1. ТИПЫ ДАННЫХ, АБСТРАКТНЫЕ ТИПЫ ДАННЫХ И СТРУКТУРЫ ДАННЫХ

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 т.п.

Абстрактный тип данных – это математическая модель плюс различные операторы, определенные в рамках этой модели. Можно разрабатывать алгоритм в терминах абстрактных типов данных, но для реализации алгоритма в конкретном языке программирования необходимо найти способ представления АТД, в терминах типов данных и операторов, поддерживаемых данным языком программирования.

Для представления АТД используются **структуры данных**, которые представляют собой набор переменных, возможно, различных типов данных, объединенных определенным образом.

Структуры данных, применяемые в алгоритмах, могут быть чрезвычайно сложными. В результате выбор правильного представления данных часто служит ключом к удачному программированию и может в большей степени сказываться на производительности программы, чем детали используемого алгоритма.

Базовым строительным блоком структуры данных является ячейка, которая предназначена для хранения значения определенного базового или составного типа данных. Структуры данных создаются путем задания имен совокупностям (агрегатам) ячеек как представителей (т.е. указателей) других ячеек.

2. КЛАССИФИКАЦИЯ СТРУКТУР ДАННЫХ. ПОНЯТИЕ ФИЗИЧЕСКОЙ И ЛОГИЧЕСКОЙ, ПРОСТОЙ И ИНТЕГРИРОВАННОЙ СТРУКТУР

Классификация структур данных

Простые базовые структуры	Статические структуры	Полустатические структуры	Динамические структуры	Файловые структуры
Числовые; Символьные; Логические; Перечисление; Интервал; Указатели.	Векторы; Массивы; Множества; Записи; Таблицы;	Стеки; Очереди; Деки; Строки.	Линейн. связанные списки; Разветвленные связанные списки; Деревья; Графы.	Последоват.; Прямого доступа; Комбинированного доступа; Организованного разделами

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

Рассмотрение структуры 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, 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, физ

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

Различают **простые** (базовые, примитивные) структуры (типы) данных и **интегрированные** (структурированные, композитные, сложные). **Простыми** называются такие структуры данных, которые не могут быть расчленены на составные части, большие, чем биты. С точки зрения физической структуры важным является то обстоятельство, что в данной машинной архитектуре, в данной системе программирования мы всегда можем заранее сказать, каков будет размер данного простого типа и какова структура его размещения в памяти. С логической точки зрения простые данные являются неделимыми единицами. **Интегрированными** называются такие структуры данных, составными частями которых являются другие структуры данных - простые или в свою очередь интегрированные. Интегрированные структуры данных конструируются программистом с использованием средств интеграции данных, предоставляемых языками программирования.

По характеру упорядоченности элементов структуры данных делятся на линейные и нелинейные структуры. В зависимости от характера взаимного расположения элементов в памяти линейные структуры можно разделить на структуры с последовательным распределением элементов в памяти (векторы, строки, массивы, стеки, очереди) и структуры с произвольным связным распределением элементов в памяти (односвязные, двусвязные списки).

Понятие "структуры данных" тесно связано с понятием "типы данных". Любые данные характеризуются своими типами.

Информация по каждому типу однозначно определяет:

1) структуру хранения данных указанного типа

2) множество допустимых значений, которые может;

3) множество допустимых операций, которые применимы к объекту .

3. СВЯЗНЫЕ И НЕСВЯЗНЫЕ СТРУКТУРЫ ДАННЫХ, ПРИВЕСТИ ПРИМЕРЫ. СТАТИЧЕЧКИЕ, ПОЛУСТАТИЧЕСКИЕ И ДИНАМИЧЕСКИЕ СТРУКТУРЫ, ПРИВЕСТИ КЛАССИФИКАЦИЮ

В зависимости от отсутствия или наличия явно заданных связей между элементами данных следует различать **несвязные** структуры (векторы, массивы, строки, стеки, очереди) и **связные** структуры (связные списки).

Весьма важный признак структуры данных - ее изменчивость - изменение числа элементов и (или) связей между элементами 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**, **полустатические**, **динамические**. Базовые структуры данных, статические, полустатические и динамические характерны для оперативной памяти и часто называются оперативными структурами. Файловые структуры соответствуют структурам данных для внешней памяти.

Классификация структур данных

Простые базовые структуры	Статические структуры	Полустатические структуры	Динамические структуры	Файловые структуры
Числовые; Символьные; Логические; Перечисление; Интервал; Указатели.	Векторы; Массивы; Множества; Записи; Таблицы;	Стеки; Очереди; Деки; Строки.	Линейн. связанные списки; Разветвленные связанные списки; Деревья; Графы.	Последоват.; Прямого доступа; Комбиниро-ванного доступа; Организован-ного разделами

По характеру упорядоченности элементов структуры данных делятся на линейные и нелинейные структуры. В зависимости от характера взаимного расположения элементов в памяти линейные структуры можно разделить на структуры с последовательным распределением элементов в памяти (векторы, строки, массивы, стеки, очереди) и структуры с произвольным связным распределением элементов в памяти (односвязные, двусвязные списки).

Понятие "структуры данных" тесно связано с понятием "типы данных". Любые данные характеризуются своими типами.

Информация по каждому типу однозначно определяет:

1) структуру хранения данных указанного типа

2) множество допустимых значений, которые может;

3) множество допустимых операций, которые применимы к объекту .

4. ВИДЫ ПАМЯТИ. ССЫЛОЧНЫЙ ТИП ДАННЫХ В ЯЗЫКЕ DELPHI, ЕГО ОБЪЯВЛЕНИЕ И СИТУАЦИИ ПРИМЕНЕНИЯ

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. Альтернативный способ – **динамическое** управление памятью, которое осуществляется во время выполнения программы.

Если необходимый для программы объем памяти известен заранее, то можно использовать статические переменные, их компилятор может обработать без выполнения программы только на основании статического текста программы. Более рациональный подход связан с использованием динамической памяти. Это оперативная память, предоставляемая программе в ходе ее выполнения, за вычетом сегмента данных, стека и тела программы. При экономном подходе к памяти заранее не резервируется ее максимальный объем для размещения данных, а предварительно определяется их тип, и каждый раз по мере необходимости создаются новые данные. Переменные, которые создаются и уничтожаются в процессе выполнения программы, называются динамическими. Кроме экономии памяти, для ряда задач даже невозможно заранее определить требуемый объем памяти, тогда динамическое распределение – единственно возможный подход. Динамическая память широко используется при работе с графическими и звуковыми средствами компьютера.

Для организации динамической памяти используется тип данных, называемый **указателем или ссылочным** типом данных. Значением указателя является адрес области памяти, содержащей переменную заранее определенного типа. В этом случае указатели называются типизированными. Для указателей область памяти выделяется статически, а для переменных, на которые они указывают, – динамически

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.

В программе на языке высокого уровня указатели могут быть типизированными и нетипизированными**. При объявлении типизированного указателя** определяется и тип объекта в памяти, адресуемого этим указателем. Для объявления переменных ссылочного типа используется символ « ^ », после которого указывается тип динамической (базовой) переменной.

Type<имя_типа>=^ <базовый тип>;

Var<имя_переменной>: <имя_типа>; или <имя_переменной>: ^< базовый тип>:

Нетипизированный указатель (тип pointer в Pascal) служит для представления адреса, по которому содержатся данные неизвестного типа.

Зарезервированное слово **Nil** обозначает константу ссылочного типа, которая ни на что не указывает.

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.

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

5. ОПЕРАЦИИ НАД УКАЗАТЕЛЯМИ В ЯЗЫКЕ DELPHI, ПРИВЕСТИ ПРИМЕРЫ

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.

К указателю можно прибавить целое число или вычесть из него целое число. 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.

6. ТИП ДАННЫХ ЗАПИСЬ. ЕГО НАЗНАЧЕНИЕ, ОБЪЯВЛЕНИЕ, ИСПОЛЬЗОВАНИЕ ЗАПИСИ БЕЗ ВАРИАНТНОЙ ЧАСТИ

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.

```
Type
    <имя типа> = record
        < идентификатор поля>:< тип компонента>;
            …
        < идентификатор поля>:< тип компонента>;
    End;
Var
    <идентификатор, …> : < имя типа >;
```

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.

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

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

Запись - конечное упорядоченное множество полей, характеризующихся различным типом данных. Записи являются чрезвычайно удобным средством для представления программных моделей реальных объектов предметной области, ибо, как правило, каждый такой объект обладает набором свойств, характеризуемых данными различных типов. В памяти эта структура может быть представлена в одном из двух видов:

а) в виде последовательности полей, занимающих непрерывную область памяти. При такой организации достаточно иметь один указатель на начало области, и смещение относительно начала. Это дает экономию памяти, но лишнюю трату времени на вычисление адресов полей записи.

б) в виде связного списка с указателями на значения полей записи. При такой организации имеет место быстрое обращение к элементам, но очень неэкономичный расход памяти для хранения.

Операции над записями. Важнейшей операцией для записи является операция доступа к выбранному полю записи - операция квалификации.

Если в программе обратиться к полю, которое отсутствует в текущем состоянии записи, то оператор будет выполняться, но результат будет бессмысленным. Пользователь должен сам прослеживать состояние записи.

Один и то же идентификатор поля не может дважды использоваться при описании записи, даже в альтернативных вариантах. Это связано с особенностями работы компьютера (интерпретатора) – с построением таблиц идентификаторов.

Существуют ограничения на использование полей упакованных записей.

Записи без вариантной части содержат только общую часть.

Для обработки записей полей удобно их объединять в массивы и обрабатывать целые массивы записей. Такое объединение позволяет хранить анкетные данные для большого количества людей а также выделять и обрабатывать из них нужную информацию.

7. ТИП ДАННЫХ ЗАПИСЬ. ЕГО НАЗНАЧЕНИЕ, ОБЪЯВЛЕНИЕ, ИСПОЛЬЗОВАНИЕ ЗАПИСИ С ВАРИАНТНОЙ ЧАСТЬЮ

Запись - конечное упорядоченное множество полей, характеризующихся различным типом данных.

Записи с вариантами. В ряде прикладных задач программист может столкнуться с группами объектов, чьи наборы свойств перекрываются лишь частично. Обработка таких объектов производится по одним и тем же алгоритмам, если обрабатываются общие свойства объектов, или по разным - если обрабатываются специфические свойства. Можно описать все группы единообразно, включив в описание все наборы свойств для всех групп, но такое описание будет неэкономичным с точки зрения расходуемой памяти и неудобным с логической точки зрения. Если же описать каждую группу собственной структурой, теряется возможность обрабатывать общие свойства по единым алгоритмам.

Для задач подобного рода развитые языки программирования предоставляют в распоряжение программиста записи с вариантами. **Запись с вариантами** состоит из двух частей. В первой части описываются поля, общие для всех групп объектов, моделируемых записью. Среди этих полей обычно бывает поле, значение которого позволяет идентифицировать группу, к которой данный объект принадлежит и, следовательно, какой из вариантов второй части записи должен быть использован при обработке. Вторая часть записи содержит описания непересекающихся свойств - для каждого подмножества таких свойств - отдельное описание. Язык программирования может требовать, чтобы имена полей-свойств не повторялись в разных вариантах (PASCAL), или же требовать именования каждого варианта (C). В первом случае идентификация поля, находящегося в вариантной части записи при обращении к нему ничем не отличается от случая простой записи:

< имя переменной-записи >.< имя поля >

Во втором случае идентификация немного усложняется:

< имя переменной-записи >.< имя варианта >.< имя поля >

Вариантная часть начинается со слова case и описывает несколько вариантов структуры записи а) case I: integer of …б) case integer of …

В каждый момент времени один из вариантов структуры является активным в зависимости от значения и признака варианта. Признак варианта описывается непосредственно после слова case. Признак является самим полем общей части записи. Поле признака может отсутствовать. В этом случае указывается точный идентификатор типа и нужно следить, какой вариант записи является активным. Список полей каждого варианта заключается в скобки.

Вариантная часть может быть вложенной в другую.

Выделение памяти для записи с вариантами. Под запись с вариантами выделяется в любом случае объем памяти, достаточный для размещения самого большого варианта. Если же выделенная память используется для меньшего варианта, часть ее остается неиспользуемой. Общая для всех вариантов часть записи размещается так, чтобы смещения всех полей относительно начала записи были одинаковыми для всех вари- антов. Очевидно, что наиболее просто это достигается размещением общих полей в начале записи, но это не строго обязательно. Вариантная часть может и "вклиниваться" между полями общей части. Поскольку в любом случае вариантная часть имеет фиксированный (максимальный) размер, смещения полей общей части также останутся фиксированными.

Поля записи последовательны в соответствии с объявлением.

8. ИСПОЛЬЗОВАНИЕ В ЗАПИСЯХ ОПЕРАТОРА ПРИСОЕДИНЕНИЯ WITH. ЗАПИСИ-КОНСТАНТЫ

При работе с полями в их составные имена нужно записывать путь к полю через все уровни иерархии, начиная с полного имени записи. Зачастую работа с такими длинными именами неудобна, а программа громоздкая. Для сокращения составного имени поля используют with

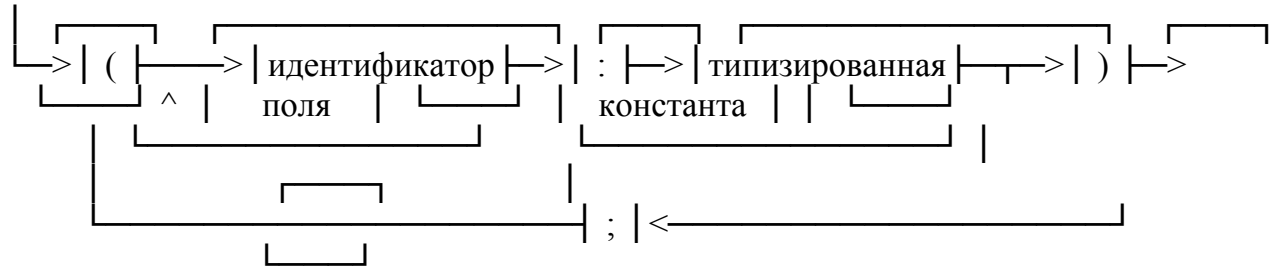
with <переменная> do
 begin
  <операторы>
 end;

В операторе указывается список переменных типа real. With облегчает доступ к полям записи и минимизирует повторные записи. Внутри оператора вложенного в оператор with к полям этих записей можно обращаться как к обычным переменным.

Адрес переменной типа record вычисляется до выполнения оператора with.

Описание константы типа запись содержит идентификатор и значение каждого поля, заключенные в скобки и разделенные точками с запятой.

константа-запись



Приведем несколько примеров констант-записей:

```
type
  Point  = record
           x,y: real;
         end;
  Vector = array[0..1] of Point;
  Month  =
     (Jan,Feb,Mar,Apr,May,Jun,Jly,Aug,Sep,Oct,Nov,Dec);
  Date   = record
           d: 1..31; m: Month; y: 1900..1999;
         end;
const
  Origin  : Point = (x: 0.0; y: 0.0);
  Line    : Vector = ((x: -3.1; y: 1.5),(x: 5.8; y: 3.0));
  SomeDay : Date = (d: 2; m: Dec; y: 1960);
```

Поля должны указываться в том же порядке, как они следуют в описании типа запись. Если запись содержит поля файлового типа, то для этого типа запись нельзя описать константу. Если запись содержит вариант, то можно указывать только поля выбранного варианта. Если вариант содержит поле признака, то его значение должно быть определено.

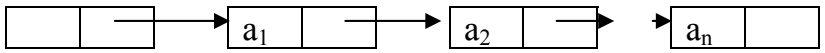
9. ОБЪЯВЛЕНИ И ПРЕДСТАВЛЕНИЕ ДИНАМИЧЕСКОЙ ЦЕПОЧКИ(ОДНОНАПРАВЛЕННОГО СПИСКА). АЛГОРИТМ И ПРОЦЕДУРА ФОРМИРОВАНИЯ ЦЕПОЧКИ(ОДНОНАПРАВЛЕННОГО СПИСКА)

Однонаправленный связанный список представляет собой динамическую структуру данных, число элементов которой может изменяется по мере того как переменная помещаются в список или удаляются из него. В отличии от массива, чей размер неизменен, список не является наперед заданным набором.

Для реализации однонаправленных списков используются указатели, связывающие последовательные элементы списка. Эта реализация освобождает от использования непрерывной области памяти для хранения списка и , следовательно, от необходимости перемещения элементов списка при вставке или удалении элементов. Однако ценой за это удобство становится дополнительная память 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 a1,a2,….,an то для i=1,2,…,n-1 ячейка, содержащая элемент a_i, имеет также указатель на ячейку, содержащую элемент a_{i+1} . Ячейка, содержащая элемент a_n , имеет указатель nil (нуль). Имеется также ячейка header (заголовок), которая указывает на ячейку, содержащую a1. 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. Связанный список



Header

Список

Объявление звена цепочки:

```
Type                                    Adrcled:adr;
  Adr=^Zveno;                         End;
   Zveno=record                     Var adr1, adrcv, adr;
        Element: char;
```

Алгоритм формирования цепочки:

1. Отвести область памяти для очередного звена. Его адрес занести в поле adrcled текущего звена.
2. Новое звено сделать текущим, занеся его адрес в указатель текущего звена adrzv.
3. В поле element текущего звена занести очередной символ.
4. В поле adrcled текущего звена занести nil.
5. Прочитать следующий символ исходного текста.
6. Повторить шаги алгоритма, начиная с первого.

Предварительно перед выполнением шагов 1-6 необходимо сформировать заглавное звено.

```
…                                   While sinv<>'.' Do
Var adr1, adrzv: adr;               Begin
Sinv:char;                          New(adrzv^.adrcled);
Begin                               Adrzv:=adrzv^.adrcled;
New(adr1);                          Adrzv^.element:=sinv;
Adrzv:=adr1;                        Adrzv^.adrcled:=nil;
Adrzv^.adrcled:=nil;                Read(sinv);
Read(sinv);                         End;
```

Динамическая цепочка является частным случаем линейного однонаправленного списка. Принцип построения и работы с линейным списком аналогичны принципам работы с динамической цепочкой. Недостатком однонаправленного списка является то, что по нему можно двигаться только в одном направлении от заглавного звена к последнему, что замедляет работу с заданной структурой.

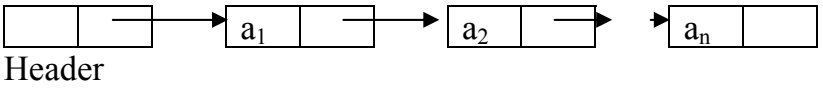
10. **ОБЪЯВЛЕНИ И ПРЕДСТАВЛЕНИЕ ДИНАМИЧЕСКОЙ ЦЕПОЧКИ(ОДНОНАПРАВЛЕННОГО СПИСКА). АЛГОРИТМ И ПРОЦЕДУРА УДАЛЕНИЯ ЗВЕНА ЦЕПОЧКИ(ОДНОНАПРАВЛЕННОГО СПИСКА)**

Однонаправленный связанный список представляет собой динамическую структуру данных, число элементов которой может изменяется по мере того как переменная помещаются в список или удаляются из него. В отличии от массива, чей размер неизменен, список не является наперед заданным набором.

Для реализации однонаправленных списков используются указатели, связывающие последовательные элементы списка. Эта реализация освобождает от использования непрерывной области памяти для хранения списка и , следовательно, от необходимости перемещения элементов списка при вставке или удалении элементов. Однако ценой за это удобство становится дополнительная память 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 a1,a2,….,an то для i=1,2,…,n-1 ячейка, содержащая элемент a_i, имеет также указатель на ячейку, содержащую элемент a_{i+1} . Ячейка, содержащая элемент a_n , имеет указатель nil (нуль). Имеется также ячейка header (заголовок), которая указывает на ячейку, содержащую a1. 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. Связанный список



Список

Динамическая цепочка является частным случаем линейного однонаправленного списка. Принцип построения и работы с линейным списком аналогичны принципам работы с динамической цепочкой. Недостатком однонаправленного списка является то, что по нему можно двигаться только в одном направлении от заглавного звена к последнему, что замедляет работу с заданной структурой.

Объявление звена цепочки:

```
Type
  Adr=^Zveno;
    Zveno=record
          Element: char;
          Adrcled:adr;
    End;
Var adr1, adrcv, adr;
```

Процедура удаления:

```
Procedure udal (zv:adr);
Var a:adr;
Begin
A:= zv^.adrcled;
Zv^.adrcled:=zv^.adcled^.adcled;
Dispose(a);
End;
```

Исключаемый элемент удобно задавать при помощи ссылки на то звено за которым следует этот элемент. При этом считать что е6сли какое-либо звено существует но на него нет ссылки из другого звена то оно не доступно при последовательном переборе звеньев цепочки. И оно считается отсутствующим.

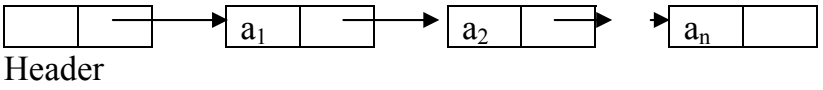
11. ОБЪЯВЛЕНИ И ПРЕДСТАВЛЕНИЕ ДИНАМИЧЕСКОЙ ЦЕПОЧКИ (ОДНОНАПРАВЛЕННОГО СПИСКА). АЛГОРИТМ И ПРОЦЕДУРА ВСТАВКИ ЗВЕНА В ЦЕПОЧКУ ПОСЛЕ ЗАДАННОГО

Однонаправленный связанный список представляет собой динамическую структуру данных, число элементов которой может изменяется по мере того как переменная помещаются в список или удаляются из него. В отличии от массива, чей размер неизменен, список не является наперед заданным набором.

Для реализации однонаправленных списков используются указатели, связывающие последовательные элементы списка. Эта реализация освобождает от использования непрерывной области памяти для хранения списка и , следовательно, от необходимости перемещения элементов списка при вставке или удалении элементов. Однако ценой за это удобство становится дополнительная память 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 a1,a2,….,an то для i=1,2,…,n-1 ячейка, содержащая элемент a_i, имеет также указатель на ячейку, содержащую элемент a_{i+1} . Ячейка, содержащая элемент a_n , имеет указатель nil (нуль). Имеется также ячейка header (заголовок), которая указывает на ячейку, содержащую a1. 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. Связанный список



Список

Динамическая цепочка является частным случаем линейного однонаправленного списка. Принцип построения и работы с линейным списком аналогичны принципам работы с динамической цепочкой. Недостатком однонаправленного списка является то, что по нему можно двигаться только в одном направлении от заглавного звена к последнему, что замедляет работу с заданной структурой.

Объявление звена цепочки:

```
Type
  Adr=^Zveno;
    Zveno=record
          Element: char;
          Adrcled:adr;
    End;
Var adr1, adrcv, adr;
```

Алгоритм вставки

1. Создать новую динамическую переменную(запись типа zveno, которая будет представлять вставляемое звено)
2. В поле element этой переменной занести вставляемый элемент
3. В поле adrcled этой переменной занести ссылку, взятую из поля adcled предшествующего звена
4. В поле adcled предшествующего звена занести ссылку на вставляемое звено

```
Procedure vstav(zv: adr;el:char);
Var
  Q;adr;
Begin
  New(Q);
  Q^.element:-el;
  Q^.adrcled:=zv^.adrcled;
  Zv^.adrcled:=q;
End;
```

12. СТРУКТУРА ЗВЕНА ДВУНАПРВЛЕННОГО СПИСКА. ДВА ВИДА ДВУНАПРАВЛЕННЫХ СПИСКОВ. АЛГОРИТМ И ПРОЦЕДУРА ВСТАВКИ ЭЛЕМЕНТА В ДВУНАПРВЛЕННЫЙ СПИСОК.

Каждое звено двунаправленного списка содержит 2 поля ссылочного типа, значением одного из них является ссылка на следующее звено списка, а второго поля ссылка на предыдущее звено списка.

```
Type
  Adr2=^zveno2;
   Zveno2=record
              Adrcled:adr2;
              Adrpred:adr2;
              Element;<тип_элемента_списка>;
    End;
```

У заглавного звена списка нет предыдущего элемента , у последнего не последующего, поэтому в поле адреса следующего последнего звена и в поле адреса предыдущего первого звена такого списка должно быть записано nil. На основе двунаправленного списка могут быть организованы кольцевые списки.

2 варианта двунаправленных списков:

В кольцевом списке значением поля адрес следующего звена является ссылка на заглавное звено списка, а адрес предыдущего первого звена – ссылка на последнее звено списка.

1-ый способ: просто реализуется вставка нового звена как в начало, так и в конец списка. При циклической обработке списка нужно проверять не является ли очередное звено заглавным. 2-ой способ: этого недостатка нет, но сложнее добавить звено в конец списка. Для такого списка нужны 2 указателя, на заглавное звено и на текущее.

Алгоритм вставки в двунаправленный список:

1. Порождение нового звена
2. Занесение вставляемого элемента в информационное поле порожденного звена
3. Занесение в поле adrcled порожденного звена ссылки на следующий элемент из звена предшествующего вставляемому
4. Занесение в поле adrpred порожденного звена ссылки на предыдущий элемент из звена следующего за вставляемым
5. Занесение в поле adpred следующего за вставляемым звена ссылки на вставляемое звено
6. Занесение в поле adrcled предшествующего звена ссылки на вставляемое звено

Процедура вставки:

```
 Procedure vstav(elem:<тип вставляемого элемента>;predzv:adr2);
  Var Q:adr2;
  Begin
    New(Q);
    Q^.element:=elem;
    Q^.adrcled:=predzv^.adrcled;
    Q^.adrpred:=prrdzv;
    Predzv^.adrcled^.adrpred:=Q;
    Predzv^.adrcled:=Q;
  End;
```

13. СТРУКТУРА ЗВЕНА ДВУНАПРАВЛЕННОГО СПИСКА. АЛГОРИТМ И ПРОЦЕДУРА УДАЛЕНИЯ ЭЛЕМЕНТА ИЗ ДВУНАПРАВЛЕННОГО СПИСКА.

Каждое звено двунаправленного списка содержит 2 поля ссылочного типа, значением одного из них является ссылка на следующее звено списка, а второго поля ссылка на предыдущее звено списка.

```
Type
  Adr2=^zveno2;
   Zveno2=record
              Adrcled:adr2;
              Adrpred:adr2;
              Element;<тип_элемента_списка>;
    End;
```

У заглавного звена списка нет предыдущего элемента , у последнего не последующего, поэтому в поле адреса следующего последнего звена и в поле адреса предыдущего первого звена такого списка должно быть записано nil. На основе двунаправленного списка могут быть организованы кольцевые списки.

В кольцевом списке значением поля адрес следующего звена является ссылка на заглавное звено списка, а адрес предыдущего первого звена – ссылка на последнее звено списка.

Алгоритм удаления из двунаправленного списка:

1. Занесение в поле adrpred следующего за удаляемым звена на предшествующее удаляемому звено из поля adrpred удаляемого звена
2. Занесение в поле adrcled предшествующего удаляемому звена ссылки на следующее за удаляемым звено из поля adcled удаляемого звена
3. Уничтожение удаляемого звена

Процедура удаления звена:

```
Procedure udalen(udzv:adr2);
 Begin
  Udzv^.adrcled^.adrpred:=udrzv^.adrpred;
  Udzv^.adrpred^.adrcled:=udzv^.adrcked;
  Dispose(udrzv);
 End;
```

14. СТРУКТУРА ЗВЕНА ДВУНАПРАВЛЕННОГО СПИСКА. АЛГОРИТМ И ПРОЦЕДУРА ПОИСКА ЭЛЕМЕНТА В ДВУНАПРАВЛЕННОМ СПИСКЕ

Каждое звено двунаправленного списка содержит 2 поля ссылочного типа, значением одного из них является ссылка на следующее звено списка, а второго поля ссылка на предыдущее звено списка.

```
Type
  Adr2=^zveno2;
   Zveno2=record
              Adrcled:adr2;
              Adrpred:adr2;
              Element;<тип_элемента_списка>;
    End;
```

У заглавного звена списка нет предыдущего элемента , у последнего не последующего, поэтому в поле адреса следующего последнего звена и в поле адреса предыдущего первого звена такого списка должно быть записано nil. На основе двунаправленного списка могут быть организованы кольцевые списки.

2 варианта двунаправленных списков:

В кольцевом списке значением поля адрес следующего звена является ссылка на заглавное звено списка, а адрес предыдущего первого звена – ссылка на последнее звено списка.

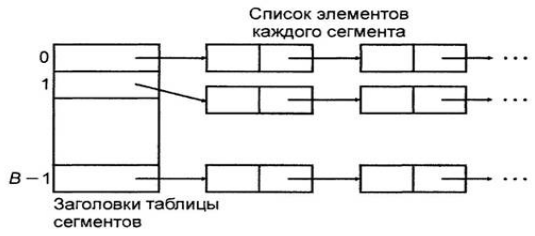
Алгоритм поиска в кольцевом двунаправленном списке:

Процедура поиска аналогична поиску элемента в цепочке. Особенность в том что в кольцевом списке формально нет последнего элемента так как каждый имеет ссылку на следующий. Это нужно учитывать при организации поиска.

Функция поиска элемента в кольцевом двунаправленном списке:

```
Function poisk(adr:adr2;elem:<тип элемента списка>;Var iskadr:adr2):boolean;
 Var
  P,Q:adr2;
  B:boolean;
 Begin
  B:=false;
  P:=adr;
  Iskadr:=nil;
  Q:=p^.adcled;
  While(P<>Q)and not B do
  Begin
    If Q^.element=elem then
    Begin
      B:=true;
      Iskadr:=Q;
    End;
    Q:=Q^.adcled;
   End;
  Poisk:=B;
End;
```

15. НАЗНАЧЕНИЕ ХЕШИРОВАНИЯ ДАННЫХ. ОТКРЫТОЕ ХЕШИРОВАНИЕ. ПРИВЕСТИ ПРИМЕР ОРГАНИЗАЦИИ ДАННЫХ



На сегодняшний день существует множество способов повышения эффективности работы с большими объемами информации. Например, для ускорения доступа к данным в таблицах можно использовать предварительное упорядочивание таблицы в соответствии со значениями ключей. При этом могут быть использованы методы поиска в упорядоченных структурах данных, что существенно сокращает время поиска данных по значению ключа. Однако при добавлении новой записи требуется переупорядочить таблицу. Потери времени на повторное упорядочивание справочника часто превышают выигрыш от сокращения времени поиска. Рассмотрим способы организации данных, лишенные указанного недостатка. Это – различные формы хеширования данных.

Открытое хеширование:

На рисунке показана базовая структура данных при открытом хешировании. Основная идея метода заключается в том, что множество данных (возможно, очень большое) разбивается на конечное число классов. Для B классов, пронумерованных от 0 до B-1, строится хеш-функция htакая, что для любого элемента x исходного множества функция h(x) принимает целочисленное значение из интервала 0, …, B-1, которое соответствует классу, которому принадлежит элемент x. Элемент x называют ключом, h(x) – хеш-значением x, а классы – сегментами. Массив (таблица сегментов), проиндексированный номерами сегментов 0, 1, … B-1, содержит заголовки для B списков. Элемент x i-го списка – это элемент исходного множества, для которого h(x)=i.

Если сегменты приблизительно равны по размеру, то в этом случае списки всех сегментов должны быть наиболее короткими при данном числе сегментов. Если исходное множество состоит из N элементов, тогда средняя длина списков будет N/B элементов. Если удается оценить величину N и выбрать B как можно ближе к этой величине, то в каждом списке будет один-два элемента. Тогда время выполнения операций с данными будет малой постоянной величиной, зависящей отN или от B. Однако не всегда ясно, как выбрать хеш-функцию htак, чтобы она примерно поровну распределяла элементы исходного множества по всем сегментам.

Идеальной хеш-функцией является такая, которая для любых двух неодинаковых ключей выдает неодинаковые адреса, т.е. k1 не =k2, следовательно h(k1)не =h(k2)(условие1)

Однако подобрать такую функцию можно в случае, если все возможные значения ключей известны заранее. Такая организация данных носит название «совершенное хеширование». Если заранее не определено множество значений ключей, и длина таблицы ограничена, подбор совершенной функции затруднителен. Поэтому часто используют хеш-функции, которые не гарантируют выполнение условия (1).

