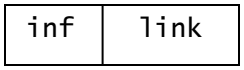
15

Линейни динамични структури от данни. Стек. Опашка. Свързан списък. Приложения

15.1 Стек

В предишната глава разгледахме логическото описание и физическото представяне на структурата от данни стек. Реализирахме както последователното, така и свързаното представяне на стек. При реализацията на свързаното представяне дефинирахме двойната кутия



чрез структура.

В тази част ще направим още една реализация на стек. Ще предложим също някои негови приложения.

15.1.1 Още една реализация на свързаното представяне на стек

Ще дефинираме класа от цели числа stack, който използва помощния и малко странен клас item за реализиране на двойната кутия.

```
class item
{friend class stack;
 private:
  item(int x = 0)    // конструктор
```

```
    {inf = x;
     link = NULL;
    }
    int inf;
    item* link;
  };
```

Тъй като item използва класа stack в декларацията си, прототипът на класа stack трябва да предшества декларацията на item. Странността на item произтича от това, че всичките му членове са капсолирани. Чрез декларацията

```
  friend class stack;
```

item позволява само класът stack да създава и обработва негови обекти. Конструкторът на item има един подразбиращ се аргумент, което позволява да бъде използван и като конструктор по подразбиране.

Класът stack има вида:

```
  class stack
  {public:
    stack();
    stack(int x);
    ~stack();
    stack(stack const &);
    stack& operator=(stack const &);
    int push(int const&);
    int pop(int & x);
    int top() const;
    bool empty() const;
    void print();
   private:
    item *start;
    void delstack();
    void copy(stack const&);
  };
```

Предложени са два конструктора:

```
  stack::stack()
  {start = NULL;
  }
```

и

```
stack::stack(int x)
{start = new item(x);
}
```

Необходими са за да могат да бъдат създавани празен стек и стек с един (указан като аргумент) елемент.

Тъй като обектите на класа stack са реализирани в динамичната памет, за него трябва да реализираме каноничното представяне – деструктор, конструктор за присвояване и операторна функция за присвояване. Тези член-функции са аналогични на съответните от реализацията на стека при предишното представяне.

деструктор

```
stack::~stack()
{delstack();
}
```

конструктор за присвояване

```
stack::stack(stack const & r)
{copy(r);
}
```

операторна функция за присвояване

```
stack& stack::operator=(stack const& r)
{if (this != &r)
{delstack();
 copy(r);
}
return *this;
}
```

където член-функциите delstack() и copy(stack const &) също са аналогични на тези от предишното представяне.

Класът stack реализира стек от цели числа. В следващото приложение ще имаме нужда от два класа стек: клас стек от цели числа и клас стек от символи. Затова, като използваме проектирания клас stack, ще дефинираме шаблон на клас stack за горното представяне.

```
template <class T>
class stack;
```

```
template <class T>
class item
{friend class stack<T>;
 private:
  item(T x = 0)
  {inf = x;
   link = 0;
  }
  T inf;
  item* link;
};
template <class T>
class stack
{public:
  stack(T x);
  stack();
  ~stack();
  stack(stack const &);
  stack& operator=(stack const &);
  void push(T const&);
  int pop(T & x);
  T top() const;
  bool empty() const;
  void print();
 private:
  item<T> *start;
  void delstack();
  void copy(stack const&);
};
template <class T>
stack<T>::stack(T x)
{start = new item<T>(x);
}
template <class T>
stack<T>::stack()
{start = NULL;
```

```
}
template <class T>
stack<T>::~stack()
{delstack();
}
template <class T>
stack<T>::stack(stack<T> const & r)
{copy(r);
}
template <class T>
stack<T>& stack<T>::operator=(stack<T> const& r)
{if (this != &r)
{delstack();
 copy(r);
}
return *this;
}
template <class T>
void stack<T>::delstack()
{item<T> *p;
 while (start)
 {p = start;
  start = start->link;
  delete p;
 }
}
template <class T>
void stack<T>::copy(stack<T> const & r)
{if (r.start)
{item<T> *p = r.start,
         *q = NULL,
         *s = NULL;
 start = new item<T>;
 start->inf = p->inf;
 start->link = q;
 q = start;
```

```
 p = p->link;
 while (p)
 {s = new item<T>;
  s->inf = p->inf;
  q->link = s;
  q = s;
  p = p->link;
 }
 q->link = NULL;
}
else start = NULL;
}
template <class T>
void stack<T>::push(T const& x)
{item<T> *p = new item<T>(x);
 p->link = start;
 start = p;
}
template <class T>
int stack<T>::pop(T & x)
{item<T> *temp,
         *p = start;
 if (p)
 {x = p->inf;
  temp = p;
  p = p->link;
  delete temp;
  start = p;
  return 1;
 }
 else return 0;
}
template <class T>
T stack<T>::top() const
{return (*start).inf;
}
```

```
template <class T>
void stack<T>::print()
{T x;
 while (pop(x))
   cout << x << " ";
 cout << endl;
}
template <class T>
bool stack<T>::empty() const
{return start == NULL;
}
```

Ще използваме тази дефиниция на шаблона stack за да покажем как става пресмятането на стойността на аритметичен израз чрез преобразуването му в обратен полски запис.

15.1.2 Приложение на структурата от данни стек за пресмятане на стойност на аритметичен израз

Ще разглеждаме аритметични изрази от вида:

```
<израз> ::= <терм>|
            <израз>+<терм>|
            <израз>-<терм>
<терм> ::= <множител>|
            <терм>*<множител>|
            <терм>/<множител>
<множител> ::= <цифра>|<променлива>|(<израз>)|
              <множител> ^ <цифра>
<цифра> ::= 0|1| ... |9
<променлива> ::= <буква>
```

където ^ означава операцията степенуване.

Пресмятането на стойността на аритметичен израз се реализира по-просто, ако изразът е записан не в обичайния му инфиксен вид, а в т. нар. **обратен полски запис**. Характерно за този вид запис на израз е, че знакът за операция се записва след аргументите си.

Примери:

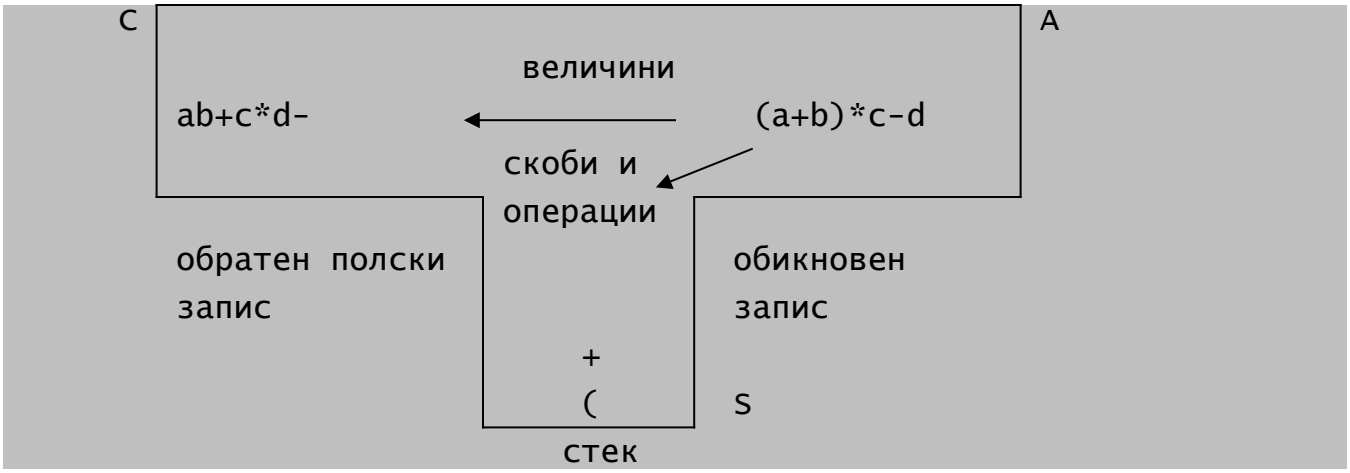
обикновен запис *обратен полски запис*

(a-b/c)*m	abc/-m*
a^n*b^m	an^bm^*
(a-b)*(a+b)	ab-ab+*
(a+b)*c-d*f+m^p	ab+c*df*-mp^+

Ако аритметичен израз е представен чрез обратен полски запис, пресмятането на стойността му се осъществява по следния начин. Сканира се изразът, представен чрез обратен полски запис отляво надясно до намиране на знак за операция. Пресмята се стойността на терма с аргументи – първите два, намиращи се непосредствено пред знака за операция. Знакът за операция и операндите се заменят с резултата от пресмятането, след което продължава търсенето на знак за операция. Сканирането продължава до достигане края на израза. За целта е удобно да се използва стек.

Преобразуване на израз от обикновен в обратен полски запис

Ще използваме стек. Фиг. 15.1 илюстрира преобразуването.



Фиг. 15.1 Преобразуване на аритметичен израз в обратен полски запис

Стекът е означен със S. Символите, участващи в обикновения запис се прехвърлят от частта A в частта C, като някои от тях временно престояват в стека S. В частта C се получава изразът, записан в обратен полски запис.

Правилата за движение са следните:

- Величините (цифри и променливи) се преместват директно от частта A в частта C.
- Скобата ‘(‘ се включва в стека S.
- Знаците за аритметични операции +, -, *, / и ^ се включват в стека S. Всяка операция има определен приоритет. Реализира се с цяло число, което се нарича тегло. Приемаме, че най-тежки са + и -, по-леки от тях са * и / и най-лека е операцията за степенуване. Ако при включване на знак за операция в стека S, под него има знак за операция с по-малко или с равно тегло, по-леката или с равно тегло операция се премества от S в частта C. Това се повтаря докато се достигне до по-тежка операция, до ‘)’ или до изпразването на стека.
- Скобата ‘)’ изключва от стека S всички знаци за операции до достигане до ‘(‘. Операциите се записват в частта C в реда на изключването им, а скобата ‘(‘ се унищожава от ‘)’.
- Ако всички символи от частта A са обработени, елементите на стека S, до изпразването му или до достигане до ‘(‘, се прехвърлят в областта C.

Задача 131. Да се напишат функции, реализиращи преобразуване на аритметичен израз от вида, описан по-горе, в обратен полски запис и също за намиране стойността на израз, представен в обратен полски запис.

Процедурата

```
void translate(char *s, char *ns);
```

превежда аритметичния израз, представен в обикновен запис чрез низа s, в обратен полски запис – низа ns. За да се избегне проверката за празен стек, в помощния стек още в началото включваме ‘(‘.

```
void translate(char *s, char *ns)
{stack<char> st;
 st.push('(');
 char x;
 int i = -1, j = -1, n = strlen(s);
```

```
while (i < n)
{i++;
 if (s[i] >= '0' && s[i] <= '9')
 {j++;
  ns[j] = s[i];
 }
 else
 if (s[i] == '(') st.push(s[i]);
 else
 if (s[i] == ')')
 {st.pop(x);
  while (x != '(')
  {j++;
   ns[j] = x;
   st.pop(x);
   }
  }
  else
  if (s[i] == '+' || s[i] == '-' ||
      s[i] == '*' || s[i] == '/' || s[i] == '^')
   {st.pop(x);
    while (x != '(' && t(x) <= t(s[i]))
    {j++;
     ns[j] = x;
     st.pop(x);
    }
    st.push(x);
    st.push(s[i]);
   }
  }
  st.pop(x);
  while (x != '(')
  {j++; ns[j] = x;
   st.pop(x);
  }
  j++;
```

```
      ns[j] = 0;
    }
```
Функцията t намира теглото на операциите, използвани в аритметичния израз, определен по-горе и има вида:
```
   int t(char c)
   {int p;
    switch (c)
    {case '+': p = 2; break;
     case '-': p = 2; break;
     case '*': p = 1; break;
     case '/': p = 1; break;
     case '^': p = 0; break;
     default:  p = -1;
    }
    return p;
   }
```
Функцията
```
   int value(char *s);
```
намира стойността на израза, като използва обратния полски запис (представен чрез низа s), който му съответства.
```
   int value(char *s)
   {stack<int> st;   // генерира стек st от цели числа
    int x, y, z;
    unsigned int i = 0, n = strlen(s);
    while (i < n)
    {if (s[i] >= '0' && s[i] <= '9') st.push((int)s[i] - (int)'0');
     else
      if (s[i] == '+' || s[i] == '-' || s[i] == '*'
                      || s[i] == '/' || s[i] == '^')
      {st.pop(y);
       st.pop(x);
       switch (s[i])
       {case '+': z = x+y; break;
        case '-': z = x-y; break;
        case '*': z = x*y; break;
        case '/': z = x/y; break;
```

```
        case '^': z = (int)pow(x,y);
       }
      st.push(z);
      }
      i++;
      }
      st.pop(z);
      return z;
     }
```

Тогава намирането на стойността на аритметичен израз може да се реализира чрез изпълнението на следната главна функция:

```
     void main()
     {char s[200];
      cout << "s: ";
      cin >> s;
      char s1[200];
      translate(s, s1);
      cout << value(s1);
      cout << endl;
     }
```

Задача 132. В масив е записан без грешка булев израз от вида:

<булев_израз> ::= t | f | (~<булев_израз>)|
(<булев_израз>*<булев_израз>)|
(<булев_израз>+<булев_израз>)

където t означава истина, f – лъжа, а ~, * и + означават съответно логическо отрицание, конюнкция и дизюнкция. Да се напише програма, която намира стойността на правилно записан в масив булев израз.

