ГЛАВА 15

ДИНАМИЧЕСКИЕ СТРУКТУРЫ

ВВЕДЕНИЕ

В языке С определены следующие структуры данных:

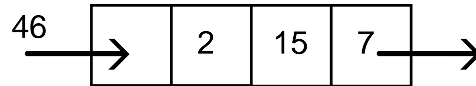
- одиночное данное (простая переменная)
- массив однотипных данных
- структура (совокупность разнородных данных, относящихся к одному объекту).

Цель описания типа данных и последующего определения некоторых переменных, как относящихся к этому типу, состоит в том, чтобы раз и навсегда зафиксировать диапвзон значений, присваиваемых этим переменным, и размер выделяемой для них памяти. Поэтому о таких переменных говорят как о *статических* переменных.

Однако не всегда такие средства работы с информацией оказываются достаточными и удобными. Некоторые задачи исключают использование структур данных фиксированного размера и требуют введения *динамических* структур, способных увеличиваться и уменьшаться в размерах в процессе работы программы. Каждая структура данных характеризуется, во-первых, взаимосвязью элементов и, во-вторых, набором типовых операций над этой структурой. В случае динамической структуры важно:

- каким образом может расти и сокращаться данная структура данных;
- каким образом можно включить в структуру новый и удалить существующий элемент;
- как можно обратиться к конкретному элементу структуры для выполнения над ним определенной операции (доступ по индексу здесь, очевидно, менее удобен, чем это было в случае массивов, т.к. любой элемент при изменении размеров структуры может изменить свою позицию. Поэтому обычно доступ к элементам динамической структуры *относительный*: найти следующий (предыдущий) элемент по отношению к текущему, найти последний элемент и т.п.).

Одной из простейших и в то же время типичных структур данных является *очередь*. Проблема очереди возникает, когда имеется некоторый механизм обслуживания, который может выполнять заказы последовательно, один за другим. Если при поступлении нового заказа данное устройство свободно, оно немедленно приступает к выполнению этого заказа; если же оно уже выполняет какой-то ранее полученный заказ, то новый заказ поступает в *конец* очереди других заказов, ожидающих выполнения. Когда устройство освобождается, оно приступает к выполнению заказа из *начала* очереди (этот заказ удаляется из очереди, и первым в ней становится следующий заказ). Если заказы поступают нерегулярно, очередь то удлиняется, то укорачивается, и даже может оказаться пустой.



Очередь часто называют системой, организованной и работающей по принципу FIFO (first in - first out). Основными операциями с очередью являются добавление нового элемента в конец и удаление элемента из начала очереди.

Рассмотрим, как может быть реализована работа с очередью на С с использованием структур, указателей на структуры и функций динамического выделения/освобождения памяти.

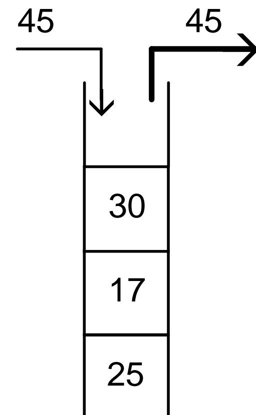
В программе, приведенной ниже, работа с очередью ведется с помощью двух функций:

- функция ***insert()*** отводит память под очередной элемент, заносит в него нужную информацию и ставит в конец очереди;
- функция ***take_off()*** удаляет из очереди ее первый элемент, освобождает память из-под него и перемещает указатель начала очереди на следующий элемент. В случае попытки удаления элемента из пустой очереди параметр ошибки *error* получает значение 1.

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

#define QUEUE struct queue

QUEUE
  { int info;          //  поле информации элемента очереди
   QUEUE *next; //  поле ссылки на следующий элемент очереди
  };

void insert (QUEUE **q, int item)
  {
   QUEUE *tek=*q;  // указатель на текущее звено очереди
   QUEUE *pred=0;  // pred содержит адрес последнего элемента
   QUEUE *new_n;

   while(tek)  // просматриваем очередь до конца
      { pred=tek; tek=tek->next; }
   new_n=(QUEUE*)malloc(sizeof(QUEUE));
   new_n->info=item;
   if(pred)  // если очередь была не пуста
     { new_n->next=pred->next;
      pred->next=new_n;
     }
    else { *q=new_n; (*q)->next=0; }
  }

int take_off (QUEUE **q, int *error)
```

```
  {   int   value=0;      //  значение  возвращаемого элемента очереди
   QUEUE *old=*q;  // указатель на старую голову очереди
   if(*q)  // если очередь не пуста
     { value=old->info; *q=(*q)->next;
       free(old); *error=0;  // элемент удален из очереди
     }
    else *error=1;
   return value;
  }
void main()
{
  int error, j;
  QUEUE *q1, *q2;
  for(j=12; j<=15; j++)
    insert(&q1, j);
  for(j=1; j<=4; j++)
    insert(&q2, take_off(&q1, &error));
  for(j=1; j<=4; j++)
    printf("\n Удален из q2:%d",take_off(&q2, &error) );
}
```

Другая часто встречающаяся структура данных - *стек* (*магазин*) - отличается от очереди тем, что она организована по принципу LIFO (last in - first out). Операции включения и удаления элемента в стеке выполняются только с одного конца, называемого *вершиной* стека. Когда новый элемент помещается в стек, то прежний верхний элемент как бы "проталкивается" вниз и становится временно недоступным. Когда же верхний элемент удаляется с вершины стека, предыдущий элемент "выталкивается" наверх и опять является доступным.



Потребность в организации стека возникает при решении, например, такой задачи. Пусть имеется текст, сбалансированный по круглым скобкам. Необходимо построить таблицу, в каждой строке которой будут находиться координаты соответствующих пар скобок. Т.е. для текста

```
(....(.........)...(..........).....)
0    17   42     61    84   95
```

таблица должна быть такой:

17	42
61	84
0	95

Поскольку текст сбалансирован по круглым скобкам, то как только встретится ')', это будет пара для *последней пройденной* '('. Поэтому алгоритм решения данной задачи может быть следующим: будем идти по тексту и как только встретим '(', занесем ее координату в стек; как только встретится ')', возьмем из стека верхнюю координату и распечатаем ее вместе с координатой данной ')'.

Рассмотрим программу, в которой реализована работа со стеком. В ней использованы функции:

push() - положить элемент на вершину стека,

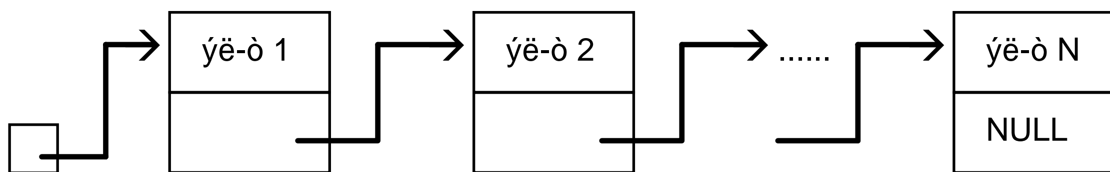
pop() - вытолкнуть верхний элемент из стека,

peek() - прочитать значение верхнего элемента, не удаляя сам элемент из стека.

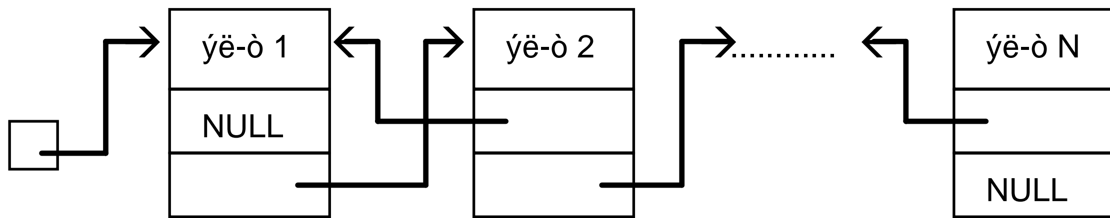
```
#include <stdio.h>
#include <alloc.h>

