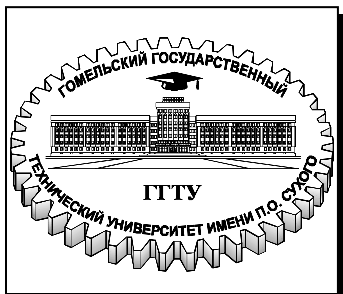


Министерство образования Республики Беларусь

Учреждение образования
«Гомельский государственный технический университет имени П. О. Сухого»

Кафедра «Информационные технологии»

О. А. Кравченко

СТРУКТУРЫ ДАННЫХ В ЯЗЫКЕ СИ

ПОСОБИЕ
по курсам «Модели и структуры данных» и «Основы алгоритмизации и программирования» для студентов специальностей 1-40 01 02 «Информационные системы и технологии (по направлениям)» и 1-36 04 02 «Промышленная электроника» дневной и заочной форм обучения

Гомель 2010

Библиотека ГГТУ им. П.О. Сухого

УДК 004.43(075.8)
ББК 32.973.26-018.2я73
К78

Рекомендовано научно-методическим советом
факультета автоматизированных и информационных систем ГГТУ им. П. О. Сухого
(*протокол № 7 от 15.03.2010 г.*)

Рецензент: ст. преподаватель каф. «Технология машиностроения» *В. С. Мурашко*

Кравченко, О. А.

К78 Структуры данных в языке СИ : пособие по курсам «Модели и структуры данных» и «Основы алгоритмизации и программирования» для студентов специальностей 1-40 01 02 «Информационные системы и технологии (по направлениям)» и 1-36 04 02 «Промышленная электроника» днев. и заоч. форм обучения / О. А. Кравченко. – Гомель : ГГТУ им. П. О. Сухого, 2010. – 149 с. – Систем. требования: PC не ниже Intel Celeron 300 МГц ; 32 Mb RAM ; свободное место на HDD 16 Mb ; Windows 98 и выше ; Adobe Acrobat Reader. – Режим доступа: http://lib.gstu.local. – Загл. с титул. экрана.

Рассматрены основные понятия структур данных, вопросы определения, машинного представления и реализации операций над такими структурами данных как очереди, стеки, списки, графы, деревья, а также хеширование и хеш-таблицы.

Для студентов специальностей 1-40 01 02 «Информационные системы и технологии (по направлениям)» и 1-36 04 02 «Промышленная электроника» дневной и заочной форм обучения.

.

УДК 004.43(075.8)
ББК 32.973.26-018.2я73

© Учреждение образования «Гомельский государственный технический университет имени П. О. Сухого», 2010

Библиотека ГГТУ им. П.О.Сухого

Содержание

1. Структуры данных
1.1. Введение
1.2. Определение очереди
1.3. Машинное представление очереди FIFO и реализация операций
1.4. Реализация операций над очередями
1.4.1. Реализация операций над не циклической очередью
1.4.2. Реализация операций над циклической очередью
1.5. Очереди с приоритетами
1.6. Очереди в вычислительных системах
1.7. Пример решения задачи с использованием очереди
2. Стеки
2.1. Логическая структура стека
2.2. Машинное представление стека и реализация операций
2.2.1. Основные понятия
2.3. Стеки в вычислительных системах
3. Динамические структуры данных.Связные списки
3.1. Связное представление данных в памяти
3.2. Связные линейные списки
3.2.1. Понятие и машинное представление связных линейных списков
3.2.2. Реализация операций над односвязным линейным списком
3.2.3. Применение линейных списков
3.3. Реализация операций над двухсвязным линейным списком
3.4. Мультисписки
5. Нелинейные разветвленные списки
5.1. Основные понятия
5.2. Представление списковых структур в памяти
5.3. Операции обработки списков
6. Хеширование и хеш-таблицы
6.1. Понятие хеширования
6.2. Функции хеширования
6.2.1. Основные понятия
6.2.2. Хеширование для чисел с плавающей точкой, заведомо относящихся к фиксированному диапазону
6.2.3. Модульное хеширование
6.2.4. Хеш-функции для длинных алфавитно-цифровых строк
7.1. Графы
7.1.1. Логическая структура, определения
7.1.2. Машинное представление орграфов

Библиотека ГГТУ им.П.О.Сухого

7.1.3. Алгоритм Флойда
7.1.4. Алгоритм Дейкстры
7.1.5. Максимальный поток
8. Деревья
8.1. Основные определения
8.2. Логическое представление и изображение деревьев
8.3. Бинарные деревья
8.4. Представление любого дерева, леса бинарными деревьями
8.5. Машинное представление деревьев в памяти ЭВМ
8.6. Основные операции над деревьями
8.7. Примеры задач "на дерево"

Библиотека ГГТУ им.П.О.Сухого

1. Структуры данных

1.5. Введение

Под ***СТРУКТУРОЙ ДАННЫХ*** в общем случае понимают множество элементов данных и множество связей между ними. Такое определение охватывает все возможные подходы к структуризации данных, но в каждой конкретной задаче используются те или иные его аспекты.

Структура данных представляет собой набор некоторым образом сгруппированных данных. Простейшим примером структуры данных служит массив. Элементами массива могут быть целые, вещественные числа, символы, строки, записи.
Для каждой структуры данных необходимо определить, каким образом эта структура храниться в памяти ЭВМ и какие операции можно выполнить над этими данными.
Так, массив реализуется в памяти последовательностью своих элементов. Эти элементы располагаются в памяти последовательно друг за другом и являются равнодоступными. В массиве легко осуществить поиск элементов, однако труднее производить удаление элемента и добавление нового элемента. Кроме того, часто при решении задач требуется учитывать порядок обработки элементов. Для этого применяются такие структуры данных, как ***очереди, стеки***, ***списки***, ***кучи***.

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

Рассмотрение структуры данных без учета ее представления в машинной памяти называется абстрактной или ЛОГИЧЕСКОЙ структурой.

Весьма важный признак структуры данных - ее изменчивость - изменение числа элементов и (или) связей между элементами структуры. В определении изменчивости структуры не отражен факт изменения значений элементов данных, поскольку в этом случае все структуры данных имели бы свойство изменчивости. По признаку

Библиотека БГТУ им. П.О. Сухого

изменчивости различают структуры СТАТИЧЕСКИЕ, ПОЛУСТАТИЧЕСКИЕ, ДИНАМИЧЕСКИЕ.

Структуры, являющиеся совокупностями элементов данных - это массив, дерево, запись. Более сложный тип данных может включать эти структуры в качестве элементов. Например, элементами записи может быть массив, стек, дерево и т.д.

Существует большое разнообразие сложных типов данных, но исследования, проведенные на большом практическом материале, показали, что среди них можно выделить несколько наиболее общих. Обобщенные структуры называют также ***моделями данных***, т.к. они отражают представление пользователя о данных реального мира.

Любая модель данных должна содержать три компонента:

структура данных - описывает точку зрения пользователя на представление данных;

набор допустимых операций, выполняемых на структуре данных. Модель данных предполагает, как минимум, наличие языка определения данных (ЯОД), описывающего структуру их хранения, и языка манипулирования данными (ЯМД), включающего операции извлечения и модификации данных.

ограничения целостности - механизм поддержания соответствия данных предметной области на основе формально описанных правил.

Библиотека ГГТУ им. П.О.Сухого

Рис. 1.1. Классификация структур данных

Библиотека ГГТУ им. П. О. Сухого

Определение очереди

Название структуры данных Очередь подразумевает, обработка (удаление) ее элементов производится в зависимости от порядка их поступления(добавления). Это значит, что из двух добавленных элементов раньше будет удален тот, который был раньше добавлен. Понятно, что с помощью такой структуры данных можно моделировать многие жизненные ситуации. Те самые очереди к прилавкам и к кассам, которые мы так не любим, являются типичным бытовым примером очереди. Модель очереди приведена на рис. 1.

Рис.1.2. модель очереди

Очередью называется такая структура данных, для которой определены операции добавления и удаления элементов и элементы которой организованы таким образом, что их удаление будет осуществляться точно в таком же порядке, в каком происходило их добавление.

Та сторона очереди, с которой осуществляется включение элементов, называется ***концом*** или ***хвостом*** очереди. Та сторона очереди, с которой осуществляется исключение элементов, называется ***началом*** или ***головой*** очереди.

Очередь, в которой нет ни одного элемента, называется ***пустой.***

Структуру данных ***очередь*** часто называют структурой FIFO (First - In - First- Out - "первым пришел - первым ушел".

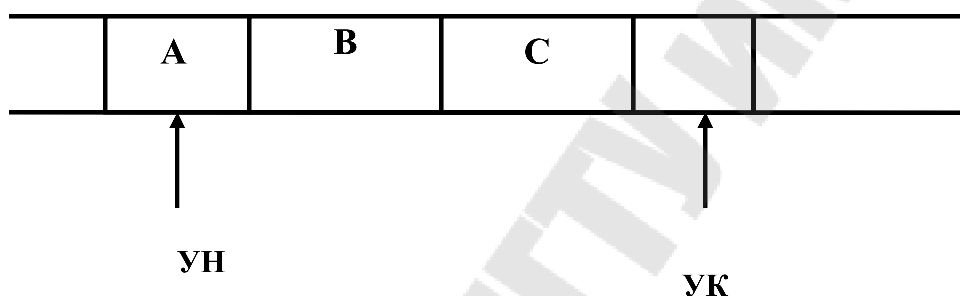
Основные операции над очередью - включение, исключение, определение размера, очистка, неразрушающее чтение.

Библиотека ГГТУ им.П.О.Сухого

1.6. Машинное представление очереди FIFO и реализация операций

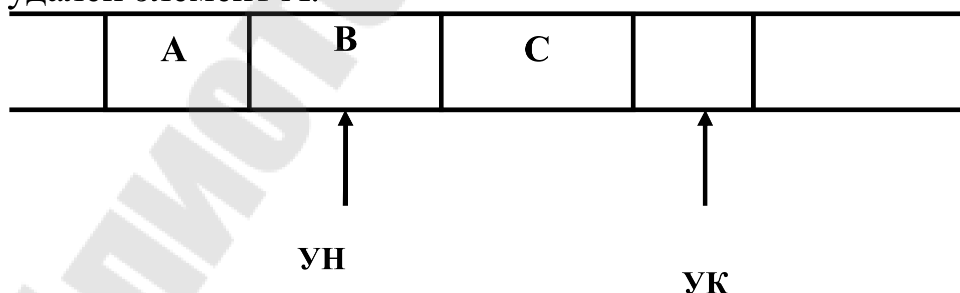
Структура данных ***очередь*** программным путем может быть реализована разными способами. Простейшей является реализация с помощью массива и двух указателей: на начало очереди (на первый элемент в очереди) и на ее конец (первый свободный элемент в очереди). При включении элемента в очередь элемент записывается по адресу, определяемому указателем на конец, после чего этот указатель увеличивается на единицу. При исключении элемента из очереди выбирается элемент, адресуемый указателем на начало, после чего этот указатель уменьшается на единицу.

Например, пусть у нас есть очередь из трех элементов А, В, С, в которую первым был помещен элемент А, затем - элемент В, последним - элемент С. Тогда очередь будет иметь вид:



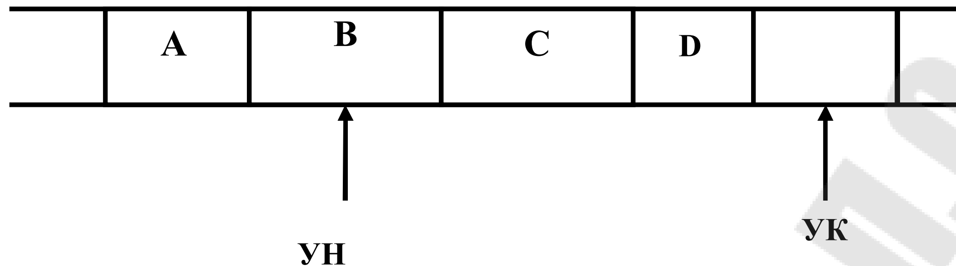
Здесь указатель конца (УК) очереди определяет то место массива, куда новый элемент будет добавляться, а указатель начала (УН) очереди определяет то место массива, откуда очередной элемент будет удаляться.

Удалим из очереди один элемент. Так как элементы очереди могут удаляться только в том порядке, в котором они добавлялись, то будет удален элемент А.



После удаления элемента А первым элементом очереди будет элемент В, а последним – элемент С. При этом элемент А остался в массиве на

своем месте, но его уже нет в очереди, т.к. указатель начала очереди УН указывает на элемент B, т.е. следующий элемент массива.
Добавим в очередь новый элемент D. Он будет помещен в конец очереди, определяемый указателем УК. При этом значение указателя УК должно измениться. После выполнения операции добавления получится следующая конструкция:



Чтобы удалить из очереди элемент C, необходимо сначала удалить элемент B, т.к. он был помещен в очередь раньше, чем элемент C (первым пришел, первым ушел).
Очевидно, что со временем указатель на конец при очередном включении элемента достигнет верхней границы той области памяти, которая выделена для очереди. Однако, если операции включения чередовались с операциями исключения элементов, то в начальной части отведенной под очередь памяти имеется свободное место. Для того чтобы места, занимаемые исключенными элементами, могли быть повторно использованы, очередь замыкается в кольцо: указатели (на начало и на конец), достигнув конца выделенной области памяти, переключаются на ее начало. Такая организация очереди в памяти называется ***кольцевой очередью***.
Возможны, конечно, и другие варианты организации: например, всякий раз, когда указатель конца достигнет верхней границы памяти, сдвигать все непустые элементы очереди к началу области памяти, но как этот, так и другие варианты требуют перемещения в памяти элементов очереди и менее эффективны, чем кольцевая очередь.
В исходном состоянии указатели на начало и на конец указывают на начало области памяти. ***Равенство этих двух указателей (при любом их значении) является признаком пустой очереди***.
Если в процессе работы с кольцевой очередью число операций включения превышает число операций исключения, то может возникнуть ситуация, в которой указатель конца "догонит" указатель начала. Это ситуация заполненной очереди, но если в этой ситуации указатели сравняются, эта ситуация будет неотличима от ситуации

Библиотека ГГТУ им. П.О. Сухого

пустой очереди. Для различения этих двух ситуаций к кольцевой очереди предъявляется требование, чтобы между указателем конца и указателем начала оставался "зазор" из свободных элементов. Когда этот "зазор" сокращается до одного элемента, очередь считается заполненной и дальнейшие попытки записи в нее блокируются. ***Очистка очереди*** сводится к записи одного и того же (не обязательно начального) значения в оба указателя.
Определение размера состоит в вычислении разности указателей с учетом кольцевой природы очереди.

1.7. Реализация операций над очередями

1.4.1. Реализация операций над не циклической очередью

Для работы с очередью определим четыре элементарные операции:

Init - создание пустой очереди;
Empty - возвращение значения 1, если очередь пуста, и 0, если не пуста;
Insert - добавление элемента в очередь;
Remove – удаление элемента из очереди.

Покажем, как можно записать операции работы с очередями на языке СИ.

Прежде всего должен быть описан массив, моделирующий очередь. Пусть это будет массив q, содержащий n элементов.

Создадим функцию **Init**, моделирующую пустую очередь. Параметрами функции Init являются параметры-результаты First и Free - указатели очереди. В примере это номера элементов массива, моделирующего очередь.

```
void Init(int *First,  int *Free)  //создание пустой очереди
  {
      *First=0;
      *Free=0;
  }
```


Библиотека ГГТУ им.П.О.Сухого

Создадим функцию **Empty**, проверяющую, пуста очередь или нет. Очередь пуста, если указатели очереди совпадают, и не пуста в противном случае.

```
int Empty (int First, int Free)   //пуста ли очередь
 {
   int p;
   if(First==Free) p=1;
              else  p=0;
   return (p);
  }
```

Создадим функцию **InsQue** добавления элемента **x** в очередь (реализация операции Insert). Операция добавления может выполняться всегда, если не существует ограничений на количество элементов очереди. Если же для реализации очереди используется массив из n элементов, то в очереди не может быть помещено более n элементов.

Поэтому, прежде чем вставить элемент в очередь, необходимо проверить, есть ли в массиве свободное место для размещения нового элемента очереди. Если места достаточно, то новый элемент помещается в массив. При этом будет формироваться код операции, для чего используется переменная p, значение которой будет равно 0, если операция добавления прошла успешно. Если места для нового элемента в массиве нет, то значение переменной p будет равно 1, что означает, что операция добавления элемента не выполнена из-за отсутствия места в очереди. Параметрами функции **InsQue** является массив q, моделирующий очередь, указатель на начало очереди Free и добавляемый элемент x. Первые два параметра являются входными и выходными, а третий параметр является входным параметром.

```
//добавление элемента в очередь

int InsQue(float q[],int n, int *Free,float x)
{
 int p;
 if(*Free>n-1) p=1; //очередь полна
   else
     {
```


Библиотека ГГТУ им.П.О.Сухого

```
        q[*Free]=x;
       *Free=*Free+1;
        p=0;
       }
  printf("YK=%d\n",*Free);
  return(p);
}
```

Реализуем операцию Remove (удаление элемента из очереди) в виде функции **RemQue** с параметрами q, Free, x, где x – имя переменной, которой будет присвоено значение элемента, удаленного из очереди (выходной параметр). Сначала необходимо убедиться в том, что очередь не пуста, лишь потом можно удалять элемент из очереди. Переменная p получает значение 2, если операция удаления элемента из очереди не выполнена вследствие отсутствия элементов в очереди, и 0 в противном случае.

```
//удаление элемента из очереди

int RemQue(float q[],int *First, int *Free,float *x)//удаление элемента из очередь
{
  int p;
  if(Empty(*First,*Free)) p=2; //очередь пуста
    else
      {
       *x=q[*First];
       *First=*First+1;
       p=0;
      }
  return(p);
}

)//вывод элементов очереди

int PrnQue(float q[],int First,int Free)
{
  int p,i;
```


Библиотека ГГТУ им. П.О. Сухого

```
            if(Empty(First,Free)){        p=2;puts("Очередь        пуста");}
//очередь пуста
            else
             {
              puts("Очередь");
              for(i=First;i<Free;i++)
                printf("%f ",q[i]);
              printf("\n");
            p=0;
             }
           return(p);
        }
```

Программа работы с очередью может выглядеть следующим образом:

```
/*Не циклическая очередь  */
#include<stdio.h>
#include<conio.h>

void Init(int *First, int *Free)//создание пустой очереди
{
 *First=0;
 *Free=0;
}

int Empty (int First, int Free) //пуста ли очередь
{
 int p;
 if(First==Free) p=1;
          else  p=0;
 return (p);
}
int InsQue(float q[],int n, int *Free,float x)
//добавление элемента в очередь
{
  int p;
  if(*Free>n-1) p=1; //очередь полна
     else
       {
```


Библиотека ГГТУ им.П.О.Сухого

```
        q[*Free]=x;
       *Free=*Free+1;
        p=0;
       }
  printf("YK=%d\n",*Free);
  return(p);
}

int RemQue(float q[],int *First, int *Free,float *x)
//удаление элемента из очередь
{
 int p;
 if(Empty(*First,*Free)) p=2; //очередь пуста
    else
       {
        *x=q[*First];
        *First=*First+1;
        p=0;
       }
  return(p);
}

int PrnQue(float q[], int First, int Free)
//вывод элементов очереди
{
  int p, i;

  if(Empty(First,Free))
     { p=2; //очередь пуста
       puts("Очередь пуста");
    }
   else
    {
     puts("Очередь");
     for(i=First;i<Free;i++)
       printf("%f ",q[i]);
     printf("\n");
    p=0;
    }
```


Библиотека ГГТУ им.П.О.Сухого

```
  return(p);
}



main()
{  int nomer, First, Free, n=20;
   float x, q[20];
   Init(&First, &Free);//создание пустой очереди
   clrscr();
   puts("Очередь пока пуста");
   printf("YH=%d YK=%d\n", First, Free);
    do
      {  puts("1- Добавление в очередь");
         puts("2- Удаление из очереди");
         puts("3- Вывод элементов очереди");
         puts("4- Выход");
         scanf("%d", &nomer);
         switch (nomer)
          { case 1:
             {
              puts("Введите элемент");
              scanf("%f",&x);
              if(InsQue(q,n, &Free,x))
               printf("Элемент не добавлен.Очередь
полна.\nYK=%d\n",Free) ;
              }
                 break;
              case 2:
                 if(RemQue(q, &First, &Free,&x)==2)
                 printf("Удаление не произошло, т.к. очередь
пуста\nYH=%d  YK=%d\n",First,Free);//удаление элемента из
очереди
                 else printf("Удаленный элемент есть
%f\n",q[First-1]);
              break;
              case 3:
              PrnQue(q,First,Free);//вывод элементов очереди;
              break;
```


Библиотека ГГТУ им. П.О.Сухого

```
                case 4: break;
                default: nomer=4;
                        break;
              }
        }
    while (nomer!=4);
    return(0);
}
```

Библиотека ГГТУ им.П.О.Сухого

1.4.2. Реализация операций над циклической очередью

Одним из недостатков очереди, реализованной в п. 1.4.1, является тот факт, что при многократном выполнении поочередно операций добавления и удаления в очереди реально будет находиться небольшое число элементов, в то время как указатель конца очереди превысит значение n.

Чтобы избежать такой ситуации, будем рассматривать кольцевую очередь. Для этого надо просто предположить, что после элемента с индексом n-1 следует элемент с индексом 0. В этом случае значение УН может быть больше значения УК. Равенство указателей соответствует не только пустой очереди, но и полностью заполненной. Поэтому необходимо знать, было ли добавление элемента или удаление. Пусть переменная OP принимает значение 1, если последней выполнялась операция добавления элемента в очередь, 0, если выполнялась операция удаления, 2, если не выполнялась операция удаления и не выполнялась операция добавления, т.е.:

$$\text{O}P = \begin{cases} 0 - \text{последней выполнялась операция удаления из очереди;} \\ 1 - \text{последней выполнялась операция добавления;} \\ 2 - \text{не выполнялось удаление и не выполнялось добавление.} \end{cases}$$

Кроме того, будет формироваться переменная p, значение которой будет равно 0, если в очереди есть место, 1, если очередь полна, и 2, если очередь пуста.

$$\text{P} = \begin{cases} 0 - \text{ в очереди есть место;} \\ 1 - \text{очередь полна;} \\ 2 - \text{очередь пуста.} \end{cases}$$

Программа работы с циклической очередью может выглядеть следующим образом:

```
/*  Циклическая очередь  */
#include<stdio.h>
```


Библиотека ГГТУ им. П.О. Сухого

```
#include<conio.h>

int Init(int *First, int *Free)
//создание пустой очереди
{
 *First=0;
 *Free=0;
 return(2);
}

int Empty (int First, int Free)
//пуста ли очередь
{
 int p;
 if(First==Free) p=1;
           else  p=0;
 return (p);
}

int InsQue(float q[],int n, int First,int *Free,int *op,float x)
//добавление элемента в циклическую очередь
{
  int p;
  if(*op==1 && First==*Free) p=1; //очередь полна
     else    //в очереди есть место
         {*op=1; // выполняется добавление
          q[*Free]=x;
         *Free=*Free+1;
         if(*Free>n-1) *Free=0;
         if(*Free==First) p=1; else p=0;
         }
   //printf("YK=%d\n",*Free);
   return(p);
}

int RemQue(float q[],int n,int *First, int *Free,int *op,float *x)//y
{
  int p;
  if((*op==0 && *Free==*First)||*op==2) p=2; //очередь пуста
```

Библиотека ГГТУ им.П.О.Сухого

```
    else
        { *op=0;// удаляется элемент из очереди
         *x=q[*First];
         *First=*First+1;
         if(*First>n-1) *First=0;
         if(*First==*Free) p=2; else p=0;
        }
   return(p);
}

void PrnQue(float q[], int n, int First, int Free, int p)
//вывод элементов цикл. очереди
{
   int i;
   if( p==2)
      puts("Очередь пуста");
    else
     {
      puts("Очередь");
      if (First<Free)    //YH<YK
       {
         for(i=First;i<Free;i++)
           printf("%.0f ",q[i]);
         printf("\n");
       }
       else
         {
          for(i=First;i<n-1;i++)//от начала до конца массива
           printf("%f ",q[i]);
         printf("\n");
         for(i=0;i<Free;i++) //от начала массива до конца очереди
           printf("%f ",q[i]);
         printf("\n");
        }
      }
}
```


Библиотека ГГТУ им.П.О.Сухого

```
main()
{   int nomer, First, Free, n=10, op=2, p, p1;
   float x, q[10];
   p=Init(&First, &Free);//создание пустой очереди
   clrscr();
   puts("Очередь пока пуста");
   printf("YH=%d YK=%d p=%d\n", First, Free, p);
     do
        {  puts(" 1 - Добавление в очередь");
           puts(" 2 - Удаление из очереди");
           puts(" 3 - Вывод элементов очереди");
           puts(" 4 - Выход");
           scanf("%d", &nomer);
           switch (nomer)
            { case 1:
                {
                 puts("Введите элемент");
                 scanf("%f", &x);
                 p=InsQue(q, n, First, &Free, &op, x);
                 printf("YH=%d YK=%d p=%d\n", First, Free, p);
                 if(p==1)
                  printf("Элемент не добавлен. Очередь
полна.\nYK=%d\n", Free);
                 else if(Free==0) printf("Добавленный элемент есть
%.2f\n",q[n-1]);
                      else printf("Добавленный элемент есть %.2f\n
",q[Free-1]);
                 }
                     break;
                 case 2: //Удаление элемента из очереди
                   {p1=p;
                    p=RemQue(q, n, &First, &Free, &op, &x);
                    printf("YH=%d YK=%d p=%d\n", First, Free, p);
                    if( p1==2)
                    printf("Удаление не произошло, т.к. очередь
пуста\nYH=%d YK=%d\n", First, Free);
                    else printf("Удаленный элемент есть %f\n", q[First-
1]);
                   }
```


Библиотека ГГТУ им. П.О.Сухого

```
            break;

            case 3: //вывод элементов очереди;
            {PrnQue(q, n, First, Free, p);
            printf("YH=%d YK=%d p=%d\n", First, Free, p);
            }
            break;
            case 4: break;
            default: nomer=4;
                   break;
           }
       }
   while (nomer!=4);
   return(0);
}
```

<u>Результаты выполнения программы приводятся ниже</u>

1.5. Очереди с приоритетами

В реальных задачах иногда возникает необходимость в формировании очередей, отличных от FIFO или LIFO (First In, Last Out – первым пришел, последним ушел). Порядок выборки элементов из таких очередей определяется приоритетами элементов. Приоритет в общем случае может быть представлен числовым значением, которое вычисляется либо на основании значений каких-либо полей элемента, либо на основании внешних факторов. Так, и FIFO, и LIFO-очереди могут трактоваться как приоритетные очереди, в которых приоритет элемента зависит от времени его включения в очередь. При выборке элемента всякий раз выбирается элемент с наибольшим приоритетом.

Очереди с приоритетами могут быть реализованы на линейных списковых структурах - в смежном или связном представлении. Возможны очереди с приоритетным включением, в которых последовательность элементов очереди все время поддерживается упорядоченной, т.е. каждый новый элемент включается на то место в последовательности, которое определяется его приоритетом, а при исключении всегда выбирается элемент из начала. Возможны и очереди с приоритетным исключением - новый элемент включается всегда в конец очереди, а при исключении в очереди ищется (этот

Библиотека ГГТУ им.П.О.Сухого

поиск может быть только линейным) элемент с максимальным приоритетом и после выборки удаляется из последовательности. И в том, и в другом варианте требуется поиск, а если очередь размещается в статической памяти - еще и перемещение элементов.

Наиболее удобной формой для организации больших очередей с приоритетами является сортировка элементов по убыванию приоритетов частично упорядоченным деревом.

1.6. Очереди в вычислительных системах

Идеальным примером кольцевой очереди в вычислительной системы является буфер клавиатуры в Базовой Системе Ввода-Вывода ПЭВМ IBM PC. Буфер клавиатуры занимает последовательность байтов памяти по адресам от 40:1E до 40:2D включительно. По адресам 40:1A и 40:1C располагаются указатели на начало и конец очереди соответственно. При нажатии на любую клавишу генерируется прерывание 9. Обработчик этого прерывания читает код нажатой клавиши и помещает его в буфер клавиатуры - в конец очереди. Коды нажатых клавиш могут накапливаться в буфере клавиатуры, прежде чем они будут прочитаны программой. Программа при вводе данные с клавиатуры обращается к прерыванию 16H. Обработчик этого прерывания выбирает код клавиши из буфера - из начала очереди - и передает в программу.

Очередь является одним из ключевых понятий в многозадачных операционных системах (Windows NT, Unix, OS/2, ЕС и др.). Ресурсы вычислительной системы (процессор, оперативная память, внешние устройства и т.п.) используются всеми задачами, одновременно выполняемыми в среде такой операционной системы. Поскольку многие виды ресурсов не допускают реально одновременного использования разными задачами, такие ресурсы предоставляются задачам поочередно. Таким образом, задачи, претендующие на использование того или иного ресурса, выстраиваются в очередь к этому ресурсу. Эти очереди обычно приоритетные, однако, довольно часто применяются и FIFO-очереди, так как это единственная логическая организация очереди, которая гарантированно не допускает постоянного вытеснения задачи более приоритетными. LIFO-очереди обычно используются операционными системами для учета свободных ресурсов.

Библиотека ГГТУ им. П.О.Сухого

Также в современных операционных системах одним из средств взаимодействия между параллельно выполняемыми задачами являются очереди сообщений, называемые также почтовыми ящиками. Каждая задача имеет свою очередь - почтовый ящик, и все сообщения, отправляемые ей от других задач, попадают в эту очередь. Задача-владелец очереди выбирает из нее сообщения, причем может управлять порядком выборки - FIFO, LIFO или по приоритету.

1.7. Пример решения задачи с использованием очереди

Условие задачи.