Функцията BulFormula решава задачата. Тя използва два помощни стека s1 и s2. Ако сканираният символ е знак за операция, се включва в стека s1, а ако е t или f - в стека s2. Затварящата скоба изключва знак за операция от стека s1, анализира го и в зависимост от вида на операцията – унарна или бинарна изключва един или два елемента от s2. Изпълнява се операцията над съответните аргументи и резултатът се включва в стека s2. Останалите символи се пропускат.

```
// Program Zad132.cpp
#include <iostream.h>
#include <string.h>
#include “stack-link”
bool BulFormula(char* s)
{stack<char> s1, s2;
 char c, x, y;
 int i = -1, n = strlen(s);
 while (i < n)
 {i++;
  if (s[i] == '~' || s[i] == '*' || s[i] == '+') s1.push(s[i]);
  else
  if (s[i] == 't' || s[i] == 'f') s2.push(s[i]);
  else
  if (s[i] == ')')
  {s1.pop(c);
   switch(c)
   {case '~': s2.pop(x);
              if (x == 't') c = 'f'; else c = 't';
              s2.push(c); break;
    case '*': s2.pop(y); s2.pop(x);
              if (x == 't' && y == 't') c = 't'; else c = 'f';
              s2.push(c); break;
    case '+': s2.pop(y); s2.pop(x);
              if (x == 'f' && y == 'f') c = 'f'; else c = 't';
              s2.push(c); break;
    }
  }
 }
  s2.pop(c);
  return c == 't';
 }
void main()
{char s[200];
 cout << "s: "; cin >> s;
 cout << BulFormula(s) << endl;
```

```
 cout << endl;
}
```

15.2 Опашка

15.2.1 Дефиниране на опашка

Логическо описание

Опашката е крайна редица от елементи от един и същ тип. Операцията включване е допустима за елементите от единия край на редицата, който се нарича **край на опашката**, а операцията изключване на елемент - само за елементите от другия край на редицата, който се нарича **начало на опашката**. Възможен е достъп само до елемента, намиращ се в началото на опашката, при това достъпът е пряк.

При тази организация на логическите операции, първият включен елемент се изключва пръв. Затова опашката се определя още като структура от данни "*пръв влязъл – пръв излязъл*".

Физическо представяне

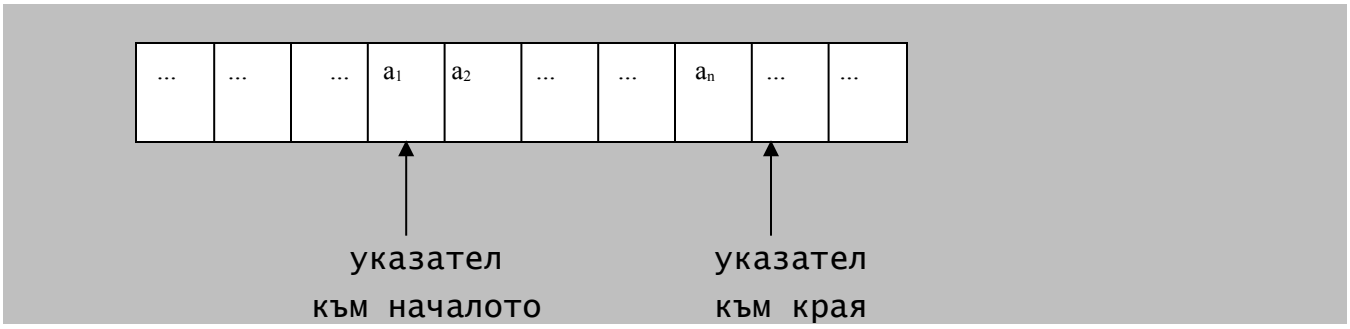
Широко се използват два основни начина за физическо представяне на опашка: *последователно* и *свързано*.

Последователно представяне

При това представяне се запазва блок от паметта, вътре в който опашката да расте и да се съкращава. Ако редицата от елементи от един и същ тип

$a_1, a_2, ..., a_n$

е опашка с начало a_1 и край a_n, последователното представяне има вида:

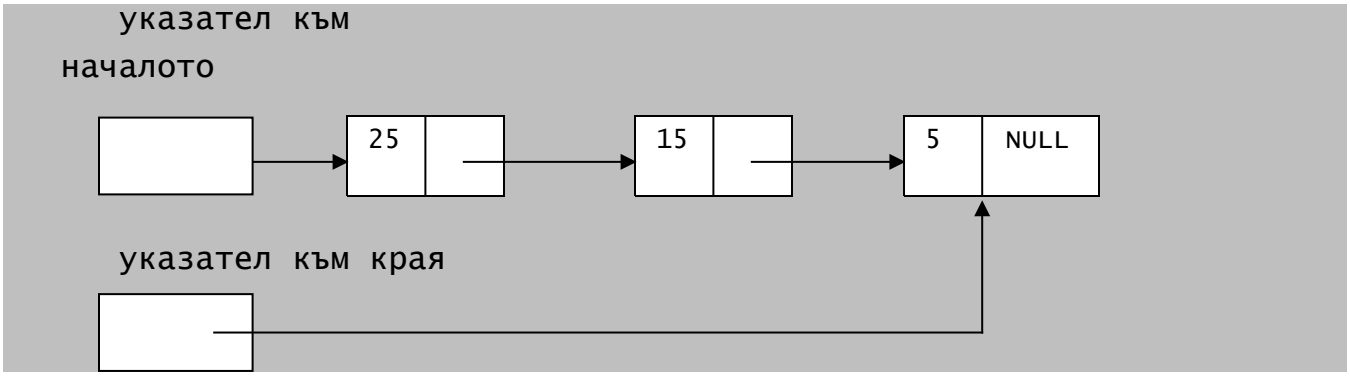


Фиг. 15.2 Последователно представяне на опашка

Включването на елемент в опашката се осъществява чрез поместването му в последователни адреси в неизползваната част веднага след края на опашката. При изчерпване на масива, ако има освободена памет в началото му, включването се извършва там.

Свързано представяне

Това представяне е аналогично на свързаното представяне на стек. За удобство, при реализирането на операцията включване, се въвежда указател и към края на опашката.



Фиг. 15.3 Свързано представяне на опашка с три елемента

15.2.2 Реализация на опашка

15.2.2.1 Реализация на последователното представяне

Ще използваме динамичен масив с размер size. С front означаваме указателя към началото, с rear – указателя към края му, а с n – текущия брой на елементите на опашката. Ще дефинираме шаблон на клас, реализиращ последователно представяне на опашка.

```
const size = 100;
template <class T>
class queue
{public:
  queue();                    // конструктор
  ~queue();                   // деструктор
```

```
  queue(queue const &);       // конструктор за присвояване
  queue& operator=(queue const &);  // операторна функция за
                                    // присвояване
  void InsertElem(T const &);  // включване на елемент в опашка
  int DeleteElem(T &);        // изключване на елемент от опашка
  void print();              // извеждане на опашка
 private:
  int front, rear, n;
  T *a;
  void delqueue();               // изтриване на опашка
  void copy(queue const &);      // копиране на опашка
};
template <class T>
queue<T>::queue()
{a = new T[size];
 n = 0;
 front = 0;
 rear = 0;
}
template <class T>
queue<T>::~queue()
{delqueue();
}
template <class T>
queue<T>::queue(queue<T> const& r)
{copy(r);
}
template <class T>
queue<T>& queue<T>::operator=(queue<T> const& r)
{if (this != &r)
{delqueue();
 copy(r);
}
return *this;
}
template <class T>
```

```
void queue<T>::InsertElem(T const & x)
{if (n == size) cout << "Impossible! \n";
 else
 {a[rear] = x;
  n++; rear++;
  rear = rear % size;
 }
}
template <class T>
int queue<T>::DeleteElem(T &x)
{if (n > 0)
 {x = a[front]; n--; front++;
  front = front % size;
  return 1;
 }
 else return 0;
}
template <class T>
void queue<T>::delqueue()
{delete [] a;
}
template <class T>
void queue<T>::copy(queue<T> const &r)
{a = new T[size];
 for (int i = 0; i < size; i++)
    a[i] = r.a[i];
 n = r.n;
 front = r.front;
 rear = r.rear;
}
template <class T>
void queue<T>::print()
{T x;
 while (DeleteElem(x))
   cout << x << "  ";
 cout << endl;
```

```
}
```

Включваме този шаблон във файл с име queue.cpp. Следва програма, която използва дефинирания шаблон.

Забележете, файлът, съдържащ реализацията на шаблона на класа queue, е включен в програмата като заглавен файл, но е ограден в кавички, а не в “<” и “>” както е при системните заглавни файлове.

```
#include <iostream.h>
#include “queue.cpp”
void main()
{queue<int> q; // създава опашка от цели числа
 for (int i = 1; i <= 10; i++)   // запълва опашката
   q.InsertElem(i);              // с целите числа от 1 до 10
 q.print();                      // извежда опашката q
 queue<char> p;                  // създава опашка от символи
 for (char c = ’a’; i <= ’z’; i++)  // запълва опашката
    p.InsertElem(c);                // с малките латински букви
 p.print();                      // извежда опашката p
}
```

15.2.2.2 Реализация на свързаното представяне

Ще реализираме шаблон на клас queue, реализиращ свързаното представяне на опашка. Двойката

inf	link

ще реализираме чрез шаблона на структурата

```
template <class T>
struct elem
{T inf;
 elem* link;
};
```

а опашката – чрез шаблона на класа queue. Указателят към началото на опашката ще означим с front, а този към края – с rear. Отново се налага реализирането на голямата тройка (деструктор, конструктор за

присвояване и операторна функция за присвояване). Реализицията на това представяне има вида:

```
template <class T>
struct elem
{T inf;
 elem* link;
};
template <class T>
class queue
{public:
  queue();
  ~queue();
  queue(queue const &);
  queue& operator=(queue const &);
  void InsertElem(T const&);
  int DeleteElem(T &);
  void print();
  bool empty() const;
 private:
  elem<T> *front, *rear;
  void delqueue();
  void copy(queue const&);
};
template <class T>   // конструктор
queue<T>::queue()
{front = NULL;
 rear = NULL;
}
template <class T>   // деструктор
queue<T>::~queue()
{delqueue();
}
template <class T>   // конструктор за присвояване
queue<T>::queue(queue const & r)
{copy(r);
```

```
}
template <class T>   // операторна функция за присвояване
queue<T>& queue<T>::operator=(queue const& r)
{if (this != &r)
 {delqueue();
  copy(r);
 }
 return *this;
}
template <class T>   // изтриване на опашка
void queue<T>::delqueue()
{T x;
 while (DeleteElem(x));
}
template <class T>      // копиране на опашка
void queue<T>::copy(queue const & r)
{rear = NULL;
 if (r.rear)
 {elem<T> *p = r.front;
  while (p)
  {InsertElem(p->inf);
   p = p->link;
  }
 }
}
template <class T>      // включване на елемент в опашка
void queue<T>::InsertElem(T const& x)
{elem<T> *p = new elem<T>;
 p->inf = x;
 p->link = NULL;
 if (rear) rear->link = p;
 else front = p;
 rear = p;
}
template <class T>   // изтриване на елемент от опашка
int queue<T>::DeleteElem(T & x)
```

```
{elem<T> *p;
 if (!rear) return 0;
 p = front;
 x = p->inf;
 if (p == rear)
 {rear = NULL;
  front = NULL;
 }
 else front = p->link;
 delete p;
 return 1;
}
template <class T>      // извеждане на опашка
void queue<T>::print()
{T x;
 while (DeleteElem(x))
   cout << x << " ";
 cout << endl;
}
template <class T>      // проверка за празна опашка
bool queue<T>::empty() const
{return rear == NULL;
}
```

Ще запишем този шаблон във файл с име queue-link.cpp, след което ще го включим в демонстрационна програма.

