# ТЕМА 1. ТИПЫ И СТР-Ы ДАН, ИХ СВЯЗЬ С АЛГОРИТМАМИ

## 1.1 Типы данных, абстрактные типы и Стр-ы Дан

# 1. 2 Классификация структур данных

# 1.3 Алгоритм и его Св-ва

# ТЕМА 2. ЗАПИСИ

## 2.1 Стр-ра записи

## 2.2. Запись без вариантной части

## 2.3 Записи с вариантами

2.4 Оператор присоединения With

2.5 Константа-запись (К-З)

# ТЕМА 3.ССЫЛОЧНЫЙ ТИП ДАННЫХ И ПРИНЦИПЫ РАБОТЫ С НИМ (с 75)

## 3.1 Общие сведения о ссылочном типе данных *(*указатель)

## 3.2. Методы работы с динамич переменными (3)

# ТЕМА 4. МЕТОДЫ ХЕШИРОВАНИЯ ДАННЫХ

## 4.1. Открытое хеширование

## 4.2. Закрытое хеширование

## 4.3 Методы разрешения коллизий при закрытом хешировании

## 4.4. Алгоритмы работы с хеш-таблицами м-дами открытой адресации

## ТЕМА 5. ДИНАМИЧЕСКАЯ ЦЕПОЧКА И ОДНОНАПРАВЛЕННЫЙ СПИСОК

## 5.1. Стр-ра динамической цепочки

5.2 Формирование цепочки

### 5.3. Поиск эл-нта в цепочке

5.4. Удаление эл-нта из цепочки

### 5.5. Вставка эл-нта в цепочку

### 5.6 Абстрактный тип данных «список»

# ТЕМА 6. ДВУНАПРАВЛЕННЫЙ СПИСОК

## 6.1 Стр-ра двунаправленного списка

## 6.2 Вставка эл-нта в двунаправленный список

## 6.3 Создание двунаправленного кольцевого списка с заглавным звеном

## 6.4 Удаление эл-нта

## 6.5 Поиск эл-нта в двунаправленном кольцевом списке

# ТЕМА 7. ТАБЛИЦЫ И СПОСОБЫ РАБОТЫ С НИМИ

## 7.1. Общие сведения

## 7.2 Способы организации таблиц

# ТЕМА 8. ОЧЕРЕДИ И ОПЕРАЦИИ, ВЫПОЛНЯЕМЫЕ НАД НИМИ

## 8.1 Общие сведения об очередях (стр 96 ОАИП 2 ч)

## 8.2 Операции, выполняемые над эл-нтами очередей (Очередь FIFO)

## 8.3. Разновидности очередей

# ТЕМА 9. СТЕКИ И ОПЕРАЦИИ, ВЫПОЛНЯЕМЫЕ НАД НИМИ

## 9.1 Абстрактный тип данных «стек»

## 9.2 Реализация стеков с помощью массивов

### 9.3 Реализация стеков с помощью указателей

### 9.4 Операции над эл-нтами стека

### 9.5 Применение стеков

# ТЕМА 10. ФОРМЫ ЗАПИСИ ВЫРАЖЕНИЙ

10.1 формы записи выражений

10.2 Построение выражений в обратной Польск Зап

## 10.3 Преобразование скобочных выражений в обратную Польск Зап

## 10.4 Вычисление выражений в виде обратной Польск Зап

ТЕМА 11. ОПРЕДЕЛЕНИЕ И ПОСТРОЕНИЕ БИНАРНОГО ДЕРЕВА

# 11.1 Основные определения нелинейные структур

# 11.2 Стр-ра и построение бинарного дерева

ТЕМА 12. ОПЕРАЦИИ НАД БИНАРНЫМИ ДЕРЕВЬЯМИ

# 12.1 Поиск записи в дереве

12. 2 Включение записи в дерево

# 12.3 Удаление записи из дерева

ТЕМА 13. ОБХОДЫ ДЕРЕВА. ПОМЕЧЕННЫЕ ДЕРЕВЬЯ

# 13.1 Обходы дерева

# 13.2 Построение бинарного дерева на основе произвольного дерева

# 13.3 Помеченные деревья и деревья выражения

# 13.4 Альтернативные способы реализации деревьев

ТЕМА 14. ПРОШИТЫЕ БИНАРНЫЕ ДЕРЕВЬЯ

# 14.1 Представление списков в виде бинарных деревьев

# 14.2 Прошитые бинарные деревья

# 14.3 Алгоритмы прошивки и обхода прошитых деревьев

ТЕМА 15. КОДЫ ХАФФМАНА

# 15.1 Применение деревьев. Представление сообщений кодами Хаффмана

ТЕМА 16. СБАЛАНСИРОВАННЫЕ БИНАРНЫЕ ДЕРЕВЬЯ

# 16.1 Идеально сбалансированные бинарные деревья

# 16.2 Бинарные деревья поиска

# 16.3 Сбалансированные деревья поиска

# 16.4 Операции над деревьями

# 16.5 Вставка (Операция включения) эл-нта в АВЛ-дерево

ТЕМА 17. ОРИЕНТИРОВАННЫЕ ГРАФЫ: ОПРЕДЕЛЕНИЯ И

ОПЕРАЦИИ

17.1 Основные определения ориентированных графов

# 17.2 Представление ориентированных графов

# 17.3 Операторы над ориентированными графами

# 17.4 Нахождение кратчайшего пути на ориентированном графе

# 17.5 Нахождение кратчайших путей м/у парами вершин

ТЕМА 18. ЦЕНТР ОРИЕНТРОВАННОГО ГРАФА. ОБХОДЫ ГРАФА

# 18.1 Транзитивное замыкание

# 18.2 Нахождение центра ориентированного графа

# 18.3 Обход ориентированных графов

# 18.4 Глубинный остовный лес

ТЕМА 19. МОДЕЛЬ ВНЕШНИХ ВЫЧИСЛЕНИЙ. СТРУКТУРЫ И

АЛГОРИТМЫ

## 19.1 Модель внешних вычислений

## 19.2 Особенности операций с внешней памятью

## 19.3 Организация данных в файлах

## 19.4 Хешированные файлы

## 19.5 Индексированные файлы

19.6 Несортированные файлы с плотным индексом

# ТЕМА 20. В-ДЕРЕВЬЯ И ОПЕРАЦИИ НАД НИМИ

## 20.1 Внешние деревья поиска

## 20.2 В-деревья

### 20.3 Операторы на В-дереве

### 20.4 Сравнение методов

# ТЕМА 1 «ТИПЫ И СТР-Ы ДАН, ИХ СВЯЗЬ С АЛГОРИТМАМИ»

## 1.1 Типы данных, абстрактные типы и Стр-ы Дан

**Типы данных** включают: 1) натуральные и целые числа, 2) вещественные (действительные) числа (в виде приближенных десятичных дробей), 3) литеры, 4) строки и др.

В некоторых языках программирования **(ЯП)** тип константы/переменной компилятор определяет по записи присваиваемого значения. В других языках прогер явно задает тип переменной. В зав-ти от назначения ЯП защита типов на этапе компиляции м б жесткой (*Pascal).* *Pascal*-компилятор расценивает смесь данных разных типов в 1ом выражении как ошибку. А язык *C (*слабая защита типов) такую смесь *C-*компилятор даёт предупреждения: дает д/п возмсти, но прогер сам отвечает за правильность своих действий.

**Абстрактный тип данных (АТД)** – матем-кая модель и набор операторов, определенных в рамках данной модели.

Первоначальная разработка алгоритма возможна в терминах АТД. Но далее для реализации алгоритма на конкретном ЯП нужен способ представления АТД в терминах типов данных и операторов выбранного языка. Для представления АТД использ-ют Стр-ы Дан в виде набора переменных разных типов данных, объединенных определенным образом.

Используемые в алгоритмах Стр-ы Дан м б сложными. Поэтому удачный выбор представления данных м влиять на производит-сть проги сильнее, чем детали алгоритма.

Базовый строительный блок Стр-ы Дан - **ячейка** для хранения значения определенного базового или составного типа данных.

# 1. 2 Классификация структур данных

***Физическая* стр-ра данных (Стр-а Дан)** **(стр-ра хранения,** внутренняя стр-ра, стр-ра памяти**)** – отражает способ физического представления данных в памяти машины.

**Абстрактная** (**логическая) Стр-а Дан** – стр-ра Дан без учета ее представления в машинной памяти. М/у логической и соответствующей ей физической стр-рами есть различие. Есть процедуры, выполняющие отображение логической структуры в физическую и, наоборот. + эти процедуры дают доступ к физическим стр-рам и выполнение операций над ними.

Есть **простые** (базовые, примитивные) Стр-ы Дан (типы) и **интегрированные** (структурированные, композитные, сложные).

**Простые** Стр-ы Дан – нельзя разделить на составные части больше, чем биты. Для физической стр-ры важно, чтобы в машинной архитектуре и с-ме програм-ния всегда заранее были известны размер простого типа и стр-ра его размещения в памяти. С логической точки зрения простые данные - **неделимые** единицы.

**Интегрированные** Стр-ы Дан – составные части которых - другие Стр-ы Дан: простые / интегрированные. Интегрированные Стр-ы Дан конструируются с использ-ием ср-тв интеграции данных, предоставляемых ЯП.

В зав-ти от отсутствия / наличия явно заданных связей м/у эл-нтами данных различают **несвязные** (векторы, массивы, строки, стеки, очереди) и **связные** структуры (связные списки).

Важный признак Стр-ы Дан - ее **изменчивость**: изменение числа эл-тов и/или связей м/у элем-ми структуры. В определении изменчивости структуры не отражено св-во изменения значений эл-тов данных, т к в этом случае все Стр-ы Дан имели бы св-во изменчивости.

По признаку изменчивости структуры**: статические, полустатические, динамические.** Базовые Стр-ы Дан, статические, полустатические и динамические характерны для оперативной памяти и часто называются оперативными стр-рами.

**Файловые структуры** соответствуют стр-рам данных для внешней памяти. На рис 1.1 классификация структур данных.

Структуры данных (5)

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Простые базовые | Статические | Полуста-тич-ие | Динамические | Файловые |
| Числовые; символьные; логические;  перечисление;  интервал; указатели. | Векторы; массивы;  множества; записи; таблицы. | Стеки;  Очереди;  Деки;  Строки. | Линейные связанные списки;  разветвленные связанные списки; деревья;  графы. | Последовательного, прямого;  комбинированного доступа; организован-  ные разделами;  индексированные |

Рис 1.1 – Классификация структур данных

В зав-ти от упорядоч-ти эл-тов Стр-ы Дан делят на **линейные** и **нелинейные**.

По признаку взаимного расположения эл-тов в памяти линейные структуры: 1) с последовательным (векторы, строки, массивы, стеки, очереди) и 2) с произвольным связным распределением эл-тов в памяти (1связные, 2связные списки).

В ЯП понятие *"*Стр-ы Дан*"* связано с понятием *"*типы данных*"*. Инфа по каждому типу определяет:

− структуру хранения данных (указанного типа): выделение памяти и представление данных в ней (с 1 стороны) и интерпретирование двоичного представления (с другой);

− множ-во допустимых значений, которые м иметь какой-либо объект (описываемого типа);

− множ-во допустимых операций, применимых к объекту этого типа.

**Статические структуры** – разряд **не**примитивных структур: структурир-ное множ-во примитивных базовых структур. Напр-р, вектор м представить упорядоченным множ-вом чисел. Т к по определению статические структуры отличаются отсутствием изменчивости, память для них выделяется автоматически на этапе компиляции или при выполнении в момент активизации программного блока, в котором они описаны. Выделение памяти на этапе компиляции - столь удобное св-во статических структур, что их использ-ют даже для представления объектов, обладающих изменчивостью. Напр-р, когда размер массива неизвестен заранее, для него резервируется максимально возможный размер.

Каждую Стр-ру Дан характер-ют её логическим и физическим представлениями. Говоря о какой-то структуре данных, имеют в виду её логическое представление. Физическое представление часто не соотв-ет логическому, + м б разное в разных программных с-мах: физической структуре ставится в соотв-вие дескриптор или заголовок, который содержит общие сведения о физической структуре. Дескриптор, как и сама физическая стр-ра, хранится в памяти. Дескриптор Стр-ы Дан (поддерживается Я-миП-ия) - «невидим» для прогера, создается компилятором, который, формируя объектные коды для доступа к структуре, включает в эти коды команды, обращающиеся к дескриптору.

Статические структуры в ЯП связаны со структурированными типами. Структурированные типы в ЯП - средства интеграции, которые позволяют строить Стр-ы Дан большой сложности: массивы, записи и множества.

## 1.3 Алгоритм и его Св-ва

**Алгоритмизация** – сведение задачи к последоват-ти этапов, выполняемых друг за другом так, что рез-ты предыдущих этапов использ-ют при выполнении следующих.

**Алгоритм** – с-ма правил, четко описывает последоват-ть действий, которые надо выполнить для решения задачи.

**Св-ва** правильно разработанного алгоритма:

1. **Дискретность** – значения величин в каждый следующий мом-т вр должны получаться по определенным правилам из значений величин предыдущего момент вр.
2. **Определенность (детерминированность***)* – каждое правило алгоритма д б 1значным. Значения, получаемые в какой-то момент вр, 1значно связаны со значениями вычисленных ранее величин.
3. **Результативность (конечность)** – алгоритм приводит к решению задачи за конечное число шагов.
4. **Массовость** – алгоритм разрабатывать в общем виде: его м применить для задач, отличных лишь исходными данными.

Чтобы разработать алгоритм решения задачи, ее надо представить как последоват-ть четких правил.

Понятие алгоритма и проги не четко разграничены. Прога на алгоритмическом языке – это окончательный вариант алгоритма решения задачи, ориентированный на конкретного исполнителя (комп / ЯП).

Способы описания алгоритмов:

1. запись на естественном языке (словесное описание);
2. изображение в виде схемы (графическое описание);
3. запись на алгоритмическом языке (прога на ЯП);

1) Словесное описание алгоритмов: отличаются применяемыми метаязыками. **Мета-язык** *-* яз для описания ЯП. Его «-» – малая наглядность.

2) Графическое описание представляет алгоритм в наглядной форме в виде схемы.

**Схема алгоритма** – графическое представление алгоритма, где этапы процесса обработки инфы и носители инфы представлены геометрическими символами, а последоват-ть процесса - направлением линий.

Структуры алгоритмов:

1) линейные; 2) разветвляющиеся; 3) циклические.

**1) Линейный вычислит-ый процесс** – процесс, где направление вычислений – единственное (!).

**2) Разветвляющийся вычислит-ый процесс** – процесс, где направление вычислений определяется некоторыми условиями (if then).

**3) Циклический вычислит-ый процесс** – отдельные участки вычислений выполняются многократно (while do; until repeat).

Многие алгоритмы имеют смешанный характер: линейные куски, ветвления, циклы с известным кол-вом повторений и итерационные циклы.

При составлении каждой проги надо оперировать со **стр-рами данных** и **алгоритмами***:* как удачно они выбраны, зависитуспех решаемой задачи. + надо определить уровень абстрагирования, чтобы перейти от фрагмента реального мира к наборам и типам данных, представляющим его в проге.

# ТЕМА 2. ЗАПИСИ (СТР 37 ОАИП 3.1.)

## 2.1 Стр-ра записи

**Запись** – стр-ра данных, состоит из иерархически упорядоченных разнородных компонентов. Компоненты записей (в отличие от массивов) м иметь разные типы, и доступ к ним - по именам (не по индексам).

Компоненты записей - **поля**. Тип поля м б любой, поэтому компонентом записи м б тоже запись – иерархическая запись. Уровень вложенности (иерархии) – до 9.

Пр-р простой записи – представление комплексного числа **a + b \* i** – рис 2.1.

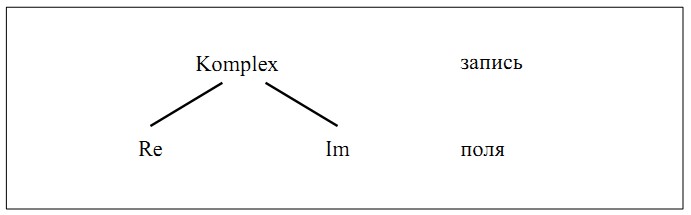


Рис 2.1 – Пр-р неиерархической записи

Пр-р иерархической записи *Anketa -* рис 2.2.

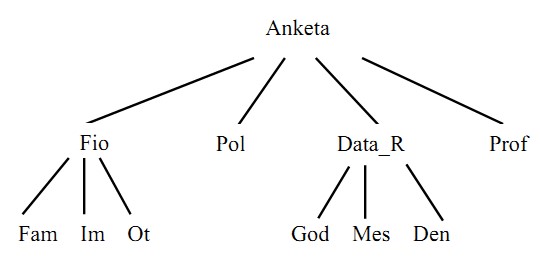


Рисунок 2.2. – Пр-р иерархической записи *Anketa*

Запись м состоять из **общей** и **вариантной** частей, каждая из них необязательна. Поле записи обозначается **идентификатором**. К полю обращаются по **имени**. Обл действия полей записи - сама запись. Имя каждого поля внутри записи д б уникальным.

## 2.2. Запись без вариантной части (СТР 38 3.2)

Такие записи имеют только **общую** часть. У каждого поля записи – свое **имя** и задается **тип** значения этого поля.

*Пр-р 2.1.* Объявление записи, ее стр-ра - на рис 2.1.

**type**

Komplex = **record**

Re: real;

Im: real; //либо Re, Im: real;

**end**;

В разделе **Var** надо ввести переменные типа Komplex:

**var**

x, y: Komplex;

Тип поля записи м б определен 2 способами: 1) прямо задан в описании записи или 2) заранее описан (- указывается имя типа).

*Пр-р 2.2.* Объявление записи: ее стр-ра - на рис 2.2.

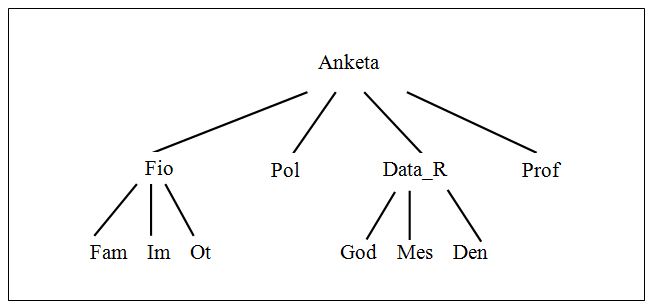


Рис3.2 – **Иерар**хическая запись

**Type**

**Data = Record**

**God: 1900..2000;**

**Mes: (Jn, Fb, Mr, Ap, Ma, Jn, Jl, Ag, Sp, Oc, Nv, Dc);**

**Den: 1..31**

**End;**

**Anketa = Record**

**Fio: Record**

**Fam: String [20];**

**Im: String [10];**

**Ot: String [20]**

**End;**

**Pol: (Man, Woman);**

**Data\_R: Data;**

**Prof: String [20]**

**End;**

2). В разделе **Var –** ввести переменную типа **Anketa:**

**Var**

**An1, An2: Anketa; //An1, An2,** – полные переменные (полные имена)

**D1, D2: Data;**

Объем памяти для записи = ∑ длин полей нижнего уровня.

**Полная переменная -** имеющая тип записи **верхнего уровня**.

Обращение к **значению поля** - с помощью идентификатора полной переменной, идентификаторов всех полей (с учетом их иерархии), в состав которых входит поле, и имени данного поля, разделенных точкой. Такое имя называется **составным именем**.

Напр-р, д/пр-ров 2.1 и 2.2 **An1, An2, X, Y** – **полные** переменные (полные имена). **Составные** имена записывают: **X.Re, X.Im, An1.Pol, An1.Fio.Fam, An1.Data\_R.God** и т.д.

Для полных переменных одного и того же комбинированного типа есть только 1 операция – *операция присваивания*. выражение в правой части оператора присваивания - только переменная того же типа – запись. Для Пр-ров 2.1, 2.2 м записать:

**X := Y; An1 := An2;**

К этому моменту записи ***Y, An2*** уже д б определены.

Составное имя м использ-ть везде, где допустимо применение типа поля. Для присваивания полям значений использ-ют оператор **:=** или ввода.

**Пример 2.3.** Присвоение значений **полям** переменной **D1** (к пр 3.2).

**D1.God := 1970;**

**D1.Mes := Jn;**

**D1.Den := 15;**

Для определения значения **полной** переменной надо :=ть значения всем полям (образующим это значение).

**Составные имена** использ-ют в операторах ввода-вывода. Напр-р:

**Read (D1.God, D1.Den); Write (A1.Fio.Fam);**

Часто удобно пользоваться массивами из записей.

Напр-р, в Пр-ре 2.2 м б объявлен массив:

**Var**

**Spisok: Array [1..100] Of Anketa; 100 -** человек

Такое объединение позволяет хранить анкетные данные на 100 человек, выделять и обрабатывать из них нужные. Напр-р, м вывести значения нужных полей записи:

**For I:=1 To 100 Do**

**Writeln (Spisok [I].Fio.Fam, Spisok [I].Fio.Im, Spisok [I].Fio.Ot)**

2.3 Записи с вариантами (СТР 40 3.3.)

Записи, содержащие только общую часть, имеют строго определенную структуру.

Записи **с вариантами** объединяют похожие описания записей, но не идентичные по форме.

Они состоят из частей: 1.необязательной **общей** + 2.**вариантной**.

1). Общая (необязательная) часть описывает поля, которые есть в каждой переменной определяемого типа.

2). Вариантная часть начинается зарезервированным словом **Case** и описывает неск-ко вариантов структуры записи.

В каждый момент выполнения программы 1 из вариантов структуры – активный в зависимости от значения признака варианта.

**Признак варианта** описывается сразу после **Case.**

**Признак** –самостоятельное поле общей части записи.

Поля признака м не быть. Тогда в ***Case*** указывают только идентификатор типа, и надо следить какой вариант записи – активный. Список полей каждого варианта заключают в скобки.

Напр-р, пусть вариантная часть записи имеет вид:

а) **Case I: Integer Of ...**

Здесь идентификатор **I** определяет (обзывает) поле признака, которое является **самосто**ятельным полем общей части записи.

б) **Case Integer Of ...**

Здесь поля признака нет. Варианты определяются целыми значениями в списке вариантов.

*Пр-р 2.4.* Запись с вариантной частью без поля признака.

**Var**

**Z: Record**

**Case Integer Of**

**1: (I1: 1..10);**

**2: (J1: Char);**

**3: (K1: Boolean)**

**End;**

Здесь в вариантной части 3 варианта. Б активен вариант, соответствующей имени поля. Напр-р, если в проге есть обращения к полю **Z.I1**, то вариантная часть воспринимается как поле типа **1..10**. При обращении к вариантной части по имени **Z.J1** она воспринимается как поле типа **Char**.

У части **Case** нет отдельного **End**. 1 слово *End* заканчивает всю конструкцию записи с вариантами.

*Пр-р 2.5*. Объявление записи с вариантами **с полем** признака.

**Type**

**Anketa1 = Record** {===== Общая часть ==============}

**Fio: Record**

**Fam: String[20];**

**Im: String[10];**

**Ot: String[20];**

**End;**

**Case Pol: (Men, Women) Of** {= Вариантная часть ===}

**Men: (Vozr1: 20..30);**

**Women: (Vozr2: 18..25)**

**End;**

В вариантной части все имена полей д б уникальны, даже если они встречаются в разных вариантах. **Запись** м иметь только 1 вариантную часть – в конце записи. При этом вариантная часть м б вложенной в другую вариантную часть.

Если вариантная часть пустая (соотв-ая какому-то значению признака), она записывается так:

**<Диапазон>: ()**

Например, если в пр 2.5. нет поля при значении признака, = Women, то этот вариант запишется так:  **Women: ()**

Объем **памяти** для записи с вариантами = ∑ объемы полей **общей** части + ∑ длин полей **вариантной** части максимальная по объему.

Поля записи размещают в памяти последовательно, в соответствии с объявлением.

## 2.4 Оператор присоединения With (с 42(3.4.)

При работе с полями в их составном имени надо писать **путь** к полю через все уровни иерархии, начиная от полного имени записи. Т к уровней иерархии **< 9**, **составное имя** поля м иметь 9 компонент. Работать с ним неудобно, а программа громоздкая.

Для сокращения составного имени поля м б оператор присоединения **With**. В нем указывают список переменных типа **Record**. Оператор **With**облегчает доступ к полям этих записей и минимизирует повторные адресные вычисления. К полям записей внутри <**Оператор**> (в ***With)*** можно обращаться как к простым переменным.

*Пр-р 2.6.* В Пр-ре 2.2 объявлена переменная **D1: Data**. Используя оператор **With**, вместо Пр-ра 2.3 м записать:

|  |  |
| --- | --- |
| **With D1 Do**  **Begin**  **God:=1970;**  **Mes:=Yan;**  **Den:=15**  **End** | Пр-р 2.3:  **D1.God := 1970;**  **D1.Mes := Jn;**  **D1.Den := 15;** |

*Пр-р 2.7.* Используя оператор **With**, к полям записи **An1** м обратиться так:

Type

Data = Record

God: 1900..2000;

Mes: (Jn, Fb, Mr, Ap, Ma, Jn, Jl, Ag, Sp, Oc, Nv, Dc);

Den: 1..31

End;

**Anketa** = Record

Fio: Record

Fam: String [20];

Im: String [10];

Ot: String [20]

End;

Pol: (Man, Woman);

Data\_R: Data;

Prof: String [20]

End;

Var

**An1,** An2: **Anketa;**

D1, D2: Data;

**With An1 Do**

***Begin***

**Fio.Fam := ’Иванов’;**

**Fio.Im := ’Петр’;**

**Fio.Ot := ’Степанович’;**

**Pol := Man;**

**Data\_R.God := 1970;**

**Data\_R.Mes := Yan;**

**Data\_R.Den := 25;**

**Prof := ’Студент’**

***End;***

Без ***With***перед всеми именами полей надо было бы писать имя записи **An1**.

Адрес переменной типа **Record** вычисляется до выполнения **With**. Любые изменения переменных, влияющие на вычисленное значение адреса, до завершения **With** отражаются на значении вычисленного ранее адреса.

**Сокращенная форма With** – список переменных типа запись в **With**:

|  |  |
| --- | --- |
| **With** Z1, Z2, Z3, …, Zn **Do <Оператор>** | =сильна **полной форме** опер-ра **With**:  **With** Z1 **Do**  **With** Z2 **Do**  **. . .**  **With** Zn **Do <Оператор>** |

Из полной формы ***With*** видно, что идентификатор поля **Zn** в **<Операторе>** обозначает компонент записи из ближайшего объемлющего оператора ***With*,** в котором указана переменная с таким полем.

*Пр-р 2.8.* В Пр-ре 2.2 для полного устранения надобности в составных именах полей м использ-ть (см. пр 2.7) **сокра**щенную форму ***With*:**

**With An1, Fio, Data\_R Do**

*Begin*

**Fam := ’Иванов’;**

**Im := ’Петр’;**

**Ot := ’Степанович’;**

**Pol := Man;**

**God := 1970;**

**Mes := Yan;**

**Den := 25;**

**Prof := ’Студент’;**

*End;*

Если у 2х переменных из списка записей ***With –*** поля с 1 и тем же **идентиф**икатором, то внутри ***With*** он обозначает поле **последней** указаной переменной.

**Имена** отдельных полей записи в программе могут совпадать с **именами** переменных. Имя переменной в ***With* недоступ**но.

**Пример 2.9.** Использ-ие **совпадающих** идентификаторов.

**Var V: Record {============ Запись =======}**

**V2: Integer;**

**V1: Record**

**A: Real**

**End;**

**A: Integer**

**End;**

**A: Char;** **{==== Переменная =====}**

**. . .**

**With V, V1 Do** (по цепочке спускаемся вниз)

***Begin***

**V2 := 1;** {значение поля **V.V2 = 1**}

**A := 1.0;**  {значение поля **V.V1.A = 1.0**}

**V.A := 1;**  {значение поля **V.A = 1**}

***End;***

**A := ’A’;**

Здесь 3 элемента с именем ***А*** – **поля** ***V.V1.A*, *V.A*** и переменная ***А*** типа **Char**. Указание имени поля ***А*** в **<Операторе>** оператора ***With*** приведет к обращению к полю ***V.V1.A*,**  т к поле ***А*** есть в записи ***V1***, находящейся в списке записей сокращенной формы оператора ***With*** после записи ***V***. Чтобы обратиться к полю **V.A,** надо использ-ть его составное имя или вынести оператор (работает с этим полем) за пределы ***With*,** где поля записи становятся недоступными, а переменная ***А*** типа ***Char –*** доступной.

2.5 Константа-запись (К-З) (с 44(3.5)

- один из видов структурных типи**зов**анных констант.

Как и остальные типизованные константы, К-З м использ-ть как **инициированные** переменные типа запись – переменные, которым при запуске программы присваивается **начальное** значение.

**Пример 3.10. Объявление** константы-записи ***D***.

**Type**

**Fam = (Ivanov, Petrov);**

**Data = *Record***

**God: 1900..2000;**

**Mes: (Jn, Fb, Mr, Ap, Ma, Jn, Jl, Ag, Sp, Oc, Nv, Dc);**

**Den: 1..31**

***End;***

**Ank = Array[Fam] Of Data;**

**Const**

**D: Data = (God: 1950; Mes: Jn; Den: 3);**

**A: Ank = ((God:1970; Mes:Dc; Den:7), (God:1945; Mes:Ma; Den:15));**

***D*** – **типизованная К-З**, **А** – массив из 2х типизованных К-Зей **А[Ivanov] + А[Petrov].** В константе ***А*** внутренние скобки относятся к К-Зсям, внешние – к типизованным **константам-массивам**.

В типизованных К-Зсях поля надо указывать в том же порядке, как они идут в объявлении типа запись.

Если запись содержит вариант (**с Case**), то м указать только поля выбранного варианта. Если в варианте – поле признака, то его знач-ие д б определено.

Использ-ть компонент **файлового** типа в структурных К-З запрещено!

# ТЕМА 3 «ССЫЛОЧНЫЙ ТИП ДАННЫХ И ПРИНЦИПЫ РАБОТЫ С НИМ» (с 71(6.1)

## 3.1 Общие сведения о ссылочном типе данных *(*указатель)

**Динамические переменные** – которые порождаются и уничтожаются в процессе выполнения проги, их размер значений определяется и м изменяться при выполнении проги.

Для работы с динамическими переменными – *ссылочный тип (***указатель***).* Его значение - ссылка на какой-либо эл-нт проги. По данной ссылке идет прямой доступ к этому эл-нту. В качестве ссылки использ-ется **адрес** соотв-его эл-нта в памяти машины.

2 **вида указателей** ***–*** *1.*указатель на **элемент некоторого типа** (верхнее ребро синтаксической диаграммы) и 2.указатель, **не связанный с типом *(*Pointer*).***

**Пр-р 3.1.**Использ-ие указателей при объявлении программных эл-тов.

**Type**

***Mas = Array [1*..10] Of Integer;**

**Admas = ^Mas;**  {**Admas** – тип указателя на массив типа Mas}

**P1 = ^Integer; {P1 –** тип указателя на тип **Integer}**

**Var**

**P: ^Integer;** {в Ука P б храниться адрес динамич-ой перем целог типа}

**Q: ^Char;** {в указат Q б храниться адрес динамич перем типа Char}

**Pp: P1;** {в указателе Pр – **адрес** динамич перем-ой целого типа}

**Pt: Pointer;** {Ука Pt **м** хранить адрес динам перемен произвольн типа}

**Adrm: Admas;** {в указат Adrm – адрес **динам массива** типа Mas}

Указатели - автоматические переменные ссылочного типа. Динамические переменные не объявляются в проге. Поэтому (!) средство доступа к ним - указатели. Занимают 4 байта памяти.

Иногда надо в качестве значения указателя принять **пустую ссылку** – ссылку, которая **не** связывает с данным указателем никакой динамической переменной. Для этого служит **предопределенная** константа **Nil-** совместимая по типу с любым типом указателя. Например, для указателей в примере 2.1, можно записать:

**Pt := Nil;**

**P := Nil;**

**Q := Nil;**

**Adrm := Nil;**

Над значениями ссылочного типа определены только присваивание и 2 операции сравнения: = и <>.

2 значения ссылочного типа =ы, **Nil /** указывают на 1у и ту же переменную. В остальных случаях - неравенство.

В правой и левой частях от оператора **:=** д б выражения 1го типа: справа - значение ссылочной **перем**енной(ЗСП), слева - тип переменной.

**Ссылочным выражением** м б:

1. **пустая** ссылка **Nil*;***
2. ссылочная **перем**енная;
3. ссылочная **ф-ция** – её значение – ссылка.

Например, встроенная ф-ция **Addr(X)** – возвращает адрес эл-та ***X***. ***X*** – любая переменная, процедура, ф-ция. Результат ф-ции **Addr(X)** совместим (по присваиванию) со всеми типами “указатель”. Поэтому можно написать такие операторы присваивания (для переменных типа указатель в прим. 2.1):

**Pt := Addr (А);** {А может иметь любой тип}

**P := Addr (В);** {В – только тип **Integer**}

**Pt: Pointer;**

**P: ^Integer;** {в Ука P б храниться адрес динамич-ой перем целог типа}

Объявление ссылочной переменной в разделе **Var** **не** порождает самой динамич переменной: по этому объявлению транслятор отводит место в памяти для размещения ссылки (**адреса** динамич переменной).

3.2. Методы работы с динамич переменными (3) (стр 73(6.2.)

с помощью **процедур:**

1. **New***и***Dispose***;*
2. **Getmem** *и***Freemem;**
3. **Mark***и***Release**.

6.2.1. Процедуры New и Dispose

**стандарт**ная процедура **New (P)** – для порождения динамической переменной. ***P*** – ссылочная переменная любого типа.

Данная процедура размещает динамическую переменную типа указателя: выделяют память для хранения наиболее длинного значения типа указателя(ЗТУка). **Адрес начала** этого места памяти присваивают заданной ссылочной переменной ***P***.

**Пример 3.2.** Использ-ие процедуры **New.**

**Var**

**P*: ^Integer;*** {Р – **указатель** на динамич перем-ую типа Integer}

**Begin**

**New (P); {-** порождает **динамич переменную** типа ***Integer***,

адрес этой переменной присваивается **указателю *P}***

**...**

обращение к **динамич** перемен (связанной с указателем ***Р)*** идет с пом. **переменной с указателем**:

**P^ := 15;** {**Р^ - переменная с указателем**; динам переменой типа Integer (связан с указател P) присвоено значение 15}

Процедура *New(P)* порождает динамическую переменную типа *Integer*, адрес которой присваивается указателю *Р*. Для динамической переменной процедура ***New*** – то же, что раздел описаний ***Var*** для автоматической переменной. Никакого значения процедура **New** динамической переменной **не** присваивает. Над **переменными с указателями** определены **любые** **операц**ии, что и д/типа **динамической переменной**.

*Пр-р 3.3.* есть объявы - прим 3.1, 2. Операции над динамич переменными.

**Type**

Mas = **Array** [1..10] **Of** Integer;

Admas = ^Mas; {**– тип** указателя на массив **типа** Mas}

P1 = ^Integer; {– **тип Ука** на **тип** Integer, м указ-ть только на Инты}

**Var**

P: ^Integer; {в P б хран адрес(дом, ул) динам перем целого типа}

Q: ^Char; {в указат Q б храниться адрес динамич перем типа Char}

Pp: P1; {в указателе Pр – адрес динамич перем-ой целого типа}

Pt: Pointer; {Ука Pt м хранить адрес динам пер произв типа}

Adrm: Admas; {в указат Adrm – адрес динам массива типа Mas}

**X: Integer;**

**Begin**

**New (P);**

**P^ := 15;**

**...**

**X := X + P^;**

**P^ := P^ Mod 5;**

**Adrm^[P^ + 2] := 14;**

**Пример 6.4.** Организация массивов указателей. есть объявы пр 6.1:

**Type**

Mas = **Array** [1..10] **Of** Integer; Admas = ^Mas;

P1 = ^Integer;

**Var ...**

**А: Array[1..20] Of P1;** {–мас-армия ссылок на **динамич** перем типа Inte}

**I:** Integer;

**Begin**

***...***

**For I:= 1 To 20 Do**

**Begin**

**New (A[I]);** {Порождение динамич перем, связ-ой с Ука A[I]}

**A[I]^ := I;** {Обращение к динамической переменной}

**End**

***...***

Здесь каждая из **ссылок** **A[I]** обеспечивает **доступ** к соответствующей переменной с указателем **A[I]^.**

Идентификатор типа в определении ссылочного типа м объявить и **до его использования, и после** (в отличие от других типов идентификаторов).

Так в Пр-ре 3.1 идентификатор ***Mas*** был объявлен до его использования в типе указателя ***Admas***. Возм и обратное объявление:

**Type**

**Admas = ^Mas;**

**Mas = Array [1..10] Of Integer;**

*Пр-р 3.5.* Работа с указателями и с динамическими переменными

***Var***

**X, Y: ^Integer;** {X, Y – указатели на динамич перем-ые типа Integer}

**Begin**

**New (X);** {Порождение динам перем-ой типа Integer, её адрес - в Х}

**New (Y);**{ Порождение динамич перем типа Integer, её адрес - в Y}

**X^ := 5;** { В динам перем, связ-ую с указат X, всунули знач 5}

**Y^ := 10;** {В динам перем, связ-ую с указат Y, занесено знач 10}

**X := Y;** {Указателю Х присвоено значение указателя Y}

**Y := Nil;** {Указателю Y := значение **пустой ссылки**}

**Dispose (P)**;

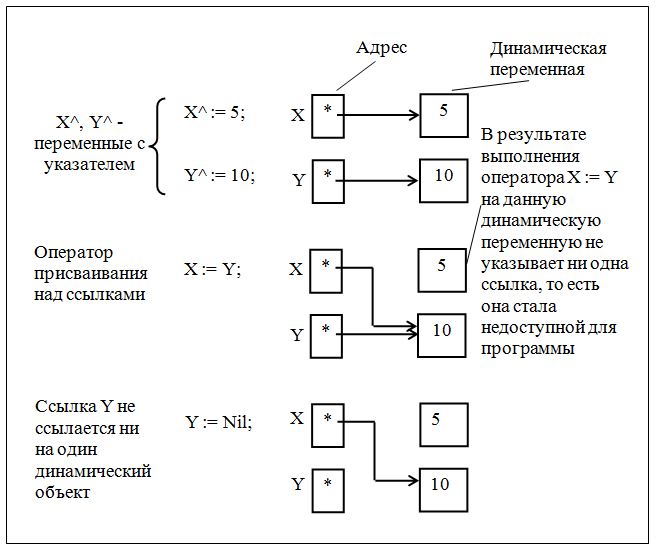
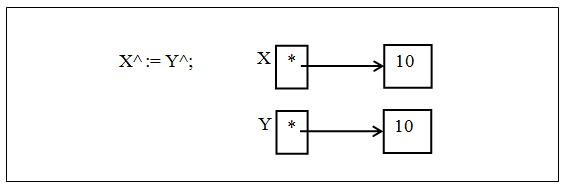
Схема действия операторов проги

Рис 6.4 – Действия операторов программы **по примеру 3.5**

Если вместо оператора присваивания **X := Y** (рис 3.4) б оператор **X^:=Y^,** то его результат б выглядеть так:



Переменная типа указатель м б параметром ф-ции **Ord.**

**(ф-ция** **ord** в **Pascal** возвращает число, соответствующее символу**)**. Результат этой ф-ции - целое значение, = адресу динамич переменной, на которую ссылается данный указатель. Обратное преобразование (Integer 🡪 указатель не допускается). В примере 6.5 после выполнения оператора

**X := Y**

динамическая переменная со значением **5** стала недоступна программе, но продолжает занимать отведенное ей место в памяти и его нельзя использ-ть для хранения других динамич переменных.

