Лекция 2. Обработка списков

2.1. Списки

Под списком будем понимать связанный список. Термин "список" в программировании может означать также *линейный список* - конечную последовательность элементов.

Список (*связанный*) можно рассматривать как способ хранения данных в виде последовательности элементов, каждый из которых, кроме информации - значения элемента, содержит указатель местоположения следующего элемента.

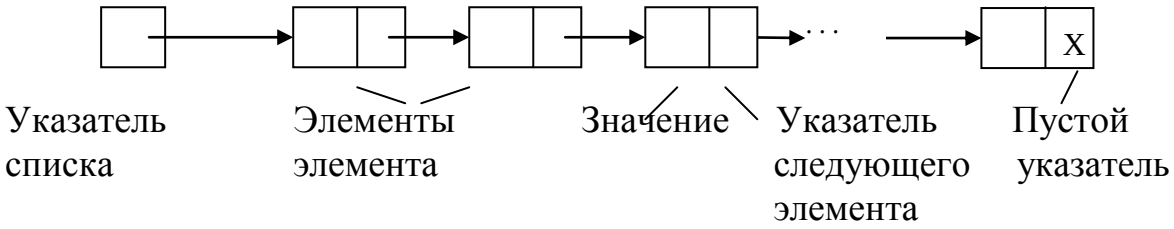


Рис. 2.1. Составные части списка

Последний элемент списка (*хвост*) содержит пустое значение указателя, показанное на рисунках 'X'. Список задается *указателем списка* - ссылкой на его первый элемент - *голову*. Количество элементов обычно не задается. Список может не иметь элементов - быть пустым (его указатель - пустая ссылка).

В виде списка можно хранить разнообразную информацию. Например, на рис. 2.2. показан список, представляющий строку символов (текст), каждый элемент которого содержит один символ в качестве информации и указатель следующего элемента.

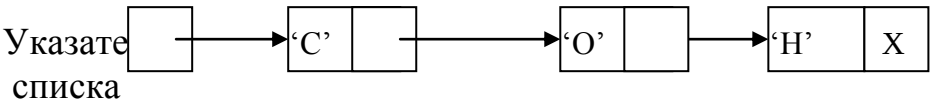


Рис. 2.2. Строка символов в виде списка

К элементам списка возможен только *последовательный доступ* с помощью указателя списка, ссылающегося на его первый элемент. Чтобы обработать, например, десятый элемент, требуется просмотреть предыдущие девять элементов. Другим недостатком списка является затрата памяти для указателей.

На рис. 2.3 показано представление в памяти списка из рис. 2.2. Пустой указатель обозначен нулем.