Задача 133. Да се напише програма, която създава и извежда опашка от опашки от цели числа.

Налага се предефиниране на член-функцията print() на шаблона на класа, тъй като основната член-функция print() извежда чрез използване на <<, с което не може да се изведе опашка.

```
// Program Zad133.cpp
#include <iostream.h>
#include “queue-link.cpp”
```

```
// дефиниране на клас IntQueue, реализиращ опашка от цели числа
typedef queue<int> IntQueue;
// дефиниране на клас QueueQueue, реализиращ опашка
// от опашки от цели числа
typedef queue<IntQueue> QueueQueue;
// предефиниране на шаблонната член-функция print()
// за извеждане на опашка от опашки.
void queue<IntQueue>::print() // специализация на член-функцията
{IntQueue x;                  // print() за извеждане на
 while (DeleteElem(x))        // опашка от опашки
   x.print();
 cout << endl;
}
void main()
{QueueQueue qq; // qq е обект, означаващ опашка от опашки
 for (int i = 1; i <= 5; i++)
  {IntQueue q;    // конструиране на опашка от цели числа
   for (int j = i; j <= 2*i; j++)
    q.InsertElem(j);
   qq.InsertElem(q); // включване на опашката q в опашката qq
  }
 qq.print();
}
```

Задачи върху структурата от данни опашка

Задача 134. Да се напише програма, която само с едно преминаване през елементите на масив от числа (без използване на допълнителни масиви) извежда вурху екрана елементите на масива в следния ред: отначало всички числа, които са по-малки от a, след това всички числа в интервала [a, b] и накрая всички останали числа, запазвайки техния първоначален ред (a и b са дадени числа, a < b).

Програма Zad134.cpp решава задачата. Тя използва две опашки q1 и q2, в които записва числата от интервала [a, b] и тези – по-големи от b, съответно в реда на тяхното срещане в масива arr.

```
// Program Zad134.cpp
#include <iostream.h>
#include "queue-link.cpp"
typedef queue<int> IntQueue;
void read(int n, int *a)
{for (int i = 0; i < n; i++)
{cout << "a[" << i << "]= ";
 cin >> a[i];
}
}
void main()
{int arr[100];
 int n;
 do
 {cout << "n= ";
  cin >> n;
 } while (n < 1 || n > 100);
 read(n, arr);
 cout << "a < b = ";
 int a, b;
 cin >> a >> b;
 IntQueue q1, q2;
 for (int i = 0; i < n; i++)
    if (arr[i] < a) cout << arr[i] << " ";
    else if (arr[i] <= b) q1.InsertElem(arr[i]);
    else q2.InsertElem(arr[i]);
  q1.print();
  q2.print();
}
```

Задача 135. Да се напише програма, която създава две опашки, елементите на които са структури, съдържащи име и възраст на човек. Сортира във възходящ ред по възраст елементите на опашките, след което ги слива и извежда получената опашка.

Програма Zad135.cpp решава задачата. За олесняване на сливането в края на всяка сортирана опашка добавяме фиктивен елемент, който се нарича **сентинел**. Този трик често се използва при работа с линейни динамични структури.

```
// Program Zad135.cpp
#include <iostream.h>
#include "queue-link.cpp"
struct people
{char name[31];
 int age;
};
typedef queue<people> pqueue;
void queue<people>::print() // предефиниция на шаблонната функция
{people x;
 while (DeleteElem(x))
 cout << x.name << " " << x.age << endl;
}
// за опашката q се намират лицето с най-малка възраст (min) и
// опашката без лицето с най-малка възраст (newq)
// newq е инициализирана с празната опашка
void minqueue(pqueue q, people & min, pqueue &newq)
{people x;
 q.DeleteElem(min);
 while (q.DeleteElem(x))
  if (x.age < min.age)
  {newq.InsertElem(min);
   min = x;
  }
  else newq.InsertElem(x);
 }
// сортиране на опашката (q) във възходящ ред
// по възраст на лицата (nq)
void sortqueue(pqueue q, pqueue& nq)
{while (!q.empty())
 {people min;
```

```
  pqueue q1;
  minqueue(q, min, q1); // q1 е инициализирана с празната опашка
  nq.InsertElem(min);
  q = q1;
 }
}
// сливане на сортираните опашки p и q
// връща резултата от сливането
pqueue merge(pqueue p, pqueue q)
{people x = {"xxx", -1}, y = {"yyy", -1}; // фиктивни елементи
 p.InsertElem(x); // включване на фиктивния елемент x в опашката p
 q.InsertElem(y); // включване на фиктивния елемент y в опашката q
 pqueue r;
 p.DeleteElem(x);
 q.DeleteElem(y);
 while (!p.empty() && !q.empty())
   if (x.age <= y.age)
   {r.InsertElem(x);
    p.DeleteElem(x);
   }
   else
   {r.InsertElem(y);
    q.DeleteElem(y);
   }
 if (!p.empty()) // q е празна
    do
      r.InsertElem(x);
    while (p.DeleteElem(x) && x.age != -1);
 else // p е празна
    do
      r.InsertElem(y);
    while (q.DeleteElem(y) && y.age != -1);
 return r;
}
void main()
{people s;
```

```
    pqueue q1;
    int i, n;
    cout << "First Queue:\n n= ";
    cin >> n;
    for (i = 0; i < n; i++)
    {cout << "Name: ";
     cin>> s.name;
     cout << "Age: ";
     cin >> s.age;
     q1.InsertElem(s);
    }
    pqueue q2;
    cout << "Second Queue \n n= ";
    cin >> n;
    for (i = 0; i < n; i++)
    {cout << "Name: ";
     cin>> s.name;
     cout << "Age: ";
     cin >> s.age;
     q2.InsertElem(s);
    }
    pqueue p, q, r;
    sortqueue(q1, p);
    sortqueue(q2, q);
    r = merge(p, q);
    r.print();
   }
```

Задача 136. Да се напише програма, която създава опашка от опашки, елементите на които са структури, съдържащи име и възраст на човек. Да се сортират по възраст компонентите на опашката, след което да се слеят.

Програма Zad136.cpp решава задачата. Някои функции в нея са пропуснати, тъй като са същите като в задача 135.

```
// Program Zad136.cpp
```

```
#include <iostream.h>
#include "queue-link.cpp"
struct people
{char name[31];
 int age;
};
typedef queue<people> pqueue;
void queue<people>::print()
{people x;
 while (DeleteElem(x))
    cout << x.name << " "
         << x.age << endl;
}
void queue<pqueue>::print()
{pqueue x;
 while (DeleteElem(x))
   {x.print();
    cout << endl;
   }
}
void minqueue(pqueue q, people & min, pqueue &newq)
...
void sortqueue(pqueue q, pqueue& nq)
...
pqueue merge(pqueue p, pqueue q)
...
void main()
{queue<pqueue> q2, q3;
 int m;
 cout << “m= “; cin >> m;
 for (int j = 1; j <= m; j++)
 {int i, n;
  pqueue q1;
  cout << j << " Queue:\n n= ";
  cin >> n;
  for (i = 0; i < n; i++)
```

```
 {people s;
  cout << "Name: ";
  cin >> s.name;
  cout << "Age: ";
  cin >> s.age;
  q1.InsertElem(s);
 }
q2.InsertElem(q1);
}
pqueue r;
while (q2.DeleteElem(r))
{pqueue t;
 sortqueue(r, t);
 q3.InsertElem(t);
}
pqueue p1, p2;
q3.DeleteElem(p1);
while (q3.DeleteElem(p2))
   p1 = merge(p1, p2);
p1.print();
}
```

15.3 Свързан списък

При стековете и опашките единствният начин за извличане на данни от структурата се осъществява чрез отсраняване на елементи. Често се налага да се използват всички данни от редица от елементи без да се отстраняват елементите от редицата. За целта се използва структурата от данни свързан списък.

15.3.1 Дефиниране на свързан списък

Логическо описание

Свързаният списък е крайна редица от елементи от един и същ тип. Операциите включване и изключване са допустими в произволно място на

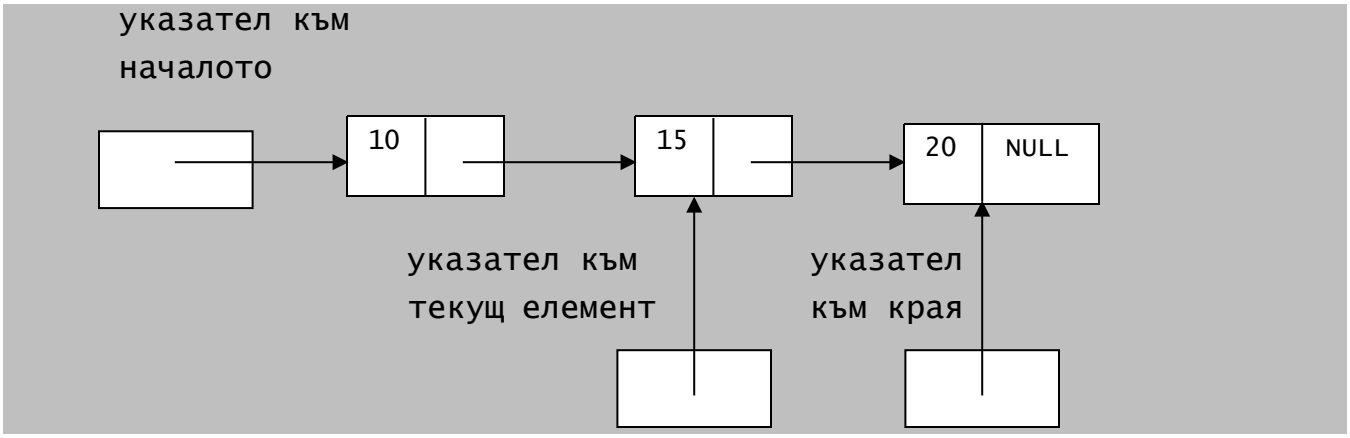
редицата. Възможен е пряк достъп до елемента в единия край на редицата, наречен **начало на списъка**, и последователен до всеки от останалите елементи.

Физическо представяне

Има два основни начина за представяне на свързания списък в паметта на компютъра: *свързано представяне с една* и *свързано представяне с две връзки*. Възможно е и *последователно представяне* (чрез масив от структури), което не се използва напоследък заради добре развитите средства за динамично разпределение на паметта.

Свързано представяне с една връзка

Използва се представяне, аналогично на свързаното представяне на стек и опашка (Фиг. 15.4). За удобство, при реализирането на операциите включване, изключване и обхождане, се въвеждат и указатели към края и към текущ елемент на списъка.

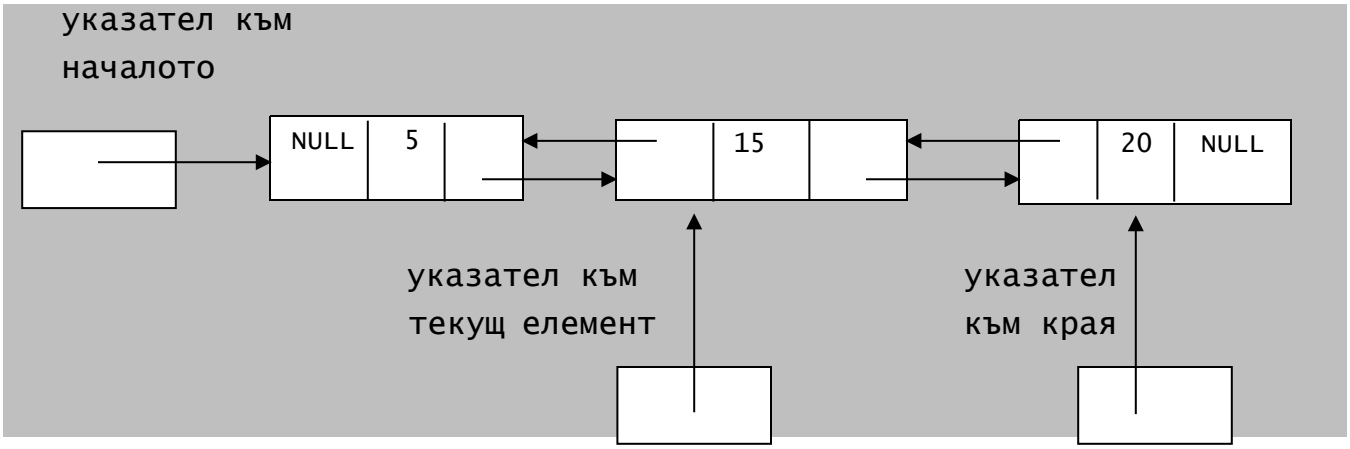


Фиг. 15.4 Представяне на свързан списък с една връзка

Свързано представяне с две връзки

За удобство, при реализирането на операциите включване, изключване и обхождане, се въвеждат тройни кутии, с едно информационно и две свързващи полета, съдържащи текущия елемент и адресите на

предшестващия и следващия го елементи на свързания списък (Фиг. 15. 5).