Пример 1. Хеш-функция, определенная на символьных строках

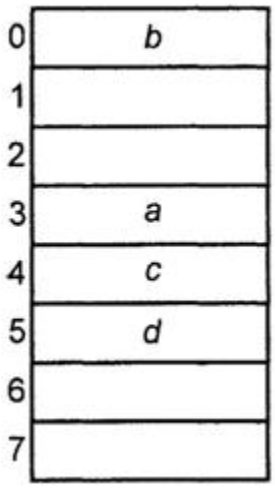
```
Program hesh;
  Var
    i,B : integer;
   s :string[10];
       Function h(x : string[10]): integer;
          Vari, sum: integer;
    Begin
      sum:=0;
     Ffor i:=1 to 10 do
        sum:=sum+ord(x[i]);
        h:=summodB;
      End;
   Begin
      Writeln ('Введите число классов B');
      Readln(B);
      Writeln ('Введитеключ');
      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. В языке Pascal есть встроенная функция ord(c), которая возвращает целочисленный код символа c. Таким образом, если x– это ключ, тип данных ключей определен как строка символов, тогда можно использовать хеш-функцию, код которой приведен ниже. В этой функции суммируются все целочисленные коды символов, результат суммирования делится на B и берется остаток от деления, который будет целым числом из интервала от 0 до B-1.

16. НАЗНАЧЕНИЕ ХЕШИРОВАНИЯ ДАННЫХ. ЗАКРЫТОЕ ХЕШИРОВАНИЕ ДАННЫХ. ПРИВЕСТИ ПРИМЕР ОРГАНИЗАЦИИ ДАННЫХ

На сегодняшний день существует множество способов повышения эффективности работы с большими объемами информации. Например, для ускорения доступа к данным в таблицах можно использовать предварительное упорядочивание таблицы в соответствии со значениями ключей. При этом могут быть использованы методы поиска в упорядоченных структурах данных, что существенно сокращает время поиска данных по значению ключа. Однако при добавлении новой записи требуется переупорядочить таблицу. Потери времени на повторное упорядочивание справочника часто превышают выигрыш от сокращения времени поиска. Рассмотрим способы организации данных, лишенные указанного недостатка. Это – различные формы хеширования данных.

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 *x* 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 h1(x),h2(x)… куда можно поместить элемент *x*. Каждое из этих местоположений последовательно проверяется, пока не будет найдено свободное. Если свободных сегментов нет, то таблица заполнена, и элемент *x* вставить в нее нельзя.



Здесь *B=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 hi(x)=(h(x)+i)modB. 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.

17. РАЗРЕШЕНИЕ КОЛЛИЗИЙ В СЛУЧАЕ ЗАКРЫТОГО ХЕШИРОВАНИЯ

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.

Разрешение коллизий при добавлении элементов:

$K1 \neq k2$

$H(k1)=h(k2)$

$K3 \neq k1$

$H(h3)=h(k1)$

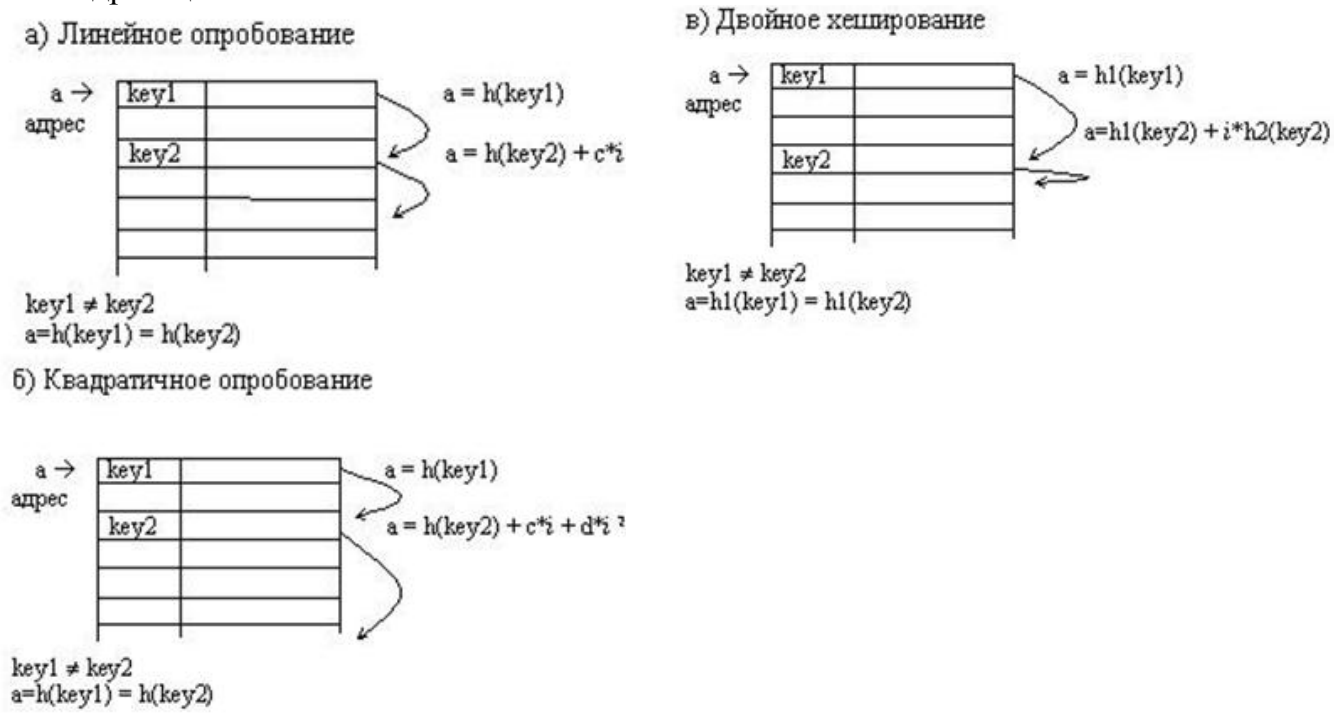
Метод открытой адресации состоит в том, чтобы, пользуясь каким-либо алгоритмом, обеспечивающим перебор элементов таблицы, просматривать их в поисках свободного места для новой записи.

Линейное опробование 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 опробование отличается от линейного тем, что шаг перебора элементов нелинейно зависит от номера попытки найти свободный элемент $a = h(key2) + c*i + d*i^2$.

Благодаря нелинейности такой адресации уменьшается число проб при большом числе ключей-синонимов. Однако даже относительно небольшое число проб может быстро привести к выходу за адресное пространство небольшой таблицы вследствие квадратичной зависимости адреса от номера попытки. На рис. показано разрешение коллизий методом открытой адресации.



18. 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. Предположим, что ключ удаляемого элемента имеет в таблице ключи-синонимы. Тогда, если за удаляемым элементом в результате разрешения коллизий были размещены элементы с другими ключами, то поиск этих элементов после удаления всегда будет давать отрицательный результат, так как алгоритм поиска останавливается на первом элементе, находящемся в состоянии свободно. Скорректировать сложившуюся ситуацию можно различными способами. Самый простой из них состоит в том, чтобы выполнять поиск элемента не до первого свободного места, а до конца таблицы. Однако такая модификация алгоритма сведет на нет весь выигрыш в ускорении доступа к данным, который достигается в результате хеширования. Другой способ сводится к тому, чтобы проследить адреса всех ключей-синонимов для ключа удаляемого элемента и при необходимости переразместить соответствующие записи в таблице. Скорость поиска после такой операции не уменьшится, но затраты времени на само переразмещение элементов могут оказаться очень значительными.

Существует подход, который свободен от перечисленных недостатков. Его суть состоит в том, что для элементов хеш-таблицы **добавляется состояние удалено**. Данное состояние в процессе поиска интерпретируется как занято, а в процессе записи как свободно.

Рассмотрим алгоритмы вставки, поиска и удаления для хеш-таблицы, имеющей три состояния элементов.

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

Алгоритм поиска для хеш-таблицы, имеющей три состояния, практически не отличается от алгоритма поиска без учета удалений. Разница заключается в том, что при организации самой таблицы необходимо отмечать свободные и удаленные элементы. Это можно сделать, зарезервировав два значения ключевого поля. Другой вариант реализации может предусматривать введение дополнительного поля, в котором фиксируется состояние элемента. Длина такого поля может составлять всего два бита, что вполне достаточно для фиксации одного из трех состояний.

Очевидно, что по мере заполнения хеш-таблицы будут происходить коллизии, и в результате их разрешения методами открытой адресации очередной адрес может выйти за пределы адресного пространства таблицы. Чтобы это явление происходило реже, можно пойти на увеличение длины таблицы по сравнению с диапазоном адресов, выдаваемым хеш-функцией. С одной стороны это

приведет к сокращению числа коллизий и ускорению работы с хеш-таблицей, а с другой – к нерациональному расходованию адресного пространства. Даже при увеличении длины таблицы в два раза по сравнению с областью значений хеш-функции нет гарантии того, что в результате коллизий адрес не превысит длину таблицы. При этом в начальной части таблицы может оставаться достаточно свободных элементов. Поэтому на практике используют циклический переход к началу таблицы.

Рассмотрим этот способ на примере **метода линейного опробования.** При вычислении адреса очередного элемента можно в качестве адреса взять остаток от целочисленного деления адреса на длину таблицы n.

Вставка

i = 0
a = (h(key) + c*i) mod n
Если t(a) = свободно или t(a) = удалено, то t(a) = key, записать элемент, стоп элемент добавлен
i = i + 1, перейти к шагу 2

В данном алгоритме не учитывается возможность многократного превышения адресного пространства. Более корректным будет алгоритм, использующий сдвиг адреса на 1 элемент в случае каждого повторного превышения адресного пространства. Это увеличивает вероятность поиска свободных элементов в случае повторных циклических переходов к началу таблицы.

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
i = i + 1, перейти к шагу 2

При большой заполненности таблицы возникают частые коллизии и циклические переходы в ее начало. При неудачном выборе хеш-функции происходят аналогичные явления. В наихудшем случае при 100% заполнении таблицы алгоритмы циклического поиска свободного места приведут к зацикливанию. Поэтому при использовании хеш-таблиц необходимо стараться избегать очень плотного ее заполнения. Обычно длину таблицы выбирают из расчета двукратного превышения предполагаемого максимального числа записей. Однако не всегда можно правильно оценить требуемую длину таблицы, поэтому в случае большой заполненности таблицы может понадобиться рехеширование. В этом случае увеличивают длину таблицы, изменяют хеш-функцию и переупорядочивают данные.

Оценку плотности заполнения таблицы целесообразно выполнять косвенным образом: по числу коллизий во время одной вставки. Достаточно определить некоторый порог числа коллизий, в случае превышения которого следует провести рехеширование. Кроме того, такая проверка гарантирует невозможность зацикливания алгоритма в случае повторного просмотра элементов таблицы.

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.

19. АБСТРАКТНЫЙ ТИП ДАННЫХ «ОЧЕРЕДЬ». АЛГОРИТМ И ПРОЦЕДУРА ЗАНЕСЕНИЯ ЭЛЕМЕНТА В 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 firstinfirstout: первым вошел – первым вышел). Две основные операции, которые определены для работы с очередью: вставка и извлечение элементов.

```
Type Adres1 = ^queue;{Указатель на элемент очереди}
   Queue = record
      element :<тип элемента очереди>;
      adrcled,left,right :adres1;{Указатели на следующий элемент, голову и хвост очереди}
end;
VarQ :adres1; {Указатель на заголовок очереди}
```

Многопоточные очереди (multiheadedqueues). Элементы в нее, как обычно, добавляются в конец очереди, но очередь имеет несколько потоков (frontend) или голов (heads).

Циклическая очередь. 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.

Приоритетные очереди. Каждый элемент в приоритетной очереди (priorityqueue) имеет связанный с ним приоритет.

Для добавления элемента в очередь используется указатель right после чего его значение изменяется и он указывает на последний занесенный элемент. Выборка осуществляется с помощью указателя left после чего он также изменяется и указывает на следующий элемент очереди. Если в очереди 1 элемент то left=right. Такое равенство можно использовать как признак окончания очереди при последовательном выборе элементов из нее.

Алгоритм занесения элемента в очередь:

1. Создание нового звена
2. Занесение в последнее звено адреса нового звена
3. Занесение nil в поле adcled нового звена
4. Занесение элемента в информационное поле нового звена
5. Созданное звено сделать концом очереди

Процедура занесения элемента в очередь:

```
Procedure dobavl(var right:adres1;el:<тип элемента очереди>);
 Var
  Q:adres1;
 Begin
  New(Q);
  Right^.adrcled:=Q;
  Q^.adrcled:=nil;
  Q^.element:=el;
  Right:=Q;
 End;
```

20. АБСТРАКТНЫЙ ТИП «ОЧЕРЕДЬ». АЛГОРИТМ И ПРОЦЕДУРА УДАЛЕНИЯ ЭЛЕМЕНТА ИЗ ОЧЕРЕДИ

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 firstinfirstout: первым вошел – первым вышел). Две основные операции, которые определены для работы с очередью: вставка и извлечение элементов.

```
Type Adres1 = ^queue;{Указатель на элемент очереди}
   Queue = record
      element :<тип элемента очереди>;
      adrcled,left,right :adres1;{Указатели на следующий элемент, голову и хвост очереди}
end;
VarQ :adres1; {Указатель на заголовок очереди}
```

Многопоточные очереди (multiheadedqueues). Элементы в нее, как обычно, добавляются в конец очереди, но очередь имеет несколько потоков (frontend) или голов (heads).

Циклическая очередь. 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.

Приоритетные очереди. Каждый элемент в приоритетной очереди (priorityqueue) имеет связанный с ним приоритет.

Для добавления элемента в очередь используется указатель right после чего его значение изменяется и он указывает на последний занесенный элемент. Выборка осуществляется с помощью указателя left после чего он также изменяется и указывает на следующий элемент очереди. Если в очереди 1 элемент то left=right. Такое равенство можно использовать как признак окончания очереди при последовательном выборе элементов из нее.

Алгоритм выбора элемента из очереди:

1. Чтение значения элемента из начала очереди
2. Запоминание ссылки на начало очереди
3. Исключение первого звена из начала очереди
4. Уничтожение первого звена

Процедура удаления элемента из очереди:

```
Procedure udal(var left:adres1;var elem:<тип элемента очереди>);
 Var
  Q:adres1;
 Begin
  Elem:=left^.element;
  Q:=left;
  Left:=left^.adrcled;
  Dispose(Q);
 End;
```

21. АБСТРАКТНЫЙ ТИП ДАННЫЙ «СТЕК». АЛГОРИТМ И ПРОЦЕДУРА ЗАНЕСЕНИЯ ЭЛЕМЕНТА В СТЕК С ПОМОЩЬЮ УКАЗАТЕЛЕЙ

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).

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). 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;
```

Если стек пуст, то значение указателя st равно nil.

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

Запись элементов с тек

```
Procedure WriteStack (var x:stack);
  Begin
    While x.top>0 then do
    Begin
      Readln(x.elements[x.top]);
      x.top:=x.top-1;
    end;
  end;
```

Извлечение элемента из стека

```
Procedure print(var x:stack);
 begin
  while x.top<=5 do
  begin
    writeln(x.elements[x.top]);
    x.top:=x.top+1;
  end;
 end;
```

Проверка полноты стека

```
Function tops(var x:stack):boolean;
 Begin
  If x.top<1 then tops:=true
             Else tops:=false;
 End;
```

Проверка пустоты стека

```
Function empty(var x:stack):boolean;
Begin
If x.top>5 then empty :=thrue
           Else empty:= false;
End;
```

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 adres1 = ^stack; {Указатель на элемент стека}
   Stack = record
   Element:<тип элемента стека>;
   Adrcled:adres1;{Указатели на следующий элемент  очереди}
  End;
Var Q :adres1;
```

Алгоритм занесения элемента в стек

1. Создание нового звена
2. Занесение Q нового звена элемента
3. Занесение в новое звено адреса предыдущей вершины стека
4. Созданное звено сделать вершиной стека

Процедура занесения элемента в стек

```
Procedure zanes(var st:stack;el:<тип элемента стека>);
 Var q:adres1;
 Begin
  New(Q);
  Q^.element:=el;
  Q^.adrcled:=st;
  St:=Q;          End;
```

22. АБСТРАКТНЫЙ ТИП ДАННЫХ «СТЕК». АЛГОРИТМ И ПРОЦЕДУРА УДАЛЕНИЯ ЭЛЕМЕНТА ИЗ СТЕКА С ПОМОЩЬЮ УКАЗАТЕЛЕЙ

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).

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). 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;
```

Если стек пуст, то значение указателя st равно nil.

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

Запись элементов с тек

```
Procedure WriteStack (var x:stack);
  Begin
    While x.top>0 then do
    Begin
      Readln(x.elements[x.top]);
      x.top:=x.top-1;
    end;
  end;
```

Извлечение элемента из стека

```
Procedure print(var x:stack);
 begin
  while x.top<=5 do
  begin
    writeln(x.elements[x.top]);
    x.top:=x.top+1;
  end;
 end;
```

Проверка полноты стека

```
Function tops(var x:stack):boolean;
 Begin
  If x.top<1 then tops:=true
             Else tops:=false;
 End;
```

Проверка пустоты стека

```
Function empty(var x:stack):boolean;
Begin
If x.top>5 then empty :=thrue
           Else empty:= false;
End;
```

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 adres1 = ^stack; {Указатель на элемент стека}
   Stack = record
   Element:<тип элемента стека>;
   Adrcled:adres1;{Указатели на следующий элемент  очереди}
  End;
Var Q :adres1;
```

Алгоритм выборки элемента из стека:

1. Прочитать значение из вершины стека
2. Запомнить ссылку на старую вершину
3. Исключить первое звено из стека
4. Уничтожить первое звено

Процедура удаления элемента из стека:

