МОСКОВСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
ПУТЕЙ СООБЩЕНИЯ (МИИТ)

Институт управления и информационных технологий

Кафедра «Вычислительные системы и сети»

Я.М. ГОЛДОВСКИЙ

Структуры и организация данных

Методические указания

Москва - 2005

МОСКОВСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
ПУТЕЙ СООБЩЕНИЯ (МИИТ)

Институт управления и информационных технологий

Кафедра «Вычислительные системы и сети»

Я.М. ГОЛДОВСКИЙ

УТВЕРЖДЕНО
ционно-издательским
оветом университета

Структуры и организация данных

**Методические указания к практическим занятиям
по дисциплине «Структуры и организация данных»
для студентов специальности
«Вычислительные машины, комплексы, системы и сети»
ИУИТ**

Москва – 2005

УДК 681.3

Г-60

Голдовский Я.М. Структуры и организация данных. Методические указания к лабораторным работам по дисциплине «Структуры и организация данных» – М.: МИИТ, 2005. – с.30.

Методические указания предназначены для студентов третьего курса специальности «Вычислительные машины, комплексы, системы и сети» и необходимы для выполнения лабораторных работ по дисциплине «Структуры и организация данных». Приводится необходимая информация для подготовки к работам, порядок выполнения, варианты заданий.

Учебно-методическое издание

Голдовский Яков Михайлович

Структуры и организация данных
Методические указания

Подписано в печать - 29.12.05.
Усл.печ.л. - 2,0.
Формат - 60×84/16 Тираж - 100.
Заказ - 810.
Изд.№307-05

127994, Москва, ул. Образцова, 15.
Типография МИИТ

Содержание

Лабораторная работа №1

МЕТОДЫ СОРТИРОВКИ И ПОИСКА ДАННЫХ

Первая лабораторная работа посвящена методам сортировки данных в списке. Цель работы: изучение основных методов сортировки списка и поиск методом половинного деления.

Теоретический материал.

1.1 МЕТОД «ПУЗЫРЬКА»

Сортировка производится попарным сравнением элементов. Если элементы стоят в нужном порядке, то сравнивают следующую пару, иначе меняют местами. Когда пройдет весь список, начинаем 2-ой проход по этому же списку и так до тех пор, пока за весь проход не будет ни одной перестановки.

```
For i:=1 to n-1 do
   Begin
      For j:=1 to n-1 do
            If M[j]>M[j+1] then
               Begin
                  D:=M[j];
                  M[j]:=M[j+1];
                  M[j+1]:=D;
               End;
   End;
```

Что бы ускорить процесс сортировки в программу надо ввести переменную-флаг, которая покажет, была ли в цикле хотя бы одна перестановка, если перестановок не было, то массив отсортирован.

1.2 МЕТОД «ЭКСТРЕМУМОВ».

При сортировке по убыванию мы находим максимальный элемент, ставим его в новый отсортированный массив. Из оставленных

элементов снова выбираем максимальный и т.д. пока из исходного массива не будут выбраны все элементы.

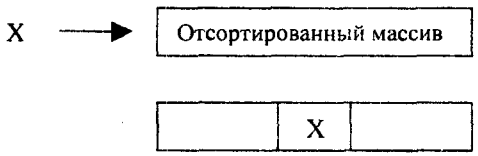
```
For j:=1 to n-1 do
   Begin
      D:=M[j];
      For i:=j+1 to n do
            If (D<M[i]) then
               Begin
                  D:=M[i];
                  M[i]:=M[j];
                  M[j]:=D;
               End;
   End;
```

Сортировка по возрастанию аналогична.

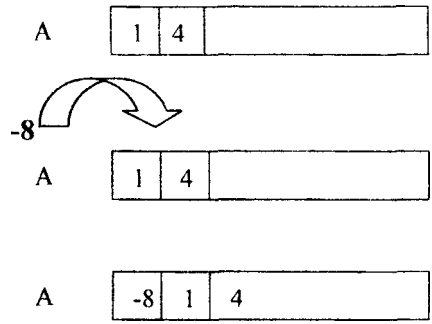
1.3 МЕТОД ПРОСТОЙ ВСТАВКИ.

В базах данных часто встречается ситуация когда информация уже отсортирована и в нее вносится новая запись, в этом случае быстрее всего сработает метод простой вставки. Он не требует полной сортировки, а позволяет раздвинуть уже отсортированный массив и вставить новый элемент на нужное место.

1) Пусть дан пустой массив. Вставим в него число.



2) Вставим второе число, если оно меньше первого, то первое сдвинем и т.д.
3) Вставляем все необходимые числа, при необходимости сдвигая и раздвигая массив.



1.4 ПОИСК МЕТОДОМ ПОЛОВИННОГО ДЕЛЕНИЯ.