Фиг. 15. 5 Представяне на свързан списък с две връзки

15.3.2 Реализация на представяне на свързан списък с една връзка

Следният шаблон на клас реализира това представяне на свързан списък.

```
template <class T>
struct elem
{T inf;
 elem *link;
};
template <class T>
class LList
{public:
  LList();
  ~LList();
  LList(LList const&);
  LList& operator=(LList const &);
  void print();
  void IterStart(elem<T>* = NULL);
  elem<T>* Iter();
  void ToEnd(T const &);
```

```
    void InsertAfter(elem<T> *, T const &);
    void InsertBefore(elem<T> *, T const &);
    int DeleteAfter(elem<T> *, T &);
    int DeleteBefore(elem<T> *, T &);
    void DeleteElem(elem<T> *, T &);
    int len();
    void concat(LList const&);
    void reverse();
   private:
    elem<T> *Start,       // указател към началото
            *End,         // указател към края
            *Current;     // указател към текущ елемент
    void DeleteList();
    void CopyList(LList<T> const &);
  };
```

Конструктора и член-функциите на голямата тройка няма да коментираме. Реализират аналогични идеи на тези при стека и опашката.

```
  template <class T>   // конструктор
  LList<T>::LList()
  {Start = NULL;
   End = NULL;
  }
  template <class T>   // деструктор
  LList<T>::~LList()
  {DeleteList();
  }
  template <class T>   // конструктор за присвояване
  LList<T>::LList(LList<T> const& r)
  {CopyList(r);
  }
  template <class T>   //операторна функция за присвояване
  LList<T>& LList<T>::operator=(LList<T> const & r)
  {if (this != &r)
   {DeleteList();
    CopyList(r);
   }
```

```
 return *this;
}
```

Следващите две функции са помощни и реализират изтриване на свързан списък и копиране на свързан списък на друго място в паметта. Използват се за реализиране на голямата тройка и затова са капсолирани в private-частта на класа. Макар, че могат да се реализират с помощта на член-функции на шаблона, по-добра е реализацията им без тях – чрез обхождане на елементите им.

Изтриване на списък

```
template <class T>
void LList<T>::DeleteList()
{elem<T> *p;
 while (Start)
 {p = Start;
  Start = Start->link;
  delete p;
 }
 End = NULL;
}
```

Следващите реализации на тази член-функция на шаблона използват член-функциите DeleteAfter, Delete Before и DeleteElem.

```
template <class T>
void LList<T>::DeleteList()
{if (End)
 {T x;
  while (DeleteAfter(Start, x));
  DeleteElem(Start, x);
 }
}
```

и

```
template <class T>
void LList<T>::DeleteList()
{if (Start)
 {T x;
  while (DeleteBefore(End, x));
  DeleteElem(Start, x);
```

```
 }
}
```

Копиране на списък

```
template <class T>
void LList<T>::CopyList(LList<T> const & r)
{elem<T> *p = r.Start, *q;
 if (r.Start)
 {Start = new elem<T>;
  End = Start;
  while (p)
  {End->inf = p->inf;
   End->link = NULL;
   p = p->link;
   if (p)
   {q = End;
    End = new elem<T>;
    q->link = End;
   }
  }
 }
}
```

Друга, по-лесна реализация на тази член-функция на шаблона е:

```
template <class T>
void LList<T>::CopyList(LList<T> const & r)
{Start = End = NULL;
 if (r.Start)
 {elem<T> *p = r.Start;
  while (p)
  {ToEnd(p->inf);
   p = p->link;
  }
 }
}
```

и я избираме за реализация на копирането.

Извеждане на елементите на списък

Извеждането на елементите на свързан списък се реализира чрез последователното им обхождане, без разрушаването на списъка.

```
template <class T>
void LList<T>::print()
{elem<T> *p = Start;
 while (p)
 {cout << p->inf << " ";
  p = p->link;
 }
 cout << endl;
}
```

Член-функциите IterStart и Iter реализират операциите с т. нар. **итератори**.

Итераторът е абстракция на означението указател към елемент на редица или по-точно може да се смята за указател към елемент на контейнер (стекът, опашката, свързаният списък са контейнери). Всеки конкретен итератор е обект (в широкия смисъл на думата) от някакъв тип. Разнообразието на типове води до разнообразие на итераторите. В някои случаи итераторите са почти обикновени указатели към обекти, в други – са указател, снабден с индекс и т.н. В случая на свързан списък итераторът е указател към двойна или тройна кутия. Общото на всички итератори е тяхната семантика и имената на техните операции. Обикновено операциите са:

```
++ -  приложена към итератор, намира итератор,
      който сочи към следващия елемент;
-- -  приложена към итератор, намира итератор,
      който сочи към предшестващия елемент;
*  -  намира елемента, към който сочи итераторът.
```

В шаблона на класа LList итераторът е означен с Current. Операцията ++ е реализирана чрез член-функцията Iter. За получаване на указател към началото на свързан списък, е използвана процедурата IterStart. Тя е с един подразбиращ се параметър. Подразбиращата се стойност е NULL. Обръщението IterStart() установява указателя Current в началото на текущия списък. В случай, че е указан ненулев параметър, Current се установява в него. Операцията * не е

реализирана, тъй като Current сочи елемент, който е реализиран като структура, но обръщението Iter()->inf я реализира.

Установяване на итератора в началото

```
template <class T>
void LList<T>::IterStart(elem<T> *p)
{if (p) Current = p;
 else Current = Start;
}
```

Преместването на указателя Current в следващата позиция се осъществява чрез член-функцията Iter.

Установяване на итератора в следващата позиция

```
template <class T>
elem<T>* LList<T>::Iter()
{elem<T> *p = Current;
 if (Current) Current = Current->link;
 return p;
}
```

Като използваме член-функциите, реализиращи основните операции за работа с итератора, член-функцията print() на шаблона LList може да се реализира и по следния начин:

```
template <class T>
void LList<T>::print()
{IterStart();
 while (Iter())
 {cout << Iter()->inf << " ";
 }
 cout << endl;
}
```

Включването на елемент в свързан списък реализираме чрез следните три член-функции:

Включване на елемент в края на списъка

```
template <class T>
void LList<T>::ToEnd(T const & x)
```

```
{Current = End;
 End = new elem<T>;
 End->inf = x;
 End->link = NULL;
 if (Current) Current->link = End;
 else Start = End;
}
```

Включване на елемент след указан елемент

```
template <class T>
void LList<T>::InsertAfter(elem<T> *p, T const & x)
{elem<T> *q = new elem<T>;
 q->inf = x;
 q->link = p->link;
 if (p == End) End = q;
 p->link = q;
}
```

Включване на елемент пред указан елемен

```
template <class T>
void LList<T>::InsertBefore(elem<T> * p, T const& x)
{elem<T> *q = new elem<T>;
 *q = *p;
 if (p == End) End = q;
 p->inf = x;
 p->link = q;
}
```

Изтриването на елемент от свързан списък реализираме чрез следните член-функции:

Изтриване на елемент след указан елемент

```
template <class T>
int LList<T>::DeleteAfter(elem<T> *p, T &x)
{if (p->link)
{elem<T> *q = p->link;
 x = q->inf;
 p->link = q->link;
```

```
 if (q == End) End = p;
 delete q;
 return 1;
 }
 else return 0;
}
```

Изтриване на указан елемент

```
template <class T>
void LList<T>::DeleteElem(elem<T> *p, T &x)
{if (p == Start)
 {x = p->inf;
  if (Start == End)
  {Start = End = NULL;
  }
  else Start = Start->link;
  delete p;
 }
 else
 {elem<T> *q = Start;
  while (q->link != p) q = q->link;
  DeleteAfter(q, x);
 }
}
```

Изтриване на елемент пред указан елемент

```
template <class T>
int LList<T>::DeleteBefore(elem<T> *p, T &x)
{if (p != Start)
 {elem<T> *q=Start;
  while (q->link != p) q = q->link;
  DeleteElem(q, x);
  return 1;
 }
 else return 0;
}
```

Дължина на списък

Член функцията len намира броя на елементите на свързан списък.

```
template <class T>
int LList<T>::len()
{int n = 0;
 IterStart();
 elem<T> *p = Iter();
 while (p)
 {n++;
  p = Iter();
 }
 return n;
}
```

или

```
template <class T>
int LList<T>::len()
{int n = 0;
 elem<T> *p = Start;
 while (p)
 {n++;
  p = p->link;
 }
 return n;
}
```

Конкатенация на списъци

Следващата член-функция реализира конкатенация на неявния свързан списък с указания като формален параметър. В резултат в края на неявния списък са включени елементите на указания списък. Така неявният списък е разрушен (носи резултата).

```
template <class T>
void LList<T>::concat(LList<T> const &L)
{elem<T> *p = L.Start;
 while (p)
 {ToEnd(p->inf);
  p = p->link;
 }
}
```

Често пъти вместо тази функция се използва следната:

```
template <class T>
void LList<T>::concat(LList const& L)
{End->link = L.Start;
}
```

Тя е неправилна, тъй като една и съща редица от елементи е достъпна чрез два обекта на класа LList, т.е. имат поделена част.

Обръщане на елементите на списък

Следващата член-функция на класа LList обръща елементите на неявния параметър като ползва помощен списък, с който работи като със стек.

```
template <class T>
void LList<T>::reverse()
{LList<T> l;
 IterStart();
 elem<T> *p = Iter();
 if (p)
 {l.ToEnd(p->inf);
  p = p->link;
  while (p)
  {l.InsertBefore(l.Start, p->inf);
   p = p->link;
  }
 }
 *this = l;
}
```

Реализацията на reverse, дадена по-долу, обръща елементите на непразен списък, зададен чрез неявния параметър без да прави негово копие в паметта. Тя поправя връзките в него така, че първите елементи да станат последни и обратно.

```
template <class T>
void LList<T>::reverse()
{elem<T> *p, *q, *temp;
 p = Start;
 if (p)
 {q = NULL;
```

```
  temp = Start;
  Start = End;
  End = temp;
  while (p != Start)
  {temp = p->link;
   p->link = q;
   q = p;
   p = temp;
  }
  p->link = q;
 }
}
```

15.3.3 Приложения на свързните списъци, представени с една връзка

Шаблонът на класа LList записваме във файла L-List.cpp.

Общи приложения

Задача 137. Да се напише програма, която създава свързан списък с една връзка по различни начини, обръща елементите на списъка и ги извежда.

```
// Program Zad137.cpp
#include <iostream.h>
#include "L-List.cpp"
void main()
{LList<int> l1, l2;
 // създава списъка l1 с елементи 1, 2, 3, 4
 l1.ToEnd(1); l1.ToEnd(2);
 l1.ToEnd(3); l1.ToEnd(4);
 // създава списъка l2 с елементи 5, 6
 l2.ToEnd(5); l2.ToEnd(6);
 // създава списъка l3 чрез конструктора за присвояване
 LList<int> l3 = l1;
 // извежда списъците
```

```
 cout << "l1= "; l1.print();
 cout << "l2= "; l2.print();
 cout << "l3= "; l3.print();
 // обръща елементите на списъка l1
 l1.reverse();
 cout << "L1-rev: "; l1.print();
}
```

Задача 138. Да се напише програма, която като използва класа LList моделира стек.

```
// Program Zad138.cpp
#include <iostream.h>
#include "L-List.cpp"
void main()
{// създаване на стека от цели числа 9,8,..., 0 с връх 9
 LList<int> l1;
 l1.ToEnd(0);
 l1.IterStart();
 elem<int>* p = l1.Iter();
 for (int i = 1; i <= 9; i++)
   l1.InsertBefore(p, i);
 // изключване на елементи от стека
 l1.IterStart();
 p = l1.Iter();
 int x;
 l1.DeleteElem(p, x);
 cout << x << " ";
 l1.IterStart();
 p = l1.Iter();
 l1.DeleteElem(p, x);
 cout << x << " ";
}
```

Задача 139. Да се напише програма, която като използва класа LList моделира опашка.

```
// Program Zad139.cpp
#include <iostream.h>
#include "L-List.cpp"
void main()
{LList<int> l1;
 l1.ToEnd(1);
 l1.IterStart();
 elem<int>* p = l1.Iter();
 for (int i = 2; i <= 10; i++)
   l1.ToEnd(i);
 l1.print();
 l1.IterStart();
 p = l1.Iter();
 int x;
 l1.DeleteElem(p, x);
 cout << "x= " << x << endl;
 l1.IterStart();
 p = l1.Iter();
 l1.DeleteElem(p, x);
 cout << "x= " << x << endl;
 l1.print();
}
```

Задача 140. Да се дефинира шаблон на функция, която реализира увеличаване на елементите на свързан списък от тип T с елемента a от тип T (T е числов тип).

```
template <class T>
LList<T> increase(LList<T> L, T a)
{elem<T> *p;
 L.IterStart();
 p = L.Iter();
 while (p)
 {p->inf = p->inf + a;
  p = L.Iter();
 }
```

```
 return L;
}
```

Задача 141. Даден е списък от цели числа. Да се напише функция, която:

а) изтрива първото срещане на дадено цяло число;

б) изтрива всяко срещане на дадено цяло число.