За какое минимальное число ходов можно конем перейти из одной заданной клетки шахматной доски в другую заданную клетку?

```
#include<stdio.h>
#include<conio.h>
main()              //За какое минимальное число ходов конем
{          //можно перейти из одной клетки шахматной доски в другую клетку
 int a[65][4];       //очередь (x коня, y коня и число ходов)
 int mark[9][9];//куда конь ходил (0 - не ходил, 1 - ходил)- матрица посещений
 int x,y,xn,yn,xk,yk, //положения коня на доске (текущее, начальное, конечное)
     h,             //число ходов
     i, j,           //указатели на начало и конец очереди
     f,             //флаг - решение найдено (1) или нет (0)
     t, r;          // параметры цикла
 puts("Введите исходное положение коня");
 scanf("%d%d", &xn, &yn);
 puts("Введите конечное положение коня");
 scanf("%d%d", &xk, &yk);
 printf("  xn=%d yn=%d\n",xn, yn);
 i=1; j=1; h=0; f=0;
 a[i][1]=xn;   a[i][2]=yn;  a[i][3]=h;     //начальное положение коня - в очередь
 printf("  xn=%d yn=%d %d\n", a[i][1], a[i][2], a[i][3]);
 printf("  xk=%d yk=%d\n",xk,yk);
```


Библиотека ГГТУ им. П.О. Сухого

```
//Обнуление матрицы посещений клеток шахматной доски
for(t=1; t<=8; t++)
  for(r=1; r<=8; r++)
    mark[r][t]=0;
//Пока очередь не пуста и решение не найдено, ходим конем
while(i<=j && f==0)
 {
  x=a[i][1];  // извлечение из очереди
  y=a[i][2];
  h=a[i][3];
  printf("  %d %d %d %d %d %d %d %d\n",x,y,h,i,j,a[i][1],a[i][2],a[i][3]);
  //****
  i++;      //сдвигается указатель начала очереди
  mark[x][y]=1; //Заполняется элемент матрицы посещений

  // из текущей клетки можно пойти 8-ью ходами
  if((x+1)<=8 && (y+2)<=8 && mark[x+1][y+2]==0)
    { //если в эту клетку можно пойти - ход в очередь
     j=j+1;
     a[j][1]=x+1;  a[j][2]=y+2; a[j][3]=h+1;
     mark[x+1][y+2]=1;
    }
  if(x+2<=8 && y+1<=8 && mark [x+2][y+1]==0)
    { //если в эту клетку можно пойти - ход в очередь
     j=j+1;
     a[j][1]=x+2;  a[j][2]=y+1; a[j][3]=h+1;
     mark[x+21][y+1]=1;
    }
   if(x+2<=8 && y-1>=1 && mark [x+2][y-1]==0)
    { //если в эту клетку можно пойти - ход в очередь
     j=j+1;
     a[j][1]=x+2;  a[j][2]=y-1; a[j][3]=h+1;
     mark[x+21][y-1]=1;
    }
   if(x+1<=8 && y-2>=1 && mark [x+1][y-2]==0)
    { //если в эту клетку можно пойти - ход в очередь
     j=j+1;
     a[j][1]=x+1;  a[j][2]=y-2; a[j][3]=h+1;
     mark[x+1][y-2]=1;
```

```
    }
   if(x-1>=1 && y+2<=8 && mark [x-1][y+2]==0)
    { //если в эту клетку можно пойти - ход в очередь
     j=j+1;
     a[j][1]=x-1;  a[j][2]=y+2; a[j][3]=h+1;
     mark[x-1][y+2]=1;
    }
   if(x-2>=1 && y+1<=8 && mark [x-2][y+1]==0)
    { //если в эту клетку можно пойти - ход в очередь
     j=j+1;
     a[j][1]=x-1;  a[j][2]=y+1; a[j][3]=h+1;
     mark[x-2][y+2]=1;
    }
   if(x-2>=1 && y-1>=1 && mark [x-2][y-1]==0)
    { //если в эту клетку можно пойти - ход в очередь
     j=j+1;
     a[j][1]=x-2;  a[j][2]=y-1; a[j][3]=h+1;
     mark[x-2][y-1]=1;
    }
   if(x-1>=1 && y-2>=1 && mark [x-1][y-2]==0)
    { //если в эту клетку можно пойти - ход в очередь
     j=j+1;
     a[j][1]=x-1;  a[j][2]=y-2; a[j][3]=h+1;
     mark[x-1][y-2]=1;
    }
  if(x==xk && y==yk) f=1;//Лучше переставить в начало цикла вместо
ком. ****
  }
 if(f==0) puts("   NO");
  // else
  printf(" h=%d f=%d i=%d j=%d\n",h,f,i,j);
  getch();
 return(0);
}
```

2. Стеки

2.1. Логическая структура стека

Стек - такая структура данных, которая является последовательным списком переменной длины, включение и исключение элементов из которого выполняются только с одной стороны последовательности, называемой ***вершиной*** стека. Другой конец стека называется ***основанием*** стека. Применяются и другие названия стека - магазин и очередь, функционирующая по принципу LIFO (Last - In - First- Out - "последним пришел - первым исключается"). Примеры стека: винтовочный патронный магазин, тупиковый железнодорожный разъезд для сортировки вагонов.

Основные операции над стеком:

- включение нового элемента (английское название push - заталкивать);
- исключение элемента из стека (англ. pop - выскакивать).

Полезными могут быть также ***вспомогательные операции***:

- определение текущего числа элементов в стеке;
- очистка стека;
- неразрушающее чтение элемента из вершины стека, которое может быть реализовано, как комбинация основных операций.

x:=pop(stack);   push(stack,x).

Некоторые авторы рассматривают также операции включения/исключения элементов для середины стека, однако структура, для которой возможны такие операции, не соответствует стеку по определению.

Для наглядности рассмотрим небольшой пример, демонстрирующий принцип включения элементов в стек и исключения элементов из стека. На рис. 2.1 (а, б, с) изображены состояния стека:

Библиотека ГГТУ им. П.О.Сухого

- а) пустого;
- б - г) после последовательного включения в него элементов с именами 'A', 'B', 'C';
- д, е) после последовательного удаления из стека элементов 'C' и 'B';
- ж) после включения в стек элемента 'D'.

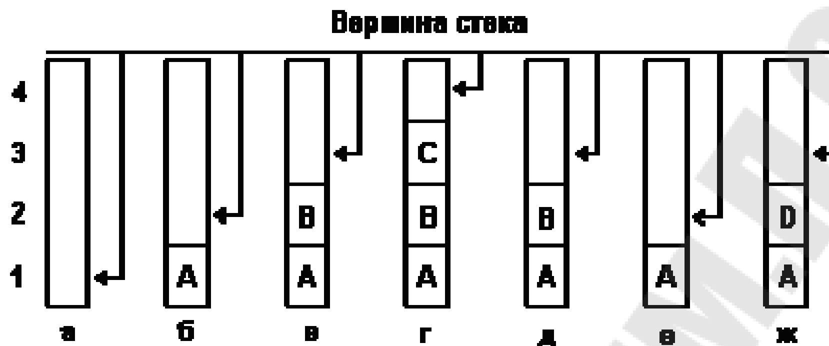


Рис 2.1. Включение и исключение элементов из стека.

Как видно из рис. 2.1, стек можно представить, например, в виде стопки книг (элементов), лежащей на столе. Присвоим каждой книге свое название, например A, B, C, D... Тогда в момент времени, когда на столе книг нет, про стек аналогично можно сказать, что он пуст, т.е. не содержит ни одного элемента. Если же мы начнем последовательно класть книги одну на другую, то получим стопку книг (допустим, из n книг), или получим стек, в котором содержится n элементов, причем вершиной его будет являться элемент n+1. Удаление элементов из стека осуществляется аналогичным образом, т. е. удаляется последовательно по одному элементу, начиная с вершины, или по одной книге из стопки.

2.2. Машинное представление стека и реализация операций

2.2.1. Основные понятия

При представлении стека в статической памяти для стека выделяется память, как для вектора или одномерного массива. Простейшей является реализация стека с помощью массива и указателя стека - адреса вершины стека. Указатель стека может указывать либо на первый свободный элемент стека, либо на последний записанный в стек элемент. (Все равно, какой из этих двух вариантов выбрать,

Библиотека ГГТУ им.П.О.Сухого

важно впоследствии строго придерживаться его при обработке стека.) В дальнейшем мы будем всегда считать, что указатель стека адресует первый свободный элемент и стек растет в сторону увеличения адресов.

При занесении элемента в стек элемент записывается на место, определяемое указателем стека, затем указатель модифицируется таким образом, чтобы он указывал на следующий свободный элемент (если указатель указывает на последний записанный элемент, то сначала модифицируется указатель, а затем производится запись элемента). Модификация указателя состоит в прибавлении к нему или в вычитании из него единицы (помните, что наш стек растет в сторону увеличения адресов).

Операция исключения элемента состоит в модификации указателя стека в направлении, обратном модификации при включении и выборке значения, на которое указывает указатель стека. После выборки слот, в котором размещался выбранный элемент, считается свободным.

Операция очистки стека сводится к записи в указатель стека начального значения - адреса начала выделенной области памяти.

Определение размера стека сводится к вычислению разности указателей: указателя стека и адреса начала области.

2.2.2. Реализация операций над стеком

Программный модуль, представленный в примере, иллюстрирует операции над стеком, расширяющимся в сторону увеличения адресов. Указатель стека указывает на последний элемент.

```
//  x – добавляемый или исключаемый элемент стека

//  top – указатель стека – адрес вершины стека

//  a – массив, реализующий стек

//  n – максимальное число элементов стека


        #include<stdio.h>
```


Библиотека ГГТУ им. П.О.Сухого

```
#include<conio.h>
int a[20],  n=20;


/*Процедура добавления элемента в стек*/

void increment (int *top, int x)
{
  if (*top>n)
       puts("В стеке нет места");
  else
   {
     *top=*top+1;
     a[*top]=x;
 }
}




/*Процедура удаления элемента из стека*/

int decrement (int *top)
{
   int x;
   if (*top<1)
       puts("Стек пуст");
   else
    {
      x=a[*top];
     *top=*top-1;
    }
  return(x);
}


*Процедура вывода элементов стека*/
```


Библиотека ГГТУ им.П.О.Сухого

```
vivod (int *top)
{
    int i;
    puts("Stek");
    for (i=1; i<=*top; i++)
        printf("%-4d",a[i]) ;
   printf("\n");
   return(0);
}

/*Основная программа*/
main()
{
   int element, nomer, top=0;
  do
   {
     puts(" 1- dobavlenie v stek");
     puts(" 2- udalenie iz steka");
     puts(" 3- vivod elementov");
     puts("  4- vixod");
     scanf("%d",&nomer);

     switch (nomer)
     { case 1:
          {
            puts("VVedite element");
            scanf("%d",&element);
            increment(&top,element);
          } break;
        case 2:
              element=decrement(&top);
             break;
        case 3:
             vivod (&top);
         case 4: break;
         default:
                 nomer=4; break;
      }
```


Библиотека ГГТУ им.П.О.Сухого

```
  }
 while (nomer!=4);
}
```

Можно вызвать программу на выполнение (stek)

Примеры задачи, использующей стек:

Пример 1. Определить, верно ли расставлены скобки в выражении.

Примеры: (5-3)-(7+(2)) –да
(5-3)) – нет
)5-6(- нет

```
#include<stdio.h>
#include<string.h>
char a[80],   b[80];

//  a – стек;
//  b – строка, содержащая выражение со скобками




// Добавление элемента в стек

void increment (int *top, int x)
{
 *top=*top+1;
 a[*top]=x;
}
// Удаление элемента из стека

void decrement (int *top)
{

 if (*top>=1)
```


Библиотека ГГТУ им.П.О.Сухого

```
      *top=*top-1;
   }

   main()
   {
      int i,           // номер символа строки
          nb,          // длина строки
          top=0,       //  номер символа '( ' в стеке
          flag=0;      //  не найдена непарная скобка
      gets(b);
      nb= strlen(b);
      i=0;
      while (i< nb && flag==0)
       {
         if ( b[i]=='(' )  increment(&top, b[i]);
         if  (b[i]==')'  && a[top]=='('  )  decrement(&top);
         else
            if (b[i]==')' && top<1) flag=1;
         i++;
       }
     if(flag==0 && top<1) puts("  Выражение верно");
     else
         puts("   Выражение не верно");
    printf("   flag= %d    top =%d\n", flag, top);
     return(0);
   }
```

<u>**Можно вызвать программу на выполнение**</u> **(stek2)**

Пример 2. Вычисление значения арифметического выражения (использование обратной польской записи)

Одной из главных причин, лежащих в основе появления языков программирования высокого уровня, явились вычислительные задачи, требующие больших объёмов рутинных вычислений. Поэтому к языкам программирования предъявлялись требования максимального приближения формы записи вычислений к естественному языку математики. В этой связи одной из первых областей системного программирования сформировалось исследование способов трансляции выражений. Здесь получены многочисленные результаты, однако наибольшее распространение получил метод

Библиотека ГГТУ им.П.О.Сухого

трансляции с помощью ***обратной польской записи***, которую предложил польский математик Я. Лукашевич.

Наиболее простая система записи математических выражений для электронной обработки - обратная польская нотация или обратная польская запись. Например, выражение

(a + b)*(c - d)/sin(e)

в этой системе записывается как

a b + c d - * e sin /

Т.е., если имеется выражение в инфиксной форме, то его можно легко преобразовать в постфиксную форму: знак операции стоит после двух своих аргументов, а не между ними (не A+B, а AB+)

В этой системе не нужны скобки и все выражения могут быть вычислены с помощью простого алгоритма, использующего стек.

Алгоритм вычисления значения выражения, представленного в виде ОПЗ, состоит в следующем.

Когда при проходе строки встречается переменная, она помещается в стек, а когда встречается знак математической операции, из стека извлекается нужное количество переменных, осуществляется операция и результат снова помещается в стек. Когда вся строка пройдена, остается лишь извлечь из стека результат. Если формула была записана правильно, то это будет единственный элемент стека.

Перевод традиционной формы записи математических выражений в ОПН - немного более сложная операция. Она также осуществляется с использованием стека, но теперь это стек математических операций.

Приоритет математических операций таков:

- Функции (напр. sin, cos) и скобки
- Умножение и деление
- Сложение и вычитание

Алгоритм перевода, предложенный Дейкстрой таков:

1. Когда в исходной строке встречается переменная, она записывается прямо в конечную строку.
2. Когда встречается символ математической операции, из стека извлекаются все операции с высшим приоритетом, затем операция помещается в стек.

Библиотека ГГТУ им. П.О. Сухого

3. Когда встречается открывающая скобка, она помещается в стек.
4. Когда встречается закрывающая скобка, из стека извлекается всё до открывающей скобки. Открывающая скобка может быть извлечена только закрывающей, сами скобки в конечную строку не записываются.
5. Когда вся строка уже пройдена, из стека извлекается всё, что там осталось.

Пример.

Дана строка, содержащая символы: латинские буквы, скобки и знаки арифметических операций (* / + -), например, a+b*(c-d)/e. Требуется получить обратную польскую запись выражения. Для рассмотренного примера это будет строка: abcd-* e/+.

```
/*	Получение обратной польской записи  заданного арифметического выражения*/

#include<stdlib.h>
#include<math.h>
#include<stdio.h>
#include<conio.h>

/*Процедура добавления элемента в стек*/
void increment(int *top, int x)
{
 if (*top>n)
  puts("V steke net mesta");
 else
 {
  *top=*top+1;
  st[*top]=x;
 }

}

/*Процедура извлечения элемента из стека*/
char del (int *top)
{char x;
```


Библиотека ГГТУ им. П.О. Сухого

```
 if (*top<1)
  puts("stek пуст");
 else
 {

  x=st[*top];

  *top=*top-1;
  }
 return(x);
}


/* Функция PRIOR возвращает приоритет арифм. операции */
int PRIOR(char a)
{ int n;
  switch(a)
  {
    case '*':
    case '/':
         { n= 3;         break;}
    case '-':
    case '+':
          {n= 2;         break;}
    case '(':
         { n=1;          break;}
  }
  return(n);
}


/*Основная программа */

main()
{
 char st[80];//Стек операций
 int n=80;   // Размер стека

  clrscr();
```


Библиотека ГГТУ им.П.О.Сухого

```
int top=0; /*Указатель вершины стека
                 (номер элемента массива st).
                 Стек операций пуст. */
char a[80],  outstring[80];   //Исходная и выходная строки
int k,  point;
do
{ puts("  Введите выражение (в конце '='):");
  fflush(stdin);
     /* Ввод арифметического выражения*/
  gets(a); puts(a); getch();
  k=point=0;
      /*Повторяем, пока не дойдем до '=' */
  while(a[k]!='\0'&&a[k]!='=')
  {
     /*Если очередной символ- ')' */
   if(a[k]==')')
              /*то выталкиваем из стека в выходную строку */
   {
            /* все знаки операций до ближайшей */
     while(st[top]!='(')
              /*открывающейся скобки */
     outstring[point++]=del(&top);
              /* удаляем из стека саму открывающуюся скобку
*/
     puts(st);
     del(&top);
   }
                /*Если очередной символ - буква, то */
   if(a[k]>='a'&& a[k]<='z')
            /*переписываем ее в выходную строку */
      outstring[point++]=a[k];
                   /*Если очередной символ - '(' , то */
   if(a[k]=='(')
            /* заталкиваем его в стек */
      increment(&top, '(');
   if(a[k]=='+'||a[k]=='-'||a[k]=='/'||a[k]=='*')
      /* Если следующий символ - знак операции , то: */
   {
             /* если стек пуст */
```


Библиотека ГГТУ им.П.О.Сухого

```
            if(top==0)
                    /* записываем в него операцию */
              {increment(&top, a[k]); puts(st);}
                          /* если стек не пуст */
            else
     /* если приоритет поступившей операции больше
                приоритета операции на вершине стека */
            if(PRIOR(st[top])<PRIOR(a[k]))
            /* заталкиваем поступившую операцию на стек */
              increment(&top, a[k]);
                             /*если приоритет меньше */
            else
            {
             while((top!=0) && (PRIOR(st[top])>=PRIOR(a[k])))
     /* переписываем в выходную строку все операции
                   с большим или равным приоритетом */
               outstring[point++]=del(&top);
                           /* записываем в стек поступившую операцию
*/
             increment(&top, a[k]);
            }
          }
             /*Переход к следующему символу входной строки*/
          k++;
         }
              /*После рассмотрения всего выражения */
         while(top!=0)
             /* переписываем все операции из */
            outstring[point++]=del(&top);
                   /* стека в выходную строку */
         outstring[point]='\0';
              /* и печатаем ее */
         printf("\nПолученная обр. польская запись\n  %s\n", outstring);
         fflush(stdin);
         puts("\nПовторить(y/n)?");
        }
     while(getchar()!='n');
      return(0);
     }
```


Библиотека ГГТУ им. П.О. Сухого

2.3. Стеки в вычислительных системах

Стек является чрезвычайно удобной структурой данных для многих задач вычислительной техники. Наиболее типичной из таких задач является обеспечение вложенных вызовов процедур.

Предположим, имеется процедура A, которая вызывает процедуру B, а та в свою очередь - процедуру C. Когда выполнение процедуры A дойдет до вызова B, процедура A приостанавливается и управление передается на входную точку процедуры B. Когда B доходит до вызова C, приостанавливается B и управление передается на процедуру C. Когда заканчивается выполнение процедуры C, управление должно быть возвращено в B, причем в точку, следующую за вызовом C. При завершении B управление должно возвращаться в A, в точку, следующую за вызовом B. Правильную последовательность возвратов легко обеспечить, если при каждом вызове процедуры записывать адрес возврата в стек. Так, когда процедура A вызывает процедуру B, в стек заносится адрес возврата в A; когда B вызывает C, в стек заносится адрес возврата в B. Когда C заканчивается, адрес возврата выбирается из вершины стека - а это адрес возврата в B. Когда заканчивается B, в вершине стека находится адрес возврата в A, и возврат из B произойдет в A.

В микропроцессорах семейства Intel, как и в большинстве современных процессорных архитектур, поддерживается аппаратный стек. Аппаратный стек расположен в ОЗУ, указатель стека содержится в паре специальных регистров - SS:SP, доступных для программиста. Аппаратный стек расширяется в сторону уменьшения адресов, указатель его адресует первый свободный элемент. Выполнение команд CALL и INT, а также аппаратных прерываний включает в себя запись в аппаратный стек адреса возврата. Выполнение команд RET и IRET включает в себя выборку из аппаратного стека адреса возврата и передачу управления по этому адресу. Пара команд - PUSH и POP - обеспечивает использование аппаратного стека для программного решения других задач.

Системы программирования для блочно-ориентированных языков (PASCAL, C и др.) используют стек для размещения в нем локальных переменных процедур и иных программных блоков. При каждой активизации процедуры память для ее локальных переменных выделяется в стеке; при завершении процедуры эта память

Библиотека ГГТУ им. П.О.Сухого

освобождается. Поскольку при вызовах процедур всегда строго соблюдается вложенность, то в вершине стека всегда находится память, содержащая локальные переменные активной в данный момент процедуры.

Этот прием делает возможной легкую реализацию рекурсивных процедур. Когда процедура вызывает сама себя, то для всех ее локальных переменных выделяется новая память в стеке, и вложенный вызов работает с собственным представлением локальных переменных. Когда вложенный вызов завершается, занимаемая его переменными область памяти в стеке освобождается и актуальным становится представление локальных переменных предыдущего уровня. За счет этого в языках PASCAL и C любые процедуры/функции могут вызывать сами себя. В языке PL/1, где по умолчанию приняты другие способы размещения локальных переменных, рекурсивная процедура должна быть определена с описателем RECURSIVE - только тогда ее локальные переменные будут размещаться в стеке.

Рекурсия использует стек в скрытом от программиста виде, но все рекурсивные процедуры могут быть реализованы и без рекурсии, но с явным использованием стека. В программном примере приведена реализация быстрой сортировки Хоара в рекурсивной процедуре.

```
{==== Программный пример 4.2 ====}
{ Быстрая сортировка Хоара (стек) }
Procedure Sort(a : Seq);   { см. раздел 3.8 }
  type board=record          { границы обрабатываемого участка }
    i0, j0 : integer;   end;
  Var i0, j0, i, j, x : integer;
    flag_j : boolean;
    stack  : array[1..N] of board;  { стек }
    stp    : integer; { указатель стека работает на увеличение }
begin   { в стек заносятся общие границы }
  stp:=1; stack[i].i0:=1; stack[i].j0:=N;
  while stp>0 do                    { выбрать границы из стека }
  begin  i0:=stack[stp].i0; j0:=stack[stp].j0; stp:=stp-1;
   i:=i0; j:=j0; flag_j:=false;{проход перестановок от i0 до j0}
    while ia[j] then                { перестановка }
       begin x:=a[i]; a[i]:=a[j]; a[j]:=x; flag_j:= not flag_j;
     end;
```


Библиотека БГУИР

```
    if flag_j then Dec(j) else Inc(i);
  end;   if i-1>i0 then {занесение в стек границ левого участка}
   begin  stp:=stp+1; stack[stp].i0:=i0; stack[stp].j0:=i-1;
   end; if j0>i+1 then {занесение в стек границ правого участка}
   begin  stp:=stp+1; stack[stp].i0:=i+1; stack[stp].j0:=j0;
   end;  end;
```

Один проход сортировки Хоара разбивает исходное множество на два множества. Границы полученных множеств запоминаются в стеке. Затем из стека выбираются границы, находящиеся в вершине, и обрабатывается множество, определяемое этими границами. В процессе его обработки в стек может быть записана новая пара границ и т.д. При начальных установках в стек заносятся границы исходного множества. Сортировка заканчивается с опустошением стека.

Библиотека ГГТУ им.П.О.Сухого

3. ДИНАМИЧЕСКИЕ СТРУКТУРЫ ДАННЫХ. СВЯЗНЫЕ СПИСКИ

3.1. Связное представление данных в памяти

Динамические структуры по определению характеризуются:

- отсутствием физической смежности элементов структуры в памяти;
- непостоянством и непредсказуемостью размера (числа элементов) структуры в процессе ее обработки.

Поскольку элементы динамической структуры располагаются по непредсказуемым адресам памяти, адрес элемента такой структуры не может быть вычислен из адреса начального или предыдущего элемента. Для установления связи между элементами динамической структуры используются указатели, через которые устанавливаются явные связи между элементами. Такое представление данных в памяти называется связным.

Элемент динамической структуры представляет собой структуру (в смысле struct) , соделжащую по крайней мере два поля:

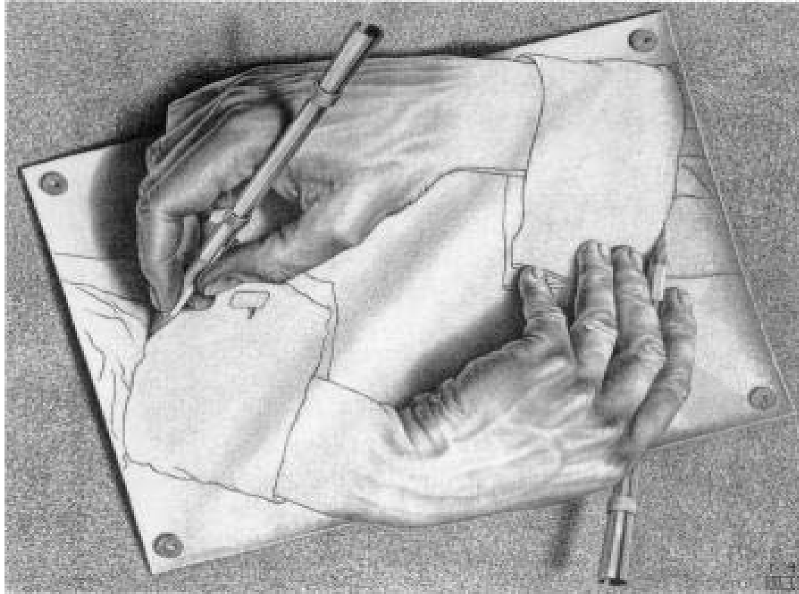
1. информационное поля или поле данных, в котором содержатся те данные, ради которых и создается структура; в общем случае информационное поле само является интегрированной структурой - вектором, массивом, записью и т.п.;
2. поле связок, в котором содержатся один или несколько указателей, связывающий данный элемент с другими элементами структуры.

Пример. Описание простейшего элемента динамической структуры выглядит следующим образом:

```
Struct Zaoch
 {
   Student d; // поле данных (должно быть определено ранее)
   Zaoch *p; //поле связок
 }
```


Библиотека ГГТУ им. П.О. Сухого

Динамические структуры являются рекурсивными т.к. структура содержит в качестве элемента указатель, ссылающийся на структуру того же типа. Одной из лучших иллюстраций рекурсии является литография голландского художника Мориса Эшера “Рисующие руки”.



Когда связное представление данных используется для решения прикладной задачи, для конечного пользователя "видимым" делается только содержимое информационного поля, а поле связок используется только программистом-разработчиком.

Достоинства связного представления данных:

- возможность обеспечения значительной изменчивости структур;
- размер структуры ограничивается только доступным объемом машинной памяти;
- при изменении логической последовательности элементов структуры требуется не перемещение данных в памяти, а только коррекция указателей.

Вместе с тем ***связное представление не лишено и недостатков***, основные из которых:

- работа с указателями требует, как правило, более высокой квалификации от программиста;

Библиотека ГГТУ им. П.О.Сухого

- на поля связок расходуется дополнительная память;
- доступ к элементам связной структуры может быть менее эффективным по времени.

Последний недостаток является наиболее серьезным и именно им ограничивается применимость связного представления данных. Если в смежном представлении данных для вычисления адреса любого элемента нам во всех случаях достаточно было номера элемента и информации, содержащейся в дескрипторе структуры, то для связного представления адрес элемента не может быть вычислен из исходных данных. Дескриптор связной структуры содержит один или несколько указателей, позволяющих войти в структуру, далее поиск требуемого элемента выполняется следованием по цепочке указателей от элемента к элементу. Поэтому связное представление практически никогда не применяется в задачах, где логическая структура данных имеет вид вектора или массива с доступом по номеру элемента, но часто применяется в задачах, где логическая структура требует другой исходной информации доступа (таблицы, списки, деревья и т.д.).

3.2. Связные линейные списки

3.2.1. Понятие и машинное представление связных линейных списков

Списком называется упорядоченное множество, состоящее из переменного числа элементов, к которым применимы операции включения, исключения. Список, отражающий отношения соседства между элементами, называется ***линейным***. Если ограничения на длину списка не допускаются, то список представляется в памяти в виде связной структуры. Линейные связные списки являются простейшими динамическими структурами данных.

Самый простой способ связать множество элементов – сделать так, чтобы каждый элемент содержал ссылку на следующий. Такой список называется ***однонаправленным*** (односвязным). Если добавить в каждый элемент вторую ссылку – на предыдущий элемент, получится ***двунаправленный список*** (двусвязный). Если последний элемент связать указателем с первым, получится ***кольцевой список***.

Библиотека ГГТУ имени П.О.Сухого

Каждый элемент списка содержит ***ключ***, идентифицирующий этот элемент. Ключ обычно бывает либо целым числом, либо строкой и является частью поля данных. В качестве ключа в процессе работы со списком могут выступать разные части поля данных. Например, если создается линейный список из записей, содержащих фамилию, год рождения, стаж работы, то при упорядочивании списка по алфавиту ключом будет фамилия, а при поиске ветеранов труда ключом будет стаж.

Над списками можно выполнять следующие ***операции***:

- начальное формирование списка (создание первого элемента)ж
- добавление элемента в конец списка;
- чтение элемента с заданным ключом;
- вставка элемента в заданное место списка (до или после элемента с заданным ключом);
- удаление элемента с заданным ключом;
- упорядочивание списка по ключу.

Графически связи в списках удобно изображать с помощью стрелок. Если компонента не связана ни с какой другой, то в поле указателя записывают значение, не указывающее ни на какой элемент. Такая ссылка обозначается специальным именем - nil.

На рис. 3.1 приведена структура односвязного списка. На нем поле INF - информационное поле, данные, NEXT - указатель на следующий элемент списка.

Каждый список должен иметь особый элемент, называемый ***указателем начала списка или головой списка,*** который обычно по формату отличен от остальных элементов. В поле указателя последнего элемента списка находится специальный признак nil, свидетельствующий о конце списка.

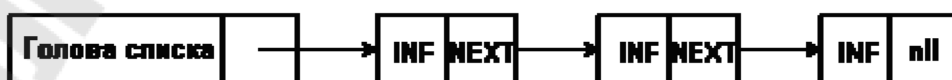


Рис. 3.1. Структура односвязного списка

Библиотека БГУИР

Однако, обработка односвязного списка не всегда удобна, так как отсутствует возможность продвижения в противоположную сторону. Такую возможность обеспечивает двухсвязный список, каждый элемент которого содержит два указателя: на следующий и предыдущий элементы списка.

Структура линейного двухсвязного списка приведена на рис. 3.2, где поле NEXT - указатель на следующий элемент, поле PREV - указатель на предыдущий элемент. В крайних элементах соответствующие указатели должны содержать nil, как и показано на рис. 3.2.

Для удобства обработки списка добавляют еще один особый элемент - ***указатель конца списка***. Наличие двух указателей в каждом элементе усложняет список и приводит к дополнительным затратам памяти, но в то же время обеспечивает более эффективное выполнение некоторых операций над списком.

