новую ячейку. В строке (3) поле *data*   вновь созданной ячейки принимает значение *х*, а в строке (4) поле *next* этой ячейки принимает значение переменной *temp*, которая хранит указатель на ячейку, содержащую элемент *b*. На рисунке 1. б) представлен результат выполнения процедуры *Insert*, где пунктирными линиями показаны новые указатели и номерами (совпадающими с номерами строк в листинге) помечены этапы их создания.**

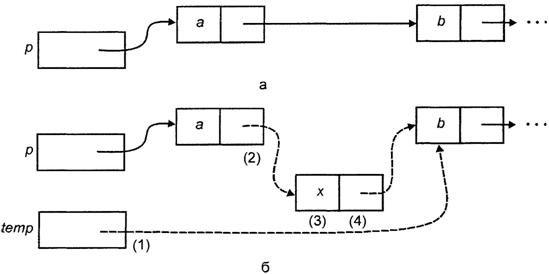


Рис. 1 – Реализация вставки элемента в список

**Показана схема манипулирования указателями в  процедуре удаления элемента из списка. Старые указатели показаны сплошными линиями, а новые – пунктирной. Ниже приведен код процедуры удаления элемента.**

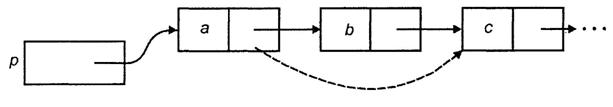
***Procedure Delete ( b : integer; p:pt);***

***Begin***

***If  p^.next^.data=b then***

***p^.next:= p^.next^.next;***

***End;***

******

Линейные списки находят широкое применение в приложениях, где непредсказуемы требования к размеру памяти, необходимой для хранения данных; большое число сложных операций над данными, особенно включений и исключений. На базе линейных списков могут строиться другие структуры.

**3.3 Сравнение различных реализаций списков**

В каких ситуациях лучше использовать реализацию списков с помощью указателей, а когда – с помощью массивов, зависит от того, какие **операторы** должны выполняться над списками, и как часто они будут использоваться. Иногда аргументом в пользу одной или другой реализации может служить максимальный размер обрабатываемых списков. Приведем несколько принципиальных соображений по этому поводу.

*1.* Реализация списков с помощью массивов требует указания максимального размера списка до начала выполнения программ. *Если  невозможно заранее ограничить сверху длину обрабатываемых списков, то более рациональным выбором будет реализация списков с помощью указателей.*

*2.     Выполнение некоторых операторов в одной реализации требует больших вычислительных затрат, чем в другой. Например, процедуры INSERT и DELETE выполняются за постоянное число шагов  в случае связанных списков любого размера, но требуют времени, пропорционального числу элементов, следующих за вставленным (или удаляемым) элементом, при использовании массивов. И наоборот, время выполнения функции для выделения предыдущего или последнего элемента списка  постоянно при реализации списков посредством массивов, но в это же время пропорционально длине списка в случае реализации, построенной с помощью указателей.*

*3.     Если необходимо вставлять или удалять элементы, положение которых указано с помощью специальной переменной типа position, и значение этой переменной будет использовано позднее, то нецелесообразно использовать реализацию с помощью указателей, поскольку эта переменная не «отслеживает» вставку и удаление элементов.*

*4.     Реализация списков с помощью массивов расточительна по отношению к компьютерной памяти, поскольку резервируется объем памяти, достаточный для максимально возможного размера списка независимо от его реального размера в конкретный момент времени. Реализация с помощью указателей использует  столько памяти, сколько необходимо для хранения текущего списка, но требует дополнительную память для указателя каждой ячейки. Таким образом, в разных ситуациях по критерию используемой памяти могут быть выгодны разные реализации.*

**3.4 Дважды связные списки**

Иногда возникает необходимость организовать эффективное перемещение по списку, как в прямом, так и в обратном направлениях; или по заданному элементу нужно быстро найти предшествующий ему и последующий элементы. В этих ситуациях можно дать каждой ячейке указатели и на следующие, и на предыдущие ячейки списка, т.е. организовать дважды связный список. На рис. 3 приведен такой список.

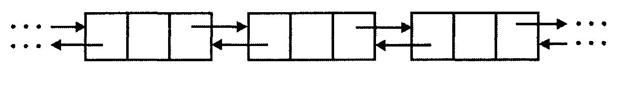


Рис. 3 – Дважды связный список

Важным преимуществом является то, что можно использовать указатель ячейки, содержащей i-й элемент, для определения i-й позиции вместо использования указателя предшествующей ячейки. Однако при этом появляются дополнительные указатели и, следовательно, удлиняются некоторые процедуры. Ниже приведен код объявления дважды связного списка.

***Type pt=^elem; {Указатель на элемент списка}***

***Elem=Record***

***Data : Integer; {Поле данных}***

***Next :pt; {Указатель на следующий элемент списка}***

***Previous : pt; {Указатель на предыдущий элемент списка}***

***End;***

***Var first:pt; {Указатель на первый элемент списка}***

       Ниже  показаны процедуры построения  двусвязного списка, а также удаления элемента из него.

