МОСКОВСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ ПУТЕЙ СООБЩЕНИЯ (МИИТ)

Кафедра «САПР транспортных конструкций и сооружений»

В. Ю. СМИРНОВ
О. В. СМИРНОВА

СТРУКТУРЫ ДАННЫХ

Методические указания к лабораторным работам

Москва – 2005

МОСКОВСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ ПУТЕЙ СООБЩЕНИЯ (МИИТ)

Кафедра «САПР транспортных конструкций и сооружений»

В. Ю. СМИРНОВ
О. В. СМИРНОВА

Утверждено
редакционно-издательским
советом университета

СТРУКТУРЫ ДАННЫХ

Методические указания к лабораторным работам
для студентов специальности «САПР»

Москва – 2005

УДК 681.3
С 50

Смирнов В.Ю., Смирнова О.В. Структуры данных: Методические указания. – М.: МИИТ, 2005. – 28 с.

Настоящие методические указания предназначены для студентов, изучающих дисциплину «Организация ЭВМ и систем». В методических указаниях рассмотрены основные структуры данных, использующиеся для хранения и обработки информации, и приведены варианты заданий для лабораторных работ.

Методические указания предназначены для студентов специальности «САПР».

© Московский государственный университет
путей сообщения (МИИТ), 2005

Содержание

Введение 4

Стек 5

Очередь 7

Связанные списки 9

Представление списка с помощью массива 11

Деревья 13

Представление двоичного дерева с помощью одномерного массива 16

Поиск и включение записей в дереве 16

Удаление элемента из дерева 19

Задания 24

Литература 27

Введение

Все структуры данных условно делятся на два класса – структуры последовательного и произвольного доступа. В первых структурах доступ возможен к одному или нескольким элементам. Обычно это элементы в начале или в конце структуры. Так, в случае очереди доступны лишь элементы в её начале и конце, в случае стека – элемент в его вершине. В структурах произвольного (прямого) доступа можно производить действия с любым элементом структуры, при этом не требуется переборов всех или части элементов.

Реализации структур данных можно также разделить на два класса – непрерывные и ссылочные. В непрерывных реализациях элементы структуры данных располагаются последовательно друг за другом в некотором непрерывном отрезке массива, причём порядок их расположения в массиве соответствует их порядку в реализуемой структуре. В ссылочных реализациях элементы структуры данных хранятся в совершенно произвольном порядке. При этом вместе с каждым элементом хранятся ссылки на один или несколько «соседних» элементов. В качестве ссылок могут выступать либо индексы ячеек массива, либо адреса памяти.

Ссылочные реализации имеют недостатки:

1. для хранения ссылок требуется дополнительная память;
2. для доступа к некоторому элементу необходимо пройти последовательно по цепочке других элементов.

Но все недостатки ссылочных реализаций компенсируются чрезвычайно важным достоинством: в них можно добавлять и удалять элементы в середине структуры данных, не перемещая остальные элементы.

В методических указаниях рассматриваются принципы работы с простейшими структурами данных: стеками, очередями, списками и бинарными деревьями. Приведена их реализация на базе массивов.

Стек

Стек – самая популярная и важная структура данных. Из стека можно удалить только тот элемент, который был добавлен в него последним. Стек работает по принципу «последним пришёл – первым ушёл» (last-in, first-out – сокращённо LIFO). Примером стека может служить стопка бумаг на столе или стопка тарелок. Тарелки берутся в порядке, обратном их добавлению. Доступна только тарелка на вершине стопки.

Операция добавления элемента в стек часто обозначается **Push** – «запихивать», а операция удаления элемента **Pop** – «вынимать».

На рисунке 1 показано, как можно реализовать стек ёмкостью не более N элементов на базе массива S. Индексы ячеек массива изменяются от 0 до N-1. Индекс элемента наверху стека обозначим через переменную *t*, которая называется указателем стека.