Для **уничтожения** динамич переменных (**освобождения** памяти) – стандартная процедура **Dispose (P)**: динамич переменная удаляется, (связанная с указателем ***P)*** и занимаемая ею память свободна, а значение ***P*** **неопределено*****=* Nil***.*

Проц-ра **Dispose**уничтожает только динам перемен, но не **указатель** на нее. Пр-р 3.6. Освобождение памяти, занятой динамической переменной *X^*.

…..

New (X); New (Y);

**X**^ := 5; Y^ := 10;

**Dispose *(X);*** {Освобож-ие обл памяти, занятая динамич пер-ой X^}

X := Y;

Y := Nil;

...

*Пр-р 3.7.* В задан тексте (предл?) найти слово **максим**-ой длины. Раздел-ся 1м пробелом. Признак конца текста – точка. Макс длина слова – 50 симв.

**Type**

Mas = **array**[1..50] **of** char;

Ykaz = ^Mas; {Тип указателя на массив}

**var**

Vspom, Rez, Tek: Ykaz; {Переменная типа указателя на масив}

Max, i: integer;

Byk: char;

**BEGIN**

Max := 0;

i:= 0;

**New**(Tek); {Порождение динам перемен д/хранен

текущ слова текста}

{1}**New**(Rez); {Порождение динам перемен д/хранен максим-го

из уже прочитанных слов}

**repeat**

read(Byk);

**if** (Byk<>' ') **and** (Byk <> '.')

**then**

**begin**

i:= i+1; {счетчик длины текущ слова}

Tek^[i]:= Byk

**end**

**else**

**if** i > Max **then**

**begin**

Max:= i;

{2} Vspom:= Tek;

{3} Tek:= Vspom;

{4} i:= 0;

**end**

**until** Byk = '.';

Dispose(Vspom);

Dispose(Tek);

writeln;

**for** i:= 1 **to** Max **do**

write(Rez^[i]);

dispose(Rez)

**END**.

Здесь выполнение оператора {**3}** не требует пересылки компонент массива типа **Mas** из динам-й перем-й **Tek^** в динам-ую перем-ую **Rez^*.*** В указатель **Rez** идет адрес динам-го массива **Tek^.** Т о, указатель **Rez** “перебрасывается” на другой массив. В указатель **Tek** с помощью операторов {2}, {4} заносится **адрес предыдущего** динам-го массива максимальной длины, массивы как бы меняются местами, но физически этого нет. На следующем цикле чтения в массив *Tek^* б заноситься очередное текущее слово. Вр выполнения этой программы сильно уменьшается по сравнению с использующей автоматические переменные: нет физической пересылки массивов. Пояснения к этой проге схематически:

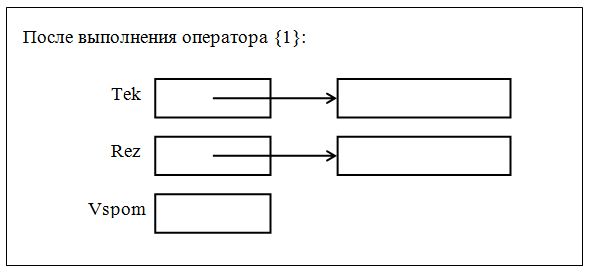


Рис 6.6 – Результат выполнения оператора **New (Rez)**

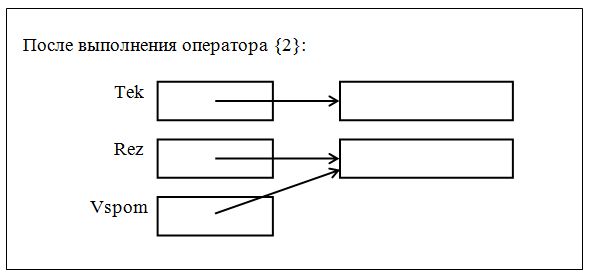


Рис 6.7 – Результат выполнения оператора **Vspom := Rez**

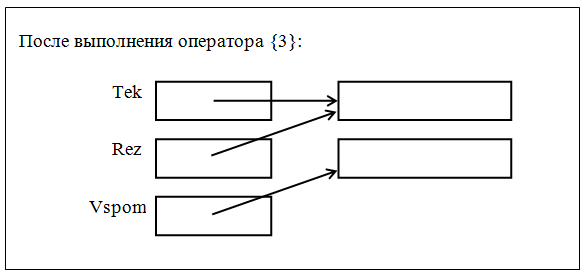


Рис 6.8 – Результат выполнения оператора **Rez := Tek**



Рис 6.9 – Результат выполнения оператора **Tek := Vspom**

3.2.2. Процедуры Getmem и Freemem(-ory)

– **2й** метод работы с **динамич переменными**. Их действие аналогичны действиям процедур *New/Dispose.* Вызов процедуры **Getmem**:

**Getmem (P, Size);** (Get – полчить)

***P*** – **указатель^** любого типа, **Size***–* выражение типа**Word*.***

Процедура **Getmem** **выделяет** из **Heap-области** блок памяти заданного размера **Size** (порождает динамическую переменную размера **Size)** и адрес этого блока присваивает ссылочной переменной **P**. Максимальный размер блока – 65521байт (**64К байта** – **$F** (**смещение**)).

**Heap*-*область –** 1. **обл динамической памяти**, 2.обл **кучи**, 3. хип-обл.

Процедура **Freemem(P, Size)** – д/**освобождения** блока памяти, занимаемого динамической переменной. **Freemem(P, Size);**

**Freemem**у**ничтожает** динам перемен **P^,** занимаемая ею память свободна, значение ***P*** – **неопред**елено.

***P*** – указатель произвольного типа, которому заранее было присвоено **значение** процедурой **Getmem** или оператором присваивания;

значение **Size** (должно точно соответствовать размеру переменной, размещенной до этого) д б таким же как было выделено проц-ой **Getmem.**

*Пр-р 3.8.* Эквивалентные фрагменты программ, использующие процедуры *Getmem/Freemem* и *New/Dispose*.



**Отличие:** процед-ры **Getmem/Freemem** дают большую свободу действий, чем **New/Dispose**(б/привязки к конкретному типу переменных. А пишут не **Size**, а **Sizeof** в параметрах пр-ра 3.8.

*Пр-р 3.9.* Использ-ие процедур *Getmem/Freemem*. Различная интерпретация выделенной области памяти.

**Type**

Zap = **Record**

X, Y: Integer;

**End**;

St = String [20];

**Var**

P: Pointer;

**Begin**

Getmem (P, 100); {Выделен блок памяти = 100 байт, адрес эт блока

присвоен ссылочной перем-ой Р}

Zap(P^).X := 124; {Интерпретац выд-ной обл как записи: блоку памяти

(j связан с указателем Р) присвоили тип Zap}

Zap(P^).Y := 47; {Интерпретац выдел-ой обл как записи}

Real(P^) := 0.213; {Интерпр-ция выдел-ой области как Real}

St(P^) := 'ПРОГА'; {Интерпр-ия выделенной области как строки}

Freemem (P, 100); {Освободили блока памяти в 100 байт}

……………………….

6.2.3. Процедуры Mark (пометить) и Release (освободить)

- повышают эффективность работы с динамической памятью. Обращение к процедуре **Mark** имеет вид: **Mark (P1);**

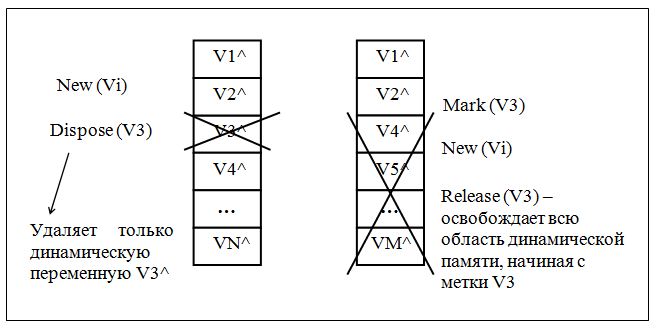
**P1** – переменная типа **указатель**, ссылается на переменную любого типа.

Процедура *Mark* присваивает своему параметру *P* адрес начала свободной области динамической памяти.

Помеченная Обл с помощью процедур ***New/*** ***Getmem***– для размещения отдельных динамических переменных.

Когда динамические переменные б ненужны, занимаемую ими память освободит процедура **Release** с параметром Р1: **Release (P1); -**  освобождает всю память, начиная с адреса, полученного при выполнении последней процедуры **Mark*.*** После **Release** значение указателя: **P1=****Nil*.***

Схематически представление работы с динамической памятью с и без использования процедур **Mark/Release:**



Параметр проц-ы **Mark** нельзя использ-ть как параметр процедур **New/Getmem** + нельзя изменять (до использования в соответствующей процедуре ***Release*)**.

Возможна **вложенность** процедур **Mark.** Тогда каждой процедуре ***Mark*** соответствует своя ***Release*.**

В 1 проге **не** использ-ют одновременно процедуры **Dispose**/**Freemem** и **Mark/Release** – рез-ты непредсказуемы (механизмы управлен свободной памятью для *Dispose/Freemem* и *Mark/Release* ***–*** отличаются).

Вместо любой **автом**атической переменной м использ-ть **динамич**ескую переменную.

Основной «+» использования динам переменных– **экономия памяти**.

«-»: 1) длинее текст программы из-за использования **New/Dispose, Mark/Release, Getmem/Freemem;** 2) снижается наглядность проги из-за использования переменной с указателем; 3) снижается быстродействие проги из-за необход-ти размещать порожденную динамическую переменную в памяти машины, за счет формирования значения соотв-его указателя и усложнения доступа к значению динамической переменной.

# ТЕМА 4. МЕТОДЫ ХЕШИРОВАНИЯ ДАННЫХ

Способы повышения эффективности работы с большими объемами инфы: для ускорения доступа к данным в таблицах.

- предварительное упорядочивание таблицы в соответствии со значениями ключей (м использ-ть методы поиска в упорядоченных стр-рах данных – сокращает вр поиска данных по значению ключа). Но при добавлении новой записи надо переупорядочить таблицу – эти потери вр на повторное упорядочивание больше выигрыша от сокращения вр поиска. Такого недостатка лишены способы (формы) хеширования данных.

## 4.1. Открытое хеширование

На рис 4.1 - базовая стр-ра данных при открытом хешировании.

Идея м-да: множ-во данных (очень большое) разбивается на конечное число **классов В**. Для ***В*** классов (нумерация от *0* до *В-1*) строится такая **хеш-ф-ция *h***, что для любого **эл-нта *x***исходного множества ф-ция ***h(x)*** принимает целочисленное значение из интервала ***0*, …, *В-1*,** которое соотв-ет классу, которому принадлежит эл-нт ***x*.** Эл-нт ***x*** называют **ключом**, ф-ция ***h(x)*** – **хеш-значением** *х*, а классы – **сегментами**. Массив (таблица сегментов), проиндексированный №№ сегментов ***0, 1, … В-1,***содержит заголовки для **В**списков. Эл-нт ***х*** *i*-го списка – эл-нт исходного множества, д/которого **h(x) = i**



Рис 4.1 – Организация данных при **открытом хешировании**

Если сегменты приблизительно = по размеру, то списки всех сегментов д б наиболее короткими (при данном числе сегментов). Если исходное множ-во состоит из ***N*** эл-тов, то средняя длина списков **=*N/B*** эл-тов. Если оценить величину ***N*** и выбрать ***B*** как м ближе к этой величине ***N***, то в каждом списке б **1-2** эл-нта. Тогда вр выполнения операций с данными б малой постоянной величиной, зависящей от *N /* *В*. Но не всегда ясно, как выбрать хеш-ф-цию *h* так, чтобы она пр-рно поровну распределяла эл-нты исходного множества по всем сегментам.

**Идеальная хеш-ф-ция** – которая для любых 2х неодинаковых ключей выдает неодинаковые адреса, т.е.

 (1)

Но подобрать такую ф-цию м, если все возможные значения ключей известны заранее. Такая организация данных называется **«совершенное хеширование**». Если заранее не определено множ-во значений ключей, и длина таблицы ограничена – трудно подобрать совершенную ф-цию. Поэтому использ-ют хеш-ф-ции, которые **не** гарантируют выполнение условия (1).

Рассмотрим хеш-ф-цию, определенную на символьных строках, которая не является совершенной, но значения **h(x)** б «хорошими».

*Пр-р 4.1.***Хеш-ф-ция** – определена на символьных строках:

**program** hesh1;

**var**

i, B: integer;

S: string[10]; // - ключ

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

**var**

i, sum: integer;

**begin**

sum := 0;

**for** i := 1 **to** j **do**

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

h := sum **mod** B;

**end**; //- конец ф-ции

**BEGIN** // ------ начало основн проги -------------------------

writeln('Введите В'); readln(B);

writeln('Введите ключ '); readln(S);

i := length(s); // - число?

writeln(**h(s, i)**);

**END**.

**Или**

**VAR**

str: string;

b: integer;

**function** h(x: string[11]):integer;

**var**

i, j, sum: integer;

**begin**

sum:= 0;

j:= length(x);

**for** i:= 1 **to** j **do**

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

h:= sum **mod** b;

**end**;

**BEGIN**

writeln('введите b'); readln(b);

writeln('введите строку'); readln(str);

writeln('хеш(?) =', h(str));

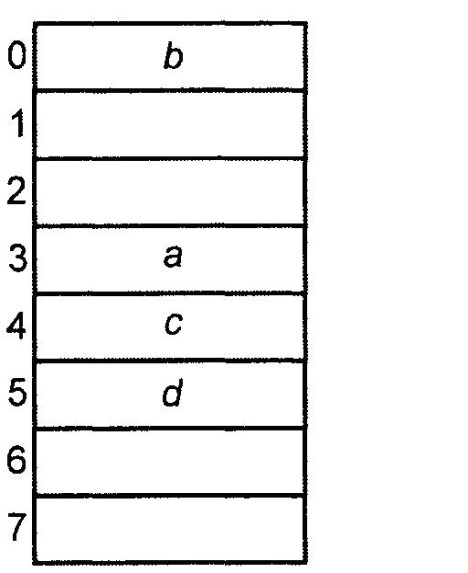
**END**.

**Идея** этой ф-ции: представить символы в виде целых чисел, используя машинные коды символов. В ***Pascal*** есть встроенная ф-ция **ord(c***)* – возвращает целочисленный **код** символа *с*. Т о, если ***х*** – ключ, тип данных ключей определен как строка символов, тогда м использ-ть хеш-ф-цию, код которой приведен ниже: в ней суммируют все целочисленные коды символов, этот результат делят на ***B*** и берут остаток от деления, который б целым числом из интервала **0 … *B-1***.

## 4.2. Закрытое хеширование

При закрытом хешировании:1) в таблице сегментов хранят прямо эл-нты словаря, а не заголовки списков. Поэтому в каждом сегменте м хранить только 1 эл-нт словаря. 2) применяется **методика повторного хеширования**: если произойдет попытка поместить эл-нт***х*** в сегмент с № ***h(x)***, который уже занят другим эл-нтом (такая ситуация называется **коллизией**), то (по методике **повторного хеширования)** выбирается последоват-ть других номеров сегментов **h1(x), h2(x),...,** куда м поместить эл-нт ***х***. Каждое из этих местоположений последовательно проверяется, пока не б найдено свободное. Если свободных сегментов нет, то таблица заполнена (эл-нт *х* вставить в нее нельзя).

Пр-р заполнения хеш-таблицы при закрытом хешировании (рис 4.2).

Рис 4.2 – **Частично** заполненная хеш-таблица

Здесь ***В = 8*** (0-7) 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 эл-нта **d** применим методику линейного хеширования, когда ***hi*(*x*) = (*h*(*x*) + *i*) mod *B***.

Тогда, если сегмент 3 уже занят (***h(a)=3***), то проверяют на занятость сегменты **4, 5, 6, 7, 0, 1, 2** (именно в таком порядке). 5-ый сегмент оказался 1ым пустым, поэтому эл-нт ***d*** был вставлен в него.

При поиске эл-нта ***х*** надо просмотреть все адреса ***h*1(*x*), *h*2(*x*), ...,** пока не б найден **х**, или пока не б пустой сегмент. Покажем это: введем условие, что в словаре нельзя удалять эл-ты. Пусть ***h*3(*x*)–** 1й пустой сегмент. Значит, нельзя найти эл-нт **х** в сегментах ***h*4(*x*), *h*5(*x*)**, … т.к. **х**вставляется в 1й свободный сегмент, значит, он находится до сегмента ***h*3(*x*)**.

## 4.3 Методы разрешения коллизий при закрытом хешировании

Для борьбы с коллизиями – спец методы, которые сводятся к м-дам "**цепочек"** и **"открытой адресации"**. Ключи, выдающие **одинаковые адреса** в таблице – называются **ключи-синонимы**.

В методе **цепочек** для разрешения коллизий во все записи вводят **указатели** для организации **списков** – "**цепочек переполнения**". При коллизии (при заполнении таблицы) в список для требуемого адреса хеш-таблицы добавляют еще 1 эл-нт.

Поиск в хеш-таблице с цепочками переполнения идет так: 1) вычисляется адрес по значению ключа. 2) последовательный поиск в списке, связанном с вычисленным адресом. **Процедура удаления** из таблицы – поиск эл-нта и его удаление из цепочки переполнения. На рис 4.3 - схема разрешения коллизий **м-дом цепочек** при добавлении эл-нта.

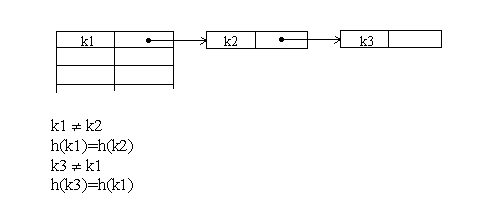


Рис 4.3 – Разрешение коллизий при добавлении эл-тов

**Метод открытой адресации**: пользуясь каким-то алгоритмом, перебирающий эл-нты таблицы, просматривать их в поисках свободного места для новой записи.

Линейное опробование – последовательный перебор эл-тов таблицы с некоторым фиксированным шагом **a = h(key) + c\*i***,*

где **i** – № попытки разрешить коллизию. При шаге, =1 идет последоват-ый перебор всех эл-тов после текущего.

Отличие **квадратичного опробования** от **линейного**: шаг перебора эл-тов нелинейно зависит от № попытки найти свободный эл-нт

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

Благодаря нелинейности такой адресации уменьшается число проб при большом числе ключей-синонимов. Но даже небольшое число проб м быстро привести к выходу за адресное пространство небольшой таблицы из-за квадратичной зав-ти адреса от № попытки. На рис 4.4 - разрешение коллизий м-дом открытой адресации.

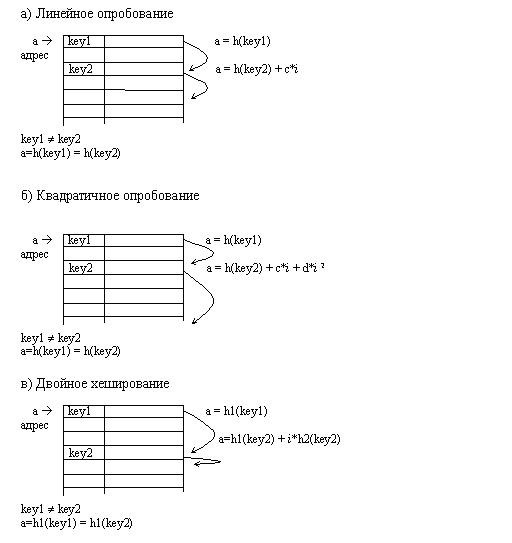


Рис. 4.4 – Разрешение коллизий при «+» эл-тов м-дами открытой адресации

**4.4. Алгоритмы работы с хеш-таблицами м-дами открытой адресации**

Алгоритмы **вставки** и **поиска** эл-тов **для м-да линейного опробования**. Здесь **t(a)** – строка (эл-нт) в хеш-таблице. Вставка

**i = 0. a = h(key) + i\*c**

Если **t(a)** = свободно, то **t(a) = key**, записать эл-нт, **стоп эл-нт добавлен** **i = i + 1**, перейти к шагу 2

Поиск

**i = 0. a = h(key) + i\*c**

Если **t(a) = key**, то **стоп эл-нт найден**

Если **t(a)** = свободно, то **стоп эл-нт не найден i = i + 1**, перейти к шагу 2

Аналогично м сформулировать алгоритмы добавления и поиска эл-тов для любой схемы открытой адресации. Отличия б только в выражении для вычисления адреса.

Процедура **удаления** здесь не б обратной процедуре **вставки**. Эл-нты таблицы находятся в 2 состояниях: **свободно** и **занято**. Если удалить эл-нт, переведя его в состояние свободно, то после такого удаления алгоритм поиска б работать некорректно. Пусть ключ удаляемого эл-нта имеет в таблице ключи-синонимы. Тогда, если за удаляемым эл-нтом в результате разрешения коллизий были размещены эл-нты с другими ключами, то поиск этих эл-тов после удаления всегда б давать отрицательный результат, т. к. алгоритм поиска останавливается на 1ом эл-нте в состоянии **свободно**.

Скорректировать эту ситуацию м по-разному. Самый простой: выполнять поиск эл-нта не до I свободного места, а до конца таблицы. Но такая модификация алгоритма аннулирует весь выигрыш в ускорении доступа к данным, который достигается в результате хеширования.

II способ: проследить адреса всех ключей-синонимов для ключа удаляемого эл-нта и при надобности переразместить соотв-ие записи в таблице. Скорость поиска после такой операции не уменьшится, но затраты вр на само переразмещение эл-тов м оказаться очень значительными.

Есть подход, который свободен от этих недостатков. Его суть: для эл-тов хеш-таблицы добавляется состояние **удалено***,* которое в процессе поиска интерпретируется как ***занято***, а в процессе записи как ***свободно***.

Рассмотрим алгоритмы вставки, поиска и удаления хеш-таблицы, имеющей 3 состояния эл-тов.

Вставка

**i = 0 a = h(key) + i\*c**

Если **t(a)** = свободно или **t(a)** = удалено, то **t(a) = key**, записать эл-нт, ***стоп эл-нт 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 эл-нт удален.* Если **t(a)** = свободно, то ***стоп эл-нт не найден* i = i + 1**, перейти к шагу 2

Поиск

**i = 0 a = h(key) + i\*c**

Если **t(a) = key**, то ***стоп эл-нт найден*** Если **t(a) = свободно**, то ***стоп эл-нт не найден* i = i + 1**, перейти к шагу 2

Алгоритм поиска для хеш-таблицы с 3я состояниями почти не отличим от алгоритма поиска без учета удалений. **Разница**: при организации самой таблицы надо отмечать свободные и удаленные эл-нты. Это м сделать, зарезервировав 2 значения ключевого поля. II вариант: введение д/п-ого поля, в котором фиксируется состояние эл-нта. Длина такого поля м составлять 2 бита - достаточно для фиксации 1го из 3 состояний. На *Delphi* это поле удобно описать типом *Byte* или *Char*.

Очевидно, что по мере заполнения хеш-таблицы б коллизии, и в результате их разрешения м-дами открытой адресации очередной адрес м выйти за пределы адресного пространства таблицы. Чтобы это было реже, м увеличить длину таблицы по сравнению с диапазоном адресов, выдаваемым хеш-ф-цией. С 1 стороны - это приведет к сокращению числа коллизий и ускорению работы с хеш-таблицей, а с другой – к нерацион-му расходу адресного пространства. Даже при увеличении длины таблицы в 2 раза по сравнению с обл-ю значений хеш-ф-ции, нет гарантий, что в результате коллизий адрес не превысит длину таблицы. При этом в начальной части таблицы м оставаться достаточно свободных эл-тов. Поэтому на практике использ-ют **циклический** переход к началу таблицы. Рассмотрим его на пр-ре м-да линейного опробования. При вычислении адреса очередного эл-нта м за адрес 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 эл-нт, *стоп эл-нт добавлен* i = i + 1, перейти к шагу 2

Здесь не учитывается возм-сть многократно превышать адресное пространство. Более корректным б алгоритм, использующий сдвиг адреса на 1 эл-нт при каждом повторном превышении адресного пространства. Это увеличивает вероятность поиска свободных эл-тов при повторных циклических переходах 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 эл-нт, *стоп эл-нт добавлен* i = i + 1, перейти к шагу 2

При большой заполненности таблицы, при неудачном выборе хеш-ф-ции - частые коллизии и циклические переходы в ее начало. Худший случай – при 100% заполнении таблицы алгоритмы циклического поиска свободного места приведут к зацикливанию. Поэтому: в хеш-таблицах избегать очень плотного заполнения. Длина таблицы = двукратное превышение максимального числа записей (предполагаемого). Но при большой заполненности таблицы м б нужно **рехеширование**. Тогда увеличивают длину таблицы, изменяют хеш-ф-цию и переупорядочивают данные.

Оценку плотности заполнения таблицы выполнять косвенно: по числу коллизий во вр 1 вставки. Достаточно определить некоторый порог числа коллизий, при превышении которого надо рехеширование. + такая проверка гарантирует невозм-сть зацикливания алгоритма при повторном просмотре эл-тов таблицы. Рассмотрим это на алгоритме вставки:

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 эл-нт, *стоп эл-нт 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 эл-нт удален* Если t(a) = свободно или i > m, то *стоп эл-нт 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 эл-нт найден*

Если t(a) = свободно или i > m, то *стоп эл-нт не найден* i = i + 1, перейти к шагу 2

Здесь № итерации сравнивается с пороговым числом *m*. Алгоритмы вставки, поиска и удаления должны использ-ть одинаковое образование адреса очередной записи.

================================

# ТЕМА 5. ДИНАМИЧЕСКАЯ ЦЕПОЧКА И 1НАПРАВЛЕННЫЙ СПИСОКhttp://static.lmb.mplabs.ru/public/c9/8c/eda1e339-8a01-164d-cb33-d6c98cc8a698.jpeg

## 5.1. Стр-ра динамической цепочки (с 81(7.1.)

Динамические цепочки - аналоги строк. В строке каждый следующий эл-нт занимает ячейку памяти со следующим по порядку адресом. Элементы строки в памяти размещают произвольно, если каждому элементу дать явн указание места: где находится следующий за ним элемент. Тогда каждый элемент строки д им **2 поля**: в 1ом (**Element)** – символ строки– **тело звена**, во 2м (**Adrcled**) – ссылка на следующий элемент строки (адрес следующего элемента) – **справочная часть.**

Каждая такая пара = **звено**. Ссылки сцепляют звенья в 1 **цепочку**, а такой способ последовательности эл-ов - **сцепление**. Сцепление применяется для представления сложных динамических структур данных – строк, списков, деревьев и т д. Т о, для представления звена надо использ-ть запись из 2 полей.

*Пр-р 5.1.* Объявление звена цепочки.

**Type**

**Adr = ^Zveno; // Adr -** ссылки на программные элементы типа **Zveno**

**Zveno = Record**

**Element: Char;**

**Adrcled: Adr**

**End;**

**Adr** *-* ссылки на программные элементы типа ***Zveno***. Имя типа ***Zveno*** используется до его описания (при описании типа ***Adr*** (ссылоч-го типа)). **Ссылочный тип –** (!) тип, где м использовать имя до его описания. Обратную последовательность объявлений (вначале описать тип ***Zveno***, а потом – тип ***Adr***) в прим 7.1 нельзя использовать: в типе **Zveno** использ-ся неописанный тип **Adr**, а **Zveno** не является ссылочным.

Последнее звено цепочки д б со ссылкой **Nil –** признак **конца** цепочки.

Для работы с цепочкой надо использовать **2 указателя**: **ссылку** на ее I звено (**Adr1**) и на текущее (**Adrzv**). Ссылки д иметь тип **Adr:**

**Var**

**Adr1, Adrzv: Adr;**

Схематично цепочка представлена так:

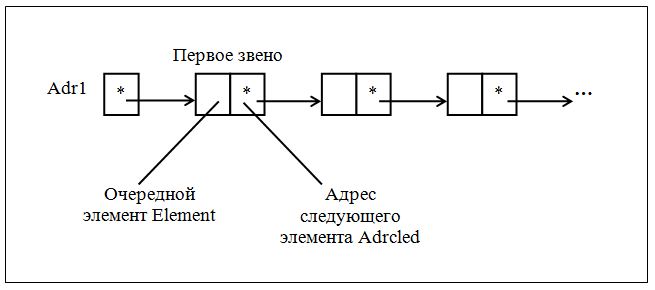


Рис 5.1. – Схематичное представление цепочки

Для удобства работы с цепочкой в нее включается заглавное «**0**-ое» звено (это **не** обязательно). В поле **Adrcled** данного звена содержится ссылка на I звено цепочки (его адрес). Поле **Element** - для хранения инфы **о строке.**

* 1. **Формирование цепочки (стр 82(7.1.2.)**

Пусть есть объява звена цепоч по прим 5.1.

Type

Adr = ^Zveno; // Adr - ссылки на программные элементы типа Zveno

Zveno = Record

Element: Char;

Adrcled: Adr

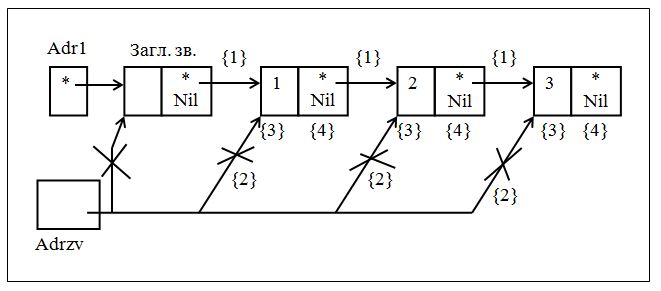
End;

**Алгоритмформирования цепочки*:***

Предварительно перед выполнением этапов 1 – 6 надо сформир-ть **заглавное** «**0**-ое» звено – голову?. При формиров цепочки текущее звено –всегда последнее, поэтому адрес следующего звена **– Nil**

1. Отвести область памяти для следующего звена. Его адрес занести в поле **Adrcled**текущего звена. Это всё происходит в 0-ом звене?
2. Новое звено (I) сделать текущим (занести его адрес в указатель текущего (предыдущ) 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** этапы алгоритма 1-5 для создания следующего звена.

Схема к алгоритму формирования цепочки с заглавным звеном - рис 5.2. №№ в фигурных скобках - **действия соответств-их этапов алгоритма.**



## Рис 5.2 – Схема алгоритма формирования цепочки

Пр-р 5.2. **Формирован** цепочки: **ввод** исходной строки и представление ее в виде цепочки. Признак конца строки – **точка**. Раздел типов = пр-ру 7.1

**Type**

Adr = ^Zveno;

Zveno = **Record**

Element: Char;

Adrcled: Adr

**End**;

**Var**

Adr1, Adrzv: Adr; {Adr1 – адр заглавн звена, Adrzv – адр текущ зв}

Simv: Char;

{======== Формирование заглавного звена цепочки – головы ========}

**Begin**  //никаких звеньев нет. Всё пусто. Начинаем с 0.

(1) **New**(Adr1); {Отвели место в памяти д/заглавного звена цепочки}

(2) Adrzv:= Adr1; {Формируем **адрес** текущего звена}/первое звено рождаем

Adrzv^.Adrcled := Nil; {При формиров цепочки текущее звено – последнее!

поэтому адрес следующ зв – Nil}/прощупали – там пусто

//---конец формир головы

**Read (Simv);** {Читается **первая** буква исходного текста}

**While** Simv <> '.' **Do** {Формирование текущих звеньев цепочки:}

**Begin**

{1} **New** (Adrzv^.Adrcled); {Дать обл памяти д/следующ звена, где б **I** буква?

Его адрес занести в поле Adrcled текущего звена}

{2} Adrzv:= Adrzv^.Adrcled; //Новое звено сделать текущим

//(его адрес занести в указатель текущего звена Adrzv)

{4} Adrzv^.Adrcled:= Nil; //В поле Adrcled текущего звена внести Nil

{3} Adrzv^.Element:= Simv;//В поле Element текущ звен занест очередн симв

{5}**Read (Simv);**  //Прочитать следующий символ исходного текста

**End;**

№№{1} – {5} = №№ этапов алгоритма формирования цепочки.

3 **операции** над динамическими цепочками : **поиск**, вставка, удаление.

### **5.3. Поиск эл-нта в цепочке (стр 84(7.1.3)**

- надо последовательно перебирать все звенья цепочки. Для перехода от звена к звену – выполнять оператор **присваивания в цикле**

**Adrzv := Adrzv^.Adrcled**

Т е присваивать указателю текущего звена **ссылку** на следующее в качестве нового значения. Данный оператор присваивания – аналог оператора

**i := i + 1**

для получения № **i** следующего эл-та при векторном представлении строки типа **Array Of Char***/***String*.***

*Пр-р 5.3.* Поиск эл-нта в строке-цепочке. Посчитать к-во его вхождений в строку:

..........

**Var**

K: Integer; {Счетчик числа вхождений искомого элемента}

Bykva: Char; {- буква, которую надо найти – ее №}

**Begin**

…

Adrzv := Adr1; {Адресу текущ зв присваив-ся знач-ие адреса нулевого зв

(заглавного)}//голова станов-ся текущим?

K := 0;

Read(Bykva); {Чтение искомой буквы}

**While** Adrzv^.Adrcled **<> Nil** **Do** {**Пока не конец строки**}

**Begin**

Adrzv:= Adrzv^.Adrcled; {Указат-ю присваив знач адреса следующ зв}

**If** Adrzv^.Element = Bykva **Then** {Сравнив содержимое поля эл-та

текущего звена с буквой}

K := K+1;

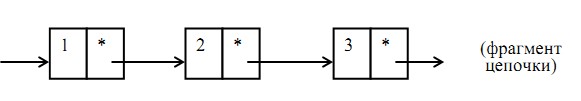
**End**;

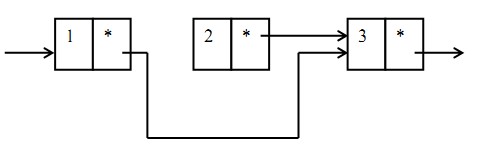
**…**

### **5.4. Удаление эл-нта из цепочки (стр 85(7.1.4.)**

Удаляемый эл-нт удобно задавать ссылкой на звено, за которым идет этот эл-нт – надо изменить **ссылку** у **предыдущего** элемента. If какое-то звено есть, но на него нет ссылки из другого звена, то оно недоступно при последов-ом переборе звеньев цепочки. => это звено в цепочку не входит.

Напр-р, в исходн цепочке надо удалить эл-нт 2: зв 1 ссылалось на звено 3





Т о, для исключения эл-нта из цепочки надо изменить ссылку у предыдущего эл-нта. За новую ссылку у этого эл-нта принять ссылку исключаемого эл-нта.

*Пр-р 5.4.* Удаление из цепочки эл-нта **2**: д/удаления использ-ть оператор

**Adrzv^.Adrcled := Adrzv^.Adrcled^.Adrcled**

Поле адреса эл-та 1 Поле **адреса** эл-та 2(исключаемого)

в указателе текущего звена **Adrzv** д б адрес эл-нта 1, предшествующего исключаемому.

*Пр-р 5.5.* Процедура удаления эл-та из цепочки (см пр 7.1–7.3)

**Procedure Udal (Zv: Adr);**

**Var** A: Adr; {A – вспомогательная переменная}

**Begin //А – зв 1?**

A := Zv^.Adrcled; {В А положили **адрес** удаляемого звена (2)}

Zv^.Adrcled := Zv^.Adrcled^.Adrcled; {Удален звена из цепочки}

{или равносильно Zv^.Adrcled := А^.Adrcled;}

Dispose (A) {Освобожден памяти, занимаемой удаленным звеном}

**End**;

**Моё понимание**: взяли некое звено (как пустой стакан для замены 2х данных), туда положили адрес на звено 2 – образовалась дырка в адресе зв 1. В эту дыру вставили адрес на зв 3, которое было у звена 2. А некое зв А с адресом на зв 2 – вэк.

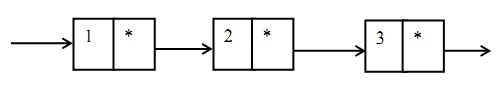
В качестве параметра Zv процедуры д передаваться адрес звена 1, (предшествующий удаляемому).

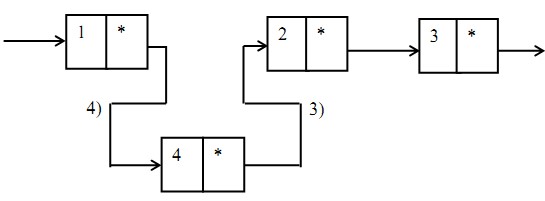
В Пр-ре 5.4 удаление эл-нта из цепочки идет быстрее, но память расходуется неэффективно: нет процедуры **Dispose**. В Пр-ре 5.5 удаление медленнее, но память расходуется эффективнее.

Способ удаления выбирать исходя из требований к характ-кам проги.

### **5.5. Вставка эл-нта в цепочку (стр 86(7.1.5.)**

- объединение отдельных звеньев в 1 цепь. Пусть в цепочку после элемента 1 надо вставить **элемент 4**.





**Алгоритм** вставки нового звена (после заданного):

1. Создать **новую** динамическую переменную (запись типа **Zveno**), которой б представлено (куда занесем) **вставляемое** звено 4.