```
Procedure vibor(var st:stack;var A:<тип элемента стека>);
 Var Q:adres1;
 Begin
  A:=st^.element;
  Q:=st;
  St:=st^.adrcled;
  Dispose(Q);                End;
```

23. ПОСТФИКСНАЯ, ПРЕФИКСНАЯ И ИНФИКСНАЯ ЗАПИСИ ПРЕДСТАВЛЕНИЯ ВЫРАЖЕНИЙ И ИХ ОСОБЕННОСТИ. ПРИВЕСТИ ПРИМЕРЫ

Введем понятие различных форм записи выражений. **A+B** – инфиксная: знак операции находится между операндами; **+AB** – префиксная (польская): знак операции расположен перед операндами; **AB+ –** 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. 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. Ниже приведены примеры различных форм записи выражений.

Инфиксное представление	**Постфиксное представление**
A+B-C	AB+C-
(A+B)*(C-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**
A+B-C	- +ABC
(A+B)*(C-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. Отсюда вытекает основное преимущество обратной польской записи перед обычной записью выражений со скобками: выражение можно вычислить в процессе однократного просмотра слева направо.

24. ИСПОЛЬЗОВАНИЕ СТЕКА ОПЕРАЦИЙ ДЛЯ ПЕРЕВОДАВЫРАЖЕНИЙ ИЗ ИНФИКСНОЙ В ПОСТФИКСНУЮ ЗАПИСЬ. ПРИВЕСТИ АЛГОРИТМ

Символы	Приоритеты
([if	0
:=)] , then else	1
⊃	2
OR	3
AND	4
NOT	5
> >= = != <= <	6
+ -	7
/ ? ±	8

Инфиксное представление	**Постфиксное представление**
A+B-C	AB+C-
(A+B)*(C-D)	AB+CD-*
A^B*C-D+E/F/(G+H)	AB^C*D-EF/GH+/+
A-B/(C*D^E)	ABCDE^*/-

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	–

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.

Входная строка	Стек				Выходная строка									
a					a									
+	+				a									
b	+				a	b								
×	+	×			a	b								
c	+	×			a	b	c							
-	-				a	b	c	×	÷					
d	-				a	b	c	×	÷	d				
/	-	/			a	b	c	×	÷	d				
(	-	/	(		a	b	c	×	÷	d				
a	-	/	(		a	b	c	×	÷	d	a			
÷	-	/	(	+	a	b	c	×	÷	d	a			
b	-	/	(	+	a	b	c	×	÷	d	a	b		
)	-	/			a	b	c	×	÷	d	a	b	÷	
					a	b	c	×	÷	d	a	b	÷	/	-

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

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

25. ИСПОЛЬЗОВАНИЕ СТЕКА ОПЕРАЦИЙ ДЛЯ ПЕРЕВОДА ВЫРАЖЕНИЙ ИЗ ИНФИКСНОЙ В ПРЕФИКСНУЮ ЗАПИСЬ. ПРИВЕСТИ АЛГОРИТМ

Алгоритм преобразования выражения из инфиксной формы в префиксную запись рассмотрим на примере выражения *a+b/(c-d)*.

1. Необходимо переписать выражение справа налево: *(d-c)/b+a*;
2. Воспользовавшись алгоритмом постфиксной трансляции, получим: *dc-b/a+;*
3. Полученную строку требуется записать справа налево, в результате чего получается выражение в префиксном виде: *+a/b-cd*.

Алгоритм преобразования в постфиксную.

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.

Инфиксное представление	**Префиксное представление**
A+B-C	- +ABC
(A+B)*(C-D)	*+AB-CD
((A+B)*C-(D-E)^(F+G)	^ -*+ABC-DE+FG
A-B/(C*D^E)	-A/B*C^DE

26. ПРАВИЛО ВЫЧИСЛЕНИЯ ВЫРАЖЕНИЯ В ПОСТФИКСНОЙ ЗАПИСИ

Правило вычисления выражения в обратной польской записи состоит в следующем. Обратная польская запись просматривается слева направо. Если рассматриваемый элемент - операнд, то рассматривается следующий элемент. Если рассматриваемый 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, имеющиеся в выражении, и запись сократится до одного элемента — результата вычисления выражения.

Простое арифметическое выражение с вещественными переменными **a + b × c - d / (a + b)** можно представить в виде обратной польской записи: **a b c × + d a b + / -.** Выполнение правила для нашего примера приводит к последовательности строк, записанных во второй графе таблицы. 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.

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.

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 + / -	$r_1 := b \times c$	× 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 r_j. 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.

27. ФОРМАЛЬНОЕ ОПРЕДЕЛЕНИЕ ТИПА ДАННЫХ «ДЕРЕВО». ОТНОШЕНИЕ МЕЖДУ УЗЛАМИ В ДЕРЕВЕ. ПОНЯТИЯ ПРЕДОК, ПОТОМОК, ПУТЬ , ДЛИНА ПУТИ. ПРИМЕРЫ

Дерево представляет собой иерархическую структуру какой-либо совокупности элементов. Структуры в виде деревьев нашли широкое применение на практике. Примерами деревьев могут служить генеалогические и организационные диаграммы. Деревья используются при анализе электрических цепей, при разборе структур математических формул, а также для организации информации в системах управления базами данных и для представления синтаксических структур в компиляторах.

Дерево – это совокупность элементов, называемых узлами, (один из которых определен как корень), и отношений, образующих иерархическую структуру узлов. Узлы дерева могут быть элементами любого типа и обычно изображаются буквами, строками или числами.

Формально дерево можно определить следующим образом.

Один узел является деревом, он же является корнем этого дерева.

Пусть n – это узел, а T1,T2…Tk – деревья с корнями n1,n2…nk соответственно. Можно построить новое дерево, сделав n родителем узлов n1,n2…nk . В этом дереве n будет корнем, а T1,T2…Tk – поддеревьями этого корня. Узлы n1,n2…nk называются **сыновьями** узла n.

Часто в это определение включают в себя понятие нулевого дерева, т.е. дерева без узлов.

Существует несколько **способов изображения структуры дерева:** вложенные множества; вложенные скобки; с помощью отступов; графически.

Отношение родитель-сын чаще всего отображается в виде линии. Дерево обычно рисуется сверху вниз, так чтобы родители (в общем случае предки) располагались выше детей (в общем случае потомки). Узлы дерева, не имеющие потомков, называются *листьями*.

Дерево – это наилучшая форма представления четко структурированных данных. **Путем** из узла n_1 в узел n_k называется последовательность узлов n1,n2…nk , где для всех i , 1□ i □ k, узел n_i является родителем узла n_{i+1} .

Длиной пути называется число узлов, составляющих этот путь. Таким образом, путем нулевой длины будет путь из любого узла к самому себе.

Предок или потомок узла, не являющийся самим этим узлом, называется истинным предком или истинным потомком данного узла. В дереве только корень не имеет истинного предка. Узел, не имеющий истинных потомков, называется листом. Тогда поддерево какого-либо дерева можно определить как узел (корень поддерева) вместе со всеми его потомками.

Высотой узла дерева называется длина самого длинного пути из этого узла до какого либо листа. В нашем примере высота узла Глава 1 равна 1, узла Глава 2 – 2, а узла Глава 3 – 0. Порядок узлов дерева. Если имеет значение относительный порядок поддеревьев $T_1, T_2 \ldots T_k$ в дереве, то можно говорить, что дерево является упорядоченным; в случае, когда порядок узлов игнорируется, дерево называют неупорядоченным. Сыновья узла обычно упорядочиваются слева направо. Поэтому два дерева на рис. 2 различны, т.к. порядок сыновей узла A различен.

Упорядочивание сыновей слева направо можно использовать для сопоставления узлов, которые не связаны отношениями предки-потомки. Соответствующее правило звучит следующим образом. Если узлы a и b являются сыновьями одного родителя, и узел a лежит слева от узла b, то все потомки узла a будут находиться слева от любых потомков узла b.

Существует простое правило, позволяющее определить, какие узлы расположены слева от данного узла n, а какие – справа. Для этого надо прочертить путь от корня дерева до узла n. Тогда все узлы и их потомки, расположенные слева от этого пути, будут находиться слева от узла n, и аналогично, все узлы и их потомки, расположенные справа от этого пути, будут находиться справа от узла n.

28. ОПИСАНИЕ ВЕРШИНЫ ДЕРЕВА. ПОНЯТИЕ БИНАРНОГО ДЕРЕВА ПОИСКА. ПРОЦЕДУРА ВСТАВКИ ЭЛЕМЕНТА В БИНАРНОЕ ДЕРЕВО ПОИСКА

В древовидной структуре число потомков вершины называется ее степенью. Максимальное значение этих степеней называется степенью дерева. Наибольшую популярность в программировании и вычислительной технике получили **бинарные (двоичные) деревья**, у которых степень дерева равна двум. В этом случае вершина дерева может иметь не более двух потомков, называемых левым и правым сыновьями. В отдельный подкласс бинарных деревьев выделены **деревья поиска**. Они характеризуются тем, что значение информационного поля, связанного с вершиной дерева, больше любого соответствующего значения из левого поддерева и меньше, чем содержимое любого узла его правого поддерева. **Описание вершины дерева** имеет следующий вид.

```
Type     pt=^node;
   node=record
       data:integer; {Информационное поле}
       left,right:pt; {Указатели на левого и правого потомков}
       end;
var  root  :pt; {Указатель на корень дерева}
```

К основным операциям, выполняемым с деревом, относятся вставка элемента, удаление элемента, обход дерева.

Создание бинарного древа можно реализовать на основе операции вставки элемента. Ниже приведена соответствующая процедура.

Процедура вставки в бинарное дерево поиска:

```
ProcedureInsert(varx : pt;  y:integer);{x – указатель на корень дерева, y -
      Begin                                вставляемая вершина}
      if x=nil then
         Begin
            new(x);
            x^.data:=y;
            x^.left:=nil;
            x^.right:=nil;
          End
      else if y<= x^.data then insert (x^.left,y)
           else  insert (x^.right,y);
  End;
```

28.1 ПОНЯТИЕ ОБХОДА ДЕРЕВА. РЕКУРСИВНОЕ ОПРЕДЕЛЕНИЕ И ПРОЦЕДУРА ПРЯМОГО ОБХОДА ДЕРЕВА. ПРИВЕСТИ ПРИМЕР

В ходе решения прикладных задач с применением структур в виде деревьев очень часто выполняются различные обходы дерева. Существует несколько способов **обхода** (прохождения) всех узлов дерева. Наиболее популярны три следующих способа обхода дерева: прямой (сверху вниз), обратный (снизу вверх), слева направо (симметричный).

Все три способа обхода дерева можно определить рекурсивно следующим образом.

Если дерево T является нулевым деревом, то в список обхода заносится пустая запись.

Если дерево T состоит из одного узла, то в список обхода записывается этот узел.

Далее, пусть T – дерево с корнем n и поддеревьями $T_1, T_2, ..., T_k$, тогда для различных способов обхода имеем следующее:

а) при прохождении в прямом порядке (т.е. при прямом упорядочивании) узлов дерева T сначала посещается корень n , а затем узлы поддерева T_1, далее все узлы поддерева T_2 и т. д. Последним посещаются узлы поддерева T_k.

После каждой процедуры приводится последовательность узлов дерева, выделенная жирным шрифтом, соответствующая указанному способу обхода. Подчеркиванием помечены узлы, в которые возвращается процедура в ходе рекурсивного вычислительного процесса.

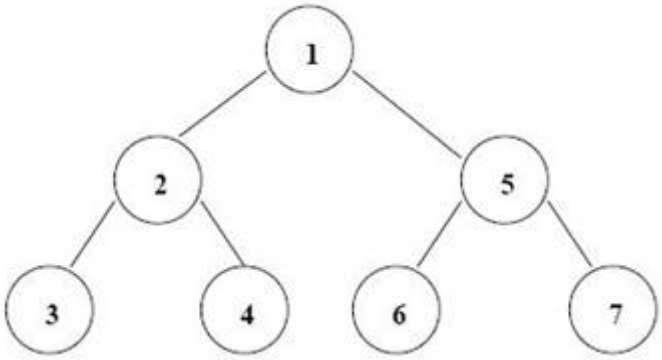


Рис. 15 – Дерево обхода

Прямой обход дерева:

```
Procedure prym_print(t: адрес элемента);
  Begin
    if  x<> nil then
      Begin
    P(t): {обработка вершины}
       write (t);
       prym_print(t^.left);
      prym_print(t^.right);
   End;
 End;
```

Последовательность обработки: **1, 2, 3,**2, **4**, 2, 1, **5, 6,**5, **7,** 5, 1

29. ПОНЯТИЕ ОБХОДА ДЕРЕВА. РЕКУРСИВНОЕ ОПРЕДЕЛЕНИЕ И ПРОЦЕДУРА СИММЕТРИЧНОГО ОБХОДА ДЕРЕВА. ПРИВЕСТИ ПРИМЕР

В ходе решения прикладных задач с применением структур в виде деревьев очень часто выполняются различные обходы дерева. Существует несколько способов **обхода** (прохождения) всех узлов дерева. Наиболее популярны три следующих способа обхода дерева: прямой (сверху вниз), обратный (снизу вверх), слева направо (симметричный).

Все три способа обхода дерева можно определить рекурсивно следующим образом.

Если дерево T является нулевым деревом, то в список обхода заносится пустая запись.

Если дерево T состоит из одного узла, то в список обхода записывается этот узел.

Далее, пусть T – дерево с корнем n и поддеревьями $T_1, T_2, ..T_k$, тогда для различных способов обхода имеем следующее:

б) при симметричном обходе узлов дерева T сначала посещаются в симметричном порядке все узлы поддерева T_1, далее корень n, затем последовательно в симметричном порядке все узлы поддеревьев $T_2, ...T_k$.

После каждой процедуры приводится последовательность узлов дерева, выделенная жирным шрифтом, соответствующая указанному способу обхода. Подчеркиванием помечены узлы, в которые возвращается процедура в ходе рекурсивного вычислительного процесса.

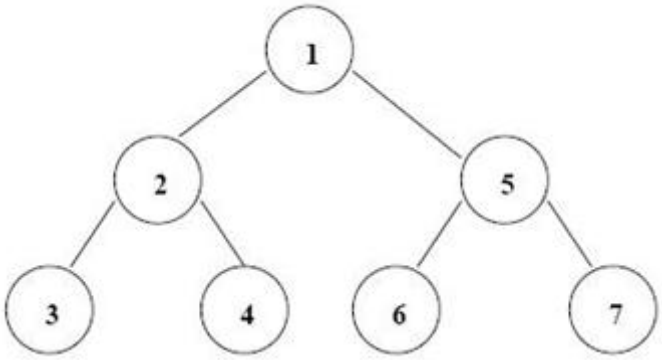


Рис. 15 – Дерево обхода

Симметричный обход дерева:

```
Procedure sim_print(var x:pt);
  Begin
    if  x<> nil then
      Begin
      sim_print(x^.left);
      write (x^.data);
      sim_print(x^.right);
     End;
End;
```

Последовательность обработки: 1, 2, **3,2,4**, 2, **1**,5, **6, 5, 7,** 5, 1

30. ПОНЯТИЕ ОБХОДА ДЕРЕВА. РЕКУРСИВНОЕ ОПРЕДЕЛЕНИЕ И ПРОЦЕДУРА ОБРАТНОГО ОБХОДА ДЕРЕВА. ПРИВЕТИ ПРИМЕР

В ходе решения прикладных задач с применением структур в виде деревьев очень часто выполняются различные обходы дерева. Существует несколько способов **обхода** (прохождения) всех узлов дерева. Наиболее популярны три следующих способа обхода дерева: прямой (сверху вниз), обратный (снизу вверх), слева направо (симметричный).

Все три способа обхода дерева можно определить рекурсивно следующим образом.

Если дерево T является нулевым деревом, то в список обхода заносится пустая запись.

Если дерево T состоит из одного узла, то в список обхода записывается этот узел.

Далее, пусть T – дерево с корнем n и поддеревьями $T_1, T_2, .. T_k$, тогда для различных способов обхода имеем следующее:

в) в случае обхода в обратном порядке сначала посещаются в обратном порядке все узлы поддерева T_1, затем последовательно посещаются все узлы поддеревьев $T_2, .. T_k$, также в обратном порядке, последним посещается корень n.

После каждой процедуры приводится последовательность узлов дерева, выделенная жирным шрифтом, соответствующая указанному способу обхода. Подчеркиванием помечены узлы, в которые возвращается процедура в ходе рекурсивного вычислительного процесса.

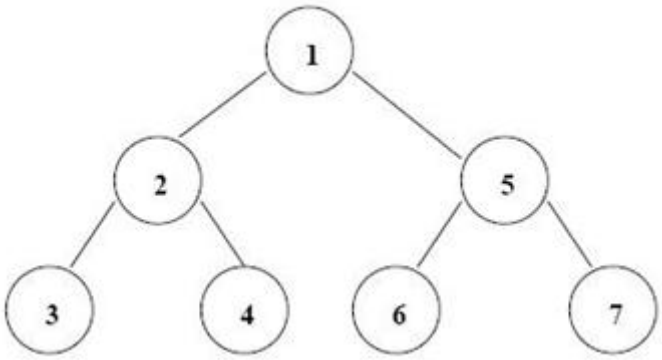


Рис. 15 – Дерево обхода

Обратный обход дерева:

```
 Procedure obr_print (t:адресэлемента);
   Begin
     if  t<> nil then
       Begin
         obr_print(t^.left);
         obr_print(t^.right);
P(t): {обработка вершины}
         write (t);
       End;
   End;
```

Последовательность обработки: 1, 2, **3,** 2,**4**, **2**, 1, 5, **6,** 5, **7, 5, 1**

31. ОПИСАНИЕ ВЕРШИНЫ ДЕРЕВА. ПРОЦЕДУРА ПОИСКА В ДЕРЕВЕ ЭЛЕМЕНТА С ЗАДАННЫМ КЛЮЧОМ

Описание вершины дерева имеет следующий вид.

```
 Type     pt=^node;
    node=record
        data:integer; {Информационное поле}
        left,right:pt; {Указатели на левого и правого потомков}
        end;
 var  root  :pt; {Указатель на корень дерева}
```

Бинарное дерево содержащее текст записи

```
Type
   Tekst=<тип значения записи>;
   Adrt=^tekst;
   Adrzv=^zveno;
   Zveno=record
        Kl:integer;
        Lev,prav:adrzv;
        Adr:adrt;
   End;
Var
   Dvder:adrzv;
```

Искать вершину дерева с заданным ключом K, D ссылка на корень дерева и rez-переменная которой:=ссылка на найденное звеноили ссылка на вершину после обработки которой поиск прекращен.

Процедура поиска в дереве элемента с заданным ключом:

```
Function poisk(k:integer;var D,rez:adrzv):boolean;
 Var
  P,Q:adrzv;
  B:boolean;
 Begin
  B:=false;
  P:=9;
  Q:=nil;
  If P<>nil then  repeat
                    Q:=P;
                    If P K!=K then B:=true
                                 Else if k<P K! then P:=P^.lev
                                                Else P:=P^.prav
                   Until B or (P=nil);
  Poisk:=B;
  Rez:=Q;
End;
```

32. ОПИСАНИЕ ВЕРШТНЫ ДЕРЕВА. ПРОЦЕДУРА ВСТАВКИ В ДЕРЕВО ЭЛЕМЕНТА С ЗАДАННЫМ КЛЮЧОМ

Описание вершины дерева имеет следующий вид.

```
Type     pt=^node;
   node=record
       data:integer; {Информационное поле}
       left,right:pt; {Указатели на левого и правого потомков}
       end;
var  root  :pt; {Указатель на корень дерева}
```

Бинарное дерево содержащее текст записи

```
Type
  Tekst=<тип значения записи>;
  Adrt=^tekst;
  Adrzv=^zveno;
  Zveno=record
       Kl:integer;
       Lev,prav:adrzv;
       Adr:adrt;
  End;
Var
  Dvder:adrzv;
```

Чтобы включить запись в дерево нужно найти вершину к которой можно присоединить новую соответствующую включаемой записи. Алгоритм поиска новой вершины аналогичен поиску с заданным ключом. Нужная вершина найдена если в качестве очередной ссылки определяется ветвь. Продолжение поиска .. nil. Rez адрес вершины. Нет записи с тем же ключом что и у добавляемой записи. К- ключ, Zap-текст вставляемой записи, D-корень дерева.

Процедура вставки элемента в дерево:

```
Procedure vkl(k:integer;var D:adrzv;
zap:tekst);
 Var
  Q,S:adrzv;
  T:adrt;
 Begin
  If not poisk(K,D,Q) then begin
                New(T);
                T^:=zap;
                New(s);
                S^.kl:=k;
                S^.adr:=T;
                S^.lev:=nil;
                S^.prav:=nil;
                D=nil then D:=s
                Else if k<Q^.kl then Q^.lev:=s
                              Else Q^.prav:=s
                 End;
 End;
```

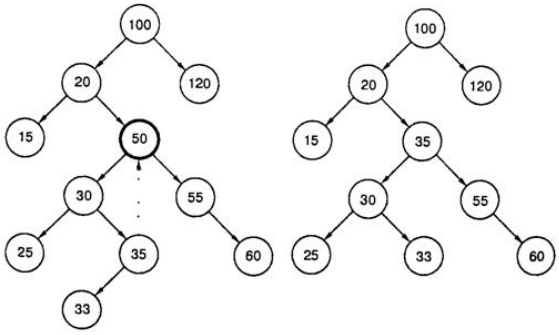
```
Function poisk(k:integer;var
D,rez:adrzv):boolean;
 Var
  P,Q:adrzv;
  B:boolean;
 Begin
  B:=false;
  P:=9;
  Q:=nil;
  If P<>nil then  repeat
          Q:=P;
          If P K!=K then B:=true
               Else if k<P K! then P:=P^.lev
                             Else P:=P^.prav
       Until B or (P=nil);
  Poisk:=B;
  Rez:=Q;
End;
```

33. СИТУАЦИЯ УДАЛЕНИЯ ЭЛЕМЕНТА ИЗ ДЕРЕВА.ПРОЦЕДУРА УДАЛЕНИЯ ЗАДАННОГО ЭЛЕМЕНТА ИЗ ДЕРЕВА

Если соответствующая вершина является листом дерева или из нее выходит только одна ветвь то для удаления достаточно скорректировать соответствующую ссылку вершины предшественника.

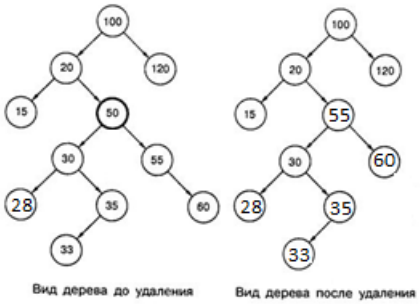
Если из удаляемой вершины выходит 2 ветви то нужно найти звено дерева которое можно было бы вставить на место удаляемого:

- самый правый элемент левого поддерева. Данный элемент является самым большим в левом от удаляемого дереве. Для достижения этого звена необходимо перейти в следующую от удаляемой вершину по левой ветви а потом переходить в очередные вершины только по правой ветви до тех пор пока очередная ссылка не будет равна nil.



Вид дерева до удаления Вид дерева после удаления

- самый левый элемент правого поддерева. Данный элемент является самым малым в правом от удаляемой вершины поддереве. Для достижения этого звена необходимо перейти в следующую от удаляемой вершину по правой ветви а потом переходить в очередные вершины только по левой ветви до тех пор пока ссылка не будет равна nil.



Вид дерева до удаления Вид дерева после удаления

Таким образом при исключении вершины из бинарного дерева необходимо учесть 3 случая:1)вершины с заданным ключом в дереве нет;

2)вершина с заданным ключом имеет не более одного поддерева;

3)вершина с заданным ключом имеет 2 ветви.

Процедура удаления вершины из дерева:(первый вариант исключения)