Допустим, существует линейный список, отсортированный по возрастанию. Для того чтобы найти нужное нам число, сравним его с серединой диапазона. Если искомое число больше стоящего в середине, продолжаем поиск справа от середины, если меньше - слева. В любом случае, отбрасываем половину диапазона, в оставшемся опять ищем середину и так, постепенно сужая поиск, мы находим искомое число, или убеждаемся, что его нет.

Рабочее задание

Разработать и отладить программу реализующую:

1) Ввод исходного списка (размерность массива задается пользователем).

2) Сортировку списка методом «пузырька», ввести переменную-флаг, подсчитать количество перестановок.

3) Сортировку массива с помощью метода «экстремумов», подсчитать количество сравнений.

4) Сортировку массива методом «простой вставки», применяя поиск методом половинного деления.

5) Меню для выбора варианта сортировки

Варианты

Если номер студента в списке группы четный, то массив сортируется по возрастанию, нечетный по убыванию.

Лабораторная работа №2

ЛИНЕЙНЫЕ СПИСКИ.

Вторая лабораторная работа посвящена методам обработки данных в распространенных видах линейных списков: стек, очередь, дек. Цель работы: изучение распространенных частных случаев линейных списков и получение навыков использования динамического определения памяти и единого адресного пространства при последовательном распределении памяти.

Теоретический материал.

Линейный список - это структура данных, в которой структурные свойства любого элемента сводятся к его положению между двумя соседними элементами.

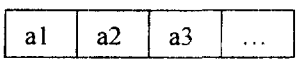
Так, на рисунке, элемент «2» расположен между элементами «3» и «4».

ОПЕРАЦИИ НАД ЛИНЕЙНЫМИ СПИСКАМИ (ОБЩИЙ СЛУЧАЙ).

1. Включение элемента в список
2. Удаление элемента из списка
3. Поиск элемента в списке
4. Сортировка
5. Соединение двух и более списков в один
6. Разделение одного линейного списка на два и более линейных списков.

ПОСЛЕДОВАТЕЛЬНОЕ РАСПРЕДЕЛЕНИЕ ПАМЯТИ.

При последовательном распределении в памяти список размещается в смежных ячейках памяти.



Достоинства последовательного размещения:

1. Возможность непосредственной адресации нужной ячейки, так как номер ячейки легко вычисляется.

2. Экономия памяти: элемент содержит только информационное поле.

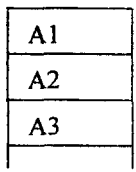
Недостатки последовательного размещения:

1. Сложность вставки и удаления элемента.

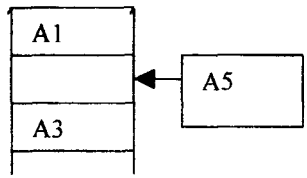
2. Необходимость заранее объявлять размер списка, это приводит к не экономному использованию памяти.

3. Фрагментация памяти.

1)



2)



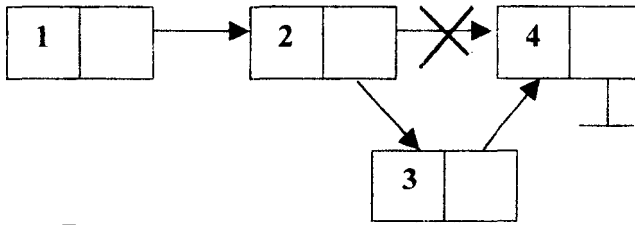
СВЯЗАННОЕ РАСПРЕДЕЛЕНИЕ ПАМЯТИ.

При связанном распределении памяти каждый элемент списка содержит адрес следующего или предыдущего элемента.

Формат элемента информации

inf- поле

S - поле ссылки на следующий элемент.



Достоинства связанного размещения:

1. Произвольное размещение элементов в памяти (отсутствует проблема фрагментации).

2. Легко произвести вставку/удаление из списка.

3. нет необходимости заранее резервировать память, когда вставляется новый элемент, под него выделяется любая свободная ячейка, когда элемент удаляется, ячейка освобождается.

Недостатки связанного размещения:

1. Больше занимаемый объем памяти в связи с необходимостью хранить адреса.

2. Невозможность прямого доступа к любому элементу.

Виды линейных списков.

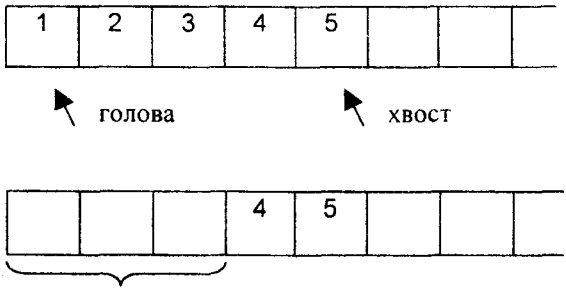
1.СТЕК.