       На рис. 3 приведены процедура и схема удаления элемента из дважды связного списка. Здесь предполагается,  что удаляемая ячейка не является ни первой, ни последней в списке. Сплошными линиями показаны указатели до удаления, а пунктирными – после удаления элемента. Сначала с помощью указателя поля *previous* определяется положение предыдущей ячейки. Затем в поле *next* этой ячейки устанавливается указатель на ячейку, предшествующую позиции *р*. Таким образом, ячейка в позиции *р* исключается из цепочек указателей и при необходимости может быть использована повторно.

*Procedure Make(var x : pt); {Построение списка из пяти элементов}*

*var t,y:pt;*

*i: integer;*

*begin*

*new(x);*

*x^.prev:=nil;*

*for i:=1 to 5 do*

*begin*

*y:=x;*

*readln(y^.data);*

*if  i<>5 then*

*begin*

*new(x);*

*y^.next:=x;*

*x^.prev:=y;*

*end*

*else*

*y^.next:=nil;*

*end;*

*Procedure Delete (p:pt);*

*Begin*

*If    p^.previous <> nil  then  {Удаление ячейки, не являющейся первой}*

*p^.previous^.next:=p^.next;*

*If    p^.next <> nil  then {Удаление ячейки, не являющейся последней}*

*p^. next^. previous:=p^. previous;*

*End;*

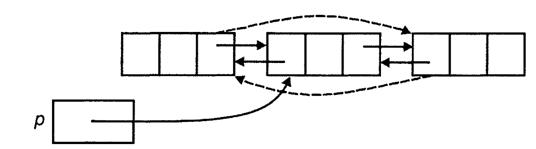


Рис. 3 – Удаление элемента из дважды связного списка

**4. ОЧЕРЕДИ И СТЕКИ**

**4.1 Абстрактный тип данных «очередь»**

**Очередь – это специальный тип списка. Очередью называют упорядоченный набор элементов, где  элементы удаляются с одного его конца, который называется началом, а вставляются с другого, который называется концом очереди.** Очереди также называются списками типа *FIFO* (аббревиатура расшифровывается как *first in first out*: первым вошел – первым вышел).

**Две основные операции, которые определены для работы с очередью: *вставка и извлечение элементов*.**  Кроме того, используются и другие операторы.

1) *MAKENULL(Q)* очищает очередь *Q*, делая ее  пустой.

2) *FRONT(Q)* – функция, возвращающая первый  элемент очереди *Q*.

3) *ENQUEUE(x, Q)* вставляет элемент *х* в конец очереди *Q*.

4) *DEQUEUE (Q)* удаляет первый элемент очереди *Q*.

5) *EMPTY (Q)*. Эта функция возвращает значение *true* (истина) тогда и только тогда, когда *Q* является  пустой очередью.

**4.2 Реализация очереди с помощью указателей**

Рассмотрим способ реализации очереди на базе списков, учитывая структурные и функциональные особенности очереди. Например, при добавлении элемента в конец очереди  вместо перемещения по списку от начала к концу можно хранить указатель на конец очереди. Указатель на начало очереди будет полезен при выполнении операторов, связанных с извлечением элементов из структуры. В языке *Pascal* в качестве заголовка можно использовать динамическую переменную и поместить в нее указатель на начало очереди, что позволит удобно организовать очищение очереди.

Определим список, содержащий указатели на начало и конец очереди. Первая ячейка очереди – это заголовок, в котором отсутствуют данные. Ниже определен абстрактный тип данных «очередь».

*Type ptr = ^queue;{Указатель на элемент очереди}*

*Queue = record*

*data : integer; {Поле данных}*

*next, front, rear : ptr;{Указатели на следующий элемент, голову и*

*End;                                 хвост очереди}*

*Var Q : ptr; {Указатель на заголовок очереди}*

Рассмотрим программную реализацию некоторых из вышеперечисленных  операторов над очередями.

***MAKENULL(Q)* очищает очередь *Q*, делая ее  пустой.**

***1)    Procedure Makenull (var q:ptr);***

***begin***

***new(q); {Создание ячейки заголовка}***

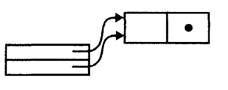
***q^. front:=q;***

***q^. front^.next:=nil;***

***q^. rear:=q^.front;***

***end;***

|  |  |
| --- | --- |
| |  | | --- | | ***Q^.front***  ***Q^.rear*** | |



**FRONT(Q) – функция, возвращающая первый  элемент очереди Q.**

**function FRONT ( Q: QUEUE ): eltype ;**

**begin**

**if EMPTY(Q)**

**then Write('Очередь пуста')**

**else Front := Q. Front^.next^.element**

**end; { FRONT }**

***ENQUEUE(x, Q)* вставляет элемент *х* в конец очереди *Q*.**

***procedure ENQUEUE(x:eltype;var Q:QUEUE) ;***

***begin***

***New (Q.rear^.next) ;***

***Q.rear:= Q.rear^.next;***

***Q.rear^.element:= x;***

***O.rear^.next:= nil***

***end; { ENQUEUE }***

***DEQUEUE (Q)* удаляет первый элемент очереди *Q*.**

***Procedure dequeue (var q:ptr); {Удаление первого элемента очереди}***

***begin***

***if empty (q) then***

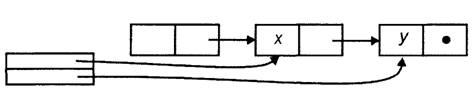
***writeln('Очередь пуста')***

***else***

***q^.front:=q^.front^.next;***

***end;***

|  |  |
| --- | --- |
| |  | | --- | | ***Q^.front***  ***Q^.rear*** | |



***EMPTY (Q)*. Эта функция возвращает значение *true* (истина) тогда и только тогда, когда *Q* является  пустой очередью.**