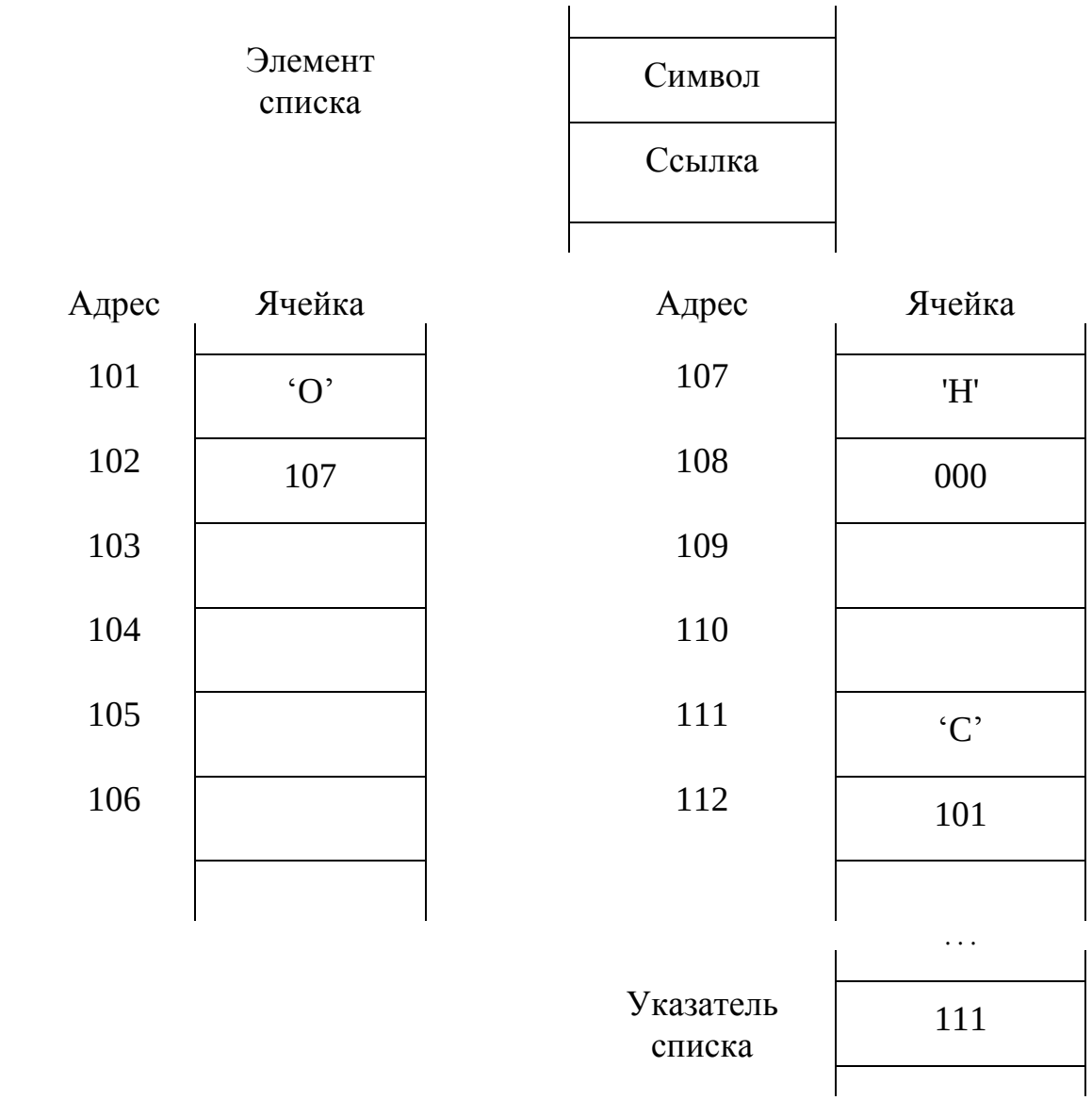


Рис. 2.3. Представление списка в памяти

Основное достоинство списка - возможность размещать его элементы в памяти в любом порядке и не обязательно подряд. Это позволяет легко добавлять и удалять элементы в любом месте списка без физического перемещения данных только за счет изменения значений указателей.

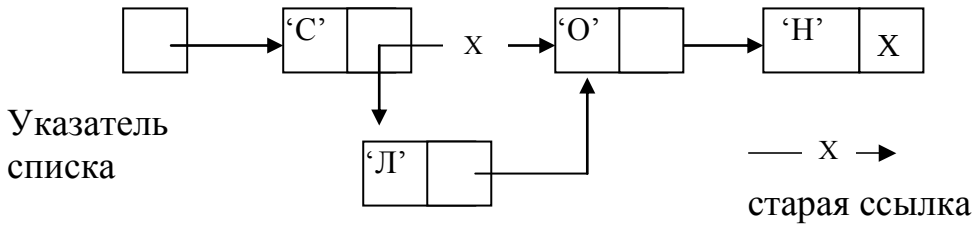


Рис. 2.4. Включение элемента в список

На рис. 2.4 показан список после включения в него элемента, содержащего букву 'Л', на рис. 2.5 - представление этого списка в памяти.

Адрес	Ячейка
101	‘О’
102	107
103	
104	
105	‘Л’
106	101

Адрес	Ячейка
107	'Н'
108	000
109	
110	
111	‘С’
112	101, 105
	…
Указатель списка	111

Рис. 2.5. Включение элемента в список в памяти (101 и 105 - старое и новое содержимое ячейки 112)

Включение элемента в список происходит в три этапа: в свободном месте памяти создается новый элемент; в новый элемент из предшествующего ему элемента списка переносится ссылка на следующий за ним в списке элемент; в предшествующий элемент списка помещается ссылка на новый элемент.

Существуют несколько разновидностей списков. В *двунаправленном списке* каждый элемент содержит ссылку не только на следующий, но и на предыдущий элемент (рис. 2.6).

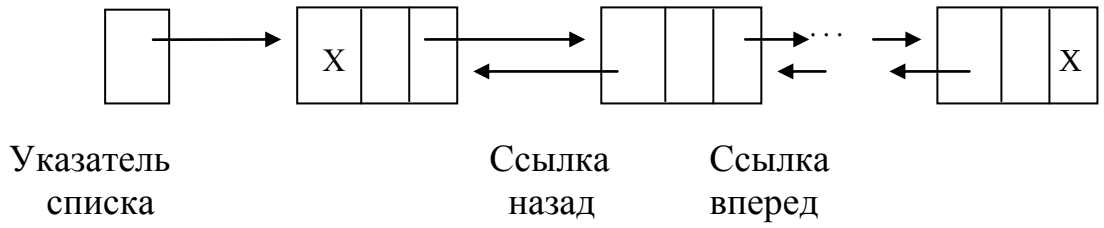


Рис. 2.6. Двунаправленный (*симметричный*) список

В *циклическом списке* последний элемент содержит ссылку на первый элемент списка (рис. 2.7).

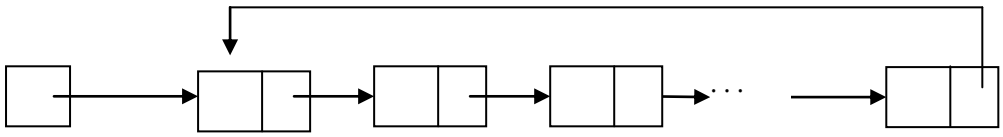


Рис. 2.7. Циклический (*кольцевой*) список

В *списковой структуре* значениями элементов являются указатели списков (рис. 2.8). Списковая структура может иметь сложное разветвленное строение.

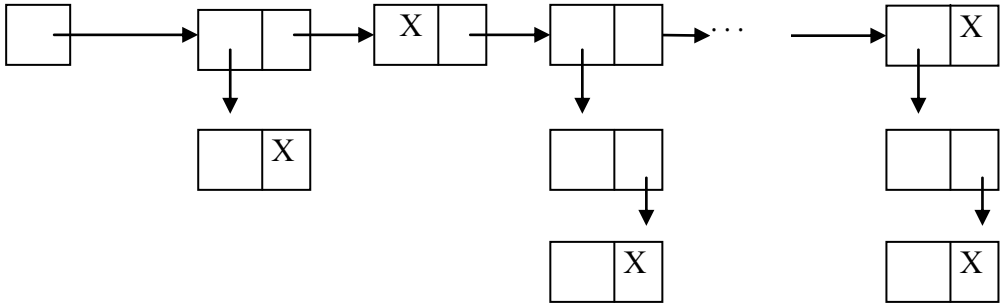


Рис. 2.8. Списковая структура (*список списков*)

Списки удобны в следующих случаях.

1. Для хранения *динамических структур данных*, строение и размеры которых меняются во время выполнения программы: создаются и уничтожаются структуры данных, добавляются и удаляются элементы и т.п.

2. Для упорядочивания данных без их физического перемещения в памяти. При этом элементы данных могут входить одновременно в разные списки, что позволяет упорядочить одни и те же данные несколькими разными способами, "прошивая" их несколькими списками (см. пример 2.14).

Для организации списков требуются языковые средства описания данных, состоящих из разнотипных полей (структуры, записи), и *ссылочный* (адресный) *тип данных*. До 1970-х годов такими средствами обладали лишь специальные языки обработки списковой информации, не получившие широкого распространения. С конца 1960-х годов эти средства стали включать в универсальные языки программирования (PL/1, Pascal, C и др.).

При отсутствии в языке специальных средств, списки можно организовать с помощью параллельных массивов. В этом случае роль указателя играет индекс. Такие средства есть в любом языке высокого уровня, в том числе в языке C.

2.2. Использование ссылочных данных

Для примера предположим, что значением элемента списка является один символ (рис. 2.9).

Организацию таких списков обеспечивают, например, описания на языке C из примера 2.1.

Пример 2.1. Описание данных для списков с использованием ссылочных данных

```
struct el_sp              /* Тип элемента списка                 */
{ char zn;                /* Значение элемента (информация)      */
  struct el_sp *uk;       /* Указатель следующего элемента       */
};
struct el_sp *p;          /* Указатель списка                    */
struct el_sp *i, *j;      /* Указатели элементов списка          */
char sim;
```

Эти описания позволяют использовать в программе обозначения, показанные на рис. 2.9 и в табл. 2.1.

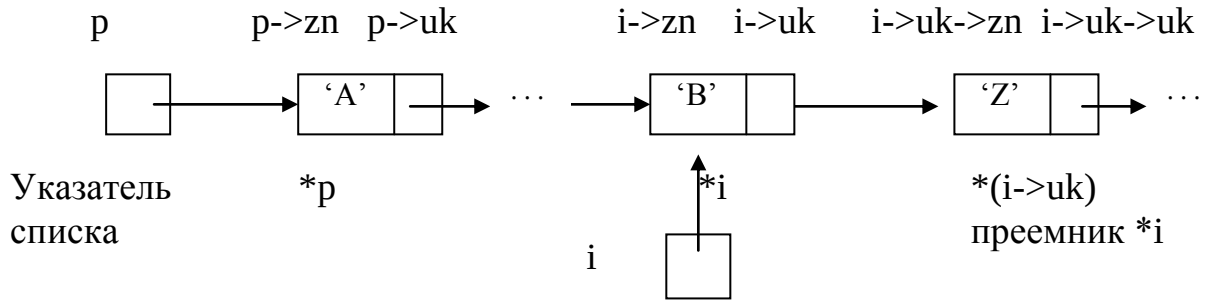


Рис. 2.9. Обозначения для списков (ссылочные данные)

Разумеется, вместо el_sp, zn, uk, p и т. д. можно использовать и другие имена.

Таблица 2.1

Пример обозначений для списков

Обозначения		Обозначаемые величины
Ссылочные данные	Параллельные массивы	
NULL	NOL	Пустой указатель
i, j, p	¦ i, j, p	Указатели элементов (ссылки на элементы)
*i	-	Элемент, на который указывает i
(*i).zn i->zn	zn[i]	Значение (поле zn) элемента *i
(*i).uk i->uk	uk[i]	Ссылка на преемник элемента *i (адрес элемента, следующего в списке за *i)
i->uk->zn	zn[uk[i]]	Значение преемника элемента *i
i->uk->uk	uk[uk[i]]	Ссылка на преемник преемника элемента *i

2.3. Использование параллельных массивов

Значения всех элементов списка можно хранить в одном массиве, а ссылки на следующий элемент - в другом массиве. Массивов может потребоваться и больше: по количеству полей, если значение элемента списка состоит из нескольких полей. Разные поля каждого элемента располагаются в разных массивах с одним и тем же индексом. Количество элементов в этих массивах одинаково. Такие *массивы* на рисунках удобно изображать рядом ("параллельно") и их называют *параллельными*.

В качестве ссылок используются индексы. Роль пустой ссылки может играть 0, -1 (чтобы можно было использовать нулевые элементы) или любое другое значение, не являющееся индексом. Одни и те же массивы образуют область памяти для хранения нескольких списков с одинаковым строением элементов - так называемую ***кучу*** (heap).

Предположим, что значением элемента списка является один символ. На рис. 2.10 показан такой список, содержащий текст "СЛОН" и расположенный в параллельных массивах zn и uk (имена взяты по аналогии с рис. 2.9).

Для организации таких списков можно, например, по аналогии с приведенными ранее обозначениями для ссылочных переменных (пример 2.1), поместить в программу описания из примера 2.2.

Рис. 2.10. Организация списков с помощью параллельных массивов (со списком свободной памяти)

Пример 2.2. Описание данных для списков с использованием параллельных массивов

```
#define N    1000                /* Максимальное кол-во элементов списков      */
```

```
#define NOL     0              /* Пустой указатель                    */
typedef int ukaz;              /* Тип указателя                       */
char zn [N+1];                 /* Значения элементов (информация)     */
ukaz uk [N+1];                 /* Указатели следующего элемента       */
ukaz p;                        /* Указатель списка                    */
ukaz i, j;                     /* Указатели элементов списка          */
char sim;
```

Эти описания позволяют использовать при программировании обозначения, показанные в табл. 2.1. Элемент списка, на который указывает i, в комментариях для краткости будем обозначать *i, т. е. так же, как при использовании ссылочных переменных, хотя по правилам языка С здесь это обозначение бессмысленно и в операторах его применять нельзя.

2.4. Управление памятью *в куче*

Для получения и освобождения памяти *в куче*, размещенной в параллельных массивах, требуется разработка своей *системы управления памятью*. Она должна учитывать свободные и занятые участки кучи, выделять по запросу программы свободный участок для создания нового элемента и освобождать его, когда он становится ненужным. Это можно сделать, сцепив свободные участки в виде списка свободной памяти (рис. 2.10).

Для управления памятью необходимы подпрограммы, аналогичные по назначению и принципам работы библиотечным функциям *динамического распределения памяти* malloc и free языка С. В примере 2.3 это - функции, названные nov_el, osvob и inic_kuchi.

Подпрограмма inic_kuchi создает список свободной памяти, включающий все элементы кучи (кроме 0-го, т. к. 0 играет роль пустой ссылки). Функция nov_el выдает в качестве значения индекс очередной свободной позиции (нулевой индекс обозначает отсутствие свободного места). Подпрограмма osvob вставляет элемент с заданным в виде параметра индексом в список свободной памяти.

Пример 2.3. Система управления памятью для списков примера 2.2

```
 ukaz isv;              /* Указатель списка свободной памяти          */
/* Инициализация кучи - создание списка свободной памяти,    */
/* охватывающего всю кучу (0-й элемент не используется)      */
void inic_kuchi ()
{ for (isv=1; isv<N; isv++)
       uk[isv] = isv + 1;
  uk[N] = NOL;  isv = 1;
}
/* Выделение памяти для нового элемента                      */
ukaz nov_el ()       /* Указатель (индекс) выделенной позиции */
{ukaz i;
```

```
  i = isv;
  if (isv != NOL)    /* Список не пуст                          */
     isv = uk[isv];  /* Удаление 1-го эл-та списка своб.пам     */
  return i;
}
/* Освобождение памяти, выделенной для элемента *i             */
void osvob (ukaz i)
{ /* Вставить *i в начало списка свободной памяти              */
  uk[i] = isv;
  isv = i;
}
```

2.5. Программирование типовых операций

Далее приведены примеры фрагментов программы выполнения типовых операций над списком из примеров 2.1 и 2.2. Алгоритмы выполнения операций для ссылочных переменных и параллельных массивов одинаковы. Отличаются только обозначения операндов (см. табл. 2.1) и функций управления памятью. Поэтому в качестве образца для параллельных массивов даны лишь некоторые операции.

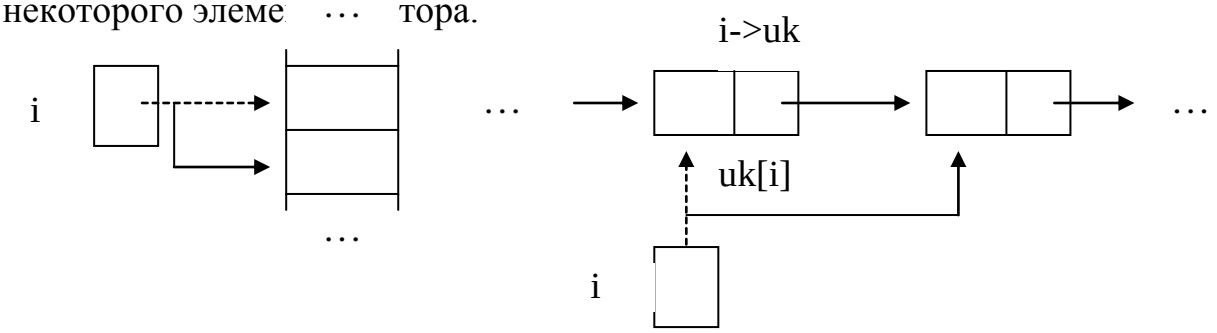
Для параллельных массивов используются функции управления памятью из примера 2.3. В этом случае перед началом работы требуется **инициализация кучи**: inic_kuchi ();

Пример 2.4. Переход к следующему элементу списка (продвижение указателя i к преемнику элемента *i) (рис. 2.11).

а) Ссылочные переменные **б) Параллельные массивы**

```
i = i->uk;          i = uk[i];
```

Эта операция играет роль, аналогичную операции перехода к следующему элементу при обработке вектора: i = i + 1; где i - индекс некоторого элеме … тора.



а) в векторе: i = i+1; б) в списке: i = i->uk; (или i = uk[i];)

Рис. 2.11. Переход к следующему элементу

Пример 2.5. Создание пустого списка с указателем p

а) Ссылочные переменные **б) Параллельные массивы**

```
p = NULL;           p = NOL;
```

Пример 2.6. Создание нового элемента *i и запись в него значения sim

а) Ссылочные переменные

```
i = malloc (sizeof(struct el_sp));      /* Получение памяти                 */
if (i != NULL)                          /* Есть место                       */
  i->zn = sim;                          /* Запись значения                  */
else ...              /* Нет свободной памяти для нового элемента   */
```

Значение функции malloc, имеющее тип char *, перед присваиванием переменной i преобразуется к типу этой переменной, который известен транслятору из ее описания. Поэтому можно не указывать явно эту операцию преобразования типа в виде:

(struct el_sp *) malloc (...)

Тип значения функции malloc транслятор узнает из прототипа (заголовка) этой функции, который необходимо вставить в программу оператором препроцессора #include <stdlib.h> .

б) Параллельные массивы

```
i = nov_el();                   /* Получение памяти                 */
if (i != NOL)                   /* Есть место                       */
   zn[i] = sim;                 /* Запись значения                  */
else ...              /* Нет свободной памяти для нового элемента   */
```

Пример 2.7. Уничтожение элемента *i (освобождение занимаемой им памяти):

а) Ссылочные переменные

```
free (i);
```

б) Параллельные массивы

```
osvob(i);
```

Пример 2.8. Включение элемента *j в начало списка с указателем p (рис. 2.12):

```
j->uk = p;              /* Соединить *j с первым элементом списка p   */
p = j;                  /* Прицепить *j к указателю списка            */
```

Эта операция выполняется правильно и для пустого списка.

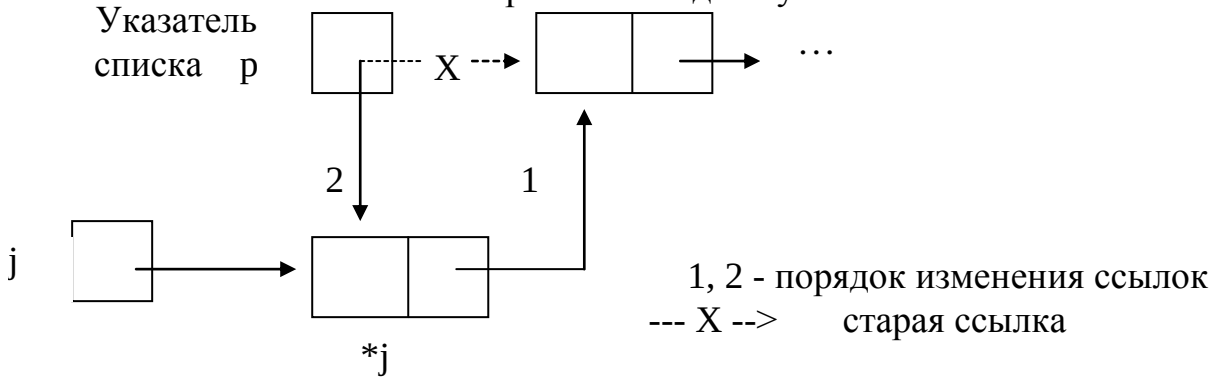


Рис. 2.12. Включение элемента *j в начало списка p

Пример 2.9. Включение элемента *j в список после элемента *i (рис. 2.13). В алгоритме примера 2.8.указатель списка p заменится на i->uk:

```
j->uk = i->uk;          /* Соединить *j с преемником элемента *i */
i->uk = j;              /* Прицепить *j к элементу *i            */
```

Эта операция выполняется правильно и в конце списка, когда элемент *i является последним в списке и не имеет преемника.

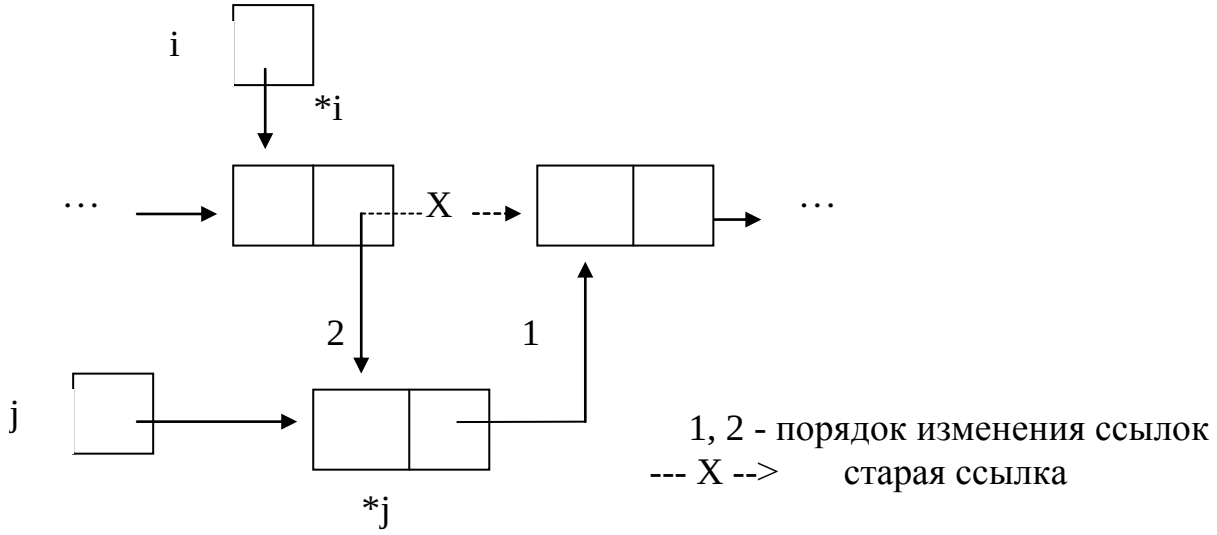


Рис. 2.13. Включение элемента *j в список после *i

Пример 2.10. Исключение первого элемента из списка p:

```
if (p != NULL)          /* Существует первый элемент             */
{   j = p;              /* Запомнить адрес исключаемого эл-та    */
    p = p->uk;          /* Соединить p со вторым элементом       */
    free(j);            /* Освободить память *j                  */
}
else ...                /* Исключение невозможно: список пуст    */
```

Пример 2.11. Исключение из списка преемника элемента *i (в примере 2.11 указатель p заменится на i->uk):

```
if (i->uk != NULL)      /* Существует преемник элемента *i       */
{  j = i->uk;           /* Запомнить адрес преемника *i          */
   i->uk = i->uk->uk;   /* Соединить *i с преемником его преемника */
   free (j);            /* Освободить память                     */
}
else ...                /* Исключение невозможно                 */
```

Если исключаемый элемент не требуется уничтожать (он может, например, входить еще и в другой список), память не освобождают.

Пример 2.12. Включение элемента *j в список перед элементом *i

Операции включения и исключения элемента в списке удобно выполнять после заданного элемента. Чтобы вставить новый элемент перед заданным элементом, можно включить его после заданного элемента, а потом обменять местами значения этих элементов.

Другой способ: просмотрев список с начала, найти предшественника, ссылающегося на заданный элемент, и вставить после него новый элемент. Этот способ удобнее для короткого списка с элементами большого размера.

а) Включение *j после *i и обмен их значений:

```
j->uk = i->uk;              /* Соединить *j с преемником элемента *i      */
i->uk = j;                  /* Прицепить *j к элементу *i                 */
sim = j->zn; j->zn = i->zn; i->zn = sim;      /* Обмен значений           */
```

б) Включение *j после предшественника *i:

```
struct el_sp *к;            /* Указатель предшественника *i               */

if (p == i)                 /* *i - 1-й в списке                          */
{ j->uk = p;   p = j; }     /* Включение *j в начало списка               */
else
{                       /* k = указатель предшественника *i               */
   k = p;                   /* Указатель 1-го элемента                    */
   while (k->uk != i)       /* *k - не предшественник *i                  */
      k = k->uk;            /* К следующему элементу                      */
    /*                  Включение *j после *k                             */
    j->uk = i;              /* Прицепить *i к элементу *j                 */
    k->uk = j;              /* Прицепить *j к элементу *k                 */
}
```

Для исключения из списка заданного элемента также можно найти его предшественника. Другой способ – скопировать в заданный элемент значение его преемника, а затем исключить этого преемника из списка.

Пример 2.13. Проход по списку p (в том числе пустому) с обработкой всех его элементов:

```
i = p;
while (i != NULL)           /* Существует *i (не конец списка)            */
{ . . .                     /* Обработка текущего элемента *i             */
  i = i->uk;                /* Переход к следующему элементу списка       */
}
```

Вариант с оператором for:

```
for (i = p; i != NULL; i = i->uk)
      . . .                 /* Обработка текущего элемента *i             */
```

Пример 2.14. Двойная списковая структура (сеть). В качестве примера применения списков рассмотрим задачу: составить описание данных для такого представления записей об именах и возрасте людей, которое обеспечивает перебор записей как по убыванию / возрастанию возраста, так и в алфавитном порядке имен.

Решением является включение каждой записи в два списка, один из которых упорядочен по возрасту, другой - по именам (по алфавиту). Эту структуру можно считать и сетью, т. к. каждый элемент содержит по две ссылки (рис. 2.14).

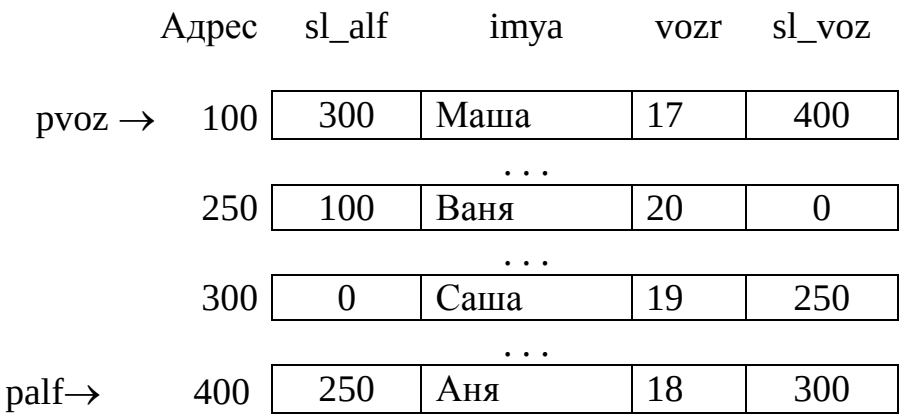


Рис. 2.14. Двойное упорядочивание данных с помощью списков

Определение данных:

```
struct el2sp                    /* Тип элемента списка                    */
{   char *imya;                 /* Имя                                    */
    int vozr;                   /* Возраст                                */
    struct el2sp *sl_alf;       /* Указатель следующего по алфавиту       */
    struct el2sp *sl_voz;       /* Указазатель следующего по возрасту     */
};
struct el2sp *palf;             /* Указатель списка по алфавиту           */
struct el2sp *pvoz;             /* Указатель списка по возрасту           */
```

Предполагается, что записи упорядочиваются при занесении в список, как в примере 5.2. Фрагмент программы, распечатывающий записи в порядке изменения возраста:

```
struct el2sp *i;                /* Указатель текущего элемента списка     */

    . . .

for (i = pvoz;  i != NULL;  i = i->sl_voz;)

    printf ("%s %d\n", i->imja, i->vozr);
```

Вопрос. Как изменится этот фрагмент, если потребуется выводить информацию в алфавитном порядке?

Фиктивные элементы

Чтобы не программировать отдельным вариантами случаи включения и исключения элемента в начале списка или в пустом списке (избавиться от «*краевого эффекта*»), можно хранить указатель списка не в виде скалярной переменной, а в поле ссылки специального *фиктивного элемента* в начале списка (его поле информации не используется).

Аналогичным образом в некоторых случаях можно избавляться от краевого эффекта, размещая в конце списка фиктивный элемент, используемый в качестве «барьера» (sentinel – часовой), предотвращающего выход за пределы списка.

Фиктивный элемент удобен для замыкания двунаправленного списка в кольцо. За время O(1) можно удалить из такого списка любую вершину или объединить два таких списка в один.

Использование фиктивных элементов упрощает алгоритмы, мало влияя на время их работы, но оборачивается дополнительной тратой памяти. Затраты памяти увеличиваются при использовании большого количества коротких списков.

Пример 2.15. Двунаправленный список с одной ссылкой можно реализовать, если в качестве ссылки хранить в каждом элементе поразрядную сумму по модулю два адресов обоих его соседей (использовать битовую операцию неравнозначность, называемую также «исключающим или» и обозначаемую в языке C знаком ^)**.**

Обозначим: T, P, S – адреса текущего, предыдущего и следующего элементов списка; UK – поле ссылки элемента списка. Тогда T->UK = P ^ S.

Прибавляя поразрядно по модулю два к обеим частям этого равенства сначала P, а потом S, получим, соответственно

P ^ T->UK = P ^ P ^ S, T->UK ^ S = P ^ S ^ S

Используя коммутативность (перестановочность) операции ^ и свойство X ^ X = 0 для любого X, отсюда получим

S = P ^ T-> UK, P = S ^ T->UK

По этим формулам переход к следующему и предыдущему элементу («вперед» или «назад») выполняется одинаково:

Адрес соседа = Адрес другого соседа ^ T->UK

и не требуется знать направление перехода. Это полезное свойство отсутствует, если вместо операции ^ использовать разность адресов соседей.

Движение по списку в обоих направлениях в данном случае не требует дополнительных затрат памяти для списка, но усложняет и замедляет операции над ним.

Кроме того, поразрядные операции над адресами (указателями) могут быть запрещены в языке программирования. Тогда придется преобразовывать адреса в целые числа или вместо адресов использовать индексы.

2.6. Задачи

2.1. K-й элемент списка. Дан список с указателем P, представленный ссылочными данными, и целочисленная переменная K. Составить фрагмент программы, который присваивает переменной X информацию K-го элемента списка P - пару вещественных значений.

2.2. Решить задачу 2.1 в виде подпрограммы.

2.3. Изменение последнего элемента списка. Дан список с указателем p, представленный с помощью параллельных массивов. Значением элемента списка является текст из 10 символов. Составить подпрограмму, присваивающую значение величины t последнему элементу списка p.

2.4. Включение в список перед заданным значением. Дан список, представленный с помощью данных ссылочного типа. Значением элемента списка является текст длиной до 10 символов. Составить подпрограмму включения в этот список элемента с заданным значением перед первым элементом, равным другому заданному значению.

2.5. Рекурсивная обработка списка. Составить рекурсивную подпрограмму подсчета количества элементов в данном списке.

2.6. Сбор "мусора". Область для хранения списков (куча) состоит из параллельных массивов z (целочисленные значения элементов) и sled (ссылки на следующий элемент). Составить подпрограмму, прицепляющую к списку свободной памяти с указателем svob все элементы кучи с отрицательными значениями.

2.7. Пусть P и Q - указатели списков, организованных с помощью ссылочных данных (или параллельных массивов); X, A - значения элемента списка; K - целочисленная переменная. Составить описание данных и фрагменты программы (или подпрограммы) для следующих операций.

а) Присвоить переменной X значение
   1) - первого элемента списка P;
   2) - второго элемента списка P;
   3) - последнего элемента списка P.

б) Присвоить значение переменной X
   1) - первому элементу списка P;
   2) - третьему элементу списка P;
   3) - предпоследнему элементу списка P.

в) Удалить из списка P

1) - первый элемент;
2) - второй элемент;
3) - K-й элемент;
4) - последний элемент;
5) - все элементы.

Примечание. При хранении списков в параллельных массивах для включения удаленного элемента в список свободной памяти использовать подпрограмму OSVOB с целочисленным параметром, равным индексу освободившейся позиции.

г) Включить элемент, равный X, в список P

1) - перед первым элементом;
2) - после первого элемента;
3) - после K-го элемента;
4) -после последнего элемента;
5) - после первого элемента, равного A.

Примечание. При хранении списков в параллельных массивах свободное место для нового элемента получать с помощью процедуры NOV_EL с целочисленным параметром, которому при каждом обращении присваивается индекс очередной свободной позиции (нулевой индекс обозначает отсутствие свободного места).

д) Присвоить переменной K количество

1) - элементов в списке P;
2) - положительных элементов в списке P.

е) Вставить список Q в список P

1) - перед первым элементом;
2) - после первого элемента;
3) - после последнего элемента.

ж) Преобразовать список P в циклический список.

з) Преобразовать список P в вектор, записав значения его элементов в массив B.

и) Получить список Q, скопировав в него значения нечетных по номеру элементов списка P (см. примечание к варианту г)).

<u>2.8.</u> Кручу, верчу - запутать хочу. (А.В. Лазарев [81]). В последовательности натуральных чисел 1, 2, ..., N делается M переворотов: порядок чисел с номерами от B_i до E_i меняется на обратный для i=1..M. Даны числа: N, M, B_1, E_1, ..., B_M, E_M ($1 \le B_i \le E_i \le N \le 130000$, $1 \le M \le 2000$). Вывести полученную последовательность. Объем памяти не более 64K.

Примеры ввода	Соответствующий вывод
5 2 1 3 4 5	3 2 1 5 4
5 2 1 4 2 5	4 5 1 2 3