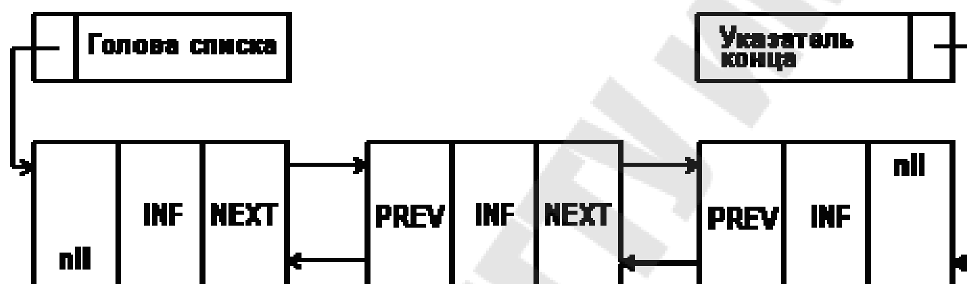


Рис.3.2. Структура двухсвязного списка

Разновидностью рассмотренных видов линейных списков является кольцевой список, который может быть организован на основе как односвязного, так и двухсвязного списков. При этом в односвязном списке указатель последнего элемента должен указывать на первый элемент; в двухсвязном списке в первом и последнем элементах соответствующие указатели переопределяются, как показано на рис.3.3.

При работе с такими списками несколько упрощаются некоторые процедуры, выполняемые над списком. Однако при просмотре такого списка следует принять некоторые меры предосторожности, чтобы не попасть в бесконечный цикл.

Библиотека ГГТУ им. П.О.Сухого

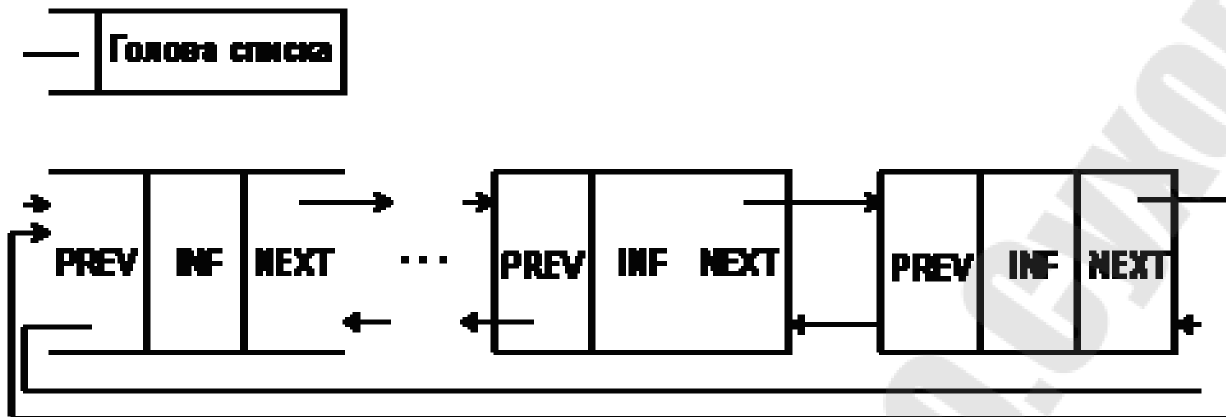


Рис. 3.3. Структура кольцевого двухсвязного списка

В памяти список представляет собой совокупность дескриптора и одинаковых по размеру и формату записей, размещенных произвольно в некоторой области памяти и связанных друг с другом в линейно упорядоченную цепочку с помощью указателей. Запись содержит информационные поля и поля указателей на соседние элементы списка, причем некоторыми полями информационной части могут быть указатели на блоки памяти с дополнительной информацией, относящейся к элементу списка. Дескриптор списка реализуется в виде особой записи и содержит такую информацию о списке, как адрес начала списка, код структуры, имя списка, текущее число элементов в списке, описание элемента и т.д., и т.п. Дескриптор может находиться в той же области памяти, в которой располагаются элементы списка, или для него выделяется какое-нибудь другое место.

3.2.2. Реализация операций над односвязным линейным списком

Ниже рассматриваются некоторые простые операции над линейными списками. Выполнение операций иллюстрируется в общем случае рисунками со схемами изменения связей и программными примерами.

На всех рисунках сплошными линиями показаны связи, имевшиеся до выполнения и сохранившиеся после выполнения операции. Пунктиром показаны связи, установленные при выполнении операции. Значком 'x' отмечены связи, разорванные при выполнении операции. Во всех операциях чрезвычайно важна последовательность коррекции указателей, которая обеспечивает корректное изменение списка, не затрагивающее другие элементы. При неправильном

Библиотека ГГТУ им.П.О.Сухого

порядке коррекции легко потерять часть списка. Поэтому на рисунках рядом с устанавливаемыми связями в скобках показаны шаги, на которых эти связи устанавливаются.

Рассмотрим однонаправленный линейный список. Сначала для простоты допустим, что список состоит из целых чисел. Тогда описание элемента списка будет выглядеть следующим образом:

```
struct Node
  { int d;     //поле данного
    Node *next; // поле связи (указатель на след. элем. списка)
  };
```

• Напишем простую программу, которая создает список из одного элемента и выводит его.

Для формирования списка и работы с ним понадобятся два указателя – на начало списка и на конец списка. Пусть их идентификаторами будут ***pbeg*** и ***pend***.

Напишем функцию **Node *first(int x)** ***создания первого элемента*** списка:

```
Node *first(int x)  // создание 1-го элемента списка
{
 Node *p;
 p=new Node; // место для p в динам. памяти
 p->d=x;
 p->next=NULL; // Нет следующего элемента
 return(p);
}
```

В программе создается шаблон для элемента структуры **Node**,

```
/*Односвязный список  */
#include<stdio.h>
#include<conio.h>
struct Node
  { int d;     //поле данного
    Node *next;  // поле связи
  };
Node *first(int x)  // создание 1-го элемента списка
{
 Node *p;
```


Библиотека ГГТУ им.П.О.Сухого

```
 p=new Node; // место для p в динам. памяти
 p->d=x;
 p->next=NULL;
 return(p);
}
main()
{
 Node *pbeg,       // указатель на начало списка
         *pend,      // указатель на конец списка
        *pv,*p2;    // указатели на элемент списка
 int x,                 // данное элемента списка
      i;                 // номер элемента списка
 clrscr();
 // создание списка из одного элемента
 x=12;
 pbeg=first(x);   //В списке из одного элемента
 pend=pbeg; //указатели на начало и на конец списка совпадают
 pv=pbeg;
i=1;
 printf("x=%d i=%d\n",pv->d,i);
 pv=pv->next;
 getch();

 return(0);
}
```

Запустим программу на выполнение (spis1)

Дополним программу операторами, которые добавят в список еще один элемент и выведут два элемента.

```
/*Односвязный список  */
#include<stdio.h>
#include<conio.h>
struct Node
  { int d;     //поле данного
    Node *next;  // поле связи
  };
Node *first(int x)  // создание 1-го элемента списка
```


Библиотека ГГТУ им.П.О.Сухого

```
{
 Node *p;
 p=new Node; // место для p в динам. памяти
 p->d=x;
 p->next=NULL;
 return(p);
}
main()
{
 Node *pbeg,    // указатель на начало списка
        *pend,      // указатель на конец списка
       *pv,*p2;    // указатель на элемент списка
 int x,i;
 clrscr();
 // создание списка из одного элемента
 x=12;
 pbeg=first(x);
 pend=pbeg;
 //Добавление в список еще одного элемента
 p2->d=24;
 p2->next=NULL;
 pbeg->next=p2;
 // Вывод списка
 pv=pbeg;
 i=1;
 while (pv)
  {
    printf("x=%d i=%d\n", pv->d, i);
    pv = pv->next;
    i++;
  }
 return(0);
 getch();
}
```

Запустим программу на выполнение (stpis2)

• ***Добавления элемента в конец списка***

```
/*Односвязный список  */
#include<stdio.h>
```


Библиотека ГГТУ им. П.О. Сухого

```
#include<conio.h>
struct Node
  { int d;     //поле данного
    Node *next;  // поле связи
  };
Node *first(int x)  // создание 1-го элемента списка
{
 Node *p;
 p=new Node; // место для p в динам. памяти
 p->d=x;
 p->next=NULL;
 return(p);
}
void add( Node **pend,int x)
 {
   Node *p;
   p=new Node;
   p->d=x;
   p->next=NULL;
  *pend=p;
 }
main()
{
 Node *pbeg,  // указатель на начало списка

      *pend,  // указатель на конец списка
      *pv,*p2, *pp;    // указатель на элемент списка
 int x,i;
 clrscr();
 // создание списка из одного элемента
 x=12;
 pbeg=first(x);
 pend=pbeg;
 //Добавление в список еще одного элемента
 p2->d=24;

 p2->next=NULL;
 pbeg->next=p2;
 pend=p2;
```

Библиотека ГГТУ им. П.О. Сухого

```
 add(&pend,45);
 p2->next=pend;
 pp=pend;
 add(&pend,99);
 pp->next=pend;
// Вывод списка
 pv=pbeg;
 i=1;
 while (pv)
  {
   printf(" \n   x=%d i=%d\n",pv->d,i);
   pv=pv->next;
   i++;
   }
 getch();
 return(0);

}
```

Запустим программу на выполнение (spis3)

• ***Поиск элемента по ключу***

/*Односвязный список. Создание, вывод, поиск элемента по ключу*/

```
#include<stdio.h>
#include<conio.h>


struct Node
  { int d;     //поле данного
    Node *next;  // поле связи
  };


Node *first(int x)  // создание 1-го элемента списка
{
```


Библиотека ГГТУ им.П.О.Сухого

```
 Node *p;
 p=new Node; // место для p в динам. памяти
 p->d=x;
 p->next=NULL;
 return(p);
}


// Добавление элемента в конец списка
void add( Node **pend, int x)
 {
  Node *p;
  p=new Node;
  p->d=x;
  p->next=NULL;
 *pend=p;
 }


//Поиск по ключу
Node *find(Node * const pbeg,  int x,  int *f)
 {
  Node *p=pbeg;
  *f=0;
  while(p && (*f==0))
  {
   if(p->d==x)  *f=1;
    else p=p->next;
  }
  return(p);
 }


main()
{
 Node *pbeg,  // указатель на начало списка
     *pend,  // указатель на конец списка
     *pv,*p2, *pp;    // указатель на элемент списка
 int x,i, nomer,f;
```


Библиотека ГГТУ им.П.О.Сухого

```
clrscr();
// создание списка из одного элемента
x=12;
pbeg=first(x);
pend=pbeg;
do
{   puts("\n");
   puts("   1- Добавление в список");
   puts("   2- Поиск элемента");
   puts("   3- Вывод элементов списка");
   puts("   4- Выход");
   scanf("%d",&nomer);

   switch (nomer)
    { case 1:
       {
        pp=pend;
        puts("      Введите элемент");
        scanf("%d",&x);
        add(&pend,x);
        pp->next=pend;
        printf("Добавление: %d %d\n",pbeg->d, pbeg->next);
        printf("%d %d\n",pend->d, pend->next);
       }
        break;
        case 2:
           {
             puts("      Введите элемент");
             scanf("%d",&x);
             pp=find(pbeg,x,&f);
             if(f)
               printf("Элем, рав. %d есть %d   %d\n",x,pp->d,
pp->next );
             else printf("В списке нет %d\n",x);
           }
        break;
        case 3:
        {
         pv=pbeg;
```


Библиотека ГГТУ им.П.О.Сухого

```
            i=1;
            while (pv)
             {
               printf("    x=%d i=%d\n",pv->d,i);
               pv=pv->next;
               i++;
              }
           }
           break;
          case 4: break;
          default: nomer=4;
           break;
       }
   }
    while (nomer!=4);
    getch();
    return(0);
  }
```

Запустим программу на выполнение (spis4)

• ***Вставка элемента в список***

Вставка элемента в середину односвязного списка показана на рис. 3.4 и в примере 3.4.

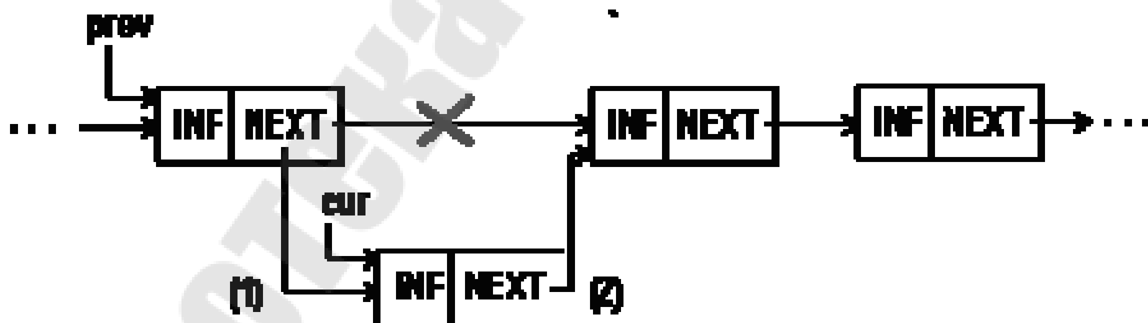


Рис. 3.4. Вставка элемента в середину 1-связного списка

Пример вставки в середину односвязного списка

```
/*Односвязный список. Создание, вывод,
 поиск элемента по ключу, вставка в середину списка*/
```


Библиотека ГГТУ им.П.О.Сухого

```
#include<stdio.h>
#include<conio.h>
struct Node
  { int d;     //поле данного
    Node *next;  // поле связи
  };
Node *first(int x)  // создание 1-го элемента списка
{
 Node *p;
 p=new Node; // место для p в динам. памяти
 p->d=x;
 p->next=NULL;
 return(p);
}
// Добавление элемента в конец списка
void add( Node **pend,int x)
 {
   Node *p;
   p=new Node;
   p->d=x;
   p->next=NULL;
  *pend=p;
 }
//Поиск по ключу
Node *find(Node * const pbeg, int x, int *f)
 {
   Node *p=pbeg;
   *f=0;
   while(p && (*f==0))
   {
    if(p->d==x)  *f=1;
      else p=p->next;
   }
   return(p);
 }
 //Вставка элемента x в середину списка  после заданного (key)
Node *insert(Node * const pbeg, int x, int key, Node **pend)
 {
```

Библиотека ГГТУ им.П.О.Сухого

```
        int f;
        Node *pkey; //Указатель на элемент, после которого надо
вставить
        pkey=find(pbeg, key,&f); //Поиск элем., после которого надо
вставить
        if(pkey)
         {
          Node *p=new Node;  // Выделяется память для нового
элемента списка
          p->d=x;
          if(pkey!=*pend)
             p->next=pkey->next;//Связь нового узла с последующим
           else {p->next=NULL; *pend =p;}
          pkey->next=p;  //Связь предыдущего с новым
          return(p);
         }
        else return(0);
      }
     main()
     {
      Node *pbeg,  // указатель на начало списка
          *pend,  // указатель на конец списка
          *pv,*p2, *pp;    // указатель на элемент списка
      int x, key,i, nomer,f;
      clrscr();
      // создание списка из одного элемента
      x=12;
      pbeg=first(x);
      pend=pbeg;
      do
      {   puts("\n");
         puts("   1- Добавление в список");
         puts("   2- Поиск элемента");
         puts("   3- Вставка элемента в середину списка");
         puts("   4- Вывод элементов списка");
         puts("   5- Выход");
         scanf("%d",&nomer);

         switch (nomer)
```


Библиотека ГГТУ им. П.О.Сухого

```
{ case 1:
   {
    pp=pend;
    puts("      Введите элемент");
    scanf("%d",&x);
    add(&pend,x);
    pp->next=pend;
    printf("Добавление: %d %d\n",pbeg->d, pbeg->next);
    printf("%d %d\n",pend->d, pend->next);
   }
    break;
    case 2:
       {
         puts("      Введите элемент");
         scanf("%d",&x);
         pp=find(pbeg,x,&f);
         if(f)
           printf("Элем, рав. %d есть %d  %d\n",x,pp->d,
pp->next );
         else printf("В списке нет %d\n",x);
       }
    break;
    case 3:
     {      puts("            Введите элемент, который надо
вставить");
       scanf("%d",&x);
       puts("         Введите элемент, после которого надо
вставить");
       scanf("%d",&key);
       pp=insert(pbeg,  x,  key, &pend);
       printf("Вставлен  элемент  %d  %d  \n",pp->d,  pp-
>next );
     }
    break;
    case 4:
    {
     pv=pbeg;
     i=1;
     while (pv)
```


Библиотека ГГТУ им. П.О.Сухого

```
            {
             printf("    x=%d i=%d\n",pv->d,i);
             pv=pv->next;
             i++;
            }
          }
           break;
          case 5: break;
          default: nomer=5;
           break;
      }
  }
   while (nomer!=5);
   getch();
   return(0);
  }
```

Запустим программу на выполнение (spis5)

Приведенный пример обеспечивают вставку в середину списка. Для вставки в начало списка должен модифицироваться указатель на начало списка, как показано на рис.3.5.

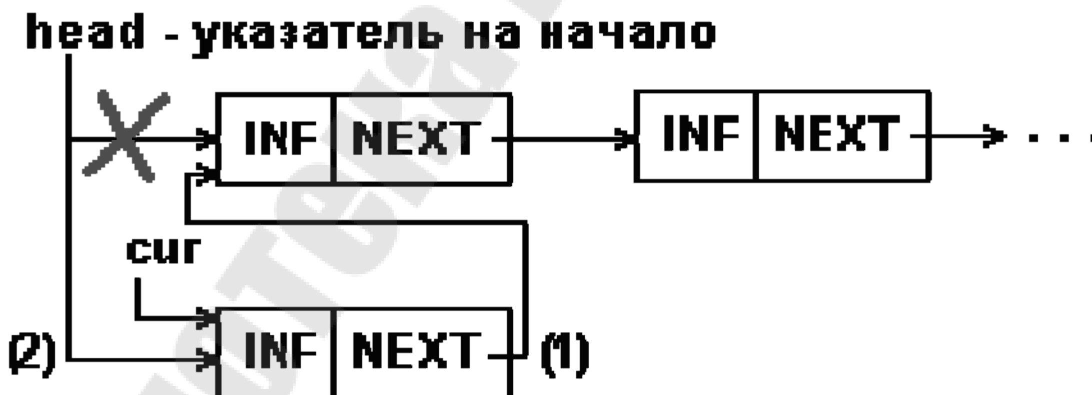


Рис. 3.5. Вставка элемента в начало 1-связного списка

Можно написать функцию, выполняющую вставку элемента в любое место односвязного списка (самостоятельно)

Библиотека ГГТУ им.П.О.Сухого

• ***Удаление элемента из односвязного списка***

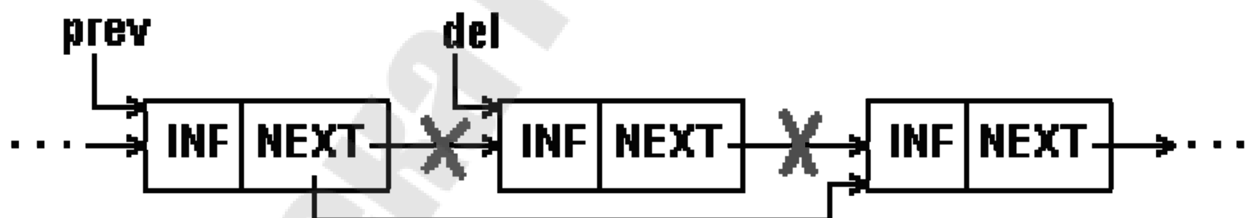
Очевидно, что процедуру удаления легко выполнить, если известен адрес элемента, предшествующего удаляемому элементу(prev на рис.3.6.а). На рис. 3.6 приводится случай, когда удаляемый элемент задается своим адресом. Процедура обеспечивает удаление как из середины, так и из начала списка.

При удалении элемента из начала списка, то указатель на начало списка надо откорректировать так, чтобы он указывал на следующий элемент в списке, адрес которого находится в поле next первого элемента.

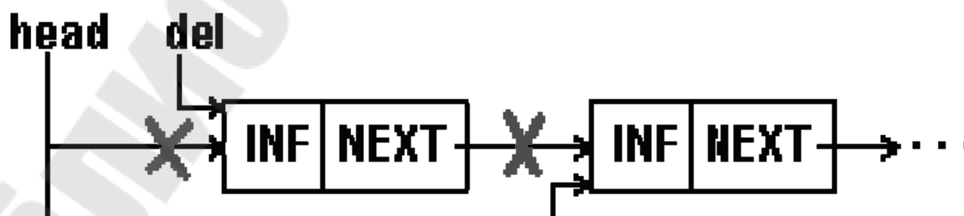
При удалении из середины списка приходится искать элемент, предшествующий удаляемому; поиск производится перебором списка с самого его начала, пока не будет найдет элемент, поле next которого совпадает с адресом удаляемого элемента.

При удалении последнего элемента списка необходимо изменить указатель на последний элемент.

Удаление элемента из односвязного списка показано на рис. 3.6.



а). из середины



б). из начала

Библиотека ГГТУ им. П.О. Сухого

Рис. 3.6. Удаление элемента из 1-связного списка

```
/*Односвязный список.Создание, вывод,
 поиск элемента по ключу,вставка в середину списка,
  удаление элемента*/
#include<stdio.h>
#include<conio.h>
struct Node
  { int d;     //поле данного
    Node *next;  // поле связи
  };
Node *first(int x)  // создание 1-го элемента списка
{
 Node *p;
 p=new Node; // место для p в динам. памяти
 p->d=x;
 p->next=NULL;
 return(p);
}
// Добавление элемента в конец списка
void add( Node **pend,int x)
 {
   Node *p;
   p=new Node;
   p->d=x;
   p->next=NULL;
  *pend=p;
 }
//Поиск по ключу
Node *find(Node * const pbeg, int x, int *f)
 {
   Node *p=pbeg;
   *f=0;
   while(p && (*f==0))
   {
    if(p->d==x)  *f=1;
      else p=p->next;
```


Библиотека ГГТУ им.П.О.Сухого

```
   }
   return(p);
  }
 //Вставка элемента x в середину списка  после заданного (key)
Node *insert(Node * const pbeg, int x, int key, Node **pend)
 {
  int f;
  Node *pkey; //Указатель на элемент, после которого надо вставить
  pkey=find(pbeg, key,&f); //Поиск элем., после которого надо вставить
  if(pkey)
   {
    Node *p=new Node;  // Выделяется память для нового элемента списка
    p->d=x;
    if(pkey!=*pend)
       p->next=pkey->next;//Связь нового узла с последующим
     else {p->next=NULL; *pend =p;}
    pkey->next=p;  //Связь предыдущего с новым
    return(p);
   }
  else return(0);
 }

//Удаление элемента
int remove(Node **pbeg, Node **pend, int key)
{
  int f;
  Node *pkey, *ppred;
  pkey=find(*pbeg, key, &f);
  if(f) //Если key содержится в списке
   {
    if(pkey==*pbeg) // Если удаляемый элемент в начале списка
        *pbeg=(*pbeg)->next;
    else  //Удаляемый элемент в середине списка
        {
```


Библиотека ГГТУ им.П.О.Сухого

```
        ppred=*pbeg;
        while(ppred->next!=pkey)
          ppred=ppred->next;
        ppred->next=pkey->next;
        if(pkey==*pend)
          {*pend=ppred; (*pend)->next=NULL;}
       }
    delete pkey;
    return(1);
  }
 else return(0);
}


main()
{
 Node *pbeg,  // указатель на начало списка
     *pend,  // указатель на конец списка
     *pv,*p2, *pp;    // указатель на элемент списка
 int x, key,i, nomer,f;
 clrscr();
 // создание списка из одного элемента
 x=12;
 pbeg=first(x);
 pend=pbeg;
 do
 {   puts("\n");
    puts("   1- Добавление в список");
    puts("   2- Поиск элемента");
    puts("   3- Вставка элемента в середину списка");
    puts("   4- Вывод элементов списка");
    puts("   5- Удаление  элементав списка");
    puts("   6- Выход");
    scanf("%d",&nomer);

    switch (nomer)
     { case 1:
        {
         pp=pend;
```


Библиотека ГГТУ им.П.О.Сухого

```
        puts("      Введите элемент");
        scanf("%d",&x);
        add(&pend,x);
        pp->next=pend;
        printf("  Добавление: %d %d\n",pbeg->d, pbeg->next);
        printf("  %d %d\n",pend->d, pend->next);
       }
        break;
        case 2:
           {
             puts("      Введите элемент");
             scanf("%d",&x);
             pp=find(pbeg,x,&f);
             if(f)
               printf("  Элем, рав. %d есть %d  %d\n",x,pp->d,
pp->next );
             else printf("  В списке нет %d\n",x);
           }
        break;
        case 3:
         {   puts(" Введите элемент, который надо вставить");
           scanf("%d",&x);
           puts("Введите элемент, после которого надо
вставить");
           scanf("%d",&key);
           pp=insert(pbeg,  x,  key, &pend);
           printf("   Вставлен элемент %d %d \n",pp->d, pp-
>next );
          }
        break;
        case 4:
        {
          pv=pbeg;
          i=1;
          while (pv)
           {
            printf("    x=%d i=%d\n",pv->d,i);
            pv=pv->next;
            i++;
```

```
          }
         }
         break;
         case 5:
          {
           puts("              Введите элемент, который надо
удалить");
           scanf("%d",&x);
           f= remove(&pbeg, &pend, x);
           printf("  %d %d\n", pend->d, pend->next);
           if(f) printf("  Удален элемент %d\n", x);
           else  printf("  Не удален элемент %d, т.к. его нет в
списке\n", x);

          }
         break;
        case 6: break;
        default: nomer=6;
         break;
      }
    }
     while (nomer!=6);
     getch();
     return(0);
    }
```

Запустим программу на выполнение (spis6)

• Перестановка элементов списка

Изменчивость динамических структур данных предполагает не только изменения размера структуры, но и изменения связей между элементами. Для связных структур изменение связей не требует пересылки данных в памяти, а только изменения указателей в элементах связной структуры. В качестве примера приведена перестановка двух соседних элементов списка. В алгоритме перестановки в односвязном списке (рис.5.7) исходили из того, что известен адрес элемента, предшествующего паре, в которой производится перестановка. В приведенном алгоритме также не учитывается случай перестановки первого и второго элементов.

Библиотека ГГТУ им. П.О. Сухого

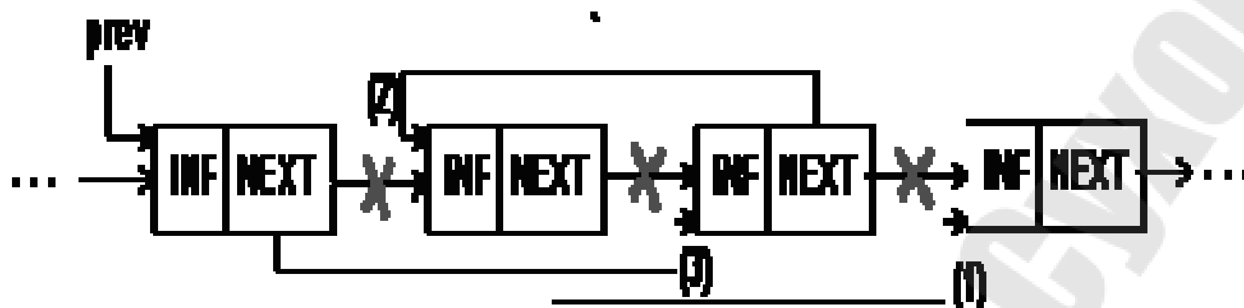


Рис. 3.7. Перестановка соседних элементов 1-связного списка

• ***Копирование части списка***

При копировании исходный список сохраняется в памяти, и создается новый список. Информационные поля элементов нового списка содержат те же данные, что и в элементах старого списка, но поля связок в новом списке совершенно другие, поскольку элементы нового списка расположены по другим адресам в памяти. Существенно, что операция копирования предполагает дублирование данных в памяти. Если после создания копии будут изменены данные в исходном списке, то изменение не будет отражено в копии и наоборот.

• ***Слияние двух списков***

Операция слияния заключается в формировании из двух списков одного - она аналогична операции сцепления строк. В случае односвязного списка слияние выполняется очень просто. Последний элемент первого списка содержит пустой указатель на следующий элемент, этот указатель служит признаком конца списка. Вместо этого пустого указатель в последний элемент первого списка заносится указатель на начало второго списка. Таким образом, второй список становится продолжением первого.

3.2.3. Применение линейных списков

Линейные списки находят широкое применение в приложениях, где непредсказуемы требования на размер памяти, необходимой для хранения данных; большое число сложных операций над данными, особенно включений и исключений. На базе линейных списков могут строиться стеки, очереди и деки. Представление очереди с помощью линейного списка позволяет достаточно просто обеспечить любые

Библиотека ГГТУ им. П.О. Сухого

желаемые дисциплины обслуживания очереди. Особенно это удобно, когда число элементов в очереди трудно предсказуемо.

3.3. Реализация операций над двухсвязным линейным списком

В двухсвязном списке каждый элемент содержит два указателя: на следующий и предыдущий элементы списка.

Структура линейного двухсвязного списка приведена на рис. 3.2, где поле NEXT - указатель на следующий элемент, поле PREV - указатель на предыдущий элемент.