Ще направим следната специализация на класа LList.

```
typedef LList<int> IntList;
а)
void deletefirst(int a, IntList& l)
{int x;
 l.IterStart();
 elem<int> *p = l.Iter();
 while (p && p->inf != a) p = l.Iter();
 if (p->inf == a) l.DeleteElem(p, x);
}
```

или

```
void deletefirst(int a, IntList& l)
{l.IterStart();
 elem<int> *p = l.Iter();
 while (p)
 if(p->inf == a)
 {int x;
  l.DeleteElem(p, x);
  p = NULL;
 }
 else p = l.Iter();
}
б)
void deleteall(int a, IntList& l)
{int x;
 l.IterStart();
 elem<int> *p = l.Iter();
 while (p)
```

```
    {if (p->inf == a) l.DeleteElem(p, x);
     p = l.Iter();
    }
   }
```
или
```
   void deleteall(int a, IntList& l)
   {while (member(a, l))
      deletefirst(a, l);
   }
```
където member(a, l) е функция, проверяваща дали a принадлежи на списъка l. Една нейна дефиниция е дадена в следващата задача.

Някои рекурсивни функции за работа със списъци с една връзка

Задача 142. Даден е списък от цели числа. Да се напише рекурсивна функция, която:

а) проверява дали дадено цяло число се съдържа в списъка;
б) намира максималния елемент на списъка;
в) изтрива от списъка първото срещане на дадено цяло число;
г) изтрива от списъка всяко срещане на дадено цяло число;
д) извежда в обратен ред елементите на списъка.

Ще използваме следната специализация на класа LList.
```
   typedef LList<int> IntList;
```
а)
```
   bool member(int x, IntList l)
   {l.IterStart();
    elem<int> *p = l.Iter();
    if (!p) return false;
    int y;
    l.DeleteElem(p, y);
    return x == y || member(x, l);
   }
```
б)
```
   int maxelem(IntList l)
   {int x;
```

```
    l.IterStart();
    elem<int> *p = l.Iter();
    if (!p->link) return p->inf;
    l.DeleteElem(p, x);
    int y = maxelem(l);
    if (x >= y) return x;
    else return y;
   }
в)
   void deletefirst(int a, IntList &l)
   {l.IterStart();
    elem<int>*p = l.Iter();
    if (p)
    {int x;
     l.DeleteElem(p, x);
     if (x == a) return;
     else
     {deletefirst(a, l);
      l.IterStart(); p = l.Iter();
      if (p) l.InsertBefore(p, x);
      else l.ToEnd(x);
     }
    }
   }
г)
   void deleteall(int a, IntList &l)
   {l.IterStart();
    elem<int>*p = l.Iter();
    if (p)
    {int x;
     l.DeleteElem(p, x);
     if (x == a) deleteall(a, l);
     else
     {deleteall(a, l);
      l.IterStart();
      p = l.Iter();
```

```
      if (p) l.InsertBefore(p, x);
      else l.ToEnd(x);
     }
    }
   }
д) void print_reverse(IntList l)
   {int x;
    l.IterStart();
    elem<int> *p = l.Iter();
    if (p)
    {l.DeleteElem(p, x);
     print_reverse(l);
     cout << x << " ";
    }
   }
```

Сортиране и сливане на списъци

Пряка селекция

Задача 143. Да се дефинира шаблон на функция, който реализира метода на пряката селекция за сотриране на списъци с елементи от тип T, допускащи сравнение.

```
   template <class T>
   void sortlist(LList<T> &l)
   {elem<T>* mp, *p;
    l.IterStart(); p = l.Iter();
    while (p->link)
    {T min = p->inf;
     mp = p;
     elem<T> *q = p->link;
      while (q)
      {if (q->inf <= min)
      {mp = q;
       min = q->inf;
      }
```

```
     q = q->link;
    }
    min = mp->inf;
    mp->inf = p->inf;
    p->inf = min;
    p = p->link;
   }
}
```

Сливане на сортирани свързани списъци

Задача 144. Да се дефинира шаблон на функция mergelists, която реализира сливане на списъците l1 и l2 от тип T и връща резултата от сливането.

```
template <class T>
LList<T> mergelists(LList<T> l1, LList<T> l2)
{LList<T> l;
 l1.IterStart();
 l2.IterStart();
 elem<T> *p = l1.Iter(),
         *q = l2.Iter();
 while (p && q)
 if (p->inf <= q->inf)
 {l.ToEnd(p->inf);
  p = l1.Iter();
 }
 else
 {l.ToEnd(q->inf);
  q = l2.Iter();
 }
 if (q)
 while (q)
 {l.ToEnd(q->inf);
  q = l2.Iter();
 }
```

```
 else while(p)
 {l.ToEnd(p->inf);
  p = l1.Iter();
 }
 return l;
}
```

Задача 145. Да се напише програма, която създава списък от списъци от имена на хора, сортира компонентите на списъка, след което ги слива и извежда получения списък.

```
Програма Zad145.cpp решава задачата.
Program Zad145.cpp
#include <iostream.h>
#include <string.h>
#include "L-List.cpp"
typedef char str[31];
// дефиниране на клас strlist,
// реализиращ списък от имена на хора
typedef LList<str> strlist;
// дефиниране на клас list_of_lists,
// реализиращ списък от списъци от имена на хора
typedef LList<strlist> list_of_lists;
// специализация на член-функцията print() на шаблона
// за извеждане на списък от списъци от имена на хора
void LList<strlist>::print()
{elem<strlist> *p = Start;
 while (p)
 {p->inf.print();
  p = p->link;
 }
}
// специализация на член-функцията за включване на
// елемент в края на свързан списък
void LList<str>::ToEnd(str const & x)
{Current = End;
```

```
 End = new elem<str>;
 strcpy(End->inf, x);
 End->link = NULL;
 if (Current) Current->link = End;
 else Start = End;
}
// предефиниране на нова sortlist
// за сортиране на списък от хора
void sortlist(strlist &l)
{elem<str>* mp, *p;
 l.IterStart(); p = l.Iter();
 while (p->link)
 {str min;
  mp = p;
  strcpy(min, p->inf);
  elem<str> *q = p->link;
  while (q)
  {if (strcmp(q->inf, min) <= 0)
  {mp = q;
   strcpy(min, q->inf);
  }
   q = q->link;
  }
   strcpy(min, mp->inf);
   strcpy(mp->inf, p->inf);
   strcpy(p->inf, min);
   p = p->link;
  }
}
// предефиниране на mergelists за сливане на
// сортирани списъци от имена на хора
strlist mergelists(strlist l1, strlist l2)
{strlist l;
 l1.IterStart();
 l2.IterStart();
 elem<str> *p = l1.Iter(),
```

```
           *q = l2.Iter();
 while (p && q)
    if (strcmp(p->inf, q->inf) <= 0)
    {l.ToEnd(p->inf);
     p = l1.Iter();
    }
    else
    {l.ToEnd(q->inf);
     q = l2.Iter();
    }
 if (q)
    while (q)
    {l.ToEnd(q->inf);
     q = l2.Iter();
    }
else
   while (p)
   {l.ToEnd(p->inf);
    p = l1.Iter();
   }
 return l;
}
void main()
{// създаване на списък от списъци от хора
 list_of_lists ll;
 cout << "брой на списъците в списъка от списъци: ";
 int n;
 cin >> n;
 for (int k = 1; k <= n; k++)
 {strlist l;
  str s;
  cout << "брой на хората в списъка: ";
  int p;
  cin >> p;
  for (int i = 1; i <= p; i++)
  {cout << "s= ";
```

```
      cin >> s;
      l.ToEnd(s);
     }
     ll.ToEnd(l);
    }
    ll.print();
    // обхождане на елементите на ll и
    // сортиране на всеки от съставящите го списъци
    ll.IterStart();
    elem<strlist> *p = ll.Iter();
    while (p)
    {sortlist(p->inf);
     p = ll.Iter();
    }
    ll.print();
    // сливане на списъците, изграждащи ll
    ll.IterStart();
    p = ll.Iter();
    strlist l = p->inf;
    p = ll.Iter();
    while (p)
    {l = mergelists(l, p->inf);
     p = ll.Iter();
    }
    l.print();
   }
```

Сортиране чрез сливане

Този вид сортиране се осъществява по следния начин: списъкът се разделя на две половини. Всяка половинка се сортира и накрая сортираните половинки се сливат.

Задача 146. Да се напише шаблон на функция, който реализира метода сортиране чрез сливане.

Шаблонът mergesort реализира този начин за сортиране.

```
template <class T>
void mergesort(LList<T> &l)
{LList<T> l1, l2;
 l.IterStart();
 elem<T> *p = l.Iter();
 if (!p || p->link == NULL) return;
 while (p)
 {l1.ToEnd(p->inf);
  p = p->link;
  if (p)
  {l2.ToEnd(p->inf);
   p = p->link;
  }
 }
 mergesort(l1);
 mergesort(l2);
 l = mergelists(l1, l2);
}
```

Проверка на свойства на списъци

Задача 147. Да се напише програма, която създава списък от имена на хора и проверява дали елементите му са подредени лексикографски.

Програма Zad147.cpp решава задачата.

```
// Program Zad147.cpp
#include <iostream.h>
#include <string.h>
#include "L-List.cpp"
typedef char str[31];
typedef LList<str> strlist;
void LList<strlist>::print()
{elem<strlist> *p = Start;
 while (p)
 {p->inf.print();
  p = p->link;
```

```
 }
}
void LList<str>::ToEnd(str const & x)
{Current = End;
 End = new elem<str>;
 strcpy(End->inf, x);
 End->link = NULL;
 if (Current) Current->link = End;
 else Start = End;
}
bool monotone(strlist l)
{l.IterStart();
 elem<str>  *p = l.Iter(),
            *q,
            *r;
 if (!p->link) return true;
 q = p->link;
 r = q->link;
 while (strcmp(p->inf, q->inf) <= 0 && r)
 {p = q;
  q = r;
  r = r->link;
 }
 return strcmp(p->inf, q->inf) <= 0;
}
void main()
{strlist l;
 str s;
 cout << "broj= ";
 int p;
 cin >> p;
 for (int i = 1; i <= p; i++)
 {cout << "s= ";
  cin >> s;
  l.ToEnd(s);
 }
```

```
 l.print();
 cout << monotone(l) << endl;
}
```

Задача 148. Да се напише програма, която създава списък от цели числа и проверява дали елементите му са:

а) монотонно намаляващи;

б) различни.

Програма Zad148.cpp решава задачата.

```
// Program Zad148.cpp
#include <iostream.h>
#include "L-List.cpp"
typedef LList<int> IntList;
bool member(int x, IntList l)
{l.IterStart();
 elem<int> *p = l.Iter();
 if (!p) return false;
 int y;
 l.DeleteElem(p, y);
 return x == y || member(x, l);
}
// a)
bool monot(IntList l)
{l.IterStart();
 elem<int>*p = l.Iter();
 if (!p->link) return true;
 int x;
 l.DeleteElem(p, x);
 IntList l1 = l;
 l.IterStart();
 p = l.Iter();
 int y;
 l.DeleteElem(p, y);
 return x >= y && monot(l1);
}
```

```
// б)
bool diffr(IntList l)
{l.IterStart();
 elem<int> *p = l.Iter();
 if (!p->link) return true;
 int y;
 l.DeleteElem(p, y);
 return !member(y, l) && diffr(l);
}
void main()
{IntList l;
 int s;
 cout << "broj= ";
 int p;
 cin >> p;
 for (int i = 1; i <= p; i++)
 {cout << "s= ";
  cin >> s;
  l.ToEnd(s);
  }
cout << diffr(l) << endl;
}
```

Конструиране на списък от елементи на други списъци

В следващата задача се реализират операциите обединение, сечение и разлика на множества, представени чрез списъци.

Задача 149. Дадени са два списъка от цели числа p и q. Да се дефинира функция, която намира списък от цели числа, съдържащ елементите, които:

а) принадлежат на поне един от списъците p или q;

б) принадлежат едновременно и на p, и на q;

в) принадлежат на p, но не принадлежат на q;

г) принадлежат на един от списъците, но не принадлежат на другия.

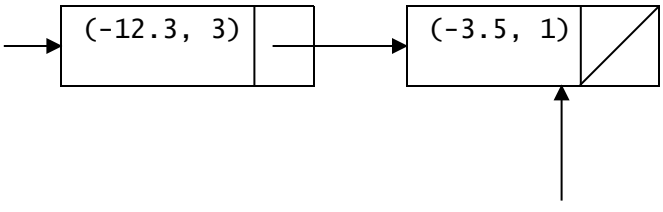
Някои от тези функции са реализирани в програма Zad150.cpp.

```
// Program Zad149.cpp
#include <iostream.h>
#include "L-List.cpp"
typedef LList<int> IntList;
bool member(int x, IntList l)
{l.IterStart();
 elem<int> *p = l.Iter();
 if (!p) return false;
 int y;
 l.DeleteElem(p, y);
 return x == y || member(x, l);
}
// a)
void unilists(IntList p, IntList q, IntList &r)
{p.IterStart();
 q.IterStart();
 elem<int> *pp = p.Iter(),
           *qq = q.Iter();
 while (pp)
 {if (!member(pp->inf, r)) r.ToEnd(pp->inf);
  pp = pp->link;
 }
 while (qq)
 {if (!member(qq->inf, r)) r.ToEnd(qq->inf);
  qq = qq->link;
 }
}
// б)
void intersections(IntList p, IntList q, IntList &r)
{p.IterStart();
 q.IterStart();
 elem<int> *pp = p.Iter(),
           *qq = q.Iter();
 while (pp)
 {if (!member(pp->inf, r) && member(pp->inf, q))
```

```
   r.ToEnd(pp->inf);
  pp = pp->link;
 }
}
void read(IntList & l)
{int s;
 cout << "number= ";
 int p; cin >> p;
 for (int i = 1; i <= p; i++)
 {cout << "elem= ";
  cin >> s;
  l.ToEnd(s);
  }
}
void main()
{IntList l, l1, l2, l3;
 read(l1);
 cout << "L1: "; l1.print();
 read(l2);
 cout << "L2: "; l2.print();
 unilists(l1, l2, l);
 l.print();
 intersections(l1, l2, l3);
 l3.print();
}
```

Използване на свързан списък за реализиране и работа с полиноми

Задача 150. Полимомът $a_n x^n + a_{n-1}x^{n-1} +... + a_1 x + a_0$ може да се представи чрез списък елементите, на който са структури с две полета: coef, означаващо коефициента и pow, означаващо степента на съответния едночлен. При това, ако коефициентът a_k е 0, съответният едночлен не се включва в списъка. Например, полиномът $-12.3x^3 - 3.5x$ се представя по следния начин:



Start

End ————————

Да се напишат следните функции за работа с полиноми:

а) въвеждане на полином;

б) извеждане на полином;

в) намиране на стойността на полином за дадено x;

г) намиране на производната на полином;

д) намиране на интеграла на полином;

е) намиране на сумата на два полинома;

ж) намиране на произведението на полином с едночлен;

з) намиране на произведението на два полинома.

Някои от тези функции са реализирани в програмата Zad150.cpp. Полинома ще реализираме чрез специализацията на класа LList с базов тип pol, определен като структура.

```
// Program Zad150.cpp
#include <iostream.h>
#include "L-List.cpp"
struct pol
{int pow;
 double coef;
};
typedef LList<pol> polinom;
//a)
void read(polinom &p)
{char flag;
 do
 {pol ed;
  cout << "pow: "; cin >> ed.pow;
  cout << "coef: "; cin >> ed.coef;
  p.ToEnd(ed);
  cout << "Input y/n: ";
  cin >> flag;
 } while (flag == 'y');
}
```

```
// б)
void LList<pol>::print()
{elem<pol> *p = Start;
 while (p)
 {if (p->inf.coef > 0) cout << " + ";
  else cout << " ";
  cout << p->inf.coef << " x ^ " << p->inf.pow;
  p = p->link;
 }
 cout << endl;
}
// г)
void diff(polinom p, polinom &newp)
{p.IterStart();
 elem<pol> *q = p.Iter();
 while (q)
 {pol ed;
  ed.coef = q->inf.coef * q->inf.pow;
  ed.pow = q->inf.pow - 1;
  newp.ToEnd(ed);
  q = q->link;
 }
}
// е)
void sum(polinom p, polinom q, polinom& r)
{p.IterStart();
 q.IterStart();
 elem<pol> *pp = p.Iter(),
           *qq = q.Iter();
 while (pp && qq)
 if (p->inf.pow > qq->inf.pow)
 {r.ToEnd(pp->inf);
  pp = pp->link;
 }
 else
 if (qq->inf.pow > pp->inf.pow)
```

```
 {r.ToEnd(qq->inf);
  qq = qq->link;
 }
 else
 {pol ed;
  ed.pow = pp->inf.pow;
  ed.coef = pp->inf.coef + qq->inf.coef;
  pp = pp->link;
  qq = qq->link;
  r.ToEnd(ed);
 }
 if (!pp)
  while (qq)
  {r.ToEnd(qq->inf);
   qq = qq->link;
  }
 else
 while (pp)
 {r.ToEnd(pp->inf);
  pp = pp->link;
 }
}
void main()
{polinom p, p1, p2;
 read(p); p.print();
 read(p1); p1.print();
 sum(p, p1, p2);
 p2.print();
 diff(p, p2);
p2.print();
}
```

Функции от по-висок ред за работа със списъци, представени с една връзка

Функция accumulate

Нека l е списък с елементи от тип T, а op е лявоасоциативна операция от вида:

op: T x T ⟶ T

с начална стойност null_val. Ще дефинираме шаблон на фукция от по-висок ред accumulate, чрез който да се реализира натрупване на елементите на списъка l чрез операцията op.

```
template <class T>
T accumulate(T (*op)(T, T), T null_val, LList<T>& l)
{T s = null_val;
 l.IterStart();
 elem<T> *p = l.Iter();
 while (p)
 {s = op(s, p->inf);
  p = p->link;
 }
 return s;
}
```

Задача 151. Като се използва функцията от по-висок ред accumulate, да се напише програма, която намира стойността на полинома $P_n(x) = (...(a_0x + a_1)x + ... + a_{n-1})x + a_n$ за дадено x по метода на Хорнер.

Полиномът се задава чрез списък от реални числа a_0, a_1, ..., a_{n-1}, a_n и реалното число x. В сила е $P_n(x) = x.P_{n-1}(x) + a_n$. Операцията op ще дефинираме по следния начин:

op: term x coef ⟶ x.term + coef

Програма Zad151.cpp решава задачата.

```
// Program Zad151.cpp
#include <iostream.h>
#include "L-List.cpp"
template <class T>
T accumulate(T (*op)(T, T), T null_val, LList<T>& l)
{T s = null_val;
 l.IterStart();
 elem<T> *p = l.Iter();
 while (p)
```

```
 {s = op(s, p->inf);
  p = p->link;
 }
 return s;
}
double x;
typedef LList<double> DList;
double op(double term, double coef)
{return x * term + coef;
}
double horner(DList l)
{return accumulate(op, 0.0, l);
}
void readpol(int n, DList & l)
{for (int i = n; i >= 0; i--)
 {cout << "coef: ";
  double coef;
  cin >> coef;
  l.ToEnd(coef);
 }
}
void main()
{DList l;
 int n; // степен на полинома
 cout << "n: ";
 cin >> n;
 readpol(n, l);
 cout << "x= "; cin >> x;
 cout << horner(l) << endl;
}
```

Функция map

Нека l е списък с елементи от тип T, а f е едноаргументна функция от вида:

f: T ⟶ T

Ще дефинираме шаблон на функция от по-висок ред map за намиране на списък, резултат от прилагането на функцията f над всеки от елементите на l.

```
template <class T>
LList<T> map(T (*f)(T), LList<T> & l)
{l.IterStart();
 elem<T> *p = l.Iter();
 LList<T> l1;
 while (p)
 {l1.ToEnd(f(p->inf));
  p = p->link;
 }
 return l1;
}
```

Ако е даден списък от реални числа l, след обръщението map(sin, l) се получава списъкът от синусите на елементите на l.

Функция filter

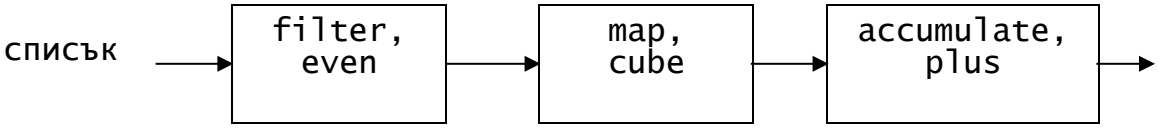
Нека l е списък с елементи от тип T, а pred е предикат. Ще дефинираме шаблон на функция от по-висок ред filter, чрез който се намира списък, състоящ се от онези елементи на l, за които е в сила pred.

```
template <class T>
LList<T> filter(bool (*pred)(T), LList<T> & l)
{LList<T> l1;
 l.IterStart();
 elem<T> *p = l.Iter();
 while (p)
 {if (pred(p->inf)) l1.ToEnd(p->inf);
  p = p->link;
 }
 return l1;
}
```

Ако l е списък от цели числа, чрез обръщението filter(odd, l), където odd е предикат, установяващ дали аргументът му е нечетно число, се получава списъкът от нечетните елементи на списъка l.

Задача 152. Да се напише програма, която намира сумата от кубовете на четните числа на даден списък от цели числа.

За решаването на тази задача ще преминем през следните стъпки:




```
// Program Zad152.cpp
#include <iostream.h>
#include "L-List.cpp"
template <class T>
T accumulate(T (*op)(T, T), T null_val, LList<T>& l)
{T s = null_val;
 l.IterStart();
 elem<T> *p = l.Iter();
 while (p)
 {s = op(s, p->inf);
  p = p->link;
 }
 return s;
}
template <class T>
LList<T> map(T (*f)(T), LList<T> &l)
{l.IterStart();
 elem<T> *p = l.Iter();
 LList<T> l1;
 while (p)
 {l1.ToEnd(f(p->inf));
  p = p->link;
 }
 return l1;
}
template <class T>
LList<T> filter(bool(*pred)(T), LList<T> & l)
```

```
{LList<T> l1;
 l.IterStart();
 elem<T> *p = l.Iter();
 while (p)
 {if (pred(p->inf)) l1.ToEnd(p->inf);
  p = p->link;
 }
 return l1;
}
int plus(int a, int b)
{return a + b;
}
bool even(int x)
{return x%2 == 0;
}
int cube(int x)
{return x*x*x;
}
void main()
{LList<int> m, n;
 int k; cin >> k;
 for (int j = 1; j <= k; j++)
 {cout << "number: ";
  int a;
  cin >> a;
  m.ToEnd(a);
 }
 m.print();
 n = map(cube, filter(even, m));
 n.print();
 cout << accumulate(plus, 0, n) << endl;
}
```

Задача 153. Да се напише програма, която създава списък от списъци от реални числа, след което конкатенира списъците, съставящи списъка.