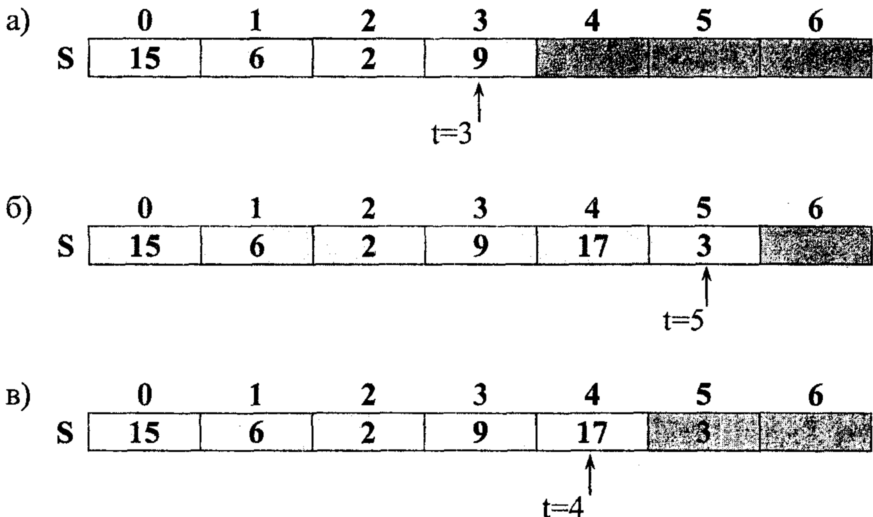


Рисунок 1. Реализация стека на базе массива S.

Светло-серые клетки заняты элементами стека.
(а) – Стек содержит 4 элемента, верхний элемент – число 9.
(б) – Тот же стек после выполнения операций **Push** для элементов 17 и 3.
(в) – Стек после того, как операция **Pop** вернула значение 3 (последний добавленный в стек элемент). Хотя число 3 присутствует в массиве S, в стеке его уже нет; на вершине стека число 17.

Когда стек **пуст**, его указатель имеет значение *t*=-1. Если *t*=N-1, то при попытке добавить элемент происходит **переполнение**.

Операции со стеком (проверка пустоты, добавление элемента, удаление элемента) записываются на языке Си так:

```
int Stack_Empty()
{
if (t==-1 ) return 0;
        else return 1;
}

void Push(int x)
{ // проверить на переполнение, если стек не заполнен, то
t++;
S[t]=x;
}

int Pop()
{
if (Stack_Empty()==0 )
        { printf("Стек пуст!");
          exit(1); }
        else { t--;
              return S[t+1];    }
}
```

Самостоятельно: написать функцию проверки стека на переполнение, в функции **Push** предусмотреть ее вызов.

Проанализируйте случай, когда начальное значение указателя на вершину стека имеет значение t=0, и внесите необходимые изменения в алгоритм.

Очередь

Очередь в программировании аналогична очереди в повседневной жизни. Она содержит элементы, как бы выстроенные друг за другом в цепочку. У очереди есть начало и конец. Элемент, добавляемый в очередь, оказывается в её конце, элемент, удаляемый из очереди, находится в её начале. Очередь работает по принципу «первым пришёл – первым ушёл» (first-in, first-out – сокращённо FIFO).

Операцию добавления элемента в очередь обозначим **Inq**, а операцию удаления элемента **Deq**.

На рисунке 2 показано, как можно реализовать очередь, вмещающую не более N элементов, на базе массива Q. Индексы ячеек массива изменяются от 0 до N-1. Массив свернут в кольцо: за N-1 следует 0. Кроме массива реализация очереди содержит три простые переменные: индекс начала очереди h, индекс конца очереди t, число элементов очереди N.

Светло-серые клетки заняты элементами очереди.

(а) – В очереди находятся 3 элемента (позиции 3…5).

(б) – Очередь после выполнения процедуры **Inq** для элементов 8 и 17.

(в) – Очередь после выполнения процедуры **Deq**, которая возвращает значение 15. Новой головой очереди стало число 6.

Первоначально $h = t = 0$. Когда очередь **пуста**, то $h = t$. Если очередь пуста, попытка удалить из неё элемент ведёт к ошибке. Если $t = h$, то очередь полностью заполнена, и попытка добавить к ней элемент вызовет **переполнение**.