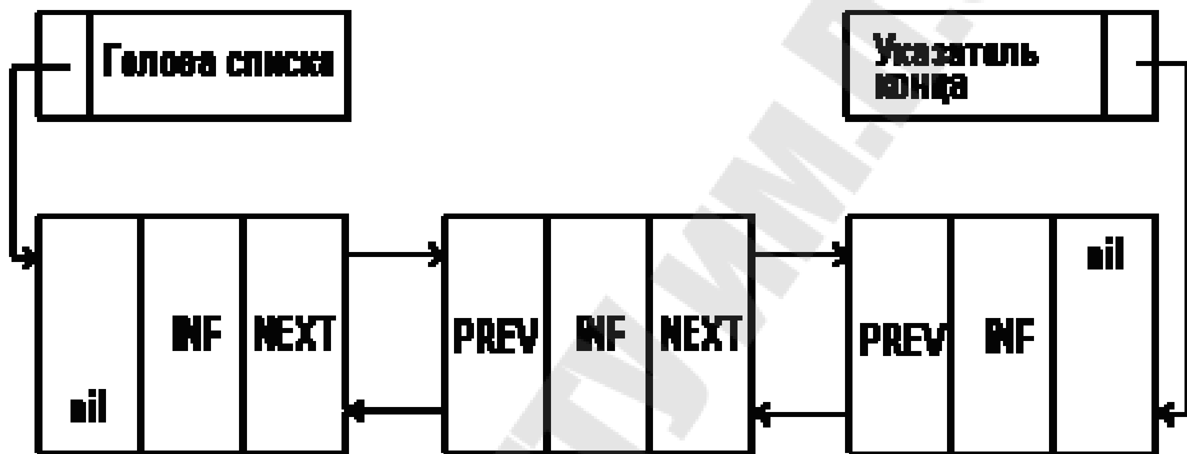


Рис. 3.8. Структура двухсвязного списка

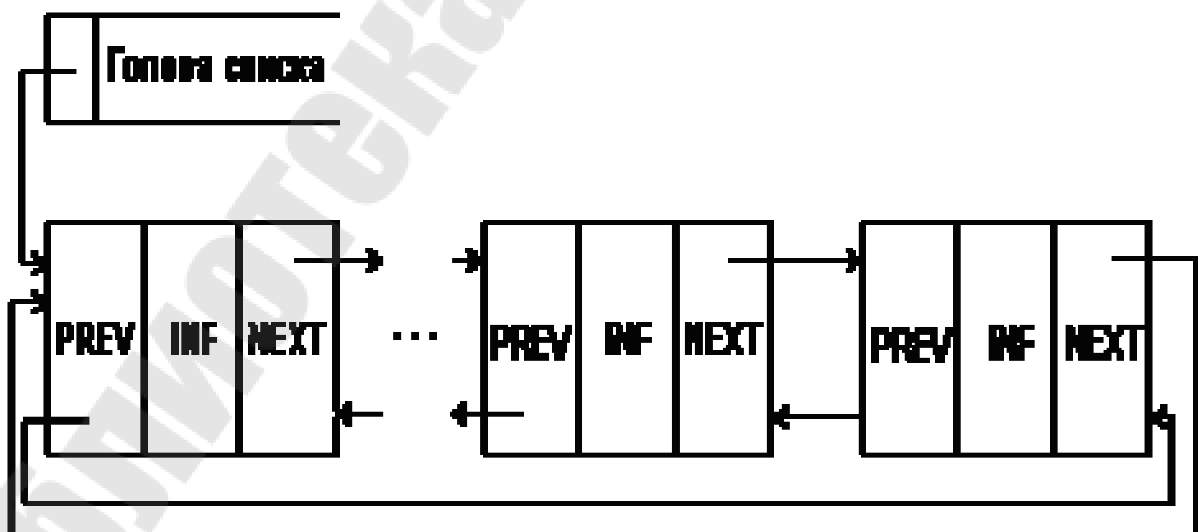


Рис.3.9. Структура кольцевого двухсвязного списка

Библиотека ГГТУ им. П.О.Сухого

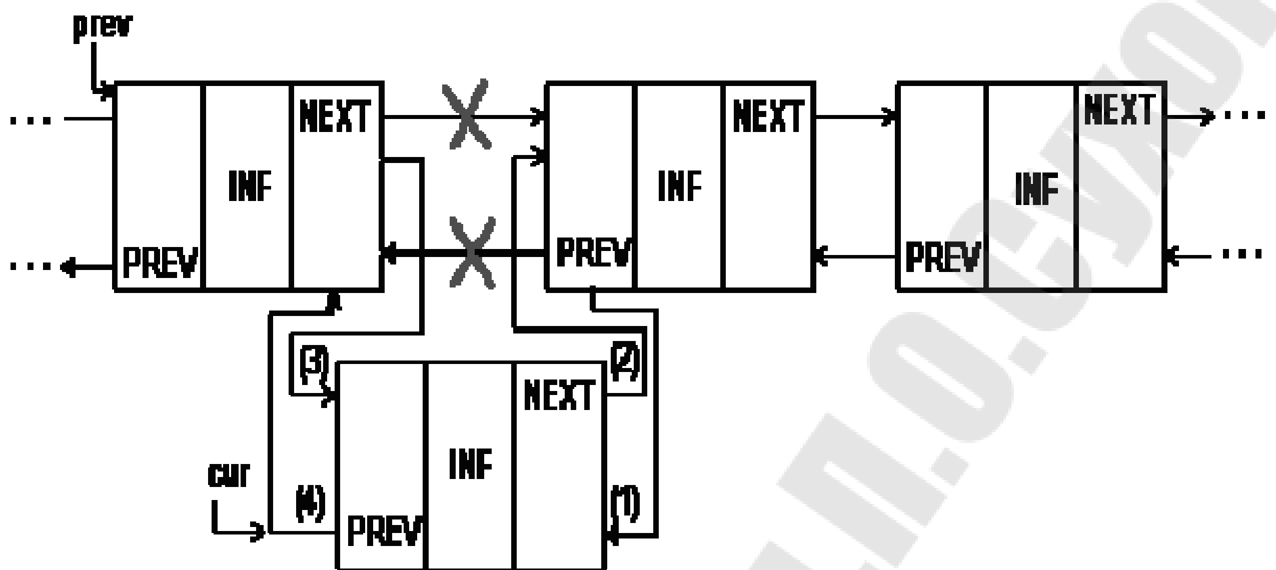


Рис.3.10. Вставка элемента в середину 2-связного списка

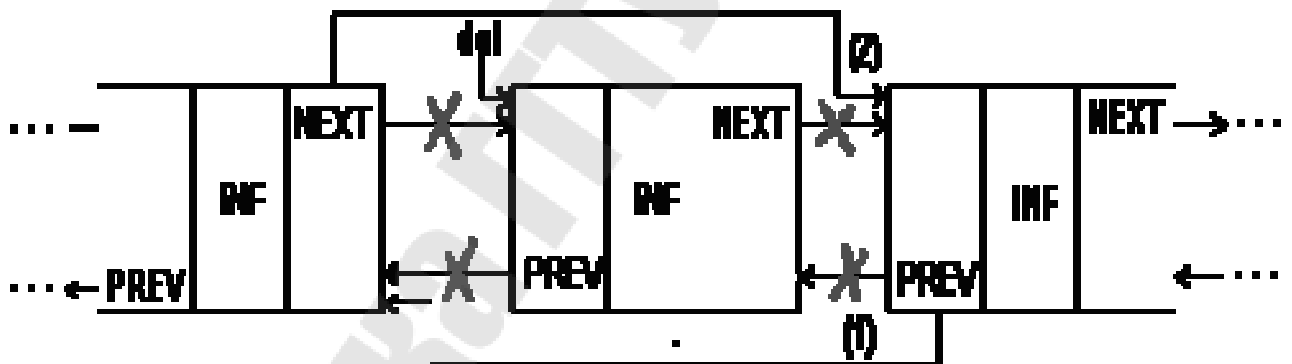


Рис. 3.11. Удаление элемента из 2-связного списка

Библиотека ГГТУ им.П.О.Сухого

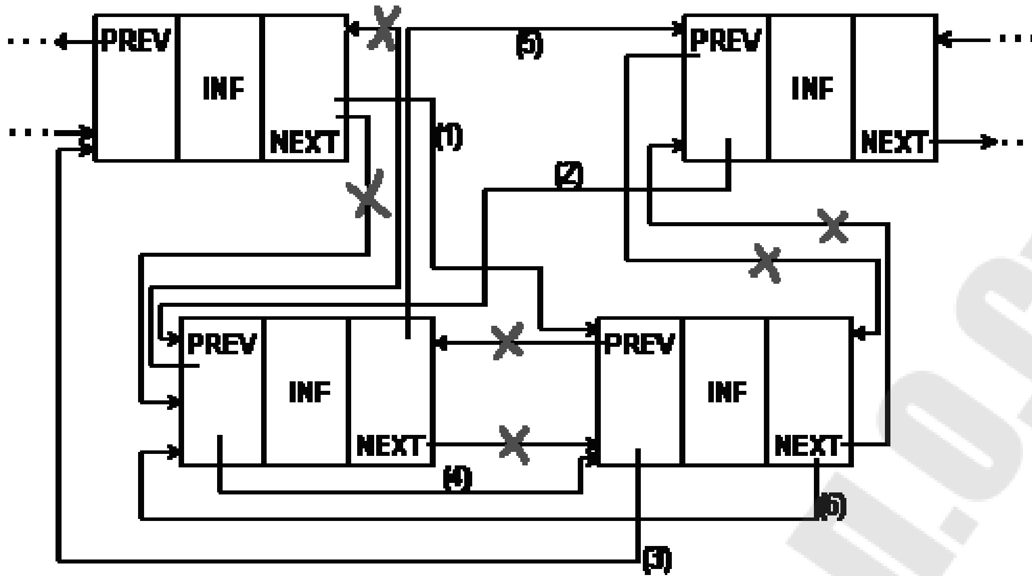


Рис.3.12. Перестановка соседних элементов 2-связного списка

Для простоты допустим, что список состоит из целых чисел. Тогда описание элемента двухсвязного списка выглядит следующим образом:

```
struct Node
  { int d;     //поле данного
    Node *next; // поле связи (указатель на след. элем. списка)
    Node *prev; //поле связи (указатель на пред. элем. списка)
  };
```

Пример функции формирования первого элемента двухсвязного списка:

```
Node *first(int x)  // создание 1-го элемента списка
{
 Node *p;
  p=new Node; // место для p в динам.  памяти
  p->d=x;
  p->next=NULL; // Нет следующего элемента
  p->prev=NULL; // Нет предыдущего элемента
  return(p);
}
```

Функция добавления в конец списка:

Библиотека ГГТУ им. П.О. Сухого

```
// Добавление элемента в конец списка
void add( Node **pend, int x)
 {
   Node *p;
   p=new Node;
   p->d=x;
   p->next=NULL;
   p->prev-*pend;
 (*pend)->next=p;
  *pend=p;
 }
```

Другие функции, реализующие операции над двунаправленными списками, см. С/С++. Программирование на языке высокого уровня / Т.А. Павловская. – СПБ.: Питер, 2002. – 464 с.: ил. (с.115-119).

4 Мультисписки

В программных системах, обрабатывающих объекты сложной структуры, могут решаться разные подзадачи, каждая из которых требует, возможно, обработки не всего множества объектов, а лишь какого-то его подмножества. Так, например, в автоматизированной системе учета лиц, пострадавших вследствие аварии на ЧАЭС, каждая запись об одном пострадавшем содержит более 50 полей в своей информационной части. Решаемые же автоматизированной системой задачи могут потребовать выборки, например:

- участников ликвидации аварии;
- переселенцев из зараженной зоны;
- лиц, состоящих на квартирном учете;
- лиц с заболеваниями щитовидной железы;
- и т.д., и т.п.

Библиотека ГГТУ им. П.О. Сухого

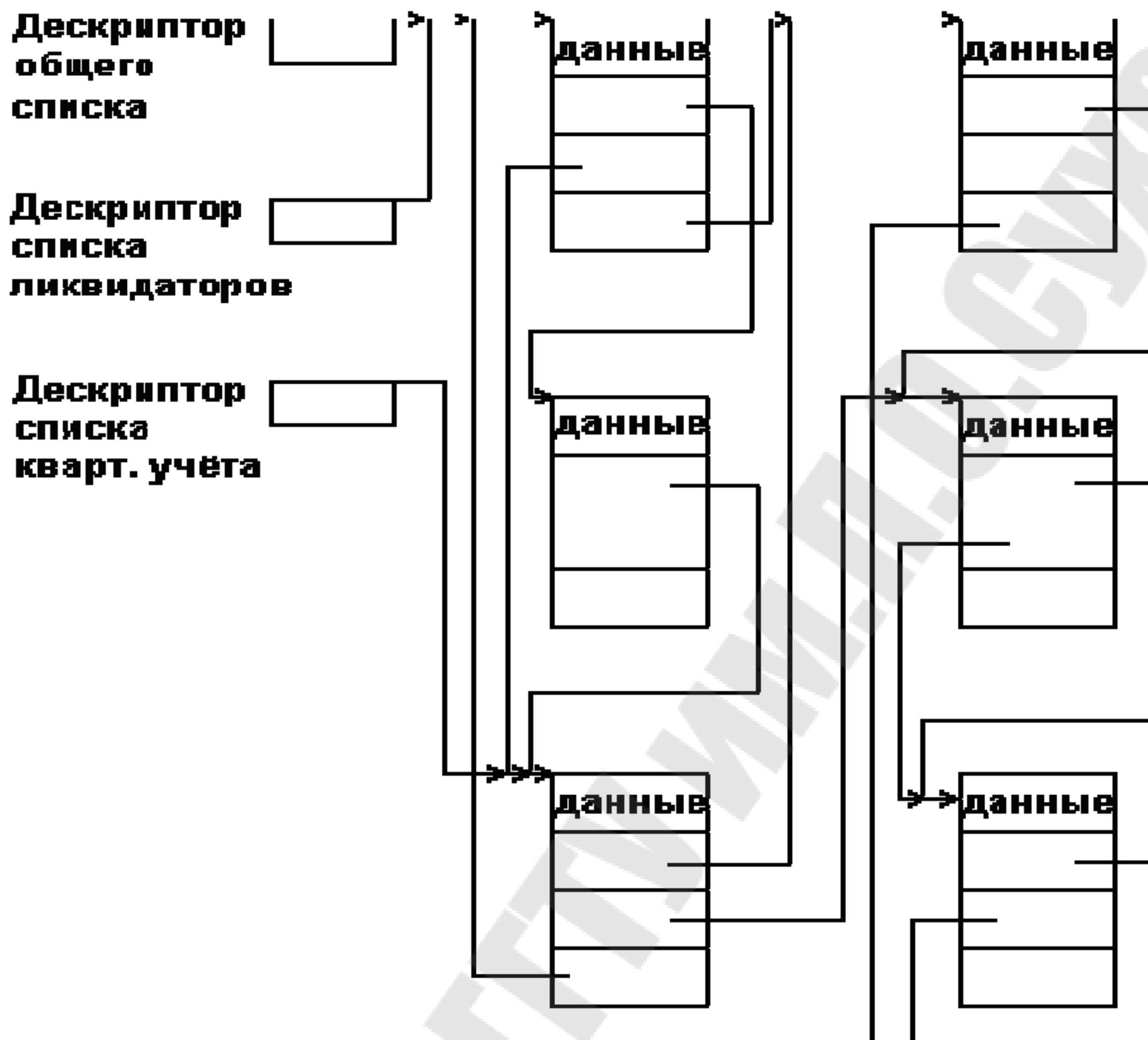


Рис.4.1. Пример мультисписка

Для того, чтобы при выборке каждого подмножества не выполнять полный просмотр с отсеиванием записей, к требуемому подмножеству не относящихся, в каждую запись включаются дополнительные поля ссылок, каждое из которых связывает в линейный список элементы соответствующего подмножества. В результате получается многосвязный список или мультисписок, каждый элемент которого может входить одновременно в несколько односвязных списков. Пример такого мультисписка для названной нами автоматизированной системы показан на рис.5.11.

К достоинствам мультисписков помимо экономии памяти (при множестве списков информационная часть существует в единственном экземпляре) следует отнести также целостность данных

Библиотека ГГТУ им.П.О.Сухого

- в том смысле, что все подзадачи работают с одной и той же версией информационной части и изменения в данных, сделанные одной подзадачей немедленно становятся доступными для другой подзадачи.

Каждая подзадача работает со своим подмножеством как с линейным списком, используя для этого определенное поле связок. Специфика мультисписка проявляется только в операции исключения элемента из списка. Исключение элемента из какого-либо одного списка еще не означает необходимости удаления элемента из памяти, так как элемент может оставаться в составе других списков. Память должна освобождаться только в том случае, когда элемент уже не входит ни в один из частных списков мультисписка. Обычно задача удаления упрощается тем, что один из частных списков является главным - в него обязательно входят все имеющиеся элементы. Тогда исключение элемента из любого неглавного списка состоит только в переопределении указателей, но не в освобождении памяти. Исключение же из главного списка требует не только освобождения памяти, но и переопределения указателей как в главном списке, так и во всех неглавных списках, в которые удаляемый элемент входил.

5. Нелинейные разветвленные списки

5.1. Основные понятия

Нелинейным разветвленным списком является список, элементами которого могут быть тоже списки. В разделе 5.2 мы рассмотрели двухсвязные линейные списки. Если один из указателей каждого элемента списка задает порядок обратный к порядку, устанавливаемому другим указателем, то такой двусвязный список будет линейным. Если же один из указателей задает порядок произвольного вида, не являющийся обратным по отношению к порядку, устанавливаемому другим указателем, то такой список будет нелинейным.

В обработке нелинейный список определяется как любая последовательность атомов и списков (подсписков), где в качестве атома берется любой объект, который при обработке отличается от списка тем, что он структурно неделим.

Библиотека ГГТУ им. П.О.Сухого

Если мы заключим списки в круглые скобки, а элементы списков разделим запятыми, то в качестве списков можно рассматривать такие последовательности:

(a,(b,c,d),e,(f,g))
( )
((a))

Первый список содержит четыре элемента: атом a, список (b,c,d) (содержащий в свою очередь атомы b,c,d), атом e и список (f,g), элементами которого являются атомы f и g. Второй список не содержит элементов, тем не менее нулевой список, в соответствии с нашим определением является действительным списком. Третий список состоит из одного элемента: списка (a), который в свою очередь содержит атом a.

Другой способ представления, часто используемый для иллюстрации списков, - графические схемы, аналогичен способу представления, применяемому при изображении линейных списков. Каждый элемент списка обозначается прямоугольником; стрелки или указатели показывают, являются ли прямоугольники элементами одного и того же списка или элементами подсписка. Пример такого представления дан на рис.5.12.

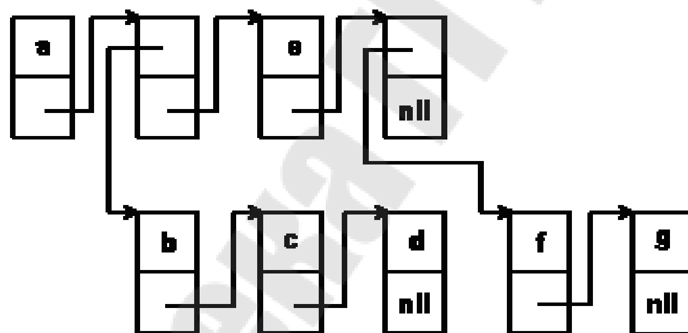


Рис.5.1. Схематическое представление разветвленного списка

Разветвленные списки описываются тремя характеристиками: порядком, глубиной и длиной.

Порядок. Над элементами списка задано транзитивное отношение, определяемое последовательностью, в которой элементы появляются внутри списка. В списке (x,y,z) атом x предшествует y, а y предшествует z. При этом подразумевается, что x предшествует z. Данный список не эквивалентен списку (y,z,x). При представлении списков графическими схемами порядок определяется

Библиотека ГГТУ им. П.О.Сухого

горизонтальными стрелками. Горизонтальные стрелки истолковываются следующим образом: элемент из которого исходит стрелка,предшествует элементу, на который она указывает.

Глубина. Это максимальный уровень, приписываемый элементам внутри списка или внутри любого подсписка в списке. Уровень элемента предписывается вложенностью подсписков внутри списка, т.е.числом пар круглых скобок, окаймляющих элемент. В списке, изображенном на рис.5.12), элементы a и e находятся на уровне 1, в то время как оставшиеся элементы - b, c, d, f и g имеют уровень 2. Глубина входного списка равна 2. При представлении списков схемами концепции глубины и уровня облегчаются для понимания, если каждому атомарному или списковому узлу приписать некоторое число l. Значение 1 для элемента x, обозначаемое как l(x), является числом вертикальных стрелок, которое необходимо пройти для того, чтобы достичь данный элемент из первого элемента списка. На рис.5.12 l(a)=0, l(b)=1 и т.д. Глубина списка является максимальным значением уровня среди уровней всех атомов списка.

Длина - это число элементов уровня 1 в списке. Например, длина списка на рис.5.12 равна 3.

Типичный пример применения разветвленного списка - представление последнего алгебраического выражения в виде списка. Алгебраическое выражение можно представить в виде последовательности элементарных двухместных операций вида:

< операнд 1 > < знак операции > < операнд 2 >

Библиотека ГГТУ им.П.О.Сухого

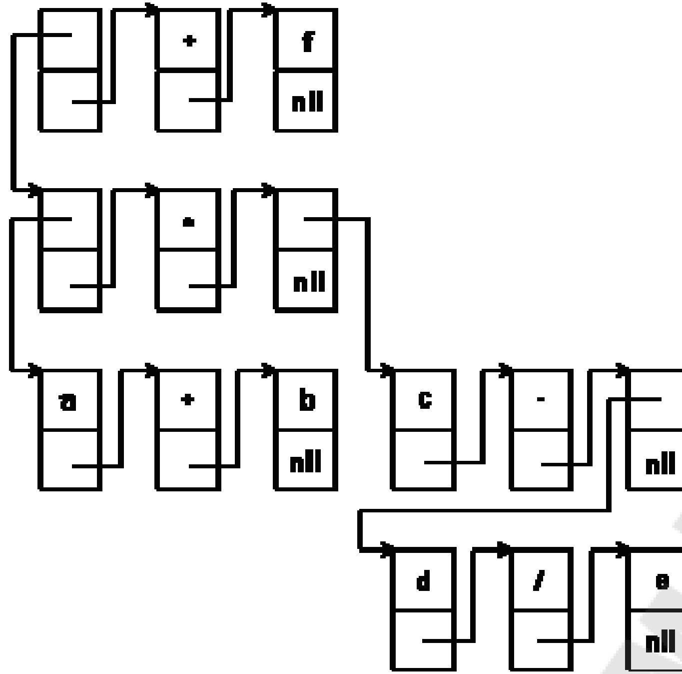


Рис.5.2. Схема списка, представляющего алгебраическое выражение

Выражение:

(a+b)*(c-(d/e))+f

будет вычисляться в следующем порядке:

a+b
d/e
c-(d/e)
(a+b)*(c-d/e)
(a+b)*(c-d/e)+f

При представлении выражения в виде разветвленного списка каждая тройка "операнд-знак-операнд" представляется в виде списка, причем, в качестве операндов могут выступать как атомы - переменные или константы, так и подсписки такого же вида. Скобочное представление нашего выражения будет иметь вид:

(((a,+,b),*,(c,-,(d,/,e)),+,f)

Глубина этого списка равна 4, длина - 3.

5.4.2. Представление списковых структур в памяти

Библиотека ГГТУ им.П.О.Сухого

В соответствии со схематичным изображением разветвленных списков типичная структура элемента такого списка в памяти должна быть такой, как показано на рис.5.14.

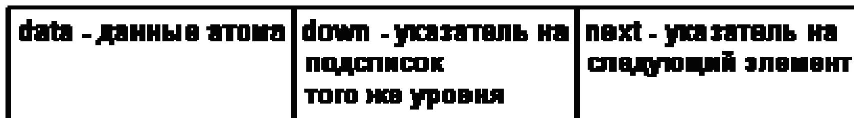


Рис.5.3. Структура элемента разветвленного списка

Элементы списка могут быть двух видов: атомы - содержащие данные и узлы - содержащие указатели на подсписки. В атомах не используется поле down элемента списка, а в узлах - поле data. Поэтому логичным является совмещение этих двух полей в одно, как показано на рис.5.15.

Рис.5.4. Структура элемента разветвленного списка

Поле type содержат признак атом/узел, оно может быть 1-битовым. Такой формат элемента удобен для списков, атомарная информация которых занимает небольшой объем памяти. В этом случае теряется незначительный объем памяти в элементах списка, для которых не требуется поля data. В более общем случае для атомарной информации необходим относительно большой объем памяти. Наиболее распространенный в данной ситуации формат структуры узла представленный на рис.5.16.

Рис. 5.5. Структура элемента разветвленного списка

В этом случае указатель down указывает на данные или на подсписок. Поскольку списки могут составляться из данных различных типов, целесообразно адресовать указателем down не непосредственно данные, а их дескриптор, в котором может быть описан тип данных, их длина и т.п. Само описание того, является ли адресуемый указателем данных объект атомом или узлом также может находиться в этом дескрипторе. Удобно сделать размер дескриптора данных таким же, как и элемента списка. В этом случае размер поля type может быть расширен, например, до 1 байта и это поле может индицировать не только атом/подсписок, но и тип атомарных данных,

Библиотека ГГТУ им. П.О. Сухого

поле next в дескрипторе данных может использоваться для представления еще какой-то описательной информации, например, размера атома. На рис.5.17 показано представление элементами такого формата списка: (КОВАЛЬ,(12,7,53),d). Первая (верхняя) строка на рисунке представляет элементы списка, вторая - элементы подсписка, третья - дескрипторы данных, четвертая - сами данные. В поле type каждого элемента мы использовали коды: n - узел, S - атом, тип STRING, I - атом, тип INTEGER, C - атом, тип CHAR.

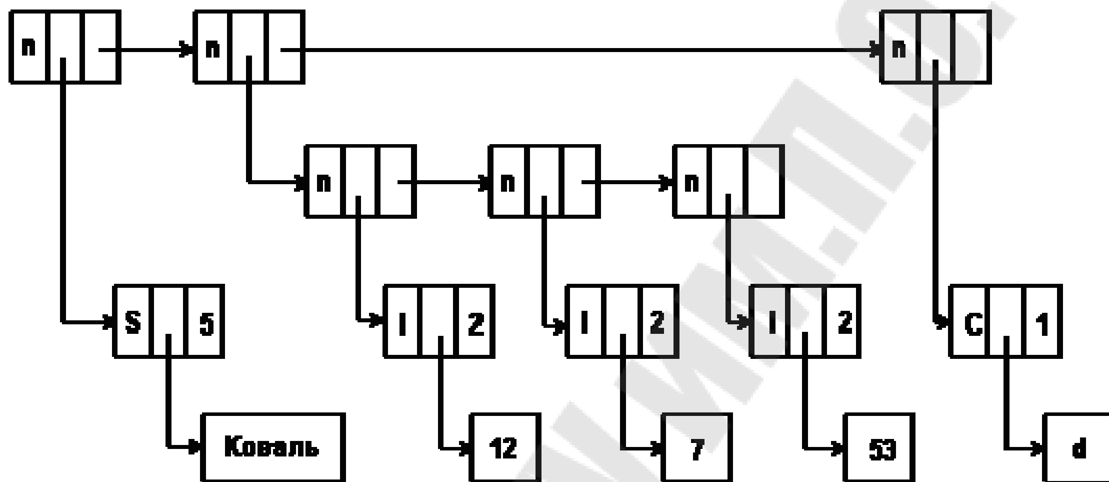


Рис.5.6. Пример представления списка элементами одного формата

5.6. Операции обработки списков

Базовыми операциями при обработке списков являются операции (функции): car, cdr, cons и atom.

Операция car в качестве аргумента получает список (указатель на начало списка). Ее возвращаемым значением является первый элемент этого списка (указатель на первый элемент). Например:

- если X - список (2,6,4,7), то car(X) - атом 2;
- если X - список ((1,2),6), то car(X) - список (1,2);
- если X - атом то car(X) не имеет смысла и в действительности не определено.

Библиотека ГГТУ им. П.О.Сухого

Операция cdr в качестве аргумента также получает список. Ее возвращаемым значением является остаток списка - указатель на список после удаления из него первого элемента. Например:

- если X - (2,6,4), то cdr(X) - (6,4);
- если X - ((1,2),6,5), то cdr(X) - (6,5);
- если список X содержит один элемент, то cdr(X) равно nil.

Операция cons имеет два аргумента: указатель на элемент списка и указатель на список. Операция включает аргумент-элемент в начало аргумента-списка и возвращает указатель на получившийся список. Например:

- если X - 2, а Y - (6,4,7), то cons(X,Y) - (2,6,4,7);
- если X - (1,2), Y - (6,4,7), то cons(X,Y) - ((1,2),6,4,7).

Операция atom выполняет проверку типа элемента списка. Она должна возвращать логическое значение: true - если ее аргумент является атомом или false - если ее аргумент является подсписком.

В программном примере 5.11 приведена реализация описанных операций как функций языка PASCAL. Структура элемента списка, обрабатываемого функциями этого модуля определена в нем как тип litem и полностью соответствует рис.5.16. Помимо описанных операций в модуле определены также функции выделения памяти для дескриптора данных - NewAtom и для элемента списка - NewNode. Реализация операций настолько проста, что не требует дополнительных пояснений.

Библиотека ГГТУ им. П.О. Сухого

6. Хеширование и хеш-таблицы

6.1. Понятие хеширования

В курсе "Основы алгоритмизации и программирования" были рассмотрены методы поиска элемента (записи) в массиве, основанные на применении операций сравнения (в общем случае функций сравнения). Прежде, чем найти требуемую запись, необходимо организовать просмотр некоторого количества ключей. Напомним, что ключ - это свойство элемента (записи), по которому элемент однозначно идентифицируется. Очевидно, что эффективными методами поиска являются те методы, которые минимизируют число этих сравнений.

Наиболее быстрым из рассмотренных методов был бинарный поиск, для которого требовался отсортированный контейнер. Так, чтобы установить наличие или отсутствие заданного элемента в массиве из 1000 элементов требуется примерно 10 сравнений. Бинарный поиск представляет собой алгоритм класса O(log(n)) (2^{10}=1024).

Возможен более эффективный подход к поиску элементов массива. Суть его заключается в следующем. Элемент массива связывается с уникальным индексом. Создается новый массив (таблица), в который в позицию, задаваемую индексом, записывается признак наличия элемента в исходном массиве, например, адрес элемента. Поиск элемента после этого можно осуществить посредством однонаправленного действия: просто извлекая элемент, расположенный в позиции.

Преобразование ключа элемента в значение индекса называется ***хешированием*** (*hashing*). Полученный индекс называется ***хеш-адресом***. Хеширование осуществляется с помощью функции, называемой ***хеш-функцией***. Массив, предназначенный для хранения элементов, с которым используется значение индекса, называют ***хеш-таблицей*** (*hash table*). В результате хеширования происходит преобразование входного массива данных произвольной длины в выходную битовую строку фиксированной длины. Такие преобразования также называются функциями свёртки, а их результаты называют хешем, хеш-кодом, или дайджестом сообщения (англ. message digest).

Т.о., ***хеширование*** - это преобразование элемента входного массива данных в короткое число фиксированной длины (которое называется ***хешем*** или ***хеш-кодом***) таким образом, чтобы с одной стороны, это число было значительно короче исходных данных, а с другой стороны, с большой вероятностью однозначно им соответствовало.

В программировании хеш-таблица — это структура данных, позволяющая хранить пары (ключ, значение) и выполнять три операции:

- операцию добавления новой пары;
- операцию поиска;
- операцию удаления пары по ключу.

Наиболее эффективным способом организации такой структуры данных является массив.

Существует два варианта хэш-таблиц: с прямой и открытой адресацией. Хэш-таблица содержит некоторый массив H, элементы которого есть пары (хэш-таблица с открытой адресацией) или списки пар (хэш-таблица с прямой адресацией).

Если каждый ключ должен быть извлечен за один доступ, то положение записи внутри такой таблицы может зависеть только от данного ключа. Оно не может зависеть от расположения других ключей.

Чтобы можно было выполнить поиск с использованием хеширования, требуется реализация двух отдельных алгоритмов. ***Первый - процесс хеширования***, при помощи которого ключ элемента преобразуется в массив значений индекса. В идеальном случае различные ключи должны были бы хешироваться в различные значения индекса. Но это нельзя гарантировать, и зачастую два различных ключа будут представлены одним и тем же значением индекса. Поэтому требуется ***второй алгоритм***, определяющий наши действия в подобных случаях.

Отображение двух и более ключей на один и тот же индекс называют ***конфликтом*** или ***коллизией*** (*collision*), а ***второй алгоритм***, необходимый для исправления этой ситуации, называется ***разрешением конфликтов*** (*collision resolution*).

Такие события не так уж и редки — например, при вставке в хеш-таблицу размером 365 ячеек, всего лишь 23-х элементов, вероятность коллизии уже превысит 50 % (если каждый элемент может равновероятно попасть в любую ячейку) — см. парадокс дней

Библиотека ГГТУ им. П.О.Сухого

рождения. Поэтому механизм разрешения коллизий — важная составляющая любой хеш-таблицы. Парадокс дней рождения утверждает, что если в комнате присутствует не менее 23 человек, имеется хороший шанс на то, что у двух из них совпадет день рождения. Иными словами, если мы выберем случайную функцию, отображающую 23 ключа в 365 элементную таблиц, то с вероятностью 0.4927 (менее половины) все ключи попадут в разные места.

В некоторых специальных случаях удаётся избежать коллизий вообще. Например, если все ключи элементов известны заранее (или очень редко меняются), то для них можно найти некоторую ***совершенную*** хеш-функцию, которая распределит их по ячейкам хеш-таблицы без коллизий. Хеш-таблицы, использующие подобные хеш-функции, не нуждаются в механизме разрешения коллизий, и называются ***хеш-таблицами с прямой адресацией***.

Хеш-таблица - прекрасный пример достижения компромисса между быстродействием алгоритма поиска и занимаемым объемом памяти. Хеш-таблицы будут занимать больше места в памяти, причем некоторые элементы окажутся пустыми, тем не менее, применение функции хеширования позволяет найти элемент в результате очень небольшого числа обращений к элементам хеш-таблицы - обычно одного при тщательно выполненном хешировании.

Простым примером хеширования может служить нахождение циклической контрольной суммы, когда берётся текст (или другие данные) и суммируются коды входящих в него символов, а затем отбрасываются все цифры, за исключением нескольких последних. Полученное число может являться примером хеш-кода исходного текста. Например, контрольная сумма может быть передана по каналу связи вместе с основным текстом. На приёмном конце, контрольная сумма может быть рассчитана заново и её можно сравнить с переданным значением. Если будет обнаружено расхождение, то это значит, что при передаче возникли искажения и можно запросить повтор.

Бытовым аналогом хеширования может служить приём, когда при переездах в памяти держат количество мест багажа. Тогда для проверки не нужно вспоминать про каждый чемодан, а достаточно их

Библиотека ГГТУ им. П.О.Сухого

посчитать. Совпадение будет означать, что ни один чемодан не потерян. То есть, количество мест багажа является его хеш-кодом.

Другим бытовым аналогом хеширования может служить помещение слов в словаре по алфавиту. Первая буква слова является его хеш-кодом, и при поиске мы просматриваем не весь словарь, а только нужную букву.

Хеширование применяется для сравнения данных: если у двух массивов хеш-функции разные, массивы гарантированно различаются; если одинаковые — массивы, скорее всего, одинаковы. В общем случае однозначного соответствия между исходными данными и хеш-кодом нет в силу того, что количество значений хеш-функций меньше чем вариантов входного массива; существует множество массивов, дающих одинаковые хеш-коды — так называемые коллизии. Вероятность возникновения коллизий играет немаловажную роль в оценке качества хеш-функций.

Пример хеширования.

Предположим, что фирма некоторая фирма выпускает детали и кодирует их семизначными цифрами. Для применения прямой индексации с использованием полного семизначного ключа потребовался бы массив из 100 млн. элементов. Ясно, что это привело бы к потере неприемлемо большого пространства, поскольку совершенно невероятно, что какая-либо фирма может иметь больше чем тысяча наименований изделий. Поэтому необходим некоторый метод преобразования ключа в какое-либо целое число внутри ограниченного диапазона. Например, если в качестве индекса записи об изделии в этом массиве использовать три последние цифры номера изде лия, то для хранения всего файла будет достаточно массива из 1000 элементов. Этот массив индексируется целым числом в диапазоне от 0 до 999 включительно.

Существует множество алгоритмов хеширования с различными характеристиками (разрядность, вычислительная сложность, криптостойкость и т. п.). Выбор той или иной хеш-функции определяется спецификой решаемой задачи. Простейшими примерами хеш-функций могут служить контрольная сумма или CRC.

Выполнение операции в хэш-таблице начинается с вычисления хэш-функции от ключа. Получающееся хэш-значение i = hash(key)

Библиотека ГГТУ им.П.О.Сухого

играет роль индекса в массиве H. Затем выполняемая операция (добавление, удаление или поиск) перенаправляется объекту, который хранится в соответствующей ячейке массива H[i].

Число хранимых элементов, делённое на размер массива H (число возможных значений хэш-функции) называется ***коэффициентом заполнения хэш-таблицы*** (load factor) и является важным параметром, от которого зависит среднее время выполнения операций.

Свойства хеш-таблицы

С хешированием мы сталкиваемся едва ли не на каждом шагу: при работе с браузером (список Web-ссылок), текстовым редактором и переводчиком (словарь), языками скриптов (Perl, Python, PHP и др.), компилятором (таблица символов). По словам Брайана Кернигана, это «одно из величайших изобретений информатики». Заглядывая в адресную книгу, энциклопедию, алфавитный указатель, мы даже не задумываемся, что упорядочение по алфавиту является не чем иным, как хешированием.

Хеширование используется для уменьшения времени доступа к дисковым файлам. Однако, тот же метод можно использовать для реализации разреженных матриц.

Хеширование полезно там, где необходимо большой диапазон возможных значений сохранить в небольшом объеме памяти, и отыскать простым, почти произвольным доступом. Также, хэш-таблицы часто используются в базах данных и, особенно в языковых процессорах типа компиляторов и ассемблеров, где они обслуживают таблицы идентификаторов. В таких приложениях, хэш-таблица - принципиальная структура данных. Так как всякий доступ к таблице должен быть выполнен через хэш-функцию, то она должна удовлетворять двум требованиям: она должна работать быстро, и выдавать хорошие ключи распределения по таблице. Последнее требование минимизирует коллизии и предотвращает хеширование элементов с одинаковыми значениями ключей только в одной части таблицы.

Библиотека БГУИР

6.2. Функции хеширования

6.2.1. Основные понятия

Функция хеширования - это подпрограмма, которая будет принимать ключ элемента и преобразовывать его в значение индекса.

Очевидно, что если в хеш-таблице предусмотрено место для n элементов, то функция хеширования должна создавать значения индексов в диапазоне от 0 до n-1 (значения индексов начинаются с нуля).

Для различных типов ключей будут использоваться различные функции хеширования.

В идеале функция хеширования должна быть подобной функции рандомизации: для любого вводимого значения в определенном смысле выводимое значение должно быть равновероятным.

Существует множество алгоритмов хеширования с различными характеристиками (разрядность, вычислительная сложность, криптостойкость и т. п.). Выбор той или иной хеш-функции определяется спецификой решаемой задачи.

В общем случае хеш-функции зависят от процесса преобразования ключей в целые числа, поэтому в реализациях хеширования иногда трудно одновременно обеспечить независимость от компьютера и эффективность.

Как правило, простое целочисленное значение или ключи типа с плавающей точкой можно преобразовать, выполнив всего одну машинную операцию, но строковые ключи и другие типы составных ключей требуют больше затрат и больше внимания в плане достижения высокой эффективности.

Считается, что совершенных хеш-функций не существует, и приходится иметь дело с неизбежными конфликтами, когда разные ключи будут иметь одинаковые индексы.

Рассмотрим ряд функций хеширования.

6.2.2. Хеширование для чисел с плавающей точкой, заведомо относящихся к фиксированному диапазону

Если ключи — числа, которые больше 0 и меньше 1, их можно просто умножить на M, округлить до ближайшего целого

Библиотека ГГТУ им. П.О.Сухого

числа и получить адрес в диапазоне между 0 и M— 1; пример показан на рис. 1.1 (M=100).

.513670656	51
.175725579	17
.308633685	30
.534531713	53
.947630227	94
.171727657	17
.702230930	70
,226416826	22
,494766086	49
.124698631	12
,083895385	8
.389629811	38
.277230144	27
.368053228	36
.983458996	98
.535386205	53
.765678883	76
,646473587	64
.767143786	76
.780236185	78
.822962105	82
,151921138	15
.625476837	62
.314676344	31
,346903890	34

РИСУНОК 6.1
МУЛЬТИПЛИКАТИВНАЯ ХЕШ-ФУНКЦИЯ ДЛЯ КЛЮЧЕЙ С ПЛАВАЮЩЕЙ ТОЧКОЙ

В этом примере имеют место три конфликта: при значениях индексов равных 17, 53 и 76.

Для преобразования чисел с плавающей точкой в диапазоне между 0 и 1 в индексы таблицы, размер которой равен, например.97, выполняется умножение на 97. Старшие разряды ключа определяют хеш-значения; младшие разряды ключей не играют никакой роли.

Библиотека ГГТУ им.П.О.Сухого

Одна из целей разработки хеш-функции — избежание дисбаланса, когда во время вычисления каждый разряд данных играет определенную роль.

Если ключи больше s и меньше t, их можно масштабировать, вычтя s из значения ключа и разделив результат на t-s, в результате чего они попадут в диапазон значений между 0 и 1, а затем умножить на M для получения адреса в таблице.

Если ключи — это w-разрядные целые числа, их можно преобразовать в числа с плавающей точкой и разделить на 2^w для получения чисел с плавающей точкой в диапазоне между 0 и 1, а затем умножить на M.

Если операции с плавающей выполняются часто, а числа не столь велики, чтобы приводить к переполнению, этот же результат может быть получен с использованием арифметических операций над целыми числами: необходимо умножить ключ на M, затем выполнить сдвиг вправо на w разрядов для деления на 2^W (или, если умножение приводит к переполнению, выполнить сдвиг, а затем умножение). Такие функции бесполезны для хеширования, если только ключи не распределены по диапазону равномерно, поскольку хеш-значение определяется только ведущими цифрами ключа.

6.2.3. Модульное хеширование

Более простой и эффективный метод для w-разрядных целых чисел — один из, вероятно, наиболее часто используемых методов хеширования — выбор в качестве размера M таблицы простого числа и вычисление остатка от деления k на M, или h(k) = к mod M для любого целочисленного ключа k. Такая функция называется ***модульной хеш-функцией***. Значение ее очень просто вычислить (k % M в языке C/C++), и она эффективна для достижения равномерного распределения значений ключей между значениями, которые меньше M. Небольшой пример показан на рис. 6.2.

Библиотека ГГТУ им. П.О. Сухого

Значение ключа (к)	к % 97	к % 100	(int) (a* к)% 100 a = .618033…
16838	57	38	6
5758	35	58	58
10113	25	13	50
17515	65	15	24
31051	11	51	90
6627	1	27	77
23010	21	10	20
7419	47	19	85
16212	13	12	19
4086	12	86	25
2749	33	49	98
12767	60	67	90
9084	63	84	14
12060	32	60	53
32225	21	25	16
17543	83	43	42
25089	63	89	5
21183	37	83	91
25137	14	37	35
25566	55	66	0
26966	0	66	65
4978	31	78	76
20495	28	95	66
10311	29	11	72
11367	18	67	25

РИСУНОК 6.2 МОДУЛЬНАЯ ХЕШ-ФУНКЦИЯ ДЛЯ ЦЕЛОЧИСЛЕННЫХ КЛЮЧЕЙ

В трех правых столбцах показан результат хеширования 16-разрядных ключей, приведенных слева, с помощью следующих функций к % 97 (слева) к % 100 (в центре) и (int) (а*к)% 100 (справа), где а = .618033. Размеры таблицы для этих функций соответственно равны 97, 100 и 100. Значения оказываются случайными (поскольку ключи случайны). Центральная функция (v % 100) использует только

Библиотека ГГТУ им. П.О. Сухого

две крайние справа цифры ключей и поэтому может показывать низкую производительность применительно к неслучайным ключам. Здесь к - значение ключа.

Модульное хеширование можно использовать также для ключей с плавающей точкой. Если ключи относятся к небольшому диапазону, можно выполнить масштабирование с целью их преобразования в числа из диапазона между 0 и 1, умножить на 2^w для получения w-разрядных целочисленных значений, а затем использовать модульную хеш-функцию. Другая альтернатива — просто использовать в качестве операнда модульной хеш-функции двоичное представление ключа (если оно доступно).

Модульное хеширование применяется во всех случаях, когда имеется доступ к разрядам, образующим ключи, независимо от того, являются ли они целыми числами, представленными машинным словом, последовательностью символов, упакованных в машинное слово, или представлены одним из множества других возможных вариантов. Последовательность случайных символов, упакованная в машинное слово — не совсем то же, что случайные целочисленные ключи, поскольку некоторые разряды используются для кодирования. Но оба эти типа (и любой другой тип ключа, который кодируется так, чтобы уместиться в машинном слове) можно заставить выглядеть случайными индексами в небольшой таблице.

Основная причина выбора в качестве размера М хеш-таблицы простого числа для модульного хеширования иллюстрируется на рис. 1.3

now	6733767	1816567	55	29
for	6333762	1685490	50	20
tip	7232360	1914096	**48**	1
ilk	6473153	1734251	43	18
dim	6232355	1651949	45	21
tag	7230347	1913063	39	22
jot	6533764	1751028	***52***	24
sob	7173742	1898466	34	26
nob	6733742	1816546	34	8
sky	7172771	1897977	57	2
hut	6435364	1719028	***52***	16
ace	6070745	1602021	**37**	3
bet	6131364	1618676	***52***	11

Библиотека ГГТУ им. П.О. Сухого

men	6671356	1798894	46	26
egg	6271747	1668071	39	23
few	6331367	1684215	55	16
jay	6530371	1749241	57	4
owl	6775754	1833964	44	4
joy	6533771	1751033	57	29
rap	7130360	1880304	**48**	30
gig	6372347	1701095	39	1
wee	7371345	1962725	**<u>37</u>**	22
was	7370363	1962227	51	20
cab	6170342	1634530	<u>34</u>	24
vad	7370344	1962212	36	5

РИСУНОК 6.3 МОДУЛЬНЫЕ ХЕШ-ФУНКЦИИ ДЛЯ КОДИРОВАННЫХ СИМВОЛОВ

В каждой строке этой таблицы приведены 3х-символьное слово, представление этого слова в ASCII-коде как 21-разрядное число в восьмеричной и десятичной формах и стандартные модульные хеш-функции для таблицы с размерами 64 и 31 (два крайних справа столбца). Размер таблицы, равный 64, приводит к нежелательным результатам, поскольку для получения хеш- значения используются только самые правые разряды ключа, а символы в словах обычного языка распределены неравномерно. Например, всем словам, завершающимся на букву у, соответствует хеш- значение 57. И напротив, простое значение 31 вызывает меньше конфликтов в таблице, размер которой составляет менее половины предыдущей.

В этом примере символьных данных с 7-разрядным кодированием ключ трактуется как число с основанием 128 — по одной цифре для каждого символа в ключе. Слово now соответствует числу 1816567, которое может быть также записано как $110*128^2+ 111*128^1 + 119*128^0$ поскольку в ASCII-коде символам n, o и w соответствуют числа $156_8= 110$, $157_8= 111$ и $167_8= 119$. Далее, выбор размера таблицы M = 64 неудачен для этого типа ключа, поскольку добавление к x значений, кратных 64 (или 128), не оказывает влияния на значение x mod 64 — для любого ключа значением хеш-функции является значение последних 6 разрядов этого ключа. Безусловно, хорошая хеш-функция должна учитывать все разряды ключа,

Библиотека БГТУ им. П.О. Сухого

особенно для ключей, образованных символами. Аналогичные ситуации могут возникать, когда M содержит множитель, являющийся степенью 2. Простейший способ избежать этого — выбрать в качестве M простое число.

Модульное хеширование весьма просто реализовать, за исключением того, что размер таблицы необходимо определить простым числом. Для некоторых приложений можно довольствоваться небольшим известным простым числом или же поискать простое число, близкое к требуемому размеру таблицы, в списке известных простых чисел. Например, числа равные $2^t - 1$ являются простыми при t= 2, 3, 5, 7, 13, 19 и 31(и ни при каких других значениях $t < 32$): это хорошо известные простые числа Мерсенне (Mersenne).

Чтобы динамически распределить таблицу определенных размеров, нужно вычислить простое число, близкое к определенному значению. Это вычисление не тривиально (хотя и существует остроумный алгоритм выполнения этой задачи), поэтому на практике обычно используют таблицу заранее вычисленных значений (см. рис. 6.4).

n	δ_n	$2^n - \delta_n$
8	5	251
9	3	509
10	3	1021
11	9	2039
12	3	4093
13	1	8191
14	3	16381
15	19	32749
16	15	65521
17	1	131071
18	5	262139
19	1	524287
20	3	1048573
21	9	2097143
22	3	4194301
23	15	8388593
24	3	16777213
25	39	33554393
26	5	67108859

Библиотека ГГТУ им. П.О.Сухого

27	39	134217689
28	57	268435399
29	3	536870909
30	35	1073741789
31	1	2147483647

РИСУНОК 6.4 ПРОСТЫЕ ЧИСЛА ДЛЯ ХЕШ-ТАБЛИЦ

Эта таблица наибольших простых чисел, которые меньше 2^n для $8 <= n < 32$, может использоваться для динамического распределения хеш-таблицы, когда требуется, чтобы размер таблицы определялся простым числом. Для любого данного положительного значения в указанном диапазоне эту таблицу можно использовать для получения простого числа, отличающегося от него менее чем в 2 раза.

Другая альтернатива обработки целочисленных ключей — объединение мультипликативного и модульного методов: следует умножить ключ на константу в диапазоне между 0 и 1, а затем выполнить деление по модулю М. Другими словами, необходимо использовать функцию $h(k) = [k\alpha] \bmod M$. Между значениями α, М и эффективным основанием ключа существует взаимодействие, которое теоретически могло бы привести к аномальному поведению, но если использовать умеренное значение ее, в реальном приложении вряд ли придется столкнуться с какой-либо проблемой. Часто в качестве α выбирают значение $\phi = 0.618033...$ (золотое сечение).

Изучено множество других вариаций на эту тему, в частности хеш-функции, которые могут быть реализованы с помощью таких эффективных машинных инструкций, как сдвиг и маскирование.

6.2.4. Хеш-функции для длинных алфавитно-цифровых строк

Во многих приложениях, в которых используются таблицы символов, ключи не являются числами и необязательно являются короткими; чаще они оказываются алфавитно-цифровыми строками, которые могут быть длинными. Например, как вычислить хеш-функцию для такого слова, как

averylongkey?

В 7-разрядном ASCII-коде этому слову соответствует 84-разрядное число

Библиотека БГУИР им. П.О. Сухого

$$97*128^{11} + 118*128^{10} + 101*128^{9} + 114*128^{8} + 121*128^{7}$$
$$+ 108*128^{6} + 111*128^{5} + 110*128^{4} + 103*128^{3}$$
$$+ 107*128^{2} + 101*128^{1} + 121*128^{0},$$

которое слишком велико, чтобы его можно было представить в большинстве компьютеров. Более того, часто требуется обрабатывать значительно более длинные ключи. Чтобы вычислить модульную хеш-функцию для длинных ключей, последние преобразуются фрагмент за фрагментом. Можно воспользоваться арифметическими свойствами функции mod и задействовать алгоритм Горнера (Homer).

Этот метод основывается на еще одном способе записи чисел, соответствующих ключам. Для рассматриваемого примера запишем следующее выражение:

$$(((((((((((97*128 + 118)*128 + 101)*128 + 114)*128 + 121)*128$$
$$+ 108)*128 + 111)*128 + 110)*128 + 103)*128$$
$$+ 107)*128 + 101)*128 + 121.$$

То есть, можно вычислить десятичное число, соответствующее коду символа строки при просмотре ее слева направо, умножая полученное значение на 128, а затем добавляя кодовое значение следующего символа. Со временем, в случае длинной строки, этот способ вычисления создал бы число, которое больше того, которое вообще можно представить в компьютере. Однако вычисленное число нас не интересует, поскольку требуется только остаток от его деления на M, который мал. Результат можно получить, даже не сохраняя большое накопленное значение, т.к. в любой момент вычисления можно отбросить число, кратное M — при каждом выполнении умножения и сложения нужно хранить только остаток деления по модулю M. Результат та- кой же, как если бы у нас имелась возможность вычислять длинное число, а затем выполнять деление. Это наблюдение ведет к непосредственному арифметическому способу вычисления модульных хеш-функций для длинных строк (см. программу 1.1). В программе используется еще одно, последнее ухищрение: вместо основания 128 в ней используется простое число 127. Причина такого изменения рассматривается в следующем абзаце.

Программа 1.1 Хеш-функция для строковых ключей:

```
int hash (char *v, int M)     //v - указатель на строку, M-размер хеш-
                                    таблицы
  {
    int h = 0, a = 127;  //h – хеш-код строки
    while (*v != 0)   //пока не конец строки
```


Библиотека ГГТУ им.П.О.Сухого

```
        {
         h = (a*h + *v) % M;
          v++;
        }
   return (h);
  }
```

Эта реализация хеш-функции для строковых ключей требует одного умножения и одного сложения на каждый символ в ключе. Если бы константу 127 мы заменили константой 128, программа просто вычисляла бы остаток от деления числа, соответствующего 7-разрядному ASCII-представлению ключа, на размер таблицы с использованием метода Горнера. Простое основание, равное 127, помогает избежать аномалий, если размер таблицы является степенью 2 или кратным 2.

Библиотека ГГТУ им. П.О.Сухого

7.1. Графы

7.1.1. Логическая структура, определения

Граф - это сложная нелинейная многосвязная динамическая структура, отображающая свойства и связи сложного объекта.

Многосвязная структура обладает следующими свойствами:

1) на каждый элемент (узел, вершину) может быть произвольное количество ссылок;

2) каждый элемент может иметь связь с любым количеством других элементов;

3) каждая связка (ребро, дуга) может иметь направление и вес.

В узлах графа содержится информация об элементах объекта. Связи между узлами задаются ребрами графа. Ребра графа могут иметь направленность, показываемую стрелками, тогда они называются ориентированными, ребра без стрелок - неориентированные.

Граф, все связи которого ориентированные, называется ***ориентированным графом*** или ***орграфом***; граф со всеми неориентированными связями - ***неориентированным графом***; граф со связями обоих типов - ***смешанным графом***.

Обозначение связей:

неориентированных - (A,B), ориентированных - < A,B > , где A, B - вершины графа.

Примеры изображений графов и скобочное представление графов даны на рис.7.1:

а). ((A,B), (B,A)) б). (< A,B >, < B,A >).

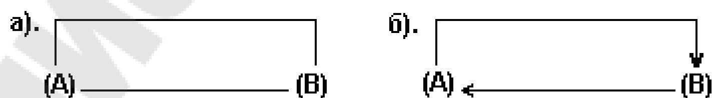


Рис.7.1. Граф неориентированный (а) и ориентированный (б).

Библиотека ГГТУ им.П.О.Сухого

Для ориентированного графа число ребер, входящих в узел, называется ***полустепенью захода узла***, выходящих из узела - ***полустепенью исхода***. Количество входящих и выходящих ребер может быть любым, в том числе и нулевым. Граф без ребер является ***нуль-графом***.

Если ребрам графа соответствуют некоторые значения, то граф и ребра называются ***взвешенными***. ***Мультиграфом*** называется граф, имеющий параллельные (соединяющие одни и те же вершины) ребра, в противном случае граф называется ***простым***.

Путь в графе - это последовательность узлов, связанных ребрами; ***элементарным*** называется путь, в котором все ребра различны, ***простым*** называется путь, в котором все вершины различны. Путь от узла к самому себе называется ***циклом***, а граф, содержащий такие пути - ***циклическим***.

Два узла графа ***смежны***, если существует путь от одного из них до другого. Узел называется ***инцидентным к ребру***, если он является его вершиной, т.е. ребро направлено к этому узлу.

Логически структура-граф может быть представлена ***матрицей смежности*** или ***матрицей инцидентности***.

Матрицей смежности для n узлов называется квадратная матрица a порядка n. Элемент матрицы a(i,j) равен 1, если узел j смежен с узлом i (есть путь < i,j >), и 0 -в противном случае (рис.6.2).

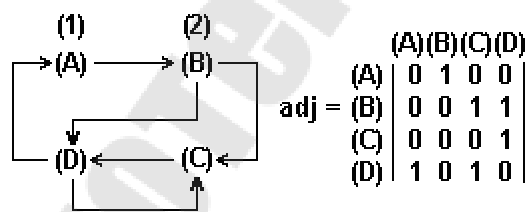


Рис.7.2. Граф и его матрица смежности

Если граф неориентирован, то a(i,j)=a(j,i), т.е. матрица симметрична относительно главной диагонали.

Библиотека ГГТУ им. П.О.Сухого

Матрицы смежности используются при построении матриц путей, дающих представление о графе по длине пути: путь длиной в 1 - смежный участок - < A,B >, путь длиной 2 - (< A,B >,< B,C >), ... в n смежных участков: где n - максимальная длина, равная числу узлов графа. На рис.7.3 даны путевые матрицы пути adj2, adj3, adj4 для графа рис.7.2.

adj2

	(A)	(B)	(C)	(D)
(A)	0	0	1	1
(B)	1	0	1	1
(C)	1	0	1	0
(D)	0	1	0	1

adj3

	(A)	(B)	(C)	(D)
(A)	1	0	1	1
(B)	1	1	1	1
(C)	0	1	0	1
(D)	0	0	1	1

adj4

	(A)	(B)	(C)	(D)
(A)	1	0	1	0
(B)	0	1	0	0
(C)	0	0	1	1
(D)	0	0	1	1

Рис.7.3. Матрицы путей

Матрицы инцидентности используются только для орграфов. В каждой строке содержится упорядоченная последовательность имен узлов, с которыми данный узел связан ориентированными (исходящими) ребрами. На рис.6.4 показана матрица инцидентности для графа рис. 7.2.

Узлы	1	2
A	B	-
B	C	D
C	D	-
D	A	C

номера связей

Рис.7.4. Матрицы инцидентности

7.1.2. Машинное представление орграфов

Существуют два основных метода представления графов в памяти ЭВМ: матричный, т.е. массивами, и связными нелинейными списками. Выбор метода представления зависит от природы данных и операций, выполняемых над ними. Если задача требует большого числа включений и исключений узлов, то целесообразно представлять граф связными списками; в противном случае можно применить и матричное представление.

МАТРИЧНОЕ ПРЕДСТАВЛЕНИЕ ОРГРАФОВ.

Библиотека ГГТУ им.П.О.Сухого

При использовании матриц смежности их элементы представляются в памяти ЭВМ элементами массива. При этом, для ***простого графа*** матрица состоит из нулей и единиц, для ***мультиграфа*** - из нулей и целых чисел, указывающих кратность соответствующих ребер, для ***взвешенного графа*** - из нулей и вещественных чисел, задающих вес каждого ребра.

Например, для простого ориентированного графа, изображенного на рис.6.2 массив определяется как:

mas:array[1..4,1..4]=((0,1,0,0),(0,0,1,1),(0,0,0,1),(1,0,1,0))

Матрицы смежности применяются, когда в графе много связей и матрица хорошо заполнена.

СВЯЗНОЕ ПРЕДСТАВЛЕНИЕ ОРГРАФОВ.

Орграф представляется связным нелинейным списком, если он часто изменяется или если полустепени захода и исхода его узлов велики. Рассмотрим два варианты представления орграфов связными нелинейными списковыми структурами.

В первом варианте два типа элементов - атомарный и узел связи На рис.7.5 показана схема такого представления для графа рис.7.2. Скобочная запись связей этого графа:

(< A,B >, < B,C >, < C,D >, < B,D >, < D,C >)

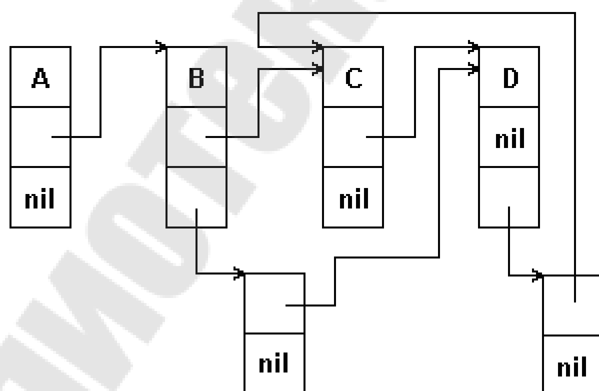


Рис.7.5. Машинное представление графа элементами двух типов

Библиотека ГГТУ им.П.О.Сухого

Более рационально представлять граф элементами одного формата, двойными: атом-указатель и указатель-указатель или тройными: указатель-data/down-указатель. На рис.7.6 тот же граф представлен элементами одного формата.

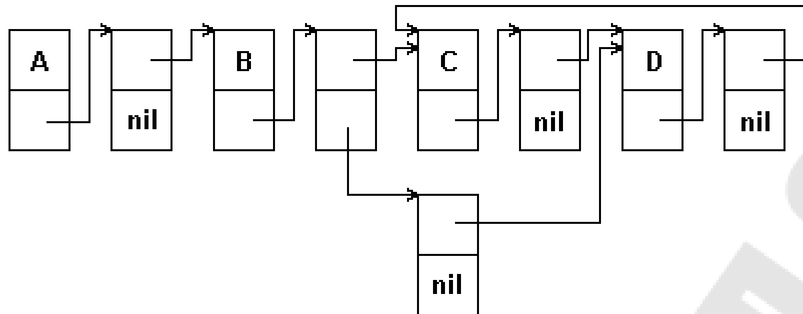


Рис.7.6. Машинное представление графа однотипными элементами

Многосвязная структура - граф - находит широкое применение при организации банков данных, управлении базами данных, в системах программного имитационного моделирования сложных комплексов, в системах искусственного интеллекта, в задачах планирования и в других сферах. В качестве примера приведем программу, находящую кратчайший путь между двумя указанными вершинами связного конечного графа.

Пусть дана часть карты дорожной сети и нужно найти наилучший маршрут от города 1 до города 5. Такая задача выглядит достаточно простой, но "наилучший" маршрут могут определять многие факторы. Например: (1) расстояние в километрах; (2) время прохождения маршрута с учетом ограничений скорости; (3) ожидаемая продолжительность поездки с учетом дорожных условий и плотности движения; (4) задержки, вызванные проездом через города или объездом городов; (5) число городов, которое необходимо посетить, например, в целях доставки грузов.

Задачи о кратчайших путях относятся к фундаментальным ***задачам комбинаторной оптимизации***.

Среди десятков алгоритмов для отыскания кратчайшего пути один из лучших принадлежит **Дейкстре**. Алгоритм Дейкстры, определяющий кратчайшее расстояние от данной вершины до конечной, легче пояснить на примере.

Библиотека ГГТУ им.П.О.Сухого

Рассмотрим граф, изображенный на рис.7.7, задающий связь между городами на карте дорог.

Представим граф матрицей смежности A, в которой: A(i, j)- длина ребра между узлами i и j. Используя полученную матрицу и матрицы, отражающие другие факторы, можно определить кратчайший путь.

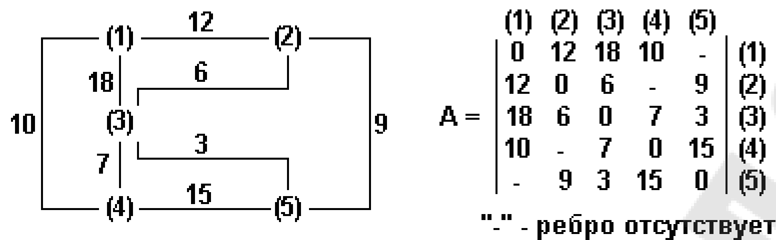


Рис.7.7. Часть дорожной карты, представленная в виде взвешенного графа и его матрицы смежности

```
{========== Программный пример 6.1 ==============}
{ Алгоритм Дейкстры }
Program ShortWay;
Const n=5;   max=10000;
Var a: Array [1..n,1..n] of Integer;
   v0,w,edges: Integer;
   from,tu,length: Array [1..n] of Integer;
Procedure adjinit;
{ Эта процедура задает веса ребер графа  посредством
определения его матрицы смежности A размером N x N }
Var i,j: Integer;
Begin
{ "Обнуление" матрицы (вершины не связаны) }
 For i:=1 to n do
  For j:=1 to n do a[i,j]:=max;
 For i:=1 to n do a[i,i]:=0;
{ Задание длин ребер, соединяющих смежные узлы графа }
            a[1,2]:=12; a[1,3]:=18; a[1,4]:=10;
 a[2,1]:=12;            a[2,3]:=6;             a[2,5]:=9;
 a[3,1]:=18; a[3,2]:=6;            a[3,4]:=7;  a[3,5]:=3;
 a[4,1]:=10;            a[4,3]:=7;             a[4,5]:=15;
            a[5,2]:=9;  a[5,3]:=3;  a[5,4]:=15;
End;
```


Библиотека БГТУ им. П.О. Сухого

```
Procedure printmat;
{ Эта процедура выводит на экран дисплея матрицу
смежности A взвешенного графа }
Var i,j: Integer;
Begin  writeln;
 writeln('Матрица смежности взвешенного графа (',n,'x',n,'):');
 writeln;
 For i:=1 to n do
  Begin  write ('Ё');
   For j:=1 to n do
    If a[i,j]=max Then write('  ----')  Else write(a[i,j]:6);
   writeln(' Ё')
  End; writeln;
 writeln (' ("----" - ребро отсутствует)')
End;
Procedure dijkst;
```

{ Эта процедура определяет кратчайшее расстояние от начальной вершины V0 до конечной вершины W в связном графе с неотрицательными весами с помощью алгоритма, принадлежащего Дейкстре.

Результатом работы этой процедуры является дерево кратчайших путей с корнем V0. ---- Входные и выходные переменные ---

A(I,J)

длина ребра, соединяющего вершины I и J. Если ребро отсутствует, то A(I,J)=10000 (произвольному большому числу).

V0 начальная вершина.

W конечная вершина.

N вершины в графе пронумерованы 1,...,N.

FROM(I)

TU(I)

содержит I-е ребро в дереве кратчайших путей от вершины FROM(I) к вершине TU(I)

LENGTH(I) длины LENGTH(I).

EDGES

число ребер в дереве кратчайших путей на данный момент.

--- Внутренние переменные ---

Библиотека ГГТУ им. П.О. Сухого

DIST(I) кратчайшее расстояние от UNDET(I) до частичного дерева кратчайших путей.

NEXT очередная вершина, добавляемая к дереву кратчайших путей.

NUMUN число неопределенных вершин.

UNDET(I) список неопределенных вершин.

VERTEX(I) вершины частичного дерева кратчайших путей, лежащие на кратчайшем пути от UNDET(I) до V0. }

```
Label stpoint;
Var dist,undet,vertex: array[1..n] of Integer;
   next,numun,i,j,k,l,jk: Integer;
Begin
 edges:=0; next:=v0; numun:=n-1;
 For i:=1 to n do
  Begin  undet[i]:=i; dist[i]:=a[v0,i]; vertex[i]:=v0 End;
 undet[v0]:=n; dist[v0]:=dist[n];
 goto stpoint;
 Repeat
{ Исключение вновь определенной вершины из списка неопределенных}
  dist[k]:=dist[numun]; undet[k]:=undet[numun];
  vertex[k]:=vertex[numun];
{ Остались ли неопределеные вершины ? }
  dec(numun);
{ Обновление кратчайшего расстояния до всех неопределенных вершин }
  For i:=1 to numun do
  Begin   j:=undet[i]; jk:=l+a[next,j];
     If dist[i] > jk Then Begin  vertex[i]:=next; dist[i]:=jk  End
  End;
stpoint:{Запоминание кратчайшего расст.до неопределенной вершины}
  k:=1; l:=dist[1];
  For i:=1 to numun do
   If dist[i] < l Then Begin  l:=dist[i]; k:=i  End;
{ Добавление ребра к дереву кратчайших путей }
  inc(edges);  from[edges]:=vertex[k]; tu[edges]:=undet[k];
  length[edges]:=l; next:=undet[k]
```


Библиотека ГГТУ им.П.О.Сухого

```
 Until next = w        { Достигли ли мы w }
End;
Procedure showway;
{ Эта процедура выводит на экран дисплея кратчайшее расстояние между
 вершинами V0 и W взвешенного графа, определенное процедурой dijkst }
Var i: Integer;
Begin
 writeln;  writeln('Кратчайшее расстояние между');
 writeln('узлами ',v0,' и ',w,' равно ',length[edges])
End;
{ Основная программа }
Begin
 adjinit;  printmat;   v0:=1;w:=5;
 dijkst;   showway;    readln
End.
```

Библиотека ГГТУ им. П.О. Сухого

7.1.3. **АЛГОРИТМ ФЛОЙДА**

Дано: непустой взвешенный граф $G=(V, E)$ с произвольными весами ребер (дуг). Требуется найти длины кратчайших путей между всеми парами вершин графа, если в графе нет циклов (контуров) отрицательной суммарной длины, либо обнаружить наличие таких контуров.

Инициализация:

1. Построим матрицу D^0 размерности $|V|$ x $|V|$, элементы которой определяются по правилу:

   1. $d_{ii}^0 = 0$;
   2. $d_{ij}^0 =$ Weight(v_i, v_j), где $i<>j$, если в графе существует ребро (дуга) (v_i, v_j);
   3. $d_{ij}^0 =$ бесконечность , где $i<>j$, если нет ребра (дуги) (v_i, v_j).

   2. $m:=0$.

Основная часть:

1. Построим матрицу D^{m+1} по D^m, вычисляя ее элементы следующим образом:
$d_{ij}^{m+1}=\min\{d_{ij}^m, \quad d_{i(m+1)}^m \quad + \quad d_{(m+1)j}^m\}$, где $i<>j$; $d_{ii}^{m+1}=0$ (*). Если $d_{im}^m + d_{mi}^m < 0$ для какого-то i, то в графе существует цикл (контур) отрицательной длины, проходящий через вершину v_i; ВЫХОД.

2. $m:=m+1$; если $m<|V|$, то повторяем шаг (1), иначе элементы последней построенной матрицы $D^{|V|}$ равны длинам кратчайших путей между соответствующими вершинами; ВЫХОД.

КОНЕЦ

Если требует найти сами пути, то перед началом работы алгоритма построим матрицу P с начальными значениями элементов $p_{ij}=i$. Каждый раз, когда на шаге (1) значение d_{ij}^{m+1} будет уменьшаться в соответствии с (*) (т.е. когда $d_{i(m+1)}^m + d_{(m+1)j}^m<d_{ij}^m$), выполним присваивание $p_{ij}:=p_{(m+1)j}$. В конце работы алгоритма матрица P будет определять кратчайшие пути между всеми парами вершин: значение p_{ij} будет равно номеру предпоследней вершины в пути между i и j (либо $p_{ij}=i$, если путь не существует).

Библиотека ГГТУ им. П.О.Сухого

Примечание: если граф - неориентированный, то все матрицы D^m являются симметричными, поэтому достаточно вычислять элементы, находящиеся только выше (либо только ниже) главной диагонали.

Текст программы

```
void Floid()
{
	for (int k = 0; k < n; k++)
	{
		for(int i = 0; i < n; i++)
		{
			for(int j = 0; j < n; j++)
			{
				if(mat[i][j] > mat[i][k]+mat[k][j])
				    mat[i][j] = mat[i][k]+mat[k][j];
			}
		}
	}
}
```

7.1.4. Алгоритм Дейкстры

Алгори́тм Де́йкстры — алгоритм на графах, изобретенный Э. Дейкстрой. Находит кратчайшее расстояние от одной из вершин графа до всех остальных. Алгоритм работает только для графов без рёбер отрицательного веса. Алгоритм широко применяется в программировании и технологиях, например, его использует протокол OSPF для устранения кольцевых маршрутов. Известен также под названием **кратчайший путь — первый** Дан простой взвешенный граф *G*(*V*,*E*) без петель и дуг отрицательного веса. Найти кратчайшие пути от некоторой вершины *a* графа *G* до всех остальных вершин этого графа.

Библиотека ГГТУ им.П.О.Сухого

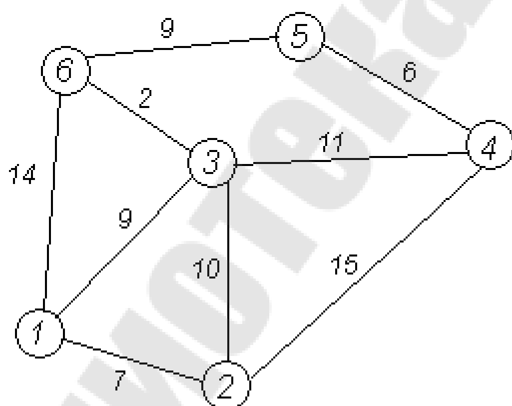
Каждой вершине из V сопоставим метку — минимальное известное расстояние от этой вершины до a. Алгоритм работает пошагово — на каждом шаге он «посещает» одну вершину и пытается уменьшать метки. Работа алгоритма завершается, когда все вершины посещены.

Инициализация. Метка самой вершины a полагается равной 0, метки остальных вершин — бесконечности. Это отражает то, что расстояния от a до других вершин пока неизвестны. Все вершины графа помечаются как непосещенные.

Шаг алгоритма. Если все вершины посещены, алгоритм завершается. В противном случае из еще не посещенных вершин выбирается вершина u, имеющая минимальную метку. Мы рассматриваем всевозможные маршруты, в которых u является предпоследним пунктом. Вершины, соединенные с вершиной u ребрами, назовем соседями этой вершины. Для каждого соседа рассмотрим новую длину пути, равную сумме текущей метки u и длины ребра, соединяющего u с этим соседом. Если полученная длина меньше метки соседа, заменим метку этой длиной. Рассмотрев всех соседей, пометим вершину u как посещенную и повторим шаг.

Пример

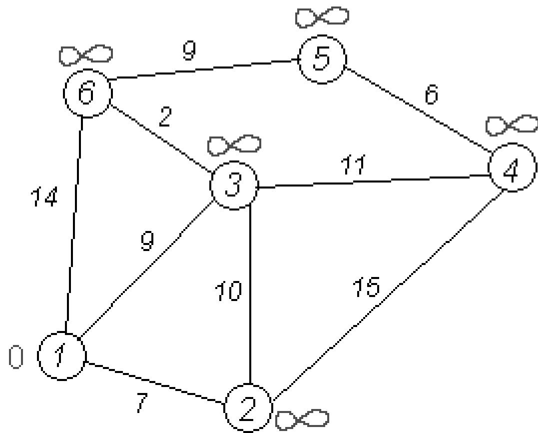
Рассмотрим выполнение алгоритма на примере графа, показанного на рисунке. Пусть требуется найти расстояния от 1-й вершины до всех остальных.



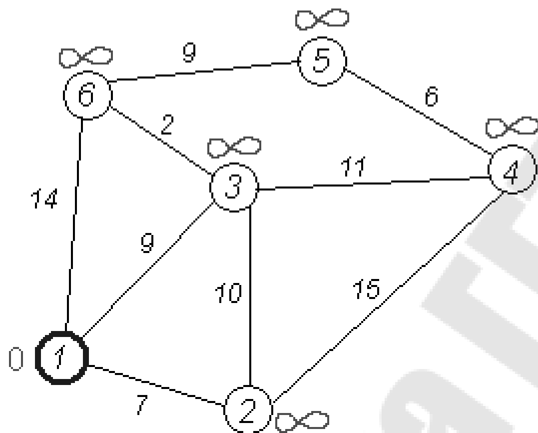
Кружками обозначены вершины, линиями — пути между ними (ребра графа). В кружках обозначены номера вершин, над ребрами обозначена их «цена» — длина пути. Рядом с каждой вершиной

Библиотека БГТУ им. П.О. Сухого

красным обозначена метка — длина кратчайшего пути в эту вершину из вершины 1.

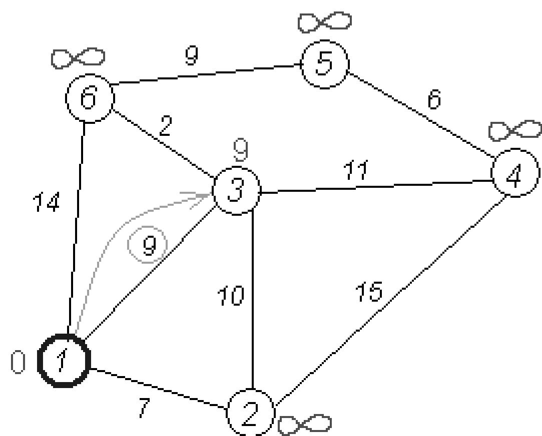


Первый шаг. Рассмотрим шаг алгоритма Дейкстры для нашего примера. Минимальную метку имеет вершина 1. Ее соседями являются вершины 2, 3 и 6.

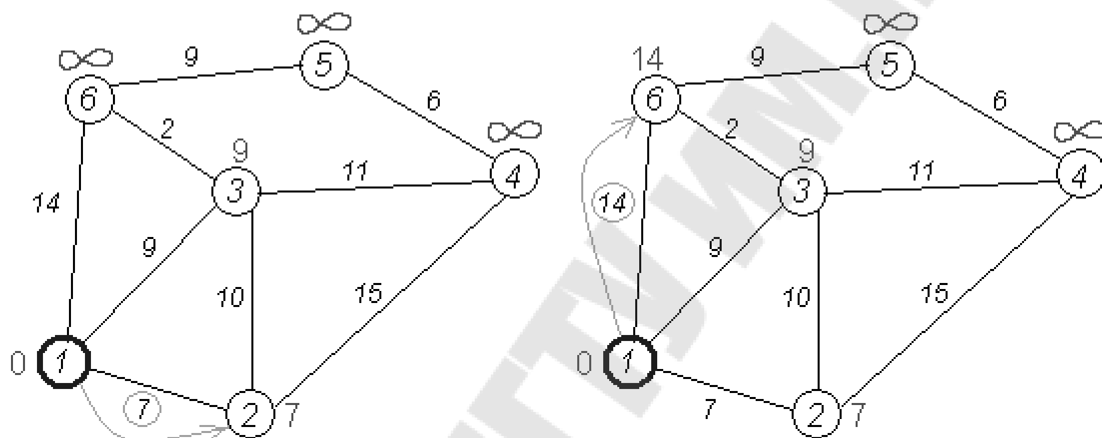


Первый по очереди сосед вершины 1 — вершина 2, потому что длина пути до нее минимальна. Длина пути в нее через вершину 1 равна кратчайшему расстоянию до вершины 1 + длина ребра, идущего из 1 в 2, то есть 0 + 7 = 7. Это меньше текущей метки вершины 2, поэтому новая метка 2-й вершины равна 7. На графике изначально рассмотрена вершина №3.

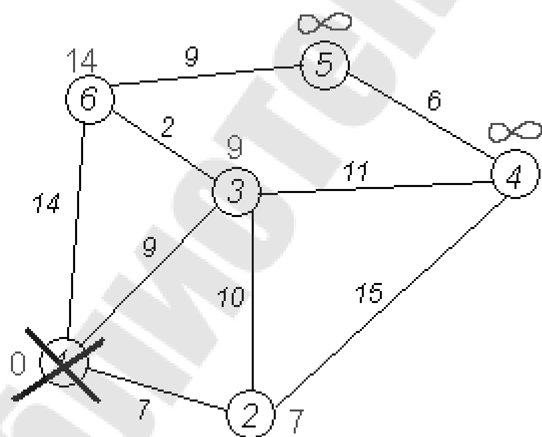
Библиотека ГГТУ им.П.О.Сухого



Аналогичную операцию проделываем с двумя другими соседями 1-й вершины — 3-й и 6-й.

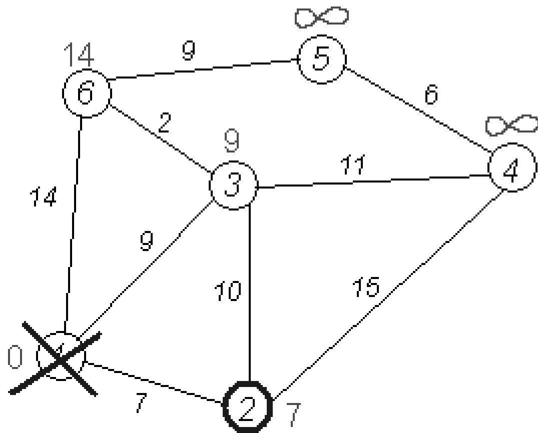


Все соседи вершины 1 проверены. Текущее минимальное расстояние до вершины 1 считается окончательным и пересмотру не подлежит (то, что это действительно так, впервые доказал Дейкстра). Вычеркнем её из графа, чтобы отметить, что эта вершина посещена.



Библиотека ГГТУ им. П.О.Сухого

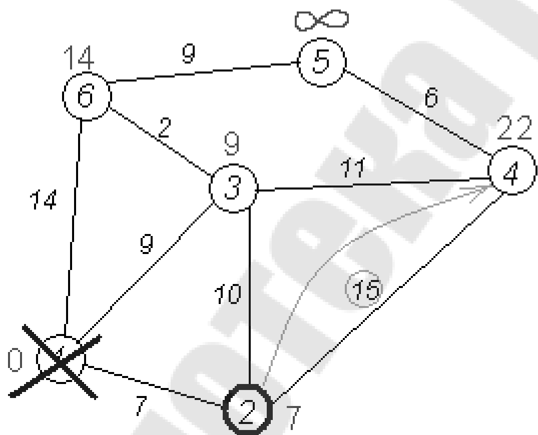
Второй шаг. Шаг алгоритма повторяется. Снова находим «ближайшую» из непосещенных вершин. Это вершина 2 с меткой 7.



Снова пытаемся уменьшить метки соседей выбранной вершины, пытаясь пройти в них через 2-ю. Соседями вершины 2 являются 1, 3, 4.

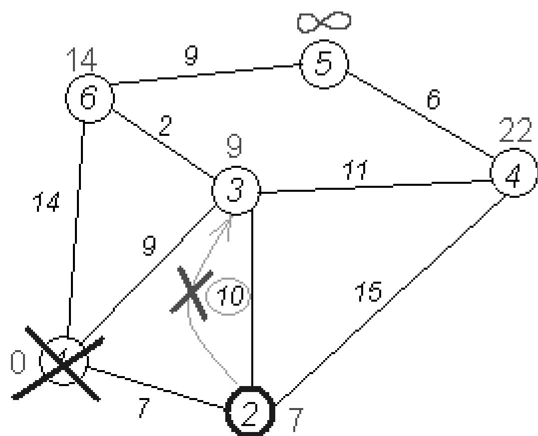
Первый (по порядку) сосед вершины 2 — вершина 1. Но она уже посещена, поэтому с 1-й вершиной ничего не делаем.

Следующий сосед вершины 2 — вершина 4/*3*/. Если идти в неё через 2-ю, то длина такого пути будет = кратчайшее расстояние до 2 + расстояние между вершинами 2 и 4 = 7 + 15 = 22. Поскольку 22<∞, устанавливаем метку вершины 4 равной 22.

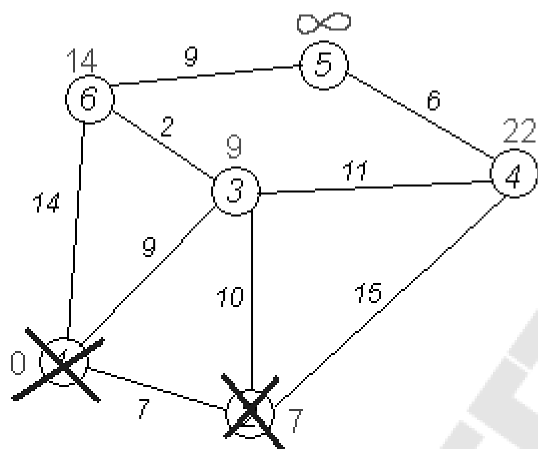


Ещё один сосед вершины 2 — вершина 3. Если идти в неё через 2, то длина такого пути будет = 7 + 10 = 17. Но текущая метка третьей вершины равна 9<17, поэтому метка не меняется.

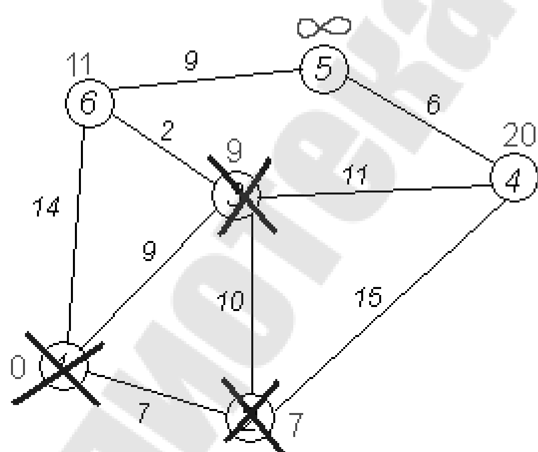
Библиотека ГГТУ им. П.О. Сухого



Все соседи вершины 2 просмотрены, замораживаем расстояние до неё и помечаем ее как посещенную.

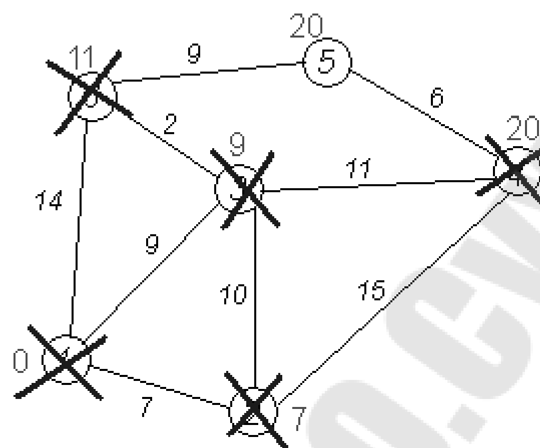


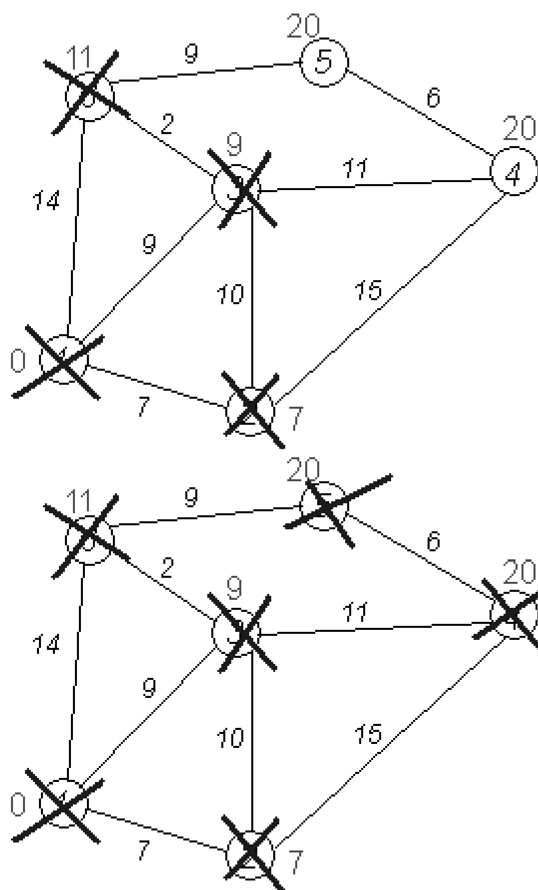
Третий шаг. Повторяем шаг алгоритма, выбрав вершину 3. После ее «обработки» получим такие результаты:



Дальнейшие шаги. Повторяем шаг алгоритма для оставшихся вершин (Это будут по порядку 6, 4 и 5).

Библиотека ГГТУ им. П.О. Сухого





Завершение выполнения алгоритма. Алгоритм заканчивает работу, когда вычеркнуты все вершины. Результат его работы виден на последнем рисунке: кратчайший путь от вершины 1 до 2-й составляет 7, до 3-й — 9, до 4-й — 20, до 5-й — 20, до 6-й — 11.

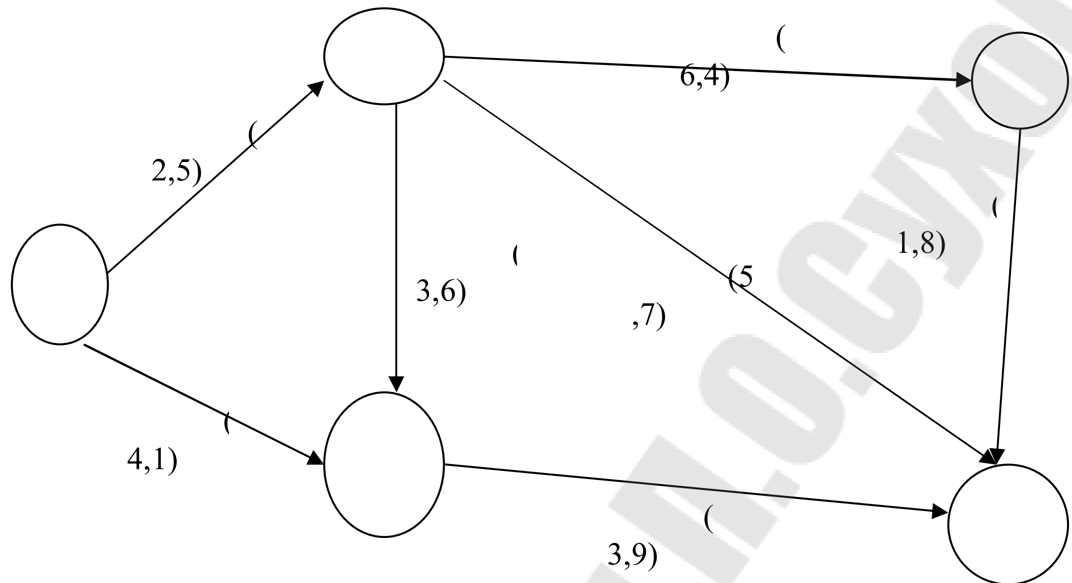
Сложность алгоритма $O(n^2 + m)$

7.1.5. Максимальный поток

Будем представлять, что по ребрам (I,j) где i<j направлен груз из истока I в сток S.

Максимальное количество которое за единицу времени может пропустить ребро (I,j) называется пропускной способностью ребра r(I,j) . В общем случае r(I,j)<>r(j,i). Если вершины l и k не соединены то r(l,k)=r(k,l)=0.

Библиотека ГГТУ им.П.О.Сухого



Первое число это пропускная способность туда второе обратно.

Пропускные способности ребер можно задать матрицей

	1	2	3	4	5
1	0	2	4	0	0
2	5	0	3	6	5
3	1	6	0	0	3
4	0	4	0	0	1
5	0	7	9	8	0

Количество x(i,j) вещества проходящего в единицу времени называется потоком по ребру (i,j). Совокупность потоков по всем ребрам сети (I,j) называется потоком по сети.

Поток по каждому ребру не превышает его пропускную способность.

X(I,j)<=r(I,j) (для всех I,j=1,..n)(1)

Количество вещества притекающего в вершину равно количеству вещества вытекающего из нее.

Если из вершины i в вершину j направляется поток x(i,j) то из j в I поток -x(I,j)(2)

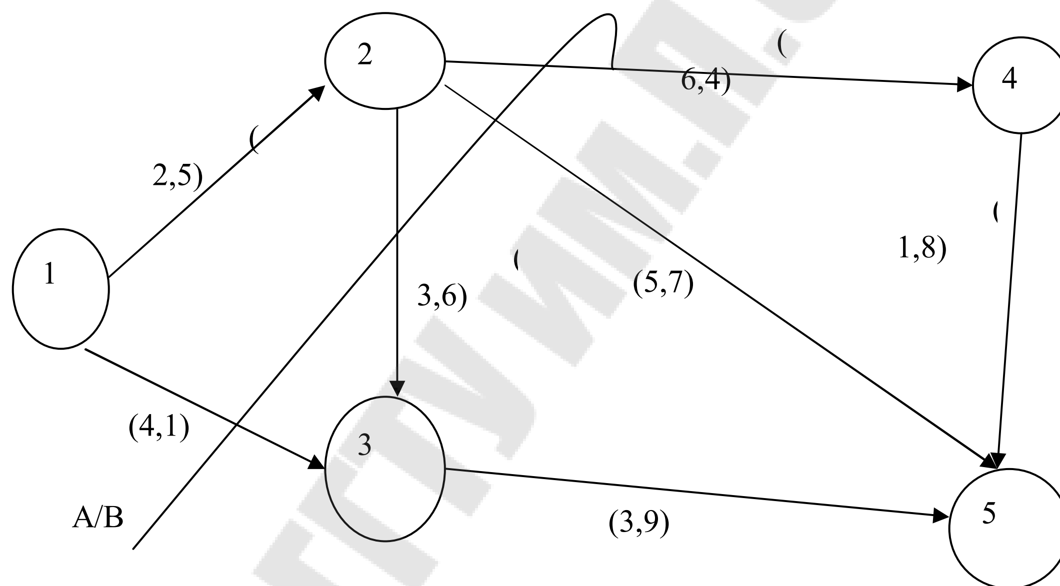
Общее количество сырья исходящее из истока равно общему количеству сырья входящего в сток.

Библиотека ГГТУ им. П.О.Сухого

$f = \sum_j x_{Ij} = \sum_i x_{iS}$ (3) Функцию f называют мощностью потока на сети.

Возникает задача ЛП найти совокупность X={x(I,j)} потоков x(I,j) по всем ребрам (I,j) сети которая удовлетворяет (1-2) и максимизирует (3)

Предположим что дана некоторая сеть . Разобъем множество вершин этой сети на на 2 непересекающихся подмножества, так что бы исток попал в множество А а сток в множество В.



A={1,2} B={3,4,5}

Совокупность ребер(I,j) начальные вершины которх принадлежат множеству А а конечные множеству В образуют разрез сети его обозначают A/B.

{1,3} {2,3} {2,4} {2,5}.

Если на сети задан некоторый поток то X={x(I,j)} то ребро (I,j) называют насыщенным, если поток x(I,j) совпадает с его пропускной способностью и ненасыщенным если поток меньше пропускной способности.

Величина $R(A/B) = \sum_{i \in A} \sum_{j \in B} r_{ij}$ называется пропускной способностью разреза

Библиотека ГГТУ им.П.О.Сухого

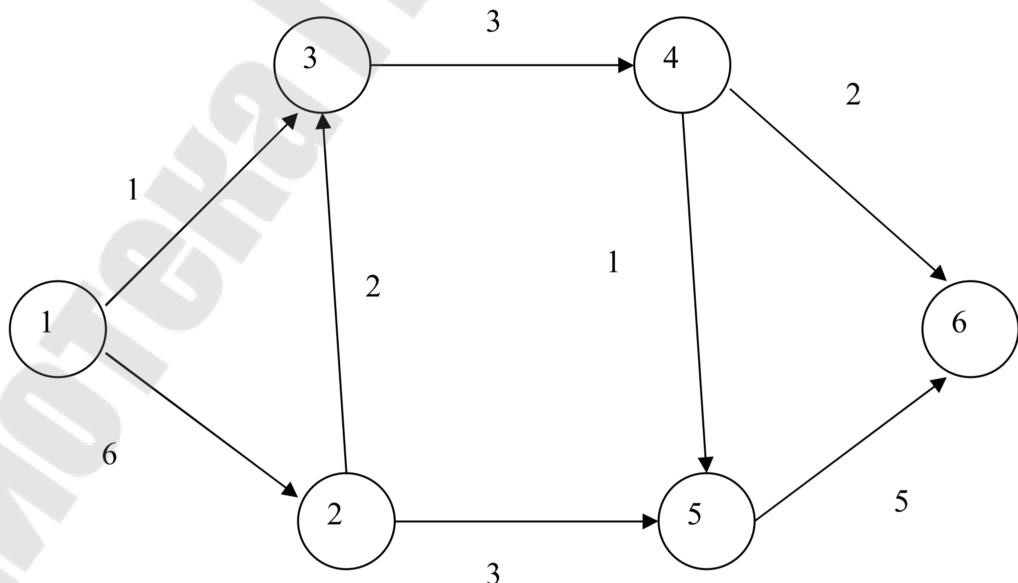
Величина $X(A/B)=\sum_{i\in A}\sum_{j\in B}x_{ij}$ называется потоком через разрез

На любой сети максимальная величина потока из истока I в сток S равна минимальной пропускной способности разреза , отделяющего I от S.

Алгоритм построения.

1. Построить некоторый начальный поток X0
2. Организовать процедуру составления подмножества вершин A, достижимых из истока по ненасыщенным ребрам. Если в этом процессе сток S не поподает в множество A то поток максимальный. Если S попал в A то перейти к пункту 3.
3. Выделить путь из I в S состоящий из ненасыщенных ребер , и увеличить поток x(I,j) по каждому ребру этого пути на $\Delta=\min(r(i,j)-x(i,j))$, где минимум берется по ребрам (I,j) упомянутым в пути. Тем самым будет построен новый поток X1. После чего возвратиться к пункту 2.

На сети сформировать поток максимальной мощности направленный из I в S.(из 1 в 6) при условии что пропускные способности ребер в обоих направлениях одинаковы



Библиотека ГГТУ им. П.О. Сухого

Решение:

Необходимо сформировать начальный поток X0.

По пути 1-3-4-6 пропустим 1 единицу груза

По пути 1-2-3-4-6 пропустим 1 единицу груза

По пути 1-2-5-6 пропустим 3 еден груза

В результате потока x12=4 x13=1 x23=1 x25=3 x34=2 x46=2 x56=3 по остальным 0.

Матрица пропускных способностей ребер R

	1	2	3	4	5	6
1	0	6	1	0	0	0
2	6	0	2	0	3	0
3	1	2	0	3	0	0
4	0	0	3	0	1	2
5	0	3	0	1	0	5
6	0	0	0	2	5	0

Матрица построенного потока X0

	1	2	3	4	5	6
1	0	4	1	0	0	0
2	-4	0	1	0	3	0
3	-1	-1	0	2	0	0
4	0	0	-2	0	0	2
5	0	-3	0	0	0	3
6	0	0	0	-2	-3	0

Построим матрицу R-X0 элементы которой позволяют судить о насыщенности ребер.

Насыщенным будет соответствовать 0 а ненасыщенным не 0.

R-X0

	1	2	3	4	5	6
1	0	2	0	0	0	0
2	10	0	1	0	3	0
3	2	3	0	1	0	0
4	0	0	5	0	1	0
5	0	6	0	1	0	2

Библиотека ГГТУ им. П.О.Сухого

6	0	0	0	4	8	0

Сформируем подмножество А вершин, в которое можно попасть из истока I , двигаясь только по ненасыщенным путям., а так же выделить эти пути и с их помощью увеличить мощность потока.

С этой целью просматриваем первую строку матрицы и выписываем номера вершин соответствующих не 0 элементам. Это и будут вершины в которые можно попасть из истока I. Далее для каждой вершины списка составляем аналогично свой список. При этом вершины встречающиеся в прежних списках повторно не выписываются. Если в этом процессе сток S не встретился поток максимален. Если встретился то мощность его можно увеличить.

I=1 S=6

1||2

2||3 3||4 4||5 5||6

Поскольку сток попал в список поток не оптимален. Значит существует путь из I в S по ненасыщенным ребрам.

Построение ненасыщенного пути начинается с последнего ребра этого пути . Им будет вершина ребро (i(n-1),s) где (i(n-1)) вершина в список которой попал сток S.

Далее выписываются ребра (i(n-1),i(n-2)) и так далее до тех пор пока не встретится (I,i1).

В нашем примере таким путем будет путь проходящий через вершины 1 2 3 4 5 6.

После выделения ненасыщенного пути найдем величину $\Delta = \min(rij - xij)$ из матрицы R-X0 на которую надо увеличить поток по каждому ребру. В нашем примере по ребру (1-2) можно увеличить на 2 по остальным на 1. Так что увеличение потока повсему пути составляет 1.

Для получения матрицы нового потока X1 к соответствующей матрице X0 добавим значение 1 к соответствующим ребрам

Новый поток надо исследовать на оптимальность.

Матрица потока X1

	1	2	3	4	5	6
1	0	5	1	0	0	0
2	-5	0	2	0	3	0
3	-1	-2	0	3	0	0
4	0	0	-3	0	1	2
5	0	-3	0	0	0	4

Библиотека ГГТУ им. П.О.Сухого

6	0	0	0	-2	-4	0

Матрица R-X1

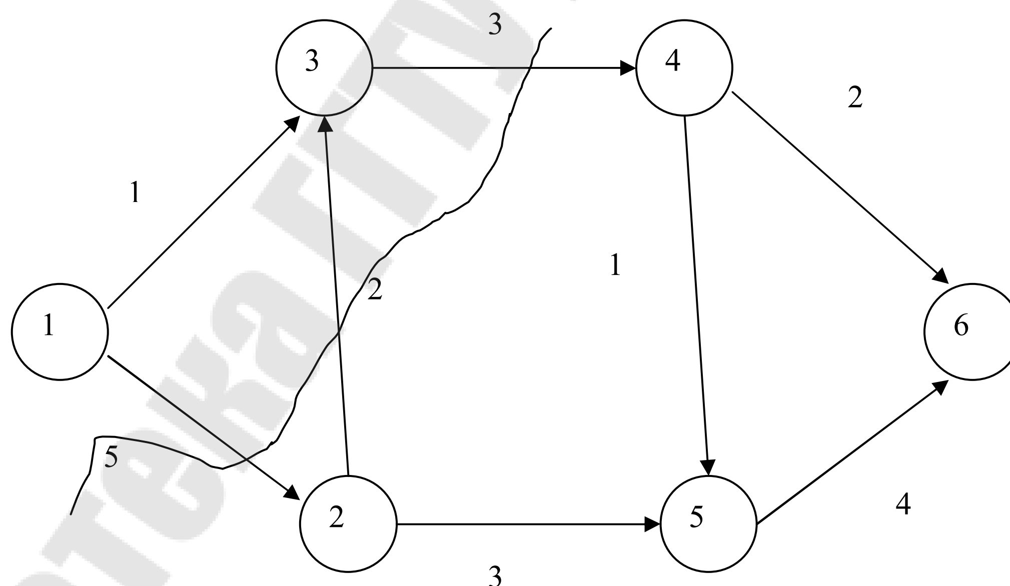
	1	2	3	4	5	6
1	1	0	0	0	0	0
2	11	0	0	0	0	0
3	2	4	0	0	0	0
4	0	0	6	0	0	0
5	0	6	0	2	0	1
6	0	0	0	4	9	0

Составим список вершин достижимых из истока по ненасыщенным путям

1||3 3||

Сток 6 не попал в список вершин. Значит X1 это максимальный поток.

Остается нанести его на сеть с указанием направлений



Выделим подмножества A и B на которые оказалось разбито множество всех вершин

A{1,3} B{2,4,5,6}

Выпишем ребра образующие разрез (1,2) (3,2) (3,4)

Сумма пропускных способностей по всем ребрам разреза равна максимальной пропускной способности сети. 5-2+3=1

Библиотека ГГТУ им. П.О.Сухого

8. Деревья
8.1. Основные определения

Дерево - это граф, который характеризуется следующими свойствами:

1. Существует единственный элемент (узел или вершина), на который не ссылается никакой другой элемент - и который называется ***корнем*** (рис. 8.8, 8.9 - A,G,M - корни).

2. Начиная с корня и следуя по определенной цепочке указателей, содержащихся в элементах, можно осуществить доступ к любому элементу структуры.

3. На каждый элемент, кроме корня, имеется единственная ссылка, т.е. каждый элемент адресуется единственным указателем.

Название "дерево" проистекает из логической эквивалентности древовидной структуры абстрактному дереву в теории графов. Линия связи между парой узлов дерева называется обычно ***ветвью***. Те узлы, которые не ссылаются ни на какие другие узлы дерева, называются ***листьями*** (или ***терминальными вершинами***) (рис. 6.8, 6.9 - b,k,l,h - листья). Узел, не являющийся листом или корнем, считается промежуточным или узлом ветвления (нетерминальной или внутренней вершиной).

Для ориентированного графа число ребер, исходящих из некоторой начальной вершины V, называется ***полустепенью исхода*** этой вершины. Число ребер, для которых вершина V является конечной, называется ***полустепенью захода*** вершины V, а сумма полустепеней исхода и захода вершины V называется ***полной сткпенью*** этой вершины.

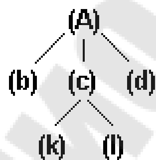


рис. 8.1. Дерево

Библиотека ГГТУ им.П.О.Сухого

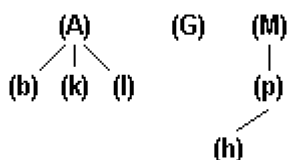


рис. 8.2. Лес

Ниже будет представлен важный класс орграфов - ориентированные деревья - и соответствующая им терминология. Деревья нужны для описания любой структуры с иерархией. Традиционные примеры таких структур: генеалогические деревья, десятичная классификация книг в библиотеках, иерархия должностей в организации, алгебраическое выражение, включающее операции, для которых предписаны определенные правила приоритета.

Ориентированное дерево - это такой ациклический орграф (ориентированный граф), у которого одна вершина, называемая корнем, имеет полустепень захода, равную 0, а остальные - полустепени захода, равные 1. Ориентированное дерево должно иметь по крайней мере одну вершину. Изолированная вершина также представляет собой ориентированное дерево.

Вершина ориентированного дерева, полустепень исхода которой равна нулю, называется ***концевой*** (***висячей***) вершиной или ***листом***; все остальные вершины дерева называют вершинами ветвления. Длина пути от корня до некоторой вершины называется ***уровнем*** (***номером яруса***) этой вершины. Уровень корня ориентированного дерева равен нулю, а уровень любой другой вершины равен расстоянию (т.е. модулю разности номеров уровней вершин) между этой вершиной и корнем. Ориентированное дерево является ациклическим графом, все пути в нем элементарны.

Во многих приложениях относительный порядок следования вершин на каждом отдельном ярусе имеет определенное значение. При представлении дерева в ЭВМ такой порядок вводится автоматически, даже если он сам по себе произволен. Порядок следования вершин на некотором ярусе можно легко ввести, помечая одну вершину как первую, другую - как вторую и т.д. Вместо упорядочивания вершин можно задавать порядок на ребрах. Если в

Библиотека БНТУ: ... сухого

ориентированном дереве на каждом ярусе задан порядок следования вершин, то такое дерево называется ***упорядоченным деревом***.

Введем еще некоторые понятия, связанные с деревьями. На рис.8.3 показано дерево:

Узел X называется ***предком*** (или ***отцом***), а узлы Y и Z называются ***наследниками*** (или ***сыновьями***). Их, соответственно, между собой называют ***братьями***. Причем левый сын является ***старшим сыном***, а правый - ***младшим***. Число поддеревьев данной вершины называется ***степенью*** этой вершины. (В данном примере X имеет 2 поддерева, следовательно, ***степень*** вершины X равна 2).

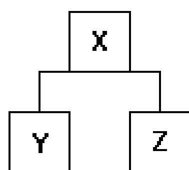


рис.8.3. Дерево

Если из дерева убрать корень и ребра, соединяющие корень с вершинами первого яруса, то получится некоторое множество несвязанных деревьев. Множество несвязанных деревьев называется ***лесом*** (рис. 8.2).

8.2. Логическое представление и изображение деревьев

Имеется ряд способов графического изображения деревьев. Первый способ заключается в использовании для изображения поддеревьев известного метода ***диаграмм Венна***, второй - метода ***вкладывающихся друг в друга скобок***, третий способ - это способ, применяемый при составлении ***оглавлений книг***. Последний способ, базирующийся на формате с нумерацией уровней, сходен с методами, используемыми в языках программирования. При применении этого формата каждой вершине приписывается числовой номер, который должен быть меньше номеров, приписанных корневым вершинам присоединенных к ней поддеревьев. Отметим, что корневые вершины всех поддереьев данной вершины должны иметь один и тот же номер.

Библиотека ГГТУ им. П.О. Сухого

МЕТОД ВЛОЖЕННЫХ СКОБОК

(V0(V1(V2(V5)(V6))(V3)(V4))(V7(V8)(V9(V10))))

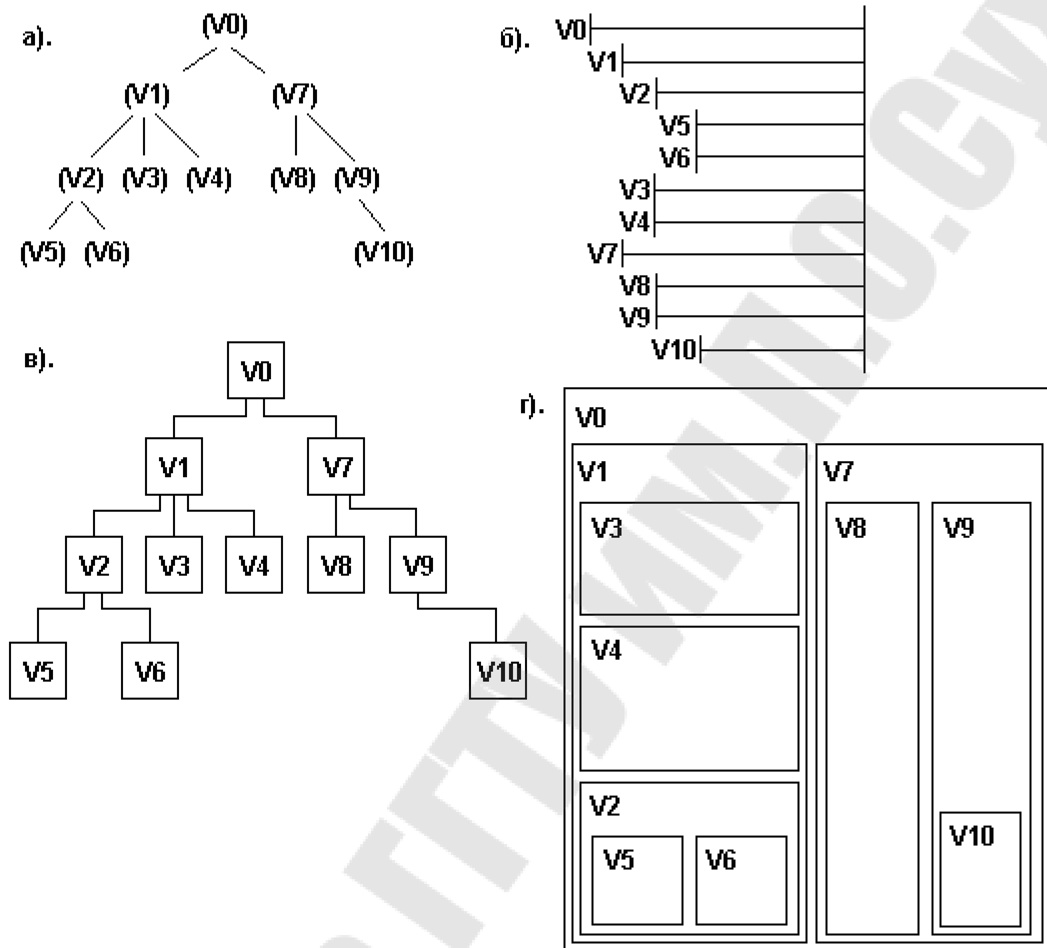


Рис.8.4. Представление дерева: а)- исходное дерево, б)- оглавление книг, в)- граф, г)- диаграмма Венна

Все эти представления демонстрируют одну и ту же структуру и поэтому эквивалентны. С помощью графа можно наглядно представить разветвляющиеся связи, которые по понятным причинам привели к общеупотребительному термину "дерево".

8.3. Бинарные деревья

Существуют m-арные деревья, т.е. такие деревья у которых полустепень исхода каждой вершины меньше или равна m (где m может быть равно 0,1,2,3 и т.д.). Если полустепень исхода каждой вершины в точности равна либо m, либо нулю, то такое дерево называется ***полным m-арным деревом***.

Библиотека ГГТУ им. П.О. Сухого

При m=2 такие деревья называются соответственно ***бинарными***, или ***полными бинарными***.

На рисунке 8.5(a) изображено бинарное дерево, 8.5(b)- полное бинарное дерево, а на 8.5(c) показаны все четыре возможных расположения сыновей некоторой вершины бинарного дерева.

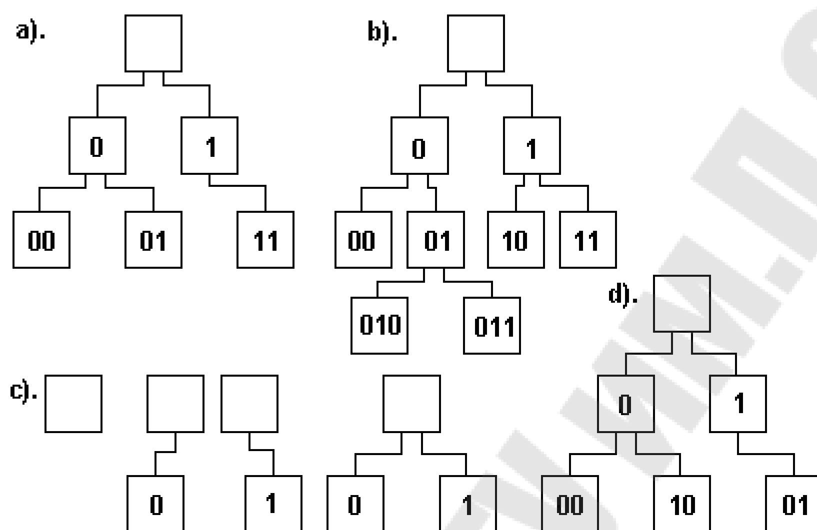


Рис. 8.5. Изображения бинарных деревьев

Бинарные деревья, изображенные на рис.8.5(a) и 8.5(d), представляют собой разные позиционные деревья, хотя они не являются разными упорядоченными деревьями.

В позиционном бинарном дереве каждая вершина представлена единственным образом посредством строки символов над алфавитом {0,1}, при этом корень характеризуется пустой строкой. Любой сын вершины "u" характеризуется строкой, префикс (начальная часть) которой является строкой, характеризующей "u".

Примером бинарного дерева является фамильное дерево с отцом и матерью человека в качестве его потомков. Еще один пример - это арифметическое выражение с двухместными операциями, где каждая операция представляет собой ветвящийся узел с операндами в качестве поддеревьев.

Библиотека ГГТУ им. П.О. Сухого

Представить m-арное дерево в памяти ЭВМ сложно, т.к. каждый элемент дерева должен содержать столько указателей, сколько ребер выходит из узла (при m-3,4.5.6... соответствует 3,4,5,6... указателей). Это приведет к повышенному расходу памяти ЭВМ, разнообразию исходных элементов и усложнит алгоритмы обработки дерева. Поэтому m-арные деревья, лес необходимо привести к бинарным для экономии памяти и упрощения алгоритмов. Все узлы бинарного дерева представляются в памяти ЭВМ однотипными элементами с двумя указателями, кроме того, операции над двоичными деревьями выполняются просто и эффективно.

8.4. Представление любого дерева, леса бинарными деревьями

Дерево и лес любого вида можно преобразовать единственным образом в эквивалентное бинарное дерево.

Правило построения бинарного дерева из любого дерева:

1. В каждом узле оставить только ветвь к старшему сыну (вертикальное соединение).
2. Соединить горизонтальными ребрами всех братьев одного отца.
3. Таким образом перестроить дерево по правилу:

   левый сын - вершина, расположенная под данной;

   правый сын - вершина, расположенная справа от данной (т.е. на одном ярусе с ней).
4. Развернуть дерево таким образом, чтобы все вертикальные ветви отображали левых сыновей, а горизонтальные - правых.

В результате преобразования любого дерева в бинарное получается дерево в виде левого поддерева, подвешенного к корневой вершине.

В процессе преобразования правый указатель каждого узла бинарного дерева будет указывать на соседа по уровню. Если такового нет, то правый указатель NIL. Левый указатель будет указывать на вершину следующего уровня. Если таковой нет, то указатель устанавливается на NIL.

Библиотека БГУИР

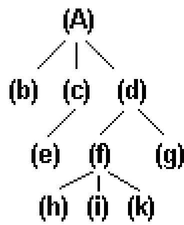


Рис.8.6. Исходное дерево

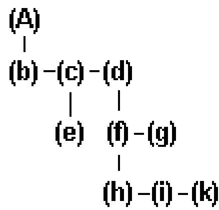


Рис.8.7 Промежуточный результат перестройки дерева

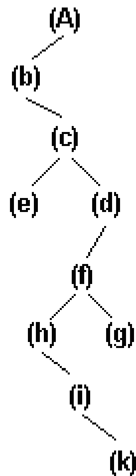


Рис. 8.8. Представление дерева в виде бинарного

Описанный выше метод представления произвольных упорядоченных деревьев посредством бинарных деревьев можно обобщить на представление произвольного упорядоченного леса.

Правило построения бинарного дерева из леса: корни всех поддеревьев леса соединить горизонтальными связями. В полученном дереве узлы в данном примере будут располагаться на трех уровнях. Далее перестраивать по ранее рассмотренному плану: в начале поддерево с корнем А, затем В и затем Н. В результате преобразования упорядоченного леса в бинарное дерево получается полное бинарное дерево с левым и правым поддеревом.

Библиотека ГГТУ им.П.О.Сухого

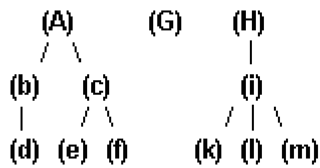


рис.8.9. Упорядоченный лес

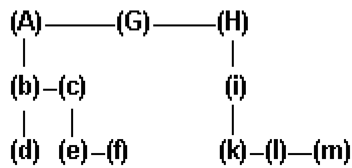


Рис.8.10. Промежуточный результат перестройки леса

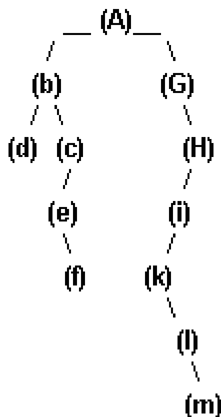


Рис. 8.11. Представление леса в виде 2-го дерева

В результате преобразования упорядоченного леса в бинарное дерево получается полное бинарное дерево с левым и правым поддеревом.

8.5. Машинное представление деревьев в памяти ЭВМ

Деревья можно представлять с помощью ***связных списков*** и ***массивов*** (или последовательных списков).

Чаще всего используется связное представление деревьев, т.к. оно очень сильно напоминает логическое. Связное хранение состоит в том, что задается связь от отца к сыновьям, поэтому удобно узел представить в виде структуры:

LPTR	DATA	RPTR

Библиотека ГГТУ им. П.О.Сухого

где LPTR - указатель на левое поддерево, RPTR - указатель на правое поддерево, DATA - содержит информацию, связанную с вершиной.

Рассмотрим машинное представление бинарного дерева, изображенного на рис. 8.20.

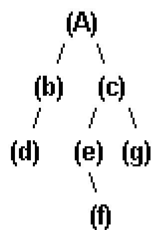


Рис. 8.12. Логическое представление дерева.

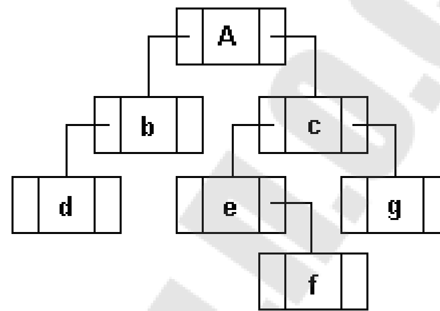
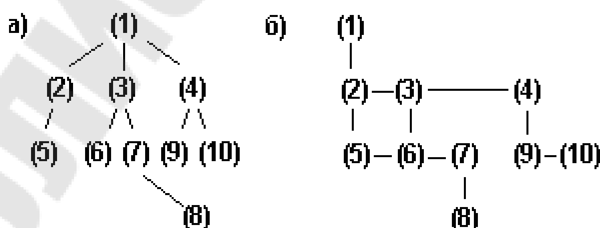


Рис. 8.13. Машинное связное представление дерева.

Рассмотрим ***последовательное представление деревьев***. Оно удобно и эффективно в случае, если древовидная структура в течение времени своего существования не подвергается значительным изменениям, за счет включения вершин, удаления вершин и т.д.

Выбор метода последовательного представления деревьев определяется также набором тех операций, которые должны быть выполнены над древовидными структурами. (Пример статистической древовидной структуры - пирамидальный метод сортировки). Простейший метод представления дерева в виде последовательной структуры заключается во введении вектора FATHER, задающего отбор для всех его вершин. Этот метод можно использовать также для представления леса. Недостаток метода - он не отображает упорядочения вершин дерева. Если в предыдущем примере поменять местами вершины 9 и 10, последовательное представление останется тем же.



Библиотека ГГТУ им. П.О. Сухого

Рис. 8.14. Диаграммы дерева: а) исходное б) перестройка в бинарное

Последовательное представление дерева, логическая диаграмма которого дана на рис. 8.14 , задается следующим образом:

i	1 2 3 4 5 6 7 8 9 10
FATHER [i]	0 1 1 1 2 3 3 7 4 4 ,

где ветви определяются как
{(FATHER[i],i)}, i = 2,3,...,10.

Вершины 2,3,4 являются сыновьями вершины 1, вершина 5 - сыном вершины 2, вершины 6,7 - сыновьями вершины 3, вершина 8 имеет отца вершина 7 и вершины 9 и 10 - сыновья вершины 4.

Числовые поля данных используются здесь, чтобы упростить представление дерева. Корневая вершина не имеет отца, поэтому вместо отца для нее задается нулевое значение.

Общее правило: если T обозначает индекс корневой вершины дерева, то FATHER[T] = 0.

Другой метод последовательного представления деревьев заключается в использовании физической смежности элементов машинной памяти вместо одного из полей LPTR или RPTR, например, способ опускания полей, т.е. чтобы вершины появлялись в нисходящем порядке. Дерево (рис.8.14 б), можно описать как:

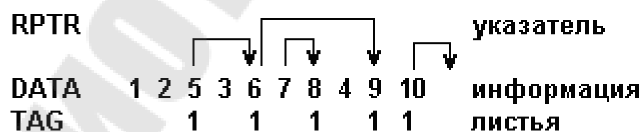


Рис. 8.15. Последовательное представление дерева методом опускания полей

где RPTR,DATA и TAG представляют векторы. В данном методе указатель LPTR не требуется, т.к. если бы он не был пуст, то указывал бы на вершину, стоящую непосредственно справа от данной. Вектор TAG - бинарный вектор, в котором единицы отмечают концевые вершины исходного дерева. При таком представлении имеются значительные потери памяти, т.к. свыше половины указателей RPTR оказываются пустыми. Эти пустые места можно использовать путем установки указателя RPTR каждой данной вершины на вершину, которая следует непосредственно за поддеревом, расположенном под ней. В таком представлении поле RPTR переименовывается в RANGE:

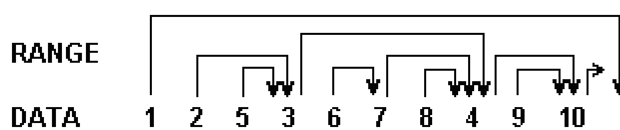


Рис. 8.16. Последовательное представление дерева с размещением вершин в возрастающем порядке

В этом случае поле TAG не требуется поскольку концевой узел определяется условием RANGE(P) = P + 1.

Третий метод состоит в представлении дерева общего вида на основе его восходящего обхода. Такое представление состоит из двух векторов: один вектор описывает все вершины дерева в восходящей последовательности, а второй - задает полустепени исхода этих вершин (см. рис.8.17). Восходящий метод представления удобен для вычисления функцией, заданных на определенных вершинах дерева (например, использование таких функций для генерации объектного кода по обратной польской записи некоторого выражения).

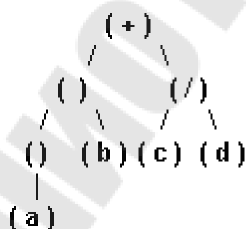


Рис. 8.17. Последовательное представление дерева на основе восходящего обхода

В заключении приведем два важных понятия.