2. В поле **Element** этой переменной занести вставляемый эл-нт (символ).

3. В поле **Adrcled** этой переменной (зв 4) занести ссылку **из** поля **Adrcled** предыдущего звена 1 (из зв 1 забрать адр зв 2).

4. В поле **Adrcled** предыдущего звена 1 занести ссылку на это **новое звено 4**.

№№ 3), 4) – действия, соответств-ие этапам 3 и 4 алгоритма на рис.

**Пример 5.6.** Процедура вставки элемента в цепочку.

№№ операторов == №№ этапов алгоритма вставки.

**Procedure** Vstav (Zv: Adr; El: Char); {**Zv** – адрес предыдущего звена}

**Var**

Q: Adr;

**Begin**

{1} **New** (Q); {Созд нов динам перем (Zveno) –ей дадим встав-мое звено 4}

{2} Q^.Element := El; {В поле Element эт премен занести встав-мый эл-т}

{3} Q^.Adrcled := Zv^.Adrcled; {В Adrcled эт перем - ссылку из Adrcled зв 1}

{4} Zv^.Adrcled := Q {В Adrcled зв 1 - ссылку на это зв 4}

**End**; //№№ операторов – соответствуют №№ алгоритма вставки 1-4.

5.6 Абстрактный тип данных «список» (c 87(7.1.6. Лин 1направл спис)

В математике список - последоват-ть эл-тов типа: ***а*1, *а*2, …., *аn*,** где *n* ≥ 0 и все **аi** имеют 1 тип. Кол-во = *n* – длина списка, если *n* ≥ 1, то *а*1 – 1й эл-нт, *аn* – последний. При *n* = 0 - пустой список.

Динам строка-цепочка - частный случай **линейного 1направл-го списка.**

**Списки** - гибкая стр-ра: их легко сделать большими или меньшими; их эл-нты доступны для вставки или удаления в любой позиции списка, объединять или разбивать на меньшие списки.

При реализации списков с помощью массивов­ – эл-нты располагаются в смежных ячейках массива: м легко просматривать содержимое списка и вставлять новые эл-нты в конец. Но вставка в середину списка требует перемещения всех последующих эл-тов на 1 позицию к концу массива. Операция удаления тоже: перемещение эл-тов списка д/освобожд ячейки.

Рассмотрим реализацию 1-онаправлен-го связанного списка с использ-ем указателей, связывающих последовательные эл-нты списка.

**1-онаправленный связанный список** - динамическая стр-ра данных, число эл-тов м меняться по мере помещения / удаляются в список. Эта реализация освобождает от непрерывной области памяти для хранения списка и от необход-ти перемещать эл-ты списка при вставке / удалении эл-тов. Но цена за это удобство - использ-ие д/п-ой памяти для хранения указателей.

Список состоит из ячеек, каждая из которых содержит эл-нт списка и указатель на следующую ячейку. Если список состоит из эл-тов ***a*1, а2, ... а*n*,** *i = 1, 2,…, n-1,*то ячейка с эл-нтом **а***i*, имеет указатель на ячейку с эл-нтом **а*i*+1.** Ячейка с эл-нтом **а*n*** имеет указатель***nil*** (0). Есть ячейка **header** (заголовок, **не содержит эл-тов списка**), которая указывает на ячейку, содержащую ***a*1.** В пустом списке заголовок имеет указатель ***nil*** (не указывающий ни на 1 ячейку) – **пустой (нулевой)** список (не содержит эл-тов). На рис 5.7 - 1направленный связанный список:

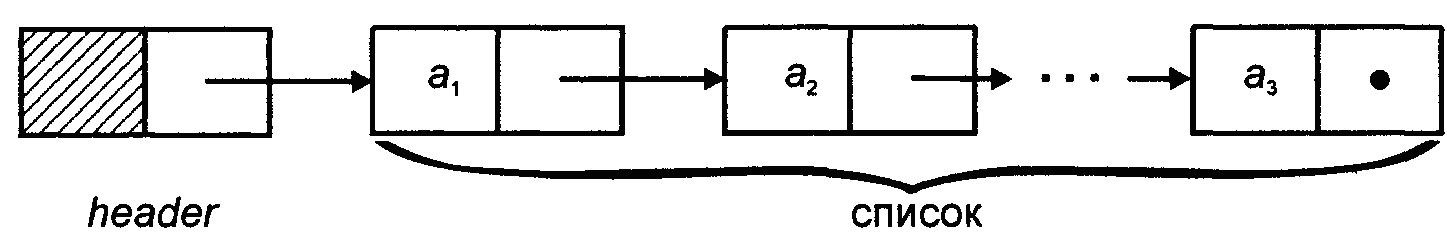


Рис 5.7 – 1направленный список

В1онаправленном списке определение позиций эл-тов отличается от определения в массиве. Здесь для ***i=2, 3, … , n*** позиция *i* определяется как указатель на ячейку, содержащую указатель на эл-нт **а*i***. Позиция 1 – указатель в ячейке заголовка, а позиция *end*(L) – указатель в последней ячейке списка L.

Динам строка-цепочка - частный случай **линейного 1направлен-го списка**. В случае цепочки информационные элементы списка – символы типа **Char**, а в общем случае информ элементы – значения любого типа: числа, массивы, записи, …. Принцип организации **информац**ионных эл-ов в список **тот же**: этот эл-нт очередного звена снабжается ссылкой на следующее звено.

**Пример 5.7.** Определение **структуры** звена 1направл-ого списка.

**Type**

**Adres1 = ^Zveno1;**

**Zveno1 = Record**

**Element: <Тип\_элемента\_списка>**

**Adrcled: Adres1;**

**End;**

Эл-нты списка - значения 1 и того же типа. Операции с эл-нтами списка те же, что и для звеньев динамической цепочки: поиск, вставка, удаление.

*«-»* – по нему можно двигаться только в 1ом направл-ии: от заглавного звена списка к последнему: замедляет работу.

# ТЕМА 6. ДВУНАПРАВЛЕННЫЙ СПИСОК

## 6.1 Стр-ра двунаправленного списка (с 87(7.2.)

- от каждого звена по списку двигаться в любом направлении. Каждое звено 2направленного списка содержит **2 поля** ссылочного типа. Значение 1о поля - ссылка на **следующее** звено списка, значение 2го поля - ссылка на **предыдущее (**звено списка).

**Структур**а звена 2направлен-го списка опред-ся описанием **типа** пр 6.1:

*Пр-р 6.1.*Описание **типа** звена 2направленного списка.

**Type**

**Adr2 = ^Zveno2;**

**Zveno2 = Record**

**Element: <Тип элемента-списка>;**

**Adrcled: Adr2;**

**Adrpred: Adr2;**

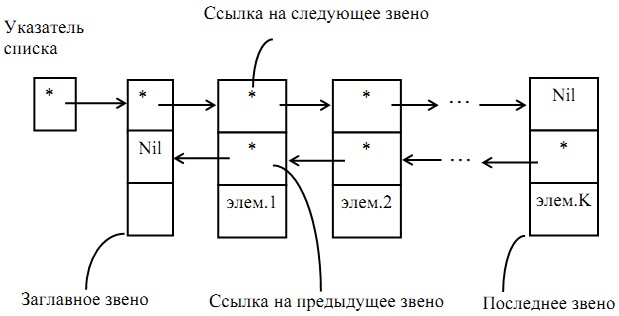
**End;** 

Рис 6.1 – Представление двунаправленного списка с заглавным звеном: «**элем. *i***» (эл-т ***К***) - **информац**ионная часть ***i-***ого звена. У заглавного звена списка **нет** предыдущего элемента. У последнего звена списка **нет** следующего элемента, => в поле **Adrcled** последнего и в поле **Adrpred** заглавного звеньев – пустая ссылка **Nil.**

На основе 2направленного списка м организовать 2**направленные кольцевые списки**: значение поля **Adrcled последнего** звена – ссылка на **заглавное** звено, а знач-е поля **Adrpred** заглавного – ссылка напоследнее:



Рис 6.2 – 2направлен кольцевойсписок с заглавным звеном (**1й способ** организации кольца**:** заглавное звено списка включено в кольцо.)

«+»1го способа – простореализуется вставка нового звена как в начало списка, так и в конец.

*«-»* – при **циклической** обработке элементов списка надо проверять, не является ли очередное звено заглавным (звеном списка). У 2го способа данного недостатока **нет**, но труднее реализовать добавление звена в конец списка.

При 2**ом** способе заглавное звено списка **не вкл**ючено в кольцо (рис 6.3).

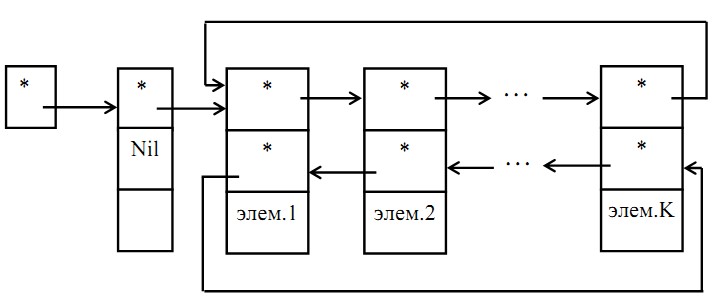


Рис 6.3 –2направлен кольцев спис с заглавным зв (2й способ организ кольца)

Над 2направленными списками те же *3 операции*, что и над строкой-цепочкой: поиск, вставка, удаление. Ниже – их реализация для 1го способа организации кольцевого списка. Для работы с двунаправленными списками использ-ть 2 указателя – на **заглавное** и **текущее** звенья.

## 6.2 Вставка эл-нта в двунаправленный список (стр 89(7.2.1)

Описание типа звена 2направленного списка.

**Type**

**Adr2 = ^Zveno2;**

**Zveno2 = Record**

**Adrcled: Adr2;**

**Adrpred: Adr2;**

**Element: <Тип** элемента-списка**>**

**End;**

**Алгоритм** **вставки** элемента в 2направленном кольцевом списке:

1. По**рождение** **нового** звена – поржденное звено Q.
2. Занесение **вставляемого элемента** **3** в информационное поле порожденного звена.
3. Заносим **в** поле **Adrcled** рожденного звена **3 ссылку** на следующ эл-т из предыдущего звена **1** **Q^.Adrcled:= Predzv^.Adrcled;**
4. вносим в поле **Adrpred** рожденного зв ссылку на предыдущ эл из следующего звена **3**. **Q^.Adrpred := Predzv;**
5. Занесение в поле «**Adrpred»** звена **2 (**следующего за вставляемым) ссылки на вставляемое (Q) звенo. **Predzv^.Adrcled^.Adrpred := Q;**
6. Занесение в поле «**Adrcled»** предшествующего звена 1 ссылки на вставляемое (**Q**) звено. **Predzv^.Adrcled := Q;**

{3} Q^.Adrcled := Predzv^.Adrcled; //из зв1 ссыл на поле зв 2

{4} Q^.Adrpred := Predzv; //из зв 2 ссыл на поле предыд зв 1 + ссылка на

предыдущ зв в проц-ру **Vstav**перед-ся как **параметр**

{5} Predzv^.Adrcled^.Adrpred := Q;

{6} Predzv^.Adrcled := Q; //заплн пустоту зв 1 поле след

Т е: 1) родить новое звено, 2) вставить эл-т в инфополе рожден звена, 3) в рожденном звене: в поле ***Adrcled*** вв ссылку на след эл-т из предыд звена + в ***Adrpred*** вв ссылку на предыд эл-т из следующего звена, 4) в следующем звене в ***Adrpred*** вв ссылку на рожденное зв, 5) в предыдущ зв в ***Adrcled*** вв ссылку на рожденное зв.

Схематично вставка элемента в 2направл кольцевой список с заглавным звеном – на рис 6.4, 6.5:

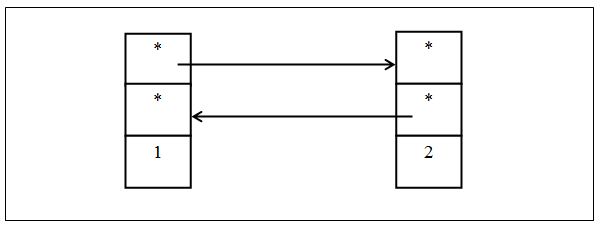


Рис 6.4 – исходноe двунаправленного списка

В исходный список после элемента **1** вставляется элемент **3**.

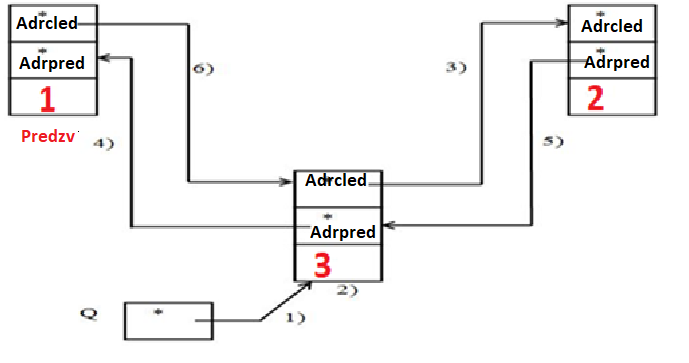


Рис 6.5 –результир 2направл спис. 1)–6) =№ этапов алгоритма. Откуда берем

*Пр-р 6.2.*Процедура вставки.

**Procedure Vstav** (Elem: <Тип\_эл-та >; **Predzv: Adr2**); {Elem – вставляем эл-т,

Predzv – ссыл на предыдущий эл-нт, передается как **параметр**}

**Var** Q: Adr2; //тип запись

**Begin**

{1} **New** (Q); //Порождение нового зв – поржденное звено (см рис)

{2} Q^.Element := Elem; //Занесен вставляемого эл-та в инфо-поле рожд зв

{3} Q^.Adrcled := Predzv^.Adrcled; {внос из предыд зв в поле **Adrcled** зв ссыл

на следующ эл-нт. (см рис)}

{4} Q^.Adrpred := Predzv; {внос в поле **Adrpred** рожд-го зв ссыл на

предыд эл-т из следующего звена.}

{5} Predzv^.Adrcled^.Adrpred := Q; {внос в поле Adrpred след-го за вставл-ым

звена (2) ссылки на вставляемое звено (3).

{6} Predzv^.Adrcled := Q; {внос в поле **Adrcled** предыдущ зв

ссыл на вставляемое (рожденное) зв}

**End**;

№№ операторов {1-6} == №№ этапов алгоритма вставки. Реализация **{4}** в примере **отличается** от этапа 4 алгоритма, т. к. ссылка на предыдущее звено в процедуру **Vstav**передается как **параметр**.

## 6.3 Создание двунаправленного кольцевого списка с заглавным звеном

**(стр 91(7.2.2.)**

Пусть в проге описание типа, как в Пр-ре 6.1.

- на примере: Для создания **кольцевого списка** м использовать процедуру **вставки** – см пр 6.2; на каждом шаге создан спис д б з**акольцо**ван

*Пр-р 6.3.* **Ввод** исходного **текста** и его представление как 2направленный кольцевой список **с заглавным** звеном. Признак окончания текста – точка. Раздел типов == прим. 6.1.

Type

Adr2 = ^Zveno2;

Zveno2 = Record

Adrcled: Adr2;

Adrpred: Adr2;

Element: <Тип элемента>

End;

Procedure Vstav (Elem: <Тип\_эл-та\_списка>; Predzv: Adr2);

Var

Q: Adr2;

Begin

{1} New (Q); //По**рожд** **нового** звена – поржденное звено (см рис)

{2} Q^.Element := Elem;

{3} Q^.Adrcled := Predzv^.Adrcled; {внос **ссылки** на след эл-т **из пред**ыд}

{4} Q^.Adrpred := Predzv; {внос ссылки на предыд эл-т **из следующего** зв}

{5} Predzv^.Adrcled^.Adrpred := Q; {внос в зв 3 след ссыл на вставляем зв}

{6} Predzv^.Adrcled := Q; {Занес ссылки на вставляемое звено}

End;

**VAR**

**Ring:** Adr2**;**{**Адрес** заглавного звена (Головы) кольца}

**Bykva:** Char**;**{**Символ** исходного текста}

**BEGIN**

{7} **New (Ring);**{- место в памяти д/**заглавн**ого звена}

{8} **Ring^. Adrcled := Ring;** {За**кольцовывается** заглавн зв, т к на кажд

промежуточном этапе список д б кольцом. Само на себя указыв?

{9} **Ring^. Adrpred := Ring;**

**Read (Bykva)**  {Чтение первой буквы текста}

**While Bykva <> ’.’ Do**

**Begin**

**Vstav (Bykva, Ring^. Adrpred);** {**Вызов** проц-ры вставки. Адрес предыдущ созданного звена (последнего звена кольца) **всегда** хранится в заглавном звене в поле **Adrpred**}

**Read (Bykva);**  {Чтение очередной **буквы текста**}

**End**

**END.**

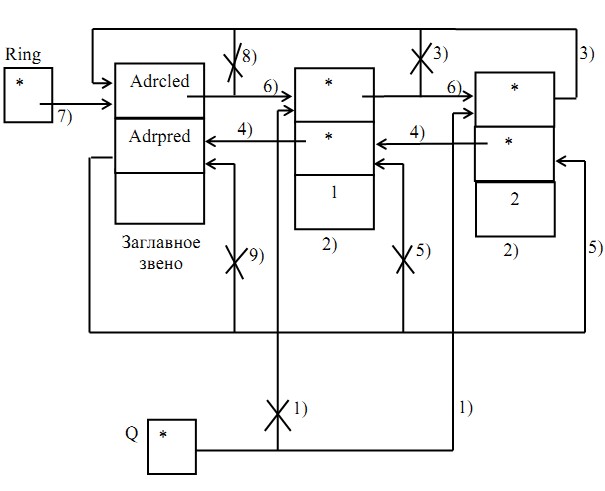
Схематические пояснения к проге - на рис 6.6.

Рис 6.6 – Схематические пояснения к проге Пр-ра 6.3. 1) – 6) - №№ этапов алгоритма вставки, приведенного в предыдущем подразделе, 7) – 9) – №№ соотв-их операторов из Пр-ра 3. Список закольцовывается после добавления каждого нового звена.

## 6.4 Удаление эл-нта (стр 92(7.2.3.)

Пусть в проге есть описание типа, как в Пр-ре 6.1.

**Алгоритм** удаления элемента (звено 2) – в 2 раза < операц, чем при вставке зв:

1) внесен ссылк в поле ***Adr*pred** зв (3) **из** поля **Adrpred** зв(2) на зв (1)

**Udzv^.Adrcled^. *Adr*pred := Udzv^. Adrpred;**

2) Занесен в поле *Adr***cled** зв (1) (предыдущего удаляемому) ссылки на зв(3) (следующ за удаляемым) из поля **Adrcled** удаляемого зв 2

**Udzv^.Adrpred^.Adrcled := Udzv^. Adrcled;**

3) **Уничтожение** удаляемого звена (2).

На рисунках - удаление элемента из 2направленного списка.

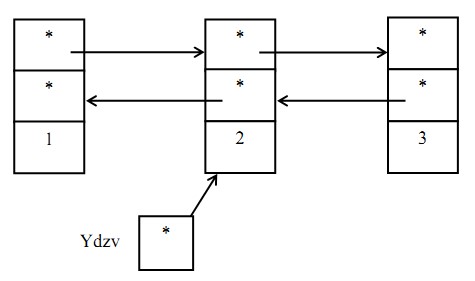


Рис 6.7 – Фрагмент исходного двунаправленного списка. ***Ydzv*** - ссылка на удаляемое звено.

Результат удаления звена 2. №№ связей соответствуют №№ этапов алгоритма удаления.

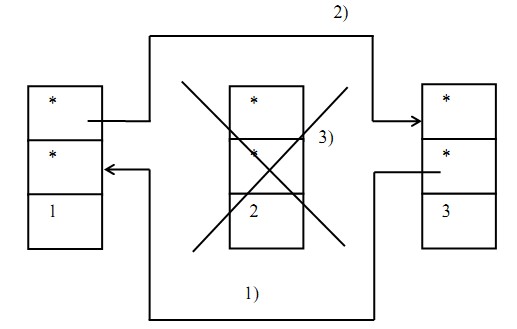


Рис 6.8 – Результат удаления звена 2

*Пр-р 6.4.* Процедура удаления звена из двунаправленного списка. №№ операторов == номерам этапов алгоритма удаления звена.

Type

Adr2 = ^Zveno2;

Zveno2 = Record

Adrcled: Adr2;

Adrpred: Adr2;

Element: <Тип элемента >

End;

**Procedure Udalen (Udzv: Adr2); {Udzv** – ссылка на удаляемое зв}

**Begin**

{1} **Udzv^.Adrcled^.Adrpred := Udzv^.Adrpred; //**внесена ссылка в поле

***Adr*pred** зв (**3**) **из** поля ***Adr*pred** зв (2) на зв (1) ;

{2} **Udzv^.Adrpred^.Adrcled := Udzv^.Adrcled;** //внесена ссылка в поле

*Adr***cled** зв (**1**) **из** поля *Adr***cled** зв (2) на зв (3) ;

{3} **Dispose (Udzv); Уничтожение** удаляемого звена (2).

**End;**

6.5 Поиск эл-нта в двунаправленном кольцевом списке - **Стр 93 ОАИП (7.2.4.)**

- как поиск элемента **в цепочке**.

Особенность: в кольцевом списке **нет** последнего элемента формально - каждый элемент имеет ссылку на следующий - учитывать при цикле поиска!

*Пр-р 6.5.* Логическая ф-ция поиска эл-нта в двунаправленном кольцевом списке **с** заглавным звеном. Если искомый эл-нт в списке есть, возвращаемое значение ф-ции = ***True***, а параметру ***Iskadr*** присваивается ссылка на звено, содержащее данный эл-нт.

**Function** Poisk (Adr: Adr2; Elem: <Тип\_эл-нта\_списка>; **Var**

Iskadr: Adr2): Boolean; {Adr – ссылка на заглавное зв; Elem – искомый эл-нт; Iskadr – адрес искомого эл-нта}

**Var**

P, Q: Adr2; {В Р б храниться адрес заглав-го зв, в Q

– адрес текущего звена}

B: Boolean;

**Begin**

B:= False; {B – логическая переменная (факт наличия

искомого эл-нта)звена}

P:= Adr; {В P занесен адрес заглавного звена}

Iskadr:= Nil;

Q:= P^.Adrcled; {В Q занесен адрес первого зв}

While (P <> Q) And Not B Do {Поиск, пока не дошли до

заглавн зв и не нашли искомый эл-нт}

**Begin**

**If** Q^.Element = Elem **Then** {Найден искомый Эл-т}

**Begin**

B := True;

Iskadr := Q {Адрес искомого эл-нта}

**End**;

Q := Q^.Adrcled {Переход к следующему звену}

**End**;

Poisk := B {Возвращаемое значение}

**End**;

# ТЕМА 7.ТАБЛИЦЫ И СПОСОБЫ РАБОТЫ С НИМИ

**Стр 99 ОАИП (7.4.)**

## 7.1. Общие сведения

использ-ются *автоматизированные информационные системы* (**АИС**). Их назначение: 1) хранение большого числа сведений, 2) прием новых сведений и 3) выдача хранимых сведений по запросам. сведения представляют записями.

Гл задача создания АИС: организация выдачи по запросу любой из записей, независимо какая запись выдавалась перед ней.

Основа АИС - базы данных (БД). Для организации БД используют стр-ы Дан, называемые **таблицами**. В таблице каждая **запись** имеет свое **имя**. Т о, **таблица** – это набор именованных записей. **Имя** записи называется **ключом записи (имя = ключ)**.

Для организации эффективного поиска записи надо, чтобы над **ключами** м было реализовать операции сравнения (=, <, >): ключи д б упорядоченными.

Ключом м б целые положительные числа или строки символов 1й длины. Операции над таблицей*:*

1. поиск в таблице записи (**Таб Зап**) с заданным ключом;
2. включение в Таб Зап с заданным ключом (если в таблице уже

есть запись с таким ключом, то старая запись заменяется на новую);

1. удаление из Таб Зап с заданным ключом.

**7.2 Способы организации таблиц Стр 100 ОАИП (**7.4.2.**)**

1) *1направленный список*

Таблица представляется с помощью 1направлен-го списка. Каждое звено списка должно иметь ключ, текст записи и ссылку на следующее зв. *Достоинства* *способа*:

а) эффективное использ-ие памяти компа (д/п инфа в звене – только ссылка на следующее звено);

б) простой алгоритм перебора записей для поиска нужной записи;

в) просто включать в таблицу заведомо новой записи – как новое

звено в конец списка.

*Недостаток* *способа* – большое вр поиска нужной записи из-за факторов:

а) при наличии в таблице *N* записей для поиска нужной надо

просмотреть в среднем ***N/2*** эл-тов списка;

б) если в таблице **нет** записи с нужным ключом, то нужно просмотреть

все ***N*** записей.

1. *1направленный список с упорядоченными записями*

Записи в списке - по возрастанию их ключей.

*Достоинства способа*: меньшее вр поиска записи по сравнению со способом 1). Поиск записи требует в среднем просмотра ***N/2*** записей, независимо от того, есть эта запись в таблице или нет.

*Недостаток*: усложняется процедура включения новой записи в таблицу.

1. *1направленный список с отдельным хранением текста записи*

- для ускорения поиска записи. Тексты записей хранятся отдельно от ключей. При ключе хранится только ссылка на текст записи. Схематическое представление - на рис 7.1.

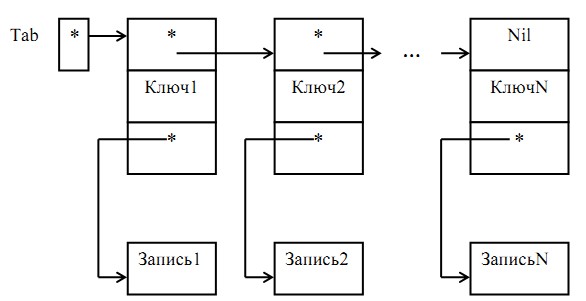


Рис 7.1 – Схематич представление табл в виде списка с отдельным хранением текста записи

1. *Представление в виде массива*

Каждый эл-т таблицы - запись, содержащая 2 поля: ключ записи и ссылка на текст записи. Такие эл-нты обьединяют не в список, а в 1-омерный массив.

*Пр-р 7.1.*Объявление типа таблицы при использовании массива записей с 2-я полями: ключ записи и ссылка на текст записи.

**Type**

Index = 1..N;

Text1 = <Тип\_текста\_записи>;

Adr = ^Text1;

Element = **Record**

Kl: Integer; {Ключ}

Adrzap: Adr

**End**;

Mas = **Array** [Index] **Of** Element;

**Var**

Tabl: Mas;

Схематическое представление таблицы в виде массива - рис 7.2.

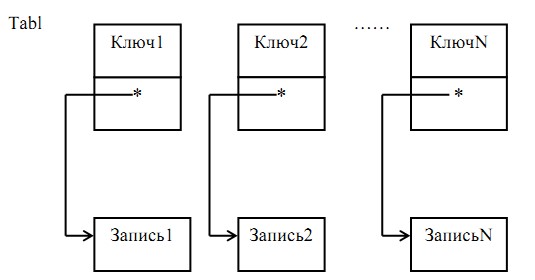


Рис 7.2 – Схематическое представление таблицы в виде массива

Для организации эффективного поиска в таблице, реализованной таким способом: эл-нты массива д б упорядочены по возрастанию ключей.

Задача поиска записи с заданным ключом сводится к задаче поиска эл-нта массива ***Tabl***, в котором содержится этот ключ. Если определен индекс *I* этого эл-нта, то искомая запись является значением переменной *Tabl[I].Adrzap^*.

Для поиска эл-нта с заданным ключом - *метод половинного деления (дихотомический поиск).* Его суть:

Берем средний эл-нт массива с индексом (номером) *N/2*. Если искомый ключ *К* меньше ключа в эл-нте с номером *N/2*, то требуемый эл-нт - в первой половине массива, иначе – во второй. Следующий этап - нужная половина массива опять делится пополам и определяем, в какой из половин соотв-ий эл-нт и т д. Конец - если серединный компонент текущей зоны поиска содержит заданный ключ (требуемая запись найдена) или если зона поиска - пустая (записи с нужным ключом в таблице нет).

Т к на каждом шаге зона поиска уменьшается в 2 раза, то для завершения поиска надо не более шагов. Т о, эффект способа по сравнению со способами 1 и 2 быстро возрастает с ростом ***N***.

*Пр-р 7.2.* Логическая ф-ция *Poisk* поиска нужного ключа в таблице, реализующая метод дихотомического поиска. Ф-ция *Poisk* возвращает значение True, если нужный ключ в таблице есть, иначе - *False*. Ф-ция *Poisk* имеет 2 параметра: *К* – искомый ключ, *Nomer* – номер эл-нта массива *Tabl*, в котором хранится нужный ключ *К* (в параметр-переменную *Nomer* ф-ция устанавливает значение индекса (№ эл-нта) массива *Tabl*, содержащего нужный ключ).

Пусть имеется объявление по Пр-ру 7.1. (стр 104 в ОАИП 2 ч)

**Type**

Index = 1..N;

Text1 = <Тип\_текста\_записи>;

Adr = ^Text1;

Element = **Record**

Kl: Integer; {Ключ}

Adrzap: Adr

**End**;

Mas = **Array** [Index] **Of** Element;

**Var**

Tabl: Mas;

**function** Poisk(k: integer; **var** nomer: index):boolean;

**var**

lev, prav: integer; {Левая и правая границы поиска}

B: boolean;

i: index;

**begin**

lev:= 1;

prav:= n; {N – размер таблицы Tabl}

B:= false;

Nomer:= 0; {if в Nomer остает знач 0,то искомого ключа в Tabl нет}

**repeat**

i:= (lev + prav) **div** 2; {Nomer – № эл-та}

**if** K = Tabl[i].Kl **then** { К – искомый ключ }//if = - т е сразу же нашли!

**begin**

B:= true;

Nomer:= i;

**end**

**else**

**if** K < Tabl[i].Kl **then**//if < - ищем в левой ½

Prav:= i - 1

**else**

lev:= i + 1; //if > - ищем в правой ½

**until** B **or** (lev > prav); {Иском ключ найд/пустая зона(нет нужн ключа)}

Poisk:= B; {Возвращаемое значение ф-ции}

**end**;

«-» *способа (массив) 4* – плохо приспособлен для реализации добавления и удаления записей, т к надо сдвигать все следующие после вставляемого или исключаемого эл-нты массива в ту или другую сторону для поддержания упорядоченности эл-тов. Поэтому способ 4 удобен, если таблица **изменяется редко.**

# ТЕМА 8. ОЧЕРЕДИ И ОПЕРАЦИИ, ВЫПОЛНЯЕМЫЕ НАД НИМИ

## 8.1 Общие сведения об очередях (стр 93 ОАИП 2 ч 7.3.)

Понятие очереди в программировании – как в реальной очереди в жизни.

**Очередь** - динамическая стр-ра. Длина очереди и набор образующих ее эл-тов меняется с **течением вр.**

**2 операции** над очередью:

1. **занесение** эл-нта в очередь (заказ на обслуживание);
2. **выбор** эл-нта из очереди (д/обслуживания); выбранный эл-нт исключают из очереди.

В очереди - **2 позиции**: начало (из этой позиции выбирается эл-нт из очереди) и конец (в эту позицию помещается эл-нт).

2вида очереди (отличаются по дисциплине (способу) обслуживания в них эл-тов):

**1).** ***FIFO*** (*First In – First Out* – первый в очередь – первый из очереди) - поступивший в очередь первым заказ, первым выбирается для обслуживания и удаляется из очереди – сквозная труба.

2). ***LIFO*** (*Last In – First Out* – последний в очередь – первый из очереди) - заказ, поступивший в очередь последним, выбирается первым для обслуживания (и удаляется из очереди) – наз-ся **стек.** Чаще использ.

## 8.2 Операции, выполняемые над эл-нтами очередей. Очередь FIFO (сквозная труба)) (СТР 97 ОАИП 7.3.2.)

Для её организации использ-ют **2 ссылочные переменные** типа **Adres1** (см. пр 7.7)**:** **Left** – **начало** и **Right** – конец очереди. Слева направо

«+»е эл-та в очередь - в соотв со значен-eм **Right.** Затем значение **Right** изменяется и указывает на **последний** занесенный эл-нт

**Выбор**ка из очереди эл-тов - в соотв со значением **Left.** Затем **Left** изменяется и указывает на **следующий** эл-нт очереди.

Если в очереди 1 эл-нт - значения **Right =** **Left.** Такое равенство м использ-ть как признак **конца** очереди при последовательном выборе эл-тов.

**А. Занесение эл-нта в очередь FIFO**

**Алгоритм** занесения эл-нта **в** очередь:

1. Cоздание нового звена.
2. Занесение в последнее звено адреса нового звена.
3. Занесение ***Nil*** в поле **Adrcled** нового звена.
4. Занесение **эл-нта** (инфы) в информационное поле нового звена.
5. Созданное звено сделать **концом очереди**.

*Пр-р 8.1.*Процедура занесения эл-нта в очередь. Параметр ***Right*** – адрес **последнего** занесенного эл-нта, ***El*** – заносимое в очередь значение.

**Procedure** Dobavl (**Var** Right: Adres1; El: <Тип\_эл-тов\_очереди>);

**Var**

Q: Adres1;

**Begin**

{5} **New** (Q); {1-й этап алгоритма занесения}

{6} Right^.Adrcled := Q; {2-й этап алгоритма занесения}

{7} Q^.Adrcled := Nil; {3-й этап алгоритма занесения}

{8} Q^.Element := El; {4-й этап алгоритма занесения}

{9} Right:=Q {5-й этап алгоритма занесения} **End**;

*2.* ***Выбор эл-нта из очереди******(удаление****)*

Пусть в проге имеется описание типа, приведенное в Пр-ре 5.7 для описания звена структуры 1направлен-го списка.

*Алгоритм выбора эл-нта из очереди:*

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

*Пр-р* 8.2. Процедура выбора эл-нта из очереди. Параметр *Left* использ-ется для передачи адреса начала очереди. В параметр *Elem* передается значение из начала очереди.

**Procedure** Udal (**Var** Left: Adres1; **Var** Elem: <Тип\_эл-ов\_очереди>);

**Var**

Q: Adres1;

**Begin**

{1} Elem := Left^.Element; //Чтение значения из начала очереди.

{2} Q := Left; //Запоминание ссылки на начало очереди.

{3} Left := Left^.Adrcled; //Исключен 1-го звена из начала очереди

{4} Dispose(Q) // Уничтожение первого звена.

**End**;

*3.* ***Организация очереди***

Пусть в проге имеется описание типа, приведенное в Пр-ре 5.7 для описания звена структуры 1направлен-го списка. Организация очереди основана на использовании операции занесения эл-нта в очередь

*Пр-р 8.3.*Организация очереди. Ввод исходного текста в очередь. Признак окончания текста – точка. Использ-ется процедура *Dobavl* занесения эл-нта в очередь (см. Пр-р 8.1).

…………….

**var**

right, left: adres1;

Bykva: char;

**begin**

read(Bykva);

{1} **new**(left); //{1}-{4} – порождение первого (нового) зв очереди (левого)

{2} left^.adrsled:= nil;

{3} left^.element:= Bykva;

{4} right:= left;

read(Bykva);

**while** Bykva <> '.' **do**

**begin**

dobavl(right, Bykva);

read(Bykva);

**end**;

**end**;

Рис 8.1 - схематические пояснения к Пр-рам 8.1 – 8.13. Номера на рис - действия над очередью, соотв-ие №№ операторов в Пр-рах.

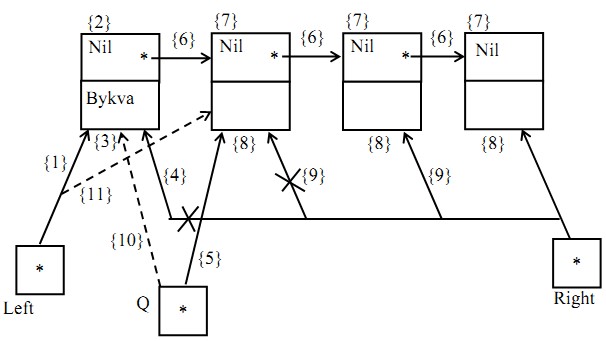


Рисунок 8.1 – Схематические пояснения к Пр-рам 8.1 – 8.3

## 8.3. Разновидности очередей (НЕТ В ОАИП)

**1) многопоточные очереди**(multi headed queues). Эл-нты в нее добавляются в конец очереди, но очередь имеет неск-ко потоков (front end) или голов (heads).

Пр-р в обычной жизни - очередь в банке: все в одной очереди, но их обслуживает неск-ко касс. Освободившаяся касса обслуживает клиента, который в очереди первый. Они эффективнее нескольких 1поточных очередей.

**2) Циклическая очередь**. Организация очереди на основе **массива** при достижении **предела** массива м вернуть указатели вставки/ удаления в/из очередь(-и) **на начало массива**. Тогда в очередь м добавлять/ удалять любое число эл-тов. Такая очередь называется циклической, т к массив использ-ется не как линейный список, а как циклический.

очередь **заполнена –** когда указатель свободного места совпадает с указателем выборки следующего эл-нта. Иначе очередь б иметь свободное место для нового эл-нта. В начале проги индекс выборки должен устанавливаться не в 0 значение, а на значение максимального числа событий. Иначе первое обращение к процедуре постановки в очередь приведет к появлению сообщения о заполнении структуры. Помнить: очередь м иметь только на 1 эл-нт меньше, чем значение максимального числа событий, т к указатели выборки и вставки в очередь всегда должны отличаться хотя бы на 1цу (иначе нельзя понять заполнена очередь или она пустая).

Циклические очереди применяются: 1) в ОС при **буферизации** инфы, которая считывается/ записывается на дисковые файлы/ консоль. 2) решение задач реального вр, когда, напр-р, пользователь м продолжать делать ввод с клавиатуры во вр выполнения другой задачи. Так работают многие текстовые процессоры, когда изменяется формат параграфа или выравнивается строка. Имеется короткий промежуток вр, когда набранная на клавиатуре инфа не выводится на экран до окончания другого процесса. Для достижения такого эффекта в проге д б постоянная проверка ввода с клавиатуры в ходе выполнения другого процесса. При вводе символа его значение должно быстро ставиться в очередь, и процесс должен продолжаться. После завершения процесса набранные символы выбирают из очереди и обрабатывают обычным образом.