#define STACK struct stack

STACK
{ int info;            // поле значения элемента стека
  STACK *next;     // поле ссылки на следующий элемент стека
};

void push ( STACK **s, int item)
{ STACK *new_n;
  new_n=(STACK*)malloc(sizeof(STACK));// запросили память под новый элемент
  new_n->info=item;       // заносим значение в новый элемент
  new_n->next=*s;         // новый элемент стал головой стека
  *s=new_n;               // указателю *s присвоили адрес головы стека
}

int pop (STACK **s, int *error)
{
  STACK *old=*s;
  int info=0;

  if(*s)                //стек не пуст
   { info=old->info; *s=(*s)->next;
     free(old);//освобождаем память из-под выбранного элемента
     *error=0;// элемент удален успешно
   }
   else *error=1;
  return(info);
}

int peek (STACK **s, int *error)
{
```

```
  if (*s) { *error=0; return(*s)->info; }
   else { *error=1; return 0;}
}
void main()
{ int error, i;
  STACK *s1, *s2;

  push(&s1, 42);
  printf("\n peek(s1)=%d", peek(&s1, &error));
  push(&s1, 53);
  printf("\n peek(s1)=%d", peek(&s1, &error));
  push(&s1, 72);
  printf("\n peek(s1)=%d", peek(&s1, &error));
  push(&s1, 86);
  printf("\n peek(s1)=%d", peek(&s1, &error));
  for(i=1; i<=4; i++)
     push(&s2, pop(&s1, &error));
  for(i=1; i<=4; i++)
     printf("\n pop(&s2)=%d", pop(&s2, &error));
}
```

Стеки и очереди являются одними из разновидностей более широкого класса структур данных - *списков*. *Связянный список* - это структура следующего вида:



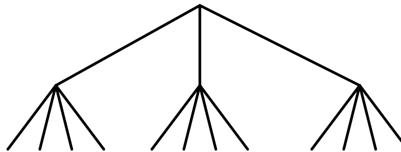
Это *простой однонаправленный список*, в котором каждый элемент (кроме последнего) имеет ссылку на следующий элемент и поле информации. Можно организовать также *кольцевой* список (в нем последний элемент будет содержать ссылку на первый) или *двунаправленный список* (когда каждый элемент, кроме первого и последнего, имеет две ссылки: на предыдущий и следующий элемент) и т.д. Кроме того, можно помещать в начале списка дополнительный элемент, называемый *заголовком списка.* Как правило, заголовок списка используется для хранения информации обо всём списке. В частности, он может содержать счётчик числа элементов в списке. Наличие заголовка приводит к усложнению одних и упрощению других программ, работающих со списками.



При работе со списком могут быть полезны следующие базовые функции:

- *insert()* - добавить новый элемент в список таким образом, чтобы список оставался упорядоченным в порядке возрастания по значению одного из полей;
- *take_out()* - удалить элемент с заданным полем (если он есть в списке);
- *is_present()* - определить, имеется ли в списке заданный элемент;
- *display()* - распечатать значения всех полей элементов списка;
- *destroy_list()* - освободить память, занимаемую списком.

Довольно часто при работе с данными бывает удобно использовать структуры с *иерархическим* представлением, которые хорошо изображаются с помощью *дерева*.



Дерево состоит из элементов, называемых *узлами* (вершинами). Узлы соединены между собой *направленными дугами*. В случае X✹Y вершина X называется *предшественником* (*родителем*), а Y - *преемником* (*сыном*). Дерево имеет единственный узел - *корень*, у которого нет предшественников. Любой другой узел имеет ровно одного предшественника. Узел, не имеющий преемника, называется *листом*.

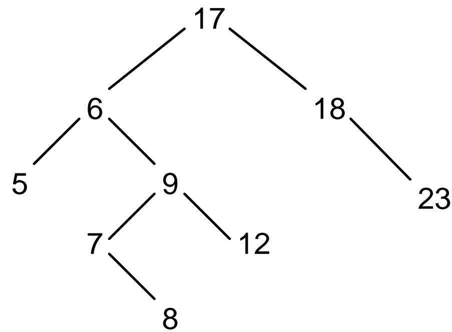
Рассмотрим работу с *бинарным* деревом (в котором у каждого узла может быть только два преемника - левый и правый сын). Необходимо уметь:

- построить дерево;
- выполнить поиск элемента с указанным значением в узле;
- удалить заданный элемент;
- обойти все элементы дерева (например, чтобы над каждым из них провести некоторую операцию).