Подобие бинарных деревьев - два дерева подобны, если они имеют одинаковую структуру (форму).

Эквивалентные бинарные деревья - два дерева эквивалентны, если они подобны, и если соответствующие вершины у них содержат одинаковую информацию.

8.6. Основные операции над деревьями

Над деревьями определены следующие основные операции, для которых приведены реализующие их программы.

1) Поиск узла с заданным ключом (Find).
2) Добавление нового узла (Dob).
3) Удаление узла (поддерева) (Udal).
4) Обход дерева в определенном порядке:

Нисходящий обход (процедура Preorder , рекурсивная процедура r_Preoder);

Смешанный обход (процедура Inorder, рекурсивная процедура r_Inorder);

Восходящий обход (процедура Postorder, рекурсивная процедура r_Postorder).

Приведенные ниже программы процедур и функций могут быть непосредственно использованы при решении индивидуальных задач. Кроме выше указанных процедур приведены следующие процедуры и функции:

процедура включения в стек при нисходящем обходе (Push_st);

функция извлечения из стека при нисходящем обходе (Pop_st);

процедура включения в стек при восходящем и смешанном обходе (S_Push);

функция извлечения из стека при восходящем и смешанном обходе (S_Pop).

Для прошитых деревьев:

функция нахождения сына данного узла (Inson);

Библиотека ГГТУ им. П.О.Сухого

функция нахождения отца данного узла (Inp);
процедура включения в дерево узла слева от данного (leftIn);

ПОИСК ЗАПИСИ В ДЕРЕВЕ(Find).

Нужная вершина в дереве ищется по ключу. Поиск в бинарном дереве осуществляется следующим образом.

Пусть построено некоторое дерево и требуется найти звено с ключом X. Сначала сравниваем с X ключ, находящийся в корне дерева. В случае равенства поиск закончен и нужно возвратить указатель на корень в качестве результата поиска. В противном случае переходим к рассмотрению вершины, которая находится слева внизу, если ключ X меньше только что рассмотренного, или справа внизу, если ключ X больше только что рассмотренного. Сравниваем ключ X с ключом, содержащимся в этой вершине, и т.д. Процесс завершается в одном из двух случаев:

1) найдена вершина, содержащая ключ, равный ключу X;
2) в дереве отсутствует вершина, к которой нужно перейти для выполнения очередного шага поиска.

В первом случае возвращается указатель на найденную вершину. Во втором - указатель на звено, где остановился поиск, (что удобно для построения дерева). Реализация функции Find приведена в программном примере 8.2.

```
 {===     Программный пример 8.2.  Поиск звена по ключу
=== }
 Function Find(k:KeyType;d:TreePtr;var rez:TreePtr):bollean;
 { где k - ключ, d - корень дерева, rez - результат }
 Var
    p,g: TreePtr;
      b: boolean;
 Begin
   b:=false;    p:=d;                    { ключ не найден }
   if d <> NIL  then
    repeat q: =p; if p^.key = k  then  b:=true  { ключ найден }
          else begin q:=p;              { указатель на отца }
                 if k < p^.key then p:=p^.left { поиск влево }
```


Библиотека ГГТУ им. П.О. Сухого

```
                    else p:=p^.right { поиск вправо}
          end;    until b or (p=NIL);
    Find:=b; rez:=q;
  End; { Find }
```

ДОБАВЛЕНИЕ НОВОГО УЗЛА (Dop).

Для включения записи в дерево прежде всего нужно найти в дереве ту вершину, к которой можно "подвести" (присоединить) новую вершину, соответствующую включаемой записи. При этом упорядоченность ключей должна сохраняться.

Алгоритм поиска нужной вершины, вообще говоря, тот же самый, что и при поиске вершины с заданным ключом. Эта вершина будет найдена в тот момент, когда в качестве очередного указателя, определяющего ветвь дерева, в которой надо продолжить поиск, окажется указатель NIL (случай 2 функции Find). Тогда процедура вставки записывается так, как в программном примере 6.3.

```
{=== Программный пример 6.3. Добавление звена ===}
 Procedure Dob (k:KeyType; var d:TreePtr; zap:data);
 { k - ключ, d - узел дерева, zap - запись }
 Var
   r,s: TreePtr;
    t: DataPtr;
 Begin
   if not Find(k,d,r) then
    begin        (* Занесение в новое звено текста записи *)
      new(t); t^:=zap; new(s);  s^.key:=k;
      s^.ssil:=t; s^.left:=NIL; s^.right:=NIL;
       if d = NIL   then d:=s      (* Вставка нового звена *)
                  else if k < r^.key
                         then r^.left:=s
                         else r^.right:=s;
          end;    End; { Dop }
```

ОБХОД ДЕРЕВА.

Во многих задачах, связанных с деревьями, требуется осуществить систематический просмотр всех его узлов в

Библиотека ГГТУ им.П.О.Сухого

определенном порядке. Такой просмотр называется прохождением или обходом дерева.

Бинарное дерево можно обходить тремя основными способами: нисходящим, смешанным и восходящим (возможны также обратный нисходящий, обратный смешанный и обратный восходящий обходы). Принятые выше названия методов обхода связаны с временем обработки корневой вершины: До того как обработаны оба ее поддерева (Preorder), после того как обработано левое поддерево, но до того как обработано правое (Inorder), после того как обработаны оба поддерева (Postorder). Используемые в переводе названия методов отражают направление обхода в дереве: от корневой вершины вниз к листьям - нисходящий обход; от листьев вверх к корню - восходящий обход, и смешанный обход - от самого левого листа дерева через корень к самому правому листу.

Схематично алгоритм обхода двоичного дерева в соответствии с нисходящим способом может выглядеть следующим образом:

1. В качестве очередной вершины взять корень дерева. Перейти к пункту 2.

2. Произвести обработку очередной вершины в соответствии с требованиями задачи. Перейти к пункту 3.

3.а) Если очередная вершина имеет обе ветви, то в качестве новой вершины выбрать ту вершину, на которую ссылается левая ветвь, а вершину, на которую ссылается правая ветвь, занести в стек; перейти к пункту 2;

3.б) если очередная вершина является конечной, то выбрать в качестве новой очередной вершины вершину из стека, если он не пуст, и перейти к пункту 2; если же стек пуст, то это означает, что обход всего дерева окончен, перейти к пункту 4;

3.в) если очередная вершина имеет только одну ветвь, то в качестве очередной вершины выбрать ту вершину, на которую эта ветвь указывает, перейти к пункту 2.

4. Конец алгоритма.

Для примера рассмотрим возможные варианты обхода дерева (рис.8.18).

Библиотека ГГТУ им. П.О. Сухого

При обходе дерева представленного на рис.8.18 этими тремя методами мы получим следующие последовательности: ABCDEFG (нисходящий); CBAFEDG (смешанный); CBFEGDA (восходящий).

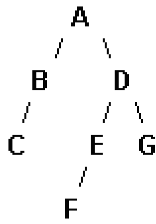


Рис.8.18. Схема дерева

НИСХОДЯЩИЙ ОБХОД (Preorder, r_Preorder).

В соответствии с алгоритмом, приведенным выше, текст процедуры имеет вид:

```
{===       Программный пример 6.4.        Нисходящий обход ===}
Procedure Preorder (t: TreePtr);
Type
   Stack=^Zveno;
  Zveno = record
            next: Stack;
              el: pointer;
          end;
Var
  st: stack;
   p: TreePtr;
(*------------ Процедура занесения в стек указателя ------*)
Procedure Push_st (var st:stack; p:pointer);
Var
  q: stack;
begin  new(q);  q^.el:=p;  q^.next:=st;   st:=g;  end;
(*----------- Функция извлечения из стека указателя ------*)
Function Pop_st (var st: stack):pointer;
Var
  e,p: stack;
begin  Pop_st:=st^.el;  e:=st;  st:=st^.next;  dispose(e);  end;
Begin
```


Библиотека ГГТУ им.П.О.Сухого

```
if t = NIL  then
 begin    writeln('Дерево пусто');  exit;   end
   else begin   st:=nil;   Push_st(St,t);   end;
while st <> nil do    { контроль заполнения стека }
  begin   p:=Pop_st(st);
        while p <> nil do
              begin    {  Обработка данных звена   }
                 if p^.right <> nil
                   then Push_st(st,p^.right);
                 p:=p^.left;
              end; end;
End; { Preorder }
```

Трассировка нисходящего обхода приведена в табл.8.1:

Таблица 8.1

РЕКУРСИВНЫЙ НИСХОДЯЩИЙ ОБХОД

Алгоритм существенно упрощается при использовании рекурсии. Так, нисходящий обход можно описать следующим образом:

1). Обработка корневой вершины;

2). Нисходящий обход левого поддерева;

3). Нисходящий обход правого поддерева.

Алгоритм рекурсивного нисходящего обхода реализован в программном примере 8.5.

{=== Программный пример 8.5. Рекурсивное выполнение нисходящего обхода ===}

```
Procedure r_Preorder (t: TreePtr);
begin
   if t = nil   then begin  writeln('Дерево пусто'); exit;  end;
(*------------------ Обработка данных звена --------------*)
          ................................
   if t^.left <> nil   then r_Preorder(t^.left);
   if t^.right <> nil  then r_Preorder(t^.right);
```


Библиотека ГГТУ им. П.О. Сухого

```
 End; { r_Preorder }
```

СМЕШАННЫЙ ОБХОД (Inorder, r_Inorder)

Смешанный обход можно описать следующим образом:

1) Спуститься по левой ветви с запоминанием вершин в стеке;
2) Если стек пуст то перейти к п.5;
3) Выбрать вершину из стека и обработать данные вершины;
4) Если вершина имеет правого сына, то перейти к нему; перейти к п.1.
5) Конец алгоритма.

В программном примере 8.6. реализован алгоритм смешанного обхода дерева.

```
{===   Программный пример 8.6.    Процедура смешанного обхода  ===}
Procedure Inorder (t: TreePtr);
 label 1;
 type   Stack=^Zveno;                        { тип стека }
        Zveno = record
               next: Stack;
               El: pointer;  end;
 Var
   st: stack;
    p: TreePtr;
 (*---------- Процедура занесения в стек указателя ---------*)
 Procedure Push_st (var st:stack; p:pointer);
 Var
   q: stack;
 begin   new(q);  q^.el:=p;  q^.next:=st;  st:=g;   end;
 (*------------ Функция извлечения из стека указателя ------*)
 Function Pop_st (var st: stack):pointer;
 Var
   e,p: stack;
 begin  Pop_st:=st^.el;  e:=st;   st:=st^.next;  dispose(e); end;
 Begin
   if t = NIL  then begin  writeln('Дерево пусто'); exit; end
      else begin  p:=t;  st:=nil;  end;
 1: while p^.left <> nil do
```


Библиотека ГГТУ им.П.О.Сухого

```
        begin   (* Спуск по левой ветви и заполнение очереди *)
           Push_st(st,p);   p:=p^.left;  end;
   if st = NIL          { контроль заполнения стека }
     then exit;
   p:=Pop_st(st);{выборка очередного элемента на обработку}
 (*-------------- Обработка данных звена -------------*)
   ................................
   if p^.right <> nil
     then begin  p:=p^.right;     { переход к правой ветви }
          goto 1;   end;
 End; { Inorder }
```

Трассировка смешанного обхода приведена в табл. 8.2:

Таблица 8.2

Рекурсивный смешанный обход описывается следующим образом:

1) Смешанный обход левого поддерева;
2) Обработка корневой вершины;
3) Смешанный обход правого поддерева.

Текст программы рекурсивной процедуры (r_Inorder) демонстрируется в программном примере 8.7.

{=== Программный пример 8.7. Рекурсивное выполнение смешанного обхода ===}

```
 Procedure r_Inorder(t: TreePtr);
 begin
   if t = nil then
   begin  writeln('Дерево пусто');  exit;   end;
   if t^.left <> nil   then R_inorder (t^.left);
 (*-------------- Обработка данных звена -------------*)
   ................................
   if t^.right <> nil  then R_inorder(t^.right);
 End;
```

ВОСХОДЯЩИЙ ОБХОД (Postorder, r_Postorder).

Библиотека ГГТУ им. П.О. Сухого

Трудность заключается в том, что в отличие от Preorder в этом алгоритме каждая вершина запоминается в стеке дважды: первый раз - когда обходится левое поддерево, и второй раз - когда обходится правое поддерево. Таким образом, в алгоритме необходимо различать два вида стековых записей: 1-й означает, что в данный момент обходится левое поддерево; 2-й - что обходится правое, поэтому в стеке запоминается указатель на узел и признак (код-1 и код-2 соответственно).

Алгоритм восходящего обхода можно представить следующим образом:

1) Спуститься по левой ветви с запоминанием вершины в сте ке как 1-й вид стековых записей;

2) Если стек пуст, то перейти к п.5;

3) Выбрать вершину из стека, если это первый вид стековых записей, то возвратить его в стек как 2-й вид стековых запи сей; перейти к правому сыну; перейти к п.1, иначе перейти к п.4;

4) Обработать данные вершины и перейти к п.2;

5) Конец алгоритма.

Текст программы процедуры восходящего обхода (Postorder) представлен в программном примере 8.8.

```
{=== Программный  пример 8.8.  Восходящий обход  ====}
 Procedure Postorder (t: TreePtr);
 label 2;
 Var
   p: TreePtr;
  top: point;         { стековый тип }
 Sign: byte;          { sign=1 - первый вид стековых записей}
                   { sign=2 - второй вид стековых записей}
 Begin     (*------------- Инициализация ------------------*)
   if t = nil  then
    begin  writeln('Дерево пусто');   exit;   end
      else begin  p:=t;  top:=nil;  end;  {инициализация стека}
 (*------- Запоминание адресов вдоль левой ветви -------*)
 2: while p <> nil do
   begin s_Push(top,1,p);  { заносится указатель 1-го вида}
         p:=p^.left;  end;
```


Библиотека БГУИР

```
(*-- Подъем вверх по дереву с обработкой правых ветвей ----*)
  while top <> nil do
  begin p:=s_Pop(top,sign); if sign = 1  then
    begin  s_Push(top,0,p); { заносится указатель 2-го вида  }
        p:=p^.right;   goto 2;  end
          else    (*---- Обработка данных звена ---------*)
    ................................
    end;    End; { Postorder }
```

РЕКУРСИВНЫЙ СМЕШАННЫЙ ОБХОДописывается следующим образом:

1). Восходящий обход левого поддерева;

2). Восходящий обход правого поддерева;

3). Обработка корневой вершины.

Текст программы процедуры рекурсивного обхода (r_Postorder) демонстрируется в програмном примере 8.9.

```
{ ====  Программный  пример 8.9.  ===== }
(*--------- Рекурсивное выполнение нисходящего обхода -----*)
Procedure r_Postorder (t: TreePtr);
Begin
  if t = nil then begin  writeln('Дерево пусто'); exit; end;
  if t^.left <> nil  then r_Postorder (t^.left);
  if t^.right <> nil  then r_Postorder (t^.right);
(*------------- Обработка данных звена -------------*)
 ..............................
End; { r_Postorder }
```

Если в рассмотренных выше процедурах поменять местами поля left и right, то получим процедуры обратного нисходящего, обратного смешанного и обратного восходящего обходов.

ПРОЦЕДУРЫ ОБХОДА ДЕРЕВА, ИСПОЛЬЗУЮЩИЕ СТЕК

Тип стека при нисходящем обходе.

```
    Stack=^Zveno;
   Zveno = record
          next: Stack;
```


Библиотека ГГТУ им.П.О.Сухого

```
        El: pointer;
      end;
```

Процедура включения элемента в стек при нисходящем и смешанном обходе (Push_st) приведена в программном примере 8.10.

```
{ ===  Програмнный  пример 8.10   ====}
 Procedure Push_st (var st: stack; p: pointer);
 Var
   q: stack;
 begin  new(q);  q^.el:=p;   q^.next:=st;  st:=q;   end;
```

Функция извлечения элемента из стека при нисходящем и смешанном обходе (Pop_st) приведена в программном примере 8.11.

```
{ ===  Програмнный  пример 8.11   ====}
 Function Pop_st (var st: stack):pointer;
 Var
    e,p: stack
 begin
    Pop_st:=st^.el;
    e:=st;       { запоминаем указатель на текущую вершину }
    st:=st^.next;{сдвигаем указатель стека на следующий
элемент}
    dispose(e);  { возврат памяти в кучу  }
 end;
```

При восходящем обходе может быть предложен следующий тип стека:

```
    point=^st;
      st = record
            next: point;
              l: integer;
            add: pointer;
          end;
```

Процедура включения элемента в стек при восходящем обходе (S_Push) приведена в программном примере 8.12.

```
{ ===  Програмнный  пример 8.12   ====}
 Procedure S_Push (var st: point; Har: integer; add: pointer);  Var
q: point;
```


Библиотека ГГТУ им. П.О. Сухого

```
begin
   new(q);             { выделяем место для элемента }
   q^.l:=Har;          { заносим характеристику }
   q^.add:=add;        { заносим указатель }
   q^.next:=st;        { модифицируем стек }
  st:=q;
end;
```

Функция извлечения элемента из стека при восходящем обходе (S_Pop) демонстрируется в программном примере 8.13.

```
{ === Програмнный пример 8.13  ====}
Function S_Pop (var st: point; var l: integer):pointer;
Var
  e,p: point;
begin
  l:=st^.l;
  S_Pop:=st^.add;
  e:=st;         { запоминаем указатель на текущую вершину}
  st:=st^.next; {сдвигаем указатель стека на след. элемент  }
  dispose(e);             { уничтожаем выбранный элемент }
end;
```

ПРОШИВКА БИНАРНЫХ ДЕРЕВЬЕВ.

Под прошивкой дерева понимается замена по определенному правилу пустых указателей на сыновей указателями на последующие узлы, соответствующие обходу.

Рассматривая бинарное дерево, легко обнаружить, что в нем имеются много указателей типа NIL. Действительно в дереве с N вершинами имеется (N+1) указателей типа NIL. Это незанятое пространство можно использовать для изменения представления деревьев. Пустые указатели заменяются указателями - "нитями", которые адресуют вершины дерева, и расположенные выше. При этом дерево прошивается с учетом определенного способа обхода. Например, если в поле left некоторой вершины P стоит NIL, то его можно заменить на адрес, указывающий на предшественника P, в соответствии с тем порядком обхода, для которого прошивается

Библиотека ГГТУ им. П.О. Сухого

дерево. Аналогично, если поле right пусто, то указывается преемник P в соответствии с порядком обхода.

Поскольку после прошивания дерева поля left и right могут характеризовать либо структурные связи, либо "нити", возникает необходимость различать их, для этого вводятся в описание структуры дерева характеристики левого и правого указателей (FALSE и TRUE).

Таким образом, прошитые деревья быстрее обходятся и не требуют для этого дополнительной памяти (стек), однако требуют дополнительной памяти для хранения флагов нитей, а также усложнены операции включения и удаления элементов дерева.

Прошитое бинарное дерево на Паскале можно описать следующим образом:

```
 type    TreePtr:=^S_Tree;
        S_Tree = record
             key: KeyType;     { ключ }
        left,right: TreePtr;     { левый и правый сыновья }
            lf,rf: boolean;     { характеристики связей }
            end;
```

где lf и rf - указывают, является ли связь указателем на элемент или нитью. Если lf или rf равно FALSE, то связка является нитью.

До создания дерева головная вершина имеет следующий вид: Здесь пунктирная стрелка определяет связь по нити. Дерево подшивается к левой ветви.

Рис. 8.19.

В программном примере 8.14 приведена функция (Inson) для определения сына (преемника) данной вершины.

```
{ ===  Програмнный  пример 8.14 ====}
 (*------ Функция, находящая преемника данной вершины X ----*)
```


Библиотека ГГТУ им. П.О.Сухого

```
(*-------- в соответствии со смешанным обходом ------------*)
Funtion Inson (x: TreePtr):TreePtr;
begin
   Inson:=x^.right;
   if not (x^.rf)  then exit;             { Ветвь левая ?}
   while Insonon^.lf do                    { связь не нить }
        Inson:=Inson^.left;      { перейти по левой ветви }
end; { Inson }
```

В программном примере 8.15 приведена функция (Int) для определения отца (предка) данной вершины.

```
{ ===  Програмнный  пример 8.15 ====}
(*---------- Функция, выдающая предшественника узла ------*)
(*------- в соответствии со смеш1анным обходом ------------*)
Function Inp (x:TreePtr):TreePtr;
begin
   Inp:=x^.left;
   if not (x^.lf)  then exit;
   while Inp^.rf do   Inp:=Inp^.right;   { связка не нить }
end;
```

В программном примере 8.16 приведена функция, реализующая алгоритм включения записи в прошитое дерево (leftIn). Этот алгоритм вставляет вершину P в качестве левого поддерева заданной вершины X в случае, если вершина X не имеет левого поддерева. В противном случае новая вершина вставляется между вершиной X и вершиной X^.left. При этой операции поддерживается правильная структура прошивки дерева, соответствующая смешанному обходу.

```
{ ===  Програмнный  пример 8.16 ====}
(*- Вставка p слева от x или между x и его левой вершиной -*)
Procedure LeftIn (x,p: TreePtr);
Var
  q: TreePtr;
begin
(*--------------- Установка указателя ------------------*)
  p^.left:=x^.left;    p^.lf:=x^.lf;    x^.left:=p;
  x^.lf:=TRUE;    p^.right:=x;        p^.rf:=FALSE;
  if p^.lf  then
(*-------- Переустановка связи с предшественником --------*)
```

```
begin  q:=TreePtr(Inp(p));    q^.right:=p;   q^.rf:=FALSE;
end;   end; { LeftIn }
```

Для примера рассмотрим прошивку дерева приведенного на рис.8.20. при нисходящем обходе.

Машинное представление дерева при нисходящем обходе с прошивкой приведено на рис.8.28.

Рис. 8.28. Машинное связное представление исходного дерева, представленного на рис.8.20 при нисходящем обходе с прошивкой

Трассировка нисходящего обхода с прошивкой приведена в табл.8.3.

Рассмотрим на примере того же дерева прошивку при смешанном обходе. Машинное представление дерева при смешанном обходе с прошивкой приведено на рис.8.28. @ указателя Узел

Обработка узла			Выходная строка
PT:=H		H	
LPH	A	A	A
LPA	B	B	AB
LPB	C	C	ABC
-LPC			
-RPC	D	D	ABCD
LPD	E	E	ABCDE
LPE	F	F	ABCDEF
-LPF			
-RPF	G	G	ABCDEFG
-LPG			
-RPG	H		Конец алгоритма

Таблица 8.3

Рис. 8.29. Машинное связное представление дерева при смешанном обходе с прошивкой

Библиотека ГГТУ им.П.О.Сухого

Трассировка смешанного обхода с прошивкой приведена в табл.6.4. @ указателя Узел Обработка узла Выходная строка

P:=PT		H	
LPH	A		
LPA	B		
LPB	C		
-LPC	C	C	C
-RPC	B	B	CB
-RPB	A	A	CBA
RPA	D		
LPD	E		
LPE	F		
-LPF	F	F	CBAF
-RPF	E	E	CBAFE
-RPE	D	D	CBAFED
RPD	G		
-LPG	G	G	CBAFEDG
-RPG	H		Конец алгоритма

Таблица 8.4.

Библиотека ГГТУ им.П.О.Сухого

Примеры задач на использование алгортитмов обработки графа

8.7. Примеры задач "на дерево"

Задача . Истоки и стоки. (на Алгоритм Флойда)

Задана карта односторонних дорог между городами. Город, из которого можно добраться до всех других городов, называется исток. Город, в который можно добраться из всех других городов, называется сток. Требуется найти все истоки и стоки.

Например. пусть имеем 6 городов (пронумерованных от 1 до 6) и 10 дорог:
2 1
2 4
4 1
4 2
4 3
1 3
3 5
5 3
3 6
6 3
Числа, описывающие дорогу, указывают, из какого города в какой можно добраться по данной дороге.
Тогда правильный ответ такой
истоки : 2 4
стоки : 3 5 6

На входе: Количество дорог N. В следующих N строках описываются дороги.
На выходе: В первой строке после слова "истоки:" через пробел указываются номера городов-истоков.
Во второй строке после слова "стоки:" через пробел указываются номера городов-стоков.

Пример ввода: 4 1 2 1 3

Библиотека ГГТУ им.П.О.Сухого

2 4 3 4
Вывод для примера ввода: истоки : 1 стоки : 4

Задача: МИНИМАЛЬНОЕ ОСТОВНОЕ ДЕРЕВО (Алгоритм Прима)

Задан связный неориентированный взвешенный граф G. В графе возможно наличие нескольких ребер между одной и той же парой вершин. Найдите вес минимального остовного дерева в графе G.

Входные данные.

Первая строка входного файла содержит целое число N ($1 \le N \le 100$) - количество вершин графа. Вторая строка входного файла содержит целое число M ($1 \le M \le 10000$) - количество ребер графа. В каждой из следующих M строк содержатся ровно три числа a, b, c ($1 \le a, b \le N$, $1 \le c \le 1000$). Эти числа описывают ребро, соединяющее вершины с номерами a и b и имеющее вес c. Вершины нумеруются последовательными натуральными числами от 1 до N.

Выходные данные.

Единственная строка выходного файла должна содержать одно число, равное весу минимального остовного дерева в графе G.

Пример.

input.txt	*output.txt*
4 4 1 2 1 2 3 2 3 4 3 1 4 4	6

Задача: (Алгоритм Дейкстры)

Библиотека ГГТУ им. П.О. Сухого

По одной корове с каждой из N (1 <= N <= 1000) ферм, пронумерованных

последовательно 1..N, собираются на вечеринку, которая состоится

на ферме X (1 <= X <= N). Каждая из M (1 <= M <= 100,000)

двунаправленных дорог, соединяет соединяет одну ферму с другой.

Всегда возможно добраться от одной фермы до другой с помощью этой

системы дорог (для любой пары ферм). Путешествие по дороге i

занимает Ti единиц времени. Одна или более пар ферм может быть

непосредственно связана более чем одной дорогой.

После того, как все коровы собрались на ферме X, они поняли,

что забыли ВСЕ что-то на своих фермах. Поэтому вечерину прервали,

все коровы отправились на свои фермы, взять то, что забыли и

вернулись на вечеринку. Все коровы выбрали оптимальный маршрут.

Какое минимальное время потребуется, чтобы продолжить вечеринку?

Формат ввода:

* Строка 1: Три разделенных пробелом целых числа: N M X

* Строки 2..M+1: Строка i+1 описывает дорогу i тремя целыми числами:

Ai Bi Ti. Эта дорога соединяет фермы Ai и Bi и для ее прохождения требуется Ti единиц времени.

Пример ввода

```
4 8 2
1 2 7
1 3 8
1 4 4
2 1 3
```


Библиотека ГГТУ им. П.О. Сухого

2 3 1
3 1 2
3 4 6
4 2 2

INPUT DETAILS:

4 коровы, 8 дорог, вечеринка на ферме 2.

Формат вывода:

* Строка 1: Одно целое число - минимальное количество времени,
через которое вечеринка сможет быть продолжена.

Пример вывода
6

OUTPUT DETAILS:

Непосредственные дороги соединяют ферму 2 с каждой из остальных ферм.
(к ферме 1 - времена 7 и 3), к ферме 3: 1, к ферме 4: 2.
Самый долгий путь (из кратчайших) - 3, поэтому ответ 6.

Задача. Максимальный поток.

Известная на весь Могилев компания "Headache" выпустила игру, для которой необходима конструкция, состоящая из маленьких платформ и труб. Платформы разделяются на стартовые (их N1 штук), финишные (N3 штук) и промежуточные (N2 штук). Все стартовые платформы находятся на одинаковой высоте. Финишные платформы также находятся на одинаковой высоте. Все высоты промежуточных платформ различны. Они меньше высоты стартовых, но больше высоты финишных. Каждой платформе соответствует уникальный номер от 1 до N1+N2+N3. Нумерация следующая:

Библиотека ГГТУ им. П.О.Сухого

сначала все стартовые платформы, затем промежуточные и, наконец, финишные. Все промежуточные платформы пронумерованы по убыванию высоты. То есть, если номер промежуточной платформы A меньше номера платформы B, то высота A больше высоты B.

На каждой из стартовых платформ находится шарик. Шарик может скатиться с платформы A на платформу B, если они соединены трубой, и высота A больше высоты B. На каждой из финишных платформ может оказаться не более одного шарика. Если шарик находится на некоторой платформе, то игрок может выбрать направление дальнейшего пути шарика, т.е. выбрать платформу, на которую шарик скатится. Также для каждой промежуточной платформы задано число C, равное максимальному количеству шариков, которые могут прокатиться по ней за время игры. Цель игры заключается в том, чтобы на финишных платформах оказалось как можно больше шариков.

Вам нужно узнать, какое максимальное количество шариков может оказаться на финишных платформах в результате игры.

Формат ввода

Во входном файле Game.in находятся информация о конструкции в следующей форме:
N1 N2 N3
C_{N1+1}
...
C_{N1+N2}
K1 A[1,1] ... A[1,K1]
K2 A[2,1] ... A[2,K2]
...
K_{N1+N2} A[N1+N2,1] ... A[N1+N2,K_{N1+N2}]
где числа N1, N2, N3 - соответственно количество стартовых, промежуточных и финишных платформ. Cj - максимальное количество шариков, которые могут прокатиться по промежуточной платформе с номером j (N1+1 <= j <= N1+N2) за все время игры. Ki - количество труб, выходящих из платформы с номером i (1 <= i <= N1+N2). Элементы массива A, перечисленные в строке, являются номерами платформ, на которые может скатиться шарик с соответствующей платформы.

Ограничения

Все числа на вводе целые.
0<N1,N3<51
1<N1+N2+N3<201
0 <= Cj <= 50
Не существует труб между стартовыми платформами.
Не существует труб между финишными платформами.

Формат вывода

В первой строке выходного файла Game.out должно находиться единственное число, равное максимальному количеству шариков, которое может оказаться на финишных платформах в результате игры.

Пример ввода

3 4 3
3
2
1
2
1 4
1 4
1 4
2 5 6
1 7
1 7
3 8 9 10

Пример вывода

2

Библиотека ГГТУ им. П.О. Сухого

Кравченко Ольга Алексеевна

СТРУКТУРЫ ДАННЫХ В ЯЗЫКЕ СИ

Пособие
по курсам «Модели и структуры данных»
и «Основы алгоритмизации и программирования»
для студентов специальностей 1-40 01 02
«Информационные системы и технологии
(по направлениям)» и 1-36 04 02 «Промышленная
электроника» дневной и заочной форм обучения

Подписано к размещению в электронную библиотеку
ГГТУ им. П. О. Сухого в качестве электронного
учебно-методического документа 22.10.2010.
Рег. № 15Е.
E-mail: ic@gstu.by
http://www.gstu.by

Библиотека ГГТУ им.П.О.Сухого