***Function empty(q:ptr):boolean;***

***var z:boolean;***

***begin***

***if***

***q^.front= q^.rear then  {Проверка пустоты очереди}***

***z:=true***

***else     z:=false;***

***end;***

**4.3. Разновидности очередей**

**Интересной разновидностью очередей являются многопоточные очереди (multi headed queues). Элементы в нее, как обычно, добавляются в конец очереди, но очередь имеет несколько потоков (front end) или голов (heads).**

*Примером многопоточной очереди в обычной жизни является* ***очередь клиентов в банке.*** *Все клиенты находятся в одной очереди, но их обслуживает несколько служащих. Освободившийся банковский работник обслуживает клиента, который находится в очереди первым. Такой порядок обслуживания кажется справедливым, поскольку клиенты обслуживаются в порядке прибытия. Он также эффективен, так как все служащие остаются занятыми до тех пор, пока клиенты ждут в очереди.*

В общем случае многопоточные очереди более эффективны, чем несколько однопоточных очередей.

*Циклическая очередь.* Некоторые программы можно улучшить, используя циклические очереди. При организации очереди на основе массива при достижении предела массива можно вернуть указатели вставки в очередь и удаления из нее на начало массива. В этом случае **в очередь можно добавлять и удалять из нее любое число элементов. Такая очередь называется циклической, поскольку массив используется не как линейный список, а как циклический.**

В действительности очередь становится заполненной только в том случае, когда указатель свободного места совпадает с указателем выборки следующего элемента. В противном случае очередь будет иметь свободное место для нового элемента. В начале программы индекс выборки должен устанавливаться не в нулевое значение, а на значение  максимального числа событий. В противном случае первое обращение к процедуре постановки в очередь приведет к появлению сообщения о заполнении структуры. Следует помнить, что очередь может содержать только на один элемент меньше, чем значение максимального числа событий, поскольку указатели выборки и постановки в очередь всегда должны отличаться хотя бы на единицу (в противном случае нельзя будет понять заполнена ли очередь или она пустая).

**Наиболее широко циклические очереди применяются в операционных системах при буферизации информации, которая считывается или записывается на дисковые файлы или консоль.** Другое широкое применение эти очереди нашли в решении задач реального времени, когда, например, пользователь может продолжать делать ввод с клавиатуры во время выполнения другой задачи. **Так работают многие текстовые процессоры, когда изменяется формат параграфа или выравнивается строка.** Имеется короткий промежуток времени, когда набранная на клавиатуре информация не выводится на экран до окончания другого процесса. Для достижения такого эффекта в программе должна предусматриваться постоянная проверка ввода с клавиатуры в ходе выполнения другого процесса. При вводе некоторого символа его значение должно быстро ставиться в очередь, и процесс должен продолжаться. После завершения процесса набранные символы выбираются из очереди и обрабатываются обычным образом.

***Приоритетные очереди*.** Каждый элемент в приоритетной очереди (priority queue) имеет связанный с ним приоритет. Если программе нужно удалить элемент из очереди, она выбирает элемент с наивысшим приоритетом. Как хранятся элементы в приоритетной очереди, не имеет значения, если программа всегда может найти элемент с наивысшим приоритетом.

Некоторые операционные системы использую **приоритетные очереди для планирования заданий.** В операционной системе UNIX все процессы имеют разные приоритеты. Когда процессор освобождается, выбирается готовый к исполнению процесс с наивысшим приоритетом. Процессы с более низким приоритетом должны ждать завершения или блокировки (например, при ожидании внешнего события, такого как чтение данных с диска) процессов с более высокими приоритетами.

**Простой способ организации приоритетной очереди — поместить все элементы в список.** Если требуется удалить элемент из очереди, можно найти в списке элемент с наивысшим приоритетом. Чтобы добавить элемент в очередь, он помещается в начало списка. При использовании этого метода, для добавления нового элемента в очередь требуется только один шаг. Чтобы найти и удалить элемент с наивысшим приоритетом, требуется O(N) шагов, если очередь содержит N элементов.

Немного лучше была бы схема с использованием связного списка, в котором элементы были бы упорядочены в прямом или обратном порядке. Чтобы добавить элемент в очередь, нужно найти его правильное положение в списке и поместить его туда. Чтобы упростить поиск положения элемента, можно использовать сигнальные метки в начале и конце списка, присвоив им соответствующие приоритеты.

