3.2 L1/L2 списки.

Список можно представить как некоторую структуру данных, где элементы связаны в линейную цепочку и совместно с каждым элементом данных хранится адрес места размещения соседнего с ним элемента. Различают однонаправленный список (хранится адрес только следующего элемента) и двунаправленный список (хранятся адреса следующего и предыдущего элементов). В каждый момент времени в списке определен один текущий элемент с которым и разрешается работать (также может быть разрешена работа с непосредственными соседями этого элемента). Положение текущего элемента определяется так называемым указателем списка. Указатель списка можно перемещать на соседние элементы по направлению ссылок и тем самым получить доступ к любому элементу списка. Добавление и удаление элементов происходит только в окрестности текущего положения указателя списка, при этом новые добавляемые элементы "раздвигают" список. Можно принять различные соглашения по поводу того, какой элемент станет текущим в результате добавления или удаления. Мы будем считать, что при добавлении текущим остается старый текущий элемент, а при удалении новым текущим будет следующий элемент (например тот, который был "правым соседом" бывшего текущего элемента).

3.26. Постройте реализацию двунаправленного списка элементов абстрактного типа `Type` на языке C.

Идеи реализации. Для описания элемента списка будем использовать следующий тип данных:

```
typedef struct L2 {
        Type  value;
        struct L2 *next, *prev;
}  ListItem;
```

Здесь `next` и `prev` являются соответственно указателями на следующий и предыдущий элементы по отношению к данному. Введем дополнительный элемент `ListItem base` и свяжем весь список в кольцо через этот элемент. Таким образом, начальный и конечный элементы списка будут ссылаться на адрес элемента `base`.

Текущую позицию в списке будем задавать указателем на текущий элемент:

```
ListItem  *current;
```

Для работы со списком примем следующий интерфейс:

```
/*---   Файл  list2.h  ----------------------------------------*/
typedef struct L2 {
        Type  value;
        struct L2 *next, *prev;
}  ListItem;
int     L2_Init (void);          /* создать (пустой) список      */
void    L2_Term (void);          /* закончить работу             */
```

```
int     L2_AddBefore (Type val); /* добавить элемент до текущего */
int     L2_AddAfter (Type val);  /* добавить элемент за текущим  */
int     L2_DelCurrent (void);    /* удалить текущий элемент      */
int     L2_ToNext (void);       /* перейти к следующему элементу */
int     L2_ToPrev (void);      /* перейти к предыдущему элементу */
int     L2_ToHead (void);        /* перейти к началу списка      */
int     L2_ToTail (void);        /* перейти к концу списка       */
int     L2_AtHead (void);        /* позиция в начале списка ?    */
int     L2_AtTail (void);        /* позиция в конце списка ?     */
Type  * L2_Current (void);       /* доступ к текущему элементу   */
int     L2_NumElem (void);       /* кол-во элементов в списке    */
/*---   конец файла stack.h   ----------------------------------*/
```

Функция `L2_NumElem` возвращает количество элементов в списке, все остальные функции, возвращающие `int`, возвращают 0 в случае успеха и -1 в случае отказа.

Решение. Приведем в качестве примера реализацию нескольких функций.

```
/*---   Файл  list2.c  -----------------------------------------*/
#include <stdlib.h>
#include "list2.h"
static ListItem  base;
static ListItem *current;
static int   num_elem;

int     L2_Init (void)
{
   current = &base;
   base.next = base.prev = &base;
   num_elem = 0;
   return 0;
}
void    L2_Term (void)
{
   for (L2_ToHead(); !L2_NumElem(); L2_DelCurrent());
}
int     L2_AddBefore (Type val)
{
   ListItem *pos;
   pos = (ListItem*)malloc(sizeof(ListItem));
   if (!pos) return -1;
   pos->prev = current->prev;
   pos->next = current;
   current->prev->next = pos;
   current->prev = pos;
   pos->value = val;
   num_elem++;
   return 0;
}
int     L2_DelCurrent (void)
```

```
{
   ListItem *pos;
   if (L2_Empty()) return -1;
   current->prev->next = current->next;
   current->next->prev = current->prev;
   pos = current;
   current = current->next;
   num_elem--;
   free (pos);
   return 0;
}
int     L2_ToNext (void)
{
   if (L2_AtTail()) return -1;
   current = current->next;
   return 0;
}
int     L2_AtHead (void)
{
   return  (current==base.next);
}
Type  * L2_Current  (void)
{
   return  current;
}
int     L2_NumElem (void)
{
   return  num_elem;
}
```