Стек-это линейный список, доступ к которому для включения/удаления элементов производится с одного конца списка.

2. ОЧЕРЕДЬ.

Очередь-это линейный список, в котором включение элемента происходит с одного конца (хвост очереди), а удаление из другого (голова очереди).

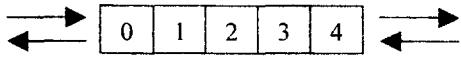
При удалении элемента из очереди (из головы) и включении элемента в очередь (в хвост) возникает эффект "погони головы за хвостом". При этом пространство освобожденной "головой" очереди остается неиспользованным. Чтобы избежать этого необходимо периодически сдвигать очередь в сторону начала.



Другой способ избавиться от неиспользованного пространства использование связанного распределения очереди.

3.ДЕК.

Дек - это линейный список, у которого включение/удаление элементов производится с любого конца списка. Аналогом дека является колода карт.

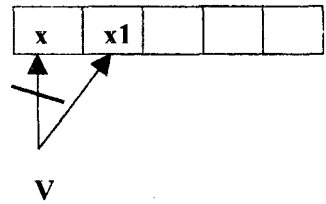


Примеры программ.

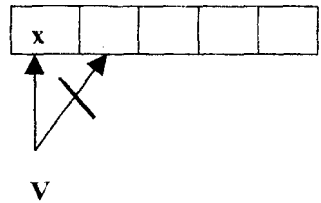
1) Работа со стеком при последовательном распределении памяти.

Пусть требуется вставить в стек элемент x, заведем для стека одномерный массив.

```
St: array [1..10] of integer;
V: integer; // {вершина стека}
V:=1;
St[V]:=x;
V:=V+1;
St[V]:=x1;
...
```



Считаем значение
из стека в переменную Y
Y:=St[V];
V:=V-1;
...



2) Работа со стеком при связанном распределении памяти.

Пусть требуется вставить в стек элементы x1, x2, x3. Затем их удалить.

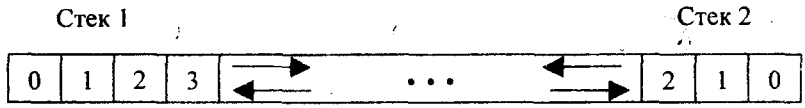
Формат элемента

inf	S

```
Type Rec=^St;
     St:=record
        inf :integer;
        S   :Rec;
  end:
Var
     Stack, v :Rec;
     x1       :integer:
Begin:
1)//вставим в стек x1
   new(Stack);
   Stack.^inf:=x1;
   Stack.^S:=nil;
   v:=Stack;
2)//чтобы внести следующий элемент повторим этот же
цикл
   new(Stack);
   Stack.^inf:=x2;
   Stack.^S:=v;
   v:=Stack;
//аналогично вставляем x3;
...
```

```
//удалим x3
   Stack:=Stack.^S;
   dispose(v);
   v:=Stack;
//теперь можно удалять x2.
```

3) Пример расположения линейных списков в едином адресном пространстве при **последовательном** распределении памяти.



Рабочее задание

Задание 1

Разработать и отладить программу, выполняющую обработку нескольких линейных списков в едином адресном пространстве при последовательном распределении памяти. Предусмотреть возможность вставки, удаления и просмотра элементов, используя меню выбора. Виды линейных списков и дополнительные операции над ними различаются по вариантам.

Задание 2

Разработать и отладить программу, выполняющую обработку нескольких линейных списков при **связанном** распределении памяти. Предусмотреть возможность вставки, удаления и просмотра элементов, используя меню выбора. Виды линейных списков и дополнительные операции над ними различаются по вариантам.

Варианты

№	Списки	Дополнительные операции
1	три стека	Объединить в один стек

2	два стека и очередь	-
3	две очереди и стек	-
4	стек, очередь, дек	-
5	два стека и дек	-
6	два дека и стек	-
7	два стека	Преобразовать в очередь
8	стек и очередь	Преобразовать в дек
9	стек и дек	Преобразовать в очередь
10	три очереди	Объединить в одну очередь
11	две очереди и дек	-
12	два дека и очередь	-
13	две очереди	Преобразовать в стек
14	очередь и дек	Преобразовать в стек
15	три дека	Объединить в один дек
16	два дека	Преобразовать в стек
17	две очереди	Преобразовать в дек
18	очередь и дек	Преобразовать в стек
19	стек и дек	Преобразовать в дек
20	два стека	Преобразовать в дек

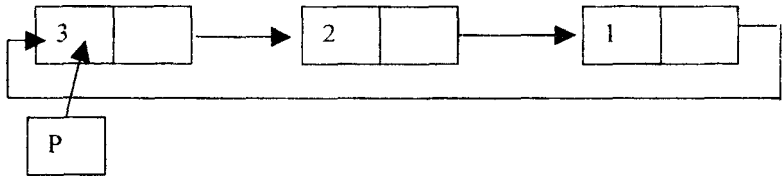
Лабораторная работа №3

Третья лабораторная работа посвящена методам обработки циклических, многомерных и двунаправленных списков

Теоретический материал.

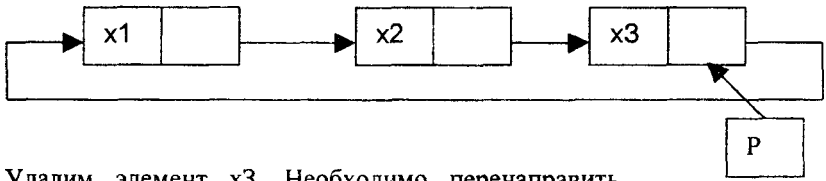
1. ЦИКЛИЧЕСКИЕ СПИСКИ.

Циклический список - это линейный список, в котором последний элемент указывает на первый.

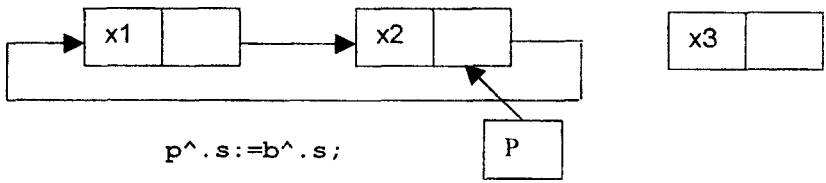


Над циклическими списками выполняются практически те же операции, что и над линейными списками вообще (см п.2.1). Однако есть ряд операций специфичных для циклического списка:
ввод/удаление элемента слева/справа от указателя;
сдвиг указателя на 1 элемент.

УДАЛЕНИЕ ЭЛЕМЕНТА СЛЕВА ОТ УКАЗАТЕЛЯ.



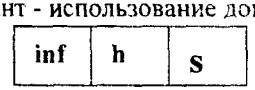
Удалим элемент x3. Необходимо перенаправить ссылку с элемента x2 на x1

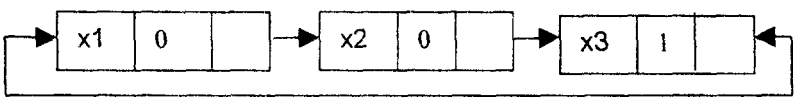



```
p^.s:=b^.s;
```

```
dispose (b);
b:=p^ s;
```

Часто приходится иметь дело с информацией, имеющей начало и конец. В тоже время в циклических списках явного начала и конца нет. Существует два подхода к решению этой проблемы. Можно ввести особый элемент - голову списка - элемент, по формату такой же, как и остальные, однако его информационное поле содержит лишь служебную информацию о том, что это голова списка.
Другой вариант - использование дополнительных битов-признаков.
Формат:



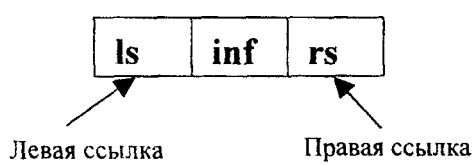


Введем в каждый элемент дополнительное поле h, которое указывает, является ли этот элемент началом списка. Для этого достаточно одного бита.

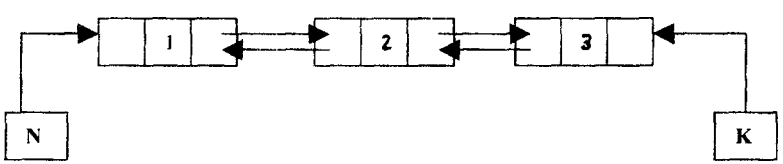
2. ДВУНАПРАВЛЕННЫЕ СПИСКИ.

В двунаправленном списке используется два поля ссылки, таким образом каждый элемент хранит адрес последующего и предыдущего элемента

Формат элемента

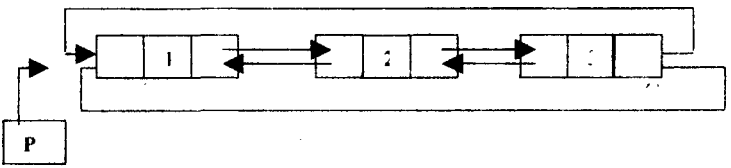


Изобразим двусвязный дек:



Двусвязным можно выполнить любой линейный список, но именно для дека он наиболее выгоден. Есть сложности удаления из конца односвязного дека. Использование двусвязного дека позволяет удалять элементы так же как и из начала без прохода по всем элементам дека.

Двунаправленным может быть и циклический список.

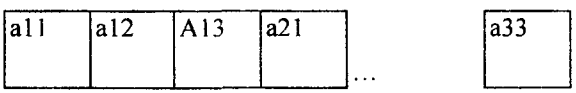


Все, что сказано выше о циклических списках справедливо и для двунаправленных циклических списках. Отличие в том, что над двунаправленным циклическим списком возможны такие операции как сдвиг указателя в обе стороны и т.д., а включение/удаление элемента одинаково легко производится и справа и слева от указателя.

5. МНОГОМЕРНЫЕ СПИСКИ.

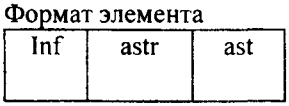
Рассмотрим пример организации двумерного массива при связном и последовательном распределении памяти.

Последовательное распределение памяти:



Связное распределение памяти:

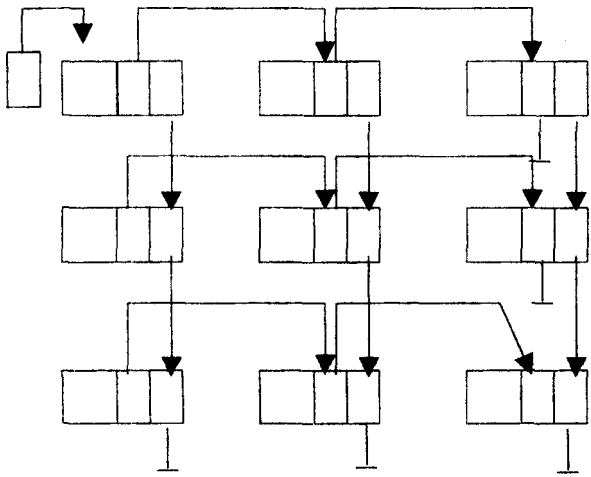
Формат элемента



Inf- поле информации;

Astr - адрес следующего элемента в данной строке;

Ast - адрес следующего элемента в данном столбце.



Рабочее задание

Разработать и отладить программу реализующую обработку списков при связанном распределении. Организовать возможность вставки. удаления и поиска элементов, используя меню выбора. Вид списка и дополнительное задание различаются по вариантам.

Варианты

1. двунаправленный дек, преобразовать в стек;
2. циклический список, преобразовать в двунаправленный дек;
3. двунаправленный циклический список, преобразовать в стек;
4. дек с тремя связями, указывающими на предыдущий, последующий и через один элемент;
5. циклический список с тремя связями, указывающими на предыдущий, последующий и через один элемент;
6. трехмерный список;
7. двумерный список;
8. двумерный список, строки представлены в виде стека;
9. двумерный список, строки представлены в виде дека;
10. двумерный список, строки представлены в виде двунаправленного дека;
11. двумерный список, строки представлены в виде очереди;
12. двумерный список, строки представлены в виде циклического списка;
13. двумерный список, строки представлены в виде двунаправленного циклического списка;
14. двумерный список, столбцы представлены в виде очереди;
15. двумерный список, столбцы представлены в виде стека;
16. двумерный список, столбцы представлены в виде дека;
17. двумерный список, столбцы представлены в виде двунаправленного дека;
18. двумерный список, столбцы представлены в виде очереди;
19. двумерный список, столбцы представлены в виде циклического списка;
20. двумерный список, столбцы представлены в виде двунаправленного циклического списка;

Лабораторная работа №4

Третья лабораторная работа посвящена методам обработки иерархических структур (деревьев)

Теоретический материал.

1. ОБЩИЕ ПОНЯТИЯ.

Линейные структуры отражают только последовательность элементов, однако, при отражении многих объектов реального мира помимо перечня элементов необходимо установить их иерархию. Поэтому для отображения таких объектов в БД необходимы иерархические структуры данных.

2.БИНАРНЫЕ ДЕРЕВЬЯ.

Бинарное дерево - это множество элементов, каждый из которых состоит из корня и не более чем двух, непересекающихся множеств.

Таким образом, бинарное дерево - это дерево, в котором каждый элемент имеет не более 2 поддеревьев.

Существует 3 основных порядка обхода бинарного дерева:

1. Прямой.
   - пройти корень;
   - пройти левое поддерево;
   - пройти правое поддерево.
2. Обратный
   - пройти корень;
   - пройти правое поддерево;
   - пройти левое поддерево.
3. Концевой
   - пройти левое поддерево;
   - пройти правое поддерево;
   - пройти корень.

ФОРМАТ ЭЛЕМЕНТА ДЕРЕВА

L	inf	R

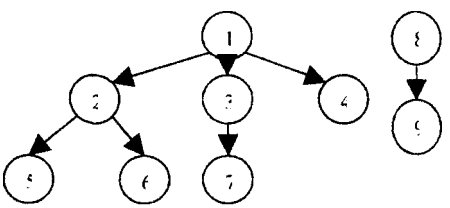
Inf –поле информации, L –адрес левого поддерева, K –адрес правого поддерева.

ПРОШИТЫЕ ДЕРЕВЬЯ.

Такие деревья позволяют избавляться от вышеперечисленных недостатков. В адресные поля концевых элементов вместо пустого адреса nill вписывается адрес вышестоящего элемента. Таким образом, мы получаем возможность двигаться не только вниз по дереву, но и вверх. Однако, новых связей, приведенных нами, нет в реальном объекте, который отражает данное дерево. Следовательно, мы должны отличать реальные связи, которые описывает объект, от мнимых связей, играющих служебную роль в БД. Поэтому формат элемента следует несколько изменить, добавив в него признаки мнимости связи.

ПРЕОБРАЗОВАНИЕ ПРОИЗВОЛЬНЫХ ИЕРАРХИЧЕСКИХ СТРУКТУР В БИНАРНЫЕ ДЕРЕВЬЯ.

Среди объектов, описываемых в БД, существуют такие, у которых каждый корень может иметь один и более поддеревьев. Надо уметь преобразовывать их в бинарные для удобства работы.



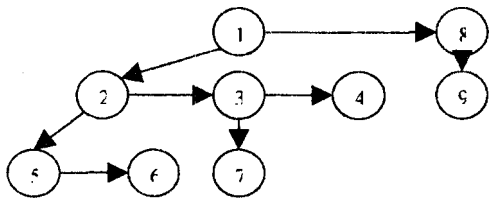
АЛГОРИТМ ПРЕОБРАЗОВАНИЯ

1. Удалить все вертикальные связи, кроме связей от корня к его первому поддереву;

2. Добавить горизонтальные связи между элементами - поддеревьями одного корня;

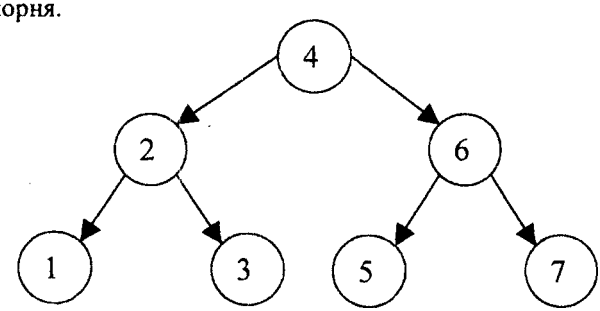
3. Если лес состоит из нескольких деревьев, связать их корни горизонтальной связью.

В результате получаем бинарное дерево, операции над которым эквивалентны операциям над исходным лесом.



ДЕРЕВО ПОИСКА.

Дерево поиска - это бинарное дерево, в котором для каждого элемента все значения в левом поддереве меньше корня, а в правом больше корня.

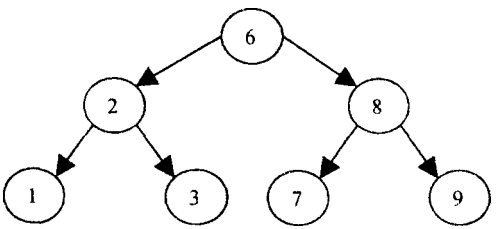


Подобная структура наилучшим образом отвечает поиску методом половинного деления.

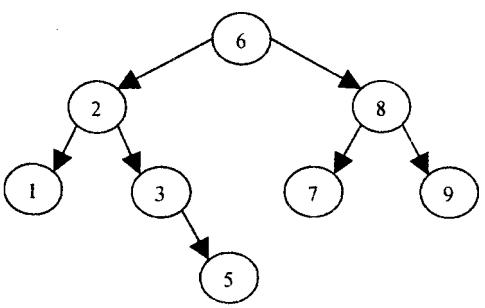
Пусть требуется найти элемент x. Сравним его значение с корнем дерева поиска. Если x больше корня дерева поиска, то x находится в правом поддереве корня (либо отсутствует). Далее x сравнивается с элементом корня поддерева и т.д.

ВКЛЮЧЕНИЕ ЭЛЕМЕНТА

Дано дерево поиска, вставить элемент "5"



На первом этапе выполняется поиск места для нового элемента. Для этого необходимо пройти по соответствующей ветви дерева от корня до пустого адреса. На место этого пустого адреса вписывается адрес включаемого элемента.



УДАЛЕНИЕ ЭЛЕМЕНТА

При удаление элемента возможны 3 ситуации:

1. Удаление концевого элемента. Для этого необходимо:
   - найти этот элемент;
   - найти его корень и заменить ссылку на nill;
   - вернуть памяти освободившуюся ячейку.
2. Удаление элемента, имеющего одно поддерево
   - переправить ссылку его корня на поддерево удаляемого элемента
3. Удаление элемента, имеющего два поддерева.
   - удаляемый элемент следует заменить на самый правый элемент его левого поддерева или самый левый элемент правого поддерева.

3. ПРОГРАММНАЯ РЕАЛИЗАЦИЯ ДЕРЕВА ПОИСКА

В разделе описания переменных зададим формат элемента дерева:

```
Type Tree=^T;
     T=record;
        Inf:integer;
        L,R,:Tree;
     End;
```

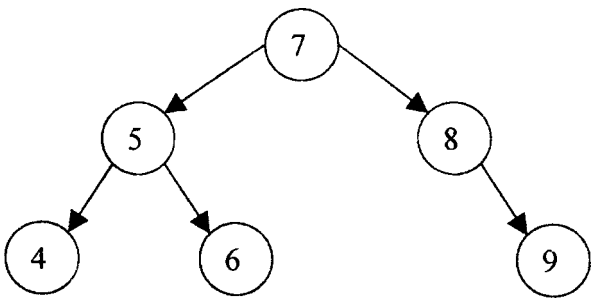
Для навигации по дереву используем рекурсивную процедуру:

```
Procedure find_tree(var P:Tree; x:integer);
Begin
   If (P<>nil) then
   Begin
      If P^.inf=x then …{нашли элемент};
      If (P^.inf>x) then find_tree(P^.L,x);
      If (P^.inf<x) then find_tree(P^.R,x);
   End;
End;
```

4. СБАЛАНСИРОВАННЫЕ ДЕРЕВЬЯ.

Дерево является сбалансированным, если для каждого элемента высота левого и правого поддеревьев различаются не более чем на «I».

ПРИМЕР: Дано дерево поиска. Рассчитаем критерий сбалансированности для каждого элемента.



Ни в одном элементе разница в высоте левого и правого поддеревьев не превышает «I». Следовательно, дерево сбалансировано.
Критерий сбалансированности - это разница в высоте левого и правого поддеревьев.

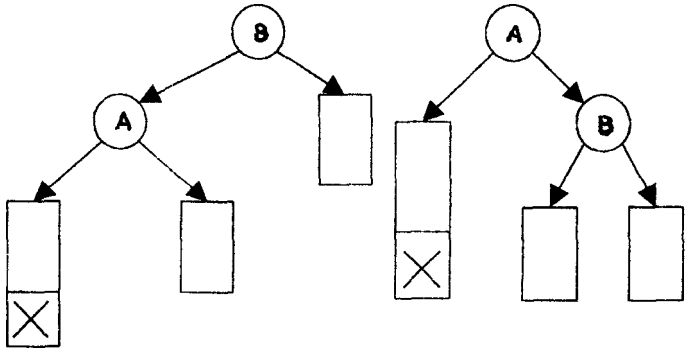
БАЛАНСИРОВКА ДЕРЕВА ПОИСКА

Рассмотрим алгоритм балансировки. Существует теорема:

Все случаи нарушения балансировки при включении сводятся к двум вариантам.

Вариант 1

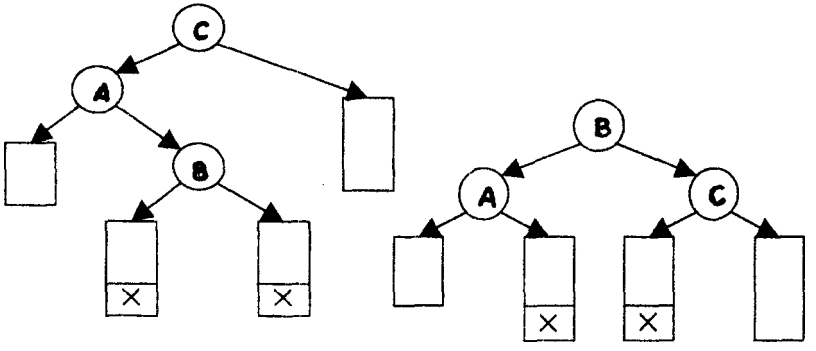
Рассмотрим рисунок на следующей странице. На данном рисунке «А» и «В» элементы, с которыми производится манипуляции. Произведем балансировку.



до балансировки после балансировки

Подняв элемент «А» и его левое поддерево мы устранили несбалансированность.

Вариант 2



до балансировки после балансировки

Обратим внимание, что в обоих случаях все преимущества элементов производились по вертикали, так как по горизонтали дерево не перемещается. Полученные деревья сохранили свойства дерева поиска. После каждого включения/удаления элемента следует рассчитывать критерий сбалансированности для каждого элемента.

ДОСТОИНСТВА:

быстрота поиска элементов - максимальное время поиска меньше, чем в каком-нибудь другом.

НЕДОСТАТКИ:

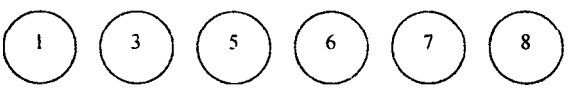
после каждого включения/удаления элемента требуется расчет критерия и возможность сбалансирования.

вывод:

Эти деревья выгодно использовать в тех базах данных, где основной и возможно единственной операцией является поиск. Примером такой БД может служить оглавление диска.

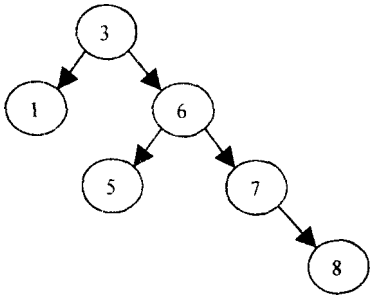
ПОСТРОЕНИЕ ДЕРЕВА ПОИСКА ПО ВЕРОЯТНОСТИ ОБРАЩЕНИЯ К ЭЛЕМЕНТАМ.

Пусть даны элементы бинарного дерева и вероятности запроса к ним:



0.1 0.4 0.1 0.2 0.1 0.1

Поместим в корень тот элемент, который нужен чаще всего. Затем, соблюдая правила дерева поиска, добавим к нему остальные элементы так, чтобы элемент с большей вероятностью запроса был бы ближе к корню.



Данное дерево является деревом поиска, но оно не является сбалансированным. Максимальное время поиска будет хуже, чем в сбалансированном дереве. Но среднее время поиска будет лучше, чем в сбалансированном дереве, так как чаще всего требуемые элементы находятся ближе к корню.

Рабочее задание

Разработать и отладить программу, реализующую обработку иерархической структуры. Обеспечить возможность вставки и удаления элементов, просмотр дерева, используя меню для выбора операции. Вид иерархической структуры и дополнительные операции задаются по вариантам.

Варианты

№	Вид иерархической структуры	Дополнительные операции
1	дерево поиска	прямой обход, после добавления элементов выполнить балансировку
2	дерево поиска	прямой, обратный, концевой обходы
3	прошитое дерево	прямой обход
4	бинарное дерево	обратный обход, преобразовать в дерево поиска

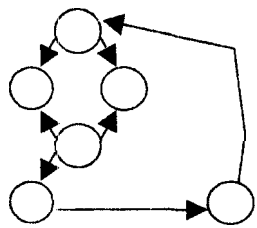
5	дерево поиска	концевой обход, вычислить критерий сбалансированности
6	дерево поиска	прямой обход, вычислить критерий сбалансированности и выполнить балансировку при необходимости
7	бинарное дерево	с помощью дерева привести выражение к бесскобочной логике: (a*b+c)/(a-(d+a))
8	бинарное дерево	с помощью дерева привести выражение к бесскобочной логике: ((d-c)*b)/a+(a+b)*d
9	бинарное дерево	с помощью дерева привести выражение к бесскобочной логике: ((a/(b+c))*(a-d*c))/b
10	дерево поиска	Построить дерево поиска по вероятности обращения к элементам. Элементы и вероятность запроса к ним задается пользователем
11	дерево поиска	Набрать статистику и перестроить дерево по вероятности обращения к элементам
12	дерево поиска	отсортировать массив с помощью дерева;
13	бинарное дерево	небинарный лес преобразовать в бинарное дерево;
14	дерево поиска	обратный обход, после добавления элементов организовать балансировку
15	дерево поиска	концевой обход, после добавления элементов организовать балансировку
16	бинарное дерево	прямой обход, преобразовать в дерево поиска
17	прошитое дерево	обратный обход

18	бинарное дерево	концевой обход, преобразовать в дерево поиска
19	прошитое дерево	концевой обход
20	дерево поиска	обратный обход, вычислить критерий сбалансированности

Лабораторная работа №5
СЕТЕВЫЕ СТРУКТУРЫ.

Теоретический материал.

На рисунке изображена произвольная сетевая структура. Хорошо заметно, что, в отличие от иерархических структур, в сетевых структурах более свободная дисциплина связей между элементами.



Необходимо при разработке сетевой структуры определить дисциплину связей. Определим ее как

i : j

где i - количество ссылок на данный элемент;

j - количество допустимых ссылок данного элемента.

Рассмотрим формат элемента сетевой структуры:

inf	S1	…	Sj

Иногда используют запись вида

M:3

Это значит, что в данной структуре каждый элемент может ссылаться на 3 других, а количество ссылок на каждый элемент не ограничено. Такое возможно, поскольку количество ссылок на каждый элемент не отражается в формате элемента.

Рабочее задание

Разработать и отладить программу, реализующую сетевую структуру с заданной дисциплиной связей.

Варианты

№ варианта	Вид сети	№ варианта	Вид сети
1	5:2	11	2:5
2	4:3	12	M:3
3	3:3	13	4:5
4	1:6	14	4:4
5	3:5	15	3:2
6	M:4	16	M:2
7	5:4	17	4:2
8	5:3	18	5:5
9	2:4	19	6:1
10	3:4	20	2:3

Примечание: M – количество входящих связей неограниченно.