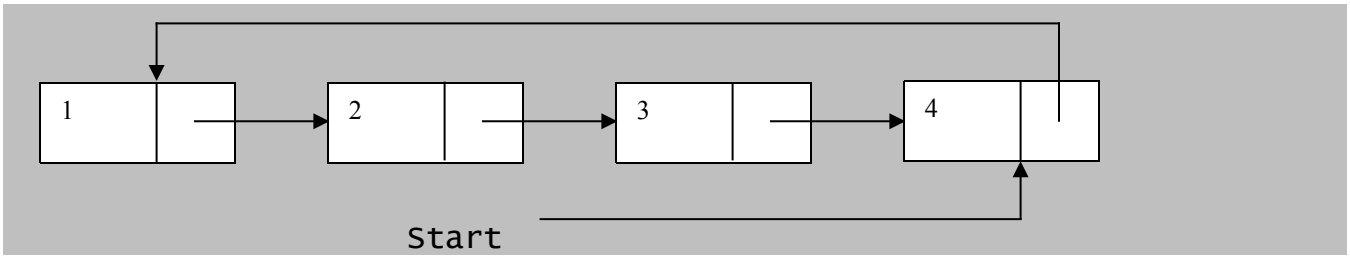
Програма Zad153.cpp решава задачата.

```
// Program Zad153.cpp
#include <iostream.h>
include "L-List.cpp"
typedef LList<double> DList;
template <class T>
T accumulate(T (*op)(T, T), T null_val, LList<T>& l)
{T s = null_val;
 l.IterStart();
 elem<T> *p = l.Iter();
 while (p)
 {s = op(s, p->inf);
  p = p->link;
 }
 return s;
}
DList concat(DList l1, DList l2)
{DList l;
 l1.IterStart();
 l2.IterStart();
 elem<double> *p = l1.Iter(),
              *q = l2.Iter();
 while (p)
 {l.ToEnd(p->inf);
  p = p->link;
 }
 while (q)
{l.ToEnd(q->inf);
  q = q->link;
 }
 return l;
}
void main()
{LList<DList> l;
 int n;
 cout << "n= ";
```

```
 cin >> n;
 for (int i = 1; i <= n; i++)
 {DList m;
  cout << "k= "; int k;
  cin >> k;
  for (int j = 1; j <= k; j++)
  {cout << "elem: ";
   int el;
   cin >> el;
   m.ToEnd(el);
  }
  l.ToEnd(m);
 }
 DList l2;
 DList l1 = accumulate(concat, l2, l);
 l1.print();
}
```

15.3.4 Циклични списъци

В редица случаи се налага да се използват разновидности на свързаните списъци. Например, ако искаме операцията включване да се осъществява само в началото или само в края на свързания списък, не е нужно да поддържаме два указателя Start и End към началото и края съответно. В този и в редица други случаи се използват т.нар. **циклични** или **кръгови списъци.** При тях се използва само един указател Start, указващ последния елемент на списъка, както е показано на Фиг. 15. 6.



Фиг. 15.6 Представяне на цикличен списък с четири елемента

15.3.4.1 Реализация на цикличен списък

Шаблонът на класа cirlist реализира циклични свързани списъци от тип T.

```
template <class T>
struct elem
{T inf;
 elem *link;
};
template <class T>
class CirList
{public:
  CirList();
  ~CirList();
  CirList(CirList &);
  CirList& operator=(CirList &);
  void print();
  void IterStart(elem<T>* = NULL);
  elem<T>* Iter();
  void ToEnd(T &);
  void Insert(T &);
  void DeleteElem(elem<T>*, T &);
 private:
  elem<T> *Start,
          *Current;
  void DeleteList();
  void CopyList(CirList &);
};
```

Конструкторът по подразбиране и член-функциите на голямата тройка реализират аналогични идеи на тези при свързания списък с една връзка. Забелязваме, че формалните параметри на член-функциите

```
cirlist(cirlist &);
cirlist& operator=(cirlist &);
```

не са const. Това се налага заради функцията CopyList, чийто формален параметър също не е const. Последното пък е свързано с използването на функциите за работа с итератора при обхождането за копиране.

```
template <class T>
CirList<T>::CirList()
{Start = NULL;
}
template <class T>
CirList<T>::~CirList()
{DeleteList();
}
template <class T>
CirList<T>::CirList(CirList<T> & r)
{CopyList(r);
}
template <class T>
CirList<T>& CirList<T>::operator=(CirList<T> & r)
{if (this != &r)
 {DeleteList();
  CopyList(r);
 }
 return *this;
}
```

Тъй като при програмирането на член-функциите на шаблона на класа CirList ще използваме функциите за работа с итератора, отначало ще разгледаме тези функции.

Установяване на итератора в началото на цикличен списък

Указателят Start сочи последния елемент на цикличния списък. Елементът, намиращ се след указания от Start (ако Start не е NULL) е началото на списъка. Функцията IterStart установява Current в указан адрес, в NULL – ако списъкът е празен и в началото на списъка в противен случай.

```
template <class T>
void CirList<T>::IterStart(elem<T> *p)
{if (p) Current = p;
```

```
 else
 if (!Start) Current = NULL;
 else Current = Start->link;
}
```

Преместване на итератора в следваща позиция

Член-функцията Iter() осигурява преместване на Current в следващата позиция. При достигане до Start (указващ последния елемент на кръговия списък), Current става NULL.

```
template <class T>
elem<T>* CirList<T>::Iter()
{if (!Current) return NULL;
 elem<T> *p = Current;
 if (Current == Start) Current = NULL;
  else Current = Current->link;
  return p;
 }
```

Помощните член-функции за изтриване и копиране на цикличен списък DeleteList и CopyList съответно използват функциите за работа с итератора.

Изтриване на списък

```
template <class T>
void CirList<T>::DeleteList()
{IterStart();
 elem<T> *p = Iter();
 while (p)
 {delete p;
  p = Iter();
 }
}
```

Копиране на списък

```
template <class T>
void CirList<T>::CopyList(CirList & r)
```

```
{Start = NULL;
 r.IterStart();
 elem<T> *p = r.Iter();
 while (p)
 {ToEnd(p->inf);
  p = r.Iter();
 }
}
```

Включване на елемент

Ще реализираме две член-функции за включване на елемент в цикличен списък. Функцията Insert включва указан като формален параметър елемент след последния (пред първия) елемент на списъка, а функцията ToEnd включва указания елемент като последен елемент на списъка.

```
template <class T>
void CirList<T>::Insert(T & x)
{elem<T> *p = new elem<T>;
 p->inf = x;
 if (Start) p->link = Start->link;
 else Start = p;
 Start->link = p;
}
template <class T>
void CirList<T>::ToEnd(T & x)
{Insert(x);
 Start = Start->link;
}
```

Изтриване на елемент

Член-функцията Delete изтрива указания от указателя p елемент и го запомня в параметъра x.

```
template <class T>
void CirList<T>::DeleteElem(elem<T>* p, T & x)
{x = p->inf;
 if (Start != Start->link)
 {elem<T> *q = Start;
```

```
  while (q->link != p) q = q->link;
  q->link = p->link;
  if (p == Start) Start = q;
  delete p;
 }
 else
 {Start = NULL;
  delete p;
 }
}
```

Извеждане на цикличен списък

```
template <class T>
void CirList<T>::print()
{IterStart();
 elem<T> *p = Iter();
 while (p)
 {cout << p->inf << " ";
  p = Iter();
 }
 cout << endl;
}
```

Ще приложим този шаблон на клас в следната задача.

Задача 154. Дадени са естествените числа n и m. Предполага се, че m човека са наредени в кръг и всеки от тях е получил пореден номер от 1 до m (броейки в посока обратна на часовниковата стрелка). След това, започвайки от първия и също в посока обратна на часовниковата стрелка се отброява n-тия човек и се отстранява от кръга. Отново започвайки от следващия човек (n+1-вия) се отброява отново n-тия човек и се отстранява. Този процес продължава до отстраняване на всички от кръга. Да се напише програма, която извежда номерата на отстранените в реда на отстраняване.

Програма Zad154.cpp решава задачата.

```
// Program Zad154.cpp
#include <iostream.h>
#include “CirList.cpp”
typedef CirList<int> IntCir;
void create(int m, IntCir &l)
{for (int i = 1; i <= m; i++)
 l.ToEnd(i);
}
void josiff(int n, IntCir l)
{l.IterStart();
 elem<int> *p = l.Iter(), *q;
 while (p != p->link)
 {q = p;
  for (int i = 1; i <= n-1; i++)
    q = q->link;
  p = q->link;
  int x; l.DeleteElem(q, x);
  cout << x << " ";
 }
 cout << p->inf << endl;
}
void main()
{IntCir l;
 create(10, l);
 l.print();
 josiff(3,  l);
}
```

15.3.4.2 Функции от по-висок ред за работа със циклични списъци

Задача 155. Да се напише шаблон на функцията от по-висок ред accumulate за циклични списъци.

```
template <class T>
T accumulate(T (*op)(T, T), T null_val, CirList<T>& l)
{T s = null_val;
```

```
 l.IterStart();
 elem<T> *p = l.Iter();
 while (p)
 {s = op(s, p->inf);
  p = l.Iter(); //използването на p=p->list ще доведе до зацикляне
 }
 return s;
}
```

Задача 156. Да се напише шаблон на функцията от по-висок ред map за циклични списъци.

```
template <class T>
CirList<T> map(T (*f)(T), CirList<T> &l)
{l.IterStart();
 elem<T> *p = l.Iter();
 CirList<T> l1;
 while (p)
 {l1.ToEnd(f(p->inf));
  p = l.Iter();
 }
 return l1;
}
```

Задача 157. Да се напише шаблон на функцията от по-висок ред filter за циклични списъци.

```
template <class T>
CirList<T> filter(bool (*pred)(T), CirList<T> &l)
{l.IterStart();
 elem<T> *p = l.Iter();
 CirList<T> l1;
 while (p)
 {if (pred(p->inf)) l1.ToEnd(p->inf);
  p = l.Iter();
 }
```

```
 return l1;
}
```

Задача 158. Да се напише програма, която създава цикличен списък, изграден от циклични списъци от числа, след което конструира цикличен списък, резултат от конкатенацията на цикличните списъци, съставящи дадения списък.

Програма Zad158.cpp решава задачата. Тя използва функцията от по-висок ред accumulate. За да се избегне зацикляването, в оператора за цикъл е използвана член-функцията Iter().

```
// Program Zad158.cpp
#include <iostream.h>
#include "CirList.cpp"
template <class T>
T accumulate(T (*op)(T, T), T null_val, CirList<T>& l)
{T s = null_val;
 l.IterStart();
 elem<T> *p = l.Iter();
 while (p)
 {s = op(s, p->inf);
  p = l.Iter(); //използването на p=p->list ще доведе до зацикляне
 }
 return s;
}
typedef CirList<int> IntList;
typedef CirList<IntList> IntListList;
// конкатениране на циклични списъци
IntList concat(IntList l1, IntList l2)
{IntList l;
 l1.IterStart();
 l2.IterStart();
 elem<int> *p = l1.Iter(),
           *q = l2.Iter();
 while (p)
```

```
 {l.ToEnd(p->inf);
  p = l1.Iter();
 }
 while (q)
 {l.ToEnd(q->inf);
  q = l2.Iter();
 }
 return l;
}
// създаване на цикличен списък от n циклични списъка
void create(int n, IntListList &l)
{for (int i = 1; i <= n; i++)
 {IntList m;
  cout << "k= ";
  int k;
  cin >> k;
  for (int j = 1; j <= k; j++)
  {cout << "x: ";
   int x;
   cin >> x;
   m.ToEnd(x);
  }
  l.ToEnd(m);
 }
}
void main()
{int n;
 cout << "n= ";
 cin >> n;
 IntListList l;
 create(n, l);
 IntList l2;
 IntList l1 = accumulate(concat, l2, l);
 l1.print();
}
```

15.3.5 Реализация на свързан списък с две връзки

Шаблонът на клас DLList реализира това представяне на свързан списък.

```
template <class T>
struct elem
{T inf;
 elem *pred,
      *succ;
};
template <class T>
class DLList
{public:
  DLList();
  ~DLList();
  DLList(DLList const&);
  DLList& operator=(DLList const &);
  void print();
  void print_reverse();
  void IterStart(elem<T>* = NULL);
  void IterEnd(elem<T>* = NULL);
  elem<T>* IterSsucc();
  elem<T>* IterPred();
  void ToEnd(T const &);
  void DeleteElem(elem<T> *, T &);
  int len();
 private:
  elem<T> *Start,
          *End,
          *CurrentS,
          *CurrentE;
  void DeleteList();
  void CopyList(DLList const &);
};
```

Конструктора и член-функциите на голямата тройка няма да коментираме. Реализират аналогични идеи на тези при свързаното представяне с една връзка.

```
template <class T>
DLList<T>::DLList()
{Start = NULL;
 End = NULL;
}
template <class T>
DLList<T>::~DLList()
{DeleteList();
}
template <class T>
DLList<T>::DLList(DLList<T> const& r)
{CopyList(r);
}
template <class T>
DLList<T>& DLList<T>::operator=(DLList<T> const & r)
{if (this != &r)
 {DeleteList();
  CopyList(r);
 }
 return *this;
}
```

Следващите две функции са помощни и реализират изтриване на свързан списък и копиране на свързан списък на друго място в паметта. Използват се за реализиране на голямата тройка и затова са капсолирани в private-частта на класа.

Изтриване на списък

```
template <class T>
void DLList<T>::DeleteList()
{elem<T> *p = Start;
 while (p)
 {Start = Start->succ;
  delete p;
```

```
  p = Start;
 }
 End = NULL;
}
```

Копиране на списък

```
template <class T>
void DLList<T>::CopyList(DLList<T> const & r)
{Start = End = NULL;
 if (r.Start)
 {elem<T> *p = r.Start;
  while (p)
  {ToEnd(p->inf);
   p = p->succ;
  }
 }
}
```

Извеждане на елементите на списък

Извеждането на елементите на свързан списък се реализира чрез последователното им обхождане, без разрушаването на списъка. При обхождане от Start, списъкът се извежда от началото му, а при обхождане, започващо от End, извеждането е в обратен ред.