**3) Приоритетные очереди.** Каждый эл-нт (priority queue) имеет свой приоритет. Если проге нужно удалить эл-нт из очереди, она выбирает эл-нт с наивысшим приоритетом. Как хранятся эл-нты в приоритетной очереди – не важно (если прога всегда м найти эл-нт с высшим приоритетом).

ОС использую такие очереди для планирования заданий. В ОС UNIX все процессы имеют разные приоритеты. Когда процессор освобождается, выбирается готовый к исполнению процесс с наивысшим приоритетом. Процессы с более низким приоритетом должны ждать завершения/ блокировки (напр-р, при ожидании внешнего события: чтение данных с диска) процессов с более высокими приоритетами.

Простой способ организации приоритетной очереди — поместить все эл-нты в список. Д/удаления эл-нта из очереди – найти в списке эл-нт с **наивысшим** приоритетом. Д/добавления эл-нта в очередь – поместить его в начало списка. При использовании этого м-да, для добавления нового эл-нта в очередь – надо (!) шаг. Д/поиска и удаления эл-нта с наивысшим приоритетом, надо O(N) шагов, если очередь имеет N эл-тов.

Лучше была бы схема со **связным списком**, где эл-нты упорядочены в прямом/ обратном порядке. Д/**добавления** эл-нта в очередь, надо найти его правильное положение в списке и поместить его туда. Д/упрощения поиска положения эл-нта, м использ-ть **сигнальные метки** в начале и конце списка, присвоив им приоритеты.

# ТЕМА 9. СТЕКИ И ОПЕРАЦИИ, ВЫПОЛНЯЕМЫЕ НАД НИМИ

## 9.1 Абстрактный тип данных «стек» (СТР 94 ОАИП 7.3.1.)

**Стек** – спец-ый тип списка, в котором все вставки и удаления выполняются только на одном конце, называемом вершиной (*top*). LIFO (last-in-first-out – последний вошел, а первый вышел) - м взять только верхний предмет, а добавить новый объект м, только положив его на верхний. Абстрактные типы данных семейства стек использ-ются 5 операторов.

## 9.2 Реализация стеков с помощью массивов

Т к вставка и удаление эл-тов - через вершину стека, его «дно» фиксируют внизу массива (в ячейке с **наибол**ьшим индексом) и дают стеку расти вверх массива (к ячейке с **наимен**ьшим индексом) – рис 9.1:

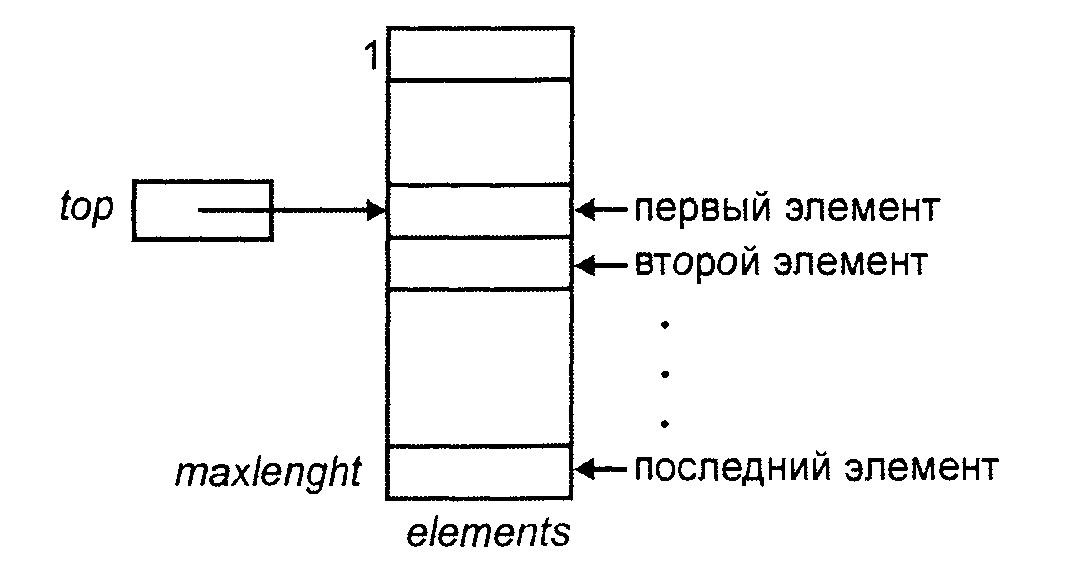


Рис 9.1 – Реализация стека на основе массива

Для такой реализации стеков абстрактный тип **Stack** м определить так:

**Type**

Stack = **record**

Top: integer; //т е число

Elements: **array**[1..maxlenght] **of** integer; //т е массив чисел

**end**;

**Var**

x : Stack;

Здесь стек состоит из последов-ти эл-тов **elements[top], elements[top+1], … elements[maxlenght]**. Если **top = maxlenght +1**, то стек **пустой**.

### **9.3 Реализация стеков с помощью указателей (СТР 95 ОАИП 7.3.1.)**

В стеке доступна (!) позиция, называемая ***вершиной стека* –** в ней находится последний стек эл-нт по вр поступления.

Наиболее быстрое выполнение операций над стеком - представление его в виде 1направлен-го списка. Вершина стека - первое звено списка. Заглавное звено в стеке не нужно: доступна только его вершина.

При использовании структуры 1направлен-го списка стек задается с помощью типа, приведенного при описании 1направлен-го списка. Д/п-о м б введен тип указателя, представляющего стек как единую структуру, т е ссылка на вершину стека *Stek*:

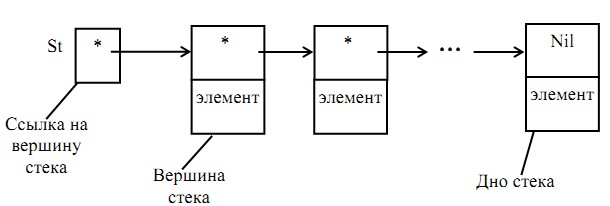
**Type**

Stek = Adres1; {Задание стека по пр-ру однонаправл-го списка}

**Var**

St: Stek; {Реальный стек вводится с пом описания переменной}

Схематично стек изображается аналогично цепочке (рисунок 9.2).



## Рис 9.2 – Схематичное представление стека

К началу создания стека его нужно сделать пустым: ***St := Nil;***

### **9.4 Операции над эл-нтами стека**

1*.* **Занесение эл-нта в стек.**Пусть в проге описание типа, приведенное в Пр-ре 5.7.

*Алгоритм занесения эл-нта в стек:*

1. **Создание** нового звена.
2. **Занесение** в новое звено эл-нта.
3. Занесение в новое звено адреса предыдущей вершины стека.
4. Созданное звено сделать **вершиной** стека.

*Пр-р 9.1.* Процедура занесения эл-нта в стек. Первый параметр задает адрес вершины стека, второй – заносимое в него значение.

**Type**

Adres1 = ^Zveno1;

Zveno1 = **Record**

Adrcled: Adres1;

Element: <Тип\_эл-нта\_списка>

End;

**Stek =** Adres1;

**Procedure Zanes (Var St : Stek; El : <Тип\_**эл-нта\_стека**>);**

**Var**

**Q: Adres1;**

**Begin**

{1} **New (Q); //** Создание **нового звена**.

{2} **Q^.Element := El;** //Занесение в новое звено **эл-нта**.

{3} **Q^.Adrcled := St;** Занесен в новое зв адреса предыд верш-ы стека

{4} **St := Q** //Созданное звено сделать **вершиной** стека.

**End;**

№№ операторов = №№ этапов алгоритма занесения эл-нта в стек.

2*.* ***Выбор*** *эл-нта из стека* Пусть в проге описание типа, как в Пр-ре 5.7.

*Алгоритм выбора эл-нта из стека:*

1. **Прочитать** значение из вершины стека.
2. **Запомнить** ссылку на текущую вершину.
3. **Исключить** текущую вершину из стека.
4. **Уничтожить** исключенную вершину.

*Пр-р 9.2.* Процедура выбора эл-нта **из** стека. 1й параметр задает нужный **стек** (если их неск-ко), во 2й передается значение из вершины стека.

**Type**

Adres1 = ^Zveno1;

Zveno1 = **Record**

Adrcled: Adres1;

Element: <Тип\_эл-нта\_списка>

End;

**Stek =** Adres1;

**Procedure Vibor (Var St: Stek; Var A: <Тип\_**эл-тов\_стека**>);**

**Var**

**Q:** Adres1**;**

**Begin**

{1} **A := St^.Element;** //Прочитать значение из вершины стека.

{2} **Q := St;** //Запомнить ссылку на старую вершину.

{3} **St := St^.Adrcled;** //**Исключить** первое звено из стека.

{4} **Dispose (Q)** //Уничтожить первое звено.

**End;**

Если надо ускорить процедуру выбора – операторы **Q:= St** и **Dispose(Q)** не применять. Но это - неэффективное использов-ие памяти.

3*. Создание стека*

Пусть в проге описание типа, как в Пр-ре 5.7. Для создания стека использована процедура ***Zanes*** из Пр-ра 9.1.

*Пр-р 9.3.* **Создан стека (в**в текста в стек) Признак end текста – точка

**…………………**

**Var**

St: Adres1;

Bykva: Char;

**Begin**

St := Nil; //обнулили, ничего пока нет

Read (Bykva);

**While** Bykva <> '.' **Do**

**Begin**

Zanes (St, Bykva); // Procedure Zanes()

Read (Bykva);

**End;**

**End**.

### **9.5 Применение стеков**

- в **рекурсивных** процедурах. Организация выполнения процедур в ЯП состоит в задании **структур данных** для хранения значений программных переменных во вр выполнения проги. Все ЯП, допускающие рекурсивные процедуры, использ-ют **стеки активационных записей** для хранения всех значений переменных, принадлежащих каждой активной процедуре. При вызове процедуры *Р* новая активационная запись для этой процедуры помещается в стек независимо есть ли в стеке другие активационные записи для процедуры *Р*. Т о, извлекая активационную запись из стека для последнего вызова процедуры Р, м управлять возвратом к точке в проге, из которой Р вызывалась (эта **точка**, называемая **адресом возврата**, помещается в активационную запись процедуры Р при вызове этой процедуры).

Типичная **конструкция рекурсивной процедуры** имеет вид.

**procedure** Rec(t: integer);

**begin**

**if** <проверка усл> **then** //действия на входе в рекурсию

Rec(**t + 1**); //действия на выходе из рекурсии

**end**;

Рекурсивные вызовы процедур упрощают структуру многих программ. Но в некоторых ЯП процедурные вызовы более «дорогие» (по вр выполнения), чем прямое выполнение операторов, поэтому прога м работать быстрее, если из нее исключить рекурсивные процедуры. На практике: после реализации части программного проекта приходится исключать рекурсию.

Неск-ко Пр-ров рекурсивных процедур и функций:

1) Вычисление факториала целого числа.

Сначала проверяем условие ***N=1***. Если оно выполняется, то ф-ции **Fakt** присваивается значение 1, на этом выполнение ф-ции закончено. Иначе (*N не=1)* выполняется вычисление произведения **N \* Fakt(N - 1).** Вычисление произведения - рекурсивного характера: идет вызов ф-ции **Fakt*(N-1)*,** значение которой вычисляется с помощью вызова ф-ции ***Fakt***, ее параметр – тоже ф-ция ***Fakt***, и т д, пока знач-ие формального параметра не станет = 1. После чего рекурсивные вызовы ф-ции ***Fakt*** больше не выполняются, а происходит вычисление ф-ции ***Fakt***для чисел, возрастающих от 1 до *N*, причем ф-ция ***Fakt*** каждый раз возвращает значение, = произведению очередного ***к***-го числа на факториал от **(*к*-1)-**го числа. Последнее возвращение результата вычисления ф-ции ***Fakt*** присвоит переменной **F** значение произведения всех чисел от 1 до *N*, т.е. факториал числа **N**.

**program** rekurs;

**var**

N: integer;

F: longint;

**function** Fakt(N: Integer): longint;

**begin**

**if** N = 1 **Then** {Ветвь, определяющая конец рекурсии}

Fakt := 1 {Возвращаемое значение ф-ции}

**Else**

**{\*}** Fakt := N \* Fakt(N - 1) {Возвращаемое значение ф-ции}

**end**;

**BEGIN**

writeln('Введите N > 0');

read(N);

F := Fakt(N);

writeln('Значение факториала ', N, ' = ', F);

**END**.

Запись имени ф-ции **Fakt** в левой части оператора **:=** **{\*}** показывает, что это – возвращаемое значение ф-ции. Вызов ф-ции **Fakt** в правой части {\*} – обращении к этой же ф-ции, т.е. есть **рекурсивность.**

2) Нахождение **максимального** эл-нта в **глобальном** массиве *А.*

**procedure** Search\_Max(n: integer; **var** x: integer);

**begin**

**if** n = 1 **then** x:= A[1] // всего 1 элемент в массиве

**else**

**begin**

Search\_Max(n-1, x);

**if** A[n] > x **then** x:= A[n];

**end**;

**end**;

3) Вычисление **наибольшего общего делителя** (НОД) 2х чисел.

**function** NOD(a, b: integer): integer;

**begin**

**if** (a = 0) **or** (b = 0) **then** NOD:= a + b

**else**

**if** a > b **then** NOD:= NOD(a - b, b)

**else** NOD:= NOD(a, b - a);

# ТЕМА 10. ФОРМЫ ЗАПИСИ ВЫРАЖЕНИЙ

## 10.1 формы записи выражений:

**А+В** – **инфиксная**: знак операции – м/у операндами;

**+АВ**  – **префиксная (польская)**: знак операции - перед операндами;

**АВ+ – постфиксная** (обратная польская): знак операция - после операндов.

**Классический метод** - трансляция выражений на основе **промежуточной** **обратной** польской записи + использ-ются и другие методы. (в честь польского математика Яна Лукашевича).

Для преобразования выражений из инфиксной в постфиксную и префиксную формы - учитывать **правила** **приоритетности** операций: операции с высшим приоритетом преобразуются первыми, а после преобразования операция рассматривается как 1 операнд. Общепринятую приоритетность операций м изменить скобками. При просмотре строки без скобок, вычисления - слева направо для операций с **один**аковым приоритетом, кроме возведения в степень, когда вычисления - справа налево. Ниже - Пр-ры различных форм записи выражений.

**Инфиксное представление Постфиксное представление**

*А+В-С АВ+С-*

*(А+В)\*(С-D) AB+CD-\**

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

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

**Инфиксное представление Префиксное представление**

***А+В-С - +АВС***

***(А+В)\*(С-D) \*+AB-CD***

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

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

**Суть** **обратной** Польск Зап: нет скобок, операнды – в том же порядке, что в исходном выражении, а знаки операций при просмотре записи слева направо в порядке, в котором выполняются соотв-ие действия. Поэтому **основное преимущество** обратной Польск Зап (чем обычная запись выражений со скобками): **выражение м вычислить в процессе 1кратного просмотра слева направо.**

## 10.2 Построение выражений в обратной Польск Зап

Правило вычисления выражения в обратной Польск Зап: просматривается слева направо. Если очередной эл-нт – **операнд**, то рассматриваем следующий эл-нт. Если текущий эл-нт – **знак** операции, то выполняется эта операция над операндами, записанными левее знака операции. **Результат** операции записывают вместо первого (самого левого) операнда, участвовавшего в операции. Остальные эл-нты (операнды и знак операции) вычеркиваются из записи. Просмотр продолжается. В результате выполнения последовательно этого правила б выполнены все операции в выражении, и запись сократится до 1 эл-нта – результата вычисления выражения.

Для управления порядком выполнения операций над операндами – **скобочная** запись выражений. Сначала вычисляем ту часть выражения, у которой скобочный уровень вложенности – самый высокий. Если неск-ко операторов с наивысшим скобочным уровнем – они вычисляются слева направо. При вычислении выражений с операциями, которым присвоен приоритет, или частично скобочных выражениях инфиксной формы записи - повторно сканировать слева направо. Повторное сканирование м исключить, если инфиксное выражение преобразовать в постфиксную или префиксную форму записи (см. таблицу ниже).

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

**Правила** выполнения **постф**иксного выражения:

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

1. Выбрать 2 операнда, стоящих прямо **слева** от найденного оператора.
2. Выполнить операцию.
3. **Заменить** оператор и операнды результатом.
4. **Повторять** действия до обработки всех операндов.

Есть компиляторы, преобразующие **инфикс**ные выражения на языках высокого уровня в обратную Польск Зап.

Постфиксное и префиксное выражения корректны только тогда, когда **ранг выражения = 1**, а ранг любой правой головы польской формулы >= (<=) 1. **Ранг корректного выражения = 1.**

Алгебраическое преобразование инфиксного выражения в обратное польское основано на **приоритетах** операторов и предлагает использ-ть **стек**. 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 соотв-ет их **приоритетам**.

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

Пр-р преобразования инфиксного выражения

**a + b \* c – d / e \* h** в обратное польское выражение **a b c \* + d e / h \* –**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Сканируемый символ** | **Содержание стека** | Обратн Польск Зап |  | **Ранг** |
|  | **|–** |  |  |  |
| **А** | **|– a** |  |  |  |
| **+** | **|– +** | **A** | 1 |  |
| **B** | **|– +b** | **A** | 1 |  |
| **\*** | **|– +\*** | **Ab** | 2 |  |
| **С** | **|– +\*c** | **Ab** | 2 |  |
| **–** | **|– –** | **abc\*+** | 1 |  |
| **D** | **|– –d** | **abc\*+** | 1 |  |
| **/** | **|– – /** | **abc\*+d** | 2 |  |
| **E** | **|– – / e** | **abc\*+d** | 2 |  |
| **\*** | **|– – \*** | **abc\*+de/** | 2 |  |
| **H** | **|– – \* h** | **abc\*+de/** | 2 |  |
| **|–** | **|–** | **abc\*+de/h\*–** | 1 |  |

Алгоритм преобразования:

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

2) Значение приоритета очередного **входного** символа сравнивается с приоритетом верхнего эл-нта стека.

3) Если приоритет символа **>** приоритета верхнего эл-нта стека, то символ помещается в стек, выбирается следующий входной символ.

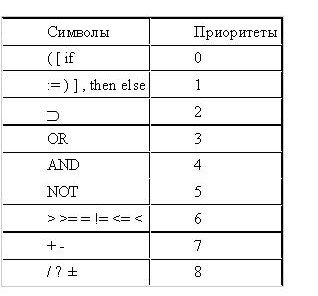
4) Если приоритет входного символа <= приоритету верхнего эл-нта стека, то этот эл-нт удаляется из стека и помещается в формируемую строку, после чего сравнивают приоритеты очередного символа и нового верхнего символа.

Каждый раз при изменении обратной Польск Зап модифицируется ранг результирующего выражения.

## 10.3 Преобразование скобочных выражений в обратную Польск Зап

Неск-ко методов получения обратной польской записи. 1 из наиболее эффективных: использование **стека с приоритетами**, изменяет порядок следования **знаков** операций в выражении так, что получается обратная польская запись. Применим этот м-д только к простым выражениям с переменными, знаками арифметических и логических операций, знаками операций отношения и круглые скобки.

Каждому символу выражения присвоим приоритет (таблица ниже). Для знаков операций приоритеты возрастают в порядке, **обратном** **старшинству операций**. Скобки – низший приоритет.



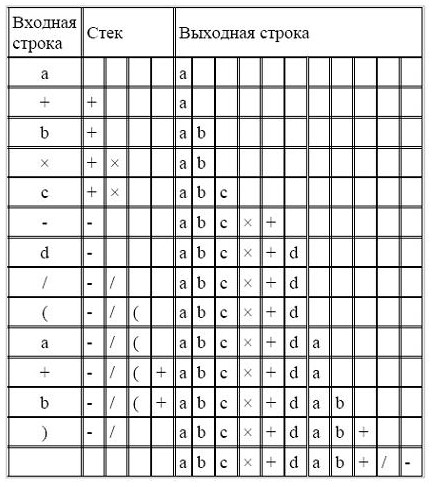
Арифметическое/ логическое выражение рассматриваем как входную строку символов. Входная строка просматривается слева направо. Операнды переписываются в выходную строку, а знаки операций помещаются сначала в стек операций.

Если приоритет входного знака = 0 или > приоритета знака в вершине стека, то новый знак добавляется к вершине стека. Иначе - из стека «выталкивается» и переписывается в выходную строку знак, находящийся в вершине, а также следующие за ним знаки с приоритетами >= приоритету входного знака. После этого входной знак добавляется к вершине стека.

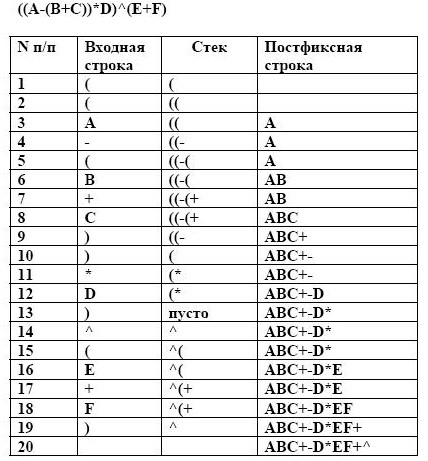
Особенности имеет лишь обработка **скобок**: открывающая круглая скобка (имеет приоритет 0) просто записывается в вершину стека и не выталкивает ни 1 знака. И ее не м вытолкнуть ни 1 знак, кроме **закрывающей** скобки – имеет приоритет 1, не превосходящий приоритета любой операции. Поэтому появление **закрывающей** скобки вызывает выталкивание всех знаков до ближайшей открывающей скобки включительно. В стек закрывающую скобку **не** записываем. Открывающая и закрывающая скобки как бы взаимно уничтожаются и в выходную строку **не** переносятся. После просмотра всех символов входной строки идет выталкивание всех оставшихся в стеке символов и дописывание их к выходной строке.

Ниже - 2 пр-ра преобразов-ия **инфикс**ных выражений в обратные польские

*Пр-р* 10.1. **a + b \* c – d / (a + b)** в обратн Польск Зап **a b c \* + d a b+/-**



*Пр-р* 10.2*.* **((a - (b + c)) \* d) ^ (e + f)** в обратн Польск Зап **a b c + - d \* e f + ^**



Преобразование инфиксного выражения в **преф**иксную запись - сканированием **инф**иксного выражения **справа** налево.

## 10.4 Вычисление выражений в виде обратной Польск Зап

**Правило** вычисления выражения в обратной Польск Зап: просматривать ее **слева** направо.

Если рассматриваемый эл-нт – операнд, то рассматриваем следующий эл-нт. Если рассматриваемый эл-нт — оператор, то эта операция выполняется над операндами, записанными **левее** знака операции. Результат операции записывается вместо 1-го (самого левого) операнда, участвовавшего в операции. Остальные эл-нты (операнды и оператор - участники операции) вычеркиваются из записи. Просмотр продолжается.

В результате последовательного выполнения этого правила б выполнены все операции выражения и запись сократится до 1 эл-нта — результата вычисления выражения.

Простое арифметическое выражение с вещественными переменными

**а + b × с - d / (а + b)** в обратной польской зап б: **а b c × + d a b + / -.**

Выполнение правила для этого Пр-ра приводит к последоват-ти строк – записаны во 2й графе таблицы. Рассматриваемый эл-нт строки на каждом шаге процесса отмечен квадратными скобками. В 3й графе таблицы - соотв-ие действия, а в 4й — эквивалентные команды трехадресной машины.

ВЫЧИСЛЕНИЕ ВЫРАЖЕНИЯ В ОБРАТНОЙ ПОЛЬСК ЗАП

|  |  |  |  |
| --- | --- | --- | --- |
|  | **Состояние строки** | Действие | **Машинная команда** |
|  | 2 | 3 | 4 |
| 1 | **[a] b c × + d a b + / -** | Просмотреть следующий эл-нт | — |
| 2 | **a [b] c × + d a b + / -** | Просмотреть следующий эл-нт | — |
| 3 | **a b [c] × + d a b + / -** | Просмотреть следующий эл-нт | — |
| 4 | **a b c [×] + d a b + / -** | *r*  1    :=    *b*    ×    *c* | **× *b* *c* *r*1** |
| 5 | **a r1 [+] d a b + / -** | *r*  1    :=    *a*    +    *r*  1 | **+ *a* *r*1 *r*1** |
| 6 | **r1 [d] a b + / -** | Просмотреть следующий эл-нт | — |
| 7 | **r1 d [a] b + / -** | Просмотреть следующий эл-нт | — |
| 8 | **r1 d a [b] + / -** | Просмотреть следующий эл-нт | — |
| 9 | **r1 d a b [+] / -** | *r*  2    :=    *a*    +    *b* | **+ *a* *b* *r*2** |
| 10 | **r1 d r2 [/] -** | *r*  2    :=    *d*    /    *r*  2 | **/ *d* *r*2 *r*2** |
| 11 | *r*  1    *r*  2    [  -  ] | *r*  1    :=    *r*  1    -    *r*  2 | **- *r*1 *r*2 *r*1** |
| 12 | *r*  1 | — | — |

Результат выполнения операции фиксируется в виде рабочей переменной вида **rj.** После очередной операции рабочая переменная ***r*1** или ***r*2** вычеркивается, освободившуюся рабочую переменную м использ-ть вновь для записи результата операции. Использ-ие каждый раз свободной рабочей переменной с минимальным номером экономит кол-во занятых рабочих переменных.

==============================================

**ТЕМА 11. ОПРЕДЕЛЕНИЕ И ПОСТРОЕНИЕ БИНАРНОГО ДЕР**

**(СТР 103 ОАИП 7.5.)**

# 11.1 Основные определения нелинейные структур

1 из важных признаков Стр-ы Дан - характер упорядоченности ее эл-тов и по этому признаку структуры делят на **линейные** и **нелинейные.** **Линейные**: векторы, строки, массивы, стеки, очереди, списки. **Нелинейные**: деревья и графы.

**Дер -** иерархическая стр-ра какой-то совокупности эл-тов. **Дер** – совокупность **эл-тов**, называемых узлами (1 из них определен как корень) и отношений, образующих иерархическую структуру узлов. Узлы дер - эл-нты любого типа, изображаются буквами, строками или числами.

Формально дер определяют: 1 узел - дер, он же - корень этого дер.

Пусть *n* – узел, а *T*1,*T*2...*Tk –* деревья с корнями *n*1,*n*2...*nk* соотв-но. М построить новое дер, сделав *n* родителем узлов *n*1,*n*2...*nk* . В этом дер *n* б корнем, а *T*1,*T*2...*Tk* – поддеревьями этого корня. Узлы ***n*1,*n*2...*nk*** называются **сыновьями узла** *n*.

это определение включают в себя понятие **нулевого дер** -

дер без узлов.

Способы изображения структуры дер: 1) вложенные множества; 2) вложенные скобки; 3) с помощью отступов; 4) графически.

Пр-р - оглавление книги представим схематически (рисунки 11.1, 11.2):

*КНИГА*

*ГЛАВА 1*

*РАЗДЕЛ 1.1*

*РАЗДЕЛ 1.2*

*ГЛАВА 2*

*РАЗДЕЛ 2.1*

*РАЗДЕЛ 2.1.1*

*РАЗДЕЛ 2.1.2*

*РАЗДЕЛ 2.2*

*ГЛАВА 3*

Рис 11.1 – 1й способ представления структуры в виде дер

*КНИГА*

*ГЛАВА 1 ГЛАВА 2 ГЛАВА 3*

*РАЗДЕЛ 1.1 РАЗДЕЛ 1.2 РАЗДЕЛ 2.1 РАЗДЕЛ 2.2 РАЗДЕЛ 2.3*

*РАЗДЕЛ 2.1.1 РАЗДЕЛ 2.1.2*

Рис 11.2 – 2й способ представления структуры в виде дер

Отношение родитель-сын отображается в виде линии. Дер рисуют сверху вниз, чтобы родители (предки) располагались выше детей (потомки).

Корень дер на рис. 1 - узел *Книга*, который имеет 3 поддер, соотв-но с корнями *Глава 1, Глава 2, Глава 3*. Узел *Книга* - родитель узлов *Глава1, Глава 2, Глава 3*, а эти узлы – сыновьями узла *Книга*.

Дер имеет 3 поддер. Первое из них (*Глава 1*) имеет 2 поддер: *Раздел 1.1* *и Раздел 1.2*. Поддер с корнем *Глава 2* имеет 3 сына: *Раздел 2.1, Раздел 2.2, Раздел 2.3. Узел Раздел 2.1* имеет 2 сына: *Раздел 2.1.1* и *Раздел 2.1.2.* Поддер с корнем *Глава 3*, состоит из 1 узла и не содержит поддеревьев. Узлы дер, не имеющие потомков, называются ***листьями***.

дер –наилучшая форма представления четко структурированных данных.

Путь из узла *n*1 в узел *nk* называется **последоват-ть** узлов *n*1,*n*2...*nk* , где для всех i , 1 i  k, узел *ni -* **родитель** узла *ni*+1.

Длиной пути называется **число узлов**, составляющих этот путь. Т о, путь нулевой длины - путь из любого узла к самому себе. (В Пр-ре на рис. 13 путем длины 2 б путь от узла *Глава 2* к узлу *Раздел 2.1.2*).

Если на дер есть путь из узла **a** в узел **b**, то узел а - **предок** узла b, а узел b – **потомок** узла а. Любой узел одновременно и предок, и потомок самого себя.

Предок или потомок узла, не являющийся самим этим узлом, называется **истинным предком** / **истинным потомком** данного узла. В дер только **корень** не имеет истинного предка. Узел, не имеющий истинных потомков, называется **листом**. Тогда **под-**дер дер – **узел** (**корень** поддер) вместе со всеми его потомками.

**Высота узла дер** - длина самого длинного пути из этого узла до какого либо листа.

Порядок узлов дер. Если имеет значение относительный порядок поддеревьев *T*1,*T*2...*Tk* в дер, то м говорить, что дер - упорядочено; если порядок узлов игнорируется, дер - неупорядоченное. Сыновья узла упорядочиваются **слева** направо – для сопоставления узлов, которые не связаны отношениями предки-потомки. Поэтому 2 дер на рис 11.3 различны, т.к. порядок сыновей узла *А* разный.



Рисунок 11.3 – 2 разных дер

Правило: если узлы *а* и *b* - сыновья 1-го родителя (с), и узел *а* лежит слева от узла *b*, то все потомки узла а б находиться слева от любых потомков узла *b*.

Правило для определения какие узлы расположены слева от данного узла *n*, а какие – справа: надо прочертить путь от **корня** дер до узла *n*. Тогда все узлы и их потомки, расположенные слева от этого пути, б находиться слева от узла *n*, и аналогично - справа (все узлы и их потомки, расположенные справа от этого пути).

# 11.2 Стр-ра и построение бинарного дер (СТР 103 ОАИП 7.5.2.)

В древовидной структуре число потомков вершины называется ее **степенью**. Максимальное значение этих степеней называется **степенью дер**. Наибольшую популярность в программировании и вычислительной технике получили *бинарные (двоичные) деревья*, у которых степень дер = 2*:* вершина дер м иметь не более 2х потомков, называемых ***левым* и *правым* сыновьями**. В отдельный подкласс *бинарных деревьев* выделены**деревья поиска**: характер-ются тем, что значение информационного поля, связанного с вершиной дер, больше любого соотв-его значения из левого поддер и меньше содержимого любого узла его правого поддер.

Схематично двоичное дер - набор вершин, соединенных стрелками (ветвями, рисунок 11.4). Из каждой вершины выходит не более 2 ветвей, направленных влево-вниз / вправо-вниз. В каждую вершину, помимо одной, входит 1 стрелка. Вершина, в которую не входит ни 1 стрелка - **корень дер**. Вершины, из которых не выходит ни 1 стрелка – **листья**.

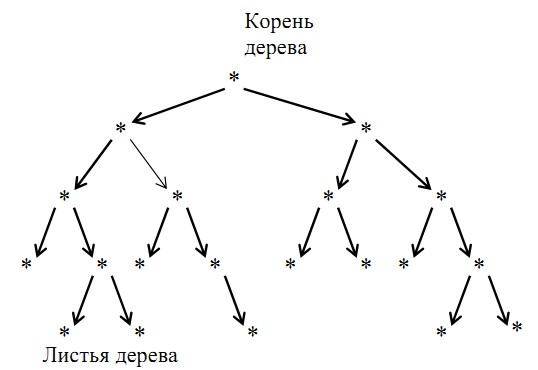


Рис 11.4 – Схематическое представление двоичного дер

Каждая вершина дер (звено) – **записью**, состоящей из 4 полей: 1) ключ записи (*Kl*), 2) ссылка на вершину влево-вниз (*Lev*), 3) ссылка на вершину вправо-вниз (*Prav*), 4) ссылка на текст записи (\*). Ее стр-ра - рис 11.5.

Принцип построения дер: 1ая запись - корень дер. Если ключ следующей записи меньше ключа корня, то эта запись ставится слева вершина, иначе – справа. Ключ *К* каждой последующей записи сравнивается последовательно с ключом корня, а затем с ключами записей на соотв-их ветвях дер. И в зав-ти от результата сравнения ключа *К* идет переход влево или вправо, пока не б найдена вершина, к которой м присоединить новую вершину с ключом *К*.

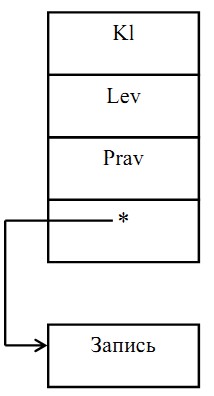


Рисунок 11.5 – Стр-ра звена бинарного дер

*Пр-р 11.1.*построить бин-ое дер из последовательно поступающих записей с ключами 100, 20, 120, 50, 15, 130, 55, 30, 35, 60, 33, 28.

В соответствии с вышеприведенным принципом построения дер имеет вид (рисунок 11.6).

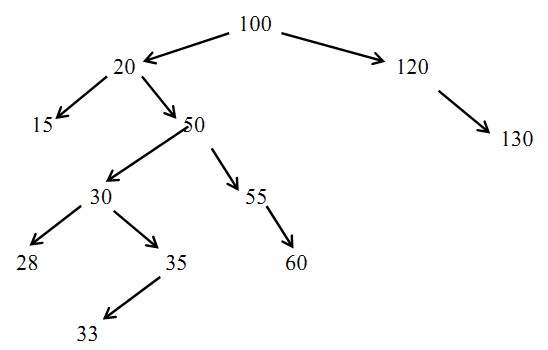


Рисунок 11.6 – Результирующее дер

С учетом структуры каждого звена дер последоват-ть построения дер по данному Пр-ру выглядит так: после поступления первой записи с ключом **К1 = 100** дер имеет вид (рисунок 11.7):

 Рис 11.7 – Вид дер после поступления 1й записи

Т к у этой вершины пока нет вершин справа-внизу и слева-внизу, то в соотв-ие поля (ссылки на левые и правые вершины) - пустая ссылка (*Nil*). После поступления второй записи с ключом *К2 = 20:*

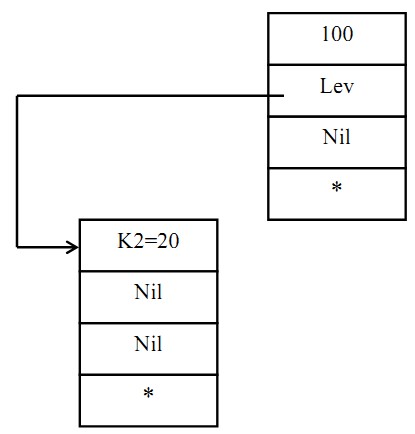


Рисунок 11.7 – Вид дер после поступления второй записи

Т к К2 < К1, то вновь созданная вершина дер становится левой по отношению к первой. Для этого в поле *Lev* (адрес левой вершины) вершины 1 заносится адрес созданной вершины 2. После поступления третьей записи с ключом *К3 = 120* дер приобретает вид (рисунок 11.8).

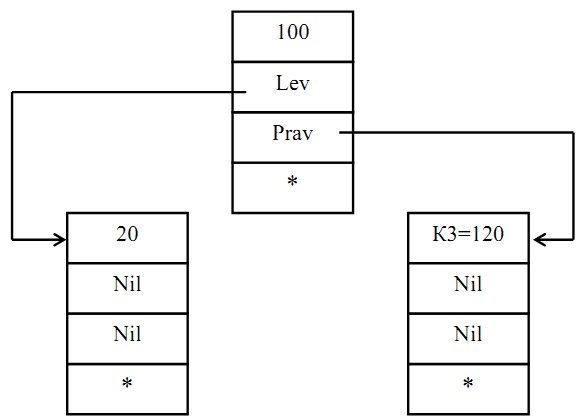


Рисунок 11.8 – Вид дер после поступления третьей записи

*К3 > К1* - вновь созданная вершина дер становится правой по отношению к первой вершине: в поле *Prav* вершины 1 заносится адрес вершины 3. Аналогично для подключения всех последующих вершин.

Т о, подходящей вершиной, к которой м подсоединить новую вершину - вершина соответствующей ветви дер, у которой поле *Lev* или *Prav* (в зав-ти от соотношения ключей ее и новой вершины) = *Nil*.

Звено 2чного дер м представить с помощью описания в Пр-ре 11.2.

*Пр-р 11.2.* Описание двоичного дер.

**Type**

Tekst = String; //<Тип\_значения\_записи>;

AdrТ = ^Tekst;

**AdrZv** = ^Zveno;

Zveno = **Record**

Kl: Integer; //1

Lev, Prav: AdrZv; //2, 3

Adr: AdrT; //4

**End**;

**Var**

DvDer: **AdrZv**;

ТЕМА 12. ОПЕРАЦИИ НАД БИНАРНЫМИ ДЕРЕВЬЯМИ

Для двоичного дер те же **операции,** как д/таблиц (3шт):

1. **поиск** записи с заданным ключом;
2. **включение** записи с заданным ключом (если уже есть запись с таким ключом, то старая запись меняется на новую);
3. **исключение** записи с заданным ключом.

# 12.1 Поиск записи в дер (СТР 105 ОАИП 7.5.3.)

Пусть имеется объявление по Пр-ру 11.2 (см. лекцию 11).

*Пр-р 12.1.*  Логич ф-ция ищет **вершину** дер с задан ключом. Парам-ры:

**К** - заданный ключ,

**D** - ссылка на **корень** дер (где б поиск)

**Rez** - переменная, которой := ссылка на найденное звено (если она есть) или ссылка на вершину, после обработки которой поиск прекращен (если её нет (закончилась ветвь)).

……………………………………………………….

**Function** Poisk(K: Integer; **Var** D, Rez: **AdrZv**): Boolean;

**Var**

P, Q: **AdrZv**; //Q – адрес обработанной вершины, Р – адрес корня дер

B: Boolean; {В – признак что ключ найден! Poisk:= B;}

**Begin**

B := False;

P := D; {В Р заносят адрес корня дер. Потом в Р б храниться

адрес вершины, подлежащей обработке}

Q := Nil; {Q – адрес обработанной вершины}

**If** D <> Nil **Then** {если дер не пустое}

**Repeat // D** - ссылка на **корень** дер

Q := P; {Q – адрес обработанной вершины}

**If** P^.Kl = K **Then** {в корне Р конкретное звено}//сразу нашли её

B:= True {Найдена! нужная вершина}

**Else** //смотрим: < или > искомая вершина корня дер

**If** K < P^.Kl **Then** {К - заданный ключ}//вершина меньше корня

P := P^.**Lev**

**Else**

P := P^.**Prav**

**Until** B **Or** (P = Nil); {Поиск, пока не найден ключ (В=True) или

пока не закончилась соответств-ая ветвь}

Poisk := B; {Возвращаемое значение}

Rez := Q; {Q – адрес звена с нужным ключом / адрес конца ветви}

**End**;

Скорость поиска в двоичном дер = скорости дихотомического поиска (см. Пр-р 7.20).

**12. 2 Включение записи в дер (СТР 106 ОАИП 7.5.4.)**

- надо найти вершину, куда м присоединить новую вершину, соответствующую включаемой записи. Алгоритм поиска **вершины** = алгоритму поиска вершины с заданным ключом (см. Пр-р 12.1). Нужная вершина найдена, если в качестве очередной ссылки, определяющей ветвь продолжения поиска, окажется ссылка *Nil*. В качестве заданного ключа в данном случае использ-ется ключ включаемой вершины.

Т о, для поиска вершины, к которой м присоединить включаемую запись, м воспользоваться алгоритмом поиска вершины с заданным ключом, реализованным в Пр-ре 12.1. Такая вершина найдена, если *В = False*. В этом случае в формальном параметре *Rez* процедуры *Poisk* находится адрес вершины, к которой м подсоединить включаемую вершину.

Для простоты будем считать, что в дер нет записи с тем же ключом, что и у включаемой записи. Пусть имеется объявление по Пр-ру 11.2.

*Пр-р 12.2.* Процедура **включения** записи в дер. Параметры процед: ***К*** – ключ, ***D*** – адрес корня Дер, Zap – текст вставляемой записи.

**Type**

Tekst = String; //<Тип\_значения\_записи>;

AdrТ = ^Tekst;

**AdrZv** = ^Zveno;

Zveno = **Record**

Kl: Integer; //1

Lev, Prav: AdrZv; //2, 3

Adr: AdrT; //4

**End**;

**Var**

DvDer: **AdrZv**;

**Procedure** Vkl(K: Integer; **Var** D: AdrZv; **Zap**: Tekst);

**Var**

Q, S: AdrZv; //S – вершина, Q-адрес вершины, к j присоед нов верш

T: AdrТ;

**Begin**

**If Not** Poisk(K, D, Q) **Then** // Poisk – процед-ра из прим 7.23

**Begin**

**New** (T); {Создано звено д/занесения текста записи}

T^ := **Zap**; {Занесен в таблицу текст записи}

**New** (**S**); {Создана просто новая вершина в дер}

S^.Kl := K; {Занесен ключ К в поле Kl новой вершины}

S^.Adr := T; {Занесен адрес текста записи в поле Adr нов верш}

S^.Lev := Nil;

S^.Prav := Nil; {Созданную вершину сделали листом дер}

**If** D = Nil **Then** {if дер еще пустоe, то созданное звено– корень дер}

D := **S**

**Else**

**If** K < Q^.Kl **Then** {В Q-адрес вершины, к которой

присоединяется новая вершина}

Q^.Lev := **S**

**Else**

Q^.Prav := **S**

**End**

**End**;

# 12.3 Удаление записи из дер

Если некая вершина - конечная (“лист”) или из нее выходит только 1 ветвь, то для удаления записи у вершины-предшественника надо скорректировать ссылку (рисунок 12.1).

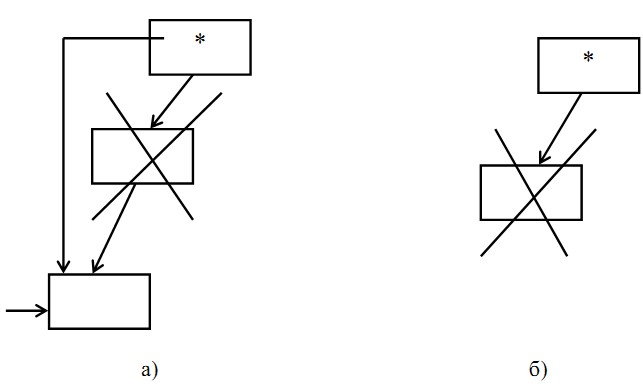


Рисунок 12.1 – Удаление записи из дер. 2 вар-та:а) из удаляемой записи выходит 1 ветвь и **б)** удаляемая запись – «**лист**» Дер

В а) и б) в соответствующее поле **Lev /** **Prav** вершины-предшественника надо занести содержимое **Lev /** **Prav** **удаляемой** вершины. If удаляем “лист” (б) – б ссылка **Nil***.*

в) у удаляемой вершины – 2 ветви: надо найти звено, которое м вставить на место удаляемого. Это м б 2 вар-а:

**1) Самый правый эл-нт левого поддер (**самый большой в левом поддер от удаляемой вершины). Д/этого надо перейти в следующую вершину по левой ветви от удаляемой, потом –по правой пока очередная правая ссылка не б = **Nil.**

*Пр-р 12.3.* Исключение из дер, созданного в Пр-ре 11.1, звена с ключом 50 (см. рисунок 11.6), используя для замещения самый правый эл-нт левого поддер. Результат исключения имеет вид, который представляет рисунок 12.2.

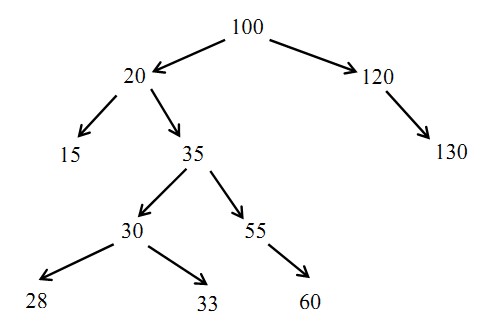


Рисунок 12.2 – Результат исключения звена с ключом 50 (вариант 1)

2) **самый левый эл-нт правого поддер (**самый маленький в правом от удаляемой вершины поддер): надо перейти в следующую от удаляемой вершину по правой ветви, потом по левой ветви пока очередная левая ссылка не б = ***Nil*** (т е наоборот (1)).

Исключение из дер, созданного в Пр-ре 11.1, звена с ключом 50 (см. рисунок 11.6), используя для замещения самый левый эл-нт правого поддер. Результат исключения представляет рисунок 12.3.



Рисунок 12.3 – Результат исключения звена с ключом 50 (вариант 2)

Т о, при исключении из двоичного дер вершины с заданным ключом надо учесть *3 случая удаления*:

1. звена с заданным ключом в дер **нет**;
2. звено с заданным ключом имеет не более 1 ветви;
3. звено с заданным ключом имеет 2 ветви.

*Пр-р 12.4* **Рекурсивная** процедура **Udder**исключения вершины с заданным ключом из дер. Процедура реализует вариант **1)** исключения в соответствии с Пр-ром 7.25: для замещения – **самый правый** эл-нт левого поддер (1). Параметры: ***K*** – искомый ключ, ***D*** – адрес **корня** дер. Объявление – по Пр-ру 11.2.

**Type**

Tekst = String; //<Тип\_значения\_записи>;

AdrТ = ^Tekst;

**AdrZv** = ^Zveno;

Zveno = **Record**

Kl: Integer; //1

Lev, Prav: AdrZv; //2, 3

Adr: AdrT; //4

**End**;

//--------- процедура Udder------------------------

**Procedure** Udder (**Var** D: **AdrZv**; K: Integer); //D – **адрес** корня дер

**Var**

Q: **AdrZv**; // в Q – адрес удаляемого звена

//----- процедура Ud - переход в **левое** д/удаляемой вершины поддер ------

**Procedure** Ud (**Var** R: **AdrZv**); {При первом вызове за фактич передается

**адрес левой** вершины после удаляемой}

**Begin**

**If** R^.Prav = Nil **Then** {Анализир-ая верш-а - самая правая в лев поддер 35}

**Begin**

{6} Q^.Kl := R^.Kl; {Занесен в поле Кl удаляемого звена ключ из

замещающего звена (в Q – **адрес** удаляемого звена)}

{7} Q^.Adr := R^.Adr; {Занесение в поле Adr удаляемого звена ссылки на

**текст** записи из замещающего звена}

{8} Q := R; {Занесение в Q адреса замещающего звена}

{9} R := Q^.Lev {Занесен в поле Prav звена- предшественника

замещающему содержимого поля Lev из замещающего звена}

**End**;

**Else** {Анализир-ая верш – не самая правая в левом поддер – *вар 2)?*}

{11} Ud (R^.Prav) {Рекурсивный вызов проц-ры Ud д/перехода к следующ

**правой** вершине **левого** поддер}

**End**; //==Конец процедуры Ud ====

**Begin** {========= Тело процедуры Udder ==========}

**If** D = Nil **Then** {Первый случай удаления}//D – **адрес** корня дер

Writeln ('Нет звена с заданным ключом в дер')

**Else // *K*** – искомый ключ

**If K** < D^.Kl **Then** {Ключ в анализир-ой вершине > заданного}

{2} Udder (D^.**Lev**, K) {Рекурсивн вызов проц-ры Udder д/перехода к

вершине **слева**-внизу}

**Else** {Ключ в анализируемой вершине <= заданному}

**If** K > D^.Kl **Then** {Ключ в анализир-ой вершине < заданного}

{3} Udder (D^.**Prav**, K) {Рекурсивн вызов проц-ры Udder д/перехода к

вершине **справа**-внизу}

**Else** {Звено с заданным ключом найдено!}

**Begin** // в Q – адрес удаляемого звена

{1} Q := D; {Занесен в Q адреса удаляемого звена (он взят

из поля Lev или Prav звена-предшественника)}

**If** Q^.Prav = Nil **Then** {2й случай удаления}

{4} D := Q^.Lev {Занесен в поле Lev/ Prav звена-предшеств-ка ***20***

удаляемому ссылки на зв, следующее за удаляемым ***55***}

**Else**

**If** Q^.Lev = Nil **Then** {2й случай удаления – *справа-налево?*}

{5} D:= Q^.Prav {Занесен в поле Lev/ Prav звена-предшест-ка ***20***

удаляемому ссылки на зв, следующее за удаляемым ***55***}

{10}**Else** {3й случай удаления}

Ud(Q^.Lev) {Вызов процедуры **Ud**; переход в **левое** д/удаляемой

вершины поддер}

**End**;

**End**; //=== Конец процедуры Udder ====

параметр *D* оформлен как параметр-переменная процедуры *Udder*. При рекурсивном вызове процедуры *Udder* в качестве соотв-его *D* фактического параметра использ-ется поле *Lev* (см. оператор {2}) или *Prav* (см. оператор {3}) того звена, от которого мы перешли к текущему звену (то есть звена-предшественника). Поэтому операторы {4} и {5} обеспечивают занесение в поле *Lev* или *Prav* звена-предшественника ссылки на звено, следующее за удаляемым, из удаляемого звена.

Аналогичные пояснения справедливы и для переменной *R* (*R* оформлена как параметр-переменная процедуры *Ud*).

Процедура *Ud* предназначена для поиска подходящей замещающей вершины, для занесения значений из полей замещающего звена в поля удаляемого звена и для удаления замещающего звена со своего предыдущего места.

Рисунок 12.4 содержит схематические пояснения к Пр-ру 12.4 (в привязке к Пр-рам 11.1 и 12.3). Удаляется звено с ключом 50. Замещающее звено – звено с ключом 35.

Номера в фигурных скобках соответствуют номерам операторов в Пр-ре 12.4.

Исходное значение *D* – адрес корня дер. По оператору {2} осуществляется переход к вершине 20, по оператору {3} – к вершине 50, по оператору {10} – к вершине 30, по оператору {11} – к вершине 35.

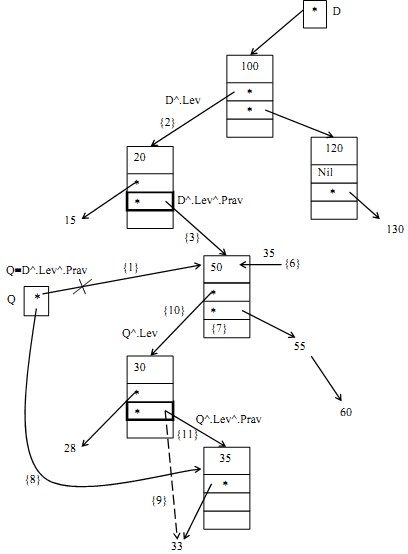
*D^.Lev, D^.Lev^.Prav* соответствуют значению фактического параметра, передаваемому на вход формального параметра *D* на текущем шаге выполнения проги; *Q^.Lev, Q^.Lev^.Prav* – значению фактического параметра, передаваемому на вход формального параметра *R* на текущем шаге выполнения проги. 

Рисунок 12.4 – Схематические пояснения к Пр-ру 12.4

Выделенные поля звеньев дер (см. рисунок 12.4) представляют собой значения фактического параметра (*D^.Lev^.Prav*), соотв-его формальному параметру *D* для удаляемого звена, и фактического параметра (*Q^.Lev^.Prav*), соотв-его формальному параметру *R* для замещающего звена.

Для физического удаления удаляемого звена надо вместо оператора {4} в процедуре *Udder* использ-ть составной оператор

**Begin**

**D := Q^.Lev;**

**Dispose (Q)**

**End;**

а вместо оператора {5} (**D := Q^.Prav)** – оператор

**Begin**

**D := Q^.Prav;**

**Dispose(Q)**

**End;**

Для физического удаления замещающего звена после оператора {9} в процедуре нужно добавить оператор **Dispose (Q*).***

**ТЕМА 13. ОБХОДЫ ДЕР. ПОМЕЧЕННЫЕ ДЕРЕВЬЯ**

# 13.1 Обходы дер

неск-ко способов обхода (прохождения) всех узлов дер. Популярны 3: 1) прямой (**сверху** вниз), 2) обратный (**снизу** вверх), 3) **слева** направо (симметричный).

Все 3 способа обхода дер м определить рекурсивно так:

Если дер *Т* - нулевое, то в список обхода заносим пустую запись.

Если дер *Т* - из 1 узла, то в список обхода записываем этот узел.

Далее, пусть *Т* – дер с корнем **n** и поддеревьями **T1,T2,...Tk**, тогда для разных способов обхода имеем:

а) **в прямом порядке** (т.е. при прямом упорядочивании) узлов дер *Т* сначала посещаем корень ***n***, а затем узлы поддер Т1, далее все узлы поддер Т2 и т. д. до узлов поддер T*k*.

б) **симметричный** обход узлов дер Т посещаем в симметричном порядке: а) все узлы поддер *T*1, б) корень *n*, в) последовательно в симметричном порядке все узлы поддеревьев Т**2**,...Т*k* . (**1, n, 2, 3, …, k**)

в) в **обратном** порядке: а) в обратном порядке все узлы поддер Т1, б) последовательно все узлы поддеревьев *T*2,...*Tk* , в) последним – корень *n*.

Для бинарного дер (рис. 13.1) рассмотрим **рекурсивные** процедуры, реализующие 3 способа обхода дер.

После каждой процедуры – последоват-ть узлов дер, выделенная жирным шрифтом, соотв-ая указанному способу обхода. Подчеркиванием помечены узлы, в которые возвращается процедура в ходе рекурсии.

Корень дер отличается от других узлов – имеет **нулевой** указатель или указатель на самого себя как на родителя.



Рис 13.1 – Дер обхода

***Прямой обход дер:***

**Procedure** prym\_print(**var** x: pt);

**Begin**

**if** x<> nil **then**

**Begin**

write (x^.data);

prym\_print(x^.left);

prym\_print(x^.right);

**End**;

**End**;

Последоват-ть обработки: **1, 2, 3,** 2, **4**, 2, 1, **5, 6,** 5, **7,** 5, 1

***Симметричный обход дер:***

**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



***Обратный обход дер:***

**Procedure** obr\_print(**var** x: pt);

**Begin**

**if** x<> nil **then**

**Begin**

obr\_print(x^.left);

obr\_print(x^.right);

write (x^.data);

**End**;

**End**;

Последоват-ть обработки: 1, 2, **3,** 2**,** **4**, **2**, 1, 5, **6,** 5, **7, 5, 1**

# 13.2 Построение бинарного дер на основе произвольного дер

Иерархические древовидные структуры разные по признаку - **степень вершин**. для прикладных задач – бинарные деревья. Любое дер м представить в виде эквивалентного бинарного. Алгоритм преобразования – на Пр-ре такого дер:

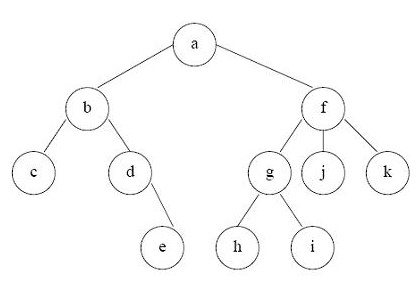


Рис 13.2 – **Произвольное** дер, которое надо преобразовать в БиДер

Порядок преобразования состоит из 2-х этапов.

1. Для каждой вершины уничтожаем все исходящие из нее **дуги**, кроме самой левой. Вместо них рисуем дуги, соединяющие выделенную вершину с вершиной, расположенной справа от нее на том же **уровне**.

2. Для каждой вершины – правило выбора левого и правого сыновей:

− левый сын – вершина непоср-но ниже данной вершины,

− правый сын – вершина непоср-но **справа** от данной на **1ом** уровне.

Рис 13.3 – 1й этап преобразования дер, а на рис 13.4 – заключительный.

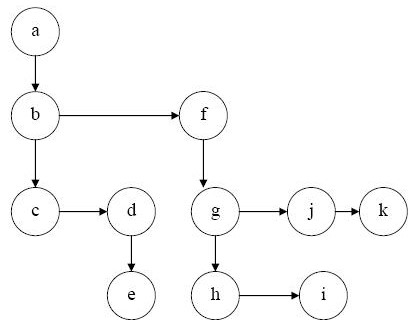


Рис 13.3 – Дер после первого этапа преобразования (*типа: держим за лев руку что было и даем ей что держали за правую «на, подержи». А за правую хватаем соседа слева на том же уровне что сам «Иди сюда»)*



Рис 13.4 – бин-ое дер, построенное на основе исходного дер

Введем неск-ко определений бинарных деревьев:

Если каждый **узел** бинар-го дер - **не лист** и имеет **непустые левое и правое** поддеревья, то дер - **строго бин-ое дер**.

***Св-во 1.***Строго бин-ое дер с ***n*** **листьями** всегда содержит ***2n-*1 узлов**. *(Напр-р: 4 листа, то узлов – 7??)*

**Полное бин-ое дер уровня *n*** – дер, в котором каждый узел уровня **меньше** ***n*** *(т е выше??)* имеет **непустое правое и левое** поддеревья, и каждый узел уровня ***n***является **листом**.

**Почти полное бин-ое дер** – дер, для кот-го есть неотрицательное целое ***k***:

1. Каждый **лист** в дер имеет уровень ***k*** или ***k+1***.
2. Если узел дер имеет **правого** потомка уровня ***k+1***, тогда все его **левые** потомки, являющиеся **листьями**, тоже имеют уровень **k+1**.

# 13.3 Помеченные деревья и деревья выражения

при работе с древовидными стр-рами бывает полезно каждому **узлу** дер сопоставить ***метку /*** ***значение***, аналогично тому, как с эл-нтами списков связывают определенные значения.

**Помеченное дер –** у которого узлам приписаны **метки**. Метка узла – значение (а не имя), которое «хранится» в узле. Аналогия: дер – список, узел – позиция, метка – эл-нт.

Рассмотрим дер с метками (рис 13.5), представляющее арифметическое выр-ие: **(а + b) \* (а + с)**,где **n1, …, n7** – **имена** узлов (**метки** проставлены рядом с соответств-ми узлами). Правила соответствия **меток** дер **эл-нтам** выражений:

1. **Метка** каждого листа соотв-ет **операнду** и содержит его **значение**, Напр-р, узел ***n4*** представляет операнд ***а***.

2. **Метка** каждого **внутреннего узла (родительского**) соотв-ет оператору.

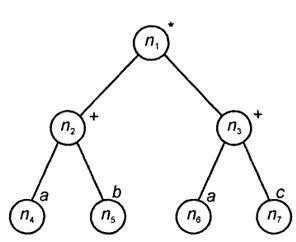


Рис 13.5 – Помеченное дер выражения

Пусть узел ***n*** помечен бинарным оператором *,* (+ или \*) и левый сын этого узла соотв-ет выражению (операнду) ***Е1*,** а правый – выражению ***Е2***. Тогда узел ***n***  и его сыновья представляют выражение

***(Е1)**(Е2)*.**

Напр-р, узел ***n2*** имеет оператор **+**, а левый и правый сыновья – **выражения (операнды)** ***а*** и ***b***, соотв-но. Поэтому узел ***n2*** – выражение ***(а)+(b)***или ***а + b***. Узел ***n1*** – выражение ***(а + b) \* (а + с)***, т к оператор **\*** - метка узла ***n1***, выражения ***а + b*** и ***а + с***представляются узлами ***n2*** и ***n3***, соответств-но.

При обходе деревьев составляется **список** не имен (**n**), а их меток (a, \*). В дер выражений при прямом упорядочивании получаем ***префиксную форму*** выражений: оператор (+-\*) предшествует и левому и правому операндам (a, b).

Для точного описания **пре**фиксной формы выражений сначала положим, что **преф**иксное выражение одиночного операнда ***а*** – сам этот операнд ***а***. Далее, **преф**иксная форма для выражений *(Е1)**(Е2)* (– бинарный оператор) имеет вид ***Р1Р2***, здесь *Р1* и *Р2* – префиксные формы для выражений ***Е1*** и ***Е2***. Note: в **преф**иксных формах не надо отделять или выделять префиксные выражения скобками – всегда м просмотреть префиксное выражение ***Р1Р2*** и определить (!) образом ***Р1*** как самый короткий префикс выражения ***Р1Р2***.

Напр-р, (рис 13.5) **прямое** упорядочивание узлов (точнее, меток) дер дает **преф**иксное выражение ***\* + аb + ас***. Самый короткий корректный префикс для выражения ***+аb+ас*** б префиксное выражение узла ***n2***: ***+аb***. (***(а + b) \* (а + с)***)

**Обратное** упорядочивание меток дер выражений дает **постф**иксное представление выражений. Здесь выражение *Р1Р2* имеет вид ***Р1Р2***, где *Р1* и *Р2* – **постф**иксные формы для выражений ***Е1* и *Е2*** соотв-но. В **пост**фиксной форме также **не** надо скобки: для любого выражения ***Р1Р2*** м проследить самый короткий суффикс **Р2**, что и б корректным составляющим постфиксным выражением. Напр-р**, постфиксная** форма выражения для рассматриваемого дер имеет вид ***а,b+ас+\**.** Если записать выражение как ***Р1Р2\****, то ***Р2*** (т.е. выражение ***ас+*)** б **самым коротким суффиксом** для ***аb + ас+***  и, следовательно, корректным составляющим постфиксным выражением.

При симметричном обходе дер выражений получим инфиксную форму выражения, которая совпадает с привычной «стандартной» формой записи, но также не использ-ет скобок. Для дер на рис. 3 инфиксное выражение запишется как **а+b \* а+с**. **Скобки** в **инф**иксное выражение добавляются с помощью отдельного алгоритма.

Алгоритм построения **помеченного** дер и вычисления выражений по нему*:*

1. Предварительная обработка выражения (контроль правильности расстановки скобок в выражении, удаление пробелов, замена унарного минуса на (0-1)\*);
2. Построение дер для данного выражения. Поиск в выражении арифметической операции с минимальным приоритетом с пропуском вложенных скобок. Разделение выражения на 2 части относительно найденной операции с минимальным приоритетом и **рекурсивное** повторение процесса.
3. Вычисление выражения по построенному дер.

На рис 13.6 - Пр-р построения **помеченного дер** по выражению. Выполняя заданный обход помеченного дер, м вычислить значение исходного выр-ия.

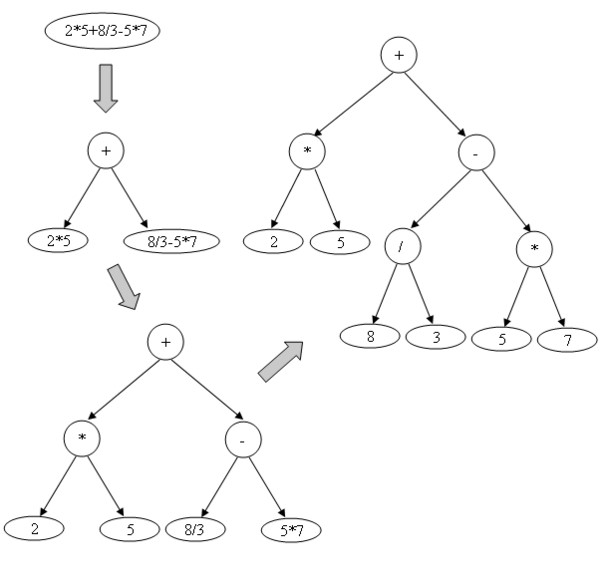
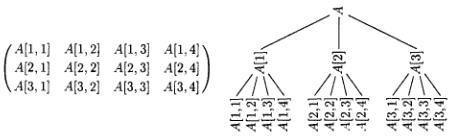


Рис 13.6 – Пр-р построения **помеченного** дер – ЛЕГ КО!!!

# 13.4 Альтернативные способы реализации деревьев

Пусть дано дер *Т* с узлами **1,2,3,…*n*.** 1 из простых представлений дер *Т* - линейный массив **А**, где каждый эл-нт **А[i]** - указатель на родителя узла **i**. **Корень** дер *Т* отличается от других узлов тем, что имеет 0й указатель или указатель на самого себя (как на родителя). Это представление использ-ет то св-во дер, что каждый узел, отличный от корня, имеет только 1го родителя. Используя это, м найти **родителя** любого узла за опред-ое вр. Прохождение по любому пути, т.е. переход по узлам от родителя к родителю м б за вр, пропорциональное кол-ву узлов пути. Так любой прямоугольный массив м представить как частный случай древовидной структуры. Напр-р, 2 представления матрицы размера 3x4.:



Note: это дер не воспроизводит адекватно **всю** структуру матрицы: связь в **строках** представлена в дер **явно**, а в **столбцах** – **нет**.

Еще 1 способ представить дер: формировать для каждого **узла** список его **сыновей**. Т к число сыновей у разных узлов м б разное, применяют **связанные списки**. Для дер на рисунке 13.7 список сыновей – на рис 13.8.

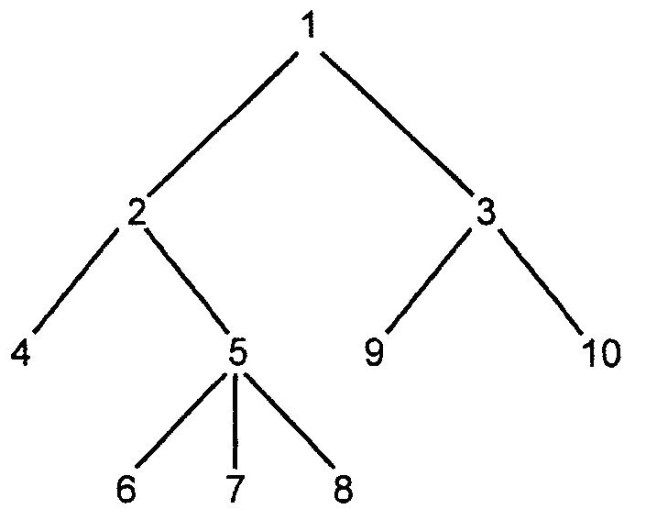


Рис 13.7 – Исходное дер

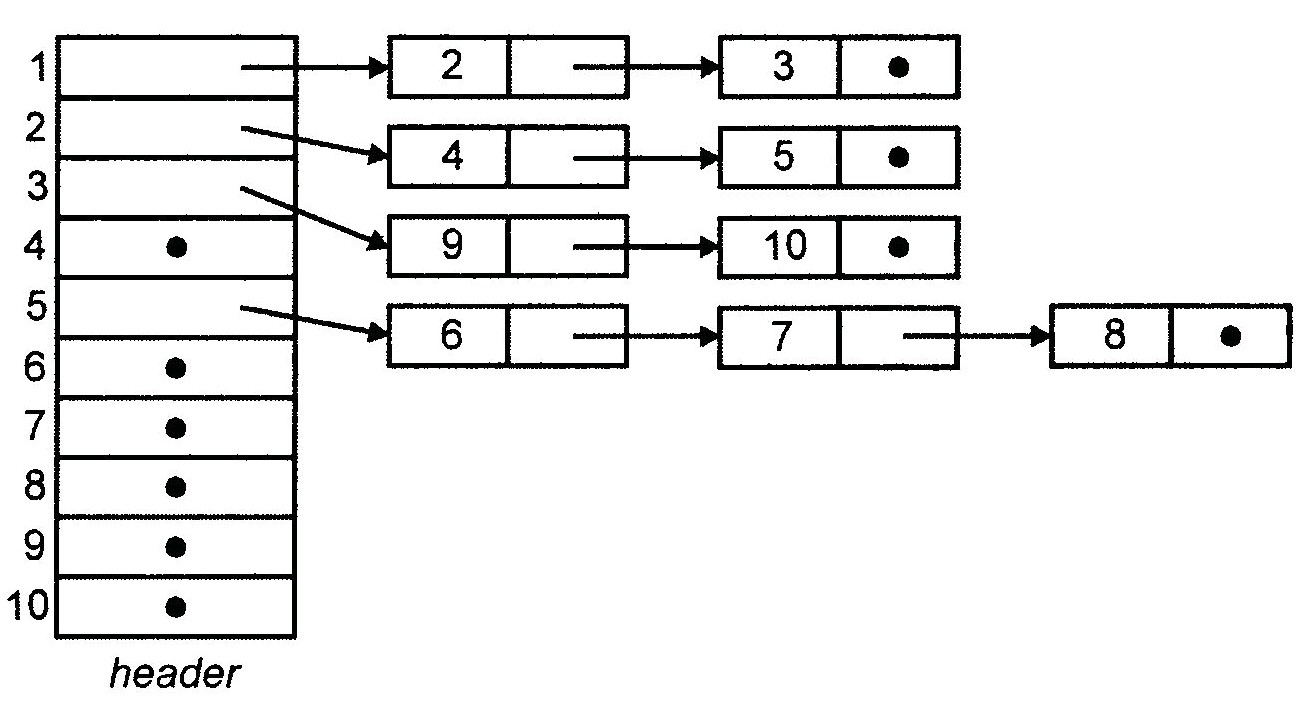


Рис 13.8 – Представление дер с помощью **связанных списков**

Здесь есть массив ячеек заголовков, индексированный №№ узлов. Каждый **заголовок** указывает на связанный список, состоящий из эл-тов узлов. Эл-нты списка **header**[i*]* - сыновья узла **i***,* Напр-р, узлы 9 и 10 – сыновья узла 3.

**ТЕМА 14. ПРОШИТЫЕ БИНАРНЫЕ ДЕРЕВЬЯ**

# 14.1 Представление списков в виде бинарных деревьев

Рассмотрим задачу нахождения ***к***-го эл-нта списка, используя дер. Значение в **узле** дер – **кол-во эл-тов** в **левом** поддер данного узла.

**Эл-нты** исходного **списка** - **листья** дер, а **узлы** - нелистья - часть внутренней структуры дер. С каждым узлом связан счетчик числа листьев в левом поддер. Эл-нты списка присваиваются листьям дер в порядке симметричного просмотра. В каждом узле ***р*** (не является листом) алгоритм определяет по значениям ***r*** и ***ListCount(p)*,** находится ли нужный лист в левом / правом поддер. Если лист в **левом** поддер, то поиск ведется по этому поддер без изменения значения ***r***, а если в правом – перед поиском по этому поддер значение ***r*** **уменьшается** на ***ListCount(p)***. Рассмотрим этот способ на Пр-ре. Пусть дан список эл-тов (рис 14.1). На рис 14.2, 14.3 2 дер, удовлетворяющие требованиям алгоритма.

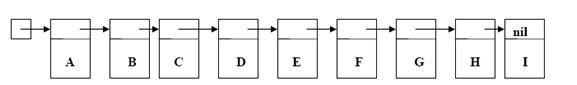


Рисунок 14.1 – Список эл-тов для поиска

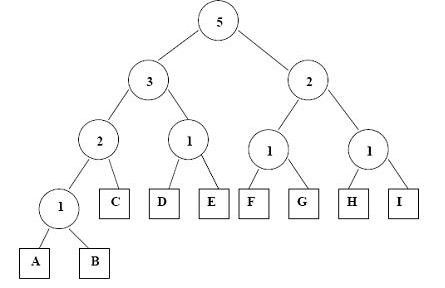


Рисунок 14.2 – 1й вариант дер с эл-нтами списка

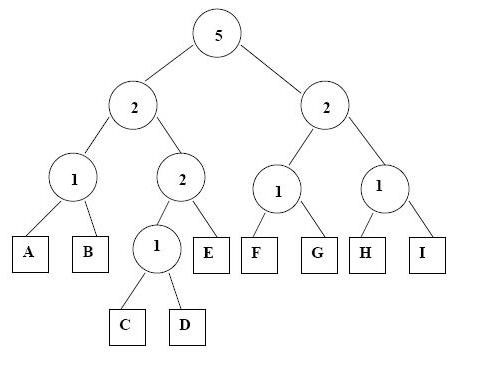


Рисунок 14.3 – 2й вариант дер с эл-нтами списка

Код алгоритма нахождения *к*-ого эл-нта списка:

**r := k; // *r*** – рабочая переменная

**p:= Tree; // Tree** – **указатель** на корень дер

**while p не является листом do**

**if r <= ListCounnt(p) then // *ListCount(p)*** – **счетчик** числа листьев

**p:= Left(p) //*Left(p)*,  *Right(p)*** – указат на лев, прав сын текущ узла

**else**

**begin**

**r:= r - ListCount(p);**

**p:= Right(p);**

**end; //** **р** – рабочий указатель

**Find:= p; // *Find***– **указатель** найденного эл-нта;

Здесь **Tree** – **указатель** на корень дер;

***ListCount(p)*** – **счетчик** числа листьев, связанных с узлом, на который указывает рабочий указатель **р***;*

***r*** – рабочая переменная; ***Find***– **указатель** найденного эл-нта;

***Left(p)*** и ***Right(p)*** – указатели на левого и правого сыновей текущего узла.

Эффективность поиска данного алгоритма растет с ростом числа эл-тов списка. Для 1000 эл-тов достаточно 10 сравнений.

# 14.2 Прошитые бинарные деревья

При представлении бинарного дер в виде узлов, содержащих инфо поле + 2 поля связи, то кол-во **полей связи** со значением **nil**, всегда > числа связей, указывающих на реально существующие узлы. Поэтому так хранить деревья неэффек-но для использования памяти, особенно, если размер информац-го поля = р-ру указателя.

Эффективность прохождения дер алгоритмами (рекурсивными и нерекурсивными) м б увеличена, если использ-ть **пустые указатели** на **отсутствующие** поддеревья для хранения в них адресов узлов преемников, которые надо посетить при заданном порядке прохождения бинарного дер. Такой **указатель** называется **нитью**. Его надо отличать от указателей в дер, которые использ-ся с левым и правым поддеревьями. Операция замены **пустых** **указателей** на **нити**, называется **прошивка**.

Она выполняется по-разному. Если **нити** заменяют пустые указатели в узлах с пустыми **правыми** поддер-ми, при просмотре в симметричном порядке, то бин-ое дер - **симметрично прошитое справа***.*Похожим образом м б определено бин-ое дер, **симметрично прошитое слева:** дер, в котором каждый пустой левый указатель изменен так, что он содержит нить – связь к предшественнику данного узла при просмотре в симметричном порядке. **Симметрично прошитое**бин-ое дер – которое симметрично прошито слева и справа. Однако левая прошивочная нить не дает преимуществ, как правая прошивочная нить. Рис 14.4 - дер, **симметрично прошитое справа***.* пунктирные линии - прошивочные нити.

777777777777777

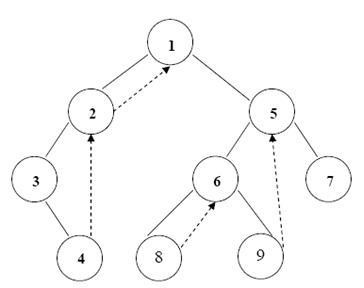


Рис 14.4 – Симметрично прошитое справа бин-ое дер

Бинарные деревья, **прямо прошитые справа и слева:** пустые правые и левые указатели узлов заменены соотв-но на их преемников и предшественников при прямом порядке просмотра. Прошитые деревья эффективно проходятся без использования стека.

Т к нужно как-то отличать обычную связь от прошивочной нити, каждому узлу добавляется 2 однобитовых (логических) поля тэга: **ltag** и **rtag**. Если значение тэга **true**, соответствующее поле связи является обычной связью, в случае значения **false** – прошивочной нитью. В этой связи **узел обычного** бинарного дер представляется структурой:

**Type**

ptr = ^node;

node = **record**

info : integer; {Инфо-ое поле}

left, right : pt; {Указатели на лев и прав **поддеревья**}

**end**;

Узел **прошитого** бинарного дер имеет иную структуру:

**Type**

ptr = ^node;

node = **record**

info: integer; {Инфо-ое поле}

ltag, rtag: boolean; {**Тэги прошивочных нитей**}

left, right: pt; {Указатели на лев и прав поддеревья}

**end**;

**Логические поля** в прошитом дер м принимать значения:

1. **ltag = true**, следовательно, **left** представляет собой обычную связь.
2. **ltag = false**, следов-но, **left** указывает на узел-предшественник. (родит)
3. **rtag = true**, следовательно, **right**представляет собой обычную связь.
4. **rtag = false**, следовательно, **right** указывает на узел-преемник. (сын)

Рассмотрим **вставку новой вершины слева** от заданной в симметрично прошитое бин-ое дер (рис 14.5). На рис 14.6 - результирующее дер.