Обычно бинарное дерево строится сразу упорядоченным, т.е. узел левого сына имеет значение меньшее, чем у родителя, а узел правого сына - большее. Например, если приходят числа

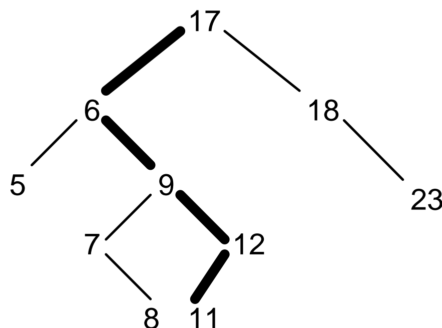
17, 18, 6, 5, 9, 23, 12, 7, 8,

то построенное по ним дерево будет выглядеть так:



Для того, чтобы вставить новый элемент в дерево, надо найти место, куда поставить этот элемент. Для этого, начиная с корня, будем сравнивать значения узлов (Y) со значением нового элемента (NEW). Если NEW < Y , то пойдем по левой ветви; в противном случае - по правой. Когда мы дойдем до узла, из которого не выходит нужная ветвь для дальнейшего поиска - это означает, что место под новый элемент найдено.

Путь поиска места для числа 11 в построенном выше дереве показан на рисунке.

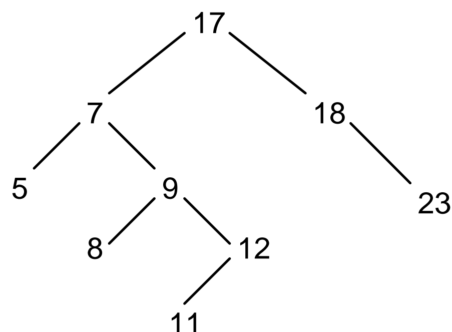


При удалении узла из дерева возможны три ситуации:

а) исключаемый узел - лист (в этом случае надо просто удалить ссылку на данный узел);

б) из исключаемого узла выходит только одна ветвь;

в) из исключаемого узла выходят две ветви (в таком случае на место удаляемого узла надо поставить либо *самый правый* узел *левой* ветви, либо *самый левый* узел *правой* ветви для сохранения упорядоченности дерева). Например, построенное ранее дерево после удаления узла 6 может стать таким:



Рассмотрим теперь задачу обхода дерева. Существуют три способа обхода, которые естественно следуют из самой структуры дерева.

1). Обход слева направо: A, R, B

(сначала посещаем левое поддерево, затем - корень и, наконец, правое поддерево)

2). Обход сверху вниз: R, A, B

(посещаем корень *до* поддеревьев)

3). Обход снизу вверх: A, B, R

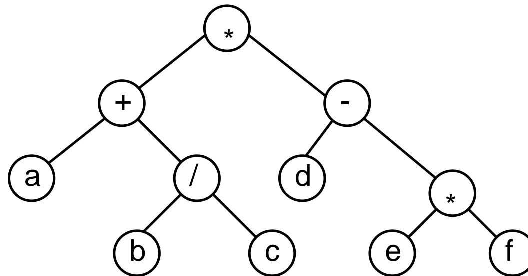
(посещаем корень *после* поддеревьев)

Интересно проследить результаты этих трех обходов на примере записи формул в виде дерева.

Например, формула

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

будет представлена так:



(дерево формируется по принципу: операция - узел, операнды - поддеревья).

Обход 1 дает обычную *инфиксную* запись выражения (правда, без скобок).

Обход 2 дает *префиксную* запись *+a/bc-d*ef.

Обход 3 - *постфиксную* (ПОЛИЗ - польская инверсная запись): abc/+def*-*.

В качестве примера работы с древовидной структурой данных рассмотрим решение следующей задачи.

Вводятся фамилии абитуриентов; необходимо распечатать их в алфавитном порядке с указанием количества повторений каждой фамилии.

В программе будет использована рекурсивная функция ***der()***, которая строит дерево фамилий, а также рекурсивная функция для печати дерева ***print_der()***, в которой реализован первый способ обхода дерева.

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

#define TREE struct der

TREE
        {
          char *w;
```

```
            int c;
            TREE *l;
            TREE *r;
          };
TREE *der(TREE *kr, char *word)
{
  int sr;
  if(kr==NULL)
   {
    kr=(TREE *)malloc(sizeof(TREE));
    kr->w=word;
    kr->c=1;
    kr->l=kr->r=NULL;
   }
   else if((sr=strcmp(word, kr->w))==0) kr->c++;
          else if(sr<0) kr->l = der(kr->l, word);
                else kr->r = der(kr->r, word);
  return kr;
}
void print_der(TREE *kr)
{
  if(kr)
   { print_der(kr->l);
    printf("слово - %-20s \tкол-во повтор.- %d\n", kr->w, kr->c);
    print_der(kr->r);
   }
}
void main()
{
   int i;
  TREE *kr;
  static char word[40][21];
  kr=NULL; i=0;
  puts("Введите <40 фамилий длиной <20 каждая"); scanf("%s", word[i]);
  while(word[i][0]!='\0')
      { kr=der(kr,word[i]);
        scanf("%s", word[++i]);
      }
  print_der(kr);
}
```

Рассмотрим еще один пример использования динамических структур данных. По введенной формуле необходимо построить ее ПОЛИЗ. (При записи формулы используются только операции +, —, *, / и буквы - операнды).

```
#include "stack.h"  // в файле - описанные выше функции работы со стеком
#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>
#include <string.h>

#define ZNAK (c=='*' || c=='/')

void main()
{ char s[50], s1[80], c;
  int er, i, l, k=0;
  STACK *st=NULL;

  puts("Введите инфиксную запись формулы, операндами в которой");
  puts("являются буквы, а знаками - символы +, —, *, /");
  gets(s);
  l=strlen(s);
  push(&st, '(');              // заносим '(' в стек и
  for (i=0; i<l; i++)          // просматриваем выражение слева направо
    if(isalpha(s[i])) s1[k++]=s[i];        // операнд заносим в выходной массив
     else if(s[i]=='(') push(&st, '(');     // '(' кладется в стек
          else if(s[i]==')') while((c=pop(&st, &er))!='(')
                              s1[k++]=c;   // извлекаем из стека все до ближайшей
                                           // '(', которую тоже удаляем из стека
               else switch(s[i])   // см. комментарий после программы
                    {
                     case '+':
                     case '—': while((c=peek(&st, &er))!='(')
                                 s1[k++]=pop(&st, &er);
                               push(&st, s[i]); break;
                     case '*':
                     case '/': while((c=peek(&st, &er))!='(')
                               { if( ! ZNAK)break;
                                 s1[k++]=pop(&st, &er);
                               }
                              push(&st, s[i]); break;
                     default:  printf("Неверный символ s[%d] : %c !", i+1, s[i]);
                               exit(-1);
                    }
  while((c=pop(&st, &er)) !='(') // в заключение выполняются такие же действия,
    s1[k++]=c;                   // как если бы встретилась закрывающая скобка
```

```
 s1[k]='\0';
 puts("\n\t\tПОЛИЗ:\n\n\t"); puts(s1);
}
```

Комментарий:

Когда в записи формулы встречается знак операции - не скобка и не операнд - то с вершины стека извлекаются (до ближайшей скобки, которая сохраняется в стеке) знаки операций, старшинство которых больше или равно старшинству данной операции, и они заносятся в выходной массив, после чего рассматриваемый знак заносится в стек.

ЗАДАНИЕ НА ПРОГРАММИРОВАНИЕ

Задача 1

В файле находится текст программы на языке FORTRAN. Написать, используя стек, препроцессор, проверяющий правильность вложений циклов в этой программе.

Задача 2

В файле находится текст, в котором используются скобки трех типов: (), [], { }. Используя стек, проверить, соблюден ли баланс скобок в тексте.

Задача 3

Используя очередь или стек, решить задачу: в файле записан текст, сбалансированный по круглым скобкам. Требуется для каждой пары соответствующих открывающей и закрывающей скобок напечатать номера их позиций в тексте, упорядочив пары номеров по возрастанию номеров позиций:

а) закрывающих скобок;

(например, для текста a+(45-f(x)*(b-c)) надо напечатать: 8 10; 12 16; 3 17)

б) открывающих скобок.

(например, для текста a+(45-f(x)*(b-c)) надо напечатать: 3 17; 8 10; 12 16)

Задача 4

Используя стек, проверить, является ли содержимое текстового файла правильной записью формулы следующего вида:

<формула>::=<терм> | <терм>+<формула> | <терм>-<формула>
<терм>::=<имя> | (<формула>)
<имя>::=x|y|z

Задача 5

В текстовом файле записана без ошибок формула следующего вида:

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

(M обозначает функцию *max*, а m - функцию *min*).

Используя стек, вычислить как целое число значение данной формулы.

Например, M(5,m(6,8))=6.

Задача 6

В текстовом файле записано без ошибок логическое выражение следующего вида:

<лог.выр.>::=true | false | !<лог.выр.> | <лог.выр.>&&<лог.выр.> | <лог.выр.> || <лог.выр.>

Используя стек, вычислить значение этого выражения с учётом общепринятого приоритета операций.

Задача 7

Написать и протестировать функции включения, удаления и чтения очередного элемента стека объёмом *n* элементов. В случае невозможности включения (переполнения стека), удаления из пустого стека выставлять флаг.

Задача 8

Написать и протестировать функции для включения, исключения и поиска элемента кругового списка для:

а) списка без заголовка;

б) списка с заголовком (заголовок может содержать некоторую информацию о списке, например, число элементов в списке).

Задача 9↑

Используя двунаправленный список, написать и протестировать функции, реализующие отдельные действия при игре в "новое домино", в котором кроме традиционных правил игрок может на каждом ходу заменить любую конечную кость на свою. Необходимы следующие функции:

а) проверить, можно ли сделать ход;

б) сделать ход;

в) проверить, можно ли сделать замену;

г) заменить кость;

д) определить, закончена ли игра.

Задача 10↑

Написать и протестировать функции поиска. вкючения и исключения элемента бинарного дерева с заданным ключом.

Задача 11

Используя стек, написать нерекурсивную версию алгоритма Хоара быстрой сортировки (см. [1], стр. 99-100; [7], стр. 114-119).

Задача 12

Напишите программу сложения двух длинных целых чисел, представленных в виде строк, используя:

а) круговые списки;

б) двунаправленные списки.

Задача 13

а). Два бинарных дерева подобны, если либо оба они пусты, либо оба непусты, и при этом их левые поддервья подобны и их правые поддеревья подобны. Определить, являются ли два дерева подобными.

б). Два бинарных дерева зеркально подобны, если либо оба они пусты, либо оба непусты, и при этом левое поддерево одного из них подобно правому поддереву другого и наоборот.

Определить, являются ли два дерева зеркально подобными.

Задача 14

Для организации вычисления значения выражения на ЭВМ удобнее вместо обычной (инфиксной) записи использовать постфиксную (или польскую инверсную) запись - ПОЛИЗ. При вычислении выражения, записанного в ПОЛИЗе, операции выполняются в том порядке, в котором они встречаются при просмотре выражения слева направо; поэтому отпадает необходимость использования скобок и многократного сканирования выражения из-за различного старшинства операций.

Например, выражению 2+3*4 соответствует ПОЛИЗ 234*+, а выражению (a+(b^c)^d)*(e+f/d) - запись abc^d^+efd/+* .

Используя стек,

а) распечатать в инфиксной форме выражение, заданное в ПОЛИЗе;

б) вычислить как целое число значение выражения, записанного в ПОЛИЗе;

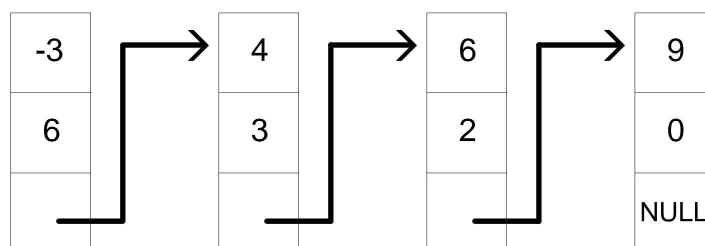
в) выражение, записанное в ПОЛИЗе, перевести в инфиксную форму.

Задача 15

Многочлен от одной переменной X можно представить как связанный список с узлами вида:

êîýôô-ò ïðè X
ñòåïåíü X
ññûëêà íà ñëåä. óçåë

Например, многочлен $-3x^6 + 4x^3 + 6x^2 + 9$ будет храниться в виде списка:



в котором узлы расположены в порядке убывания степеней X.

а) По введенной безошибочной записи многочлена построить его представление в виде списка.

б) Сложить два многочлена, представленных в виде списка (нулевые слагаемые исключить из результирующего списка).

в) Проверить на равенство два многочлена.

г) Вычислить значение многочлена ***s(x)***, представленного в виде списка, в целочисленной точке X.

д) По многочлену ***s(x)*** построить его производную - многочлен ***p(x)***.

е) Привести подобные члены в многочлене и расположить его по убыванию степеней X (например, из $-8x^4 - 74x + 8x^4 + 5 - x^3$ получить $-x^3 - 74x + 5$).

ж) Перемножить два многочлена, заданных в виде списков.

з) Распечатать многочлен, заданный в виде списка, в обычном виде (например, так: 52y^3 - 6y^2 + y).

Задача 16↑

Составить программу, которая читает произвольный С-файл и печатает в алфавитном порядке каждую группу имен переменных, совпадающих в первых N символах, но различающихся где-либо дальше. Параметр N задается из командной строки. При решении задачи использовать дерево с узлами вида

```
struct NODE
 {
  char *word;
  int k;
  NODE *left;
  NODE *right;
 };
```

(Не рассматривать слова внутри символьных строк и комментариев !)

Задача 17↑

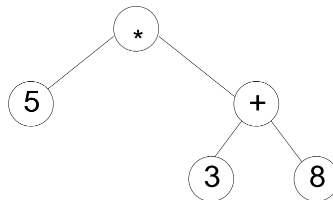
Формулу вида:

<формула>::=<цифра>|(<формула><знак><формула>)
<знак>::=+|-|*
<цифра>::=0|1|2|3|4|5|6|7|8|9

можно представить в виде бинарного дерева по следующим правилам:

- формула из одной цифры представляется деревом из одной вершины с этой цифрой;
- формула вида f1 # f2 представляется деревом, в котором корень - это знак # соответствующей операции, а левое и правое поддеревья - это представления формул f1, f2 в виде бинарных деревьев.