- извеждане от началото към края

```
template <class T>
void DLList<T>::print()
{elem<T> *p = Start;
 while (p)
 {cout << p->inf << " ";
  p = p->succ;
 }
 cout << endl;
}
```

- извеждане от края към началото

```
template <class T>
void DLList<T>::print()
```

```
{elem<T> *p = End;
 while (p)
 {cout << p->inf << " ";
  p = p->pred;
 }
 cout << endl;
}
```

Следващите четири член-функции реализират операциите с итераторите CurrentS и CurrentE.

Установяване на итератора в началото на списъка

```
template <class T>
void DLList<T>::IterStart(elem<T> *p)
{if (p) CurrentS = p;
 else CurrentS = Start;
}
```

Установяване на итератора в края на списъка

```
template <class T>
void DLList<T>::IterEnd(elem<T> *p)
{if (p) CurrentE = p;
 else CurrentE = End;
}
```

Преместване на итератора в следващата позиция

```
template <class T>
elem<T>* DLList<T>::IterSucc()
{elem<T> *p = CurrentS;
 if (CurrentS) CurrentS = CurrentS->succ;
 return p;
}
```

Преместване на итератора в предходната позиция

```
template <class T>
elem<T>* DLList<T>::IterPred()
{elem<T> *p = CurrentE;
```

```
 if (CurrentE) CurrentE = CurrentE->pred;
 return p;
}
```

Включване на елемент в списък

Включването на елемент в края на свързан списък ще реализираме чрез член-функцията ToEnd.

```
template <class T>
void DLList<T>::ToEnd(T const & x)
{elem<T>* p = End;
 End = new elem<T>;
 End->inf = x;
 End->succ = NULL;
 if (p) p->succ = End;
 else Start = End;
 End->pred = p;
}
```

Изключване на указан елемент от списък

Изключването на указан елемент в свързан списък ще реализираме чрез следната член-функция:

```
template <class T>
void DLList<T>::DeleteElem(elem<T> *p, T &x)
{x = p->inf;
 if (Strat == End)
 {Start = End = NULL;
 }
 else
 if (p == Start)
 {Start = Start->succ;
  Start->pred = NULL;
 }
 else
 if (p == End)
 {End = p->pred;
  End->succ = NULL;
```

```
 }
 else
 {p->pred->succ = p->succ;
  p->succ->pred = p->pred;
 }
 delete p;
}
```

Дължина на свързан списък

Намирането дължината на свързан списък се осъществява по същия начин, като при представянето с една връзка.

```
template <class T>
int DLList<T>::len()
{int n = 0;
 elem<T> *p = Start;
 while (p)
 {n++;
  p = p->succ;
 }
 return n;
}
```

Задачи върху свързан списък с две връзки

Шаблонът на класа DLList записваме във файла DL-List.cpp.

Задача 159. Свързан списък, съдържащ 2n цели числа a_1, a_2,...,a_{2n-1}, a_{2n}, е представен чрез две връзки. Да се напише функция, която намира:

а) $S = a_1.a_{2n} + a_2.a_{2n-1} + a_n.a_{n+1}$

б) $M = max\{min\{a_1, a_{2n}\}, min\{a_2, a_{2n-1}\}, ..., min\{a_n, a_{n+1}\}\}$

в) $M = max\{min\{a_1, a_2, ..., a_n\}, min\{a_{n+1}, a_{n+2}, a_{2n}\}\}$.

Програма Zad159.cpp решава задачата.

```
// Program Zad159.cpp
#include <iostream.h>
#include "DL-List.cpp"
```

```
typedef DLList<int> IntList;
// създаване на свързан списък с две връзки
void CreateList(int n, IntList &l)
{for (int i = 1; i <= 2*n; i++)
{cout << "Enter a number: ";
 int x;
 cin >> x;
 l.ToEnd(x);
}
}
// a)
int sum(IntList &l)
{l.IterStart();
 elem<int> *p = l.IterSucc();
 l.IterEnd();
 elem<int> *q = l.IterPred();
 int n = l.len()/2;
 int s = 0;
 for (int i = 1; i <= n; i++)
 {s = s + p->inf * q->inf;
  p = l.IterSucc();
  q = l.IterPred();
 }
 return s;
}
// б)
int maximin(IntList &l)
{int max;
 l.IterStart();
 elem<int> *p = l.IterSucc();
 l.IterEnd();
 elem<int> *q = l.IterPred();
 int n = l.len()/2;
 if (p->inf <= q->inf) max = p->inf;
 else max = q->inf;
 for (int i = 1; i <= n-1; i++)
```

```
 {p = l.IterSucc();
  q = l.IterPred();
  int min;
  if (p->inf <= q->inf) min = p->inf;
  else min = q->inf;
  if (min > max) max = min;
 }
 return max;
}
void main()
{IntList l;
 cout << "n= ";
 int n;
 cin >> n;
 CreateList(n, l);
 l.print();
 cout << “sum: “ << sum(l) << endl;
 cout << “maximin: “ << maximin(l) << endl;
}
```

Задача 160. Даден е свързан списък, съдържащ 2n цели числа a_1, a_2, ..., a_{2n-1}, a_{2n}, представен чрез две връзки така, че първите му n елемента са сортирани във възходящ, а вторите n елемента – в низходящ ред. Да се напише функция, която сортира във възходящ ред елементите на списъка.

Функцията Sort решава задачата. Тя реализира следната идея. Поставя указатели p и q към началото и към края съответно. Докато p и q са различни, по-малкият от сочените от p и q елементи се включва в нов списък, след което се прескача.

```
IntList Sort(IntList &l)
{l.IterStart();
 elem<int> *p = l.IterSucc();
 l.IterEnd();
 elem<int> *q = l.IterPred();
 IntList l1;
```

```
 while (p != q)
 if (p->inf <= q->inf)
 {l1.ToEnd(p->inf);
  p = l.IterSucc();
 }
 else
 {l1.ToEnd(q->inf);
  q = l.IterPred();
 }
 l1.ToEnd(p->inf);
 return l1;
}
```

Задачи

Задача 1. Да се напише нерекурсивна (рекурсивна) функция, която проверява дали елемент принадлежи на стек.

Задача 2. Да се напише нерекурсивна (рекурсивна) функция, която проверява дали елементите на стек от числа (думи) са наредени във възходящ (лексикографски) ред.

Задача 3. Да се напише нерекурсивна (рекурсивна) функция, която проверява дали елементите на стек от числа са различни.

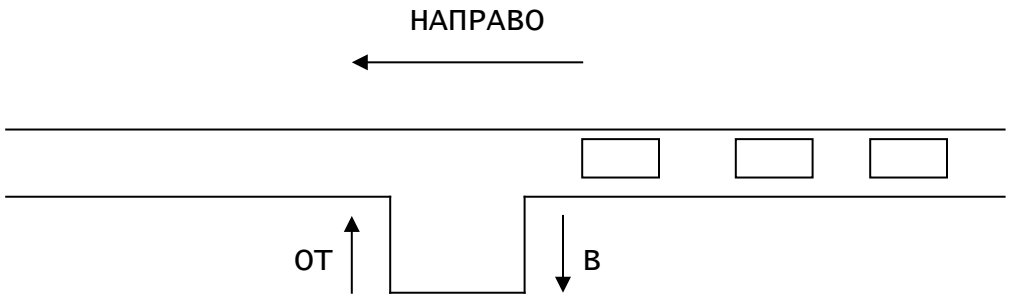
Задача 4. Даден е стек от цели числа. Да се напише булева функция, която установява дали елементите на стека могат да се пренаредят така, че разликата на всеки два съседни елемента да е наредена по намаляване. Ако за дадения стек това е възможно, да се пренаредят елементите на стека, след което се изведат.

Задача 5. В стек е записан без грешка израз от вида:

```
<израз> ::= <цифра>|
            s(<израз>)|
            p(<израз>)
```

където s(x) и p(x) намират x+1 и x-1, съответно (s(9) се приема за 0, а p(0) – за 9). Като се използва стек да се намери стойността на израза.

Задача 6. Железопътен сортировъчен възел е построен по следния начин:



разклонение

В дясната страна са събрани n черни и n бели вагона. В разклонението могат да се поместят всички вагони. Като се използват трите операции В, ОТ и НАПРАВО, да се съберат вагоните в лявата страна така, че да се редуват вагоните от двата вида. За целта да се използват не повече от 3n-1 операции В, ОТ и НАПРАВО.

Задача 7. Железопътен сортировъчен възел е построен по начина от предната задача. В дясната част са разположени n различни вагона, номерирани с 1, 2, ..., n. Да се напише програма, която като използва операците В, ОТ и НАПРАВО, намира всички възможни начини за подреждане на вагоните.

Задача 8. Да се напише програма, която намира колко различни пермутации на елементите 1, 2, ..., n могат да се конструират като се използва стекът от задача 6 и операциите В, ОТ и НАПРАВО.

Задача 9. Да се напише нерекурсивна (рекурсивна) функция, която проверява дали елемент принадлежи на опашка.

Задача 10. Да се напише нерекурсивна (рекурсивна) функция, която проверява дали елементите на опашка от числа (думи) са наредени във възходящ (лексикографски) ред.

Задача 11. Да се напише нерекурсивна (рекурсивна) функция, която проверява дали елементите на опашка от числа са различни.

Задача 12. Дадени са две опашки от цели числа p и q. Да се дефинира функция, която намира опашка от цели числа, съдържаща елементите, които:

а) принадлежат на поне една от опашките p или q;

б) принадлежат едновременно и на p, и на q;

в) принадлежат на p, но не принадлежат на q;

г) принадлежат на една от опашките, но не принадлежат на другата.

Задача 13. Да се дефинира функцията от по-висок ред accumulate за стек (опашка).

Задача 14. Да се дефинира функцията от по-висок ред map за стек (опашка).

Задача 15. Да се дефинира функцията от по-висок ред filter за стек (опашка).

Задача 16. Символен низ съдържа правилен текст от вида:

<текст> ::= <интервал> | <елемент><текст>

<елемент> ::= <буква> | (<текст>).

Като се използват опашка и/или стек, да се напише функция, която за всяка двойка съответстващи си отваряща и затваряща скоби извежда позициите им в текста, подредени по нарастване на позицията на отварящите скоби.

(Например, за текста A+(TP-F(X)*(B-C)) резултатът е 3 17; 8 10; 12 16.)

Задача 17. Символен низ съдържа правилен текст от вида:

<текст> ::= <интервал> | <елемент><текст>

<елемент> ::= <буква> | (<текст>).

Като се използват опашка и/или стек, да се напише функция, която за всяка двойка съответстващи си отваряща и затваряща скоби извежда позициите им в текста, подредени по нарастване на позицията на затварящите скоби. (Например, за текста A+(TP-F(X)*(B-C)) резултатът е 8 10; 12 16; 3 17.)

Задача 18. Даден е списък от четен брой символи. Да се напише булева функция, която определя дали елементите от първата половина съвпадат с елементите от втората половина на списъка. Елементите на списъка да се сканират отляво надясно.

Задача 19. Да се напише функция, която изтрива от списъка l:

а) първия отрицателен елемент, ако такъв съществува;

б) всички отрицателни елементи.

Задача 20. Да се напише програма, която създава свързан списък, съдържащ информация за студентите от една група. Програмата да може да:

а) добавя данни за нов студент;

б) изтрива данни за студент;

в) търси данни за студент;

г) сортира по някаква данна елементите на списъка.

Задача 21. Да се напише програма, която създава свързан списък, елементите, на който са свързани списъци от вида, описан в предната задача. Сортира всеки от свързаните списъци по успеха на студентите и извежда получения списък от списъци.

Задача 22. Даден е свързан списък с 2n елемента, представен с две връзки. Да се напише функция, която проверява дали елементите на списъка са симетрични относно средата на списъка.

Задача 23. Да се напише програма, която създава цикличен списък от циклични списъци от символи. Да се конструира цикличен списък от думите, получени след конкатенирането на символите на всеки от цикличните списъци. Да се конструира изречение, получено след конкатенирането на всяка от думите на цикличния списък.

Допълнителна литература

1. B. Stroustrup, C++ Programming Language. Third Edition, Addison – Wesley, 1997.
2. А. Берстисс, Структуры данных, М. Статистика, 1974.
3. Ст. Липман, Езикът C++ в примери, “КОЛХИДА ТРЕЙД” КООП, София, 1993.
4. Л. Амерал, Алгоритми и структури от данни в C++, София, ИК СОФТЕХ, 2001.
5. М. Тодорова, Програмиране на Паскал, София, Полипринт, 1993.