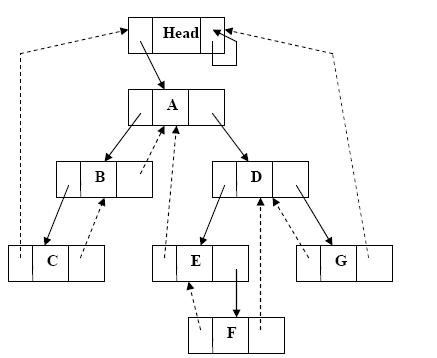


Рис 14.5 – Исходное симметрично прошитое дер

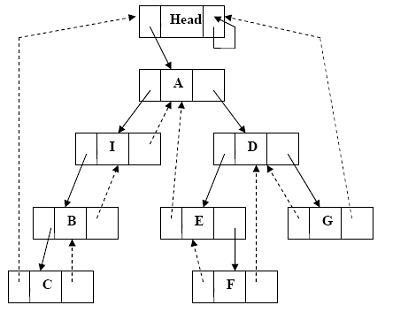


Рис 14.6 – Прошитое дер после вставки в него нового эл-нта

Здесь надо вставить вершину **I**как левое поддер **вершины *А***, если**А** не имеет левого поддер. Иначе - новая вершина **вставляется м/у *А* и ее левым сыном.**

Для удобства создания и обхода дер использ-ется д/п головная вершина ***Head***, которая служит предшественником его первой вершины и приемником всех его концевых вершин при симметричном обходе.

«+» прошитых деревьев: быстрый обход,нет необход-ти в стеке, м определить предшественника (отца?) и преемника (сына?) вершины.

«-»: включение новой вершины в дер занимает больше вр, т.к. надо поддерживать 2 типа связей: структурные и по нитям. Поэтому использ-ть прошитые деревья где изменения в дер редкие, а обходы частые.

# 14.3 Алгоритмы прошивки и обхода прошитых деревьев

Алгоритм **правос**торонней **симметр**ичной прошивки бинарного дер:

1. Строится бин-ое дер, поля ***ltag*** и ***rtag*** создаваемых узлов дер остаются неопределенными, а ***left***и ***right***соотв-но указывают на левое/ правое поддер / = ***nil***. На корень построенного дер указывает ***root***.
2. Создается **головной узел**, ***left***которого указывает на **корень** дер, а ***right***на сам **головной узел**:

***new(HEAD);***

***HEAD^.left := root; //*** ***root*** указывает на корень построенного дер

***HEAD^.right := HEAD;***

Инфо-ое поле и поля тэгов головного узла оставить неопределенными.

1. **Прошивка правых связей**. Вводим д/п **глобальный указатель** ***y*** (указатель на узел, предшествующий текущему – отец?). Указатель на текущий узел ***p*** устанавливаем на корень дер

***p := HEAD^.llink***

Выполняется симметричный обход дер. При обработке каждого узла проверяется:

если **p^.right <> nil**, то **p^.rtag:= true;**

иначе **p^.rtag:= false; y:= p;** (указатель на предшественника – отца?).

В любом случае выполняем.

Программный код процедур симметричного обхода **sim\_print** и прошивки правых связей **rightsew** б выглядеть так:

**procedure** sim\_print(**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;

**end**

**else** y^.rtag := true;

**end**;

y := p;

**end**;

**Begin // procedure** sim\_print(**var** x: pt)

**if** x <> nil **then**

**begin**

sim\_print(x^.left);

rightsew(x);

sim\_print(x^.right);

**end**;

**End**;

Алгоритм симметричного обхода прошитого бинарного дер:

Пусть **HEAD** – указатель на головной узел прошитого дер, ***p*** – указатель на текущий узел. Тогда алгоритм:

1. Переход к корню дер ( ***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***, Напр-р, **печать** ***p^.info***.
4. Если ***p^.rtag* = *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 станет ***p* = HEAD**.

**ТЕМА 15. КОДЫ ХАФФМАНА**

# 15.1 Применение деревьев. Представление сообщений кодами Хаффмана

Пр-р применения двоичных дер как структур данных: рассмотрим задачу конструирования ***кодов Хаффмана*** для **сжатия** инфы.Пусть есть сообщения из последоват-ти символов. В каждом сообщении символы независимы и появляются с известной вероятностью, не зависящей от позиции в сообщении. Напр-р, сообщения состоят из 5 символов ***а*, *b*, *с*, *d*, *е,***которые появляются в сообщениях с вероятностями 0.12, 0.4, 0.15, 0.08 и 0.25 соотв-но.

Надо: закодировать каждый символ последоват-тью из 0 и 1 так, чтобы **код** любого символа являлся **префиксом кода сообщения**, состоящего из последующих символов. Это ***префиксное св-во***позволяет декодировать строку из 0 и 1 последовательным удалением префиксов (т.е. кодов символов) из этой строки.

В табл 2 – возможные кодировки для наших 5 символов. Ясно, что первый код 1 обладает префиксным св-вом, т к любая последоват-ть из 3 битов б префиксом для другой последоват-ти из 3 битов, т.е. любая префиксная последоват-ть однозначно идентифицируется символом.

Алгоритм **декодирования** для этого кода: надо поочередно брать по 3 бита и преобразовывать каждую группу битов в соотв-ие символы. Напр-р, последоват-ть 001010011 соотв-ет исходному сообщению ***bсd****.*

Таблица. 2 двоичных кода

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Символ** |  | **Вероятность** | **Код 1** |  | **Код 2** |
|  | **а** | 0.12 |  | 000 | 000 |
|  | **b** | 0.40 |  | 001 | 11 |
|  | **с** | 0.15 |  | 010 | 01 |
|  | **d** | 0.08 |  | 011 | 001 |
|  | **е** | 0.25 |  | 100 | 10 |

Несложно проверить, что 2й код тоже имеет префиксные св-ва. Процесс **декодиро**вания здесь не отличается от аналогичного процесса для первого кода. (!) сложность для 2го кода – нельзя сразу всю последоват-ть битов разбить на отдельные сегменты, соотв-ие символам, т к символы м кодироваться и 2я и 3я битами. Для пр-ра рассмотрим двоичную последоват-ть **1101001**, которая опять представляет символы *bсd.* Первые 2 бита **11** однозначно соответствуют символу *b*, поэтому их м удалить, тогда получится **01001**. Здесь **01** также однозначно определяет символ ***с***и т.д.

**Задача конструирования кодов Хаффмана**: имея множ-во символов и значения вероятностей их появления в сообщениях, построить такой **код** с **префи**ксным св-вом, чтобы средняя длина кода (в вероятностном смысле) последоват-ти символов была **минима**льной. Мы хотим минимизировать среднюю длину кода, чтобы уменьшить длину вероятного сообщения, т.е. чтобы **сжать** сообщение. Чем короче среднее значение длины кода символов, тем короче за**код**ированное сообщение. В частности, 1й код из нашего Пр-ра имеет среднюю длину кода **3**: умножение длины кода каждого символа на вероятность его появления. 2й код имеет среднюю длину 2.2: символы *a* и *d* имеют суммарную вероятность = 0.20 и длина их кода = 3 бита, тогда как другие символы имеют код длиной 2.

Появляется вопрос: м ли придумать код, который был бы лучше 2го? - Да: есть код с префиксным св-вом, его средняя длина = 2.15 – наилучший возможный код с теми же вероятностями. Способ нахождения оптимального префиксного кода называется **алгоритмом Хаффмана***.* В этом алгоритме находятся 2 символа *а* и *b* с наименьшими вероятностями появления и заменяются 1м **фиктивным** символом, Напр-р ***х***, его вероятность появления = сумме вероятностей *а* и *b.* Затем, используя эту процедуру **рекурсивно**, находим **оптимальный префиксный код** для меньшего множества символов (где символы *а* и *b* заменены 1м символом *х*). Код для исходного множества символов получается из кодов замещающих символов путем добавления 0 и 1 перед кодом замещающего символа, и эти 2 новых кода принимаем как коды заменяемых символов. Напр-р, код символа ***а***б соответствовать коду символа ***х***с добавленным **0** перед этим кодом, а для кода ***b –***перед ***х***б добавлена 1.

М рассматривать **преф**иксные коды как пути на двоичном дер: проход от узла к его левому сыну соотв-ет **0** в коде, а к правому сыну – **1**. Если пометить листья дер кодируемыми символами, то получим представление префиксного **кода** в виде двоичного дер. Префиксное св-во гарантирует, что нет символов, которые были бы метками внутренних узлов дер (не листьев), и наоборот, помечая кодируемыми символами только листья, обеспечиваем префиксное св-во кода этих символов.

Двоичные деревья для кодов 1 и 2 из таблицы - на рис 15.1: дер слева соотв-ет коду 1, а справа – коду 2.

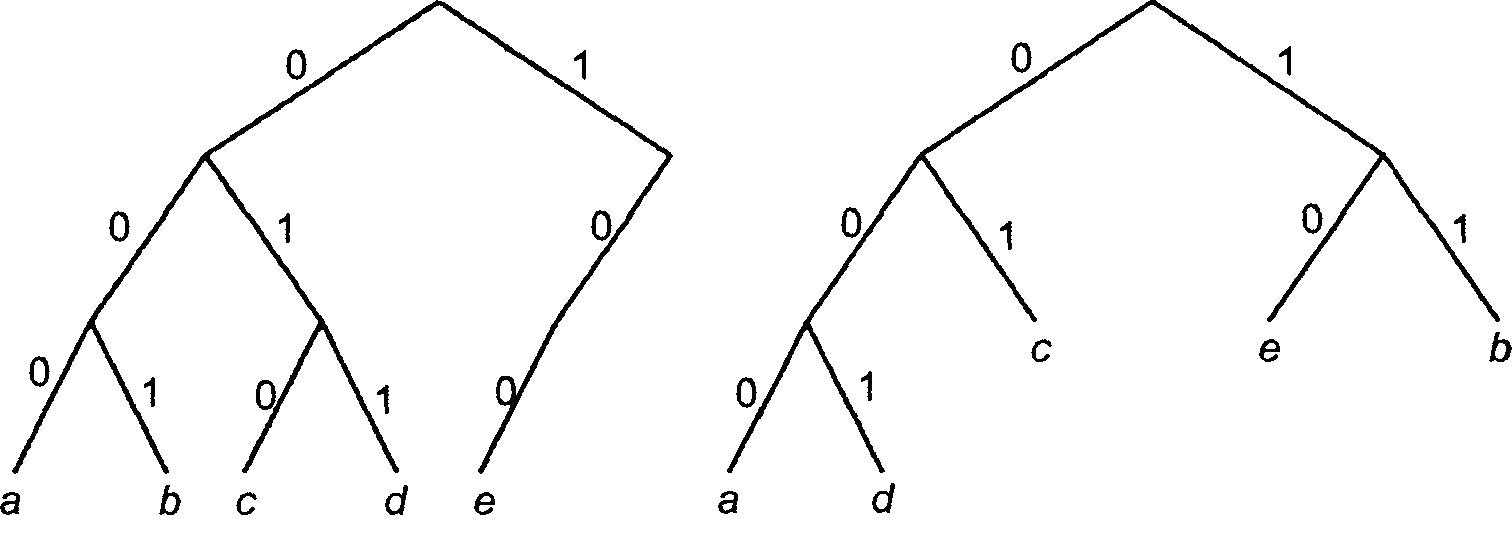


Рис 15.1 – Двоичные деревья, представляющие коды с **префик**сным св-вом

Для реализации алгоритма Хаффмана использ-ется **лес** *–* совокупность деревьев, чьи листья помечаются символами, для которых разрабатывается кодировка, а корни помечаются суммой вероятностей всех символов, соотв-их листьям дер. Эти суммарные вероятности называются **весом**дер. Вначале каждому символу соотв-ет дер, состоящее из 1 узла, а в конце работы алгоритма получается 1 дер: все его листья б помечены кодируемыми символами. В результирующем дер путь от корня к любому листу – код для символа-метки этого листа, составленный по схеме: левый сын узла соотв-ет **0**, а правый – **1**.

Важный этап в работе алгоритма - выбор из леса 2х деревьев с наименьшими весами. Эти 2 дер комбинируются в 1, а его вес = сумме весов этих деревьев. При слиянии деревьев создается новый узел, который становится **корнем** объединенного дер и имеет в качестве левого и правого сыновей корни старых деревьев. Этот процесс идет до получения только 1 дер. Это дер соотв-ет коду, который при заданных вероятностях имеет минимально возможную среднюю длину.

Рассмотрим на пр-ре шаги выполнения алгоритма Хаффмана (рис 15.2). Кодируемые символы и их вероятности – в табл. Символы ***а*, *b*, *с*, *d,* *е***получили соотв-но коды **1111, 0, 110, 1110** и **10**. В этом Пр-ре есть только 1 нетривиальное дер, соответствующее оптимальному коду, но их м б неск-ко. Напр-р, если бы символы ***b***и ***е***имели вероятности соотв-но 0.33 и 0.32, то после шага алгоритма на рис. 2.в, м было бы комбинировать ***b***и *е,* а не присоединять ***е***к большому дер (рис 15.2.г).

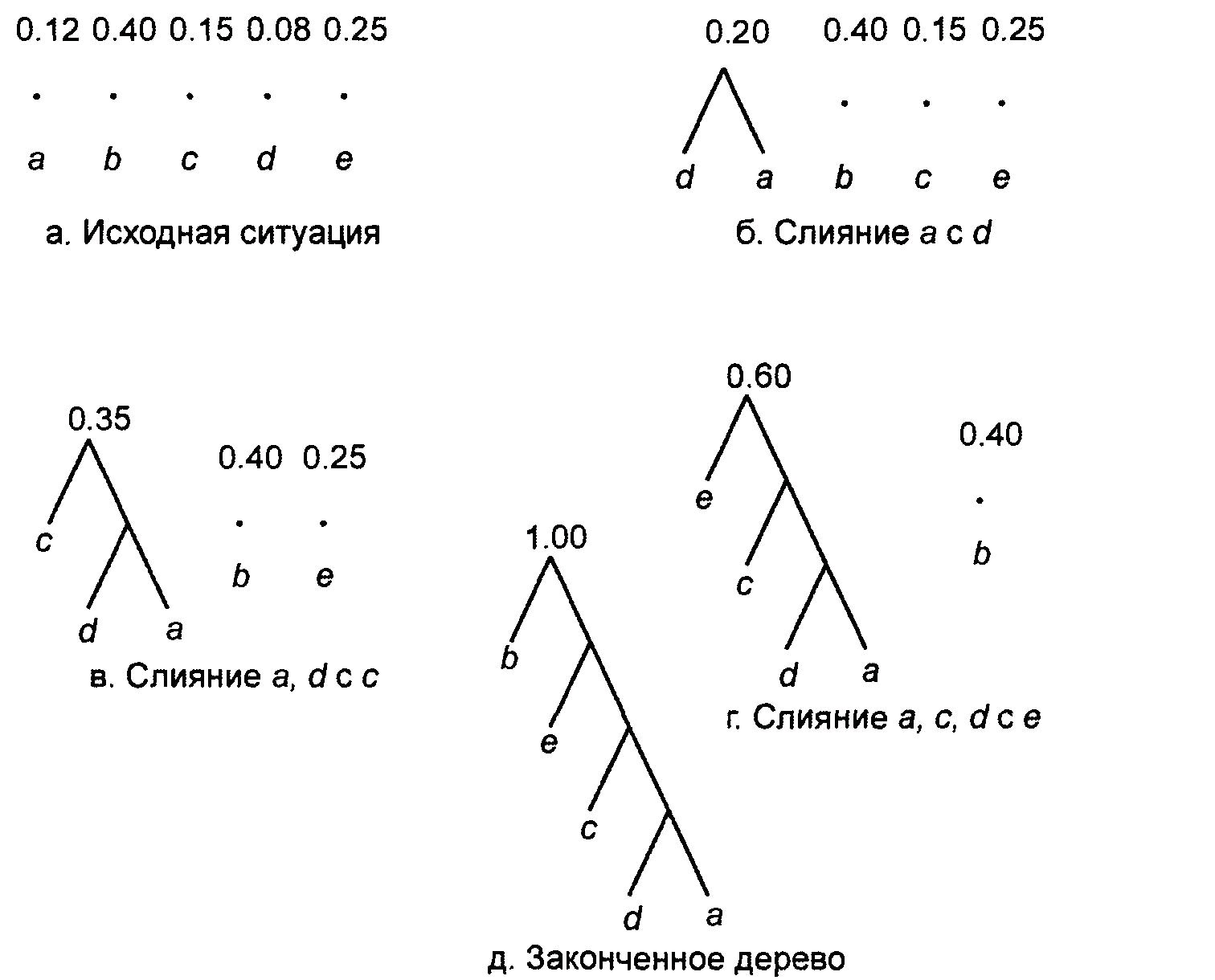


Рис 15.2 – Этапы построения дер Хаффмана

Д/представления бинарных дер исп-ся массив *TREE*, сост из записей типа:

**Type**

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. А в массиве ***ALPHABET*** каждому символу, подлежащему кодированию, ставится в соотв-вие вероятность его появления и лист, меткой которого он является. Массив ***ALPHABET*** состоит из записей, имеющих тип:

**Type**

ALPHABET\_elem = **record**

symbol : char;

probability : real;

leaf : integer

**end**;

**var**

ALPHABET : **array**[1..100] **of** ALPHABET\_elem;

Для представления прямо деревьев надо массив ***FOREST***, состоящих из записей, имеющих тип:

**Type**

FOREST\_elem = **record**

weight : real;

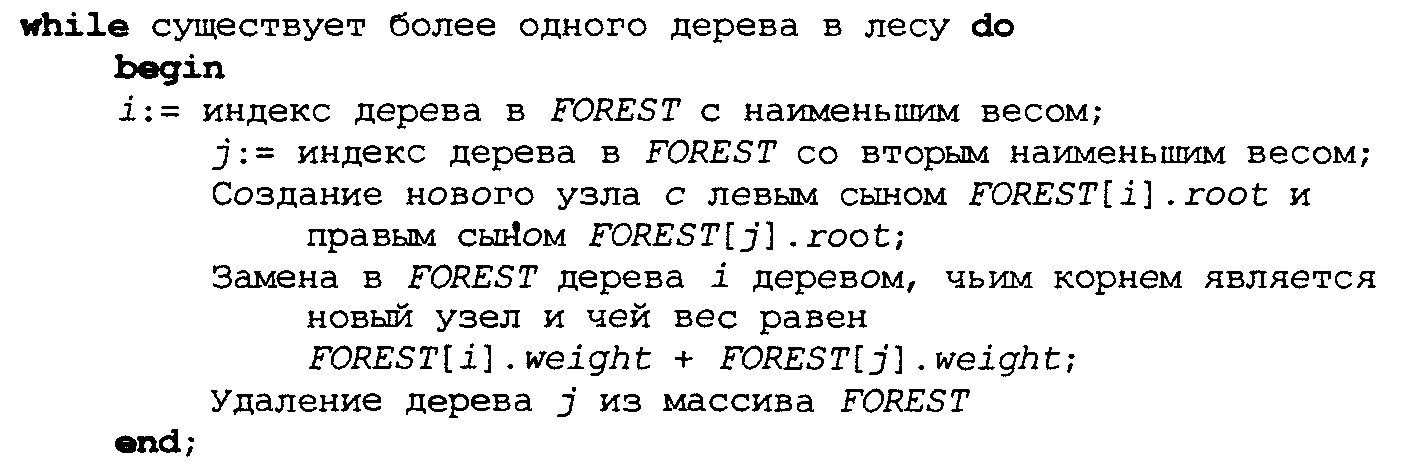
root : integer

**end**;

**var**

FOREST: **array**[1..100] **of** FOREST\_elem;

Ниже - эскиз проги построения дер Хаффмана.



**While существует > 1 дер в лесу в лесу do**

**Begin**

**i:= индекс дер в FOREST с наименьшим весом;**

**Создание нового узла с левым сыном FOREST[i].root и**

**правым сыном FOREST[j].root;**

**Замена в FOREST дер i дерм, чьим корнем явл-ся**

**новый узел и чей вес =**

**FOREST[i].weight + FOREST[j].weight;**

**Удаление дер j из массива FOREST**

**end;**

Исходное состояние 3х массивов – на рис 15.3.

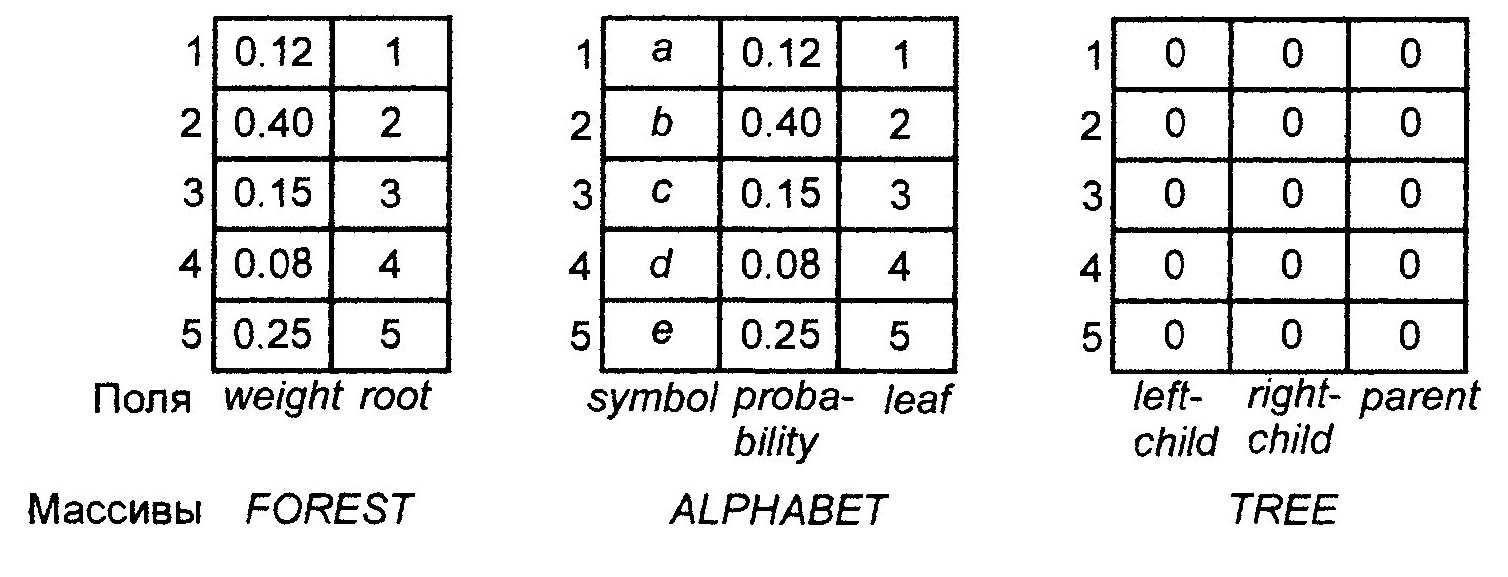
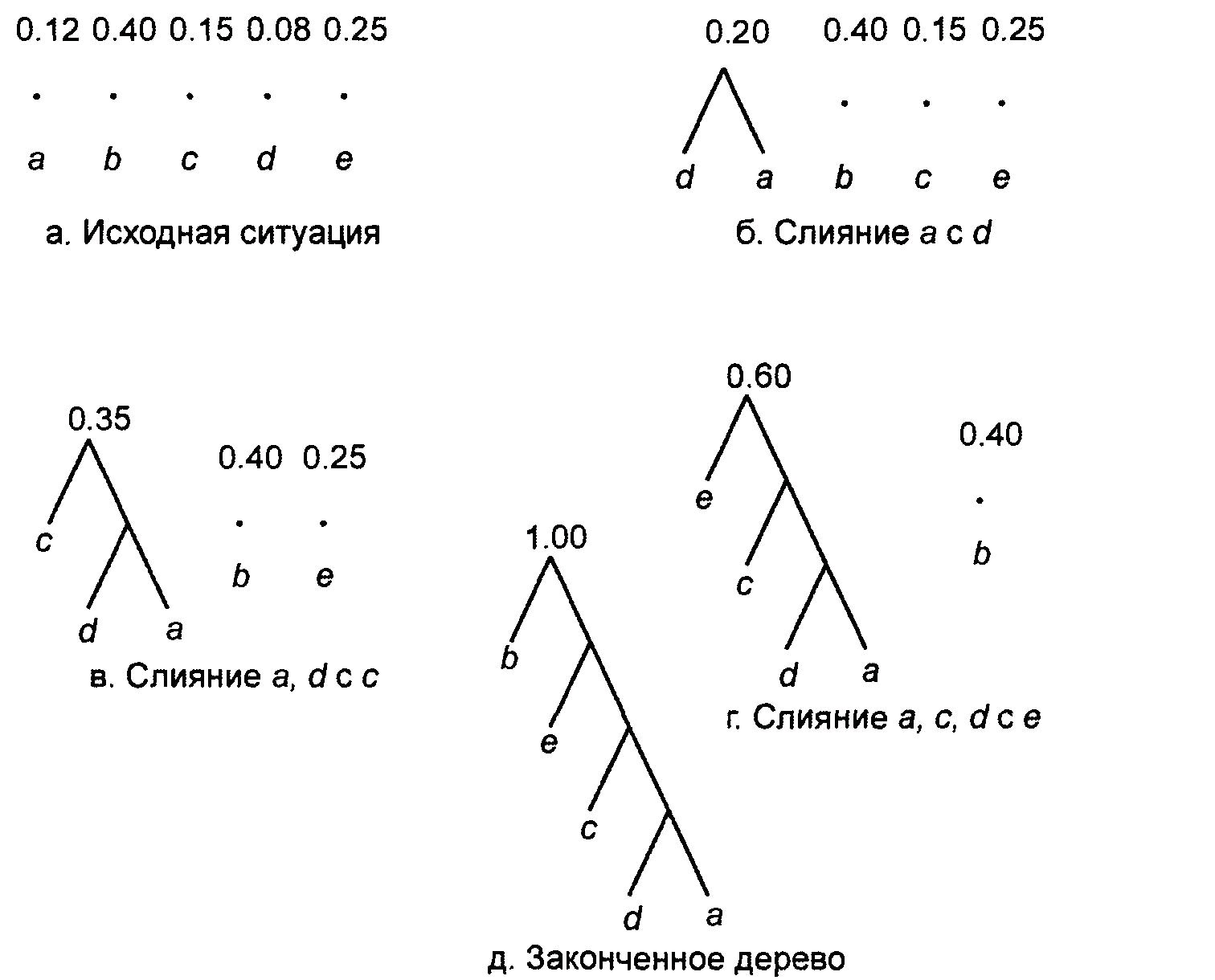


Рис 15.3 – Исходное состояние массивов

На рис 15.4 - состояние 3х массивов, соответ-щее лесу на рис 15.2.в.



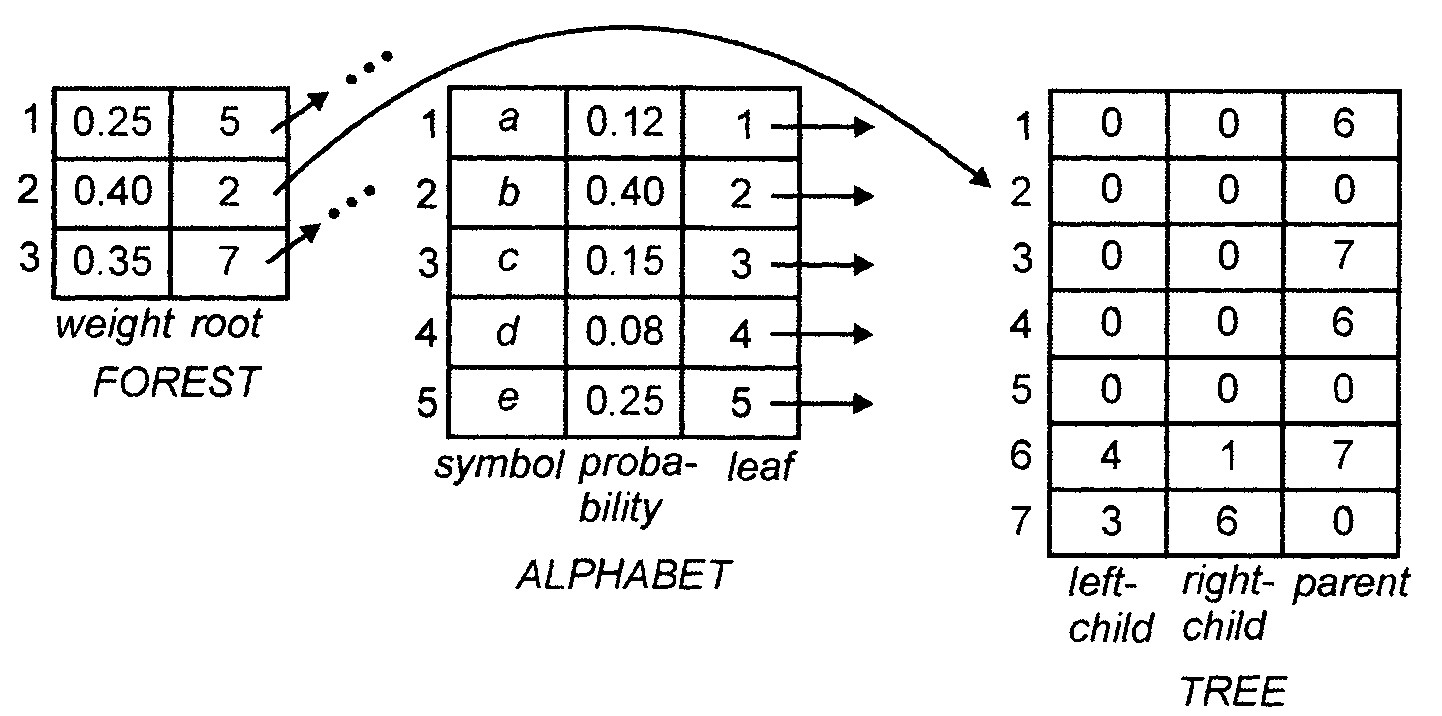


Рисунок 15.4 – Промежуточное состояние массивов

После окончания работы алгоритма код каждого символа м определить так: 1) найти в массиве ***ALPHABET***запись с нужным символом в поле ***symbol***.

2) по значению поля **leaf** этой же записи определить местоположение записи в массиве ***TREE***, которая соотв-ет листу, помеченному рассматриваемым символом.

3) последовательно переходить по указателю ***parent*** от текущей записи, Напр-р, соответствующей узлу *n*, к записи в массиве ***TREE***, соответствующей его родителю ***р***. По родителю *р* определяют, в каком его поле,**leftchild** или **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**(**правого**).

4) выполняется переход к родителю узла ***р*** и определяется, является ли его сын ***р*** правым или левым, и в соответствии с этим печатается следующая 1 или 0.

Так идет до корня дер. В результате код символа б напечатан в виде последоват-ти битов, но в обратном порядке. Чтобы распечатать полученную последоват-ть в прямом порядке, нужно каждый очередной бит помещать в стек и распечатать содержимое стека в обычном порядке.

**ТЕМА 16. СБАЛАНСИРОВАННЫЕ БИНАРНЫЕ ДЕРЕВЬЯ**

# 16.1 Идеально сбалансированные бинарные деревья

Т к максимальный путь до листьев дер определяется высотой дер, то при заданном числе узлов дер его надо строить **мини**мальной высоты: надо размещать **максим**ально узлов на всех уровнях, кроме последнего – в бинарном дер надо все поступающие при построении узлы распределять поровну слева и справа от каждого из вышерасположенных узлов.

Бин-ое дер идеально сбалансировано, если для каждого его узла кол-во потомков в левом и правом поддеревьях различается не более чем на 1.

Рекурсивный алгоритм для построения идеально сбалансир-го бинарного дер, если даны кол-во его узлов и последоват-ть значений его вершин *(a[1], a[2],…, a[n]):*

начиная с a[1]*,* выбираем очередное a[i] в качестве значения **корня дер** (поддер).

1. Тем же способом строим **лево**е поддер с кол-вом узлов **nl = n/2**.
2. Строим **правое** поддер с кол-вом узлов **nr = n − nl − 1.**

Т о, значение **a[1]** окажется в корне дер, и именно на него б ссылаться указатель дер. Значения **a[2], a[3], ..., a[nl ]** попадут в **левое** поддер, а значения ***a*[*nl* +1], *a*[*nl* + 2], ...*a*[*n*]** – в **правое** поддер. 🡺, исходная последоват-ть данных полностью определяет распределение значений по узлам дер.

Рис 16.1 – идеально сбалансир-ое бин-ое дер, построено по набору знач-ий 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 на дер не накладываются никакие требования к значениям в узлах – эти значения данных не упорядочены, что ведет к тому, что поиск узла с нужными данными идет последовательно всех узлов дер, и вр поиска б прямо пропорционально количеству узлов дер. Поэтому на практике применяют сбалансированные деревья поиска, которые обеспечивают минимальное вр поиска порядка **log2 *n***, где ***n*** – кол-во узлов дер.

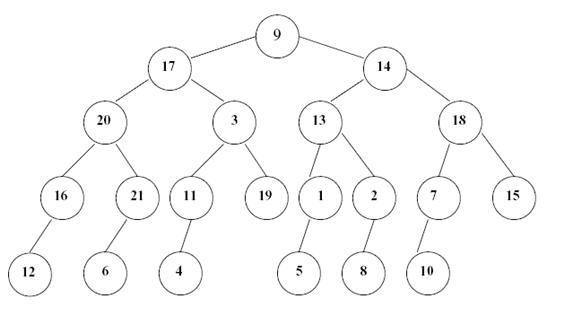


Рис 16.1 – Идеально сбалансированное бин-ое дер

Добавление и удаление узлов идеально сбалансированного дер трудно и для поддержания сбалансиров-сти надо иметь инфу о сбалансир-ти каждого поддер и поддерживать ее. Поэтому их применяются для работы с данными, которые мало изменяются при обработке.

# 16.2 Бинарные деревья поиска

Бинарные деревья применяют для данных, эл-нты которых ищут по уникальному ключу.

**Бинарное дер поиска** – если оно организовано так, что для каждого узла **ti** все ключи в **левом** поддер < ключа **ti**, а ключи в **правом** поддер > ключа **ti**.

В дер поиска м найти место каждого ключа, начиная двигаться от корня и переходить на левое / правое поддер каждого узла (в зав-ти от значения ключа).

Т о, места эл-тов в дер определяются значениями ключей (определяющий фактор) + последоват-тью их поступления (зависит степень сбалансирован-ти дер). При случайном распределении ключей в исходной последоват-ти получается почти сбалансированное дер. Если же исходная последоват-ть упорядочена по возрастанию / убыванию ключей, то дер вырождается в последовательный список. Высота такого дер = число эл-тов - 1.

Построение дер поиска: 1й эл-нт образует корень дер. Для следующих эл-тов идет поиск места включения по ветвям дер, пока не б найден подходящий узел с нулевым указателем, туда и подключается эл-нт. Для каждого узла запрос динамической памяти, ее адрес заносится в указатель узла-предка, данные эл-нта помещаются в узел, и обнуляются левый и правый указатели нового узла.

Дер поиска, построенное из последоват-ти ключей

*9, 17, 20, 16, 12, 21, 6, 3, 11, 4, 19, 14, 13, 1, 5, 2, 8, 18, 7, 10, 15*, имеет вид:

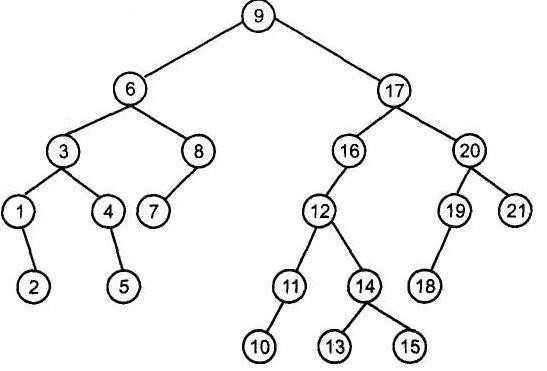


Рис 16.2 – Бин-ое дер поиска

# 16.3 Сбалансированные деревья поиска

Вр поиска в бинарном дер поиска определяется высотой самого дер. Для заданного числа эл-тов *n* высота дер зависит от порядка поступления эл-тов при построении дер и м колебаться от log2 *n* - идеально сбалансированное дер, до ***n*-1 -** вырождение дер в линейный список. 🡺, затраты на поиск и включение б порядка от log2 *n* до *n*.

Доказано, что при случайном порядке включения эл-тов в дер средние затраты на поиск эл-нта б **1.386\*log2 *n*.** Увеличение затрат на **39%** объясняется простыми средствами поддержания обычного дер поиска по сравнению с идеально сбалансированным дерм поиска. Но при работе с большими деревьями св-во случайности распределения поступающих данных редко соблюдается. Обычно выходные последоват-ти - частично упорядочены.

Для таких последовательностей наиболее подходящие - сбалансированные деревья поиска. Рассмотрим 1 из них: **АВЛ-деревья**.

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

**АВЛ-дер** – высота поддеревьев для каждой вершины отличается не более чем на 1.

Максимальная высота АВЛ-дер с *n* вершинами не превосходит

**1.44\* log2(*n* + 1) − 1.33**, т.е. затраты на поиск не выше **1.45\* log2 *n***.

Отличие АВЛ-дер от обычного дер поиска: при включении / удалении эл-тов надо поддерживать сбалансированность дер в целом: в каждый узел дер добавляется 1 вспомогательное поле, содержащее инфу о равновесн-ти поддеревьев (показатель сбалансированности узла). Его значениями м б:

1. 0 – высоты правого и левого поддеревьев =ы;
2. 1 – высота **правого** поддер больше;
3. -1 – высота **левого** поддер больше.

При попытке добавить / удалить эл-нт в поддер с показателем сбалансированности, отличным от 0, дер м стать несбалансир-ым, и потребуется операция балансировки.

# 16.4 Операции над деревьями

Динамические структуры в виде деревьев – для хранения и манипулирования данными. Распространенные операции с данными на бинарных деревьях, размещенные в динамической памяти:

1) **Создание дер***.* Доступ к дер - по указателю, объявленному в проге как переменная. Алгоритм включения в дер отдельных эл-тов:

- поиск места включения;

- получение динамической памяти для эл-нта + образование связи узла с дер посредством указателя;

- занесение данных эл-нта в узел.