Например, формуле 5*(3+8) соответствует дерево:



Требуется:

а) построить дерево по формуле указанного выше вида;
б) вычислить как целое число значение дерева-формулы;
в) по заданному дереву распечатать соответствующую формулу.

Задача 18

С использованием структуры "стек" переписать содержимое текстового файла, разделенного на строки, в другой файл. Причем, необходимо переносить в конец каждой строки все входящие в нее цифры в обратном порядке.

Задача 19

С использованием структуры "очередь" за один просмотр файла, содержащего целые числа, распечатать файл в следующем виде: сначала - все числа, меньшие А; затем - все числа из [А, В]; потом - все остальные числа.

Задача 20

Пусть имеется ЭВМ, у которой один регистр и шесть команд:

LD A	загрузить А в регистр;
ST A	запомнить содержимое регистра в А;
AD A	сложить содержимое регистра с А;
SB A	вычесть А из содержимого регистра;
ML A	умножить содержимое регистра на А;
DV A	разделить содержимое регистра на А.

Напечатать последовательность машинных команд, необходимых для вычисления выражения, задаваемого в постфиксной форме. Выражение может состоять из однобуквенных операндов и знаков операций сложения, вычитания, умножения и деления. В качестве рабочих использовать переменные вида Tn.

Например, выражению ABC*+DE--/ будет соответствовать следующая последовательность команд:

```
LD   B
ML   C
ST   T1
LD   A
AD   T1
ST   T2
LD   D
SB   E
ST   T3
LD   T2
DV   T3
ST   T4
```

Задача 21↑

Составить программу, формирующую "перекрестные ссылки" - т.е., печатающую список слов, которые встречаются в анализируемом файле, а для каждого слова - список номеров строк, в которых это слово встречается. При решении задачи рекомендуется использовать следующие структуры данных:

```
struct LIST    // список номеров строк для данного слова
  { int num;
    struct LIST *p;
  }
struct NODE  // узел дерева с информацией об очередном слове
 { char *word;
   struct LIST *lines;
   struct NODE *left;
   struct NODE *right;
 }
```

Задача 22↑

Закодировать введённое сообщение с использованием алгоритма Хаффмена
(см. [8], стр.331).

Задача 23

Распечатать слова текста, отсортированные в порядке убывания частоты их встречаемоти (рядом с каждым словом выводить значение счетчика частоты его вхождений в текст). При решении задачи использовать дерево следующей структуры:

```
struct NODE
 { char *word;
   int k;
   struct NODE *left;
   struct NODE *right;
 }
```

ЛИТЕРАТУРА

1. Вирт Н. Алгоритмы + структуры данных = программы. -М.: Мир, 1985
2. Пратт Т. Языки программирования. Разработка и реализация (стр. 84-95, 141-153, 195). -М.: Мир, 1979.
3. Пильщиков В.Н. Сборник упражнений по языку ПАСКАЛЬ (стр. 113). -М.: Наука, 1989.
4. Абрамов В.Г. и др. Введение в язык ПАСКАЛЬ. (стр. 280). -М.: Наука, 1988.
5. Кнут Д. Искусство программирования для ЭВМ (т. 1, гл. 2). -М.: Мир, 1976.
6. Трифонов Н.П., Громыко В.И. Программирование на автокоде ЕС ЭВМ (стр. 286). -М.: Наука, 1985.
7. Вирт Н. Алгоритмы и структуры данных. -М.: Мир, 1989.
8. Лэнгсам Й. и др. Структуры данных для персональных ЭВМ. -М.: Мир, 1976.
9. Уэзерелл Ч. Этюды для программистов. - М.: Мир, 1982.