**4.4 Абстрактный тип данных «стек»**

**Стек – это специальный тип списка,  в котором все вставки и удаления выполняются только на одном конце, называемом вершиной (*top*). В англоязычной литературе для обозначения стеков используется аббревиатура LIFO (last-in-first-out – последний вошел, а первый вышел).**

**Интуитивными моделями стека могут служить колода карт, лежащая на столе; книги, сложенные в стопку и т.д. В таких моделях можно взять только верхний предмет, а добавить новый объект можно, только положив его на верхний.**

**Абстрактные типы данных семейства стек обычно используются следующие пять операторов.**

**1) MAKENULL(S). Делает стек пустым.**

**2) TOP(S). Возвращает элемент из вершины стека *S*. Обычно вершина стека идентифицируется позицией 1.**

**3) POP(S). Удаляет элемент из вершины стека (выталкивает из стека).**

**4) PUSH(x, S). Вставляет элемент *х* в вершину стека *S* (заталкивает элемент в стек). Элемент, ранее находившийся в вершине стека, становится элементом, следующим за вершиной и т.д.**

**5) EMPTY (S). Эта функция возвращает значение *true* (истина), если список  S пустой и значение *false* (ложь) –  в противном случае.**

**4.5 Реализация стеков с помощью массивов и списков**

Для реализации стека можно рационально приспособить массивы. Поскольку вставка и удаление элементов происходит только через вершину стека,  его «дно» фиксируют в самом низу массива (в ячейке с наибольшим индексом) и позволяют  стеку расти вверх массива (к ячейке с наименьшим индексом). Схема такого представления стека показана на рис. 1.

**Для такой реализации стеков можно определить абстрактный тип STAK следующим образом:**

***Type***

***Stack= record***

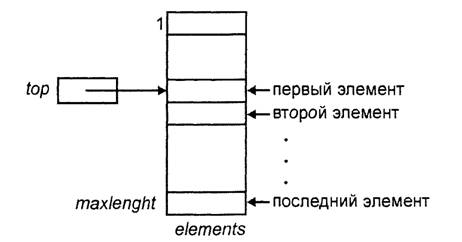
***Top: integer;***

***Elements : array[1..maxlenght] of integer***

***End;***

***Var   x : Stack;***

**В этой реализации стек состоит из последовательности элементов *elements[top], elemenst[top+1],… elements[maxlenght ].* Если  *top = maxlenght +1*, то стек пустой.**



**Рис. 1 – Реализация стека на основе массива**

**Еще одним удобным вариантом представления стека является способ на основе  списка, в котором добавление новых элементов и извлечение имеющихся происходит с одного конца списка.** Значением указателя, представляющего стек, является ссылка на вершину стека. Каждый элемент стека содержит поле ссылки на следующий элемент**.** В этом случае описать стек можно следующим образом.

*Type ptr = ^stack; {Указатель на элемент стека}*

*Stack = record*

*data : integer; {Поле данных}*

*next : ptr;{Указатели на следующий элемент  очереди}*

*End;*

*Var st : ptr;*

**Если стек пуст, то значение указателя *st* равно *nil*.**

**Рассмотрим программную реализацию нескольких наиболее часто выполняемых операторов с элементами стека.**

1. **Процедура записи элементов в стек содержит один параметр: указатель на начало стека. Эта процедура выполняется аналогично вставке элемента в начало списка.**

***1)       Procedure WriteStack (var st: ptr);***

***var i :integer; x :ptr;***

***begin***

***for i:=1 to 5 do {В стек заносится пять элементов}***

***begin***

***new(x);***

***readln(x^.data);***

***x^.next:=st;***

***st:=x;***

***end;***

***end;***

1. **Процедура извлечения элементов из стека содержит один параметр: указатель на начало стека. Сначала печатается элемент из вершины стека, а затем изменяется указатель на вершину стека.**

***2)******Procedure print(var st:ptr);***

***begin***

***writeln('Вывод элементов стека');***

***while st<> nil do***

***begin***

***writeln (st^.data);***

***st:=  st^.next;***

***end;***

***end;***

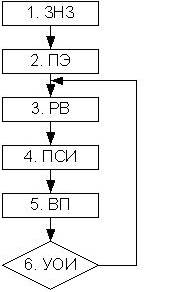
**5. ПРОЦЕДУРЫ И ФУНКЦИИ ВЫЧИСЛИТЕЛЬНОГО ПРОЦЕССА**

**5.1 Итерационный вычислительный процесс**

**Итерационным вычислительным процессом называется такой  циклический процесс, который продолжается до тех пор, пока разность между соседними, уточняемыми на каждом шаге цикла (итерации) значениями, не окажется меньше или равной некоторой заданной величине.** В виде итерационных вычислительных процессов реализуется большинство численных методов решения задач высшей математики. Характерной особенностью итерационного процесса является то, что в нем количество повторений (итераций) заранее неизвестно и становится определенным только после окончания вычислений.

**Второй особенностью** итерационного процесса является то, что **результаты вычислений очередного выполнения цикла используются как исходные данные при следующем выполнении цикла**, т.е. решение находится последовательными приближениями, путем уточнения на каждом шаге цикла.

**Структура итерационного процесса**



**В блоке 1 задаются начальные значения переменных, точность, с которой должно быть найдено решение (ЗНЗ).**

**Блок 2 (подготовительный этап) предназначен для выполнения некоторых вспомогательных операций по подготовке первой итерации. Этот блок в некоторых задачах может отсутствовать (ПЭ).**

**Описание: F:\12_116608_1_53138\ЭУМКД_СиАОД\Теория\content\lk5\lk5.files\image002.gifБлок 3 (или несколько блоков) реализуют вычисления, необходимые для выполнения итерации (РВ).**

**В блоке 4 выполняются действия для подготовки следующей итерации (ПСИ).**

**В блоке 5 (вычисление погрешности) определяется, насколько найденное решение отличается от эталонного или полученного на предыдущей итерации (ВП).**

**Блок 6 предназначен для проверки условия окончания итерации.**

**Например:  определение факториала может быть записано таким образом**

***Если  n=0              n! =1***

***n>0             n! = n\* (n-1)\*(n-2) …1.***

**Видно, что алгоритм требует явного повторения вычислений.**

**5.2 Рекурсивный вычислительный процесс**

**Объект называется *рекурсивным*, если он содержит сам себя или определен с помощью самого себя.** Мощность рекурсии связана с тем, что она позволяет определить бесконечное множество объектов с помощью конечного высказывания. Аналогичным образом бесконечные вычисления можно описать с помощью конечной рекурсивной программы, даже если она не содержит явных циклов. Однако лучше всего использовать рекурсивные алгоритмы в тех случаях, когда решаемая задача или вычисляемая функция определены с помощью рекурсии.

**Необходимое и достаточное средство для рекурсивного представления программ – это описание процедур или функций, т.к. оно позволяет присваивать какому-либо оператору имя, с помощью которого можно вызывать этот оператор.**

**Если процедура *Р* содержит явное обращение к самой себе, то она называется *прямо рекурсивной*; если *Р* содержит обращение к процедуре *Q*, которая содержит (прямо или косвенно) обращение к *Р*, то *Р* называется косвенно рекурсивной.**

С процедурой связывают некоторое множество локальных объектов (переменных, констант и т.д.), которые определены локально в этой процедуре, а вне ее не существуют или не имеют смысла. Каждый раз, когда такая процедура рекурсивно вызывается, для нее создается новое множество локальных переменных. Хотя они имеют те же имена, что и соответствующие элементы множества локальных переменных, созданного при предыдущем обращении к этой же процедуре, их значения различны. Следующие правила области действия идентификаторов позволяют исключить какой-либо конфликт при использовании имен: идентификаторы всегда ссылаются на множество переменных, созданное последним. То же правило относится к параметрам процедуры.

**Подобно операторам цикла, рекурсивные процедуры могут привести к бесконечным вычислениям.** Поэтому необходимо рассмотреть проблему окончания работы процедур.

**Для того, чтобы работа когда-либо завершилась, необходимо, чтобы рекурсивное обращение к процедуре *Р* подчинялось условию *В*, которое в какой-то момент перестает выполняться.**

**На практике необходимо убедиться в том, что наибольшая  глубина рекурсии не только конечна, но и достаточно мала. Поскольку при каждом рекурсивном вызове процедуры Р выделяется некоторая память для размещения ее переменных. Кроме этих локальных переменных, нужно еще сохранять текущее состояние вычислений, чтобы вернуться к нему, когда закончится выполнение новой активации Р, и нужно будет вернуться к старой.**

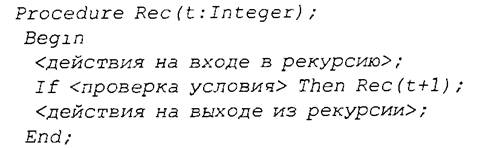
Сравнивая рекурсию с итерационными методами, следует отметить, что рекурсивные алгоритмы наиболее пригодны в случаях, когда поставленная задача или используемые данные определены рекурсивно. В тех случаях, когда вычисляемые значения определяются с помощью простых рекуррентных соотношений, гораздо эффективнее применять итеративные методы.

Рекурсивные алгоритмы особенно подходят для задач, где обрабатываемые данные определяются в терминах рекурсии.

**5.3 Реализация рекурсивного вычислительного процесса**

Стеки находят важное применение при реализации рекурсивных процедур в языках программирования. Организация выполнения процедур в языках программирования состоит в задании структур данных, которые используются для хранения значений программных переменных во время выполнения программы. Все языки программирования, допускающие рекурсивные процедуры, используют стеки активационных записей для хранения всех значений переменных, принадлежащих каждой активной процедуре. При вызове процедуры *Р* новая активационная запись для этой процедуры помещается в стек независимо от того есть ли в стеке другие активационные записи для процедуры *Р*. Таким образом, извлекая активационную запись из стека для последнего вызова процедуры Р, можно управлять возвратом к точке в программе, из которой Р вызывалась (эта точка, называемая адресом возврата, помещается в активационную запись процедуры Р при вызове этой процедуры).

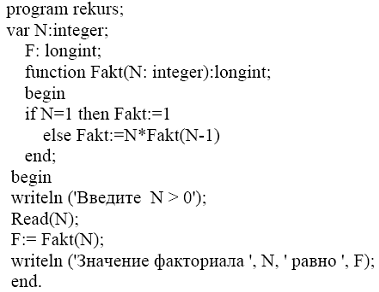
**Типичная конструкция рекурсивной процедуры имеет следующий вид.**

****

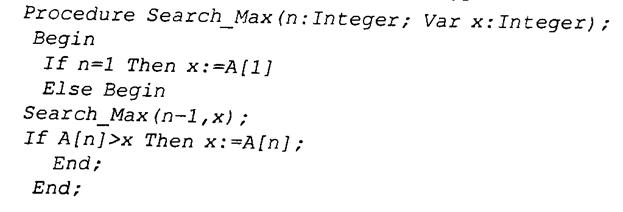
Рекурсивные вызовы процедур упрощают структуру многих программ. Но в некоторых языках программирования процедурные вызовы более «дорогие» (по времени выполнения), чем непосредственное выполнение операторов, поэтому программа может работать быстрее, если из нее исключить рекурсивные процедуры. На практике бывают ситуации, когда после реализации части программного проекта возникает необходимость исключить рекурсию.

**Рассмотрим несколько примеров рекурсивных процедур и функций.**

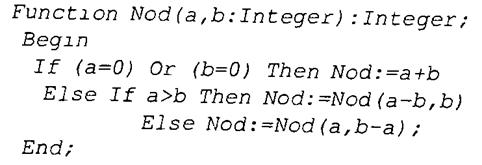
**1) Вычисление факториала целого числа.** Сначала проверяется  условие *N=1*. Если оно выполняется, то функции *Fakt* присваивается значение 1, на этом выполнение функции завершается. Если условие *N=1* не соблюдается, то выполняется вычисление произведения *N\*Fakt(N-1)*. Вычисление произведения носит рекурсивный характер, т.к. при этом осуществляется вызов функции *Fakt(N-1)*, значение которой вычисляется посредством вызова функции *Fakt*, параметром которой также является функция *Fakt*, и т.д., пока значение формального параметра не станет равным 1. После чего рекурсивные вызовы функции *Fakt* больше не выполняются, а происходит вычисление функции *Fakt*  для чисел, возрастающих от 1 до *N*, причем функция *Fakt* каждый раз возвращает значение, равное произведению очередного *к*-го числа на факториал от (*к*-1)-го числа. Последнее возвращение результата вычисления функции *Fakt* присвоит переменной *F* значение произведения всех чисел от 1 до *N*, т.е. факториал числа *N*.

****

**2) Нахождение максимального элемента в глобальном массиве *А.***

****

**3) Вычисление наибольшего общего делителя двух чисел.**



**6. ПОСТФИКСНАЯ И ПРЕФИКСНАЯ ФОРМЫ ЗАПИСИ**

**6.1 Различные формы записи выражений**

Введем понятие различных форм записи выражений. **А+В – инфиксная: знак операции находится между операндами; +АВ  – префиксная (польская): знак операции расположен перед операндами; АВ+  – постфиксная (обратная польская): знак операция находится после операндов.**

Хотя префиксная и постфиксная формы записи, на первый взгляд, кажутся не очень наглядны, они чаще инфиксной используются в вычислительной технике для обработки выражений.

Большую часть задач, решаемых с помощью программирования, составляют задачи,  в которых широко применяются методы вычислительной математики, а в них входят арифметические и логические выражения. Поэтому трансляцией выражений занимались очень многие исследователи и разработчики трансляторов. Сейчас классическим стал метод трансляции выражений, основанный на использовании промежуточной обратной польской записи, названной так в честь польского математика Яна Лукашевича, который впервые использовал эту форму представления выражений в математической логике. Однако в существующих трансляторах используются и другие методы. Польская запись – это префиксная, а обратная польская – постфиксная.

Для преобразования выражений из инфиксной в постфиксную и префиксную формы нужно учитывать правила приоритетности операций. Операции с высшим приоритетом преобразуются первыми, а после преобразования операция рассматривается как один операнд. Общепринятую приоритетность операций можно изменить при помощи скобок. При просмотре строки, не содержащей скобок, вычисления выполняются слева направо для операций с одинаковым приоритетом, за исключением случая возведения в степень, когда вычисления выполняются справа налево.

**Примеры различных форм записи выражений.**

**Инфиксное представление                           Постфиксное представление**

**А+В-С                                                                                    АВ+С-**

**(А+В)\*(С-D)                                                                          AB+CD-\***

**A^B\*C-D+E/F/(G+H)                                                      AB^C\*D-EF/GH+/+**

**A-B/(C\*D^E)                                                                         ABCDE^\*/-**

**Инфиксное представление                                   Префиксное представление**

**А+В-С                                                                                    - +АВС**

**(А+В)\*(С-D)                                                                          \*+AB-CD**

**((A+B)\*C-(D-E)^(F+G)                                                      ^ -\*+ABC-DE+FG**

**A-B/(C\*D^E)                                                                         -A/B\*C^DE**

Рассмотрим сущность обратной польской записи. В ней отсутствуют скобки, операнды располагаются в том же порядке, что в исходном выражении, а знаки операций при просмотре записи слева направо встречаются в том порядке, в котором нужно выполнять соответствующие действия.

**6.2 Построение выражений в обратной польской записи**

**Основное преимущество обратной польской записи перед обычной записью выражений со скобками: *выражение можно вычислить в процессе однократного просмотра слева направо***

**Правило вычисления выражения в обратной польской записи состоит в следующем.** Обратная польская запись просматривается слева направо.

**Если очередной элемент – операнд, то рассматривается следующий элемент.**

**Если текущий элемент –  знак операции, то выполняется эта операция над операндами, записанными левее знака операции. Результат операции записывается вместо первого (самого левого) операнда, участвовавшего в операции. Остальные элементы (операнды и знак операции), участвовавшие в операции, вычеркиваются из записи. Просмотр продолжается. В результате последовательного выполнения этого правила будут выполнены все операции в выражении, и запись сократится до одного элемента –  результата вычисления выражения.**

Для управления порядком выполнения операций над операндами используется скобочная запись выражений. При вычислении сначала вычисляется та часть выражения, скобочный уровень вложенности которой самый высокий. При наличии нескольких операторов с наивысшим скобочным уровнем, они вычисляются слева направо. При вычислении выражений с операциями, которым присвоен приоритет, или частично скобочных выражениях инфиксной формы записи, необходимо повторное сканирование слева направо.

**Повторное сканирование можно исключить, если инфиксное выражение преобразовать в постфиксную или префиксную форму записи (см. таблицу ниже).**

|  |  |  |
| --- | --- | --- |
| **Инфиксное**  **выражение** | **Постфиксное**  **Выражение** | **Префиксное**  **Выражение** |
| ***а+в*** | ***ав+*** | ***+ав*** |
| ***а+в+с*** | ***ав+с+*** | ***++авс*** |
| ***а+(в+с)*** | ***авс++*** | ***+а+вс*** |
| ***а+в\*с*** | ***авс\*+*** | ***+а\*вс*** |
| ***а\*(в+с)*** | ***aвс+\**** | ***\*а+вс*** |
| ***а\*в\*с*** | ***ав\*с\**** | ***\*\*авс*** |

**Перечислим основные правила выполнения постфиксного выражения:**

**1)    Найти в выражении крайний левый оператор.**

**2)   Выбрать два операнда, стоящих непосредственно слева от найденного оператора.**

**3)    Выполнить операцию.**

**4)    Заменить оператор и операнды результатом.**

**5)   Повторять указанные действия, пока не будут обработаны все операнды.**

Существуют компиляторы, которые преобразуют инфиксные  выражения на языках высокого уровня в обратную польскую запись.

**Постфиксное и префиксное  выражения корректны тогда и только тогда, когда ранг выражения равен 1, а ранг любой правой головы польской формулы больше (меньше) или равен 1. Ранг корректного выражения равен 1.**

Алгебраическое преобразование инфиксного выражения в обратное польское основано на приоритетах операторов и предлагает использование стека. Обратное польское выражение хранится в виде выходной строки, используемой в дальнейшем при генерации объектного кода.

В ходе преобразования инфиксного выражения в обратное польское порядок всех переменных и констант не меняется, а порядок операторов выходной строки соответствует  их приоритетам.

|  |  |  |
| --- | --- | --- |
| **Символ** | **Приоритет** | **Ранг** |
| **+,–** | **1** | **–1** |
| **\*, /** | **2** | **–1** |
| **A,b,…,z** | **3** | **1** |
| **Дно стека** | **0** | **–** |

**Рассмотрим пример преобразования инфиксного выражения *a+b\*c–d/e\*h* в обратное польское**

|  |  |  |  |
| --- | --- | --- | --- |
| **Сканируемый символ** | **Содержание стека** | **Обратная польская**  **запись** | **Ранг** |
|  | **|–** |  |  |
| **А** | **|–  a** |  |  |
| **+** | **|–  +** | **a** | **1** |
| **B** | **|–  +b** | **a** | **1** |
| **\*** | **|–  +\*** | **ab** | **2** |
| **С** | **|–  +\*c** | **ab** | **2** |
| **–** | **|–  –** | **abc\*+** | **1** |
| **D** | **|–  –d** | **abc\*+** | **1** |
| **/** | **|–  – /** | **abc\*+d** | **2** |
| **E** | **|–  – / e** | **abc\*+d** | **2** |
| **\*** | **|–  – \*** | **abc\*+de/** | **2** |
| **H** | **|–  – \* h** | **abc\*+de/** | **2** |
| **|–** | **|–** | **abc\*+de/h\*–** | **1** |

**Алгоритм преобразования.**

**1.  В стек помещается признак пустого стека.**

**2. Значение приоритета очередного входного символа сравнивается с приоритетом верхнего элемента стека.**

**3.     Если приоритет символа больше приоритета верхнего элемента стека, то символ помещается в стек,  выбирается следующий входной символ.**

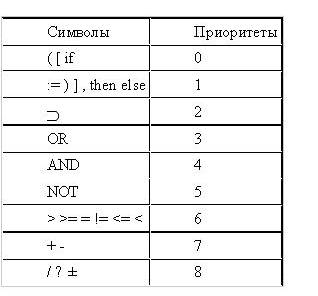
**4. Если приоритет входного символа меньше или равен приоритету верхнего элемента стека, то этот элемент удаляется из стека и помещается в формируемую строку, после чего сравнивается приоритеты очередного символа и нового верхнего символа.**

**Каждый раз при изменении обратной польской записи модифицируется ранг результирующего выражения.**

**6.3 Преобразование скобочных выражений в обратную польскую запись**

Существует несколько методов получения обратной польской записи. Один из наиболее эффективных методов предложен в 1960 г. голландским ученым  Дикстрой. Его метод основан на использовании *стека с приоритетами*, позволяющего изменить порядок следования знаков операций в выражении так, что получается обратная польская запись. Простейший вариант этого метода применим только к простым арифметическим и логическим выражениям, содержащим переменные, знаки арифметических и логических операций, знаки операций отношения и круглые скобки.

Каждому символу, входящему в выражение, присваивается приоритет (таблица ниже). Для знаков операций приоритеты возрастают в порядке, обратном старшинству операций. Скобки имеют низший приоритет.



Арифметическое или логическое выражение рассматривается как входная строка символов. Входная строка просматривается слева направо. Операнды переписываются в выходную строку, а знаки операций помещаются сначала в стек операций.

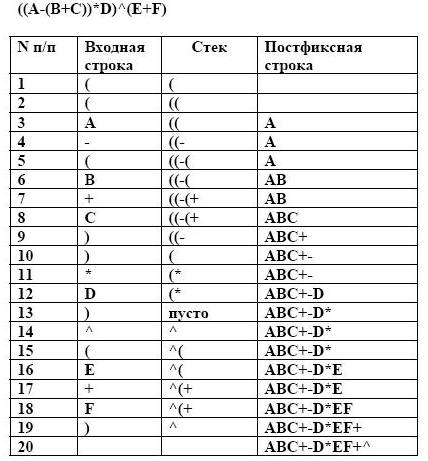
Если приоритет входного знака равен нулю или больше приоритета знака, находящегося в вершине стека, то новый знак добавляется к вершине стека. В противном случае из стека «выталкивается» и переписывается в выходную строку знак, находящийся в вершине, а также следующие за ним знаки с приоритетами, большими или равными приоритету входного знака. После этого входной знак добавляется к вершине стека. Особенности имеет лишь обработка скобок. Открывающая круглая скобка, имеющая  приоритет нуль, просто записывается в вершину стека и не выталкивает ни одного знака. В то же время ее не может вытолкнуть ни один знак, кроме закрывающей скобки. Закрывающая скобка имеет приоритет 1, не превосходящий приоритета любой операции. Поэтому появление закрывающей скобки вызывает выталкивание всех знаков до ближайшей открывающей скобки включительно. В стек закрывающая скобка не записывается. Открывающая и закрывающая скобки как бы взаимно уничтожаются и в выходную строку не переносятся. После просмотра всех символов входной строки происходит выталкивание всех оставшихся в стеке символов и дописывание их к выходной строке.

Ниже приведены два примера преобразования инфиксных выражений в обратные польские.

Пример 1.



 Пример 2.



Преобразование инфиксного выражения в префиксную запись осуществляется сканированием инфиксного выражения справа налево.

**6.4 Вычисление выражений в виде обратной польской записи**

В результате последовательного выполнения этого правила будут выполнены все операции, имеющиеся в выражении, и запись сократится до одного элемента — результата вычисления выражения.

Простое арифметическое выражение с вещественными переменными **а + b × с - d / (а + b)** можно представить в виде обратной польской записи: **а b c × + d a b + / -.**

Рассматриваемый на каждом шаге процесса элемент строки отмечен квадратными скобками. В третьей графе таблицы записаны соответствующие действия, а в четвертой графе — эквивалентные команды трехадресной машины.

ВЫЧИСЛЕНИЕ ВЫРАЖЕНИЯ В ОБРАТНОЙ ПОЛЬСКОЙ ЗАПИСИ

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  |  |  |  | | --- | --- | --- | --- | | № | Состояние строки | Действие | Машинная команда | | 1 | 2 | 3 | 4 | | 1 | [*a*] *b* *c* × + *d* *a* *b* + / - | Просмотреть следующий элемент | — | | 2 | *a* [*b*] *c* × + *d* *a* *b* + / - | Просмотреть следующий элемент | — | | 3 | *a* *b* [*c*] × + *d* *a* *b* + / - | Просмотреть следующий элемент | — | | 4 | *a* *b* *c* [×] + *d* *a* *b* + / - | *r*1 := *b* × *c* | × *b* *c* *r*1 | | 5 | *a* *r*1 [+] *d* *a* *b* + / - | *r*1 := *a* + *r*1 | + *a* *r*1 *r*1 | | 6 | *r*1 [*d*] *a* *b* + / - | Просмотреть следующий элемент | — | | 7 | *r*1 *d* [*a*] *b* + / - | Просмотреть следующий элемент | — | | 8 | *r*1 *d* *a* [*b*] + / - | Просмотреть следующий элемент | — | | 9 | *r*1 *d* *a* *b* [+] / - | *r*2 := *a* + *b* | + *a* *b* *r*2 | | 10 | *r*1 *d* *r*2 [/] - | *r*2 := *d* / *r*2 | / *d* *r*2 *r*2 | | 11 | *r*1 *r*2 [-] | *r*1 := *r*1 - *r*2 | - *r*1 *r*2 *r*1 | | 12 | *r*1 | — | — | |

Результат выполнения операции фиксируется в виде рабочей переменной вида *rj*. После очередной операции рабочая переменная *r*1 или *r*2 вычеркивается, освободившуюся рабочую переменную можно использовать вновь для записи результата операции. Использование каждый раз свободной рабочей переменной с минимальным номером экономит количество занятых рабочих переменных.