2 последних действия не зависят от вида дер, а поиск места – зависит от его вида. После создания дер, бин-ое дер поиска легко наращиваются добавлением новых эл-тов. Идеально сбалансир-ые деревья для наращивания с сохранением своих свойств требуют гораздо больших затрат.

Т к новые эл-нты м включать в дер поиска в любой момент выполнения проги, то создание дер сводится к управлению вызовом ф-ции включения эл-нта.

2) **Поиск эл-нта в дер***.* Цели: 1) определить, есть ли искомый эл-нт в дер; 2) поиск для выборки и обработки данных эл-нта: результат – адрес вершины; 3) поиск для удаления эл-нта (часть операции удаления, а не самостоятельная операция).

Алгоритм поиска зависит от вида дер: в идеально сбалансированном дер поиск – обходом вершин дер в некоторой последоват-ти. Минимальная длина поиска = 1; максимальная – *n* (*n* – кол-во узлов дер); средняя – *n*/2.

Алгоритм поиска в БД поиска простой:

1) Если ключ поиска = ключу в вершине, значит, ключ найден и его адрес возвращается через параметр-указатель.

2) Если ключ поиска меньше ключа в вершине, то движение вниз влево, иначе – вниз вправо.

3) Если в очередной вершине движение вниз невозм (указатель = **nil**), значит искомого ключа в дер **нет**.

Максимальная длина поиска = высоте дер.

3) **Вставка эл-нта в дер*.***Особенности операций – зависит от вида дер, состоит из 3 действий. Если в дер уже есть эл-нт с включаемым ключом: такой эл-нт м отвергнуть, либо включить. Напр-р, если дер использ-ся для сортировки по ключам или использ-ие такого эл-нта для обновления данных в узле дер.

# 16.5 Вставка (Операция включения) эл-нта в АВЛ-дер

- 2 этапа. Т к сбалансированное дер – частный случай обычного дер поиска, то 1й этап – включение эл-нта в дер == при включении в обычное дер: проход по пути поиска, получение динамической памяти и формирование в ней новой вершины дер. Д/п-о новой вершине устанавливается признак ее сбалансированности**, = 0**: у новой вершины нет поддеревьев. 2й этап – при обратном движении по дер: восстановление сбалансированности в узлах дер (если она была нарушена): учитывается, откуда (слева / справа) идет возврат в вершину, каков показатель сбалансированности в ней, выросла ли высота поддер. Рассмотрим особенности **восстановления** сбалансиров-ти на пр-рах.

Пусть есть дер с вершинами 3 и 2 (рис. 3, 1е дер). Высота ***hl***левого **под**дер вершины 3 =1, высота ***h****r* правого поддер вершины 3 =0, сбалансированность ***bal***3 =−1.

У вершины 2 *hl* = ***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 поддер соблюдается. В вершине 3 стало ***h****l* =2, ***h****r* =0, т.е. критерий сбалансированности нарушен и дер нужно перестраивать (ср дер на рис16.3) однократным **LL поворотом**, в результате получаем сбалансированное дер с вершиной 2 (правое дер рис. 16.3). Необходимые операции балансировки – обмен указателями ***ti*** и *Т* **по кругу по час стрелке**. + надо изменить показатели сбалансированности вершин.

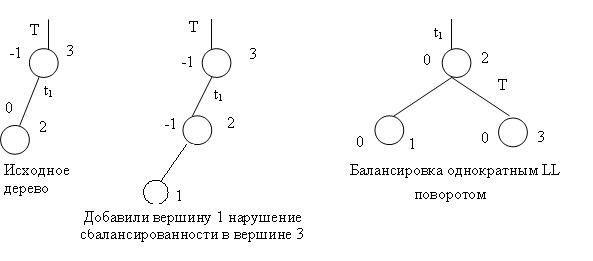


Рис 16.3 – Вставка эл-нта в АВЛ-дер и его балансировка LL-поворотом

2й пр-р: в дер с вершинами 4 и 5 включим вершину 6. Для балансировки в вершине 4 выполняются **RR-поворот**, **обмен указателями по кругу против часовой стрелки** (рис 16.4).

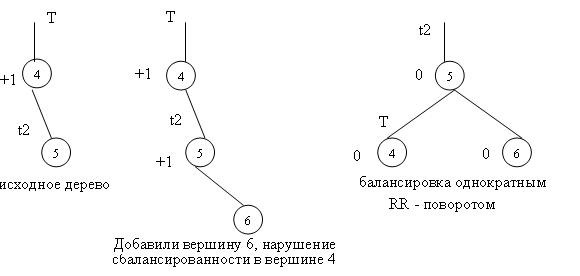


Рис 16.4 – Вставка эл-нта в АВЛ-дер и его балансировка RR поворотом

Более сложные ситуации приводят к **двукратным поворотам**: направо и налево – **RL-поворот**; налево и направо – **LR-поворот**. Пр-р двукратного RL-поворота - включение в дер с верш-ми 5 и 7 новой вершины 6 (рис 16.5). Возникла несбалансированность в вершине 5. Поэтому вначале правый поворот 3х вершин, затем левый поворот 2х вершин (7 и 6).

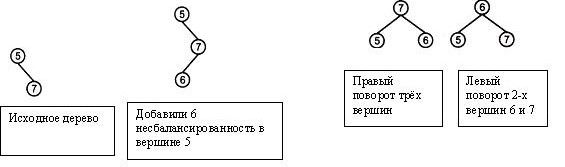


Рис 16.5 – Вставка эл-нта в АВЛ-дер и его балансировка RL поворотом

При включении в дер с вершинами 9 и 3 новой вершины 8 – несбалансир-ть поддер с корнем. Она устраняется двукратным LR поворотом: сначала левым поворотом 3х вершин, а затем правым поворотом 2х вершин – 3 и 8 (рис 16.6).

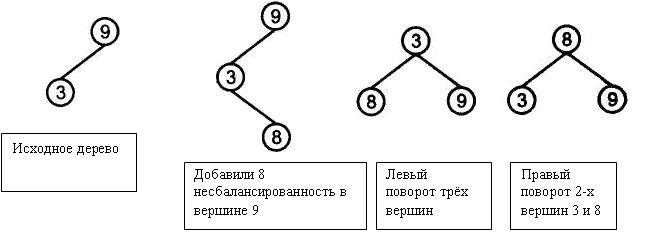


Рис 16.6 – Вставка эл-нта в АВЛ-дер и его балансировка LR поворотом

Порядок построения сбалансированного дер из набора эл-тов с ключами: 4, 5, 7, 2, 1, 3, 6, - на рис 16.7.

**Обработка данных в вершине дер***.*

2 вида обработки: 1) выборочная обработка отдельной вершины и 2) последовательная обработка всех вершин. 1) Выборочная обработка: включает поиск эл-нта, и если он найден, то обработку данных в вершине. Обработка данных зависит от решаемой задачи и м сводиться к извлечению данных без их изменения либо обновлению данных в вершине дер. 2)Последовательная обработка: идет обход дер.

**Сохранение и восстановление дер***.*

Выполняется **нисход**ящий обход дер с сохранением ключа и данных каждого узла. Указатели сохранять **не** надо. По сохраненным ключам и данным узлов строится БД.

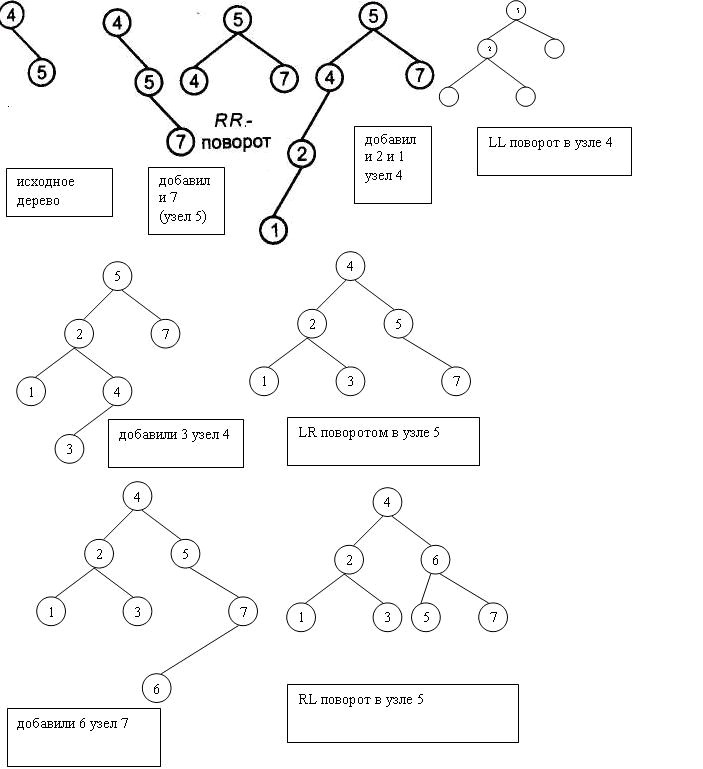


Рис 16.7 – Порядок построения сбалансированного дер

**Уничтожение дер** (БД) - достаточно выполнить обход дер и при посещении каждого узла освободить память, занимаемую им. Но нельзя уничтожать узел, имеющий поддеревья: занятая им память не б освобождена и станет недоступной для повторного использования. Это требование обеспечивается при восходящем обходе дер. После освобождения всей памяти, указатель дер обнулить.

**===============================================**

**ТЕМА 17. ОРИЕНТИРОВАННЫЕ ГРАФЫ:**  **ОПРЕДЕЛЕНИЯ И**

**ОПЕРАЦИИ**

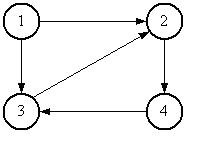
# 17.1 Основные определения ориентированных графов

**Ориентированный граф** (или **орграф**) **G = (V, E)** состоит из множества вершин (узлы) *V* и множества дуг (ориентированные ребра) *Е*. Дуга – в виде упорядоченной пары вершин **(v, w***)*, где вершина **v** называется **начало**, а **w**– **конец** дуги. Дугу ***(v, w****)* записывают как ***v* →*w*** и изображают в виде



дуга ***v* →*w*** ведет от вершины ***v*** к вершине ***w***, а вершина ***w*** смежная с верш-й *v*.

На рис 17.1 – орграф с 4я вершинами и 5 дугами.

Рис 17.1 – Пр-р орграфа

Вершины орграфа – для представления объектов (города), а дуги – для отношений м/у объектами (маршруты).

**Путь**в орграфе – последоват-ть вершин *v*1,*v*2,… v*n*−1,*vn* , для которых существуют дуги *v*1→*v*2,*v*2 →*v*3,...*vn*−1→*vn*. Этот путь начинается в вершине ***v*1** и, проходя через вершины *v*2 ,...*vn*−1, заканчивается в вершине***vn***. Длина пути = кол-во дуг, составляющих путь, здесь длина пути **= *n–1***. 1 вершина рассматривается как путь длины 0 от вершины *v* к этой же вершине *v*.

**Простой путь** - если все вершины на нем разные (за исключением 1й и последней). **Цикл** – простой путь длины не менее 1, который начинается и заканчивается в 1 и той же вершине. На рис 17.1 вершины 3, 2, 4, 3 образуют цикл длины 3.

В приложениях к вершинам и дугам присоединяют инфу. Для этого использ-ется **помеченный граф** – орграф, у которого каждая дуга и/или каждая вершина имеет соотв-щие **метки**. Меткой м б имя, вес, стоимость (дуги), или значение данных заданного типа.

# 17.2 Представление ориентированных графов

Для представления орграфов использ-т разные Стр-ы Дан. Выбор Стр-ы Дан зависит от операторов, которые б применяться к вершинам и дугам орграфа.

Представление орграфа *G=(V, E)* **- матрица смежности.** Пусть множ-во вершин орграфа ***V = {1, 2, … n}*,** тогда матрица смежности графа *G* - матрица *А* размера ***n**n***со значениями булевого типа, где ***А[i, j]= true*** только тогда, когда существует дуга из вершины *i* в вершину *j*. В матрице смежности значение **true** заменяют 1, а **false** – 0. Вр доступа к эл-нтам матрицы смежности зависит от размеров множества вершин и множества дуг. Представление орграфа в виде матрицы смежности применять в 1) алгоритмах, где часто проверяют существование данной дуги; 2) помеченные орграфы: эл-нт А[i, j] = метке дуги i→j. Если дуги от вершины i к вершине j нет, то значение *А[i, j]-* как пустая ячейка. На рис. 2 – помеченный орграф и соотв-ая ему матрица смежности.

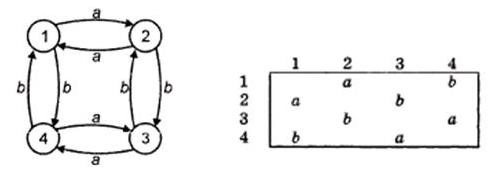


Рис 17.2 – Помеченный орграф и соотв-ая ему матрица смежности

«-» матриц смежности: она требует объема памяти = *n*2, даже если дуг значительно меньше, чем*n*2. Поэтому для чтения матрицы или нахождения в ней эл-нта надо вр порядка *n*2, что не позволяет создавать алгоритмы с временем *n* для работы с орграфами, имеющими порядка ***n*** дуг.

Поэтому вместо матриц смежности использ-т представления орграфов как **списки смежности –** для вершины***i*** называется список всех вершин, смежных с вершиной ***i***, упорядоченный определенным образом. Т о, орграф *G* м представить массивом ***HEAD***, чей эл-нт ***HEAD[i]***- указатель на список смежности вершины ***i***. Представление орграфов как списки смежности требует для хранения объем памяти, пропорциональный сумме количества вершин **+** количества дуг. Если кол-во дуг имеет порядок ***n*,** то общий объем памяти имеет такой же порядок. Но и для списков смежности вр поиска определенной дуги м иметь порядок n, т.к. такой же порядок м иметь кол-во дуг у определенной вершины. На рис 17.3 – стр-ра данных, представляющая орграф с рис. 17.1 с пом связанных списков смежности. Если дуги имеют метки, то их хранить в ячейках связанных списков. Для вставки и удаления эл-тов в списках смежности надо иметь массив ***HEAD***, содержащий указатель на ячейки заголовков списков смежности, но не сами смежные вершины.

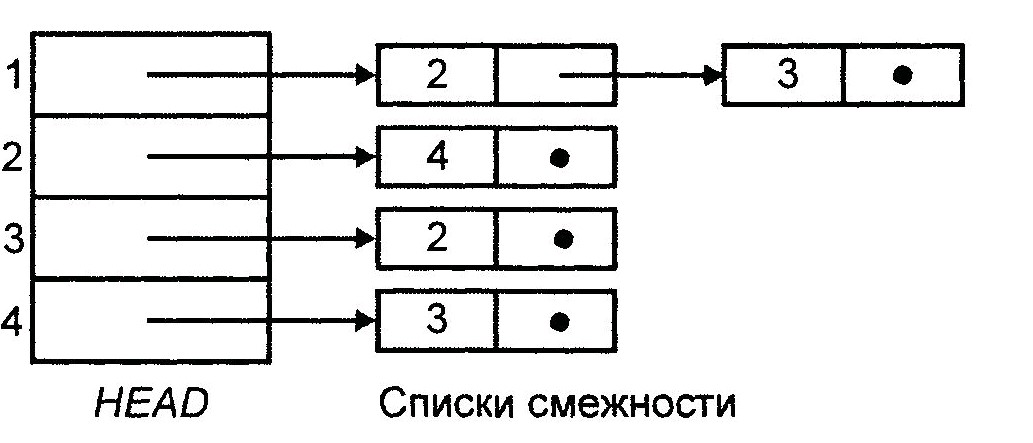
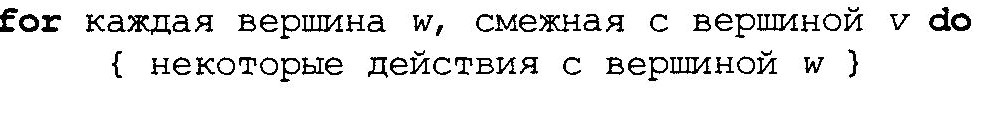


Рис 17.3 – Стр-ра списков смежности для орграфа

# 17.3 Операторы над ориентированными графами

1) чтения меток вершин и дуг; 2) вставки и удаления вершин и дуг; 3) оператор перемещения по последовательностям дуг.

В прогах встречаются операторы, которые неформально м описать подобно оператору: 

For каждая вершина w, смежная с вершиной v do

{нероторые действия с вершиной w}

Для реализации такого оператора нужен индексный тип данных для работы с множ-вом вершин, смежных с заданной вершиной *v*. Напр-р, если для представления орграфа использ-ются списки смежности, то **индекс**

– это позиция в списке смежности вершины *v*. Если применяется матрица смежности, тогда индекс – целое число, соответствующее смежной вершине. Для просмотра множества смежных вершин надо 3 оператора:

1. ***FIRST****(v)* возвращает индекс 1й вершины, смежной с вершиной *v*. Если у *v* нет смежных вершин, то возвращается «нулевая» вершина .
2. ***NEXT****(v, i)* возвращает индекс вершины, следующий за индексом i, смежной с *v*. Если *i* – индекс последней вершины, смежной с вершиной *v*, то возвращается .
3. ***VERTEX****(v, i)* возвращает вершину с индексом i из множества вершин, смежных с *v.*Если для представления орграфа использ-ется матрица смежности, то ф-ция ***VERTEX****(v, i)* просто возвращает индекс **i.**

Ниже – код функций ***FIRST****(v)* и***NEXT****(v, i)*.

В этом листинге матрица смежности *А* размера *n**n* определена вне этих функций с помощью объявления:

{-------- Программная реализация ф-ции FIRST() ---------------}

A : **array**[1..n, 1..n] **of** integer; {Матрица смежности}

**Function** FIRST(v: integer): integer;

**Var**

i: integer;

**begin**

**for** i:= 1 **to** n **do**

**if** A[v,i]= 1 **then**

**begin**

first:= i; **exit**;

**end**;

first:= -1; {Признак отсутствия смежных вершин}

**end**;

{--------- Программная реализация ф-ции NEXT()--------------}

**Function** NEXT(v: integer; i: integer): integer;

**Var**

j: integer;

**begin**

**if** i= n **then**

**begin**

next:= -1; **exit**;

**end** {i – последняя вершина, смежная с v}

**else**

**for** j:= i+1 **to** n **do**

**if** A[v, j] = 1 **then**

**begin**

next:= j; **exit**;

**end**;

next:= -1; {i – последняя вершина, смежная с v}

**end**;

С помощью рассмотренных ф-ций м организовать последовательный просмотр вершин, смежных с вершиной *v*.

# 17.4 Нахождение кратчайшего пути на ориентированном графе

Пусть есть орграф *G = (V, E)*, где все дуги имеют неотрицательные метки, а 1 вершина определена как **источник**. Задача – найти стоимость кратчайших путей от источника ко всем другим вершинам граф *G*. Длина пути = сумме стоимостей дуг, составляющих путь. Эта задача - нахождение кратчайшего пути с 1 источником (длина пути м измеряться и в нелинейных единицах (временных)).

Для решения задачи **– алгоритм Дейкстры**. Алгоритм строит: множ-во *S* вершин, для которых кратчайшие пути от источника уже известны. На каждом шаге к множеству ***S*** добавляется та вершина (из оставшихся), расстояние до которой от источника меньше, чем для других (оставшихся вершин). Если стоимости всех дуг неотрицательны, то кратчайший путь от источника к конкретной вершине идет только через вершины множества *S*. Такой путь называют **особым***.*

На каждом шаге алгоритма использ-ется массив ***D***, в который записываются длины кратчайших особых путей для каждой вершины. Когда множ-во *S* б имеють все вершины орграфа, т.е. для всех вершин б найдены особые пути, тогда массив ***D*** б имеють длины кратчайших путей от источника к каждой вершине. Ниже – фрагмент проги алгоритма Дейкстры. **FL** – массив типа **Boolean**. Если значение его эл-нта = **false**, то соотв-ий ему эл-нт массива ***D*** должен проверяться на предмет поиска кратчайшего пути, иначе - этот эл-нт больше не участвует в рассмотрении.

**For** m:= 1 **to** n **do**

P[m]:= 1; {Р – массив, использ-ый д/построен кратчайших путей}

**For** j:= 1 **to** n **do**

D[j]:= C[i, j]; {C - матрица цен}

**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**

**if** fl[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. множ-во вершин ***V={1, 2, … n}*,** причем вершина 1 – источник. Массив *С* – это 2мерный массив стоимостей, где эл-нт***C[i, j]*** = стоимости дуги ***i* → *j*.** Если дуги *i*→ *j* нет, 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 орграфа на рис 17.4. Вначале S={1}, D[2]=10, D[3]=∞, D[4]=30, D[5]=100. На 1ом 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. Последоват-ть значений эл-тов массива D после каждой итерации цикла показаны в таблице ниже.

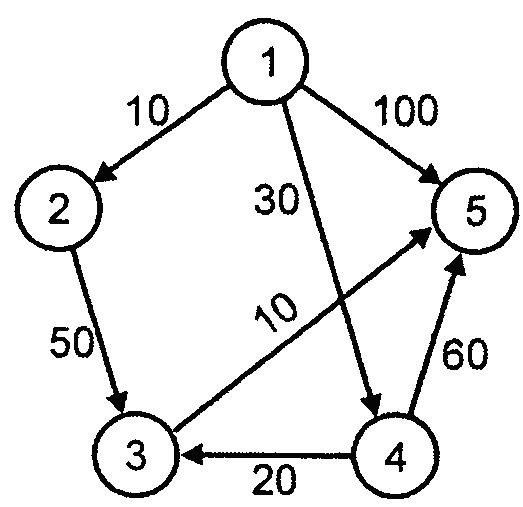


Рис 17.4 – Помеченный орграф, к которому применен алгоритм Дейкстры

Вычисления для орграфа на рис 17.4.



В рассмотренный алгоритм м внести изменения, которые позволят определить кратчайший путь для любой вершины графа: надо ввести еще 1 массив *P*, где **P[v]** содержит вершину, прямо предшествующую вершине *v* в кратчайшем пути. Вначале **P[v] = 1** для всех *v*1. В листинге алгоритма Дейкстры при выполнении условного оператора **D[v] + C[v, i] < D[i]**, эл-нту P[i] присваивается значение*v*. После выполнения алгоритма кратчайший путь к каждой вершине м найти с помощью обратного прохождения по предшествующим вершинам массива *Р*. Для рассмотренного орграфа массив *Р* имеет значения: **P[2]= 1, P[3]= 4, P[4]= 1, P[5]= 3**. Для определения кратчайшего пути, Напр-р от вершины 1 к вершине 5, надо отследить в обратном порядке предшествующие вершины, начиная с вершины 5. Т о, кратчайший путь из вершины 1 в 5 составляет последоват-ть вершин: **1, 4, 3, 5**.

# 17.5 Нахождение кратчайших путей м/у парами вершин

Пусть дан орграф *G=(V, E)* и надо определить кратчайшие пути м/у всеми парами вершин. Каждой дуге ***v*→*w*** этого графа сопоставлена неотрицательная стоимость **C[v, w].** Общая задача нахождения кратчайших путей - найти для каждой упорядоченной пары вершин (*v, w)* любого пути от вершины *v* в вершины *w*, длина кот-го минимальна среди всех возможных путей от *v* к *w*.

Для решения применим алгоритм Флойда. Для этого пронумерует вершины графа последовательно от 1 до *n*.

**Алгоритм Флойда** использ-ет матрицу *А* размера *n**n*, в которой вычисляются длины кратчайших путей. Вначале **А[i, j]= C[i, j]** для всех ***i ≠ j*.** Если дуги ***i*→ *j*** нет, то *C[i, j]= ∞*. Каждый диагональный эл-нт матрицы ***А* = 0**.

Над матрицей *А* выполняется **n** итераций. После *к*-ой итерации *А[i, j]* содержит значение наименьшей длины путей из вершины *i*  в вершину *j*, которые не проходят через вершины с номером, большим ***к***,т.е. м/у концевыми вершинами пути из *i*  в *j* м находиться только вершины, номера которых <= ***к***. На***к***-ой итерации для вычисления матрицы А применяется формула:

***Ak*[*i*, *j*] = min( *Ak*−1[*i*, *j*], *Ak*−1[*i*, *k*]+ *Ak*−1[*k*, *j*]).**

Нижний индекс***к*** обозначает значение матрицы *А* после ***к***-ой итерации.

Графическая интерпретация формулы - на рис 17.5.

Для вычисления ***Ak*[*i*, *j*]** проводится сравнение величины ***Ak*−1[*i*, *j*]** с величиной (***Ak*−1[*i*, *k*] + *Ak*−1[*k*, *j*]).** Если путь через вершину ***к*** дешевле, чем ***Ak*−1[*i*, *j*],** то величина ***Ak*[*i*, *j*]** изменяется. На рис 17.6 - помеченный орграф, а на рис 17.7 – значения матрицы *А* после 3 итераций.

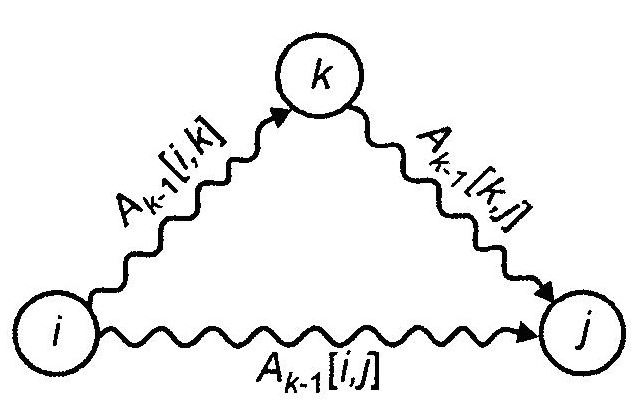


Рис 17.5 – Включение вершины *к* в путь от вершины *i* к вершине *j*

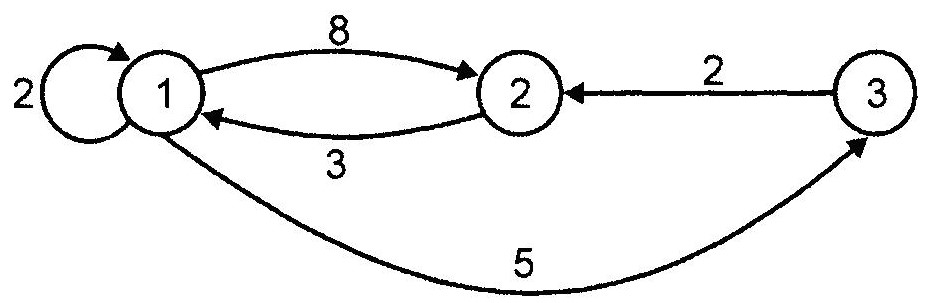


Рис 17.6 – Помеченный орграф

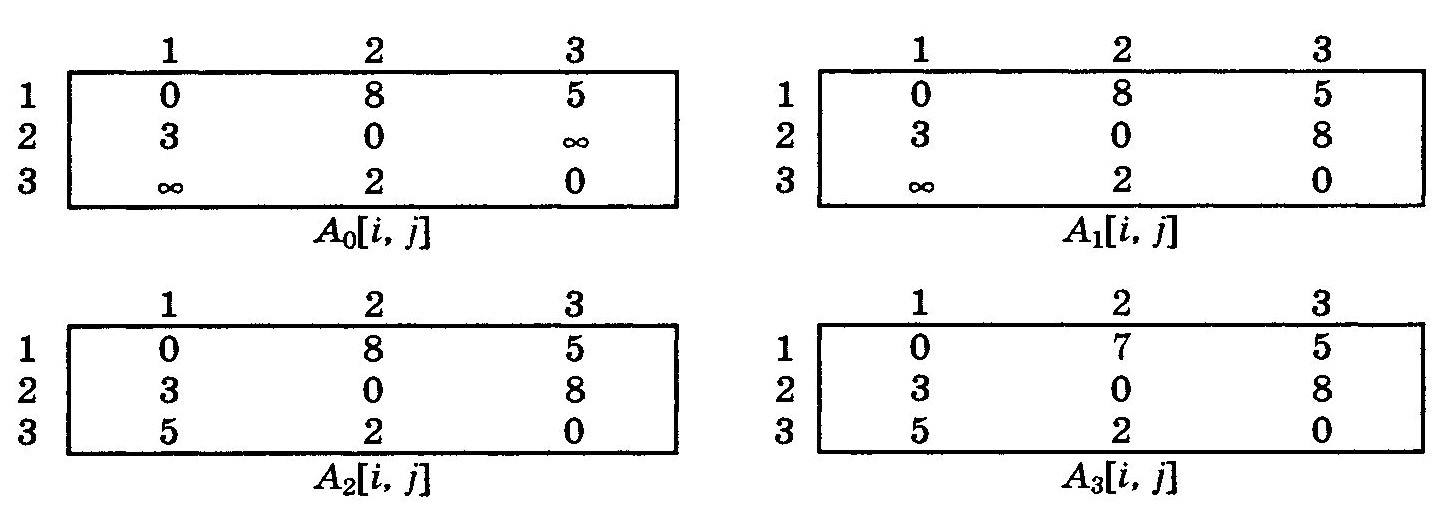


Рис 17.7 – Последовательные значения матрицы *А*

Равенства А*k*[*i*, *k*] = А*k*−1[*i*, *k*] и А*k*[*k*, *j*] = А*k*−1[*k*, *j*] означает, что на *к*-ой итерации эл-нты матрицы **А**, стоящие в *к*-ой строке и *к*-ом столбце, не изменяются. Процедура реализации **алгоритма 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**.

Вр выполнения этой проги имеет порядок *n*3. Т к алгоритм Дейкстры с использ-ием матрицы смежности находит кратчайшие пути от 1 вершины за вр порядка ***n*2**, то в случае применения алгоритма Дейкстры для нахождения всех кратчайших путей потребует вр порядка ***n*3**, т.е. получается такой же временной порядок, как и в алгоритме Флойда. Если *e*, кол-во дуг в орграфе, значительно меньше, чем *n*2, то применять алгоритм Дейкстры со списками смежности. Тогда вр нахождения кратчайших путей имеет порядок ***ne*\*log2 *n***, что луче алгоритма Флойда для больших разреженных графов.

**ТЕМА 18. ЦЕНТР ОРИЕНТРОВАННОГО ГРАФА. ОБХОДЫ ГРАФА**

# 18.1 Транзитивное замыкание

Задача - факт существования пути, длиной не меньше 1, от вершины *i* до вершины *j*. Для решения таких задач применяется **алгоритм Уоршелла**.

Пусть матрица стоимостей *С* совпадает с матрицей смежности для данного орграфа *G*, т.е. ***C[i, j]* = 1** только если есть дуга ***i*→ *j***, и ***C[i, j]* = 0** - если такой дуги нет. Требуется определить матрицу А такую, что ***А[i, j]* = 1** только тогда, когда есть путь от вершины *i* до верш *j* длиной не менее 1, иначе  ***А[i, j]* = 0**. Такую матрицу **А** называют **транзитивным замыканием**матрицы смежности.

На рис 18.1 - орграф, а на рис 18.2 – транзитивное замыкание матрицы смежности этого орграфа.

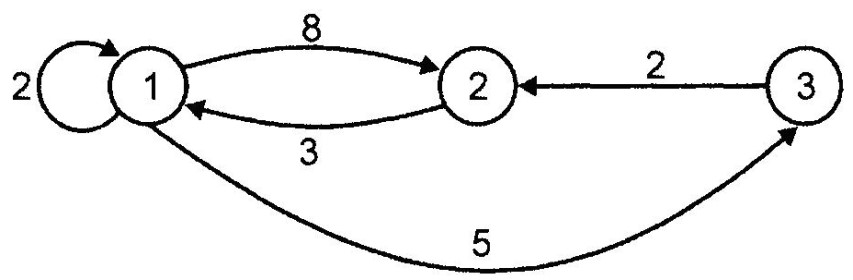


Рис 18.1 – Помеченный орграф

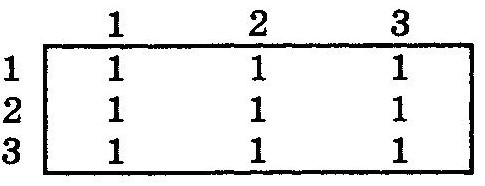


Рисунок 18.2 – Транзитивное замыкание матрицы смежности

Транзитивное замыкание м вычислить, применяя на *к*-ом шаге формулу к булевой матрице *А*:

***Ak*[*i*, *j*] = *Ak*−1[*i*, *j*] *or* (*Ak*−1[*i*, *k*] *and Ak*−1[*k*, *j*]).**

Эта формула устанавливает, что есть путь от вершины *i* до вершины j, проходящих через вершины с номерами, не превышающими *k*, только если:

1. Уже есть путь от вершины *i* до вершины *j*, который проходит через вершины с номерами, не превышающими ***к*-1.**

2. Есть путь от вершины *i* до вершины *к*, проходящий через вершины с номерами, не превышающими *к-1*, и путь от вершины *к* до вершины *j*, который проходит через вершины с номерами, не превышающими *к*-1.

# 18.2 Нахождение центра ориентированного графа

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* }.

**Центр орграфа *G*** - вершина с минимальным эксцентриситетом, т.е. вершина, для которой максимальное расстояние (длина пути) до других вершин минимально. Рассмотрим помеченный орграф на рис. 18 3.

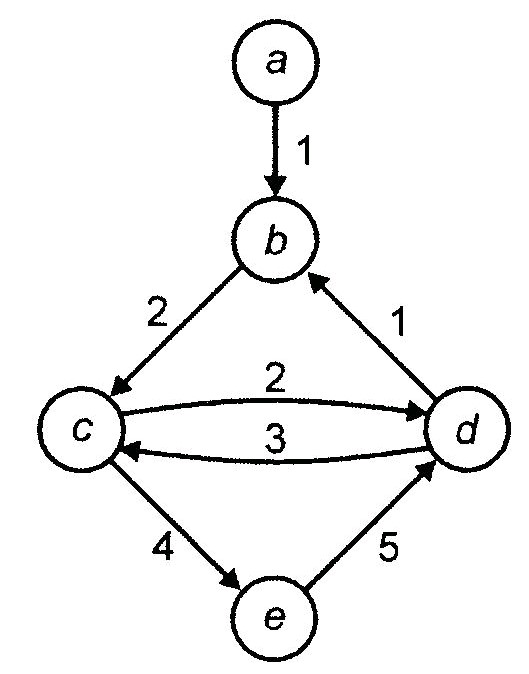


Рис 18.3 – Помеченный орграф

В этом графе вершины имеют эксцентриситеты: ПОЧЕМУ?????



Здесь центр данного орграфа - вершина *d*.

Пусть *С* – матрица стоимостей для орграфа *G*. Тогда центр орграфа находится по алгоритму:

1. Применить **алгоритм Флойда** к матрице *С* для вычисления матрицы *А*, содержащей все кратчайшие пути орграфа *G*.
2. Найти м**акс**имальное значение в каждом столбце ***i*** матрицы *А*. Это значение = эксцентриситету вершины ***i***.
3. Найти вершину с минимальным эксцентриситетом. Она и б центром графа *G*.

Матрица всех кратчайших путей для орграфа из рис 18.3 – на рис 18.4. Максимальные значения в каждом столбце приведены под матрицей.

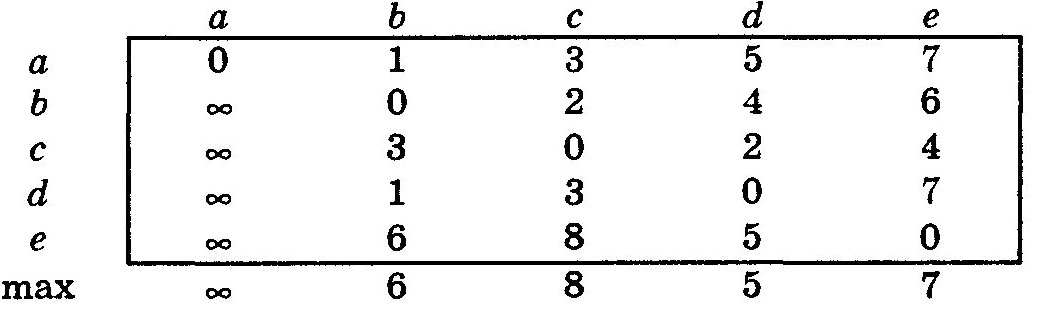


Рис 18.4 – Матрица кратчайших путей

# 18.3 Обход ориентированных графов

При решении задач с ориентированными графами, есть метод сис-матического обхода вершин и дуг орграфов – ***поиск в глубину***– обобщение м-да обхода дер в прямом порядке.

Пусть есть орграф *G*, в котором первоначально все вершины помечены меткой **unvisied** (не посещались). 1) Поиск в глубину начинается с выбора начальной вершины *v* графа *G*, для этой вершины метка **unvisited** меняется на метку **visited**. 2) Затем для каждой вершины, смежной с вершиной *v* и не посещалась ранее, рекурсивно применяется поиск в глубину. 3) Когда все вершины б посещены (которые м достичь из вершины *v*), поиск заканчивается. Если некоторые вершины не были посещены, то выбирается 1 из них и поиск повторяется. Этот процесс продолжается, пока обходом не б охвачены все вершины орграфа *G*.

Этот метод обхода вершин орграфа называется **поиском в глубину**, т к поиск непосещенных вершин идет в направлении вперед (вглубь), пока это возм. Напр-р, пусть *х* – последняя посещенная вершина. Для продолжения процесса выбирается какая-либо нерассмотренная дуга ***x*→*y***, выходящая из вершины *х*. Если вершина *y* ранее посещалась, то она помечается меткой *visited* и поиск начинается заново от вершины *y*. После того, как пройдены все пути, начинающиеся в вершине *y*, происходит возврат в вершину *х*, т.е. туда, откуда впервые была достигнута вершина *y*. Затем продолжается выбор нерассмотренных дуг, исходящих из вершины *x*, пока не б исчерпаны все такие дуги.

Для представления вершин, смежных с вершиной *v*, использ-ется список смежности ***L[v]***, а для определения ранее посещенных вершин – массив**mark**: его эл-нты б принимать только 2 значения (***visited,*  *unvisited)*.** Чтобы применить процедуру поиска в глубину к графу из *n* вершин, надо сначала присвоить всем эл-нтам массива **mark** значения ***unvisited***, затем начать поиск в глубину для каждой вершины, помеченной как***unvisited*** с помощью кода:

**for** v:= 1 **to** n **do**

mark[v]:= unvisited;

**for** v:= 1 **to** n **do**

**if** mark[v] = unvisited **then**

dfs(v);

Рекурсивная процедура **dfs** реализует метод поиска в глубину (см ниже). Здесь изменяются только значения массива ***mark****.*

**procedure** dfs(v: вершина);

**var**

w: вершина;

**begin**

mark[v]:= visited;

**for** каждаяя вершина w из списка L[v] **do**

**if** mark[w] = unvisited **then**

dfs(w)

**end**; // dfs

Рассмотрим метод поиска в глубину для графа на рис 18.5, начиная с верш *А*.

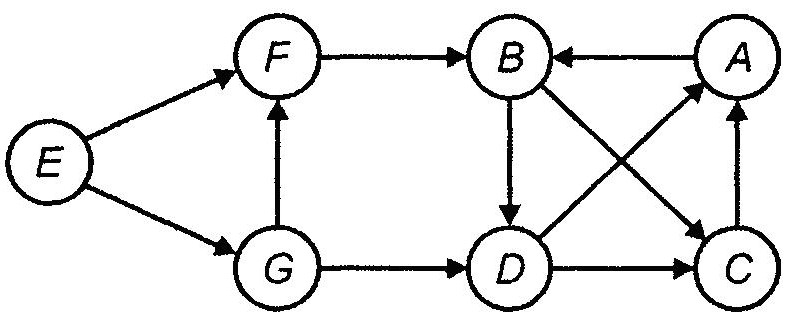


Рис 18.5 – Ориентированный граф

Алгоритм помечает вершину *А* как ***visited***и выбирает вершину *В* из списка смежности вершины *А*. Т к вершина В помечена как ***unvisited***, обход графа продолжается вызовом процедуры **dfs*(B)*.** Затем процедура помечает вершину *В* как ***visited***и выбирает первую вершину из списка смежности вершины *В*. В зав-ти от порядка представления вершин в списке смежности следующей рассматриваемой вершиной м б или вершина *С* или *D*. Пусть вершина *С* предшествует вершине *D*. Тогда происходит вызов **dfs*(С****)*. В списке смежности вершины *С* присутствует только вершина *А*, но она уже посещалась. Т к все вершины в списке смежности вершины *С* исчерпаны, то поиск возвращается в вершину *В*, откуда процесс поиска продолжается вызовом процедуры **dfs*(D)***. Вершины *А* и *С* из списка смежности вершины *D* уже посещались, поэтому поиск возвращается сначала в вершину *В*, а затем в вершину *А*. На этом первоначальный вызов **dfs*(А)*** завершен, но орграф имеет еше не посещенные вершины: ***E, F, G*.** Для продолжения обхода вершин графа выполняется вызов **dfs*(E).***

# 18.4 Глубинный остовный лес

В процессе обхода орграфа м-дом поиска в глубину только определенные дуги ведут к ранее не поссещенным вершинам. Такие дуги называют **дугами дер** и формируют для данного графа **глубинный остовный лес**, построенный м-дом поиска в глубину (рис 18.6 для графа из рис 18.5): сплошные линии - дуги дер. Именно они формируют лес, т к м-дом поиска в глубину к любой ранее не посещавшейся вершине м прийти только по 1 дуге.

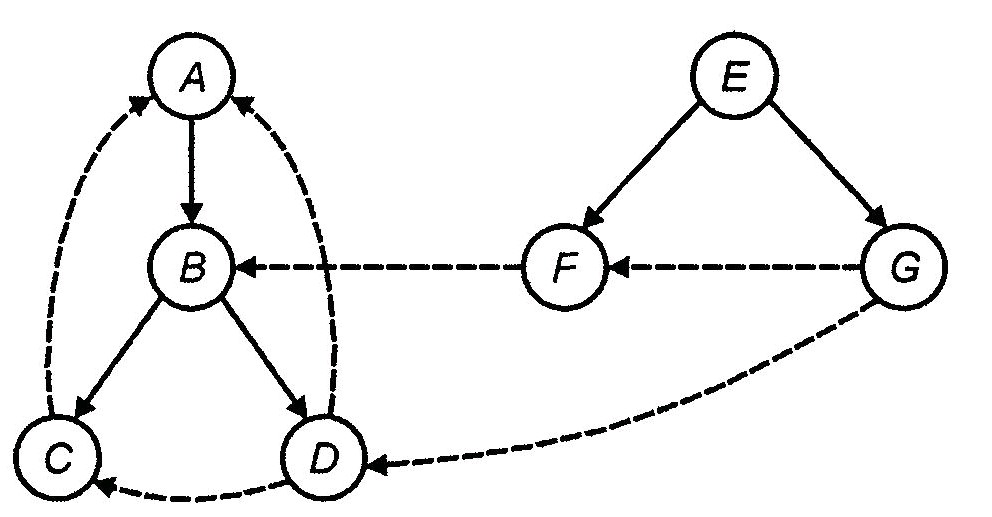


Рис 18.6 – Глубинный остовный лес

Кроме дуг дер, есть еще 3 типа дуг, определяемых в процессе обхода орграфа м-дом поиска в глубину: обратные, прямые и поперечные дуги.

1. ***Обратные дуги***, как дуга *C→A*: идут от потомков к предкам. Дуга из вершины в саму себя также является обратной дугой.
2. ***Прямые дуги*** - дуги, идущие от предков к истинным потомкам, но которые не являются дугами дер. На рис 18.6 прямых дуг нет.
3. ***Поперечные дуги -*** дуги, такие как *D→C* и *G→D*, соединяющие вершины, не являющиеся ни предками, ни потомками друг друга*.* Если при построении остовного дер сыновья одной вершины располагаются слева направо, то все поперечные дуги идут справа налево. Такое расположение вершин и деревьев помогает формировать остовный лес.

Дуги дер легко отличить от других, т.к. они получаются в процессе обхода графа как дуги, ведущие к тем вершинам, которые ранее не посещались. Если *v* – вершина орграфа, то всем потомкам вершины *v* присваиваются глубинные номера, не меньшие, чем номер, присвоенный вершине *v*. Фактически вершина *w* б потомком вершины *v* только тогда, когда выполняются неравенства: ***dfnumber(v)* *dfnumber(w)*  *dfnumber*(*v*)+**кол-во потомков вершины *v*. Очевидно, что прямые дуги идут от вершин с низкими номерами к вершинам с более высокими номерами, а обратные и поперечные дуги – от вершин с высокими номерами к вершинам с более низкими номерами.

**=============================================**

**ТЕМА 19. МОДЕЛЬ ВНЕШНИХ ВЫЧИСЛЕНИЙ.**

# СТРУКТУРЫ И АЛГОРИТМЫ

## 19.1 Модель внешних вычислений

Во всех ранее рассмотренных алгоритмах предполагалось, что объем входных данных позволяет использ-ть только оперативную память. Однако на практике - задачи (работа с большими БД), когда объемы обрабатываемых данных превышают ОП. В компах - устройства внешней памяти: жесткие диски, запоминающие устройства большой емкости. Однако характеристики доступа к устройствам внешней памяти отличаются от характеристик доступа к устройствам основной памяти. Чтобы повысить эффективность использования внешних устройств – ряд алгоритмов и структур данных.

В ЯП есть файловый тип данных для представления данных, хранящихся во вторичной памяти. При работе с файлами всегда действуешь в рамках ограничений способов доступа к файлам. ОС делит вторичную память на блоки одинакового размера. Размер блока зависит от типа ОС в пределах от 512 до 4096 байт. Файл м рассматривать как связанный список блоков, хотя ОС использ-ет древовидную организацию блоков, где блоки, составляющие файл, - листья дер, а каждый внутренний узел содержит указатели на множ-во блоков файла.

Базовая операция с файлами - перенос 1 блока в буфер, находящийся в основной памяти. **Буфер** - зарезервированная Обл в основной памяти, ее размер = размеру блока. Типичная ОС обеспечивает чтение блоков в порядке как они появляются в списке блоков, который содержит соотв-ий файл, т.е. сначала в буфер читается 1й блок файла, затем он заменяется на 2й блок, который записывается в тот же буфер, и т.д.

Каждый файл хранится в виде определенной последоват-ти **блоков**, а каждый блок содержит целое кол-во записей. Указатель считывания всегда указывает на 1 из записей в блоке, который в данный момент находится в буфере. Когда этот указатель должен переместиться на запись, отсутствующую в буфере, надо прочитать следующий блок файла. Аналогично, процесс записи файла – процесс создания файла в буфере. Когда записи заносятся в файл, фактически они помещаются в буфер для этого файла. Если очередная запись не помещается в буфер целиком, содержимое буфера копируется в свободный блок вторичной памяти, который присоединяется к концу списка блоков для данного файла. После этого буфер считается свободным для помещения в него очередной порции записей.

## 19.2 Особенности операций с внешней памятью

Природа устройств внешней памяти: вр поиска блока и чтения его в основную память > вр обработки данных в этом блоке.

Пусть есть блок из 1000 целых чисел на диске вращение со скоростью 1000 об./мин). Вр позиционирования считывающей головки над дорожкой с этим блоком *(вр установки головок)* + вр ожидания пока требуемый блок сделает оборот и окажется под головкой (*вр ожидания*) = 100 миллисек. Процесс записи блока в определенное место во вторичной памяти - столько же вр. При этом за те же 100 миллисекунд машина выполняет 100 000 команд. Этого вр достаточно выполнить простую обработку 1000 целых числе, когда они находятся в основной памяти.

Оценивая вр работы алгоритмов, в которых использ-ются данные, хранящиеся в виде файлов, надо учитывать кол-во обращений к блокам, т.е. ск-ко раз блок считывается в основную память или записывается во вторичную память. Такая операция называется **доступом к блоку.** Т к размер блока фиксирован в ОС, нет возмсти ускорить работу алгоритма, увеличив размер блока и сократив тем самым кол-во обращений к блокам. Поэтому мера качества алгоритма работы с внешней памятью – кол-во обращений к блокам.

## 19.3 Организация данных в файлах

Рассмотрим файл как последоват-ть записей (каждая запись состоит из 1й и той же совокупности полей). Поля м иметь либо фиксированную, либо переменную длину.

Будем работать с файлами с фиксированной длины записями (использ-ют в с-мах управления БД для хранения данных сложной структурой). Для работы с файлами применяют операторы ( +способ реализации указны):

1.***INSERT***вставляет запись в определенный файл. Вставку в файл – присоединение соответствующей записи к концу файла.

*2.* ***DELETE***удаляет из определенного файла все записи, содержащие указанные значения в указанных полях. При удалении ищутся записи, чьи поля соответствуют условиям удаления, а затем выбирается способ удаления требуемых записей.

*3.* ***MODIFY*** изменяет все записи в определенном файле, задав указанные значения определенным полям в тех записях, которые имеют указанные значения в других полях. При изменении записей надо просмотреть файл, проверить каждую запись и решить, соотв-ет ли она заданным условиям. Если соотв-ет, в запись вносятся требуемые изменения.

*4.* ***RETRIEVE*** отыскивает все записи, содержащие указанные значения в указанных полях. поиск записи с указанными значениями в определенных полях - полный просмотр файла и проверка каждой его записи на наличие в ней заданных значений.

«-» последовательного файла: медленно выполняются операторы с такими файлами: каждая операция требует прочтения всего файла + перезаписи некоторых блоков. Но есть способы организации файлов: обращаться к записи, считывая в основную память лишь небольшую часть файла. Для этого у каждой записи д б **ключ** – совокупности полей, которая уникально идентифицирует каждую запись. Поиск записи, когда заданы значения ее ключевых полей - типичная операция. Еще 1 атрибут быстрого выполнения операцией с файлами - прямой доступ к блокам. Рассмотрим неск-ко способов **ускорения** операций с файлами.

## 19.4 Хешированные файлы

**Хеширование** – метод обеспечения быстрого доступа к инфе, хранящейся во вторичной памяти. Основная идея этого м-да подобна открытому хешированию (см выше). Записи файла распределяются м/у **сегментами**, каждый из них состоит из связного списка 1 / нескольких блоков внешней памяти.

Есть таблица сегментов, содержащая **В** указателей – по 1 на каждый сегмент. Каждый указатель в таблице сегментов - физический адрес первого блока связного списка блоков для соотв-его сегмента. Сегменты пронумерованы от 1 до *В*. Хеш-ф-ция **h** отображает каждое значение ключа в 1 из целых чисел от 1 до *В*. Если *х* – ключ, то **h(x)** - **№** сегмента, содержащий запись с ключом *х* (если такая запись есть). Блоки, составляющие каждый сегмент, образуют связный список. Т о, заголовок **i**-ого блока содержит указатель на физический адрес (***i*+1)**-ого блока. Последний блок сегмента содержит в своем заголовке **nil**-указатель (рис 19.1). Note: эл-нты, хранящиеся в 1 блоке сегмента, не надо связывать друг с другом указателями, м/у собой связывать нужно только блоки.

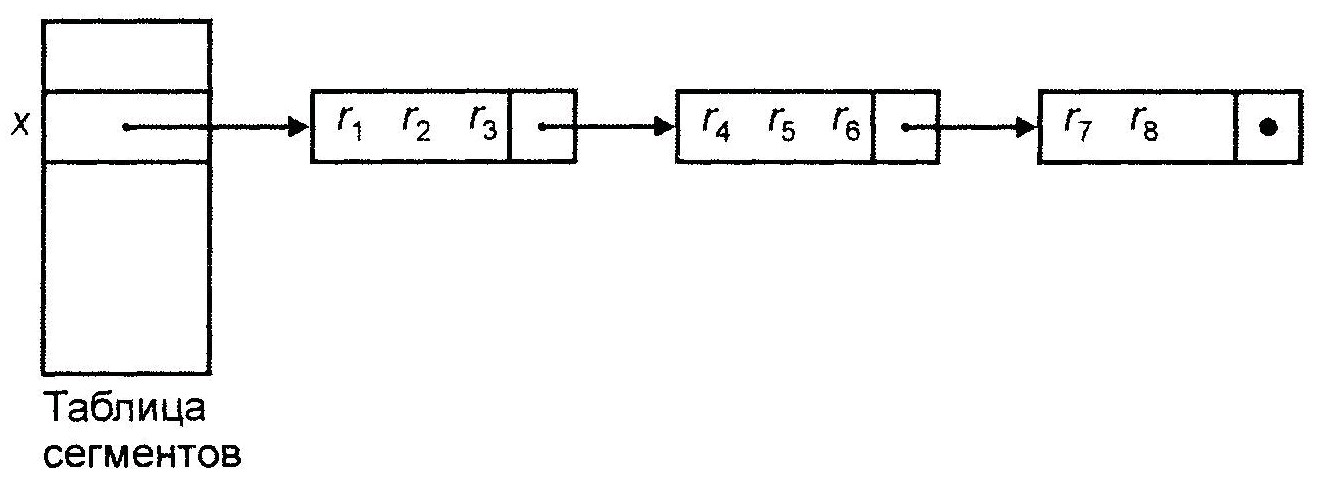


Рис 19.1 – Сегменты, состоящие из связанных блоков

Если размер таблицы сегментов невелик, ее м хранить в основной памяти, иначе ее м хранить последовательным способом в отдельных блоках. Если надо найти запись с ключом *х*, вычисляется ***h(x)*** и находится блок таблицы сегментов, содержащий указатель на 1й блок сегмента ***h(x****)*. Затем последовательно считываются блоки сегмента ***h(x****)*, пока не обнаружится блок, содержащий запись с ключом ***х***. Если исчерпаны все блоки в связном списке для сегмента *h(x)*, значит *х* не является ключом ни 1 из записей.

Такая стр-ра эффективна, если в выполняемом операторе указывать значения ключевых полей.

Среднее кол-во обращений к блокам для выполнения оператора, в котором указан ключ записи, = среднему количеству блоков в сегменте, которое **= n/bk**, если ***n*** – кол-во записей, блок содержит ***b*** записей, а *k –* кол-во сегментов. При такой организации данных операторы, использующие значения ключей, выполняются в ***к*-**раз быстрее, чем в неорганизованном файле.

Чтобы вставить запись с ключом со значением ***х***, надо сначала проверить наличие в файле записи с таким значением ключа. Если есть, то сообщение об ошибке, т к ключ уникальным образом идентифицирует каждую запись. Если записи с ключом *х* нет, новая запись вставляется в 1й блок цепочки для сегмента ***h(x)***, в который ее удается вставить.

Если запись нельзя вставить ни в 1 из блоков сегмента ***h(x)***, файловой с-ме дается команда найти новый блок, в который б помещена эта запись. Затем новый блок добавляется в конец цепочки блоков сегмента ***h(x****)*.

Для **удаления** записи с ключом *х:* сначала найти эту запись, а затем установить ее бит удаления.

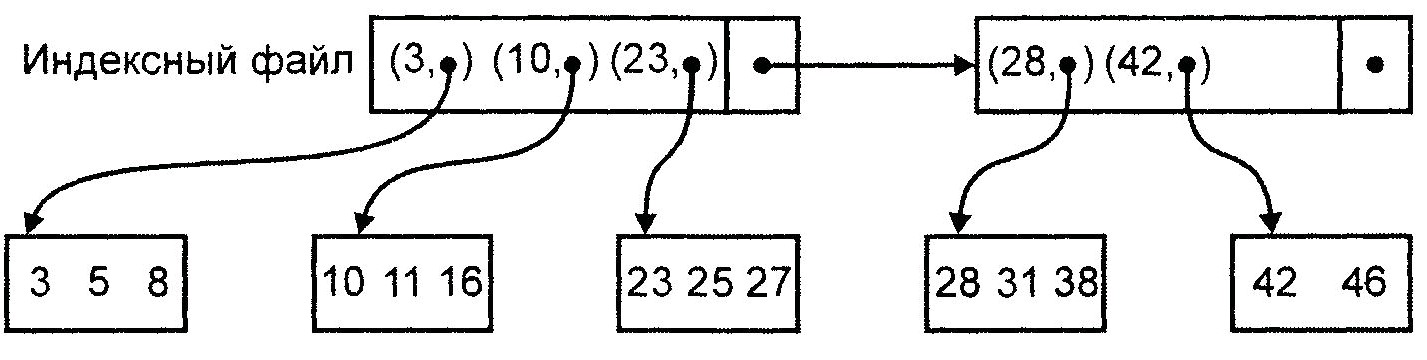
Удачная организация файлов с хешированным доступом требует лишь небольшое число обращений к блокам при выполнении каждой операции с файлами. Если выбрана удачная ф-ция хеширования, а кол-во сегментов приблизительно = к-во записей в файле, деленное на кол-во записей, которые м поместиться в 1 блоке, тогда средний сегмент состоит из 1 блока.

Если не учитывать обращения к блокам для просмотра таблицы сегментов, операция поиска данных, основанного на ключах, потребует 1 обращения к блоку, а операция вставки, удаления или изменения потребуют 2х обращений к блокам. Если среднее кол-во записей в сегменте намного > кол-ва записей, которые м поместить в 1 блоке, м периодически реорганизовывать таблицу сегментов, удваивая кол-во сегментов и деля каждый сегмент на 2 части.

## 19.5 Индексированные файлы

Еще 1 способ организации файла записей - поддержание файла в порядке отсортированном по значениям ключей: файл м просматривать как словарь / телефонный справочник, когда просматриваются лишь заглавные слова на каждой странице. Чтобы облегчить процедуру поиска, м создать 2й файл, называемый **разреженным индексом**, который состоит из пар ***(x, b****)*, где ***x*** – значение ключа, **а *b*** – физический адрес блока, в котором значение ключа 1й записи = *х*. Разреженный индекс отсортирован по значениям ключей.

Рис 19.2 - файл с инфой и соотв-ий ему файл разреженного индекса.



## Рисунок 19.2 – Файл и его разреженный индекс

Предполагается, что 3 записи основного файла или 3 пары индексного файла помещаются в 1 блок. Записи основного файла представлены только значениями ключей, которые здесь – целочисленные величины. Чтобы найти запись с заданным ключом ***х*:** сначала просмотреть индексный файл, отыскивая пару ***(x, b)*.** В действительности ищется наибольшее **z**, такое, что ***z**x*** и далее находится пара **(*z, b)*.** Тогда ключ *х* 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 м поместить: 1) столько записей, сколько туда помещается, 2) оставить в нем вакантные места для добавления записей. «+» такого подхода: ниже вероятность переполнения блока, куда вставляются новые записи, иначе – 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 как это сделано в основном файле, в блоках, содержащих индексный файл, м оставить какое-то место для последующего роста.

Пусть есть отсортированный файл записей, хранящихся в блоках

***B*1,В2, …, В*m*.** Для вставки новой записи в отсортированный файл используем индексный файл, с помощью которого определяется, блок с каким номером должен имеють новую запись. Если новая запись помещается в блок **В*i***, она заносится в корректной последоват-ти. Если новая запись – 1ая записью в **В*i***, тогда идет корректировка индексного файла.

Если новая запись не помещается в блок В*i*, то: перейти на блок **В*i*+1**, и узнать, м ли последнюю запись **В*i*** переместить в начало **В*i* +1**. Если м, последняя запись перемещается в **В*i* +1**, а новую запись вставить на подходящее место в В*i*. Тогда нужно откорректировать вход индексного файла для **В*i* +1** и **В*i***.

Если блок **В*i* +1** также заполнен или если **В*i*** - последний блок (*i=m*) – из файловой системы получить новый блок, куда вставляется новая запись (блок размещантсяся за блоком ***Bi*)**. Далее – процедура вставки в индексном файле записи для нового блока.

**19.6 Несортированные файлы с плотным индексом**

Еще 1 способ организации файла записей - сохранение произвольного порядка записей в файле и создание другого файла, с его помощью б отыскиваться требуемые записи. Этот файл называется **плотным индексом***. Он* состоит из пар ***(x, p)*,** где ***р*** – указатель на запись с ключом *х* в основном файле. Эти пары отсортированы по значениям ключа. Структуру, подобную разреженному индексу, м использ-ть для поиска ключей в плотном индексе. Здесь плотный индекс – для поиска в основном файле записи с заданным ключом. Если надо вставить новую запись, ищется последний блок основного файла и туда она вставляется. Если последний блок полностью заполнен, то получаем новый блок из файловой системы. Одновременно вставляется указатель на соответствующую запись в файле плотного индекса. Для удаления записи в ней просто устанавливаем бит удаления и удаляется соотв-ий вход в плотном индексе.

# ТЕМА 20. В-ДЕРЕВЬЯ И ОПЕРАЦИИ НАД НИМИ

## 20.1 Внешние деревья поиска

Для представления внешних файлов использ-ют древовидные Стр-ы Дан. В-дер - обобщение бинарных деревьев, стандартный способ организации индексов в с-мах БД. Обобщением дер двоичного поиска является *m*-арное дер, где каждый узел имеет не более *m* сыновей. И для деревьев бинарного поиска, и для *m*-арного дер выполняется условие: если *n*1 и *n*2 - оба сына 1го узла, и *n*1 - слева от *n*2, то все эл-нты вниз от *n*1 меньше эл-тов, исходящих вниз от *n*2. Операции поиска, вставки и удаления эл-тов для *m*-арного дер поиска – путем обобщения аналогичных операция для деревьев бинарного поиска.

Но для внешних деревьев поиска есть проблема хранения записей в файлах, когда файлы хранятся в виде блоков внешней памяти. Применение идеи разветвленного дер - представлять узлы как физические блоки. Внутренний узел содержит указатели на своих *m* сыновей, а также *m*-1 ключевых значений, которые разделяют потомков этих сыновей. Листья также - блоки, а они имеют записи основного файла.

Если использ-ть дер двоичного поиска из *n* узлов для представления файла, хранящегося во внешней памяти, то для поиска записи в таком файле надо log**2*n*** обращений к блокам. Если вместо бинарного дер поиска использ-ть для представления файла *m*-арное дер поиска, то для поиска записи в таком файле надо log *m n* обращений к блокам. При n = 1 000 000 дер бинарного поиска - 20 обращения к блокам, а 128-арное дер поиска – 3 обращения к блокам. Но нельзя сделать *m* очень большим: чем > *m*, тем > размер блока. + считывание и обработка более крупного блока занимают > вр, поэтому есть оптимальное значение *m*, чтобы минимизировать вр для просмотра внешнего *m*-арного дер поиска.

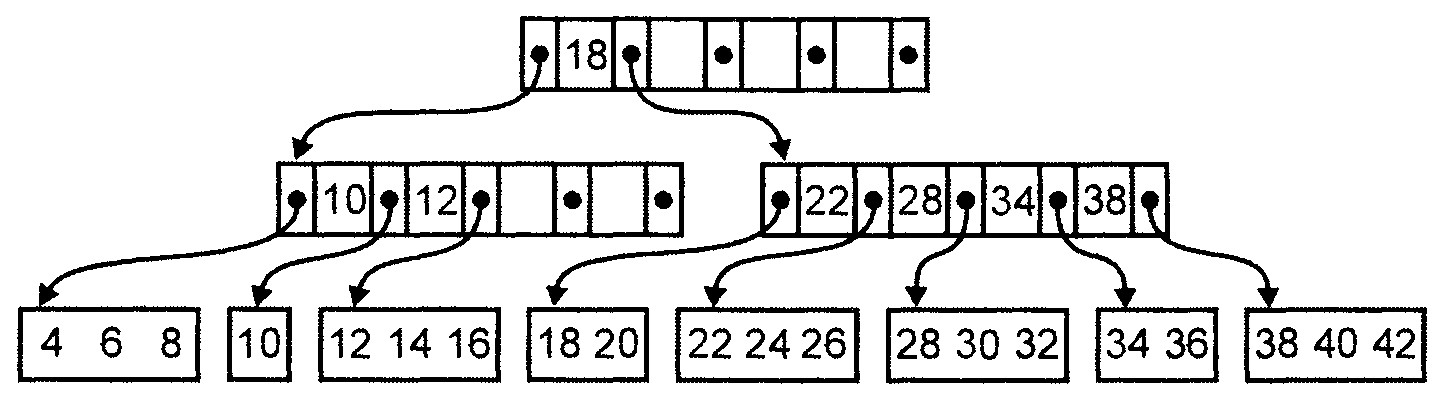
## 20.2 В-деревья

– особый вид сбалансированного *m*-арного дер для выполнения операций поиска, вставки и удаления записей из внешнего файла с гарантированной производит-стью для самой неблагоприятной ситуации.

**В-дер порядка m** - *m*-арное дер поиска, характеризующееся св-вами:

1. Корень либо является листом, либо имеет хотя бы 2 сына.
2. Каждый узел кроме корня и листьев, имеет от *m/2* до *m* сыновей.

3. Все пути от корня до любого листа имеют одинаковую длину. На рис 20.1 - В-дер порядка 5: предполагается, что в блоке листа помещается до 3х записей.



## Рис 20.1 – В-дер порядка 5

В-дер – как иерархический индекс, где каждый узел занимает блок во внешней памяти. Корень В-дер - индекс 1го уровня. Каждый нелистовой узел на В-дер имеет форму

(**р0, к1, р1, к2, р2,... к*n*, р*n*)**,

где **р*i*** - указатель на ***i*-**ого сына, **0*i**n***, а **к*i*** – ключ, **1*i* *n***. Ключи в узле упорядочены, поэтому **к1к2 ...к*n***. Все ключи в поддер, на которое указывает р0 , меньше, чем к1.

При **1*i* *n*** все ключи в поддер, на которое указывает **р*i***, имеют значения, не меньшие, чем **к*i***, и меньшие, чем к*i*+1. Все ключи в поддер, на которое указывает р*n* , имеют значения, не меньшие, чем к*n* .

Неск-ко способов организации листьев (записи основного файла хранятся только в листьях, и каждый лист занимает 1 блок):

### **20.3 Операторы на В-дер**

**1) Поиск записей.** Найти запись ***r*** со значением ключа ***х***, надо проследить путь от корня до листа, который содержит *r* (если эта запись есть в файле). При прохождении этого пути последовательно считываются из внешней памяти в основную внутренние узлы (р0, к1, р1, к2, р2,... к*n*, р*n*) и вычисляется положение ***х*** относительно ключей *k*1,*k*2, ...*kn*. Если к*i**x*к*i*+1, то в основную память считывается узел, на который указывает **р*i***, и описанный процесс повторяется. Если *x*к1, для считывания в основную память использ-ется указатель р0. Если *x*к*n*, то использ-ется Р*n*.

Когда в результате этого процесса попадают на какой-то лист, то ищут запись со значением ключа ***х***. Если кол-во входов в узле невелико, то в узле м использ-ть линейный поиск, иначе – использ-ть двоичный поиск.

**2) Вставка записей*.*** Вставить в В-дер запись ***r*** со значением ключа ***х***: сначала – процедура поиска, чтобы найти лист *L*, которому принадлежит запись *r*. Если в *L* есть место для новой записи, то она вставляется в *L* и тогда не надо внесений изменений в предков листа *L*.

Если в блоке листа *L* нет места для записи ***r***, у файловой системы запрашивается новый блок *L****`*** и из *L* в *L`* перемещается последняя 1/2 записей. При этом *r* вставляется в требуемом порядке в *L* или *L`*. Пусть узел *Р* - родитель узла *L*. *Р* известен, т к процедура поиска отследила путь от корня к листу *L* через узел *Р*. Теперь м рекурсивно применить **процедуру вставки**, чтобы разместить в *Р* ключ *k`* и указатель *l`* на *L`*. *k`* и *l`* вставляются сразу же после ключа и указателя для листа *L*. Значение *k`* - наименьшее значение ключа в *L`*.

Если *Р* уже имеет **m** указателей, вставка ***k`*** и ***l`*** в *P* приведет к расщеплению P и потребует вставки ключа и указателя в узел родителя *P*. Эта вставка м произвести эффект домино, распространяясь на предков узла *L* в направлении корня, вдоль пути, который уже был пройден процедурой поиска. Это м привести расщеплению корня, тогда создается новый корень, а 2 половины старого корня становятся его сыновьями. Это (!) ситуация, когда узел м иметь менее *m/2* потомков.

**3) Удаление записей***.* Удалить запись ***r*** со значением ключа *х*: а) найти лист *L* с записью *r.* б) если она есть, она удаляется из *L*. Если r - первая запись в *L*, после этого – переход в узел *P* (родитель листа *L)*, чтобы установить новое значение первого ключа для *L*. Если *L* - первый сын узла *P*, то 1й ключ *L* не зафиксирован в *P*, а появляется в 1ом из предков *P*. Т о, надо распространить изменение в наименьшем значении ключа *L* в обратном направлении вдоль пути от *L* к корню.

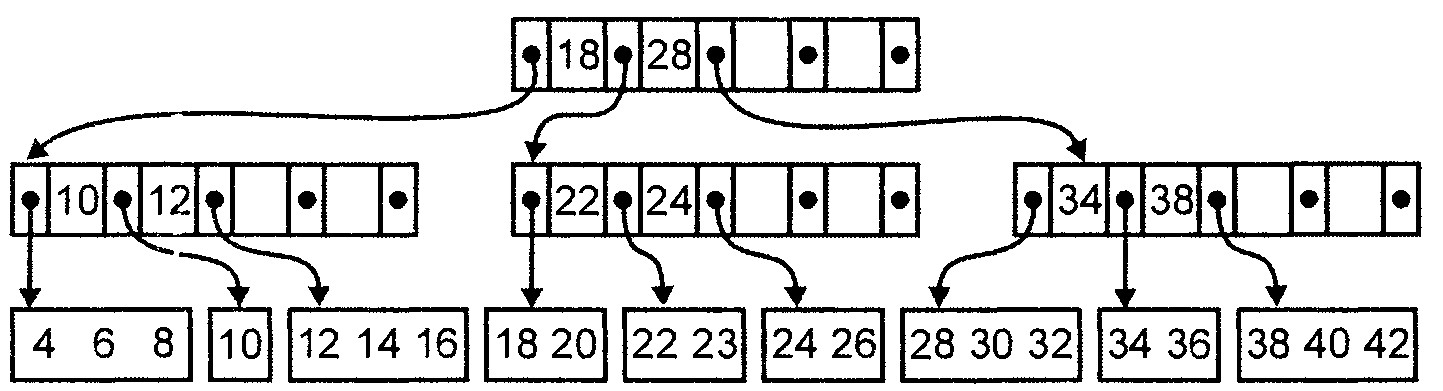
Если блок листа *L* после удаления записи - пустой, он отдается файловой с-ме. После этого корректируем ключи и указатели в *P*, чтобы отразить факт удаления листа *L*. Если кол-во сыновей узла *P* оказывается теперь < *m/2*, проверяется узел *P`*, расположенный в дер на том же уровне прямо слева или справа от *P*.

Если узел *P`* имеет хотя бы *m/2+2* сыновей, ключи и указатели распределяются поровну м/у *P* и *P`* так, чтобы оба узла имели хотя бы по *m/2* потомков, сохраняя упорядоченность записей. Затем надо изменить значения ключей для *P* и *P`* в родителе *P* и, если надо, рекурсивно распространить воздействие внесенного изменения на всех предков узла *P*, на которых это изменение отразилось.

Если у *P`* - ровно *m/2* сыновей, *P* и *P`* объединяют в 1 узел с 2(*m*/2)-1 сыновьями. Затем надо удалить ключ и указатель на *P`* из родителя для *P`*. Это удаление м выполнить с помощью рекурсивного применения процедуры удаления.

Если обратная волна воздействия удаления доходит до самого корня, возм, надо объединить только 2х сыновей корня. Тогда новым корнем становится объединенный узел, а старый корень м вернуть файловой с-ме. Высота В-дер уменьшается на 1.

Рассмотрим эти операторы на пр-ре В-дер на рис 20.1. Вставка записи со значением ключа 23 порождает В-дер (рис 20.2). Чтобы вставить эту запись, надо расщепить блок, содержащий записи с ключами 22, 23, 24 и 26. Пусть в 1 блок помещается не более 3 записей. 2 меньших остаются в этом блоке, а 2 больших помещаются в новый блок. Пару указатель-ключ для нового узла надо вставить в родителя, который в таком случае расщепляется, т к не м имеють 6 указателей. Корень принимает пару указатель-ключ для нового узла, но корень не расщепляется, т.к. имеет достаточную емкость.



## Рис 20.2 – В-дер после вставки в него записи

Удаление записи с ключом **10** из В-дер на рис 20.2, приводит к В-дер на рис 20.3. В этом случае блок, содержащий запись с ключом 10, отбрасывается. У его родителя теперь 2 сына, а у правого брата этого родителя минимальное кол-во сыновей – 3. Т о, в результате объединения родителя и его брата получается 1 узел с 5 сыновьями.

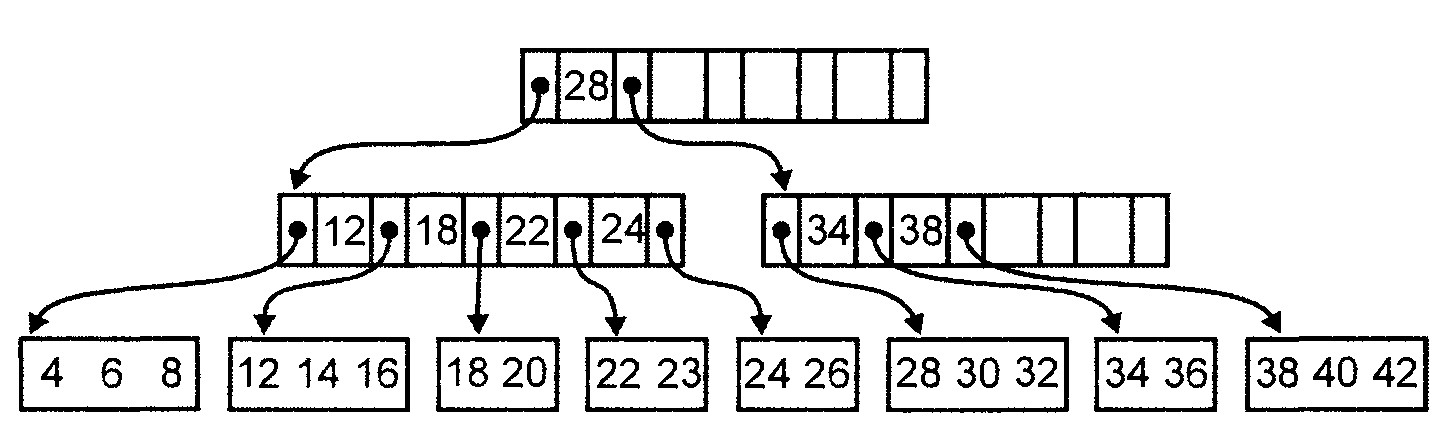


Рис 20.3 – В-дер послед удаления записи

### **20.4 Сравнение методов**

Методы организации внешних файлов – были рассмотрены **хеширование**, **разреженные индексы** и **В-деревья**.

На практике сравнивают кол-во обращений к блокам для разных методов, связанное с выполнением операций с файлами.

**Хеширование** - самый быстрый из методов, требует 2х обращений к блокам по каждой операции, не считая обращений для просмотра самой таблицы сегментов, если кол-во сегментов велико, чтобы типичный сегмент использ-ал только 1 блок. Но в хешировании сложно обращаться к записям в отсортированной последоват-ти. Разреженный индекс для файла из ***n*** записей, позволяет выполнять операции с файлами, ограничиваясь использ-ием Пр-рно **2 +log2(*n*/*bb*`)** обращений к блокам в случае двоичного поиска, где *b* – кол-во записей, помещающихся в 1 блок, а *b`* – кол-во пар ключ-указатель, помещающихся в 1 блок для индексного файла.

**В-деревья** позволяют выполнять операции с файлами с использ-ием **2 + log2*m*/2(*n*/*b*)** обращений к блокам, где **m** – максимальное кол-во сыновей у внутренних узлов, что приблизительно = b`. Как разреженные указатели, так и В-деревья допускают обращение к записям в отсортированной последоват-ти.

В-деревья - более популярны как средство доступа к файлам в с-мах БД. Причина: способность обрабатывать запросы, запрашивая записи с ключами, относящимися к определенному диапазону. Разреженный индекс обрабатывает подобные запросы также эффективно, но хуже, чем В-деревья. Кроме того, В-деревья удачно использ-ть в качестве вторичных указателей, когда ключи в действительности не определяют уникальную запись.

Все перечисленные методы эффективнее обычного последовательного просмотра файла. Временные различия м/у ними невелики.

**====== ФСЁ ===========================**

YES!!!!!!!!!!!!!!!