Остальные функции реализуйте самостоятельно.

3.27. Добавьте к реализации списка из предыдущей задачи функцию последовательного поиска элемента с заданным значением поля `value` (при успехе текущая позиция `current` устанавливается на найденный элемент, при неудаче — не изменяется) и функцию, применяющую заданную процедуру к каждому элементу списка (текущая позиция не изменяется). Заголовки этих функций могут иметь вид

```
int    L2_Search (Type val);
void   L2_Process (void (*action)(Type));
```

Функция поиска должна возвращать 0, если элемент найден, и -1 при отсутствии элемента в списке.

3.28. Постройте реализацию двунаправленного списка элементов абстрактного типа `Type` на языке C++.

3.29. Постройте реализацию двунаправленного списка элементов абстрактного типа `Type` без использования элемента `base`.

Идеи реализации. Свяжем весь список в кольцо (т.е. свяжем ссылками первый и последний элементы). Для запоминания позиций начального и

конечного элементов введем два указателя: `ListItem *first, *last`. Для пустого списка значения этих указателей равны нулю. В процедурах добавления и удаления теперь появятся дополнительные проверки, необходимые для корректной работы, когда текущая позиция находится в начале или конце списка.

3.30. Постройте реализации однонаправленного списка элементов абстрактного типа `Type` на языках C и C++ с интерфейсом, аналогичным двунаправленному списку с естественными изменениями (недоступны перемещение и доступ к элементам в одном из направлений).

Идеи реализации. Однонаправленный список можно легко получить соответствующим упрощением реализации двунаправленного списка. Однако теперь недостаточно хранить позицию только текущего элемента (существенно усложняется процедура удаления). Наряду с указателем текущей позиции здесь будет удобно хранить указатель и на предыдущий элемент (назовем этот указатель `before`).

Решение. В структуре `ListItem` оставим только ссылку `next`. Приведем здесь в качестве иллюстрации процедуры удаления текущего элемента и перемещения по списку.

```
int     L1_DelCurrent (void)
{
   ListItem *pos;
   if (L1_Empty()) return -1;
   before->next = current->next;
   pos = current;
   current = current->next;
   num_elem--;
   free (pos);
   return 0;
}
int     L1_ToNext (void)
{
   if (L1_AtTail()) return -1;
   before = current;
   current = current->next;
   return 0;
}
int     L1_ToHead (void)
{
   if (L1_Empty()) return -1;
   before = &base;
   current = base.next;
   return 0;
}
```

3.31. Добавьте к реализациям списков из задач 3.26-3.30 процедуры упорядочивания списка по некоторому критерию. Заголовок соответствующей функции может иметь вид

```
void  Sort (int (*compare)(Type a,Type b));
```

где `compare` — указатель на функцию сравнения значений двух элементов списка. Естественно, при упорядочивании нужно ограничиться только перестройкой взаимных ссылок, не перемещая в памяти сами элементы списка. Реализуйте на списках следующие алгоритмы сортировки (см. 1.147-1.157):

1) сортировка выбором максимального элемента;

2) пузырьковая сортировка;

3) сортировка просеиванием;

4) вставка в упорядоченную часть списка с последовательным поиском;

5) слияние подсписков (алгоритм Неймана);

6) быстрая сортировка (алгоритм quicksort) для двунаправленного списка.

Какие из этих алгоритмов могут быть эффективно реализованы на однонаправленных списках?