```
Procedure Udder (var D : adrzv;  K:integer);
 var Q:adrzv;
 procedure Ud (var R :adrzv);
  begin
   if R^.prav=nil then begin
                         Q^.kl:=R^.kl;
                         Q^.adr:=R^.adr;
                         Q:=R;
                         R:=Q^.lev;
                        End;
                  Else Ud(R^.prav);
 End;
  begin
   if D=nil then writeln('звена с заданным
ключом в дереве нет');
     else if k<D^.kl thenUdder(D^.lev,k)
          else if k>D6.kl thenUdder(D^.prav,k)
               else begin
                      Q:=D;
             If Q^.prav=nil then D:=Q^.lev
                Else If Q^.lev=nil then D:=Q^.prav
                     Else Ud(Q^.lev)
                        End;
 End;
```

34.ПОМЕЧЕННЫЕ ДЕРЕВЬЯ ПРАВИЛА СООТВЕТСТВИЯ МЕТОК ДЕРЕВЬЕВ ЭЛЕМЕНТАМ ВЫРАЖЕНИЙ. ПРИВЕСТИ ПРИМЕР ПРЯМОГО ОБРАТНОГО И СИММЕТРИЧНОГО ОБХОДОВ ПОМЕЧЕННОГО ДЕРЕВА

Часто при работе с древовидными структурами бывает полезным сопоставить каждому узлу дерева метку или значение, аналогично тому, как с элементами списков связывают определенные значения. Дерево, у которого узлам приписаны метки, называется **помеченным деревом**. Метка узла – это не имя, а значение которое «хранится» в узле. Полезна следующая аналогия: дерево – список, узел – позиция, метка – элемент.

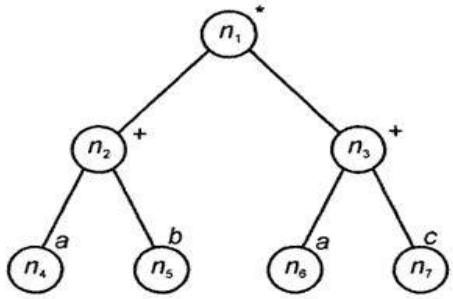
Рассмотрим дерево с метками (рис), представляющее арифметическое выражение: (a+b)*(a+c), где n1,…n7 – имена узлов (метки проставлены рядом с соответствующими узлами). **Правила соответствия меток деревьев элементам выражений следующие:**

1. Метка каждого листа соответствует операнду и содержит его значение, например узел n4 представляет операнд a.
2. Метка каждого внутреннего узла (родительского) соответствует оператору.

Предположим, что узел n помечен бинарным оператором □, (например, + или *) и левый сын этого узла соответствует выражению E1, а правый – выражению E2. Тогда узел n и его сыновья представляют выражение (E1)□(E2).

Например, узел n2 имеет оператор +, а левый и правый сыновья представляют выражения (операнды) a и b, соответственно. Поэтому узел n2 представляет выражение (a)+(b) или a+b. Узел n1 представляет выражение (a+b)*(a+c), поскольку оператор * является меткой узла n1, выражения a+b и a+спредставляются узлами n2 и n3, соответственно.

Часто при обходе деревьев составляется список не имен, а их меток. В случае дерева выражений при прямом упорядочивании получаем префиксную форму выражений, где оператор предшествует и левому и правому операндам. Для точного описания префиксной формы выражений сначала положим, что префиксным выражением одиночного операнда a является сам этот операнд. Далее, префиксная форма для выражений (E1)□(E2), где □– бинарный оператор, имеет вид □P1P2, здесь P1 и P2 – префиксные формы для выражений E1 и E2. Отметим, что в префиксных формах нет необходимости отделять или выделять префиксные выражения скобками, так как всегда можно просмотреть префиксное выражение □P1P2 и определить единственным образом P1 как самый короткий префикс выражения P1P2.



Например, при прямом упорядочивании узлов (точнее, меток) дерева (рис.3) получаем префиксное выражение *+ab+ac. Самым коротким корректным префиксом для выражения +ab+ac будет префиксное выражение узла n2: +ab.

Обратное упорядочивание меток дерева выражений дает постфиксное представление выражений. Выражение □P1P2 в постфиксной форме имеет вид P1P2□, где P1 и P2 – постфиксные формы для выражений E1и E2 соответственно. При использовании постфиксной формы также нет необходимости в применении скобок, поскольку для любого выражения P1P2 легко проследить самый короткий суффикс P2, что и будет корректным составляющим постфиксным выражением. Например, постфиксная форма выражения для дерева на рис. 3 имеет вид ab+ac+*. Если записать выражение как P1P2*, то P2 (т.е. выражение ac+) будет самым коротким суффиксом для ab+ac+ и, следовательно, корректным составляющим постфиксным выражением.

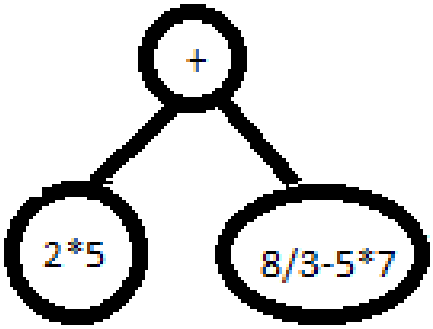
При симметричном обходе дерева выражений получим инфиксную форму выражения, которая совпадает с привычной «стандартной» формой записи, но также не использует скобок. Для дерева на рис. 3 инфиксное выражение запишется как a+b*a+c. Скобки в инфиксное выражение добавляются с помощью отдельного алгоритма.

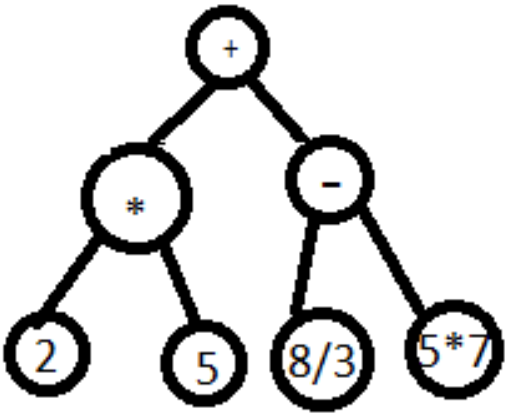
35. АЛГОРИТМ ПОСТРОЕНИЯ ПОМЕЧЕННОГО ДЕРЕВА ПО ВЫРАЖЕНИЮ И ОБХОДЫ ДЕРЕВА. ПРИМЕРЫ

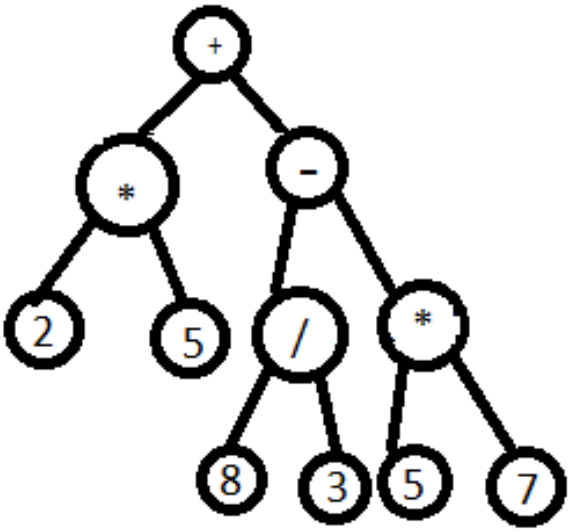
Алгоритм построения дерева и вычисления выражения:

1. Предварительная обработка выраения(контроль правильности расстановки скобок в выражении, удаление пробелов, замена унарного минуса на (0-1)*);
2. Построение дерева для даного выражения. Поиск в выражении арифметической аперации с минимальным приоритетом с пропуском вложенных скобок . разделение выражения на 2 части относительно найденной операции с наименьшим приоритетом и рекурсивное повторение;
3. Вычисление выражения по дереву.

Пример: 2*5+8/3-5*7







обходы сверху вниз и снизу вверх дают результат:
префиксная запись:+*2 5 -/8 3*5 7;(прямой обход)
постфиксная запись: 2 5*8 3 /5 7 * - +(обратный обход)
симметричнй обход дает инфиксную запись.

36. СТРУКТУРА УЗЛА ПРОШИТОГО ДЕРЕВА. ПРОЦЕДУРА СИММЕТРИЧНОЙ ПРОШИВКИ БИНАРНОГО ДЕРЕВА

Эффективность прохождения дерева рекурсивными и нерекурсивными алгоритмами может быть увеличена, если использовать пустые указатели на отсутствующие поддеревья для хранения в них адресов узлов преемников, которые надо посетить при заданном порядке прохождения бинарного дерева. Такой указатель называется нитью. Его следует отличать от указателей в дереве, которые используются с левым и правым поддеревьями. Операция, заменяющая пустые указатели на нити, называется прошивка. Она может выполняться по-разному. Если нити заменяют пустые указатели в узлах с пустыми правыми поддеревьями, при просмотре в симметричном порядке, то бинарное дерево называется симметрично прошитым справа.Похожим образом может быть определено бинарное дерево, симметрично прошитое слева:дерево, в котором каждый пустой левый указатель изменен так, что он содержит нить – связь к предшественнику данного узла при просмотре в симметричном порядке. Симметрично прошитое бинарное дерево – это то, которое симметрично прошито слева и справа. Однако левая прошивочная нить не дает тех преимуществ, что правая прошивочная нить. Также используются бинарные деревья, прямо прошитые справа и слева. В них пустые правые и левые указатели узлов заменены соответственно на их преемников и предшественников при прямом порядке просмотра. Поскольку нужно каким-то образом отличать обычную связь от прошивочной нити, каждому узлу добавляется два однобитовых (логических) поля тэга: ltag и rtag. Если значение тэга true, соответствующее поле связи является обычной связью, в случае значения false– прошивочной нитью.

Узел прошитого бинарного дерева имеет иную структуру

```
Type ptr = ^node;
node = record
info : integer;   {Информационноеполе}
      ltag, rtag : boolean; {Тэгипрошивочныхнитей}
      left, right : pt;  {Указатели на левое и правое поддеревья}
    end;
```

Логические поля в прошитом дереве могут принимать следующие значения:

1. ltag=true, следовательно, left представляет собой обычную связь.
2. ltag=false, следовательно, left указывает на узел-предшественник.
3. rtag=true, следовательно, right представляет собой обычную связь.
4. rtag=false, следовательно, rightуказывает на узел-преемник.

*Наряду **с преимуществами прошитых деревьев**: быстрый обход, отсутствие необходимости в стеке, можно определить предшественника и преемника вершины, существуют **недостатки**. Включение новой вершины в дерево занимает больше времени, т.к. необходимо поддерживать два типа связей: структурные и по нитям. Поэтому прошитые деревья целесообразно использовать в тех задачах, где изменения в деревьях происходят редко, а обходы выполняется часто.*

Процедура симметричнй прошивки бинарного дерева:

```
proceduresim_print(var x:pt);
    procedurerightsew( var p:pt);
      begin
        if y <> nil then
          begin
            ify^.right=nil then
              begin
    y^.rtag := false;      y^.right := p;
                          end
          else    y^.rtag := true;
          end;
        y:=p;
      end;
    begin
      if  x<> nil then
        begin
    sim_print(x^.left);     rightsew(x);
          sim_print(x^.right);        end;          end;
```

37. СТРУКТУРА УЗЛА ПРОШИТОГО ДЕРЕВА. ПРОЦЕДУРА ОБХОДА СИММЕТРИЧНО ПРОШИТОГО ДЕРЕВА

Эффективность прохождения дерева рекурсивными и нерекурсивными алгоритмами может быть увеличена, если использовать пустые указатели на отсутствующие поддеревья для хранения в них адресов узлов преемников, которые надо посетить при заданном порядке прохождения бинарного дерева. Такой указатель называется нитью. Его следует отличать от указателей в дереве, которые используются с левым и правым поддеревьями. Операция, заменяющая пустые указатели на нити, называется прошивка. Она может выполняться по-разному. Если нити заменяют пустые указатели в узлах с пустыми правыми поддеревьями, при просмотре в симметричном порядке, то бинарное дерево называется симметрично прошитым справа.Похожим образом может быть определено бинарное дерево, симметрично прошитое слева:дерево, в котором каждый пустой левый указатель изменен так, что он содержит нить – связь к предшественнику данного узла при просмотре в симметричном порядке. Симметрично прошитое бинарное дерево – это то, которое симметрично прошито слева и справа. Однако левая прошивочная нить не дает тех преимуществ, что правая прошивочная нить. Также используются бинарные деревья, прямо прошитые справа и слева. В них пустые правые и левые указатели узлов заменены соответственно на их преемников и предшественников при прямом порядке просмотра. Поскольку нужно каким-то образом отличать обычную связь от прошивочной нити, каждому узлу добавляется два однобитовых (логических) поля тэга: ltag и rtag. Если значение тэга true, соответствующее поле связи является обычной связью, в случае значения false– прошивочной нитью.

Узел прошитого бинарного дерева имеет иную структуру

```
Type ptr = ^node;
node = record
info : integer;   {Информационноеполе}
      ltag, rtag : boolean; {Тэгипрошивочныхнитей}
      left, right : pt;  {Указатели на левое и правое поддеревья}
     end;
```

Логические поля в прошитом дереве могут принимать следующие значения:

1. ltag=true, следовательно, left представляет собой обычную связь.
2. ltag=false, следовательно, left указывает на узел-предшественник.
3. rtag=true, следовательно, right представляет собой обычную связь.
4. rtag=false, следовательно, rightуказывает на узел-преемник.

*Наряду **с преимуществами прошитых деревьев**: быстрый обход, отсутствие необходимости в стеке, можно определить предшественника и преемника вершины, существуют **недостатки**. Включение новой вершины в дерево занимает больше времени, т.к. необходимо поддерживать два типа связей: структурные и по нитям. Поэтому прошитые деревья целесообразно использовать в тех задачах, где изменения в деревьях происходят редко, а обходы выполняется часто.*

Процедура обхода симметрично прошитого бинарного дерева:

```
Procedure obxod_proshiv(var x:pt);
 Begin
  While x<>head do
   Begin
    While x^.left <>nil do x:=x^.left;
    Writeln(x^.data);
    While x^.rtad=false do begin
          X:=x^.right;
     If x=head then exit;
     Writeln(x^.data);
       End;
     X:=x^.right;
    End;
   End;
  End;
```

Алгоритм симметричного обхода прошитого дерева можно сформулировать следующим образом: 1. Переход к корню дерева (p := HEAD^.left).

2. До тех пор, пока p^.left<>nil, повторять: p := p^.left, то есть идти по левой ветви до самого левого узла.

3. Обработка узла p, например, печать p^.info.

4. Если p^.rtagравен false, то p := p^.right и переход к шагу 3 (к преемнику). Иначе p:=p^.right и переход к шагу 2.

Алгоритм заканчивает работу, когда p станет равным HEAD.

38. СТРУКТУРА УЗЛА ПРОШИТОГО ДЕРЕВА. ПРОЦЕДУРА ПРЯМОЙ ПРОШИВКИ БИНАРНОГО ДЕРЕВА

Эффективность прохождения дерева рекурсивными и нерекурсивными алгоритмами может быть увеличена, если использовать пустые указатели на отсутствующие поддеревья для хранения в них адресов узлов преемников, которые надо посетить при заданном порядке прохождения бинарного дерева. Такой указатель называется нитью. Его следует отличать от указателей в дереве, которые используются с левым и правым поддеревьями. Операция, заменяющая пустые указатели на нити, называется прошивка. Она может выполняться по-разному. Если нити заменяют пустые указатели в узлах с пустыми правыми поддеревьями, при просмотре в симметричном порядке, то бинарное дерево называется симметрично прошитым справа.Похожим образом может быть определено бинарное дерево, симметрично прошитое слева:дерево, в котором каждый пустой левый указатель изменен так, что он содержит нить – связь к предшественнику данного узла при просмотре в симметричном порядке. Симметрично прошитое бинарное дерево – это то, которое симметрично прошито слева и справа. Однако левая прошивочная нить не дает тех преимуществ, что правая прошивочная нить. Также используются бинарные деревья, прямо прошитые справа и слева. В них пустые правые и левые указатели узлов заменены соответственно на их преемников и предшественников при прямом порядке просмотра. Поскольку нужно каким-то образом отличать обычную связь от прошивочной нити, каждому узлу добавляется два однобитовых (логических) поля тэга: ltag и rtag. Если значение тэга true, соответствующее поле связи является обычной связью, в случае значения false– прошивочной нитью.

Узел прошитого бинарного дерева имеет иную структуру

```
 Type ptr = ^node;
node = record
info : integer;   {Информационноеполе}
       ltag, rtag : boolean; {Тэгипрошивочныхнитей}
       left, right : pt;  {Указатели на левое и правое поддеревья}
     end;
```

Логические поля в прошитом дереве могут принимать следующие значения:

1. ltag=true, следовательно, left представляет собой обычную связь.
2. ltag=false, следовательно, left указывает на узел-предшественник.
3. rtag=true, следовательно, right представляет собой обычную связь.
4. rtag=false, следовательно, rightуказывает на узел-преемник.

*Наряду **с преимуществами прошитых деревьев**: быстрый обход, отсутствие необходимости в стеке, можно определить предшественника и преемника вершины, существуют **недостатки**. Включение новой вершины в дерево занимает больше времени, т.к. необходимо поддерживать два типа связей: структурные и по нитям. Поэтому прошитые деревья целесообразно использовать в тех задачах, где изменения в деревьях происходят редко, а обходы выполняется часто.*

Процедура прямой прошивки бинарного дерева:

```
procedure sim(var x:pt);
procedure rightsew(var p:pt);
begin
if y <> nil then
begin
if y^.right=nil then
begin
y^.rtag := false;
y^.right := p;
write (y^.data,'->');
write (p^.data,' ');
writeln;
end
else begin y^.rtag := true; end;
end;
y:=p;
end;
begin
if (x<> nil) and (x<>y) and (k<=t) then
begin
inc(k);
rightsew(x);
sim(x^.left);
sim(x^.right);
end;
if x<>nil then
if (x^.right=nil) and (k=t+1) then
begin
x^.rtag := false;
x^.right:=head;
write (x^.data,'->');
write (head^.data);        end;       end;
```

39. СТРУКТУРА УЗЛА ПРОШИТОГО ДЕРЕВА. ПРОЦЕДУРА ОБХОДА ПРЯМО ПРОШИТОГО БИНАРНОГО ДЕРЕВА

Эффективность прохождения дерева рекурсивными и нерекурсивными алгоритмами может быть увеличена, если использовать пустые указатели на отсутствующие поддеревья для хранения в них адресов узлов преемников, которые надо посетить при заданном порядке прохождения бинарного дерева. Такой указатель называется нитью. Его следует отличать от указателей в дереве, которые используются с левым и правым поддеревьями. Операция, заменяющая пустые указатели на нити, называется прошивка. Она может выполняться по-разному. Если нити заменяют пустые указатели в узлах с пустыми правыми поддеревьями, при просмотре в симметричном порядке, то бинарное дерево называется симметрично прошитым справа.Похожим образом может быть определено бинарное дерево, симметрично прошитое слева:дерево, в котором каждый пустой левый указатель изменен так, что он содержит нить – связь к предшественнику данного узла при просмотре в симметричном порядке. Симметрично прошитое бинарное дерево – это то, которое симметрично прошито слева и справа. Однако левая прошивочная нить не дает тех преимуществ, что правая прошивочная нить. Также используются бинарные деревья, прямо прошитые справа и слева. В них пустые правые и левые указатели узлов заменены соответственно на их преемников и предшественников при прямом порядке просмотра. Поскольку нужно каким-то образом отличать обычную связь от прошивочной нити, каждому узлу добавляется два однобитовых (логических) поля тэга: ltag и rtag. Если значение тэга true, соответствующее поле связи является обычной связью, в случае значения false– прошивочной нитью.

Узел прошитого бинарного дерева имеет иную структуру

```
Type ptr = ^node;
node = record
info : integer;   {Информационноеполе}
      ltag, rtag : boolean; {Тэгипрошивочныхнитей}
      left, right : pt;  {Указатели на левое и правое поддеревья}
    end;
```

Логические поля в прошитом дереве могут принимать следующие значения:

1. ltag=true, следовательно, left представляет собой обычную связь.
2. ltag=false, следовательно, left указывает на узел-предшественник.
3. rtag=true, следовательно, right представляет собой обычную связь.
4. rtag=false, следовательно, rightуказывает на узел-преемник.

*Наряду **с преимуществами прошитых деревьев**: быстрый обход, отсутствие необходимости в стеке, можно определить предшественника и преемника вершины, существуют **недостатки**. Включение новой вершины в дерево занимает больше времени, т.к. необходимо поддерживать два типа связей: структурные и по нитям. Поэтому прошитые деревья целесообразно использовать в тех задачах, где изменения в деревьях происходят редко, а обходы выполняется часто.*

Процедура обхода симметрично прошитого бинарного дерева:

Алгоритм обхода прошитого дерева можно сформулировать следующим образом:

1. Обработка узла p, например, печать p^.info.

2. До тех пор, пока p^.left<>nil, повторять: p := p^.left, то есть идти по левой ветви до самого левого узла.

3. Переход к корню дерева (p := HEAD^.left).

4. Если p^.rtagравен false, то p := p^.right и переход к шагу 3 (к преемнику). Иначе p:=p^.right и переход к шагу 2.

Алгоритм заканчивает работу, когда p станет равным HEAD.

40. МЕТОД ПРЕДСТАВЛЕНИЯ СООБЩЕНИЙ КОДАМИ ХАФФМАНА
41. ЭТАПЫ СОЗДАНИЯ ДЕРЕВА ХАФФМАНА ДЛЯ ЗАДАННЫХ СООБЩЕНИЙ

Мы имеем сообщения, состоящие из последовательности символов. В каждом сообщении символы независимы и появляются с известной вероятностью, не зависящей от позиции в сообщении. Например, мы имеем сообщения, состоящие из пяти символов a, b, c, d, e, которые появляются в сообщениях с вероятностями 0.12, 0.4, 0.15, 0.08 и 0.25 соответственно.

Задача конструирования кодов Хаффмана заключается в следующем: имея множество символов и значения вероятностей их появления в сообщениях, построить такой код с префиксным свойством, чтобы средняя длина кода (в вероятностном смысле) последовательности символов была минимальной. Мы хотим минимизировать среднюю длину кода для того, чтобы уменьшить длину вероятного сообщения, т.е. чтобы сжать сообщение. Появляется вопрос. Можно ли придумать код, который был бы лучше второго кода? Ответ положительный: существует код с префиксным свойством, средняя длина которого равна 2.15. Это наилучший возможный код с теми же вероятностями появления символов. Способ нахождения оптимального префиксного кода называется алгоритмом Хаффмана. В этом алгоритме находятся два символа a и bc наименьшими вероятностями появления и заменяются одним фиктивным символом, например x, который имеет вероятность появления, равную сумме вероятностей появления символов a и b. Затем, используя эту процедуру рекурсивно, находим оптимальный префиксный код дляменьшего множества символов (где символы a и bзаменены одним символом x). Код для исходного множества символов получается из кодов замещающих символов путем добавления 0 и 1 перед кодом замещающего символа, и эти два новых кода принимаются как коды заменяемых символов. Например, код символа a будет соответствовать коду символа x с добавленным нулем перед этим кодом, а для кода символа bперед кодом символа x будет добавлена единица.

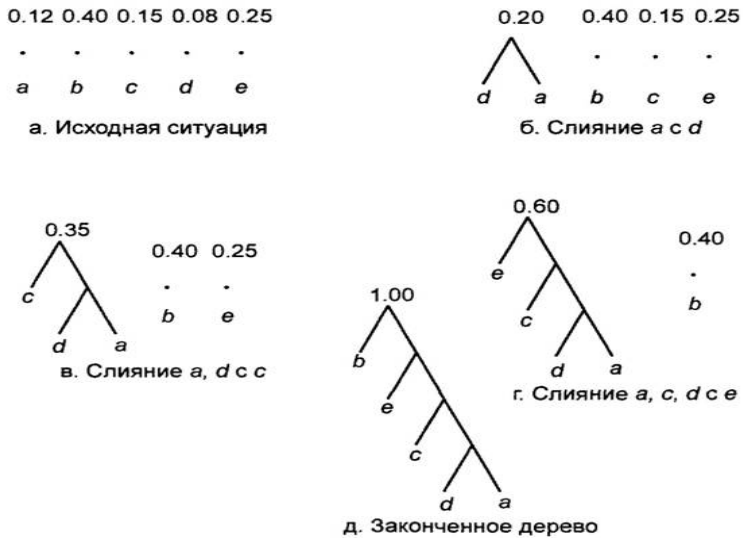
Можно рассматривать префиксные коды как пути на двоичном дереве: прохождение от узла к его левому сыну соответствует 0 в коде, а к правому сыну – 1. Если мы пометим листья дерева кодируемыми символами, то получим представление префиксного кода в виде двоичного дерева. Префиксное свойство гарантирует, что нет символов, которые были бы метками внутренних узлов дерева (не листьев), и наоборот, помечая кодируемыми символами только листья дерева, мы обеспечиваем префиксное свойство кода этих символов.

Для реализации алгоритма Хаффмана используется лес, т.е. совокупность деревьев, чьи листья помечаются символами, для которых разрабатывается кодировка, а корни помечаются суммой вероятностей всех символов, соответствующих листьям дерева. Эти суммарные вероятности называются весом дерева. Вначале каждому символу соответствует дерево, состоящее из одного узла, в конце работы алгоритма получается одно дерево, все листья которого будут помечены кодируемыми символами. В результирующем дереве путь от корня к любому листу представляет код для символа-метки этого листа, составленный по схеме, согласно которой левый сын узла соответствует 0, а правый – 1.

Важным этапом в работе алгоритма является выбор из леса двух деревьев с наименьшими весами. Эти два дерева комбинируются в одно с весом, равным сумме весов составляющих деревьев. При слиянии деревьев создается новый узел, который становится корнем объединенного дерева и который имеет в качестве левого и правого сыновей корни старых деревьев. Этот процесс продолжается до тех пор, пока не получится только одно дерево. Это дерево соответствует коду, который при заданных вероятностях имеет минимально возможную среднюю длину.

Рассмотрим на примере последовательные шаги выполнения алгоритма Хаффмана. Кодируемые символы и их вероятности заданы в таблице. **Этапы построения дерева Хаффмана показаны на рис. 2.**

На рисунке видно, что символы a, b, c, di e получили соответственно коды 1111, 0, 110, 1110 и 10. В этом примере существует только одно нетривиальное дерево, соответствующее оптимальному коду, но в общем случае их может быть несколько. Например, если бы символы bи e имели вероятности соответственно 0.33 и 0.32, то после шага алгоритма, показанного на рис. 2.в, можно было бы комбинировать bи e, а не присоединять e к большому дереву, как это сделано на рис. 2.г.



Для представления бинарных деревьев используется массив TREE, состоящий из записей следующего типа:

```
ype
    Tree_uz=record
        leftchild : integer;
        rightchild : integer;
        parent : integer
    end;
var  Tree : array[1..100] of Tree_uz;
```

Этот массив облегчает поиск путей от листа к корню при записи кода символа. Также используется массив ALPHABET, в котором каждому символу, подлежащему кодированию, ставится в соответствие вероятность его появления и лист, меткой которого он является. Массив ALPHABETсостоит из записей, имеющих следующий тип:

```
Type ALPHABET_elem  =record
        symbol : char;
        probability : real;
        leaf : integer
    end;
var ALPHABET : array[1..100] of ALPHABET_elem;
```

Для представления непосредственно деревьев необходим массив FOREST, состоящих из записей, имеющих тип:

```
Type   FOREST_elem  =record
        weight : real;
        root : integer
    end;
var FOREST: array[1..100] of FOREST_elem;
```

После завершения работы алгоритма код каждого символа можно определить следующим образом. Найти в массиве ALPHABETзапись с нужным символом в поле symbol. Затем по значению поля leaf этой же записи определить местоположение записи в массиве TREE, которая соответствует листу, помеченному рассматриваемым символом. Далее нужно последовательно переходить по указателю parentот текущей записи, например соответствующей узлу n, к записи в массиве TREE, соответствующей его родителю p. По родителю p определяют, в каком его поле,leftchildили rightchild, находится указатель на узел n, т.е. является ли узел n левым или правым сыном, и в соответствии с этим печатается 0 (для левого сына) или 1(для правого сына). Затем выполняется переход к родителю узла p и определяется, является ли его сын p правым или левым, и в соответствии с этим печатается следующая 1 или 0. Таки образом продолжается до корня дерева. В результате код символа будет напечатан в виде последовательности битов, но в обратном порядке. Чтобы распечатать полученную последовательность в прямом порядке, нужно каждый очередной бит помещать в стек, а затем распечатать содержимое стека в обычном порядке.

42. ИДЕАЛЬНО СБАЛАНСИРОВАННОЕ БИНАРНОЕ ДЕРЕВО. ПРАВИЛА ПОСТРОЕНИЯ. ДОСТОИНСТВА И НЕДОСТАТКИ. ПРИВЕСТИ ПРИМЕР ТАКОГО ДЕРЕВА

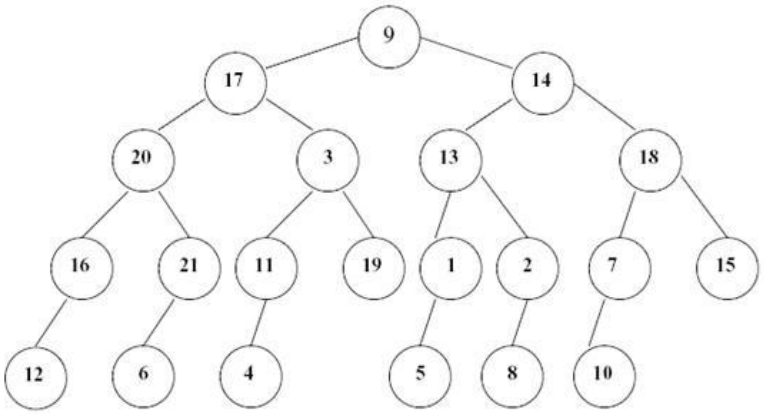
Поскольку максимальный путь до листьев дерева определяется высотой дерева, то при заданном числе узлов дерева его стремятся построить минимальной высоты. Этого можно добиться, если размещать максимально возможное количество узлов на всех уровнях, кроме последнего. В случае бинарного дерева это достигается за счет того, что все поступающие при построении дерева узлы распределяются поровну слева и справа от каждого из вышерасположенных узлов.

Бинарное дерево идеально сбалансировано, если для каждого его узла количество потомков в левом и правом поддеревьях различается не более чем на 1.

Рассмотрим **алгоритм построения идеально сбалансированного бинарного дерева**. Если количество узлов дерева известно, и дана последовательность значений его вершин a[1], a[2],…a[n], то можно применить следующий рекурсивный алгоритм построения идеально сбалансированного бинарного дерева.

1. Начиная с a[1], выбираем очередное a[i] в качестве значения корня дерева (поддерева).
2. Тем же способом строим левое поддерево с количеством узлов Nl=n/2
3. Строим правое поддерево с количеством узлов Nr=N=Nl-1.

Таким образом, значение a[1] окажется в корне дерева, и именно на него будет ссылаться указатель дерева. Значения $a[2], a[3], \ldots a[n_l]$ попадут в левое поддерево, а значения $a[n_l+1], a[n_l+2], \ldots a[n]$ – в правое поддерево. Следовательно, распределение значений по узлам дерева полностью определяется исходной последовательностью данных. На рис. 1 показано идеально сбалансированное бинарное дерево, построенное по следующему набору значений узлов: 9, 17, 20, 16, 12, 21, 6, 3, 11, 4, 19, 14, 13, 1, 5, 2, 8, 18, 7, 10, 15.



При таком построении идеально сбалансированного дерева на него не накладываются никакие требования относительно значений в узлах, т.е. значения данных в узлах в общем случае не упорядочены. **Это ведет к тому, что поиск узла с нужными данными осуществляется последовательным обходом всех узлов дерева, и в общем случае время поиска будет прямо пропорционально количеству узлов дерева**. Поэтому на практике обычно применяются сбалансированные деревья поиска, которые обеспечивают минимальное время поиска порядка $\log_2 n$, где n – количество узлов дерева.

Добавление и удаление узлов идеально сбалансированного дерева вызывают некоторые трудности. Чтобы поддерживать сбалансированность дерева при добавлении и удалении узлов, необходимо иметь информацию о сбалансированности каждого поддерева и поддерживать ее. Поэтому идеально сбалансированные деревья применяются для работы с данными, которые мало изменяются в процессе обработки.

43. СБАЛАНСИРОВАННОЕ БИНАРНОЕ ДЕРЕВО. СРАВНИТЬ С ИДЕЛЬНО СБАЛАНСИРОВАННЫМ ДЕРЕВОМ. ПРИВЕСТИ ПРИМЕР ТАКИХ ДЕРЕВЬЕВ

Время поиска в бинарном дереве поиска определяется высотой самого дерева. Для заданного числа элементов nвысота дерева зависит от порядка поступления элементов при построении дерева и может колебаться от $\log_2 n$ в случае идеальной сбалансированности дерева до n-1 в случае вырождения дерева в линейный список. Следовательно, затраты на поиск и включение будут порядка от $\log_2 n$ доп.

Доказано, что при случайном порядке включения элементов в дерево средние затраты на поиск элемента будут $1.386*\log_2 n$. Увеличение затрат на 39% объясняется простыми средствами поддержания обычного дерева поиска по сравнению с идеально сбалансированным деревом поиска. Однако при работе с большими деревьями свойство случайности распределения поступающих данных редко соблюдается – это, скорее, исключение, чем правило. Обычно выходные последовательности являются частично упорядоченными. Для таких последовательностей наиболее подходящими оказываются **сбалансированные деревья поиска**. Рассмотрим один из видов таких деревьев: **АВЛ** – деревья, предложенные в 1962 г. Адельсоном-Вельским и Ландисом.

Требования к сбалансированности в АВЛ-деревьях менее жесткие, чем в идеально сбалансированных деревьях.

АВЛ-деревом называется такое дерево, у которого высота поддеревьев для каждой вершины различается не более чем на 1.

Максимальная высота АВЛ-дерева с nвершинами не превосходит 1.44* $\log_2(n+1) \quad 1.33,$ т.е. затраты на поиск не превосходят $1.45*\log_2 n$.

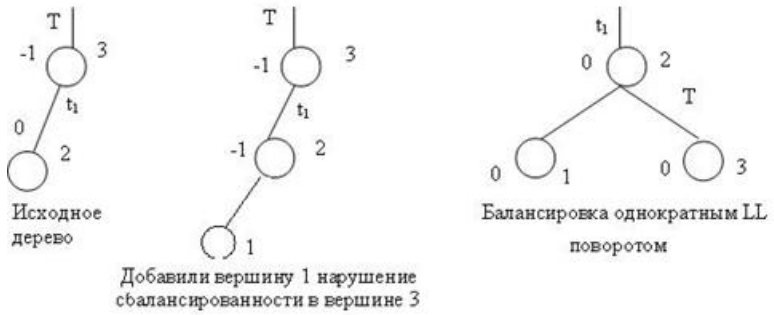
Отличие АВЛ-дерева от обычного дерева поиска заключается в том, что при включении и удалении элементов необходимо поддерживать сбалансированность дерева в целом. Для этого в каждый узел дерева добавляется одно вспомогательное поле, содержащее информацию о равновесности поддеревьев (показатель сбалансированности узла). Его значениями могут быть: 0 – высоты правого и левого поддеревьев равны; 1 – высота правого поддерева больше; -1 – высота левого поддерева больше.

При попытке добавить или удалить элемент в поддереве с показателем сбалансированности, отличным от 0, дерево может стать несбалансированным, и потребуется операция балансировки.

44. ВСТАВКА ЭЛЕМЕНТА В АВЛ-ДЕРЕВО. ПРИВЕСТИ ПРИМЕР

Операция включения элемента а АВЛ-дерево выполняется в два этапа. Поскольку сбалансированное дерево является частным случаем обычного дерева поиска, то на первом этапе выполняется включение элемента в дерево точно так же, как и при включении в обычное дерево: проходом по пути поиска, получением динамической памяти и формированием в ней новой вершины дерева. Дополнительно новой вершине устанавливается признак ее сбалансированности, равный 0, т.к. новая вершина не имеет поддеревьев. Второй этап выполняется при обратном движении по дереву и заключается в восстановлении сбалансированности в узлах дерева, если она была нарушена. При этом учитывается, откуда (слева или справа) осуществляется возврат в вершину, каков показатель сбалансированности в ней, выросла ли высота поддерева. Возникающие ситуации и особенности восстановления сбалансированности рассмотрим на примерах.

Пусть имеется дерево с вершинами 3 и 2 (рис. 3, первое дерево). Высота h1 левого поддерева вершины 3 равна 1, высота hr правого поддерева вершины 3 равна 0, сбалансированность $bal_3 = -1.$ У вершины 2 $h_l = h_r = 0, bal_2 = 0.$ Добавим вершину 1, у нее $h_l = h_r = 0, bal_1 = 0.$ Возвращаемся в вершину 2, теперь у нее $h_l = 1, h_r = 0, bal_2 = 1,$ но критерий сбалансированности поддерева соблюдается. В вершине 3 стало $h_l = 2, h_r = 0,$ т.е. критерий сбалансированности нарушен и дерево нужно перестраивать (среднее дерево на рис. 3). Это достигается однократным LL-поворотом, в результате получаем сбалансированное дерево с вершиной 2 (правое дерево на рис. 3). Необходимые операции балансировки заключаются в обмене указателями t_i иТ по кругу по часовой стрелке. Кроме этого, необходимо изменить показатели сбалансированности вершин.



Теперь в дерево с вершинами 4 и 5 включим вершину 6. Для балансировки в вершине 4 выполняются RR-поворот, обмен указателями по кругу против часовой стрелки (рис. 4).

Более сложные ситуации приводят к двукратным поворотам: направо и налево – RL-поворот; налево и направо – LR-поворот. Пример двукратного RL-поворота демонстрируется включением в дерево с вершинами 5 и 7 новой вершины 6 (рис. 5). Возникла несбалансированность в вершине 5. Поэтому вначале выполняется правый поворот трех вершин, затем левый поворот двух вершин (7 и 6).

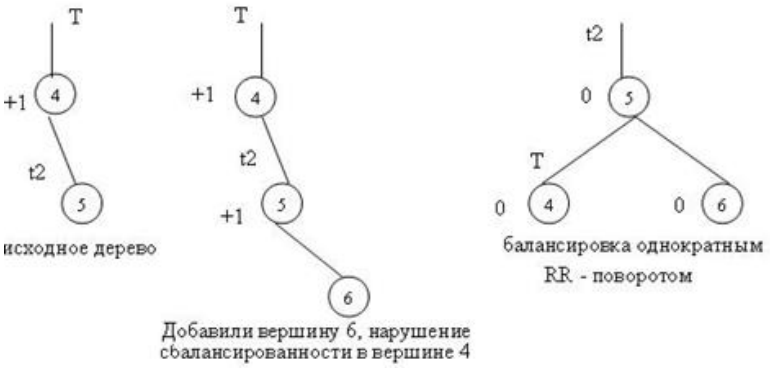


Рис. 4 – Вставка элемента в АВЛ-дерево и его балансировка RR-поворотом

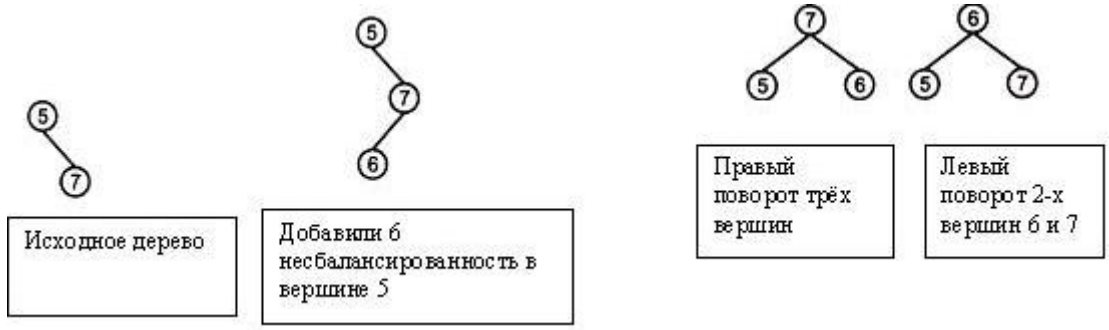


Рис. 5 – Вставка элемента в АВЛ-дерево и его балансировка RL-поворотом

При включении в дерево с вершинами 9 и 3 новой вершины 8 возникает несбалансированность поддерева с корнем. Она устраняется двукратным LR-поворотом: сначала левым поворотом трех вершин, а затем правым поворотом двух вершин – 3 и 8 (рис. 6).

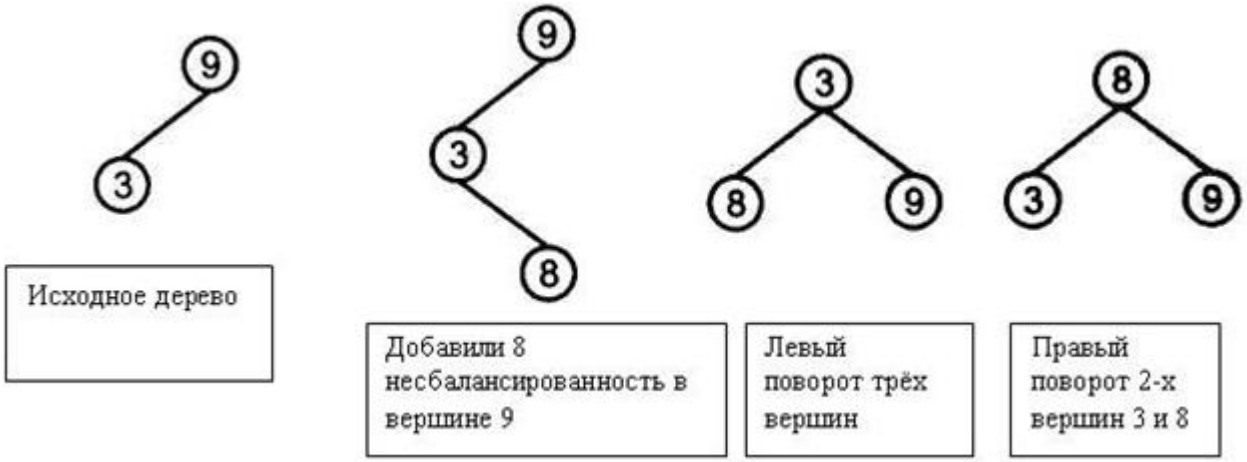
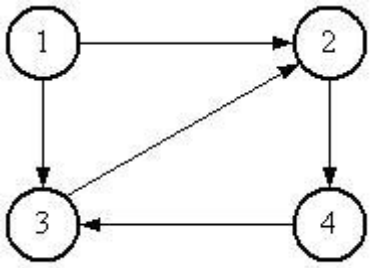


Рис. 6 – Вставка элемента в АВЛ-дерево и его балансировка LR-поворотом

45. ОСНОВНЫЕ ОПРЕДЕЛЕНИЯ ОРИЕНТИРОВАННЫХ ГРАФОВ: ВЕРШИНА, ДУГА, ПУТЬ, ЦИКЛ. ПОМЕЧЕННЫЙ ОРГРАФ. ПРИВЕСТИ ПРИМЕРЫ. МАТРИЦЫ СМЕЖНОСТИ И ИНЦИДЕНТНОСТИ



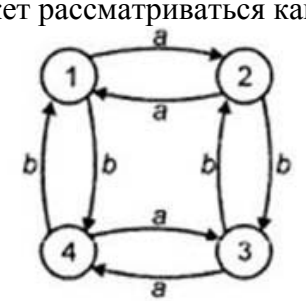
Ориентированный граф (или орграф) G=(V, E) состоит из множества вершин Vи множества дуг E. Вершины также называют узлами, а дуги – ориентированными ребрами. **Дуга** представляется в виде упорядоченной пары вершин (v, w), где вершина v называется началом, а w– концом дуги. Дугу (v, w) часто записывают как v → w Кроме того, дугаv → wведет от вершины v к вершине w, а вершина w смежная с вершиной v. Рис. 1– Пример орграфа

Вершины орграф можно использовать для представления объектов, а дуги – для отношений между объектами. Например, вершины орграфа могут представлять города, а дуги – маршруты рейсовых полетов самолетов из одного города в другой. В виде орграфа может быть представлена блок-схема потока данных в компьютерной программе. В последнем примере вершины соответствуют блокам операторов программы, а дугам – направленное перемещение потоков данных. **Путем** в орграфе называется последовательность вершин $v_1, v_2, \ldots v_{n-1}, v_n$, для которых существуют дуги $v_1 \to v_2, v_2 \to v_3, \ldots v_{n-1} \to v_n$. Этот путь начинается в вершине v_1 и, проходя через вершины $v_2, \ldots v_{n-1}$, заканчивается в вершине v_n. **Длина пути** – количество дуг, составляющих путь, в данном случае длина пути равна n–1. Одна вершина рассматривается как путь длины 0 от вершины vк этой же вершине v. Путь называется простым, если все вершины на нем, за исключением, может быть, первой и последней, различны. **Цикл** – это простой путь длины не менее 1, который начинается и заканчивается в одной и той же вершине. На рисунке 1 вершин 3, 2, 4, 3 образуют цикл длины 3.

Во многих приложениях удобно к вершинам и дугам присоединить какую-либо информацию. Для этих целей используется **помеченный граф**, т.е. орграф, у которого каждая дуга и/или каждая вершина имеет соответствующие метки. Меткой может быть имя, вес или стоимость (дуги), или значение данных какого-либо заданного типа.

Одним из часто используемых способов представления орграфа G=(V, E) является **матрица смежности.** Предположим, что множество вершин орграфа V={1, 2, … n}, тогда матрица смежности графа G – это матрица A размера n× nco значениями булевого типа, где A[i, j]=trueтогда и только тогда, когда существует дуга из вершины i в вершину j. Часто в матрице смежности значение true заменяется на 1, а значение false – на 0. Время доступа к элементам матрицы смежности зависит от размеров множества вершин и множества дуг. Представление орграфа в виде матрицы смежности удобно применять в тех алгоритмах, в которых надо часто проверять существование данной дуги.

С помощью матрицы смежности можно представлять и помеченные орграфы. В этом случае элемент A[i, j] равен метке дуги i → j. Если дуги от вершины i к вершине j не существует, то значение A[i, j] может рассматриваться как пустая ячейка



	1	2	3	4
1		a		b
2	a		b	
3		b		a
4	b		a	

Основной недостаток матриц смежности заключается в том, что она требует объема памяти, равного N^2 даже если дуг значительно меньше, чем N^2. Поэтому для чтения матрицы или нахождения в ней необходимого элемента требуется время порядка N^2, что не позволяет создавать алгоритмы с временем nдля работы с орграфами, имеющими порядка n дуг.

Поэтому вместо матриц смежности могут использовать представления орграфов с помощью **списков смежности**. Списком смежности для вершины iназывается список всех вершин, смежных с вершиной i, причем упорядоченный определенным образом.

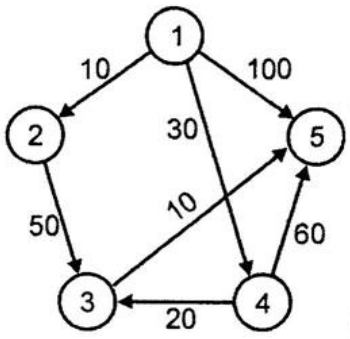
46. НАХОЖДЕНИЕ КРАТЧАЙШЕГО ПУТИ НА ОРГРАФЕ С ПОМОЩЬЮ АЛГОРТМА ДЕЙКСТРЫ. ПРИВЕСТИ ПРИМЕР

Пусть есть ориентированный граф G=(V, E), у которого все дуги имеют неотрицательные метки, а одна вершина определена как источник. Задача состоит в нахождении стоимости кратчайших путей от источника ко всем другим вершинам граф G. Длина пути определяется как сумма стоимостей дуг, составляющих путь. Эта задача часто называется задачей нахождения кратчайшего пути с одним источником. При этом длина пути может измеряться даже в нелинейных единицах, например во временных. Для решения поставленной задачи будем использовать а**лгоритм Дейкстры**. Алгоритм строит множество S вершин, для которых кратчайшие пути от источника уже известны. На каждом шаге к множеству S добавляется та из оставшихся вершин, расстояние до которой от источника меньше, чем для других оставшихся вершин. Если стоимости всех дуг неотрицательны, то кратчайший путь от источника к конкретной вершине проходит только через вершины множества S. Такой путь называют особым. На каждом шаге алгоритма используется массив D, в который записываются длины кратчайших особых путей для каждой вершины. Когда множество S будет содержать все вершины орграфа, т.е. для всех вершин будут найдены особые пути, тогда массив D будет содержать длины кратчайших путей от источника к каждой вершине. Ниже приведен фрагмент программного кода алгоритма Дейкстры. FL – массив типа Boolean. Если значение его элемента равно false, то соответствующий ему элемент массива D должен проверяться на предмет поиска кратчайшего пути, в противном случае этот элемент больше не участвует в рассмотрении.

```
    Form:=1 to n do
  P[m]:=1;  {P – массив, используемый для
построение кратчайших путей}
    For j:=1 to n do
     D[j]:=C[i,j]; {C -  матрицацен}
      For j:=2 to n do
    Begin
          m:=2;
           While fl[m]= true do m:=m+1;
           w:=D[m];
          s[j]:=m;
         For k:=m+1 to n do
           iffl[k]=false then
           if D[k]< w then
            begin
              w:=D[k];
              s[j]:=k;
            end;
          v:=s[j];
          fl[v]:=true;
          for i:=1 to n do begin
           if D[i]> D[v]+C[v,i] then P[i]:=v;
           D[i]:=min(D[i], D[v]+C[v,i]);
          end;
end;
```

Здесь предполагается, что в орграфе G вершины поименованы целыми числами, т.е. множество вершин V={1, 2, … n}, причем вершина 1 является источником. Массив C – это двумерный массив стоимостей, где элементC[i, j] равен стоимости дуги $i \to j$. Если дуги $i \to j$ не существует, то C[i, j] присваивается значение ∞, т.е. большее любой фактической стоимости дуг. На каждом шаге D[i] содержит длину текущего кратчайшего особого пути к вершине i.

Применим алгоритм Дейкстры для ориентированного графа, показанного на рис. 4. Вначале S={1}, D[2]=10, D[3]=∞, D[4]=30, D[5]=100. На первом шаге цикла w=2, т.е. вершина 2 имеет минимальное значение в массиве D. Затем вычисляется D[3]=min(∞, 10+50)=60. D[4] и D[5] не изменяются, т.к. не существует дуг, исходящих из вершины 2 и ведущих к вершинам 4 и 5. Последовательность значений элементов массива D после каждой итерации цикла показаны в таблице ниже.



Итерация	S	w	D[2]	D[3]	D[4]	D[5]
Начало	{1}	–	10	∞	30	100
1	{1, 2}	2	10	60	30	100
2	{1, 2, 4}	4	10	50	30	90
3	{1, 2, 4, 3}	3	10	50	30	60
4	{1, 2, 4, 3, 5}	5	10	50	30	60

В рассмотренный алгоритм можно внести изменения, которые позволят определить кратчайший путь для любой вершины графа. Для этого нужно ввести еще один массив P, где P[v] содержит вершину, непосредственно предшествующую вершине vв кратчайшем пути. Вначале P[v]=1 для всех $v \neq 1$. В листинге алгоритма Дейкстры при выполнении условного оператора D[v]+C[v,i]<D[i], элементу P[i] присваивается значение v. После выполнения алгоритма кратчайший путь к каждой вершине можно найти с помощью обратного прохождения по предшествующим вершинам массива P.

Для рассмотренного орграфа массив P имеет следующие значения: P[2]=1, P[3]=4, P[4]=1, P[5]=3. Для определения кратчайшего пути, например от вершины 1 к вершине 5, надо отследить в обратном порядке предшествующие вершины, начиная с вершины 5. Таким образом, кратчайший путь из вершины 1 в вершину 5 составляет последовательность вершин: 1, 4, 3, 5.

47. НАХОЖДЕНИЕ КРАТЧАЙШИХ ПУТЕЙ МЕЖДУ ПАРАМИ ВЕРШИН

Пусть дан орграф *G=(V, E)* и необходимо определить кратчайшие пути между всеми парами вершин орграфа. Каждой дуге $v \to w$ этого графа сопоставлена неотрицательная стоимость *C[v, w]*.

Алгоритм Флойда использует матрицу *A* размера $n \times n,$ в которой вычисляются длины кратчайших путей. Вначале *A[i, j]= C[i, j]* для всех $i \neq j.$ Если дуга $i \to j$ отсутствует, то *C[i, j]*=∞. Каждый диагональный элемент матрицы *A* равен 0.

$$A_k[i,j] = \min(A_{k-1}[i,j], A_{k-1}[i,k] + A_{k-1}[k,j]).$$

Нижний индекс *к* обозначает значение матрицы *A* после *к*-ой итерации. Графическая интерпретация приведенной формулы показана на рис. 7.5.

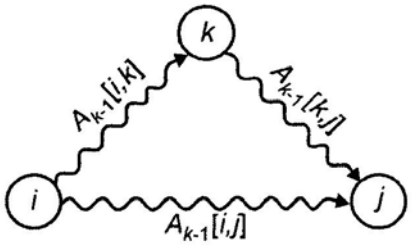


Рис. 7.5 – Включение вершины *к* в путь от вершины *i* к вершине *j*

На рис. 7.6 приведен помеченный орграф, а на рис. 7.7 – значения матрицы *A* после трех итераций.

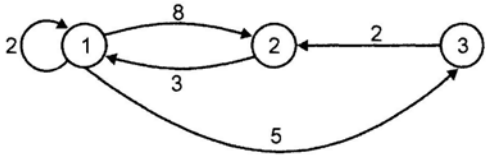


Рис. 7.6–Помеченный орграф

	1	2	3
1	0	8	5
2	3	0	∞
3	∞	2	0

$A_0[i, j]$

	1	2	3
1	0	8	5
2	3	0	8
3	∞	2	0

$A_1[i, j]$

	1	2	3
1	0	8	5
2	3	0	8
3	5	2	0

$A_2[i, j]$

	1	2	3
1	0	7	5
2	3	0	8
3	5	2	0

$A_3[i, j]$

Рис.7.7–Последовательные значения матрицы *A*

Равенства $A_k[i,k] = A_{k-1}[i,k]$ и $A_k[k,j] = A_{k-1}[k,j]$ означает, что на *к*-ой итерации элементы матрицы *A*, стоящие в *к*-ой строке и *к*-ом столбце, не изменяются. Процедура, реализующая алгоритм Флойда, представлена ниже.

```
For i:=1 to n do
  For j:=1 to n do
    readln(C[i,j]);
  For i:=1 to n do
     For j:=1 to n do
      A[i,j]:=C[i,j];
  For i:=1 to n do
      A[i,i]:=0;
For k:=1 to n do
  For i:=1 to n do
    For j:=1 to n do
    if A[i,k]+ A[k,j]< A[i,j] then
       A[i,j]:= A[i,k]+ A[k,j];
End.
```

Флойд, Дейкстры – сложность n^3

Если *e*, количество дуг в орграфе, значительно меньше, чем n^2, то рекомендуют применять алгоритм Дейкстры со списками смежности, порядок - $ne * \log_2 n,$

48. ТРАНЗИТИВНОЕ ЗАМЫКАНИЕ НА ОРГРАФЕ

Найти транзитивное замыкание графа по алгоритму Уоршелла.

Анализ условия:

Так как матрица может содержать только 0 и 1, ввод других символов мог бы привести к ошибке. Тогда для удобства ввода за единицу был принят любой символ (а не только 1).

Мат. постановка:

По определению, матрица транзитивного замыкания- это матрица смежности согласованного отношения, обладающего свойством транзитивности.

Стандартный алгоритм нахождения транзитивного замыкания заключается в следующем:

A-матрица смежности m*m, тогда A(m-1)-матрица транзитивного замыкания.

Алгоритм Уоршелла является улучшением этого алгоритма.

Заключается он в следующем:

Если существует путь из вершины A в вершину B через третью вершину, то проводим путь непосредственно из A в B.

R - отношение. И uRv uRw, wRv.

Тогда A[u,v]=max{A[u,v], A[u,w]*A[w,v]}

Формально алгоритм можно записать так:

```
Для всех W из M
   Для всех V из M
      Для всех V из M
         A[U,V]=A[U,V] или A[U,W]*A[W,V]
```

Пример:

Есть матрица смежности графа:

```
   1 1 0
A=   0 1 1
   1 0 1
```

```
W=1 A[i,1]*A[1,j]
   A[1,3]
   A[2,1]
   A[3,2]=A[3,1]*A[1,2]=1 дуга исправится
W=2 A[i,2]*A[2,j]
   A[1,3]=A[1,2]*A[2,3]=1 дуга исправится
W=3 A[2,1]=A[2,3]*A[3,1]=1 дуга исправится
```

49. НАХОЖДЕНИЕ ЦЕНТРА ОРИЕНТИРОВАННОГО ГРАФА

Определим понятие *центральной вершины* орграфа. Пусть v - произвольная вершина орграфа *G=(V, E)*. Эксцентриситет (максимальное удаление) вершины *v* определяется как *max{минимальная длина пути от вершины w до вершины v }*.

Центром орграфа G называется вершина с минимальным эксцентриситетом, т.е. это вершина, для которой максимальное расстояние (длина пути) до других вершин минимально.

Рассмотрим помеченный орграф, показанный на рис. 7.8.

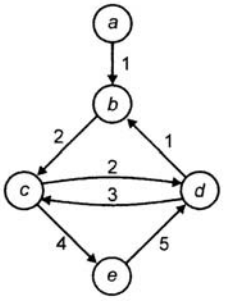


Рис. 7.8 – Помеченный орграф

В этом графе вершины имеют следующие эксцентриситеты.

Вершина	Эксцеитриситет
a	∞
b	6
c	8
d	5
e	7

Откуда видно, что центром данного орграфа является вершина *d*.

Пусть *C* – матрица стоимостей для орграфа *G*. Тогда центр орграфа можно найти, применив следующий алгоритм.

1. Применить алгоритм Флойда к матрице *C* для вычисления матрицы *A*, содержащей все кратчайшие пути орграфа *G*.

2. Найти максимальное значение в каждом столбце *i* матрицы *A*. Это значение равно эксцентриситету вершины *i*.

3. Найти вершину с минимальным эксцентриситетом. Она и будет центром графа *G*.

Матрица всех кратчайших путей для орграфа из рис. 7.8 представлена на рис. 7.9. Максимальные значения в каждом столбце приведены под матрицей.

	a	*b*	*c*	*d*	*e*
a	0	1	3	5	7
b	∞	0	2	4	6
c	∞	3	0	2	4
d	∞	1	3	0	7
e	∞	6	8	5	0
max	∞	6	8	5	7

Рис. 7.9 – Матрица кратчайших путей

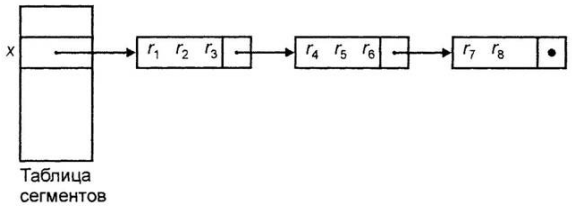
50. ОСОБЕННОСТИ АЛГОРИТМОВ ДЛЯ ВНЕШНЕЙ ПАМЯТИ.ХЕШИРОВАННЫЕ ФАЙЛЫ

Природа устройств внешней памяти такова, что время, необходимое для поиска блока и чтения его в основную память, достаточно велико по сравнению со временем, которое требуется для относительно простой обработки данных, содержащихся в этом блоке.

Оценивая время работы алгоритмов, в которых используются данные, хранящиеся в виде файлов, приходится в первую очередь учитывать количество обращений к блокам, т.е. сколько раз блок считывается в основную память или записывается во вторичную память. Такая операция называется доступом к блоку. Поскольку размер блока фиксирован в операционной системе, нет возможности ускорить работу алгоритма, увеличив размер блока и сократив тем самым количество обращений к блокам. Поэтому мерой качества алгоритма, работающего с внешней памятью, является количество обращений к блокам.

Все ранее рассмотренные алгоритмы предназначены для работы с оперативной памятью. При обработке больших объемов данных их приходится хранить во внешней памяти. В результате усложняется доступ к данным. Основной единицей хранения данных является файл. Его можно рассматривать как связный список блоков. В свою очередь блок состоит из записей. И в каждый блок помещается целое количество значений. Базовыми операциями выполняемыми по отношению к файлу является перенос блока в буфер находящийся в оперативной памяти. Буфер – зарезервированная область в оперативной, соответствующая размеру блока. По окончанию обращения блок возвращается из буфера во внешний накопитель.

Хеширование – распространенный метод обеспечения быстрого доступа к информации, хранящейся во вторичной памяти. Основная идея этого метода подобна открытому хешированию, рассмотренному ранее. Записи файла распределяются между сегментами, каждый из которых состоит из связного списка одного или нескольких блоков внешней памяти.



Имеется таблица сегментов, содержащая B указателей, – по одному на каждый сегмент. Каждый указатель в таблице сегментов представляет собой физический адрес первого блока связного списка блоков для соответствующего сегмента. Сегменты пронумерованы от 1 до B. Хеш-функция h отображает каждое значение ключа в одно из целых чисел от 1 до B. Если x – ключ, то h(x) является номером сегмента, который содержит запись с ключом x, если такая запись вообще существует. Блоки, составляющие каждый сегмент, образуют связный список. Таким образом, заголовок i-ого блока содержит указатель на физический адрес (i+1)-ого блока. Последний блок сегмента содержит в своем заголовке nil-указатель. Такой способ организации показан на рис. 1. При этом в данном случае элементы, хранящиеся в одном блоке сегмента, не требуется связывать друг с другом с помощью указателей, связывать между собой нужно только блоки.

Если размер таблицы сегментов невелик, ее можно хранить в основной памяти, иначе ее можно хранить последовательным способом в отдельных блоках. Если требуется найти запись с ключом x, вычисляется h(x) и находится блок таблицы сегментов, содержащий указатель на первый блок сегмента h(x). Затем последовательно считываются блоки сегмента h(x), пока не обнаружится блок, содержащий запись с ключом x. Если исчерпаны все блоки в связном списке для сегмента h(x), делается вывод, что x не является ключом ни одной из записей.

Такая структура оказывается эффективной, если в выполняемом операторе указываются значения ключевых полей. Среднее количество обращений к блокам, требующееся для выполнения оператора, в котором указан ключ записи, приблизительно равняется среднему количеству блоков в сегменте, которое равно n/bk, если n– количество записей, блок содержит bзаписей, a kсоответствует количеству сегментов. В результате, при такой

организации данных операторы, использующие значения ключей, выполняются в среднем в к раз быстрее, чем в случае неорганизованного файла.

Чтобы вставить запись с ключом, значение которого равняется x, нужно сначала проверить, нет ли в файле записи с таким значением ключа. Если такая запись есть, то выдается сообщение об ошибке, поскольку предполагается, что ключ уникальным образом идентифицирует каждую запись. Если записи с ключом x нет, новая запись вставляется в первый блок цепочки для сегмента h(x), в который ее удается вставить. Если запись не удается вставить ни в один из существующих блоков сегмента h(x), файловой системе выдается команда найти новый блок, в который будет помещена эта запись. Затем новый блок добавляется в конец цепочки блоков сегмента h(x).

Для удаления записи с ключом x, требуется сначала найти эту запись, а затем установить ее бит удаления.

Удачная организация файлов с хешированным доступом требует лишь незначительного числа обращений к блокам при выполнении каждой операции с файлами. Если выбрана удачная функция хеширования, а количество сегментов приблизительно равно количеству записей в файле, деленному на количество записей, которые могут поместиться в одном блоке, тогда средний сегмент состоит из одного блока. Если не учитывать обращения к блокам, которые требуются для просмотра таблицы сегментов, типичная операция поиска данных, основанного на ключах, потребует одного обращения к блоку, а операция вставки, удаления или изменения потребуют двух обращений к блокам. Если среднее количество записей в сегменте намного превосходит количество записей, которые могут поместиться в одном блоке, можно периодически реорганизовывать таблицу сегментов, удваивая количество сегментов и деля каждый сегмент на две части.

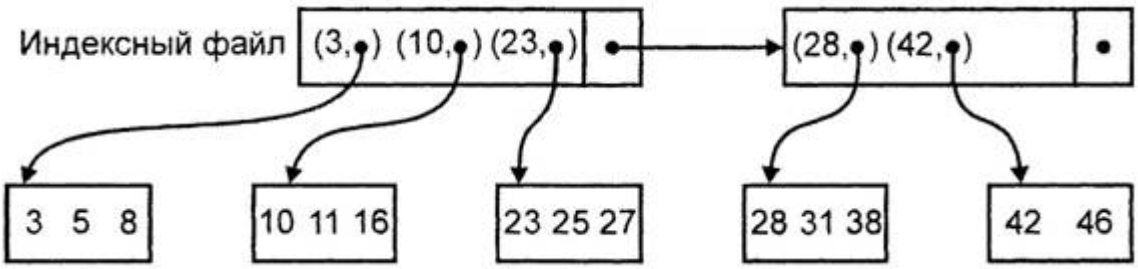
51. ОСОБЕННОСТИ АЛГОРИТМОВ ДЛЯ ВНЕШНЕЙ ПАМЯТИ. ИНДЕКСИРОВАННЫЕ ФАЙЛЫ

Природа устройств внешней памяти такова, что время, необходимое для поиска блока и чтения его в основную память, достаточно велико по сравнению со временем, которое требуется для относительно простой обработки данных, содержащихся в этом блоке.

Оценивая время работы алгоритмов, в которых используются данные, хранящиеся в виде файлов, приходится в первую очередь учитывать количество обращений к блокам, т.е. сколько раз блок считывается в основную память или записывается во вторичную память. Такая операция называется доступом к блоку. Поскольку размер блока фиксирован в операционной системе, нет возможности ускорить работу алгоритма, увеличив размер блока и сократив тем самым количество обращений к блокам. Поэтому мерой качества алгоритма, работающего с внешней памятью, является количество обращений к блокам.

Все ранее рассмотренные алгоритмы предназначены для работы с оперативной памятью. При обработке больших объемов данных их приходится хранить во внешней памяти. В результате усложняется доступ к данным. Основной единицей хранения данных является файл. Его можно рассматривать как связный список блоков. В свою очередь блок состоит из записей. И в каждый блок помещается целое количество значений. Базовыми операциями выполняемыми по отношению к файлу является перенос блока в буфер находящийся в оперативной памяти. Буфер – зарезервированная область в оперативной, соответствующая размеру блока. По окончанию обращения блок возвращается из буфера во внешний накопитель.

Еще одним распространенным способом организации файла записей является поддержание файла в отсортированном по значениям ключей порядке. В этом случае файл можно просматривать как словарь или телефонный справочник, когда просматриваются лишь заглавные слова или фамилии на каждой странице. Чтобы облегчить процедуру поиска, можно создать второй файл, называемый **разреженным индексом**, который состоит из пар *(x, b)*, где *x*– значение ключа, а *b* – физический адрес блока, в котором значение ключа первой записи равняется *x*. Разреженный индекс отсортирован по значениям ключей.



Предполагается, что три записи основного файла или три пары индексного файла помещаются в один блок. Записи основного файла представлены только значениями ключей, которые в данном случае являются целочисленными величинами. Чтобы найти запись с заданным ключом *x*, нужно сначала просмотреть индексный файл, отыскивая в нем пару *(x, b)*. В действительности ищется наибольшее *z*, такое, что $z \leq x$ и далее находится пара (*z, b)*. В этом случае ключ *x* оказывается в блоке *b*, если такой ключ вообще присутствует в основном файле.

Чтобы создать индексированный файл, записи сортируются по значениям их ключей, а затем распределяются по блокам в возрастающем порядке ключей. В каждый блок можно поместить или столько записей, сколько туда помещается, или оставить в нем вакантные места с возможностью добавления записей впоследствии. Преимущества такого подхода заключаются в том, что вероятность переполнения блока, куда вставляются новые записи, в этом случае оказывается ниже, иначе нужно будет обращаться к смежным блокам. После распределения записей по блокам создается индексный файл: просматривается по очереди каждый блок и находится первый ключ в каждом блоке. Подобно тому, как это сделано в

основном файле, в блоках, содержащих индексный файл, можно оставить какое-то место для последующего роста.

Допустим есть отсортированный файл записей, хранящихся в блоках $B_1, B_2, \ldots B_m$. Для вставки новой записи в отсортированный файл используем индексный файл, с помощью которого определяется, блок с каким номером должен содержать новую запись. Если новая запись помещается в блок B_i, она туда заносится в корректной последовательности. Если новая запись становится первой записью в блоке B_i, тогда выполняется корректировка индексного файла.

Если новая запись не помещается в блок B_i, можно применить следующую стратегию. Перейти на блок B_{i+1}, и узнать, можно ли последнюю запись B_i переместить в начало B_{i+1}. Если можно, последняя запись перемещается в B_{i+1}, а новую запись можно затем вставить на подходящее место в B_i. В этом случае нужно откорректировать вход индексного файла для B_{i+1} и, возможно, для B_i.

Если блок B_{i+1} также заполнен или если B_i является последним блоком ($i=m$), из файловой системы нужно получить новый блок. Новая запись вставляется в этот новый блок, который должен размещаться вслед за блоком B_i. Затем используется процедура вставки в индексном файле записи для нового блока.

Еще одним способом организации файла записей является сохранение произвольного порядка записей в файле и создание другого файла, с помощью которого будут отыскиваться требуемые записи. Этот файл называется **плотным индексом**. Плотный индекс состоит из пар (x, p), где p – указатель на запись с ключом *x* в основном файле. Эти пар отсортированы по значениям ключа. Структуру, подобную разреженному индексу, можно использовать для поиска ключей в плотном индексе.

При использовании такой организации плотный индекс служит для поиска в основном файле записи с заданным ключом. Если требуется вставить новую запись, ищется последний блок основного файла и туда вставляется новая запись. Если последний блок полностью заполнен, то надо получить новый блок из файловой системы. Одновременно вставляется указатель на соответствующую запись в файле плотного индекса. Для удаления записи в ней просто устанавливается бит удаления и удаляется соответствующий вход в плотном индексе.