ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ПРОФЕССИОНАЛЬНОГО ОБРАЗОВАНИЯ «САМАРСКИЙ ГОСУДАРСТВЕНЫЙ УНИВЕРСИТЕТ»

ДИНАМИЧЕСКИЕ СТРУКТУРЫ ДАННЫХ. РЕКУРСИЯ

САМАРА 2015

МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РОССИЙСКОЙ ФЕДЕРАЦИИ

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ПРОФЕССИОНАЛЬНОГО ОБРАЗОВАНИЯ «САМАРСКИЙ ГОСУДАРТСВЕННЫЙ УНИВЕРСИТЕТ»

ДИНАМИЧЕСКИЕ СТРУКТУРЫ ДАННЫХ. РЕКУРСИЯ

Методические указания

САМАРА

2015

УДК 519.683.6
ББК 32.973

Составители ***И.В. Семенова***

Рецензент д.ф.-м.н., профессор А.Н. Степанов

Динамические структуры данных. Рекурсия: метод. указания / сост. *И.В. Семенова* – Самара, 2015. –56 с.: ил.

В пособии описываются основные принципы организации и алгоритмы работы с базовыми динамическими структурами, а также такие приемы программирования как рекурсия и рекурсия с возвратом.

Предназначено для студентов специальностей «Фундаментальная математика и механика», «Прикладная математика и информатика», «Математическое обеспечение и администрирование информационных систем».

Подготовлено на кафедре информатики и вычислительной математики.

УДК
519.683.6
ББК 32.973

СОДЕРЖАНИЕ

1 ЛИНЕЙНЫЕ ОДНОНАПРАВЛЕННЫЕ СПИСКИ

1.1 Основные понятия

Линейный список – это динамическая структура данных.

У данных с динамической структурой с течением времени изменяется сама структура, а не только количество элементов, как у файлов или последовательностей.

Линейный список характеризуется тем, что каждый его элемент связан с предшествующим (рисунок 1); известно, какой элемент находится в начале списка, какой в конце, а также, какой элемент стоит перед текущим; переходить от текущего элемента к следующему можно только с помощью указанных связей между соседними элементами.

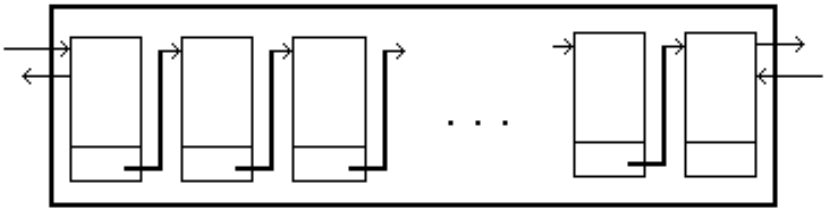

Рис. 1. Структура линейного однонаправленного списка

Основные операции над линейными однонаправленными списками:

- создание списка;
- включение нового элемента;
- уничтожение списка;
- исключение элемента;
- полный перебор элементов списка;
- поиск элемента;
- упорядочение (сортировка) списка.

На базе линейного списка организуются много других типов динамических структур в частности: кольца (рисунок 2), очереди (рисунок 5), деки (рисунок 6) и стеки (рисунок 4).

Отличие кольца от линейного списка в том, что у кольца имеется связь между последним элементов списка и его первым элементом.

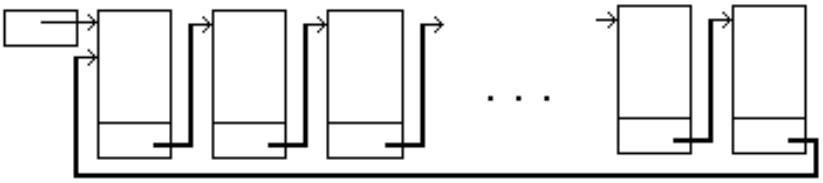

Рис. 2. Структура кольца

У линейного списка и у кольца возможен доступ к любому элементу структуры. Для этого нужно последовательно перемещаться от одного элемента к другому. Во многих реальных ситуациях такой доступ отсутствует и разрешается взаимодействовать только с первым и последним элементами или же только с одним из них. Для моделирования таких объектов используются очереди, деки и стеки.

1.2 Примеры решения задач

Задача 1. Написать программу, реализующую работу с линейным односвязным списком. Реализовать следующие операции:

- инициализация списка;
- добавление элемента в начало списка;
- добавление элемента в конец списка;
- удаление первого вхождения заданного элемента в список;
- печать элементов списка;
- удаление всех элементов списка.

Решение

```
type
TElem=integer;
   TList = ^TNode;
   TNode = record
       Info: integer;
       Next: TList;
   end;
           {Инициализация списка}
procedure List_Init(var first:TList);
begin
  first:=nil;
end;
      {Добавление элемента в начало списка}
procedure List_AddFirst(var first:TList; el:TElem);
var p:TList;   {добавляемое звено списка}
begin
```

```
  new(p);
  p^.Info:=el;
  p^.Next:=first;
  first:=p;
end;

      //Добавление элемента в конец списка
procedure List_AddLast(var first: TList; el:TElem);
var p:TList;    //добавляемое звено списка
p1:TList;   //вспомогательный указатель для поиска последнего
            //элемента списка
begin
   new(p);
   p^.Info:=el;
   p^.Next:=nil;
   if first=nil then
      first:=p
   else
    begin
            //поиск последнего элемента списка
      p1:=first;
      while p1^.Next<>nil do
      p1:=p1^.Next;
            //добавление в список нового звена
      p1^.Next:=p;
    end;
end;

          //удаление первого элемента из непустого списка
procedure List_Del_FirstElem (var first:TList);
var p:TList;   //указатель на удаляемое из списка звено
begin
   p:=first;   //запоминаем указатель на удаляемое звено
   first:=first^.Next;    //удаляем звено из списка
   dispose(p)
end;

     //удаление первого вхождения элемента el в список
       //результат функции:
           // true - элемент найден и удален
           // false - элемент в списке не найден
function List_Del_Elem(var first:TList; el:TElem):boolean;
var
  p:TList;   //указатель на удаляемое из списка звено
             //вспомогательный указатель для поиска звена списка,
             //предшествующего удаляемому
  p1:TList;
           //признак: найден ли элемент el в списке
  found: boolean;
begin
  found :=false;
  if first<>nil  then       //если список не пуст
    if first^.Info=el then  //если первое звено является удаляемым
```

```
      begin
        found:=true;
        List_Del_FirstElem (first);
      end
      else
      begin
               //ищем звено, предшествующее удаляемому
        p1:=first;
        while not found and (p1^.Next<>nil) do
        if p1^.next^.Info=el then
          found:=true
          else
          p1:=p1^.Next;
          if found  then   //если найдено удаляемое звено
            begin
                    //запоминатель указатель на удаляемое звено
              p:=p1^.Next;
              p1^.Next:=p^.Next;
              dispose(p);
            end;
      end;
List_Del_Elem:=found;
end;
            //печать элементов списка
procedure List_Print(first:TList);
begin
   while first<>nil do
    begin
      write(first^.info);
      if first^.Next<>nil then
      write(', ');
      first:=first^.Next;
    end;
    writeln;
end;

            //удаление всех элементов списка
procedure List_Clear(var first:TList);
var p:TList;     //указатель на удаляемое звено списка
begin
  while first<>nil do
  begin
    p:=first;
    first:=first^.Next;
    dispose(p);
  end;
end;

var first: TList;
elem:TElem;
begin
 //............
List_Init(first);
```

```
writeln('Список инициализирован');
writeln('Введите значение добавляемого элемента');
readln(elem);
List_AddFirst(first, elem);
writeln('Введите значение добавляемого элемента');
readln(elem);
List_AddLast(first, elem);
List_print(first);
writeln('Введите значение удаляемого элемента');
readln(elem);
if List_Del_Elem(first, elem) then
  writeln('Элемент в списке найден и удален')
  else
  writeln('Заданный элемент в списке не найден');
List_print(first);
List_Clear(first);
//.................
readln;
end.
```

1.3 Задачи

1. Проверьте, содержатся ли элементы списка A в списке B в указанном списком A порядке.

2. Пусть дан список A, содержащий целые числа. Занесите в список B порядковые номера максимальных элементов списка A.

3. Определите, что является результатом работы следующей процедуры:

```
Procedure ZZZ (h1:TList);
var h2,h3:TList;
f:boolean;
begin
   while h1<>nil do
        begin
        h2:=h1^.next;
        h3:=h1;
        f:=true;
        while (h2<>nil) and f do
            if h1^.info=h2^.info then f:=false
                       else
                          begin
                            h3:=h2;
                            h2:=h2^.next
                         end;
        if not f then
                  begin
                  h2:=h1^.next;
                  new(h1^.next);
                  h1^.next^.info:=h3^.info;
                  h1^.next^.next:=h2;
```

```
                    h1:=h1^.next
                    end;
          h1:=h1^.next
        end;
end.
```

4. Вычислите среднее арифметическое элементов непустого списка.

5. Многочлен $P(x)=a_nx^n+a_{n-1}x^{n-1}+\ldots+a_1x+a_0$ с вещественными коэффициентами можно представить в виде списка, причем если $a_i=0$, то соответствующее звено в список не включается. Опишите:

а) функцию Value(P,x), вычисляющую значение многочлена – списка P в точке x;

б) программу, рассчитывающую производную многочлена – списка P по переменной x;

в) программу, интегрирующую многочлен – список P.

6. Определить количество слов списка, начинающихся и оканчивающихся одним и тем же символом. Удалить из списка все элементы, содержащие слова из 4 и менее символов.

7. Проверить элементы списка L на упорядоченность. Если список является упорядоченным, то удалить элемент списка, содержащий значение наиболее близкое к среднему арифметическому значению элементов всего списка, иначе удалить элемент, находящийся в середине списка.

8. Определить совпадают ли список L1 и список L2. Если списки не совпадают, то удалить из списка L1 элементы, входящие в список L2, иначе удалить из списка L2 все вождения наибольшего элемента.

9. Сформировать список L путем включения в него по одному разу элементов, входящих в оба списка L1 и L2. Удалить из списков L1 и L2 общие элементы.

10. Определить входит ли список L1 в список L2. Если список L1 входит в L2 , то удалить его из списка L2, иначе удалить из списка L1 все вхождения наименьшего элемента.

11. Из списка L сформировать два новых списка: L1 — из положительных элементов и L2 — из остальных элементов списка L. Удалить из списка L все отрицательные элементы.

12. Подсчитать число вхождений элемента E в список L. Удалить из списка L каждое второе вхождение элемента E.

13. Вставить элемент E перед каждым вхождением в список L заданного элемента M. Удалить из списка L все вхождения заданного элемента P.

14. Добавить в конец списка L1 все элементы списка L2. Удалить из измененного списка L1 все повторяющиеся элементы.

15. Переставить первый и последний элементы непустого списка. Удалить из списка все возрастающие серии. Если таких серий нет, то вывести соответствующее сообщение.

16. Скопировать в список L2 за каждым вхождением заданного элемента все элементы списка L1. Удалить из списка L1 все нечетные элементы.

17. Сформировать список L путем включения в него по одному разу элементов, входящих хотя бы в один из списков L1 и L2. При этом из списков L1 и L2 эти элементы должны быть удалены.

18. Объединить два упорядоченные по неубыванию списка L1 и L2 в один упорядоченный по неубыванию список путем построения нового списка L и изменением соответствующим образом ссылок в L1 и L2. В полученном списке L удалить элементы, расположенные на четных местах.