3.32. На основе списочной реализации, постройте полный словать встречающихся в заданном файле слов с подсчетом частоты вхождения. На примере нескольких произведений известного автора, найдите его любимые слова.

3.33. Массив списков. Постройте совместную реализацию нескольких списков на языке C. Каждая функция работы со списком при этом получит дополнительный параметр — номер списка. Процедура инициализации получает в качестве параметра начальное количество списков. Прототипы функций могут иметь следующий вид (ср. с задачей 3.26)

```
int     L_Init (int k);                /* создать k списков  */
int     L_AddAfter (int k, Type val); /* добавить в k-й список */
```

и т.д. Реализуйте задачу 3.32 на основе массива списков — по отдельному списку для каждой начальной буквы.

3.34. Матрица, в которой число ненулевых элементов значительно меньше общего количества элементов, называется разреженной. Сохраняя в элементе списка значения a_{ij} и j для каждого i, мы можем представить такую матрицу как набор списков элементов матрицы по строкам. На основе решения предыдущей задачи реализуйте на языке C процедуры работы с разреженной матрицей со следующим интерфейсом:

```
int     CreateMatr (int m, int n);    /* создать m*n матрицу */
int     Put (double x, int i, int j); /* записать элемент x(i,j) */
double  Get (int i, int j);           /* прочитать элемент x(i,j) */
```

Обратите внимание на то, что нулевые элементы не должны храниться в матрице, т.е. при записи нулевого значения в существующий элемент списка, он должен удаляться из памяти.

3.35. Добавьте к реализации разреженной матрицы из предыдущей задачи функции, выполняющие линейные комбинации и перестановки строк (и столбцов). Используя построенные функции напишите программу решения системы линейных уравнений с разреженной матрицей методом Гаусса.

3.36. Реализуйте разреженную матрицу на языке C++ со следующим интерфейсом:

```
class sparse_matrix
{
 private:
   .....
 public:
   sparse_matrix (int m, int n);
  ~sparse_matrix ();
   double & x(int i, int j);    /* элемент матрицы */
}
```

3.37. Добавьте в класс `sparse_matrix` из предыдущей задачи переопределения арифметических операций с матрицами (сложение, вычитание, умножение).

3.38. На основе реализации разреженной матрицы постройте электронную таблицу. Смысл задачи состоит в автоматической модификации значений некоторых элементов матрицы при изменении значений других элементов. Простейшим примером может служить числовая матрица, в которой последний столбец содержит суммы элементов соответствующих строк, а последняя строка содержит суммы элементов из соответствующих столбцов. При изменении элементов матрицы эти суммы соответствующим образом модифицируются. Для повышения универсальности реализации можно продумать способы предоставления пользователю возможности менять алгоритмы пересчета элементов матрицы.

3.3 Деревья

Прежде чем формулировать задачи, касающиеся деревьев, напомним ряд необходимых понятий.

Дерево — это структура вполне соответствующая обычному математическому понятию дерева, как связного планарного (плоского без пересечений ребер) графа, не содержащего циклов и имеющего одну выделенную вершину — корень дерева. Связь между отдельными вершинами дерева обычно описывается в терминах "родитель—потомок". Каждая вершина дерева, кроме корня, связана с одной и только одной родительской вершиной и с некоторым (возможно, нулевым) числом вершин-потомков. При этом каждая вершина является родительской для своих потомков. Корень не имеет родительской вершины.

Будем говорить, что корень составляет нулевой уровень дерева. Непосредственные потомки корня составляют первый уровень дерева. Непосредственные потомки вершин k-го уровня образуют $k+1$-й уровень дерева.

Если каждая вершина дерева имеет не более двух потомков, то это дерево называется бинарным, в противном случае — произвольным (иногда используется термин “сильно ветвящееся дерево”). Для бинарного дерева можно естественным образом ввести понятия левого и правого поддеревьев. Например, мы можем иметь дерево, представленное такой схемой: