Лекция 3. Методы программирования структур: очереди, стеки, деки, строки, массивы, множества

3.1. Очередь. Стек. Дек

Рассмотрим часто используемые разновидности ***линейного списка*** (конечной последовательности элементов) с разными правилами выполнения операций.

Очередь

Очередь - это упорядоченная последовательность элементов некоторого типа, в которой выполняются операции включения и исключения элемента по принципу FIFO (First-In-First-Out) - "первым пришел - первым ушел": исключение происходит в начале очереди, а включение в конце (рис. 3.1).

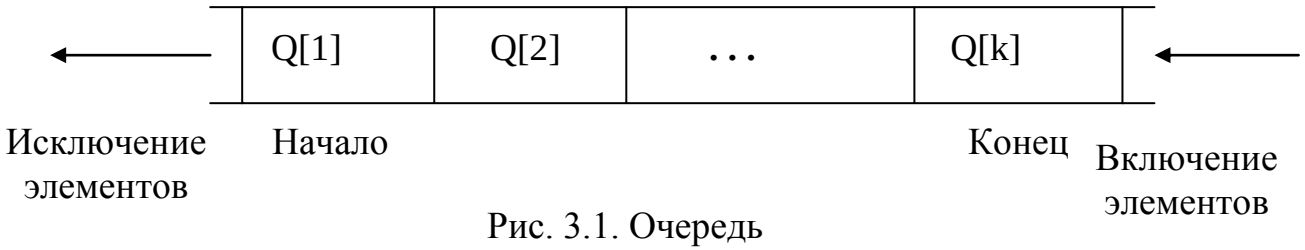


Рис. 3.1. Очередь

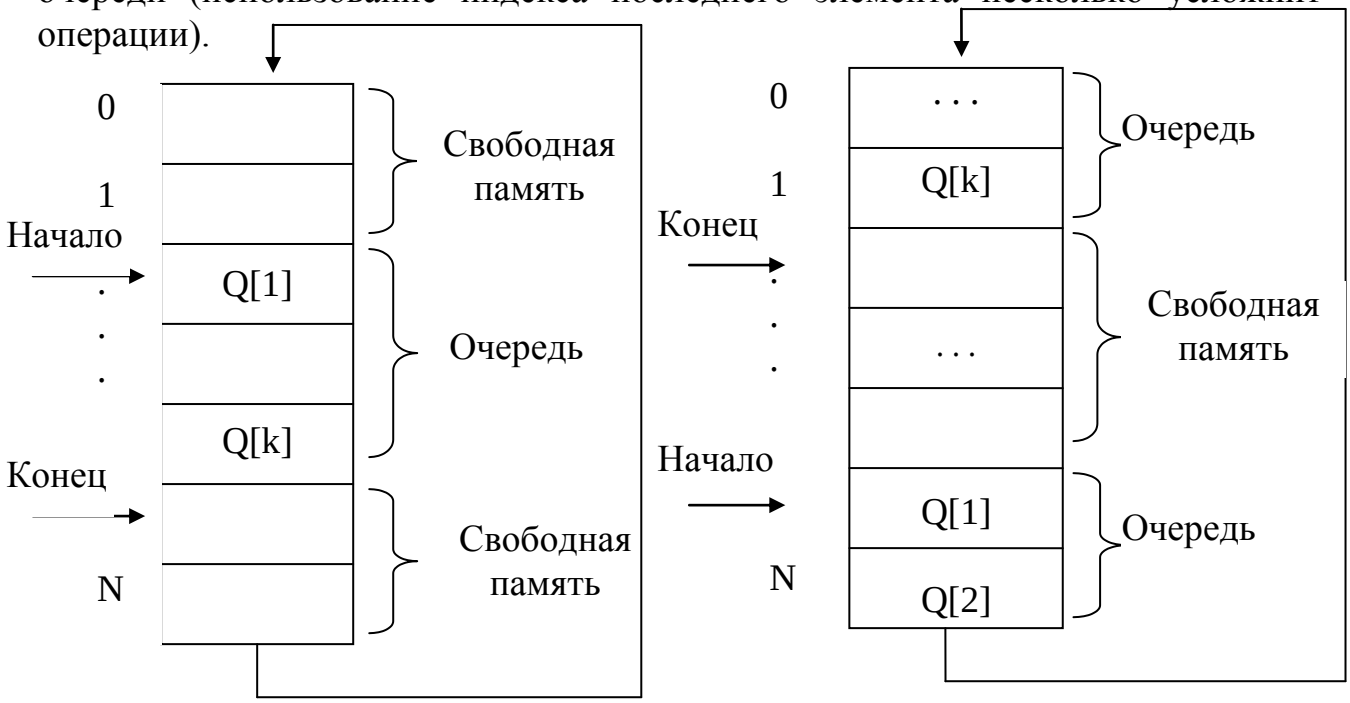
Элементы данных помещаются в очередь, когда скорость поступления данных временно превышает скорость их обработки. Примером очереди является буфер ввода / вывода данных - область памяти для ввода / вывода записей файла параллельно с их обработкой. В операционных системах возникают многочисленные очереди программ: на загрузку в память, на выполнение к процессору, на обработку файлов и т.д. Очередь можно также рассматривать как механизм запоминания подзадач, возникающих при решении некоторой задачи, с последующим решением подзадач в порядке их возникновения. Пример такого применения очереди приведен в алгоритме обхода дерева в ширину.

Типовые операции над очередью:

1. *Инициализация* очереди (создание, подготовка к работе);
2. *Включение* элемента в очередь;
3. *Исключение* элемента из очереди;
4. *Проверка пустоты* очереди;
5. *Проверка переполнения* очереди;
6. *Доступ к началу и концу* (получение / изменение значения первого / последнего элемента).

Представление очереди в виде вектора

Очередь удобно представлять в виде *циклического (кольцевого) вектора* (в котором за последним элементом следует первый), с указателями начала и конца (рис. 3.2). Часть вектора занимает очередь, часть - свободное пространство. *Указателем начала* служит индекс первого из поступивших элементов, *указателем конца* - индекс первой свободной позиции в конце очереди (использование индекса последнего элемента несколько усложнит операции).

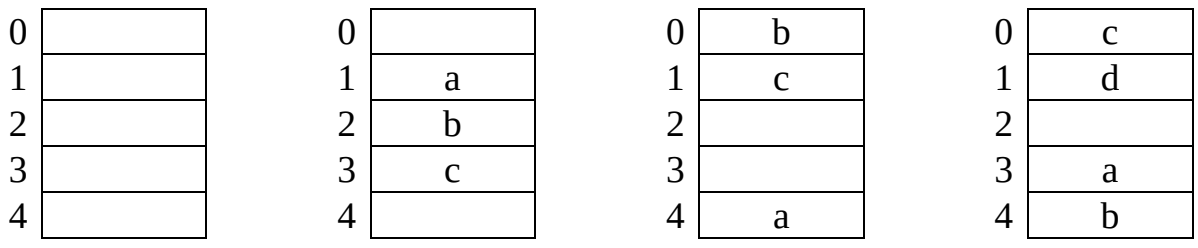


а) Индекс начала меньше индекса конца

б) Индекс начала больше индекса конца

Рис. 3.2. Хранение очереди в виде циклического вектора (показаны два состояния)

Указатель начала увеличивается на одну позицию циклически (за N следует 0) при исключении элемента, указатель конца - при включении. У пустой очереди указатели равны друг другу и могут принимать любое значение от 0 до N. Чтобы отличать максимально заполненную очередь от пустой очереди, оставляют свободным хотя бы один элемент вектора. Поэтому у максимально заполненной очереди указатель конца отстает от указателя начала на одну позицию (рис. 3.3 г).



а) пустая

б) un < uk

в) un > uk

г) полная

очередь очередь:a,b,c очередь:a,b,c очередь:a,b,c,d

Рис. 3.3. Примеры возможных состояний очереди
(un, uk - указатели начала и конца)

Пример 3.1. Описание данных для представления очереди циклическим вектором:

```
#define N    100              /* Максимальная длина очереди                 */
Тип-элемента q[N+1];          /* Отображающий вектор очереди                */
int un;                       /* Указатель начала (индекс первого элемента) */
int uk;                       /* Указатель конца (индекс первого свободного
                                 элемента в конце очереди)                  */
```

Пример 3.2. Инициализация (создание) пустой очереди:

```
un = uk = 0;
```

Пример 3.3. Исключение из очереди элемента и присваивание его величине x (обозначим эту операцию Очередь ==> x):

```
Тип-элемента x;                  /* Значение исключенного элемента      */
        . . .
if (un != uk)                    /* В очереди есть элементы             */
{   x = q[un];                   /* Запоминание значения                */
    if (un < N) un++;   else un=0;   /* Исключение элемента
        */
}
else ...                         /* Пустая очередь                      */
```

Пример 3.4. Включение в очередь элемента, равного x (обозначим эту операцию Очередь<== x):

```
Тип-элемента     x;              /* Включаемое значение                 */
int     i;
  . . .
if (uk < N) i=uk+1; else i=0;    /* Новое значение uk                   */
if (i != un)                     /* Есть место в очереди                */
{   q[uk] = x;                   /* Включение в очередь значения x      */
    uk = i;
}
else   . . .                     /* Переполнение очереди                */
```

При невозможности выполнения операции действия зависят от решаемой задачи. Обычно пустота очереди используется как условие окончания алгоритма, а переполнение означает ошибку.

Представление очереди в виде списка

Можно хранить очередь в виде списка с указателями начала и конца. Операции реализуются методами обработки списков (раздел 2). Чтобы не программировать отдельным вариантом случай включения элемента в пустую очередь, можно хранить указатель начала не в виде скалярной переменной, а в поле ссылки специального *фиктивного элемента* в начале списка (его поле информации не используется). У пустой очереди указатель конца ссылается на этот фиктивный элемент (рис. 3.4). Можно использовать кольцевой список.

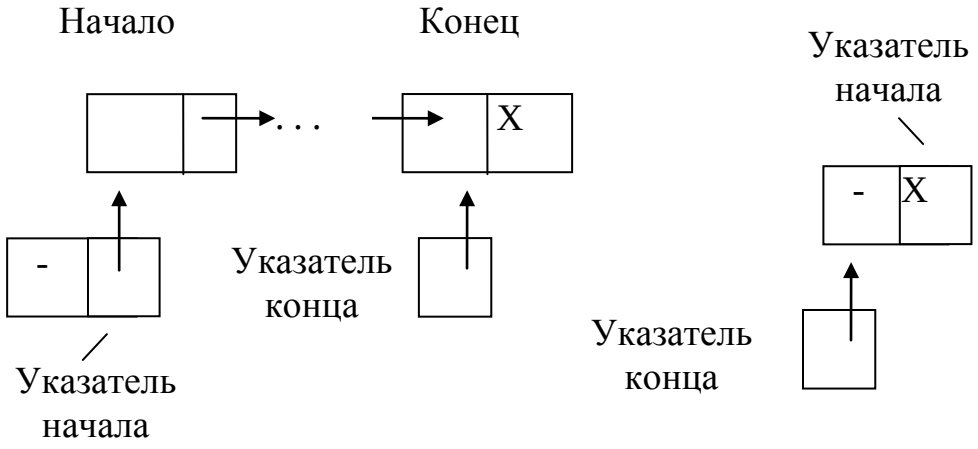


Рис. 3.4. Очередь в виде списка с фиктивным элементом

Представление в виде списка удобно для элементов переменного размера, очереди трудно предсказуемого размера и для очереди с приоритетами.

В *очереди с приоритетами* исключение элементов происходит не в порядке поступления, а *по приоритету* (старшинству). Для этого элементы упорядочиваются в очереди по убыванию приоритета (т.е. включаются не обязательно в конец), что удобно делать в списке. Более эффективна реализация с помощью адресной арифметики, приведенная в разделе 3.3.3 (пример 3.23).

Рассмотрим описание данных и алгоритмы типовых операций для представления очереди списком, как на рис. 3.4.

Пример 3.5. ***Описание данных*** для представления очереди в виде списка:

```
struct el_och                    /* Тип элемента списка          */
{  Тип-элемента inf;             /* Информация                   */
   struct el_och *sled;          /* Ссылка на следующий элемент  */
 };
struct el_och f;                 /* Фиктивный начальный элемент  */
struct el_och *uk;               /* Указатель конца очереди      */
```

Пример 3.6. ***Инициализация*** (создание) пустой очереди:

```
f.sled = NULL;                   /* Указатель начала очереди     */
uk = &f;                         /* Указатель конца очереди      */
```

Пример 3.7. ***Исключение*** из очереди элемента и присваивание его переменной x (обозначим эту операцию Очередь ==> x):

```
Тип-элемента x;                  /* Значение исключенного элемента  */
struct el_och *i;                /* Указатель исключенного элемента */
        . . .
if (f.sled != NULL)              /* В очереди есть элементы      */
{  x = f.sled->inf;              /* Запоминание значения         */
   i = f.sled;
   f.sled = f.sled->sled;        /* Исключение элемента          */
   free (i);                     /* Освобождение памяти          */
   if (f.sled == NULL) uk = &f;  /* Очередь стала пустой         */
}
else ...                         /* Очередь пуста                */
```

Пример 3.8. ***Включение*** в очередь элемента равного x (обозначим эту операцию Очередь <== x):

```
Тип-элемента x;                  /* Включаемое значение          */
struct el_och *i;                /* Указатель включаемого элемента */
        . . .
i = malloc (sizeof(struct el_och));  /* Получение памяти         */
if (i != NULL)                   /* Есть место                   */
{  i->inf = x;                   /* Запись информации            */
   i->sled = NULL;               /* Запись ссылки                */
   uk.sled = i;                  /* Соединить *i с концом очереди */
   uk = i;                       /* Новый указатель конца        */
} else ...                       /* Переполнение очереди         */
```

Стек

Стек (stack) - это упорядоченная последовательность элементов, в которой выполняются операции включения и исключения элемента по принципу LIFO (Last-In-First-Out) - "последним пришел - первым ушел", т. е. включение и исключение всегда происходят в одном конце (рис. 3.5). Этот конец называют *верхом*, противоположный - *дном* стека. Другие названия стека: магазин (по аналогии с магазином пистолета или автомата), очередь типа LIFO.

Рис. 3.5. Стек

Примеры стека: заднее сиденье легкового автомобиля, когда посадка и высадка происходит с одной стороны; стопка подносов в столовой, где подносы берутся и кладутся только сверху.

Назовем некоторые из многочисленных применений стека.

1. Переупорядочивание данных для обработки в порядке, отличающемся от порядка поступления.

2. Запоминание подзадач некоторой задачи с последующим решением подзадач в порядке, обратном порядку их возникновения (см. алгоритм обхода дерева в глубину).

3. Области локальных данных (включая параметры и адреса возврата) вложенных вызовов подпрограмм.

4. Области локальных данных вложенных блоков программы.

5. Трансляция скобочных структур: выражений, вложенных ветвлений, циклов, блоков и всей программы (см. раздел 7).

6. ЭВМ и микрокалькуляторы с безадресной магазинной памятью.

7. Стек в мозге человека (7 ± 2 элемента). Психологи обнаружили, что человек может воспринимать именно такую глубину вложенности, например, придаточных предложений фразы или такое количество рассматриваемых вместе дел, понятий и т. п.

Типовые операции над стеком:

1. *Инициализация* (создание, подготовка к работе);
2. *Вталкивание* (включение) элемента - PUSH;
3. *Выталкивание* (исключение) элемента - POP;
4. *Проверка пустоты* стека;
5. *Проверка переполнения* стека;
6. *Доступ к вершине* (получение / изменение значения последнего поступившего элемента).

Представление стека в виде вектора

Чаще всего стек представляется в виде вектора с указателем (рис. 3.6).

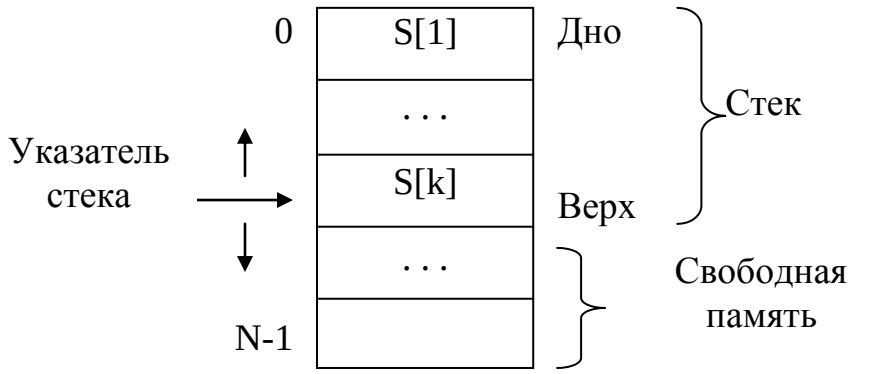


Рис. 3.6. Хранение стека в виде вектора

Часть вектора занимает стек, часть - свободное пространство. *Указателем стека* служит индекс последнего из поступивших элементов (либо индекс первой свободной позиции). Указатель стека увеличивается на единицу при вталкивании элемента и уменьшается при выталкивании (или наоборот, если стек расположен в конце вектора).

Два стека удобно хранить в одном векторе, заполняя их навстречу друг другу (рис. 3.7).

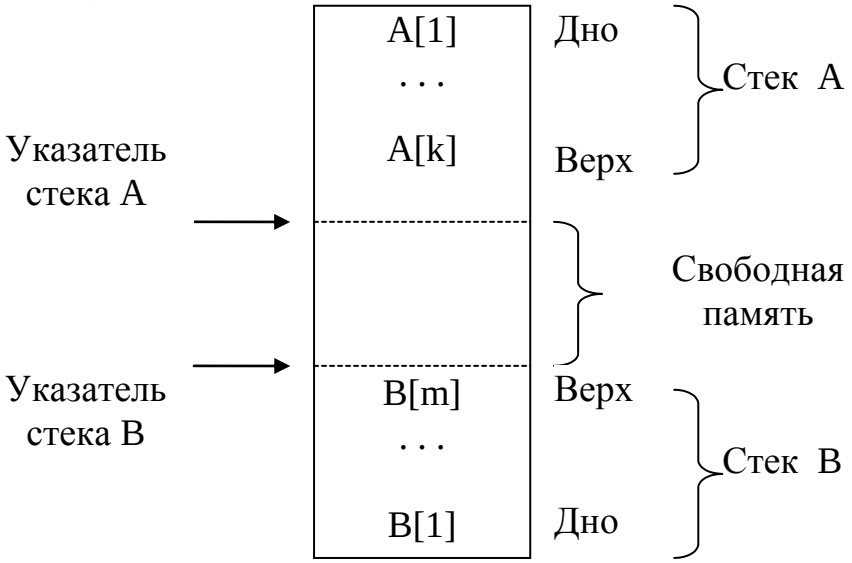


Рис. 3.7. Хранение двух стеков в одном векторе

Этот способ позволяет расти любому из двух стеков, пока есть хотя бы одна свободная ячейка. Максимальные размеры каждого стека и их суммы равны длине вектора. Для большего количества стеков подобного метода не существует.

Пример 3.9. ***Описание данных*** для представления стека вектором

```
#define N 100          /* Максимальная длина стека
      */
Тип-элемента st[N];    /* Отображающий вектор стека                    */
int ist;               /* Указатель стека (индекс последнего элемента) */
```

Пример 3.10. ***Инициализация*** (создание) пустого стека:

```
ist = -1;
```

Как и в очереди, при невозможности выполнения операции действия зависят от решаемой задачи. Обычно пустота стека используется как условие окончания алгоритма, а переполнение означает ошибку.

Пример 3.11. ***Выталкивание*** из стека элемента и присваивание его величине x (обозначим эту операцию Стек ==> x; без запоминания элемента: Стек ==>):

```
Тип-элемента x;          /* Значение вытолкнутого элемента */
        . . .
if (ist >= 0)            /* В стеке есть элементы          */
{  x = st[ist];          /* Получение значения             */
   ist--;                /* Выталкивание элемента          */
}
else . . .               /* Стек пуст                      */
```

Пример 3.12. ***Вталкивание*** в стек элемента, равного x (обозначим эту операцию Стек <== x):

```
Тип-элемента x;          /* Вталкиваемое значение          */
 . . .
if (ist < N-1)           /* Есть место в стеке             */
{  ist++;                /* Увеличение стека               */
   st[ist] = x;          /* Запись x в вершину стека       */
}
else . . .               /* Стек переполнен                */
```

Представление стека в виде списка

В виде списка удобно хранить стек с элементами переменной длины или трудно предсказуемым количеством элементов. Вершина стека размещается в начале списка, где выполнять операции включения и исключения элемента проще, чем в конце (рис. 3.8).

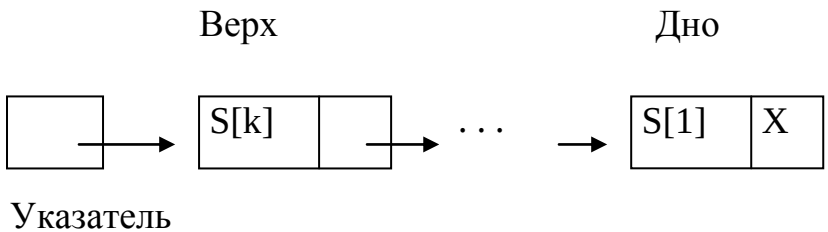


Рис. 3.8. Стек в виде списка

Дек

Дек (deque - double-ended queue: двусторонняя очередь) - это упорядоченная последовательность элементов, в которой включение и исключение элемента могут выполняться в обоих концах (рис. 3.9). Дек является обобщением очереди и стека.

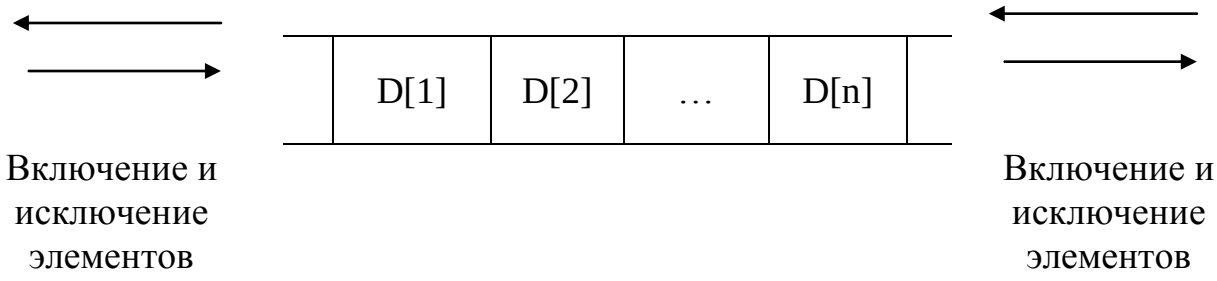


Рис. 3.9. Дек

Назначение дека - переупорядочивание поступающих данных перед обработкой. Стек и дек часто сравнивают с железнодорожными разъездами для перестановки вагонов поезда. Возможности дека по перестановкам больше, чем у стека.

Операции с деком аналогичны операциям над очередью и стеком. Хранится дек, подобно очереди, в виде циклического вектора или списка с двумя указателями.

Строки, массивы, множества

3.2. Строка

Строка - последовательность элементов, доступ к которым - только последовательным перебором в прямом или обратном направлении.

В дальнейшем рассматривается только наиболее распространенный пример строки - *строка символов* (текст). Символьная обработка (обработка

текстовой информации) является важным разделом программирования. Сюда относятся: редактирование текста (поиск, удаление, вставка, замена и перемещение его частей, оформление строчек, абзацев, страниц, документов и др.); трансляция языков программирования, поиск информации в Интернет и т.п. В частности, ввод данных (в том числе числовых) с клавиатуры, а во многих случаях и из файлов, производится в форме текста, который затем преобразуется в другие формы представления.

Примеры символьной обработки даны в разделе 7.

Типовые операции над строками:

1. *Объединение* (сцепление, [кон]катенация) строк.
   Пример: "Петер" + "бург" --> "Петербург"
2. *Разделение* строки на части (выделение подстроки).
3. *Сравнение* строк элемент за элементом.
4. *Замена, перемещение, удаление и вставка* части строки.
5. *Поиск* вхождений одной строки в другую и др.

3.2.1. Представление строк символов

Представление строк включает хранение символов, отдельных строк и совокупностей строк.

1. Кодировка символов

Как правило, код символа имеет фиксированную длину $\log_2 n$ битов (чаще всего 8), где n - количество символов алфавита. На персональных компьютерах наиболее распространен 8-битовый код ASCII (American Standard Code for Information Interchange - Американский стандартный код для обмена информацией). Он позволяет кодировать 256 разных символов, но этого мало для национальных алфавитов и специальных символов (отсюда, например - несколько его вариантов для представления кириллицы).

Поэтому консорциум ведущих фирм разрабатывает совместимый с ASCII международный стандарт 16-битового кода Unicode (65536 кодов, 30000 пока свободны). Он содержит десятки национальных алфавитов, включая китайские иероглифы и забытые языки, математические и технические символы, орнаменты и т. п.

В Unicode ,буквам кириллицы выделены следующие шестнадцатеричные коды: ‘А’ .. ‘Я’ = 0410 .. 042F, ‘а’ .. ‘я’ = 0430 .. 044F, ‘Ё’= 0401, ‘ё’= 0451.

Текст займет меньше памяти при использовании кодов переменной длины: более коротких для часто встречающихся символов и более длинных для редких (коды Хафмана), но они замедляют работу и применяются сравнительно редко.

2. Отдельные строки

На рис. 3.10 показаны разные способы представления отдельных строк на примере слов "Рим" и "Гамбург". Эти способы можно сочетать.

Главная проблема строк - переменная длина. Поэтому *в виде векторов фиксированной длины* хранятся обычно лишь короткие строки, например, имена в трансляторах и операционных системах. Они дополняются пробелами либо усекаются до некоторой постоянной длины (рис. 3.10а).

Недостатки этого представления - неиспользуемые области памяти для коротких строк и возможность потери информации для слишком длинных строк. Изменение размеров вектора уменьшает вероятность одних потерь, но увеличивает вероятность других.

Для устранения этих недостатков используют *векторы переменной длины* со счетчиком символов или признаком конца (рис. 3.10б, 3.10в). Признак конца - особый символ (полезный алфавит уменьшается на один символ).

Признак конца удобнее человеку, например, при вводе данных. Во внутреннем представлении *счетчик символов* дает больше возможностей (обработка строки не только посимвольно, но и целиком; произвольный доступ к символам без опасения выйти за пределы строки), но обычно занимает больше места, чем признак конца (2-4 байта).

Списки удобнее векторов для операций над строками, но требуют памяти для указателей (рис. 3.10г-ж). Двусторонний список поэтому применяется редко. Компромиссом являются списки с многосимвольными элементами (рис. 3.10 е, 3.10 ж).

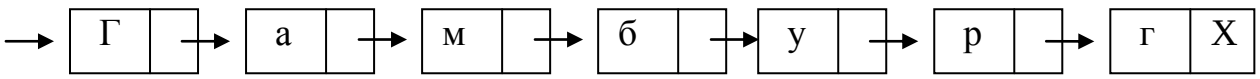
а) Вектор фиксированной длины

Потери памяти Потери данных

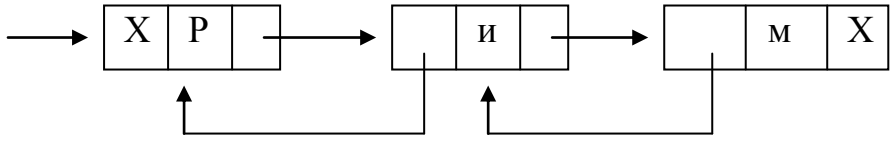
б) Вектор переменной длины со счетчиком символов

в) Вектор переменной длины с признаком конца ∇

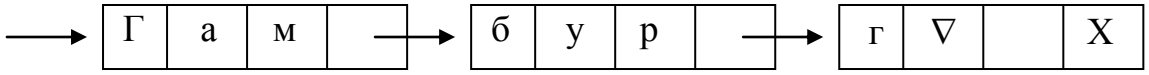
г) Список с односимвольными элементами



д) Двусвязный список с односимвольными элементами



е) Список с элементами фиксированной длины 3



ж) Список с элементами переменной длины и признаком конца ∇

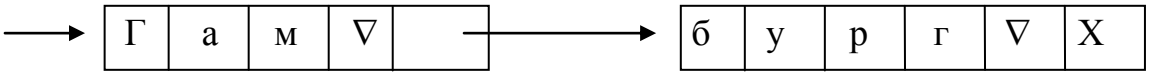
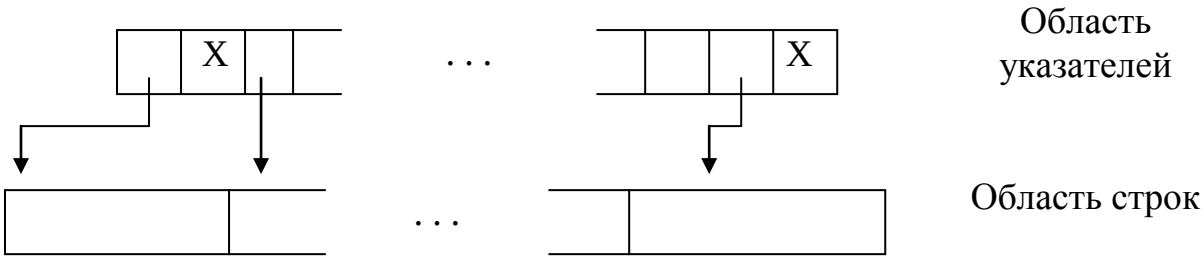


Рис. 3.10. Способы хранения строк

3. Совокупности строк

Совокупность строк обычно хранится в виде двух областей памяти: в одной - сами строки (любым способом), в другой - указатели на них (рис. 3.11). Используется система динамического распределения памяти (см. раздел 2).

а) Векторы указателей на строки



б) Списки указателей на строки

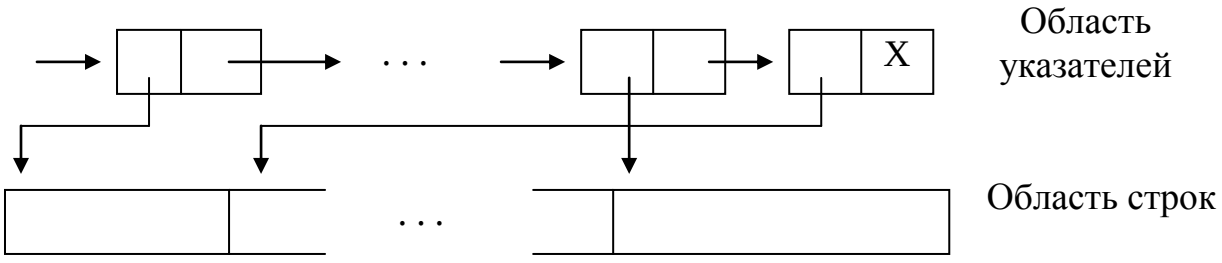


Рис. 3.11. Хранение совокупности строк

Например, при сортировке строк, необходимо передвигать не строки, а указатели на них, организованные в виде вектора или списка.

3.2.2. Строки символов в языках программирования

Строки символов в языке С

В языке С в явном виде строкового типа данных нет. Имеются лишь строковые константы, представляемые в памяти вектором символов с признаком конца '\0': транслятор автоматически добавляет к ним нулевой байт.

строковая-константа ::= "символ..."

Значением строковой константы является указатель (адрес) соответствующего текста. Переменная строка описывается как массив символов. Можно описать строку как указатель на символ (пример 3.9). Тогда может потребоваться запрос памяти для нее (функция malloc()). Если необходимо использовать библиотечные функции над строками, программист должен заботиться о наличии в строке признака конца '\0'. В иных случаях можно применять и другое представление строк.

Пример 3.13. Строки в языке С

```
char *p;              /* Указатель на символ                              */
 ...
p= "КАИ";             /* Адрес текста, занимающего 4 байта: 'К', 'А', 'И', '\0' */
printf (p);           /* Эквивалентно printf ("КАИ");                     */
```

Системная библиотека содержит функции над строками символов, рассчитанные на представление строк с нулевым признаком конца (как у строковых констант), например: определение длины строки strlen, копирование строки strcpy, сцепление строк strcat, сравнение строк strcmp, поиск в строке указанного символа strchr и др. Для использования этих функций в программу требуется включить их прототипы командой #include <string.h>.

Пример 3.14. Использование строковых функций в языке С

```
char t1[81], t2[51], t3[81], *p;
gets(t1);                    /* Ввод в t1 строки до '\n', '\t' или пробела   */
if (strlen(t1) < 51)         /* Длина строки < 51                            */
strcpy (t2, t1);             /* Копирование строк: t1 в t2                   */
if ((p=strrchr(t2,'.'))!=NULL)  /* Есть точка в строке t2 */
{  *p = '!';                 /* Замена в строке t2 правой точки на '!'       */
   *(p+1) = '\0';            /* Удаление части t2 после правой точки         */
}
gets(t3);
if (strcmp(t1,t3)==0)  puts ("Строки равны");
```

Строки символов в языке Pascal

В стандартном языке Pascal отсутствует строковый тип, строка символов описывается как упакованный массив символов. В отличие от других массивов, такой массив, кроме посимвольной обработки, может целиком участвовать в некоторых операциях: ввода, вывода, присваивания ему строковой константы (но не переменной!) такой же длины, как у него.

Пример 3.15. Упакованные символьные массивы в языке Pascal

```
var t: packed array [1..4] of char;        { t - текст длиной 4          }
 . . .
read (t);                                  { Ввод строки из 4 символов   }
if (t[1]='К') and (t[2]='Г')
then t := 'КГТУ';                          { Нельзя 3 символа: t := 'КАИ' }
writeln (t);                               { Вывод строки t              }
```

В Turbo-Pascal имеется тип данных string - строка переменной длины: от 0 до max-длина символов.

описание-строки ::= string [[_max-длина_]]

max-длина = 1..255, по умолчанию 255.

Символы строки записываются как элементы массива с индексами от 1 до максимальной длины. Дополнительный нулевой элемент (байт) содержит длину строки, но для использования его требуется преобразовать из символьного типа в целочисленный с помощью функции Ord или Length.

3.2.3. Операции над строками символов

1. Сравнение строк

а) *точное сравнение* производится на полное совпадение строк или для определения, какая из них предшествует другой лексикографически (как в словарях). В языке С для этого служит функция strcmp.

б) *приближенное сравнение* используется при поиске слова, точное написание которого неизвестно, и желательно найти похожие на него слова (варианты написания) с измененными, пропущенными или переставленными буквами: Вильсон вместо Уилсон, Высотский вместо Высоцкий и т. п. Для этого есть много методов. Меру близости слов X и Y (расстояние между ними) можно, например, определять по формуле:

$$D(X,Y) = (1-K/(DX-1))*(1-K/(DY-1)) \quad (3.1)$$

где DX, DY - длины слов; K - число пар соседних букв, входящих в оба слова одновременно; D(X,Y) = 0..1.

Пример: D("Саша","Маша") = (1-2/(4-1))*(1-2/(4-1)) = 1/9.

в) Другая разновидность приближенного сравнения - *сопоставление слова с шаблоном* (образцом). Так, в командах операционных систем UNIX и MS DOS имя файла может содержать метасимволы: '*' обозначает любое количество любых символов, '?' - один любой символ. Такое имя является шаблоном и заменяется последовательностью соответствующих ему имен файлов заданного каталога.

Например, имя a*.c обозначает все файлы с расширением "c", имена которых начинается на букву "a"; *.* - все файлы; ??.txt - все файлы с расширением "txt", имена которых состоят из двух символов.

2. Объединение и разъединение, вставку и удаление строк удобнее выполнять при списковом представлении строк, но для коротких строк вполне возможна и векторная реализация.

3. *Поиск строки* S длиной M в тексте T длиной N - определение индекса первого вхождения.

а) *Прямой метод* - просмотр текста и строки слева направо с продвижением на один символ. В худшем случае (когда T содержит N-1 букв 'A' и одну 'B', S содержит M-1 букв 'A' и одну 'B') требует порядка O(N*M) сравнений.

б) *БМ-алгоритм* (Боуер и Мур, 1975) - просмотр текста слева направо, строки справа налево; при несовпадении строка сдвигается на расстояние, равное смещению от конца строки до символа текста в ней. Почти всегда меньше N сравнений, в лучшем случае O(N/M) сравнений.

в) *КМП-алгоритм* (Кнут, Моррис и Пратт, 1970) - при несовпадении строка сдвигается на длину максимальной предшествующей подстроки, совпадающей с началом строки. Требует порядка O(N+M) операций.

3.3. Массив

Массив – это набор однотипных элементов, с каждым из которых связан упорядоченный набор из одинакового количества целочисленных индексов, определяющих положение элемента в массиве.

Размерность (число измерений) массива – это количество индексов каждого элемента. Массив с n индексами называют n-мерным. Одномерный массив называют вектором, двумерный - матрицей. Матрица состоит из строк и столбцов. Первый индекс - номер строки, второй - номер столбца.

В *прямоугольном массиве* каждый индекс изменяется с постоянным шагом (обычно 1) от нижней до верхней границы, причем границы не зависят от других индексов.

Пример 3.16. Непрямоугольный массив - матрица Y - состоит из следующих элементов

	Y[1,2]	Y[1,3]	
Y[2,1]	Y[2,2]	Y[2,3]	Y[2,4]
Y[3,1]	Y[3,2]		

Здесь пределы изменения номера столбца зависят от номера строки: в 1-й строке от 2 до 3, во 2-й строке от 1 до 4, в 3-й – от 1 до 2. Аналогично диапазон номеров строки зависит от номера столбца: в 1-м столбце – от 2 до 3, во 2-м столбце – от 1 до .3 и т. д.

Массив представляет множество элементов информации, обрабатываемых с прямым доступом (в произвольном порядке).

Массивы имеются во всех языках высокого уровня, причем в большинстве языков, в том числе C и Pascal, используются только прямоугольные массивы. Обычно допускаются только поэлементные операции, а над всем массивом иногда разрешается выполнять ввод/вывод.

В некоторых языках (APL, PL/1 и др.) имеются групповые операции над всем массивом. В языке ассемблера программисту самому приходится реализовать массив.

Базовой операцией над массивом является доступ к элементу по его индексам для определения его адреса и получения или изменения его значения.

3.3.1. Хранение прямоугольных массивов

Для примера используем язык Pascal, допускающий массивы с отрицательными, нулевыми и положительными индексами, т.е. более общего вида чем большинство других языков.

Прямоугольный n-мерный массив X определяется на языке Pascal следующим образом

$$X: \text{array } [I_1 \,..\, K_1, \ldots, I_n \,..\, K_n] \text{ of тип-элемента,} \qquad (3.1)$$

где I_m, K_m - начальная и конечная границы (*граничная пара*) m-го индекса (m = 1, 2, …, n).

Массив обычно хранят в виде вектора (его называют *отображающим* век*тором)*. Пронумеруем элементы этого вектора от 0 до N-1, где N - количество элементов массива:

$$N = \prod_{m=1}^{n} (K_m - I_m + 1). \qquad (3.2)$$

Здесь Π обозначает произведение (аналогично сумме Σ), K_m-I_m+1 равно количеству возможных значений m-го индекса.

Объем занимаемой массивом памяти V равен

$$V = N * d, \quad \text{где d - длина элемента (количество ячеек).} \qquad (3.3)$$

Обычно элементы массива размещаются в памяти по возрастанию индексов, начиная с элемента $X[I_1,...,I_n]$, имеющего минимальные индексы. Тогда, считая от 0, порядковый номер L элемента $X[J_1,...,J_n]$ в памяти равен:

$$L = \sum^{n} D_m * (J_m - I_m), \qquad (3.4)$$

m=1

где D_m – перемещение по памяти при изменении индекса J_m на 1, выраженное количеством элементов,

Обычно элементы размещают “*строками”* (когда при просмотре элементов в порядке увеличения адресов быстрее изменяется правый индекс). В этом случае

$$D_n = 1; \qquad D_{m-1} = D_m * (K_m - I_m + 1), \qquad (m = 2,..., n). \qquad (3.5)$$

При размещении “*столбцами*” (когда быстрее изменяется левый индекс)

$$D_1 = 1; \qquad D_m = D_{m-1} * (K_{m-1} - I_{m-1} + 1), \qquad (m = 2,..., n). \qquad (3.6)$$

Согласно формуле (1.1) адрес элемента $X[J_1,...,J_n]$ с номером L: равен

$$адрес(X[J_1,...,J_n]) = адрес\ (X[I_1,...,I_n]) + d*L.$$

Из формулы (3.4) получим:

$$адрес(X[J_1,...,J_n]) = B + C + d * \sum_{m=1}^{n} D_m*J_m, \qquad (3.7)$$

где $B = адрес\ (X[I_1,...,I_n])$ - база отображающего вектора,

$$C = - d * \sum_{m=1}^{n} D_m*I_m \qquad (3.8)$$

Числа D_m и C зависят только от строения массива - граничных пар его индексов и длины элемента. Граничные пары используются также для контроля правильности индексов при выполнении программы.

Таким образом, для вычисления адреса элемента по его индексам нужна база B отображающего вектора и *дескриптор массива* - *определяющий вектор* из 3*n+2 целых чисел:

$$n;\ C;\ n\ чисел\ d*D_m;\ n\ граничных\ пар\ I_1,\ K_1,\ ...,\ I_n,\ K_n.$$

Для массивов с одинаковыми описаниями транслятор создает общий дескриптор.

Для доступа через дескриптор к элементам массива перед каждой операцией с элементом транслятор вставляет довольно громоздкий фрагмент программы, вычисляющий адрес этого элемента по формуле (3.7) и содержащий n сложений и n умножений, а для проверки правильности индексов требуется еще 2*n сравнений.

Таким образом, любая операция с элементом массива, особенно многомерного, требует много памяти и времени. Поэтому следует минимизировать число таких операций, особенно внутри циклов.

Пример 3.17. ***Определение дескриптора массива.*** Требуется составить функцию зависимости адреса элемента от его индексов для доступа через дескриптор к элементам целочисленного массива с данным Pascal-описанием

```
Y: array [1 .. 3,    0 .. 2,    2 .. 3] of  integer;
```

$I_1\ K_1 \quad I_2\ K_2 \quad I_3\ K_3$

Здесь n = 3, d = объем(integer) = 2 байта.

Если быстрее изменяется правый индекс, то по формулам (3.5) и (3.8) найдем

$D_3 = 1, \quad D_2 = D_3*(K_3-I_3+1) = 1*2 = 2, \quad D_1 = D_2*(K_2-I_2+1) = 2*3 = 6,$

$C = - d*(D_1*I_1 + D_2*I_2 + D_3*I_3) = -2*(6*1 +2*0 + 1*2) = -16.$

Подставив найденные величины в формулу (3.7), получим требуемое выражение

адрес(Y[i, j, k]) = адрес(Y[1,0,2]) - 16 + 12*i + 4*j + 2*k.

Для случая, когда быстрее изменяется левый индекс, аналогичным образом по формулам (3.6), (3.8) и (3.7) получим

$D_1 = 1, \quad D_2 = D_1*(K_1-I_1+1) = 1*3 = 3, \quad D_3 = D_2*(K_2-I_2+1) = 3*3 = 9,$

$C = - d*(D_1*I_1 + D_2*I_2 + D_3*I_3) = -2*(1*1 +3*0 + 9*2) = -38.$

и окончательный результат

адрес(Y[i, j, k]) = адрес(Y[1,0,2]) - 38 + 2*i + 6*j + 18*k

Вместо использования формулы (3.7) и дескриптора можно (как делается, например, в языке C) рассматривать многомерный массив как одномерный массив массивов, а индексные скобки [ ] как операцию доступа к элементу:

$$A[J] = *(A + J), \qquad (3.9)$$

где имя массива A обозначает базовый адрес массива; индекс J автоматически умножается на длину элемента массива (размер объекта, адресуемого указателем A); *(адрес) обозначает операцию обращения по адресу – значение элемента с заданным адресом. При этом индексы изменяются от 0, а операции [ ] выполняются слева направо. Например, трижды применяя формулу (3.9), для трехмерного массива Z получим:

$$Z[i][j][k] = *(*(*(Z + i) + j) + k) \qquad (3.10)$$

Для n-мерного массива в этом случае, как и при доступе через дескриптор по формуле (3.7), потребуется n сложений, но вместо n умножений используется такое же количество обращений по адресу, выполняемых на порядок быстрее, чем умножения.

Этот метод позволяет экономить время, но требует больше памяти (для хранения структуры указателей).

3.3.2. Хранение непрямоугольных массивов

Формулу (3.10) можно рассматривать как частный случай метода Айлифа (Iliffe), применимого и для непрямоугольных массивов. Рассмотрим этот метод на примере.

Пример 3.18. ***Представление непрямоугольного массива*** На рис. 3.12 справа показан трехмерный непрямоугольный массив T, размещенный в памяти при более быстром возрастании правого индекса.

Этот массив по своему строению аналогичен матрице Y из примера 3.16, но является трехмерным. Его можно представить себе в виде параллелепипеда, у которого некоторые углы “вырезаны”, причем неправильным бессистемным образом.

Обозначим индексы массива буквами i, j, k. Индекс i изменяется от 2 до 3. При i=2 индекс j пробегает значения от –1 до 1, а при i=3 индекс j изменяется уже в другом диапазоне: от 1 до 2. Пределы индекса k также зависят от значений других индексов, например, при i=2 и j=-1 индекс k изменяется от 1 до 2, а при i=2 и j=0 уже в другом диапазоне: от 0 до 2 и т.д.

Слева на рис. 3.12 изображены “*векторы Айлифа*”, образующие древовидную структуру ссылок для доступа к элементу массива троекратным применением формулы (3.9) подобно (3.10).

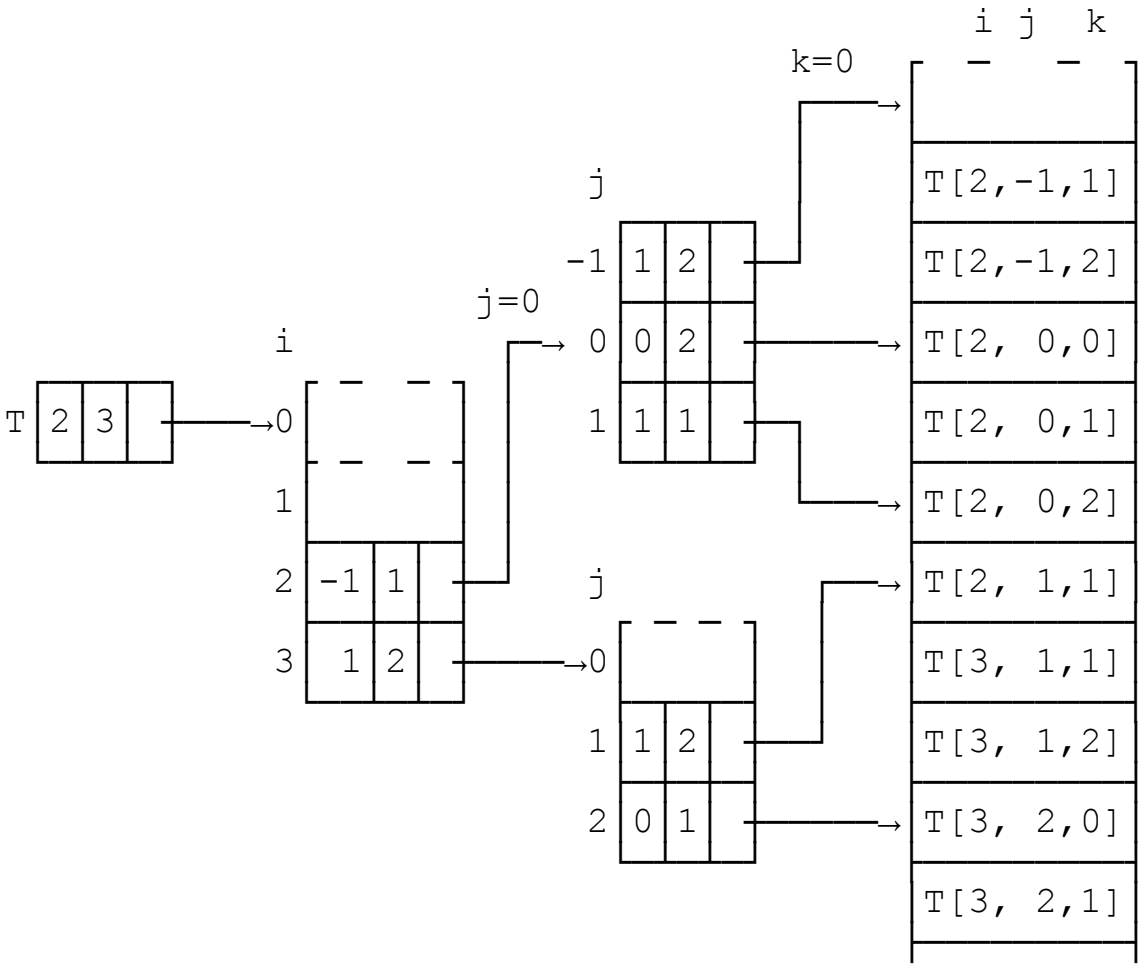


Рис. 3.12. Векторы Айлифа для непрямоугольного массива

Каждая ссылка показывает на нулевое значение соответствующего индекса. Кроме ссылки, для проверки корректности этого индекса вектор Айлифа содержит его граничную пару.

Для прямоугольного массива несколько векторов Айлифа содержали бы одинаковые границы индекса. Чтобы экономить память, в случае прямоугольного массива в древовидной структуре хранятся только указатели, а граничные пары располагаются в отдельной области. Векторы Айлифа обеспечивают более быстрый доступ к прямоугольному массиву, но требуют больше памяти по сравнению с дескриптором.

Векторы Айлифа занимают меньше памяти, когда диапазон изменения индексов увеличивается слева направо, и наименее экономичны при наибольшем диапазоне у первого индекса.

Векторы Айлифа можно рассматривать как еще один пример древовидной таблицы, отличающейся от дерева бинарного поиска (раздел 5.5). Ключом служит набор индексов. Каждый индекс образует как бы "цифру" ключа и такую таблицу можно назвать *деревом цифрового поиска*. Идею цифрового поиска можно применять и для текстового ключа, роль "цифр" которого играют его символы.

Разреженный массив, большинство элементов которого равны одному и тому же значению (например, 0) и лишь немногие "значащие" элементы отличаются от него, хранят в виде таблицы "значащих" элементов. Ключом служит набор индексов. Здесь могут быть удобными сети и списковые структуры.

В следующем разделе рассматриваются методы работы с другими нестандартными массивами, в частности, с упакованными массивами, хранимыми по несколько элементов в ячейке (см. также пример 1 из раздела 3.4).

3.3.3. Примеры адресной арифметики

Пример 3.19. ***Треугольная матрица*** содержит элементы, расположенные выше главной диагонали квадратной матрицы размером N*N, включая диагональ, и хранится по строкам в виде вектора. Выразить индекс элемента в векторе в зависимости от его индексов в матрице.

Решение. Матрица содержит элементы:

```
X[0,0]  x[0,1]  x[0,2]  . . .          x[0,j]  . . .  x[0,N-1]
        x[1,1]  x[1,2]  . . .          x[1,j]  . . .  x[1,N-1]
                . . .
                        x[i,i]  . . .  x[i,j]  . . .  x[i,N-1]
                                       . . .
                                                      x[N-1,N-1]
```

Если считать от нуля, порядковый номер k элемента x[i,j] в памяти равен количеству расположенных до него элементов:

k = длина 0-й строки + ... + длина (i-1)-й строки + j - i =
= N + N-1 + N-2 + ... + N-(i-1) + j - i

Поскольку сумма арифметической прогрессии $a_1 + ... + a_n$ равна

$$(a_1 + a_n) * n / 2,$$

получим

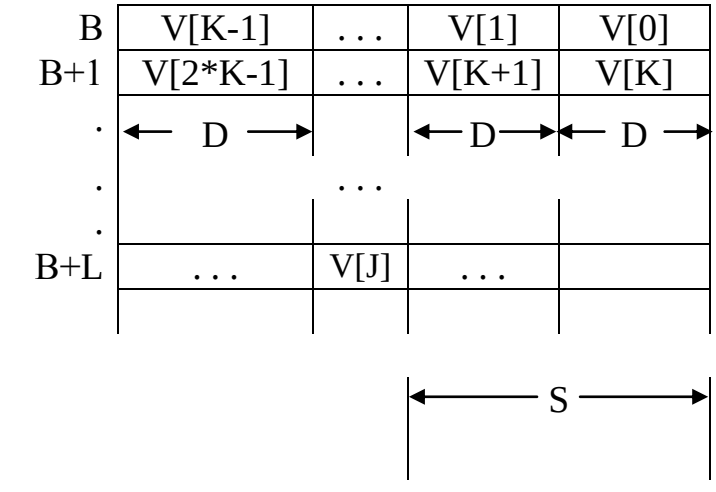
k = (N + (N-i+1)) * i / 2 + j - i = (2*N-i+1)*i/2 +j-i .

Пример 3.20. ***Упакованный вектор***. Написать выражения, определяющие адрес элемента вектора V[J] и его сдвиг S от начала ячейки в зависимости от индекса элемента J, адреса B начала вектора и длины D его элементов в битах для случая, когда в одной ячейке размещается целое число K элементов ($K \geq 1$).

Решение. На рис. 3.13 а показано размещение элементов вектора V в ячейках памяти. Допустим, элемент V[J] находится в ячейке B+L. Из рис. 3.13 а получена таблица зависимости L и S от J (рис.3.13 б). Из этой таблицы видно, что L=J / K и искомые зависимости выражаются формулами:

Адрес V[J] = B + J / K, S = J % K * D (3.11)

а) Размещение элементов



б) Зависимость L и S от J

J	L	S
0	0	0
1	0	D
2	0	2*D
. . .	. . .	. . .
K-1	0	(K-1)*D
K	1	0
K+1	1	D
K+2	1	2*D
. . .	. . .	. . .
2*K-1	1	(K-1)*D
. . .	. . .	. . .

Рис. 3.13. Хранение упакованного вектора

Пример 3.21. ***Упаковка и распаковка***. В каждом из двухбайтовых элементов массива упакованы несколько целых чисел, имеющих значения от 2000 до 2007. Составить фрагменты программы

- упаковки числа: присвоения данного значения числу с указанным номером;
- распаковки числа: получения значения числа по его номеру.

Решение. Диапазон от 2000 до 2007 содержит 8 значений. Такое число можно разместить в 3 бита, если хранить его уменьшенным на 2000. Тогда в каждом двухбайтовом элементе массива будут храниться по 5 трехбитовых чисел.

Воспользовавшись формулами (3.11), получим фрагменты программы:

```
#define  N 100       /* Количество чисел / 3                  */
unsigned x[N],       /* x содержит 3*N чисел v[i]             */
         z,          /* Значение числа (в 3 младших битах)    */
         m,          /* Заданный номер числа                  */
         j,          /* x[j] содержит v[m]                    */
         s;          /* Расстояние v[m] от правого края x[j]  */
            . . .
/*                    а) Упаковка: v[m] = z                   */
j = m/5;  s = m%5*3;
x[j]=x[j] & !(07 << s);     /* Очистка в x[j] места для v[m] */
x[j]=x[j] | (z-2000)<<s;    /* Запись z в x[j] на место v[m] */
            . . .

/*                    б) Распаковка: z = v[m]                 */
j = m/5;  s = m%5*3;
z = (x[j]>>s & 07) + 2000;     /* Получение числа v[m]        */
```

Пример 3.22. ***Быстрый поиск минимума***. Требуется реализовать массив X из n элементов с быстрым получением минимального элемента, т.е. структуру данных с операциями:

- подготовка к работе — за время O(n),
- присвоить j-му элементу заданное число — за время O(log n),
- получить значение j-го элемента — за время O(log n),
- получить номер минимального элемента (или одного из минимальных элементов) — за время O(log n).

Решение. Надстроим над элементами массива как над листьями бинарное дерево. В каждой вершине дерева хранится меньшее из значений ее сыновей, которое является также минимальным элементом соответствующего поддерева (рис. 3.14).

Корректировка этой информации при изменении значения какого-либо элемента массива, а также прослеживание пути из корня к минимальному элементу требуют числа действий порядка log n. Это быстрее, чем обычный последовательный поиск минимума в массиве за время O(n). Аналогично определяется максимум.

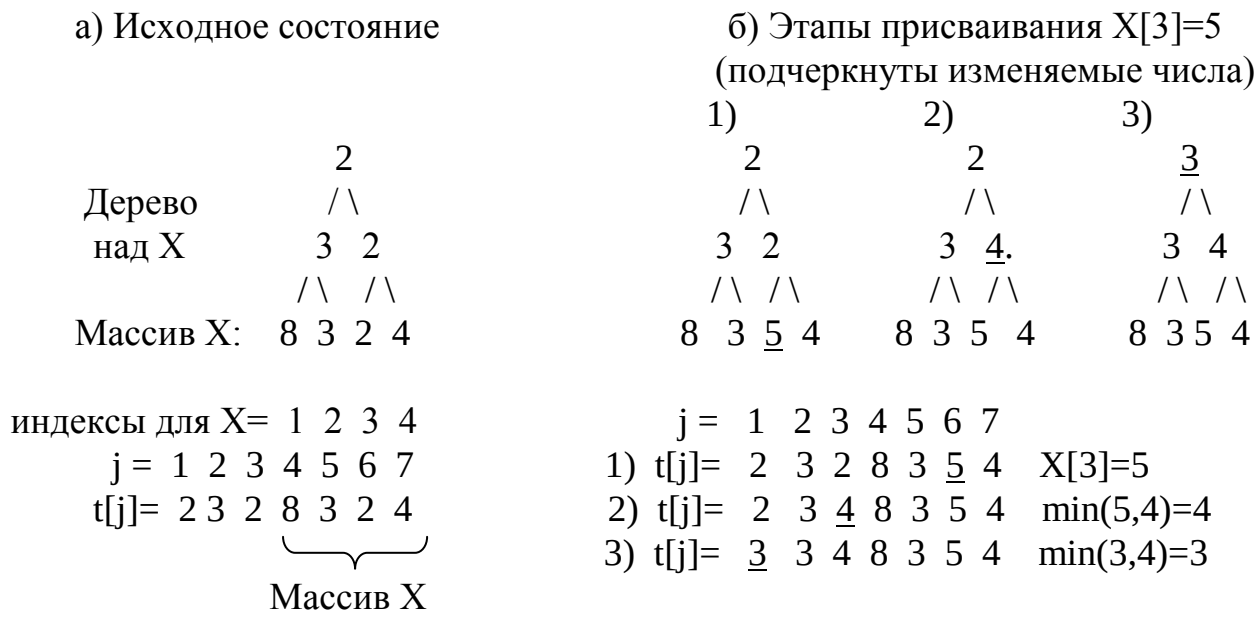


Рис. 3.14. Массив X с быстрым поиском минимального элемента

Бинарное дерево (вместе с листьями - массивом X) хранится в векторе t с помощью адресной арифметики: сыновья вершины с индексом j имеют индексы 2*j и 2*j+1, а ее отец имеет индекс j/2. Корень дерева t[1] равен меньшему из своих сыновей t[2] и t[3] (и минимальному значению массива), t[2] равен меньшему из своих сыновей t[4] и t[5] и т.д. Вектор t содержит 2*n-1 элементов: t[1]..t[n-1] - внутренние вершины дерева (не листья), а t[n]..t[2*n-1] играют роль элементов массива X (X[i] отображается в t[i+n-1]). Реализация операций - в алг. 3.1.

Алгоритм 3.1. Операции над массивом с быстрым поиском минимального элемента

```
/*                  Подготовка к работе                          */
t[n]..t[2*n-1] = X[1]..X[n];   /* Запись исходного массива  */
/*              Надстройка дерева над массивом                   */
for (j=2*n-1; j>1; j=j-2)
{
   if (t[j] < t[j-1])  t[j/2]=t[j];
   else  t[j/2]=t[j-1];
}
/*              Присвоить i-му элементу заданное число           */
t[i+n-1] = заданное число;
for (j=i+n-1; j>1; j=j/2)
{  /* отцу t[j] присвоить min (t[j], брат t[j])                  */
   if (j%2)  k=j-1; else k=j+1;       /* индекс брата t[j]  */
   if (k<2*n && t[k]<t[j])  t[j/2]=t[k];
   else  t[j/2]=t[j];
}

/*           Получить номер минимального элемента                */
j=1;
while (j < n)
```

```
   if (t[j]==t[2*j])  j=2*j; else j=2*j+1;
результат = j-n+1;
```

Примечание.В языках, где это возможно, в том числе С, умножение и деление на 2 быстрее выполнять сдвигом, т.е. заменить:

2*n на n<<1; j / 2 на j >> 1.

Проверка if (j % 2) быстрее выполнится в виде if (j & 1).

Пример 3.23. ***Очередь с приоритетами***. Требуется реализовать структуру данных для запоминания и удаления элементов с информацией о времени некоторых событий. Элементы поступают в любом порядке, а удаляются в хронологическом порядке по времени событий. Время события считать целым числом.

Решение. Реализуемую структуру данных можно рассматривать как очередь с приоритетами. В приоритетной очереди указывается приоритет поступающего элемента. При удалении из очереди выбирается элемент с наивысшим приоритетом (или один из таких элементов), а не тот, который поступил первым.В данной задаче приоритетом служит время события, высшим считается меньшее значение приоритета.

Если хранить приоритетную очередь в виде списка из n элементов, упорядоченных по возрастанию приоритета, то время включения элемента пропорционально n, время исключения из начала списка фиксировано: O(1) (не зависит от n) и суммарное время обработки элемента O(n). Приоритетную очередь можно реализовать так, чтобы добавление и удаление элемента для больших очередей происходило быстрее, за время O(log n). Для этого используется идея пирамидальной сортировки (ее также называют "сортировка деревом"), описанной в первой части учебника [67].

Под пирамидой будем понимать (полностью сбалансированное) бинарное дерево, каждая вершина которого (элемент очереди) содержит значение (приоритет элемента), причем значение вершины не больше значений всех ее потомков (рис. 3.15).

Сбалансированное дерево удобно представлять в векторе q[1]..q[k] с помощью адресной арифметики: сыновьями вершины q[j] являются вершины q[2*j] и q[2*j+1], отцом вершины q[j] будет q[j/2]. (При целочисленном делении дробная часть результата отбрасывается.) Корнем дерева является q[1].

Примечание. В отличие от элементов массива примера 3.21 элементы очереди можно переставлять. Поэтому здесь внутренние вершины дерева (не листья) вместе с листьями содержат основную информацию, а не дублируют значения листьев, т.е. память используется экономно.

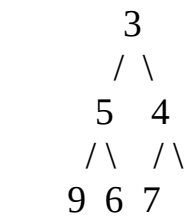
Будем размещать элементы очереди в векторе q[1], ..., q[k], (где k - количество элементов очереди), поддерживая свойство ,пирамиды: q[j] не больше своих сыновей q[2*j] и q[2*j+1], если таковые существуют, и, следовательно, всякий элемент не больше своих потомков. На рис. 3.15

очередь содержит 6 элементов, в векторе q показаны только значения их приоритетов.

а) Элементы очереди (могут поступать в любом порядке):

3, 4, 5, 6, 7, 9

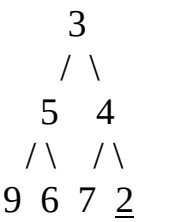
б) Исходное состояние



j = 1 2 3 4 5 6 7
q[j]= 3 5 4 9 6 7 -

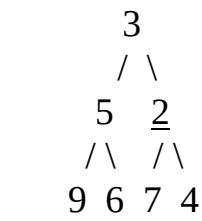
в) Добавление в очередь числа 2 (подчеркнуты изменяемые числа)

Запись 2



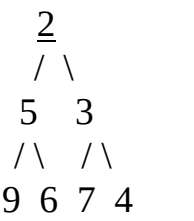
j = 1 2 3 4 5 6 7
q[j] = 3 5 4 9 6 7 2 Запись 2

Обмен 4 ↔ 2



q[j] = 3 5 2 9 6 7 4 Обмен 4 ↔ 2

Обмен 3 ↔ 2



q[j] = 2 5 3 9 6 7 4 Обмен 3 ↔ 2

Рис. 3.15. Очередь с приоритетами в виде пирамиды

Новый элемент добавляется в конец занятой части вектора, затем восстанавливается пирамидальность дерева (алг. 3.2).

Алгоритм 3.2. Включение нового элемента в очередь с приоритетами (Очередь <== новый элемент)

```
k++;  q[k]= новый элемент;  j=k;
while (q[j] - не корень и q[j] меньше своего отца)
{  /* инвариант: в дереве любой предок не больше потомка,
                если этот потомок - не q[j]                  */
   Обменять q[j] с его отцом;
}
```

Очередь образуют элементы, стоящие в вершинах дерева. За каждым элементом стоят двое, а перед каждым (кроме первого) - один. Встав в конец очереди, приоритетный элемент пробирается к началу, вытесняя впереди стоящих, пока не встретит более приоритетного элемента (рис. 3.15).

Алгоритм 3.3. Удаление элемента из очереди с приоритетами и присваивание его величине x (Очередь ==> x)

```
x=q[1];  q[1]=q[k];  k--;  t=1;
/*       Восстановление пирамиды из дерева с корнем t,      */
/*       поддеревья которого являются пирамидами            */
while (минимальное число не в t, а в одном из ее сыновей)
{  /* Все вершины не больше потомков, кроме, быть может t   */
   if (минимальное число в правом сыне)
   {  Обмен t с ее правым сыном; t = правый сын; }
   else                /* минимальное число - в левом сыне  */
```

```
   {  Обмен t с ее левым сыном; t = левый сын;    }
}
```

При удалении элемента q[1] на его место в корень дерева переписывается q[k] - последний элемент занятой части вектора. Пирамидальность дерева нарушается в его корне, но оба поддерева корня сохраняют пирамидальность (алг. 3.3).

Для восстановления пирамидальности всего дерева его корень сравнивается с меньшим из сыновей. Если корень меньше, дерево уже является пирамидой; а если корень больше, он меняется местами с меньшим сыном и затем таким же образом восстанавливается нарушенная пирамидальность поддерева с замененным корнем.

Процесс оканчивается когда восстанавливаемое поддерево состоит из одной вершины-листа. Заметим, что q[j] является листом тогда и только тогда, когда 2*j > k.

3.4. Множество

Множество – это совокупность элементов с обычными теоретико-множественными операциями: проверка вхождения элемента (принадлежности) данному множеству, пересечения, объединения, вычитания множеств и т. п.

Рассмотрим два основных подхода к представлению множеств: в виде таблицы и в виде характеристического вектора.

1. Таблица содержит описание каждого элемента, принадлежащего множеству. Операции над элементом и множеством - проверка принадлежности элемента, его присоединение или удаление из множества - соответствуют поиску, включению и исключению в таблице. Объединение, пересечение и вычитание множеств удобно реализовать слиянием упорядоченных таблиц.

2. Для хранения подмножеств некоторого базового множества удобен *характеристический вектор* - битовый вектор длиной N, i-й разряд которого соответствует i-му элементу базового множества и равен 1, если этот элемент принадлежит хранимому (под)множеству, и 0, если не принадлежит. Характеристический вектор позволяет представлять 2^N возможных подмножеств базового множества с мощностью (количеством элементов) N.

Операции над множествами реализуются в этом случае очень быстрыми поразрядными (битовыми) операциями: объединение - дизъюнкцией (ИЛИ, в языке Си обозначается одной вертикальной чертой |), пересечение - конъюнкцией (И, в Си &), вычитание из базового множества (дополнение) - инверсией (отрицанием, в Си ~). Эти операции во многих случаях выполняются одной машинной командой. Таким же образом в множество легко включать и исключать элементы, представляя их как одноэлементные множества.

Характеристический вектор эффективен для "плотных" подмножеств базового множества при наличии простого соотношения между элементом и его номером i. Этот метод обычно применяется для реализации множеств, имеющихся в языке Pascal. Их мощность не превышает 256.

Пример 3.24. ***Дни недели.*** Для хранения подмножества дней недели достаточно одного байта: Пусть, например, бит с номером 0 представляет понедельник, бит 1 - вторник, ..., бит 6 - воскресенье. Бит 7 остается неиспользованным и будет равен нулю. На рис. 3.16 показаны несколько примеров представления множеств дней недели битовым вектором и операций над ними. Эти множества реализуются следующим фрагментом С-программы.

```
typedef char dni;              /* Тип - множество дней недели: 0..6      */
dni     rab_dni = 0x1F,        /* Рабочие дни - биты:          0001 1111 */
                               /* Номера дней (битов)          7654 3210 */
        vyh_dni = 0x60,        /* Выходные дни - биты:         0110 0000 */
        zan, dezh, svob;       /* Дни занятий, дежурств, свободные       */
```

```
    . . .
svob = ~(zan | dezh) & 0x7F;        /* Свободные дни                    */
if (svob & vih_dni)      . . .      /* Есть свободные выходные дни?     */
```

	Дни недели:	-всп чсвп
	Номера битов:	7654 3210
Дни занятий: {пн,ср,пт}	zan	0001 0101
Дни дежурств: {ср,сб}	dezh	0010 0100
Занятые дни	zan | dezh	0011 0101
Незанятые дни	~(zan | dezh)	1100 1010
Маска использованных битов (в 16-й системе)	0x7F	0111 1111
Свободные дни:	svob = ~(zan | dezh)&0x7F	0100 1010
Выходные дни:	vyh_dni = 0x60	0110 0000
Есть свободные выходные дни	(svob & vyh_dni) ≠ 0	0100 0000 ≠ 0

Рис. 3.16. Множества дней недели

Пример 3.25. ***Простые числа < 1000000.*** Задача: проверить, является ли заданное целое X < 1000000 простым числом (делится только на 1 и на себя).

Решение 1. Деление на все нечетные числа, не превышающие корень квадратный из X (до 500 делений - медленно!)

Решение 2. Хранить таблицу простых чисел меньших 1000000 в виде упорядоченного по возрастанию вектора из 78498 чисел по 4 байта, всего 313992 байта. Поиск требует до $\log_2 78498 = 17$ делений пополам. Много памяти!

Решение 3. Хранить таблицу простых чисел меньших 1000000 как подмножество нечетных чисел в виде характеристического вектора из 500000 бит = 62500 байт (в 5 раз меньше, чем в решении 2!).

Используем массив tpr из 31250 двухбайтовых чисел без знака типа unsigned. Тогда j-й бит i-го элемента массива tpr соответствует нечетному числу X = 2*(16*i+j)+3.

Обозначим: y = 16*i+j. Тогда при данном X найдем:

y = (X-3)/2, i = y/16, j = y%16.

Отсюда получается подпрограмма проверки простоты числа:

```
/*          Таблица: j-й бит tpr[i] = Число 2*(16*i+j)+3 - простое          */

unsigned tpr[31250] =
  { 0x65B7,      /*  0  1  1  0  0  1  0  1  1  0  1  1 0 1 1 1          */
                 /* 33 31 29 27 25 23 21 19 17 15 13 11  9  7  5  3          */
```

```
        ...
    };
/*        Таблица степеней 2:              st2[i] = 2**i     (I = 0..15)                */
unsigned st2[16] = {1, 2, 4, 8, 16, 32, 64, 128, 256, 512,1024,
                    2048, 4096, 8192, 16384, 32768};
                    . . .
/* Функция: X - простое число (X < 1000000)                                              */
int  prostoe_chislo (long X) {
        if (X == 1)  return  0;              /* 1 не считают простым числом */
        if (X%2)      { X = (X - 3) / 2;    return  tpr[X / 16] & st2[X % 16]; }
        return 0;
}
```

Быстрее и проще заменить в операторе return деление и степень сдвигом:

return tpr[X >> 4] & (1 << (X % 16)).

Тогда не нужна таблица степеней 2.

Еще быстрее вычислять остаток от деления на 16 выделением четырех младших бит: X%16 заменить на X & 0xF.

Решение 4. Компромисс между решениями 1 и 3: для проверки простоты выполнять деление X на простые числа, не превышающие квадратного корня из X, который меньше 1000. Для этого хранить таблицу из 168 двухбайтовых простых чисел, меньших 1000. Требует до 168 делений.

3.6. Задачи

3.1. Дан вектор, содержащий очередь и значения указателей начала очереди UN = 7 и конца очереди UK = 3.

индекс	1	2	3	4	5	6	7	8
значение	10	8	11	17	13	5	12	0

Перечислить значения элементов очереди в порядке их поступления.

3.2. Как изменятся значения вектора и указателей после включения числа 15 в очередь из упражнения 3.1?

3.3. Отображающий вектор очереди содержит числа: 6, 9, 1, 3, 7, 2, 8. Перечислить элементы очереди в порядке поступления. Указатель начала - индекс первого элемента, указатель конца - индекс позиции, следующей за последним элементом.

а) Указатель начала равен 3, указатель конца равен 1.

б) Указатель начала равен 2, указатель конца равен 5.

Определить значения элементов вектора и указателей после каждой операции: включения в очередь числа 4, последующего исключения элемента из очереди, последующего включения в очередь числа 5.

3.4. Очередь содержит в порядке поступления слова: бак, вид, гол, дом, пол, фен и представлена вектором из 8 элементов. Определить значения элементов отображающего вектора и указателей для двух вариантов: указатель начала меньше указателя конца и больше указателя конца.

3.5. Дана очередь, содержащая один элемент - букву 'Z', и представленная в виде вектора (списка). Составить трассировочную таблицу выполнения последовательности операций: удалить; включить 'X'; включить 'Y'; удалить; включить 'A'; включить 'B'; удалить; включить 'X'; включить 'Y'.

3.6. Разработать алгоритм вывода в порядке возрастания первых n натуральных чисел, разложение которых на простые множители не содержит других чисел, кроме 2, 3, 5.

3.7. Составить программу для задачи 3.6.

3.8. Отображающий вектор стека содержит числа: 3, 5, 8, 1, 7, 3, 2. Указатель стека равен 5. Перечислить элементы стека в порядке поступления, если указателем стека служит индекс

а) последней занятой позиции;

б) первой свободной позиции.

Определить значения элементов вектора и указателя

1) после выталкивания элемента из стека;

2) после вталкивания в стек числа 6.

3.9. Стек содержит, в порядке поступления, слова: тир, дом, сад, пол, шаг, вал и размещен с конца вектора из 10 элементов. Определить значения элементов отображающего вектора и указателя.

3.10. Дан стек с элементами 'K' и 'Z', представленный в виде

а) вектора;

б) списка.

Составить трассировочную таблицу выполнения следующей последовательности операций: включить 'R'; включить 'S'; удалить; включить 'T'; включить 'U'; удалить; удалить; включить 'S'; включить 'C'.

3.11. Составить описание данных и

а) фрагменты программы;

б) подпрограммы

выполнения типовых операций для стека в виде списка, содержащего до 100 слов длиной до 8 символов. Список организован с помощью

1) ссылочных данных;

2) параллельных массивов.

3.12. Дана последовательность открывающих и закрывающих скобок. Составить программу определения пар номеров соответствующих друг другу скобок.

Пример. Вход: (( ) (( )))
1 2 3 4 5 6 7 8
Выход: 2-3 5-6 4-7 1-8

3.13. Дек. Деком называют структуру, сочетающую очередь и стек: добавлять и удалять элементы можно с обоих концов. Реализовать дек размером до 100 вещественных чисел в виде вектора

- составить описание данных и фрагменты программы выполнения операций.

3.14. Оценить необходимый объем памяти для структур данных из задач: 3.4, 3.5, 3.8, 3.9, 3.11. Недостающие детали представления уточнить самостоятельно.

3.15. Отображающий вектор очереди содержит числа: 7, 3, 2, 5, 8, 4. Перечислить элементы очереди в порядке поступления. Указатель начала - индекс первого элемента, указатель конца - индекс позиции непосредственно за последним элементом.

а) Указатель начала равен 4, указатель конца равен 2. б) Указатель начала равен 1, указатель конца равен 4.

Определить значения элементов вектора и указателей после каждой из следующих операций: исключения элемента из очереди, последующего включения в очередь числа 6, последующего включения в очередь числа 9.

3.16 Отображающий вектор стека содержит числа: 7, 3, 2, 5, 8, 4. Указатель стека равен 4. Перечислить элементы стека в порядке поступления, если указателем стека служит индекс последней занятой (первой свободной) позиции.

Определить значения элементов вектора и указателя

а) после выталкивания элемента из стека;

б) после вталкивания в стек числа 6.

3.17. Очередь (стек) содержит в порядке поступления слова: акт, бор, кол, рот и представлена вектором из 6 элементов. Определить значения элементов отображающего вектора и указателей.

3.18. Дан пустой стек (очередь), представленный в виде вектора (списка). Составить трассировочную таблицу выполнения следующих операций: включить 'A'; включить 'B'; включить 'C'; удалить; включить 'D'; удалить; удалить; включить 'A'; включить 'E'.

3.19. Описать данные для хранения стека (очереди) в виде списка. Составить программы операций включения и исключения элемента. Значением элемента является вещественное число.

3.20. Составить описание данных для представления строки длиной до 100 символов в виде

а) вектора фиксированной длины.

б) вектора переменной длины со счетчиком.

в) вектора переменной длины с признаком конца.

г) списка с односимвольными элементами.

д) двустороннего списка с односимвольными элементами.

е) списка с элементами фиксированной длины 8.

ж) списка с элементами переменной длины.

Необходимые детали представления уточнить самостоятельно.

3.21. Составить фрагменты программы (подпрограммы) ввода строки в виде последовательности символов, заканчивающейся переводом строки, и ее преобразования в представления а-г из задачи 3.20.

3.22. Составить функцию определения меры близости слов X и Y (расстояния между ними) по формуле:

$$D(X,Y) = (100-100*K/(DX-1))*(100-100*K/(DY-1)),$$

где DX, DY - длины слов, K - число пар соседних букв, входящих в оба слова одновременно.

Пример:

D("Саша", "Даша") = (100-100*2/(4-1))*(100-100*2/(4-1)) =

= 10000/9 = 1111.11

3.23. Решение задачи из примера 3.20 оформить в виде подпрограмм.

3.24. В каждом байте массива упакованы по 4 двухбитовых числа. Составить фрагмент программы (подпрограмму)

а) распаковки числа: получения значения числа по его номеру;

б) упаковки числа: присвоения данного значения числу с указанным номером.

3.25. Подсчитать объем памяти для доступа с помощью векторов Айлифа к элементам массива с Pascal-описанием

```
var S:  array [1..2, -1..0, 0..2] of char;
```

3.26. Для массива с Pascal-описанием

1) S: array [1..2, -1..0, 0..2] of char;
2) var S: array [-4..1, 1..3, 0..4] of char;

а) определить количество элементов и объем памяти;

б) подсчитать объем памяти для дескриптора;

в) составить функцию зависимости адреса элемента от его индексов для доступа через дескриптор;

г) подсчитать объем памяти для векторов Айлифа;

д) изобразить структуру хранения с векторами Айлифа.

3.27. Для массивов с элементами

1) U[0,1,1], U[0,1,2], U[0,2,0], U[0,2,1], U[1,0,2], U[1,0,3], U[1,1,1], U[2,2,-1], U[2,2,0], U[2,3,1], U[2,3,2]
2) X[1,0,1], X[1,0,2], X[1,1,0], X[1,1,1], X[1,1,2], X[2,1,0], X[2,1,1], X[2,2,-1], X[2,2,0], X[2,3,2], X[2,3,3]

а) подсчитать объем памяти для векторов Айлифа;

б) изобразить структуру хранения с векторами Айлифа.

3.28. Составить фрагмент программы инициализации пустой очереди с приоритетами из примера 3.23.

3.29. Составить трассировочную таблицу поступления элементов в очередь с приоритетами из примера 3.23 в следующем порядке: 8, 5, 3, 7, 2 и последующего удаления двух элементов.

3.30. Для очереди с приоритетами из примера 3.23 составить фрагменты программ выполнения операций.

3.31. Оформить в виде подпрограмм выполнение операций для очереди с приоритетами из примера 3.23.

3.32. Для примера 3.24 написать фрагмент программы
а) определения множества рабочих дней с дежурствами;
б) проверки, есть ли совпадения дней дежурства с занятиями.

3.33. Составить фрагменты программы (подпрограммы) решений 1, 2 и 4 примера 3.25.

3.34. Даны множества S1 и S2 в виде характеристических векторов: S1 = 10011001, S2 = 01110101. Определить количество элементов в S1 и в S2.

Получить характеристический вектор множества S, равного объединению множеств S1 и S2, из которого вычтено пересечение этих множеств. Записать эти операции в виде фрагмента программы.

3.35. Составить фрагмент программы подсчета количества элементов множества, представленного характеристическим вектором длиной 16 бит.

3.36. Задача W. Взвешивание (Д.Г. Хохлов, турнир ICL-2005 [81]) Из монет с номерами от 1 до N одна фальшивая (легче или тяжелее настоящей), остальные настоящие одного веса. Произведено K взвешиваний. Для каждого взвешивания даны номера равного числа монет на левой и правой чаше весов и результаты в виде знака =, < или >, например 5 4 2 > 1 9 7 (монеты 5, 4, 2 в сумме тяжелее монет 1, 9, 7). Вывести номер фальшивой монеты (или "?", если это невозможно). Сообщить также знаком неравенства, легче "<" фальшивая монета или тяжелее ">" настоящей (или вывести "?", если это невозможно).

Входной файл INPUT.TXT

N K

<номера монет левой чаши> <знак> <номера монет правой чаши>

. . .

Числа отделены друг от друга и знака пробелами и/или символами новой строки ($0 < N < 100000$, $0 \le K \le 100$).

Выходной файл OUTPUT.TXT
номер знак (через пробел, или "?" вместо номера и/или знака)

Примеры INPUT.TXT	OUTPUT.TXT
5 1 1 = 5	? ?
5 1 1 5 = 2 4	3 ?
5 2 3 < 5 5 = 1	3 <