2 ДВУНАПРАВЛЕННЫЕ СПИСКИ

2.1 Основные понятия

Структура двунаправленного (двусвязного) списка приведена на рисунке 3.

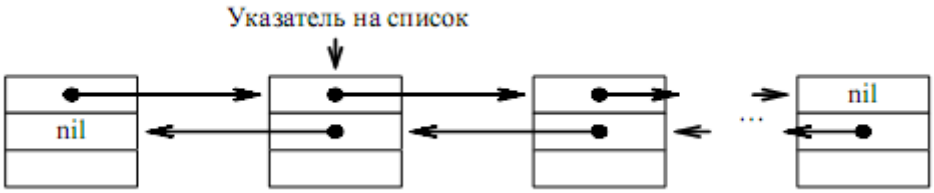

Рис.3. Структура двунаправленного списка

Основные операции над двунаправленным списком:

- создание списка;
- включение нового элемента;
- уничтожение списка;
- исключение элемента;
- полный перебор списка;
- поиск элемента;
- упорядочение (сортировка) списка.

2.2 Задачи

1. Напишите программу, реализующую работу с линейным двусвязным списком. Программа должна делать следующее:

a) предложить пользователю создать список и запросить, как должны добавляться новые элементы (при создании списка): в начало списка, в конец списка, так, чтобы список был упорядочен по возрастанию, или так, чтобы список был упорядочен по убыванию?

b) когда список будет сформирован, выдать запрос пользователю о желании продолжить работу со списком. В случае положительного ответа должны предоставляться следующие возможности (пока не будет отказа от продолжения работы):

- поиск элемента с заданным значением информационного поля;
- удаление элемента с заданным значением информационного поля (если такого элемента нет, выводить сообщение об этом);

- добавление элемента с заданным значением информационного поля (в начало, в конец, с соблюдением упорядоченности);
- выяснение длины списка;
- выяснение того, что список не пуст.

с) Реализовать процедуру полной очистки списка, которая должна выполняться сразу же после того, как пользователь завершит работу со списком. Процедура должна выводить сообщение о том, что память очищена. На этом программа завершает работу.

2. Дан указатель $P1$ на начало непустого двусвязного списка. Перегруппировать его элементы, переместив все элементы с нечетными значениями в конец списка (в том же порядке). Операции выделения и освобождения памяти не использовать, информационные поля не изменять.

3. Пусть в двусвязном списке A хранится информация о людях (фамилия, имя, отчество, профессия). Имеется список B, содержащий перечень профессий. Удалить из списка A тех людей, чья профессия не указана в списке B.

4. Даны два непустых двусвязных списка и связанные с ними указатели: $P1$ и $P2$ указывают на первый и последний элементы первого списка, $P0$ — на один из элементов второго. Объединить исходные списки, поместив все элементы первого списка (в том же порядке) перед данным элементом второго списка. Операции выделения и освобождения памяти не использовать.

5. Дан указатель $P1$ на первый элемент непустого двусвязного списка. Перегруппировать его элементы, переместив все элементы с нечетными номерами в конец списка (в том же порядке). Операции выделения и освобождения памяти не использовать, информационные поля не изменять.

6. Даны два непустых двусвязных списка и связанные с ними указатели: $P1$ и $P2$ указывают на первый и последний элементы первого списка, $P0$ — на один из элементов второго. Объединить исходные списки, поместив все элементы первого списка (в том же порядке) после данного элемента второго списка. Операции выделения и освобождения памяти не использовать.

7. Дано число K (> 0) и указатели $P1$ и $P2$ на первый и последний элементы непустого двусвязного списка. Осуществить циклический сдвиг элементов списка на K позиций вперед (т. е. в направлении от начала к концу списка). Для выполнения циклического сдвига преобразовать исходный список в кольцевой, после чего «разорвать» его в позиции, соответствующей данному значению K. Операции выделения и освобождения памяти не использовать.

8. Дано число K (> 0) и указатели $P1$ и $P2$ на первый и последний элементы непустого двусвязного списка. Осуществить циклический сдвиг элементов списка на K позиций назад (т. е. в направлении от конца к началу списка). Для выполнения циклического сдвига преобразовать исходный список в кольцевой, после чего «разорвать» его в позиции, соответствующей данному значению K. Операции выделения и освобождения памяти не использовать.

9. Даны указатели $P1$, $P2$ и $P3$ на первый, последний и текущий элементы двусвязного списка (если список является пустым, то $P1 = P2 = P3 =$ nil). Также дано число N (> 0) и набор из N чисел. Описать тип TList — запись с полями First, Last и Current типа PNode (поля указывают соответственно на первый, последний и текущий элементы списка) — и процедуру InsertLast(L, D), которая добавляет новый элемент со значением D в конец списка L (L — входной и выходной параметр типа TList, D — входной параметр целого типа). Добавленный элемент становится текущим. С помощью этой процедуры добавить в конец исходного списка данный набор чисел (в том же порядке).

10. Дан непустой двусвязный список, первый, последний и текущий элементы которого имеют адреса $P1$, $P2$ и $P3$. Также даны пять чисел. Используя тип TList, описать процедуру InsertAfter(L, D), которая вставляет новый элемент со значением D после текущего элемента списка L (L — входной и выходной параметр типа TList, D — входной параметр целого типа). Вставленный элемент становится текущим. С помощью этой процедуры вставить пять данных чисел в исходный список.

11. Дан непустой двусвязный список, первый, последний и текущий элементы которого имеют адреса $P1$, $P2$ и $P3$. Используя тип TList, описать

процедуры ToFirst(*L*) (делает текущим первый элемент списка *L*), ToNext(*L*) (делает текущим в списке *L* следующий элемент, если он существует), SetData(*L*, *D*) (присваивает текущему элементу списка *L* значение *D* целого типа) и функцию IsLast(*L*) логического типа (возвращает True, если текущий элемент списка *L* является его последним элементом, и False в противном случае). Параметр *L* имеет тип TList; в процедурах ToFirst и ToNext он является входным и выходным. С помощью этих процедур и функций присвоить нулевые значения элементам исходного списка с нечетными номерами и вывести количество элементов в списке (текущим элементом должен стать последний элемент списка).

12. Даны указатели на первый, последний и текущий элементы двух непустых двусвязных списков. Используя тип TList (см. задание 8), описать процедуру AddList(*L*1, *L*2), которая добавляет все элементы из списка *L*1 (в том же порядке) в конец списка *L*2; в результате список *L*1 становится пустым. Текущим элементом списка *L*2 становится первый из добавленных элементов. Оба параметра процедуры имеют тип TList и являются входными и выходными. Операции выделения и освобождения памяти в процедуре не использовать. С помощью этой процедуры добавить первый из исходных списков в конец второго.

13. Диапазон представления целых чисел (integer и т.д.) ограничен. Так например факториал числа 13! Выходит за диапазон LongInt. Использование типа real частично решает проблему, но не гарантирует точность вычисления. Напишите программу представления многозначного числа в виде двусвязного списка (каждая цифра – элемент списка), а также подпрограммы их сложения и умножения.

2.3 Контрольные вопросы

1. Любой ли список является связным?

2. В чем отличие первого элемента однонаправленного (двунаправленного) списка от остальных элементов этого же списка?

3. В чем отличие последнего элемента однонаправленного (двунаправленного) списка от остальных элементов этого же списка?

4. Почему при работе с однонаправленным списком необходимо позиционирование на первый элемент списка?

5. Почему при работе с двунаправленным списком не обязательно позиционирование на первый элемент списка?

6. В чем принципиальные отличия выполнения добавления (удаления) элемента на первую и любую другую позиции в однонаправленном списке?

7. В чем принципиальные отличия выполнения основных операций в однонаправленных и двунаправленных списках?

8. С какой целью в программах выполняется проверка на пустоту однонаправленного (двунаправленного) списка?

9. С какой целью в программах выполняется удаление однонаправленного (двунаправленного) списка по окончании работы с ним? Как изменится работа программы, если операцию удаления списка не выполнять?

3 СТЕКИ

3.1 Основные понятия

Стек представляет собой динамическую структуру данных, основанную на линейном однонаправленном списке. У стека для взаимодействия доступен только один конец структуры ─ вершина стека (рисунок 4). И включение нового элемента в стек и выборка последнего ранее включенного идет через вершину стека. Таким образом, на обслуживание попадает первым элемент, поступивший последним. Говорят, что стек ─ это структура с дисциплиной обслуживания ***LIFO*** (***L***ast ***I***n, ***F***irst ***O***ut) ─ «последним пришёл, первым ушёл».

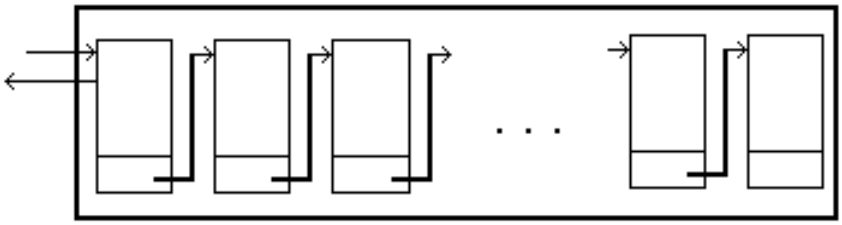

Рис. 4. Структура стека

Описание типов, используемых для работы со стеком:

```
Type
T_info=char; {Зависит от типа информационного поля, может быть
              любой тип}
T_Stack = ↑T_Elem;
T_Elem = record
          body :  T_info;
          Next  :  T_Stack
          end;
```

Основные операции над стеками:

- создание стека;
- включение нового элемента;
- проверка стека на пустоту;
- извлечение элемента.

Для работы со стеком обычно создаются стандартные процедуры обеспечивающие выполнение основных операций над стеком.

Под созданием стека понимается создание *пустого* стека. Подпрограмма должна вернуть в качестве результата пустой указатель типа T_Stack. Если

подпрограмма реализуется как процедура, то она должна иметь формальный параметр-переменную.

Проверку стека на пустоту лучше всего реализовывать с помощью функции логического типа. Функция должна иметь в качестве формального параметра указатель на вершину проверяемого стека.

Включение нового элемента в вершину стека эквивалентно включению элемента в начало линейного однонаправленного списка. Формальные параметры процедуры — указатель на вершину изменяемого стека и вставляемый элемент. Указатель на вершину стека является одновременно входным и выходным параметром, следовательно, это параметр-переменная.

Выборка (исключение) элемента из стека эквивалентно исключению элемента из начала линейного однонаправленного списка. Формальные параметры подпрограммы — указатель на вершину изменяемого стека и переменная для запоминания выбранного элемента. Указатель на вершину стека является одновременно входным и выходным параметром, следовательно, это параметр-переменная.

3.2 Задачи

1. Пусть символ # определен в текстовом редакторе как стирающий символ Backspace, т.е. строка abc#d##c в действительности является строкой ac. Дан текст, в котором встречается символ #. Преобразовать его с учетом действия этого символа.

2. Напечатать содержимое текстового файла t, выписывая литеры каждой его строки в обратном порядке.

3. Даны 2 строки s1 и s2. Из каждой можно читать по одному символу. Выяснить, является ли строка s2 обратной s1.

4. В текстовом файле записана без ошибок формула вида:

```
<формула>=<цифра>|M(<формула>,<формула>)|m(<формула>,<формула>)
 <цифра>=0|1|2|3|4|5|6|7|8|9
```

M обозначает вычисление максимума, m – минимума. Вычислить значение этой формулы: например M(m(3,5),M(1,2))=3.

5. В текстовом файле записана без ошибок формула вида:

```
<формула>=<цифра>|p(<формула>,<формула>)|m(<формула>,<формула>)
<цифра>=0|1|2|3|4|5|6|7|8|9
m (a, b) = (a-b) mod 10,
p (a, b) = (a+b) mod 10.
```

Вычислить значение этой формулы. Например, m(9,p(p(3,5),m(3,8))) = 6.

6. Напечатать в обратном порядке символы слова наибольшей длины из заданного текстового файла.

7. Пусть в текстовом файле записано без ошибок логическое выражение следующего вида:

```
<ЛВ>::=true| false| (not <ЛВ>) | (<ЛВ> and <ЛВ>) | (<ЛВ> or <ЛВ>),
```

где not, or, and обозначают соответственно отрицание, конъюнкцию и дизъюнкцию. Вычислите значение этого выражения.

8. Вычислите значение арифметического выражения, преобразовав его в префиксную запись. В выражение входят:

а) целые числа;

б) знаки +, -, *, /.

9. Вычислите значение арифметического выражения в инфиксной записи. В выражениях входят:

а) целые числа;

б) знаки +, -, *, /.

10. Карту, определяющую прямоугольную область моря, представили матрицей с логическими элементами (false – море, true - суша). Островом будем называть совокупность соприкасающихся (вертикальной или горизонтальной стороной) клеток матрицы со значениями true. Рассчитайте число островов на матрице-карте.

11. Пусть дана логическая матрица, описывающая лабиринт (true – стена, false - проход) и начальное положение человека в лабиринте (x,y). Необходимо предложить любой вариант обхода всех доступных клеток лабиринта.

3.3 Контрольные вопросы

1. В чем преимущества и недостатки организации структур в виде стека?
2. Для моделирования каких реальных задач удобно использовать стек?
3. Какое значение хранит указатель на стек?
4. Какие существуют ограничения на тип информационного поля стека?
5. С какой целью в программах выполняется проверка на пустоту стека?
6. При работе со стеком доступны позиции ограниченного числа элементов. Возможна ли ситуация записи новых элементов стека на уже занятые собственными элементами участки памяти (запись себя поверх себя)?
7. С какой целью в программах выполняется удаление стека по окончании работы с ними? Как изменится работа программы, если операцию удаления не выполнять?

4 ОЧЕРЕДИ

4.1 Основные понятия

У очереди доступен для включения конец, а для исключения (выборки) ─ начало. Элемент, поступивший в очередь раньше и обслуживается раньше. Говорят, что очередь это структура с дисциплиной обслуживания ***FIFO*** (***F***irst ***I***n, ***F***irst ***O***ut) ─ «первый пришёл, первый ушел».

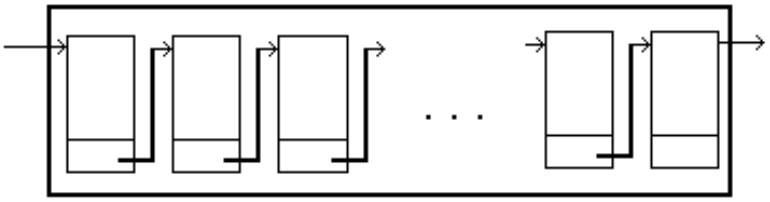

Рис.5. Структура очереди

У дека оба конца доступны, как для включения, так и для выборки (рисунок 6). Таким образом, можно сказать, что дек ─ это двусторонняя очередь.

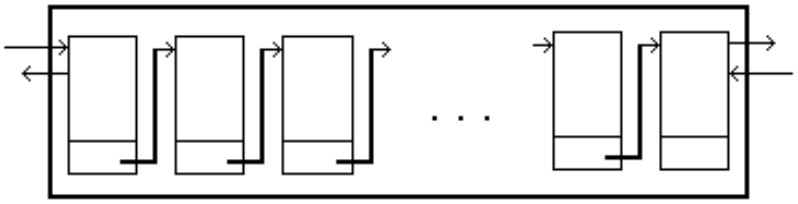

Рис.6. Структура дека

Основные операции над очередью:

- создание очереди;
- включение нового элемента;
- проверка очереди на пустоту;
- извлечение элемента.

4.2 Задачи

1. Пусть дан текстовый файл A. Перепишите его содержимое в файл B, перенося при этом в конец каждой строки все входящие в нее знаки препинания.

2. Пусть дан текстовый файл A. Перепишите его содержимое в файл B, удалив при этом слова, длина которых меньше заданной.

3. Напечатать наибольшее по длине предложение из заданного текстового файла.

4. Пусть даны две очереди X и Y, содержащие вещественные числа. Из каждой очереди одновременно извлекается по одному числу x и y соответственно. Если x<y, то число (x+y) помещается в конец очереди X, иначе число (x-y) помещается в конец очереди Y. Необходимо определить число шагов, через которые одна из очередей станет пустой.

5. Дан текстовый файл. За один просмотр файла напечатать элементы файла в следующем порядке: сначала все символы, отличные от цифр, а затем все цифры, сохраняя исходный порядок в каждой группе символов.

6. Дан файл, содержащий числа. За один просмотр файла напечатать элементы файла в следующем порядке: сначала все числа, из интервала [a,b], потом все числа, меньшие a, потом все числа, большие b, сохраняя исходный порядок в каждой группе чисел.

7. Дан текстовый файл. За один просмотр файла напечатать элементы файла в следующем порядке: сначала все слова, начинающиеся на гласную букву, потом все слова, начинающиеся на согласную букву, сохраняя исходный порядок в каждой группе слов.

8. Дан текстовый файл. За один просмотр файла напечатать элементы файла в следующем порядке: сначала все слова, начинающиеся с прописной буквы, потом все слова, начинающиеся со строчной буквы, сохраняя исходный порядок в каждой группе слов.

9. Дан файл, содержащий информацию о сотрудниках фирмы: фамилия, имя, отчество, пол, возраст, размер зарплаты. За один просмотр файла напечатать элементы файла в следующем порядке: сначала все данные о мужчинах, потом все данные о женщинах, сохраняя исходный порядок в каждой группе сотрудников.

10. Дан файл, содержащий информацию о сотрудниках фирмы: фамилия, имя, отчество, пол, возраст, размер зарплаты. За один просмотр файла напечатать элементы файла в следующем порядке: сначала все данные о

сотрудниках, зарплата которых меньше 10000, потом данные об остальных сотрудниках, сохраняя исходный порядок в каждой группе сотрудников.

11. Дан файл, содержащий информацию о сотрудниках фирмы: фамилия, имя, отчество, пол, возраст, размер зарплаты. За один просмотр файла напечатать элементы файла в следующем порядке: сначала все данные о сотрудниках младше 30 лет, потом данные об остальных сотрудниках, сохраняя исходный порядок в каждой группе сотрудников.

12. Дан файл, содержащий информацию о студентах: фамилия, имя, отчество, номер группы, оценки по трем предметам текущей сессии. За один просмотр файла напечатать элементы файла в следующем порядке: сначала все данные о студентах, успешно обучающихся на 4 и 5, потом данные об остальных студентах, сохраняя исходный порядок в каждой группе студентов.

3.3 Контрольные вопросы

1.В чем преимущества и недостатки организации структур в виде очереди?

2. Для моделирования каких реальных задач удобно использовать очередь?

3. Какое значение хранит указатель на очередь?

4. Какие существуют ограничения на тип информационного поля очереди?

5. С какой целью в программах выполняется проверка на пустоту очереди?

6. С какой целью в программах выполняется удаление очереди по окончании работы с ними? Как изменится работа программы, если операцию удаления не выполнять?

5 РЕКУРСИЯ

5.1 Основные понятия

В математике для решения подавляющего большинства задач используются методы, которые в конечном счете могут быть сведены к одному из двух базовых способов: итерации или рекурсии.

Итерация означает неоднократное повторение одних и тех же действий, которые после некоторого количества шагов приводят к желаемому результату. Характерным примером итерационного способа решения задачи являются методы последовательных приближений решения нелинейных уравнений, в том числе метод касательных, метод хорд и т.д.

Рекурсия представляет собой ссылку при описании объекта, действия на описываемый объект, действие. Рекурсия означает решение задачи с помощью сведения решения к самому себе. При этом вычисления зависят от других, в некотором смысле более простых (обычно меньших) значений аргумента или аргу-ментов задачи. Полностью аналогичные механизмы используются в базовой теории рекурсивных функций, в методе математической индукции, а также в рекуррентных последовательностях, например, $a_k=2a_{k-1}+k,\ \forall k>0,\ a_0=1$

Исполнитель рекурсивного алгоритма сводит неизвестное к другому неизвестному, накапливая информацию (прямой ход) и откладывая фактические вычисления до момента, когда выполнится условие, позволяющие напрямую вычислить искомое значение. Затем выполняется обратный ход рекурсии.

Глубиной рекурсии считается количество рекурсивных вызовов прямого хода, находящихся на различных уровнях рекурсии.

Если каждый рекурсивный вызов изображать узлом дерева, то можно построить ***дерево рекурсивных вызовов,*** высота которого равна глубине рекурсии.

Основное требование к построению рекурсивных подпрограмм состоит в том, чтобы в подпрограмме имелось условие (одно или несколько), которое

ограничивает глубину рекурсии, которое в некоторый момент времени становится истинным и обрывает цепочку рекурсивных вызовов.

Следовательно, в общем случае, можно строить рекурсию не только «назад», к меньшим значениям параметров, к «предшествующим» ситуациям, но и «вперёд», к большим значениям параметров, к «будущим» ситуациям.

5.2 Задачи

1. Напишите рекурсивную функцию сложения двух чисел ($a + b$).

2. Напишите рекурсивную функцию перевода числа из десятичной системы счисления в двоичную.

3. Напишите рекурсивную функцию, проверяющую, является ли заданное натуральное число простым.

4. Дано натуральное число n >=1. Напишите рекурсивную функцию, определяющую число a, такое, что $2^{a-1} \le n < 2^a$.

5. Напишите рекурсивную функцию, определяющую наибольший общий делитель.

6. Описать рекурсивную функцию GCD(A, B) целого типа, находящую *наибольший общий делитель* (НОД, greatest common divisor) двух целых положительных чисел A и B, используя *алгоритм Евклида*:

НОД(A, B) = НОД(B, A mod B), $B \neq 0$; НОД(A, 0)=A.

7. Напишите программу, которая считает количество чётных цифр введённого числа.

8. Описать рекурсивную функцию DigitSum(K) целого типа, которая находит сумму цифр целого числа K, не используя оператор цикла.

9. Описать рекурсивную функцию PowerN(X, N) вещественного типа, находящую значение N-й степени числа X по формулам:

$X^0=1$,

$X^N=(X^{N/2})^2$ при *четных* $N>0$,

$X^N=X \cdot X^{N-1}$ при *нечетных* $N>0$,

$X^N=1/X^{-N}$ при $N<0$

($X \neq 0$ — вещественное число, N — целое; в формуле для четных N должна использоваться операция *целочисленного деления*).

10. Опишите рекурсивную функцию, вычисляющую сумму первых n членов арифметической прогрессии.

11. Используя процедуру write(x) только для x = 0, 1, ..., 9, напишите рекурсивную процедуру печати десятичной записи целого положительного числа x.

12. Описан тип (для реализации на компьютере замените русские буквы латинскими):

type имя = (Алла, ..., Юрий, нет);

Предполагая уже описанными функции Father(x) и Mother(x), значениями которых являются имена соответственно отца и матери человека по имени *x* или идентификатор *нет*, если отсутствуют сведения о соответствующем родителе, описать рекурсивную логическую функцию Descendant(*a, b*), проверяющую, является ли человек с именем *b* потомком (ребенком, внуком, правнуком и т.п.) человека с именем *a*.

13. Решить предыдущую задачу в предположении, что имеются функция AmountOfChildren(x), указывающая число детей человека с именем *x*, и функция Child(x, k), указывающая имя k-го ребенка человека с именем *x* (k не должно превышать число детей человека по имени *x*).

14. Описана функция

```
function f(n: integer): integer;
  begin
    if n > 100 then result := n-10
      else result := f(f(n+11));
  end;
```

Вычислить f(106), f(99), f(85). Какие вообще значения принимает эта функция?

15. Описать рекурсивную функцию, отыскивающую минимальный элемент в массиве.

16. Описать рекурсивную (логическую) функцию, проверяющую, является ли симметричной часть строки, начинающаяся *i*-ым и заканчивающаяся *j*-ым элементом.

17. Описать рекурсивную функцию без параметров, подсчитывающую сумму заданных во входном файле положительных вещественных чисел. Признаком окончания ввода (последовательности) является отрицательное число.

18. Описать рекурсивную функцию без параметров, подсчитывающую количество цифр в тексте, заданном во входном файле (текст заканчивается точкой).

19. Дана последовательность ненулевых целых чисел, за которой следует 0. Напечатать сначала все отрицательные числа этой последовательности, а затем – все положительные (порядок – любой).

20. Во входном файле без ошибок записано логическое выражение следующего вида:

```
<логическое выражение> ::= true | false | <операция> (<операнды>)
<операция> ::= not | and | or
<операнды> ::= <операнд> | <операнд>, <операнды>
<операнд> ::= <логическое выражение>
```

(У операций and и or может быть любое число операндов, у not – только один). Ввести это выражение и вычислить его значение. Например, and(or(false, not(false)), true, not(true)) –> false

21. Во входном файле задан текст, за которым следует точка. Проверить, удовлетворяет ли его структура следующему определению:

```
<текст> ::= <элемент> | <элемент> <текст>
<элемент> ::= a | b | (<текст>) | [<текст >] | {< текст >}
```

23. Имеется *n* населенных пунктов, перенумерованных от 1 до *n*. Некоторые пары пунктов соединены дорогами. Написать рекурсивную функцию, определяющую, можно ли попасть по этим дорогам из пункта *i* в пункт *j*. Информация о дорогах задается в виде последовательности пар чисел

m и n (m < n), указывающих, что пункты m и n соединены дорогой. Признак окончания последовательности – пара нулей.

5.3 Контрольные вопросы

1. Можно ли случай косвенной рекурсии свести к прямой рекурсии?

2. Является ли рекурсия универсальным способом решения задач?

3. Почему для оценки трудоемкости рекурсивного алгоритма недостаточно одного метода подсчета вершин рекурсивного дерева?

6 РЕКУРСИЯ С ВОЗВРАТОМ

6.1 Основные понятия

Во многих случаях решение задачи получается в результате полного перебора всех возможных вариантов решения задачи.

Например, задача линейного поиска — в случае когда искомого объекта в совокупности нет, то результат получается только после полного перебора всех её элементов. Класс сложности подобных задач является полиномиальным.

Однако, существует класс задач, в которых получить решение за приемлемое время не удается. Так, решения задач о Ханойских башнях и о размещениях тоже получаются только как результат полного перебора всех возможных вариантов, но у них класс сложности экспоненциальный.

В первом примере полный перебор не доставляет проблем. Но в последних двух примерах этот полный перебор уже катастрофически увеличивает время получения решения. К сожалению, ни в задаче о получении всех размещений, ни в более простой задаче о Ханойских башнях уйти от полного перебора, приводящего к экспоненциальной сложности, невозможно.

Вместе с тем существует множество задач, в которых пользуясь специальными приёмами можно избежать полного перебора и уйти от экспоненциальной сложности.

Одним из таких специальных приёмов является применение ***алгоритмов с возвратом,*** которые фактически представляют собой реализацию одного из вариантов метода проб и ошибок.

Общая схема организации алгоритмов с возвратом.

1. Решение задачи получается в результате выполнения нескольких шагов.
2. На каждом шаге осуществляется полный, линейный перебор всех допустимых, возможных вариантов действий.
3. После выбора очередного допустимого варианта действий фиксируются все последствия, вытекающие из такого выбора.

4. Осуществляется рекурсивный вызов вперёд подпрограммы для выполне-ния по точно такой же схеме всех последующих шагов.

5. В реализации алгоритма с возвратом важную роль играет специальная величина — формальный параметр, который играет роль индикатора успешного выбора варианта действий на текущем шаге.

6. Если в результате сделанного на текущем шаге выбора на одном из последующих шагов (на одном из следующих уровней рекурсивного вызова) не окажется ни одного допустимого варианта действий, то такой параметр должен возвратить на текущий шаг сообщение об ошибке. В этом случае на текущем шаге нужно отменить все зафиксированные последствия выбора и перейти к выбору следующего допустимого варианта.

7. Если же на каждом последующем шаге, вплоть по последнего существуют допустимые варианты действий, такой параметр должен возвратить на текущий шаг сообщение об успешном решении задачи.

Задача решена успешно, поскольку на каждом шаге от первого до последнего выбраны допустимые варианты действий.

В качестве параметра — индикатора можно взять переменную логического типа и договориться, что значение false у этой переменной означает что произошла ошибка и допустимых вариантов нет.

Естественно в этом алгоритме, как и в любом рекурсивном алгоритме необходимо контролировать достижение предельного уровня глубины рекурсии (достижение последнего шага) и осуществлять выход на терминальную ветвь.

6.2 Примеры решения задач

*Задача***.** У игрока имеется набор костей домино (не обязательно полный). Найти последовательность выкладывания этих костей таким образом, чтобы получившаяся в результате цепочка была максимальной длины.

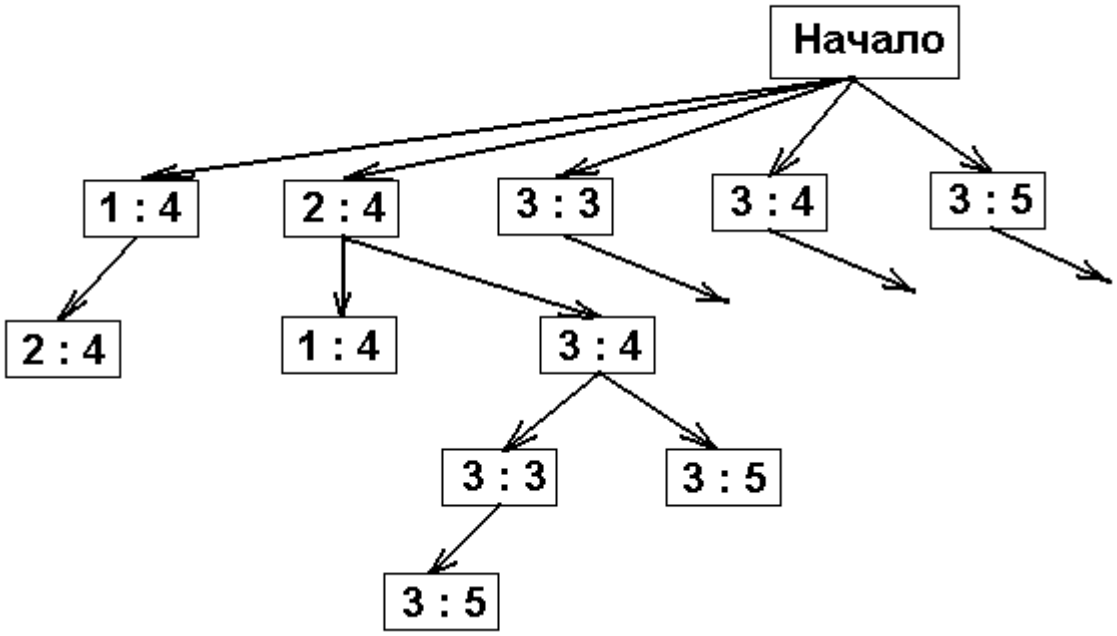

Рис.7. Схема поиска решений

Решение

```
const n=28; // максимальное количество костей домино
  type kost_domino=record // кость домино
      left,right:0..6;  //значения
      used:boolean;  //использована ли кость
  end;

var
f: array[1..n] of kost_domino;  //набор костей домино у игрока
kol:integer; //количество костей у игрока
CurPos: array[1..n] of 0..n;//текущий порядок выставления костей
BestPos: array[1..n] of 0..n;//лучший порядок выставления костей
m:integer; //количество костей в лучшем варианте

// рекурсивный алгоритм с возвратом постановки очередной кости в
//цепочку с границами left,right
procedure Step(k,left,right:integer);
var i:integer;
begin
    if k-1>m then //если текущий вариант лучший по длине
    begin                //мы его сохраняем
      m:= k-1;
      for i := 1 to kol do
          BestPos[i]:= CurPos[i];
    end;

    for i:=1 to kol do//перебор всех вариантов k-го шага
      if not f[i].used then
      begin
          //пробуем поставить i-ую кость
```

```
          f[i].used:=true;
          CurPos[k]:=i;
          if f[i].left=left then // если левую сторону можно
//приложить к цепочке слева
            Step(k+1,f[i].right,right); //то пробуем следующую
          if f[i].left=right then // если левую сторону можно
//приложить к цепочке справа
            Step(k+1,left,f[i].right);
          if f[i].right=left then  //если правую сторону можно
//приложить к цепочке слева
            Step(k+1,f[i].left,right);
          if f[i].right=right then  //если правую сторону
//можно приложить к цепочке справа
            Step(k+1,left,f[i].left);
          f[i].used:=false;
          CurPos[k]:=0;
      end;
end;


 var i:integer;
begin
  readln(kol);
  for i := 1 to kol do
  begin
    readln(f[i].left, f[i].right);
    f[i].used:=false;
    CurPos[i]:=0;
  end;
m:=0;
 for i := 1 to kol do
  begin
    CurPos[1]:=i; //выставляем i-ую кость в качестве первой
    f[i].used:=true; //помечаем как использованную
    Step(2,f[i].left,f[i].right);//вызов процедуры выставления
                                 //2-ой, 3-ей и т.д. кости
    f[i].used:=false;
  end;
 for i := 1 to m do
 writeln(i:4,f[BestPos[i]].left:4,f[BestPos[i]].right:4);

 readln;
end.
```

6.3 Задачи

1. Во входном файле задан текст, за которым следует точка. Проверить, удовлетворяет ли его структура следующему определению:

```
<текст> ::= <элемент> | <элемент> <текст>
<элемент> ::= a | b | (<текст>) | [<текст >] | {< текст >}
```

2. Имеется *n* населенных пунктов, перенумерованных от 1 до *n*. Некоторые пары пунктов соединены дорогами. Написать рекурсивную функцию, определяющую, можно ли попасть по этим дорогам из пункта *i* в пункт *j*. Информация о дорогах задается в виде последовательности пар чисел m и n (m < n), указывающих, что пункты m и n соединены дорогой. Признак окончания последовательности – пара нулей.

3. Напишите (рекурсивную) программу, которая печатала бы все перестановки чисел от 1 до n по одному разу.

4. Напишите (рекурсивную) программу, которая печатала бы все возрастающие последовательности длины *m*, элементами которых являются натуральные числа от 1 до *n* (*m* <= *n*).

5. Напишите (рекурсивную) программу, которая перечисляла бы все представления положительного целого числа *n* в виде суммы последовательных невозрастающих целых положительных слагаемых.

6. Дана доска *n* x *n*. Конь, который ходит согласно шахматным правилам, помещается на поле с начальными координатами x0, y0. Нужно покрыть всю доску ходами коня, т.е. вычислить обход доски (если он существует), из $n^2 - 1$ ходов такой, что каждое поле посещается ровно один раз. Напишите программу, которая промаркирует клетки доски номерами ходов. В начальную клетку поставьте 0.

7. Обобщите решение задачи о 8 ферзях так, чтобы находить не одно из решений, а все возможные решения данной задачи.

8. Получите все расстановки 8 ладей на шахматной доске, при которых ни одна ладья не угрожает другой.

9. Найдите такую расстановку пяти ферзей на шахматной доске, при которой каждое поле будет находиться под ударом одного из них.

10. Имеется *n* предметов, веса которых равны A1, A2, …, AN. Разделите эти предметы на две группы так, чтобы общие веса двух групп были максимально близки.

11. Игра «Ханойские башни» состоит в следующем: есть три стержня. На первый из них надета пирамидка из N колец (большие кольца снизу, меньшие сверху). Требуется переместить кольца на другой стержень. Разрешается перекладывать кольца со стержня на стержень, но класть большее кольцо поверх меньшего нельзя. Составить программу, указывающую требуемые действия.

12. Из 92 решений, получающихся с помощью решения задачи 7, только 12 по существу различны. Все другие решения можно получить с помощью осевой или центральной симметрии. Напишите программу для получения именно этих 12 решений. Обратите внимание, что поиск на первой вертикали можно ограничить позициями 1-4.

13. Имеется *n* населенных пунктов, перенумерованных от 1 до *n*. Некоторые пары пунктов соединены дорогами. Написать рекурсивную функцию, определяющую, можно ли попасть по этим дорогам из пункта *i* в пункт *j*. Информация о дорогах задается в виде последовательности пар чисел m и n (m < n), указывающих, что пункты m и n соединены дорогой. Признак окончания последовательности – пара нулей.

14. Напишите (рекурсивную) программу, которая печатала бы все перестановки чисел от 1 до n по одному разу.

15. Напишите (рекурсивную) программу, которая печатала бы все возрастающие последовательности длины *m*, элементами которых являются натуральные числа от 1 до *n* (*m* <= *n*).

16. Напишите (рекурсивную) программу, которая перечисляла бы все представления положительного целого числа *n* в виде суммы последовательных невозрастающих целых положительных слагаемых.

17. Дана доска *n* x *n*. Конь, который ходит согласно шахматным правилам, помещается на поле с начальными координатами x0, y0. Нужно покрыть всю доску ходами коня, т.е. вычислить обход доски (если он существует), из $n^2 - 1$ ходов такой, что каждое поле посещается ровно один раз.

Напишите программу, которая промаркирует клетки доски номерами ходов. В начальную клетку поставьте 0.

18. Получите все расстановки 8 ладей на шахматной доске, при которых ни одна ладья не угрожает другой.

19. Найдите такую расстановку пяти ферзей на шахматной доске, при которой каждое поле будет находиться под ударом одного из них.

20. Имеется *n* предметов, веса которых равны A1, A2, …, AN. Разделите эти предметы на две группы так, чтобы общие веса двух групп были максимально близки.

21. Из 92 решений, получающихся с помощью решения задачи 7, только 12 по существу различны. Все другие решения можно получить с помощью осевой или центральной симметрии. Напишите программу для получения именно этих 12 решений. Обратите внимание, что поиск на первой вертикали можно ограничить позициями 1-4.

22. В трехмерном пространстве задано множество точек. Найти разбиение этого множества на два непустых непересекающихся множества, чтобы их центры тяжести находились как можно ближе друг к другу.

23. Дан набор слов. Составить из них цепочку максимальной длины по количеству слов (или по количеству букв). Цепочка образуется, если первая буква следующего слова совпадает с последней буквой предыдущего слова. Повторно использовать слова нельзя.

24. Построить такой многоугольник (не обязательно выпуклый) с вершинами в заданном множестве, периметр которого максимален.

25. Найти минимальное множество прямых, на которых можно разместить все точки заданного множества

26. По двум конвейерам двигаются молочные бутылки. Для каждой бутылки известно время заполнения и закупоривания. Найти расстановку бутылок, при которой время обработки минимально.

6.4 Контрольные вопросы

1. В чем проявляется рекурсивность метода перебора с возвратом?

2. Почему полный метод перебора с возвратом гарантирует отыскание всех решений задачи?

3. Поясните, почему данные описания характеризуют описание действий над ферзем в контексте модели шахматной доски:

- в позицию (i,j) можно поставить ферзь, если $ho_i+du_i+j+dd_{n+i-j}=0$;
- поставить ферзь в позицию (i,j) равносильно присваиваниям: $ho_i=1$, $du_{i+j}=1$, $dd_{n+i-j}=1$;
- убрать ферзь из позиции (i,j) равносильно присваиваниям: $ho_i=0$, $du_{i+j}=0$, $dd_{n+i-j}=0$.

7 ДЕРЕВЬЯ

7.1 Основные понятия

Деревом называется структура данных, элементы которой называемые вершинами (узлами) связаны отношениями подчиненности, когда одному элементу может быть подчинено несколько, но при этом сам он может быть подчинен только одному. В структуре имеется только один элемент, не подчиняющийся никаким другим — ***корень*** дерева.

Дерево может рассматриваться как частный случай графа. Тогда можно сказать, что ***деревом*** называется связный ориентированный ациклический граф. Узлы, находящиеся на верхнем уровне и имеющие подчинённые узлы, считаются родительскими или порождающими, а узлы, находящиеся на нижнем уровне, считаются дочерними, потомками или порождёнными. В общем случае каждый узел дерева может иметь любое количество дочерних узлов.

Узлы, не умеющие потомков, принято называть ***листьями***.

Основные операции над деревьями:

- создание дерева;
- включение нового элемента;
- уничтожение структуры;
- исключение элемента;
- полный перебор структуры;
- поиск элемента.

В информатике широко применяются несколько частных случаев, несколько разновидностей деревьев: ***бинарные*** деревья, ***идеально сбалансированные*** деревья, ***АВЛ*** деревья, деревья ***поиска***, ***B***-деревья и некоторые другие виды.

Бинарными называются деревья, каждая вершина которых может иметь не более двух дочерних.

Бинарное дерево – это конечное множество узлов, которое либо является пустым, либо состоит из корня и двух непересекающихся бинарных деревьев, которые называются левым и правым поддеревьями данного корня (рисунок 6).

Согласно этому определению бинарное дерево имеет *рекурсивную* структуру.

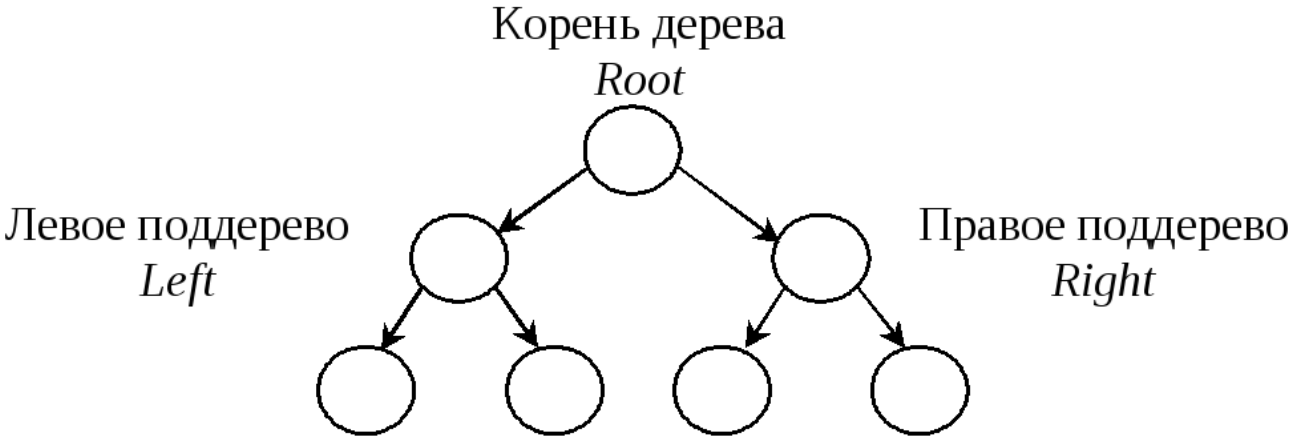

Рис.8. Структура бинарного дерева

С каждым узлом дерева обычно связана некоторая полезная информация, для обработки которой собственно и создается дерево. Эта информация может храниться в отдельном поле узла или может быть связана с узлом ссылкой. Для удобства обработки каждый узел дерева обычно некоторым образом обозначают и это обозначение принято называть ***ключом вершины*** (узла). Чаще всего для в качестве ключей используются целые числа. К ключам обычно предъявляется требование уникальности: один и тот ключ не может быть у двух и более вершин.

Описание типа для работы с деревьями:

```
type
  T_item = integer; // здесь может быть любой тип
  P_Node = ^T_Node;
  T_Node = record
             info: T_item;     //полезная информация
             key: integer;             //ключ
             left, right: P_Node;    //левое и правое поддерево
           end;
var root: P_Node; // корень дерева
```

Существует класс задач, в которых для достижения результата нужно посетить все узлы бинарного дерева. Это, например, задача определения суммы всех информационных элементов, связанных с узлами дерева, задача

определения наибольшего элемента, печать значений всех элементов и т.д. Такой обход может выполняться в любом бинарном дереве, а не только в дереве поиска, в котором упорядоченность ключей используется для сокращения пути, проходимого по дереву и для уменьшения сложности алгоритма.

На практике используется несколько различных схем обхода. К основным, базовым схемам обхода относят ***прямой*** (в ширину), обратный (центрированный, симметричный) и концевой (в глубину) обходы (рисунок 7).

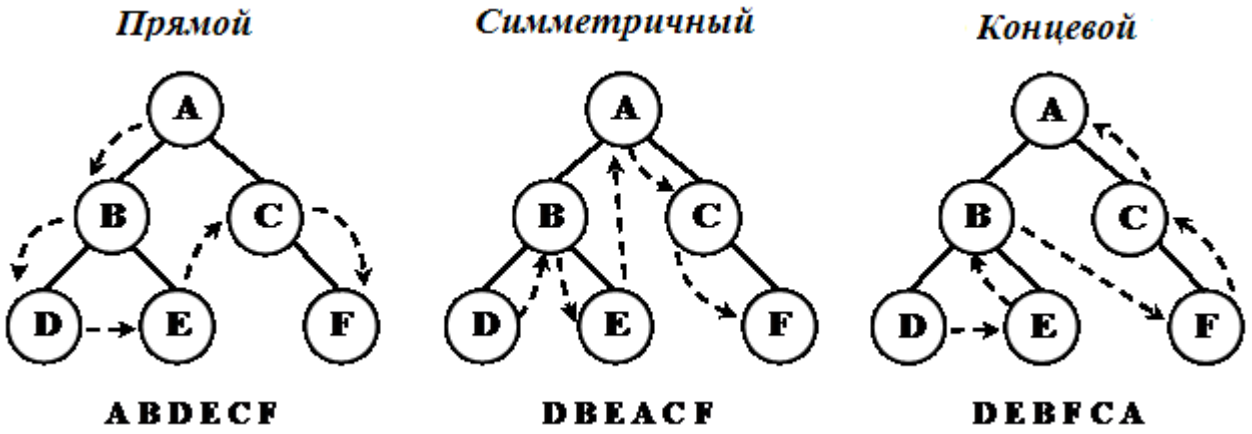

Рис.9. Схемы обхода дерева

```
// вспомогательная процедура
procedure PrintKey(t: P_Node);
begin
  write(t^.key: 5);
end;

        //Прямой обход: корень, левое поддерево, правое поддерево.
                  // обход сверху вниз          К-Л-П
procedure preorder(t: P_Node);
begin
  if t <> nil
    then begin
      PrintKey(t);    // обработать корневой узел
      preorder(t^.left);// обойти левое поддерево в нисходящем порядке
      preorder(t^.right);//обойти правое поддерево в нисходящем пор-ке
    end;
end;

       //Концевой обход: левое поддерево, правое поддерево, корень.
                  // обход снизу вверх          Л-П-К
procedure postorder(t: P_Node);
begin
  if t <> nil
    then begin
     postorder(t^.left);//обойти левое поддерево в восходящем порядке
     postorder(t^.right);//обойти правое поддерево в восходящем пор-ке
```

```
     PrintKey(t); // обработать корневой узел
    end;
end;



     // Симметричный обход: левое поддерево, корень, правое поддерево.
                // обход слева направо(смешанный)    Л-К-П
procedure inorder(t: P_Node);
begin
  if t <> nil
    then begin
      inorder(t^.left);//обойти левое поддерево в смешанном порядке
      PrintKey(t); //обработать корневой узел
      inorder(t^.right); //обойти правое поддерево в смешанном порядке
    end;
end;
```

Одна из наиболее часто встречающихся операций в дереве — поиск узла с заданным ключом. Чтобы упростить работу с деревом в задачах такого типа используют особую разновидность бинарного дерева ***—дерево поиска***.

На ключи узлов дерева поиска накладывается требование: ключ любого родительского узла дерева должен быть больше ключа левого порождённого узла и при этом меньше ключа правого порождённого узла.

Выполнение этого условия требует специального порядка формирования дерева, специального порядка включения в дерево новых вершин.

Общая идея алгоритма включения: если в дерево нужно включить новую вершину с ключом k, то начиная с корня дерева в зависимости от результатов сравнения k с ключом в каждой очередной вершине выбирается для дальнейшего движения левое или правое поддерево. Если ключ k меньше ключа узла, то выбирается левое поддерево, если больше — правое. Если ключи совпадают, то включение невозможно. Поиск места включения новой вершины завершается при попадании в лист.

Построение нового дерева сводится к включению в дерево новых узлов, при этом первый узел включается в пустое дерево и становится его корнем.

Удобным является оформление этих действий в виде процедуры включения нового узла в дерево.

Анализ выполняемых при включении действий показывает, что перед включением нужно найти место включения нового узла в дерево так, чтобы соблюдалось требуемое соотношение между ключами узлов дерева поиска.

Анализ действий, выполняемых при поиске места включения нового узла, показывает, что такой поиск по своей сути аналогичен поиску в дереве узла, имеющего заданный ключ.

Таким образом, нужно построить процедуру поиска в дереве узла, имеющего точно такой же ключ, что и включаемый узел. Если такой узел найдется, то делается вывод о невозможности включения в дерево узла с этим ключом. Если такого узла не окажется, то поиск должен привести нас к тому месту, в которое следует вставить новый узел.

Поиск продолжается пока искомый узел ещё не найден и при этом ещё имеются узлы для просмотра. Анализ каждого очередного узла (в том числе корня) сводится к сравнению искомого ключа с ключом узла. Если ключи совпадают, то добавление невозможно. Если искомый ключ больше, чем ключ вершины, то следует сместиться в правое поддерево. Если искомый ключ меньше, чем ключ вершины, то следует сместиться в левое поддерево. Узел, в котором прекратился поиск, может рассматриваться как место вставки нового узла с ключом равным искомому.

Формальными параметрами процедуры являются: t — указатель на изменяемое дерево, (параметр-переменная), ключ вставляемого узла k — параметр-значение целого типа, полезная информация связанная с вставляемым узлом E — параметр-значение типа T_item.

Новый узел создается стандартным образом, его поля заполняются нужными значениями, а затем он прикрепляется в выбранное место дерева.

```
          // вставка элемента в дерево (рекурсивная процедура)
procedure InsertInTree(var t: P_Node; k: Integer);
begin
  if t = nil     // если найдено место для нового узла
    then begin
           new(t);     // распределяем память
           t^.left := nil;   // устанавливаем пустые ссылки для левого и
           t^.right := nil;   // правого поддеревьев
           t^.key := k;     // заполняем ключевое поле
```

```
            {а здесь не помешало бы заполнить информационное поле}
          end   // then
{иначе определяем, в каком поддереве узла t должен располагаться элемент
с таким ключом}
    else if k < t^.key // если новый ключ меньше или равен текущему
           then InsertInTree(t^.left, k) // вставим в левое поддерево
           else InsertInTree(t^.right, k); // а иначе – в правое
end;
```

Автором алгоритма удаления узла из дерева поиска является Н.Вирт.

Будем считать, что удаляемый узел задается своим ключом, а дерево — ссылкой на его корень. Чтобы удалить узел нужно сначала спуститься от корня дерева к удаляемому узлу, а затем исключить его из структуры дерева.

Спуск к нужному узлу, можно организовать почти так же как и в алгоритме решения задачи вставки InsertInTree. Однако нужно помнить об особенностях удаления из динамических структур данных. Например, чтобы удалить звено из линейного списка нужно скорректировать поле связи у звена, к которому присоединено удаляемое.

При исключении узла могут встретиться три случая:

1) нужно удалить лист;
2) нужно удалить узел, который имеет только один порождённый;
3) нужно удалить узел, который имеет два порождённых.

Удаление в двух первых случаях организуется точно также, как исключается элемент из линейного однонаправленного списка. Запоминается ссылка на удаляемый узел, корректируется ссылка у родительского узла, а затем освобождается память.

Для сохранения необходимого свойства ключей дерева поиска Н.Вирт предложил выбирать для замены самый правый узел из левого поддерева или самый левый узел из правого поддерева, т.к самый правый из левого поддерева является наибольшим в этом поддереве, а самый левый в правом поддереве является наименьшим в этом поддереве.

Таким образом, ***результат выполнения операции исключения узла из дерева поиск является неоднозначным***, он зависит от выбора заменяющего узла.

Кроме того, Н.Вирт предложил для выполнения операций по удалению узла использовать не вспомогательную внешнюю, по отношению к дереву ссылку, а непосредственно поле узла.

Идея Вирта состоит в том, чтобы само поле ссылки узла выступало в качестве фактического параметра переменной для рекурсивных процедур поиска удаляемого и заменяющего узлов дерева.

Н. Вирт предложил построить рекурсивную подпрограмму, которая в рекурсивных ветвях осуществляет спуск от корня дерева к удаляемому узлу. В терминальной ветви этого алгоритма определяется тип удаляемого узла и в зависимости от этого организуется удаление.

Обсуждаемая подпрограмма может быть реализована только как процедура. Назовём её Delete_Tree. Формальными параметрами этой процедуры являются: параметр переменная t типа P_Node, указывающий на дерево (поддерево), и параметр-значение целого типа k, определяющий ключ удаляемого узла. Для освобождения памяти потребуется вспомогательная ссылка q типа P_Node на удаляемый узел. Удаляемый узел найден, причём формальный параметр t процедуры показывает на найденный узел. t обозначает поле ссылки в родительском узле дерева.

В рекурсивной ветви нужно спускаться по правому поддереву, пока правое поле ссылки в узле, на который показывает формальный параметр, не окажется пустой ссылкой. Это и будет означать, что в левом поддереве достигнут самый правый узел, то есть узел, который может заменить удаляемый.

В соответствии с записанным в основной подпрограмме вызовом процедуры Del она должна иметь один формальный параметр-переменную (пусть он называется w), который показывает на корень левого поддерева удаляемого узла. Кроме того, это должна быть рекурсивная процедура.

В терминальной ветви (то есть после определения заменяющего узла) нужно переписать ключ и информационное поле заменяющего узла в соответствующие поля удаляемого узла.

Для организации такой замены нужно иметь ссылку на заменяющий узел (это ссылка w , которая найдена в рекурсивной ветви) и ссылку на удаляемый узел (это ссылка q, найденная в основной процедуре удаления). После чего следует удалить тот узел, который выступает в роли заменяющего. Для этого нужно: изменить значение q на значение w (для освобождения памяти) и скорректировать поле в родительском узле заменяющего.

```
// процедура удаления элемента
procedure Delete_Tree(var t: P_Node; k: Integer);
var q: P_Node;

procedure Del(var w: P_Node); // поиск самого правого элемента
begin
  if w^.right <> nil then Del(w^.right)
                     else begin
                            q := w;
 // запоминаем адрес, чтобы потом удалить этот элемент
                            t^.key := w^.key;
                            w := w^.left;
                     end;
end;

begin
  if t <> nil then
    if k < t^.key then Delete_Tree(t^.left, k)
    else if k > t^.key then Delete_Tree(t^.right, k)
         else begin
           q := t;
           if t^.right = nil then t := t^.left
           // правого поддерева нет
           else if t^.left = nil then t := t^.right
          // левого поддерева нет
                else Del(t^.left);
               // находим самый правый элемент в левом поддереве
           dispose(q);
         end;
end;
```

7.2 Задачи

1. Написать процедуру или функцию, которая:

а) вычисляет сумму элементов непустого дерева

б) находит величину наибольшего элемента непустого дерева

в) печатает элементы из всех листьев дерева

2. Написать логическую функцию equal (T1, T2), проверяющую на равенство деревья T1 и T2.

3. Написать процедуру copy (T, T1), которая строит T1 – копию дерева T.

4. Описать логическую функцию same(T), определяющую, есть ли в дереве T хотя бы два одинаковых элемента.

5. Написать процедуру удаления (из существующего бинарного дерева) всех отрицательных элементов.

6. Дан текстовый файл. Использовать дерево для определения частоты вхождения каждого слова в текст (подумайте, какого типа должно быть ключевое поле, какого – информационное). Считайте, что длина слова не может превосходить 255 символов.

7. Напомним, что двоичное дерево считается идеально сбалансированным, если для каждой его вершины количество вершин в левом и правом поддеревьях различается не более чем на 1. Нужно написать функцию проверки идеальной сбалансированности двоичного дерева.

8. В задаче 6 (о текстовом файле) определить количество вершин дерева, содержащих слова:

а) начинающиеся на одну и ту же букву;

б) являющиеся палиндромами;

в) содержащие все гласные буквы латинского алфавита.

9. В файле даны N целых чисел в двоичной системе счисления (M бит каждое). Построить двоичное дерево, в котором числам соответствуют листья дерева, а путь по дереву определяет значение информационного поля этого листа.

7.3 Контрольные вопросы

1. С чем связана популярность использования деревьев в программировании?

2. Можно ли список отнести к деревьям?

3. Какие данные содержат адресные поля элемента бинарного дерева?

4. Куда может быть добавлен элемент в бинарное дерево?

5. Чем отличаются, с точки зрения реализации алгоритма, прямой, симметричный и обратный обходы бинарного дерева?

8 СБАЛАНСИРОВАННЫЕ ДЕРЕВЬЯ

8.1 Основные понятия

Некоторую информацию в дереве легче найти, если оно низкое и ветвистое. В этом случае каждый шаг позволяет отсечь довольно много вариантов сразу. Такое «правильное» дерево называют сбалансированным. Напомним, что дерево называется идеально сбалансированным, если число вершин в его левых и правых поддеревьях отличается не более чем на единицу. Построение такого дерева принципиально не очень сложная задача. Если заранее известно число вершин N, то одну из вершин нужно выбрать в качестве корня, а оставшиеся распределить между левым и правым поддеревьями (с половинным числом вершин в каждом). Однако, восстановление идеальной сбалансированности дерева после включения в него элемента с произвольным значением ключа – довольно сложная и, главное, «невыгодная» операция. Поэтому на практике используются менее строгие критерии сбалансированности. Одно из таких определений было предложено Г.М. Адельсоном-Вельским и Е.М. Ландисом (1962). Их критерий сбалансированности был сформулирован следующим образом:

дерево называется сбалансированным тогда и только тогда, когда высоты двух поддеревьев каждой из его вершин отличаются не более чем на единицу.

Деревья, удовлетворяющие такому условию, часто называют АВЛ – деревьями (по первым буквам фамилий их «разработчиков»). Заметим, что идеально сбалансированное дерево всегда является АВЛ-деревом. Адельсон-Вельский и Ландис доказали теорему, которая утверждает, что сбалансированное дерево никогда не будет более чем на 45% выше, чем соответствующее идеально сбалансированное дерево.

Рассмотрим операции, который могут потребоваться для восстановления сбалансированности АВЛ-деревьев после включения или удаления элементов. Очевидно, что это должны быть преобразования, не меняющие множества

ключей в вершинах, не нарушающие упорядоченности, но вместе с тем способствующие лучшей сбалансированности.

При включении в дерево нового узла возможны следующие варианты:

а) высоты левого и правого поддеревьев были равны, включение нового узла изменяет высоту одного из них на единицу, при этом критерий сбалансированности не нарушается;

б) левое и правое поддеревья имеют разные высоты, новый узел включается в поддерево с меньшей высотой, критерий сбалансированности не нарушается;

в) левое и правое поддеревья имеют разные высоты, новый узел включается в поддерево с большей высотой, критерий сбалансированности нарушается, требуется перестройка дерева.

Оказывается, имеются всего два существенно различающихся варианта того, каким станет дерево, и, соответственно, два способа ребалансировки.

Первый вариант (рисунок 10): вершина A является корнем поддерева, вершина B – корень правого поддерева вершины A. Левое поддерево вершины A обозначено как P (структура этого поддерева непринципиальна, оно рассматривается просто как множество вершин). Левое и правое поддеревья вершины B обозначены соответственно как Q и R (из рисунка ясно, что новый элемент был добавлен как раз в поддерево R). Поскольку дерево упорядочено, можно записать следующее:

$$P < A < Q < B < R,$$

где P, Q, R следует читать как «любая вершина из множества P (Q, R соответственно). Перестроенное дерево должно удовлетворять этому неравенству (это значит, что перемещать вершины допустимо только по вертикали, но не по горизонтали). Нужная конфигурация вершин изображена справа.

Такое преобразование называют ***малым правым поворотом*** или вращением (правым, поскольку существует симметричное ему левое, а малым – потому что есть и большое).

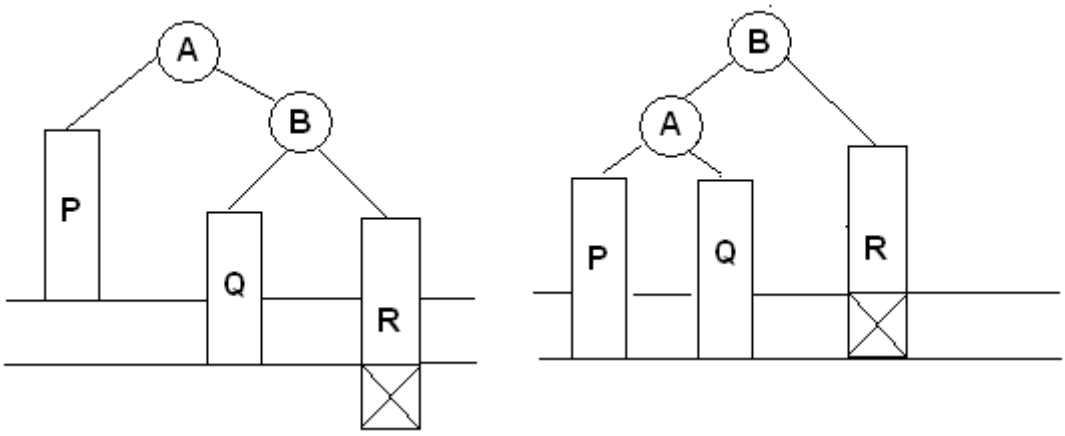

Рис. 10 а) б). Малый правый поворот (малое правое вращение)

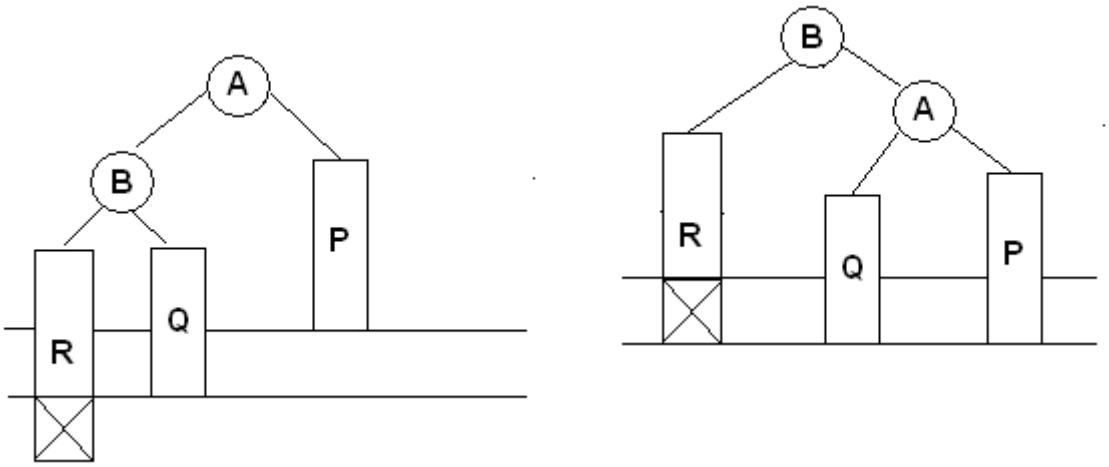

Рис. 11 а) б). Малый левый поворот (малое левое вращение)

Второй вариант изображен на рисунке 12. Поддерево с корнем в A имеет левое поддерево P и правое поддерево с вершиной B. Поддерево с вершиной B, в свою очередь, имеет левое поддерево с вершиной C и правое поддерево S. Наконец, поддерево с вершиной C имеет левое и правое поддеревья Q и R, добавление элемента в любое из которых нарушает сбалансированность дерева. Для определенности будем считать, что элемент добавился в поддерево R. Возможность добавления элемента в поддерево Q обозначена штриховкой.

Как можно видеть, соблюдается следующая цепочка неравенств:

$$P < A < Q < C < R < B < S,$$

где по-прежнему P, Q, R и S символизируют любой элемент из этого множества. Эта же цепочка неравенств соблюдается и при конфигурации

вершин, показанной на рисунке 13. Такое преобразование называют ***большим правым поворотом*** (или вращением). Большой левый поворот определяется аналогичным образом.

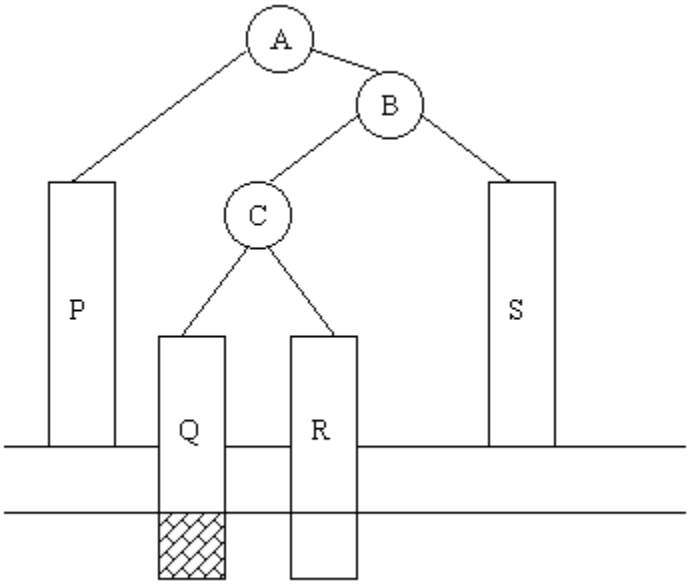

Рис.12 Нарушение сбалансированности после добавления элемента

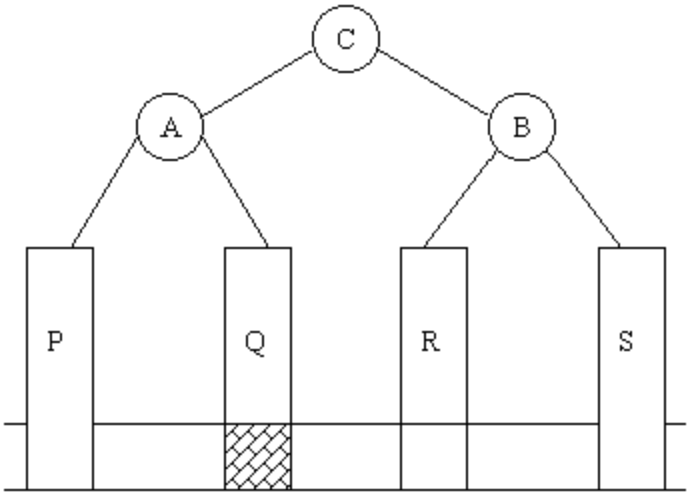

Рис.13. Большой правый поворот (большое правое вращение)

Аналогичные преобразования требуются, чтобы восстановить сбалансированность дерева после удаления из него какого-то элемента.

Можно доказать следующие утверждения.

1. Дерево, сбалансированное всюду, кроме корня, в котором разница высот равна 2 (т.е. левое и правое поддеревья корня сбалансированы и их высоты отличаются на 2), может быть превращено в сбалансированное одним из четырех описанных выше преобразований, причем высота его останется прежней или уменьшится на 1.

2. Если в сбалансированном дереве X одно из его поддеревьев Y заменили на сбалансированное дерево Z, причем высота Z отличается от высоты Y не более чем на 1, то полученное в результате такой «прививки» дерево можно превратить в сбалансированное с помощью описанных выше поворотов (причем число поворотов не превосходит высоты, на которой делается «прививка»).

3. (Следствие из 2). После добавления или удаления листа из сбалансированного дерева восстановить сбалансированность можно после нескольких поворотов, причем их число не превосходит высоты дерева.

Нужно отметить, что алгоритм включения и балансировки практически полностью определяется способом хранения информации о сбалансированности дерева. Здесь возможны разные варианты между двумя крайними случаями. Одна из крайностей состоит в том, что явно такая информация не хранится, а вычисляется для каждого узла при каждом обращении к нему. Такой подход, конечно, требует больших затрат по времени. Другая же крайность – сохранять показатель сбалансированности в каждом узле. Это требует переопределения типа узла:

```
type
  T_item = integer; // здесь может быть любой тип
  P_Node = ^T_Node;
  T_Node = record
             count: T_item;
             key: integer;
             left, right: P_Node;
             bal: -1..1;
           end;
```

Показатель сбалансированности (bal) интерпретируется как разность между высотами его правого и левого поддеревьев и может принимать значения в диапазоне от –1 до 1.

Кроме этого, принято вводить еще переменную логического типа, которая показывает, увеличилась ли высота поддерева.

Действительно, рассмотрим, к примеру, процедуру вставки узла. Потребуется, во-первых, пройти по дереву в поисках подходящего места для вставки этого узла. Второе действие будет состоять собственно во включении узла в дерево и корректировке показателя сбалансированности. Наконец, в-третьих, придется пройти обратно по пути поиска и проверить показатель сбалансированности для каждого узла. Именно для этих целей и нужна подобная переменная.

Заметим также, что включение «обратного прохода» в обычную процедуру вставки элемента в дерево не должно вызывать больших трудностей, поскольку она рекурсивная.

Преобразования вращения после вставки очередного элемента в дерево в виде процедуры может быть записана следующим образом:

```
1. procedure search(x: integer; var p: P_Node; var h: boolean);
2. var p1,p2:P_Node;
3. begin
4. if p=nil then
5.                 begin
6.                 new(p);
7.                 h:=true;
8.                 p^.key:=x;
9.                 p^.count:=1;  // включение
10.                     p^.left:=nil;
11.                     p^.right:=nil;
12.                     p^.bal:=0
13.                     end
14.    else if p^.key>x then
15.       begin
16.         search(x,p^.left,h);
17.         if h then          //выросла левая ветвь
18.           begin
19.             case p^.bal of
20.             1: begin p^.bal:=0; h:=false;end;
21.             0:p^.bal:=-1;
22.            -1: begin            //балансировка
23.                     p1:=p^.left;
24.                        if p1.bal=-1 then
25.                                    //однократный LL-поворот
26.                                       begin
27.
  p^.left:=p1^.right;
```

```
                                        p1^.right:=p;
                                        p^.bal:=0;
                                        p:=p1;
                                        end
                              else      //двойной LR-поворот
                                        begin
                                   p2:=p1^.right;
                                   p1^.right:=p2^.left;
                                   p2^.left:=p1;
                                   p1^.left:=p2^.right;
                                   p2^.right:=p;
                                     if p2^.bal=-1 then
                               p^.bal:=1 else p^.bal:=0;
                                        if p2^.bal=1 then
                               p1^.bal:=-1 else p1^.bal:=0;
                                        p:=p2;
                                        end;
                  end;
               end;
            end;
      end
    else if p^.key<x then
    begin
    search(x,p^.right,h);
    if h then           //выросла правая ветвь
            begin
            case p^.bal of
            -1: begin p^.bal:=0; h:=false; end;
            0: p^.bal:=1;
            1: begin              //балансировка
                    p1:=p^.right;
                    if p1^.bal=1 then//однократный RR-поворот
                                  begin
                                  p^.right:=p1^.left;
                                  p1.left:=p;
                                  p^.bal:=0;
                                  p:=p1;
                                  end
                    else             //двойной RL-поворот
                            begin
                            p2:=p1^.left;
                            p1^.left:=p2^.right;
                            p2^.right:=p1;
                            p^.right:=p2^.left;
                            p2^.left:=p;
                            if p2^.bal=1 then p^.bal:=-1
                            else p^.bal:=0;
                            if p2^.bal=-1 then p1^.bal:=1
                            else p1^.bal:=0;
                            p:=p2;
                            end;
              p^.bal:=0; h:=false;
              end;
```

```
79.             end;
80.     end
81.     else p^.count:=p^.count+1;
82.     end;
83.     end;
```

Схема ***удаления элемента*** с учетом балансировки аналогична схеме добавления элемента. Простые случаи – удаление терминальных вершин или вершин только с одним потомком. Если же от исключаемой вершины «отходят» два поддерева, то, она заменяется на самую правую вершину ее левого поддерева. Вводится параметр-переменная h логического типа, указывающий, уменьшилась ли высота поддерева. Балансировка идет только если h – истина. Это значение присваивается переменной h при обнаружении и исключении какой-либо из вершин или уменьшении высоты какого-либо поддерева в процессе самой балансировки. Введем две (симметричные) операции балансировки в виде процедур. BalanceL используется при уменьшении высоты левого поддерева, а BalanceR – правого.

```
type
  T_item = integer; // здесь может быть любой тип
  P_Node = ^T_Node;
  T_Node = record
             count: T_item;
             key: integer;
             left, right: P_Node;
             bal: -1..1;
           end;
procedure balanceL (var p: P_Node; var h: boolean);
var p1,p2:P_Node; b1, b2: -1..1;
begin                                    // левая ветка стала короче
case p^.bal of
-1: p^.bal:=0;
0: begin p^.bal:=1; h:=false; end;
1: begin
       p1:= p^.right; b1:= p1^.bal;                  // балансировка
       if b1>=0 then                     // малый правый поворот
          begin
               p^.right:= p1^.left;
               p1^.left:=p;
               if b1=0 then
                         begin
                              p^.bal:=1;
                              p1^.bal:=-1;
                              h:=false;
                         end
                    else
```

```
                        begin
                             p^.bal:=0;
                             p1^.bal:=0;
                        end;
               p:=p1;
          end
      else                                  // большой правый поворот
          begin
               p2:= p1^.left;
               b2:= p2^.bal;
               p1^.left:= p2^.right;
               p2^.right:=p1;
               p^.right:= p2^.left;
               p2^.left:=p;
               if b2=1 then p^.bal:=-1
                    else p^.bal:=0;
               if b2=-1 then p1^.bal:=1
                    else p1^.bal:=0;
               p:=p2;
               p2^.bal:=0
          end;
end;                 //case
end;

procedure balanceR (var p: P_Node; var h: boolean);
var p1,p2:P_Node; b1, b2: -1..1;
begin                                  // правая ветка стала короче
case p^.bal of
-1: p^.bal:=0;
0: begin p^.bal:=-1; h:=false; end;
1: begin
       p1:= p^.right;
       b1:= p1^.bal;                     // балансировка
       if b1<=0 then                    // малый левый поворот
          begin
               p^. left:= p1^.right;
               p1^.right:=p;
               if b1=0 then
                        begin
                             p^.bal:=-1;
                             p1^.bal:=1;
                             h:=false;
                        end
                     else
                        begin
                             p^.bal:=0;
                             p1^.bal:=0;
                        end;
               p:=p1;
          end
      else                              // большой левый поворот
          begin
               p2:= p1^.right;
```

```
                b2:= p2^.bal;
                p1^. right:= p2^.left;
                p2^.left:=p1;
                p^.left:= p2^.right;
                p2^.right:=p;
                if b2=-1 then p^.bal:=1
                     else p^.bal:=0;
                if b2=1 then p1^.bal:=-1
                     else p1^.bal:=0;
                p:=p2;
                p2^.bal:=0
           end;
end;                 //case
end;


procedure Delete(x:integer; var p: P_Node; var h: boolean);
var q: P_Node;

procedure Del(var r: P_Node; var h: boolean);
                        // поиск самого правого элемента
begin
  if r^.right <> nil then
                begin
                     Del(r^.right)
                     if h then balanceR(r,h)
                end
                      else begin
                             q := r;
             // запоминаем адрес, чтобы потом удалить этот элемент
                             q^.key := r^.key;
               q^.count := r^.count;
                             r := r^.left;
               h:=true;
                      end;
end;

begin
  if p <> nil then
    if x < p^.key then
                begin
                     Delete(x, p^.left, h);
                     if h then balanceL(p,h);
                end
    else if x > p^.key then
                begin
                     Delete(x, p^.right, h);
                     if h then balanceR(p,h);
                end
         else
     begin
                     q := p;
                      if q^.right = nil then// правого поддерева нет
```

```
                    begin
                         p := q^.left;
                         h:=true;
                    end
               else if q^.left = nil then//левого поддерева нет
                    begin
                         p := q^.right;
                         h:=true;
                    end
                              else
                    begin
              // находим самый правый элемент в левом поддереве
                         Del(q^.left,h);
                         if h then balanceL(p,h)
                    end
        dispose(q);
      end;
end;
```

8.2 Задачи

1. Рассмотрите формирование сбалансированного дерева на следующих входных данных: 4, 5, 7, 2, 1, 3, 6. Определите, какие процедуры балансировки нужно применить при добавлении каждого узла.

2. Рассмотрите дерево, сформированное на следующих входных данных: 5, 3, 8, 2, 4, 1, 7, 10, 6, 9, 11. Рассмотрите, как будут удаляться из дерева следующие элементы (последовательно): 4, 8, 6, 5, 2, 1, 7. Определите, какие процедуры балансировки придется применять.

3. Найти минимально и максимально возможное количество вершин в АВЛ-дереве высоты h.

4. Доказать все три утверждения о восстановлении сбалансированности деревьев.

5. Проверьте, является ли заданное двоичное дерево сбалансированным.

6. В файловой системе каталог файлов организован в виде сбалансированного дерева. Каждый узел обозначает файл, содержащий имя и атрибуты файла, в том числе дату последнего обращения к файлу. Написать программу, которая обходит дерево и удаляет те файлы, последнее обращение к

которым происходило до определенной даты. Сбалансированность дерева должна сохраняться.

7. Рассмотрите тип узла дерева, в который добавлено поле parent типа P_Node, указывающее на родителя этого узла. Перепишите все процедуры, рассмотренные в рамках данного занятия, для деревьев с таким типом узлов.

8. Двоичное дерево называется красно-черным, если оно обладает следующими свойствами (RB-свойствами):

а) Каждая вершина – либо красная, либо черная

б) Каждый лист (**nil**) – черный

в) Если вершина красная, оба ее потомка – черные

г) Все пути, идущие вниз от корня к листьям, содержат одинаковое количество черных вершин. Число таких вершин называют черной высотой дерева.

Разработайте программы вставки и удаления элементов в такое дерево.

Указание. Вместо поля bal используйте поле color. В остальном тип узла дерева должен воспроизводить тип из задачи 4.

9. Пусть дан файл, содержащий целые числа. Сформируйте из них дерево поиска и выведите элементы дерева в порядке убывания значения.

10. Проверьте, является ли данное двоичное дерево деревом поиска.

11. Пусть дан текстовый файл. Слова содержат не более 20 символов. Определите частоту использования каждого слова в тексте. Результат оформить в виде таблицы, содержащей слова в лексикографическом порядке.

УКАЗАНИЯ К ВЫПОЛНЕНИЮ РАБОТ

1. Каждое задание необходимо выполнять в соответствии с изученными в рамках темы алгоритмами.

2. Помимо приведенного теоретического материала, также рекомендуется воспользоваться материалами лекции.

3. Этапы решения должны сопровождаться комментариями в коде.

4. Следует реализовывать каждое задание в соответствии со следующими этапами:

- изучить словесную постановку задачи, выделив при этом все виды данных;
- сформулировать математическую постановку задачи;
- выбрать метод решения задач;
- разработать графическую схему алгоритма;
- записать разработанный алгоритм на языке программирования;
- разработать контрольный тест к программе;
- отладить программу.

СПИСОК ИСПОЛЬЗУЕМЫХ ИСТОЧНИКОВ

1. Кнут Д. Искусство программирования для ЭВМ [Текст] / Д. Кнут. - М.: Мир, 1995.

2. Вирт Н. Алгоритмы и структуры данных [Текст] / Н. Вирт. - М.: Мир, 1989.

3. Королев Л.Н.Информатика. Введение в компьютерные науки [Текст] / Л.Н. Королев, А.И. Миков. - М.: Высшая школа, 2003.

4. Бакнелл Дж. Фундаментальные алгоритмы и структуры данных в Delphi [Текст] / Дж. Бакнелл. - СПб.:Питер, 2006.

5. Алгоритмы. Построение и анализ [Текст] / [Т. Кормен и др.]. - М: Изд. Дом «Вильямс», 2007.

Методические материалы

Семенова Ирина Владимировна

ДИНАМИЧЕСКИЕ СТРУКТУРЫ ДАННЫХ. РЕКУРСИЯ

Методические указания