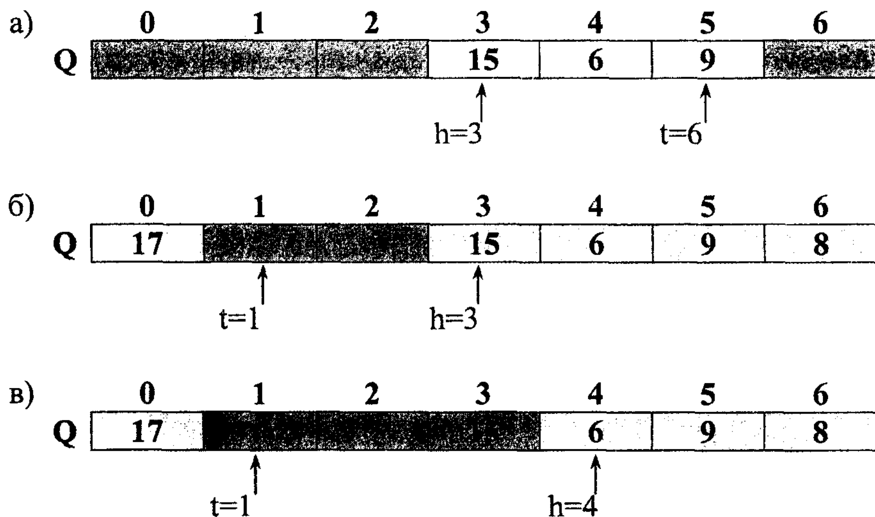


Рисунок 2. Реализация очереди на базе массива Q.

В приведенных ниже реализациях процедур **Inq** и **Deq** не учитывается возможность переполнения или попытки изъятия элемента из пустой очереди.

```
void Inq(int x)
  {
// проверить на переполнение, если очередь не заполнена, то
  Q[t]=x;
  if (t==N) t=0;
          else t++;
  }

int Deq()
  {
// проверить на пустоту, если очередь не пуста, то
  x=Q[h];
  if (h==N) h=0;
```

```
        else h++;
    return x;
    }
```

Самостоятельно: написать функцию проверки очереди на переполнение, функцию проверки очереди на пустоту. Предусмотреть их вызов в соответствующих функциях (**Inq** и **Deq**).

Связанные списки

В **связанном списке** элементы линейно упорядочены, но порядок определяется не номерами, как в массиве, а указателями, входящими в состав элементов списка. Например, если каждый стоящий в очереди запомнит, кто за ним стоит, после чего все в беспорядке рассядутся на лавочке, получится односторонне связанный список; если он запомнит ещё и впереди стоящего, будет двусторонне связанный список.

Другими словами, как показано на рисунке 3, элемент двусторонне связанного списка – это запись, содержащая три поля: *key* (ключ) и два указателя – *next* (следующий) и *prev* (предыдущий). Помимо этого, элементы списка могут содержать дополнительные данные. Если *x* – элемент списка, то *next* указывает на следующий элемент списка, а *prev* – на предшествующий.

Если *prev* = NULL, то у элемента x нет предшествующего: это голова списка. Если *next* = NULL, то *x* – последний элемент списка, или его хвост.

Прежде, чем двигаться по указателям, надо знать хотя бы один элемент списка; предполагаем, что для списка L известен указатель *head* на его голову. Голова списка хранит указатели на первый и последний элементы списка, но не хранит никакого элемента.

Если *head* = NULL, то список пуст.

В различных ситуациях используются разные виды списков. В *односторонне связанном* списке отсутствуют поля *prev*.

В *упорядоченном* списке элементы расположены в порядке возрастания ключей, так что у головы списка ключ наименьший, а у хвоста списка – наибольший, в отличие от неупорядоченного списка. В *кольцевом* списке поле *prev* головы списка указывает на хвост списка, а поле *next* хвоста списка указывает на голову списка.

Операцию поиска в списке обозначим **LSearch**, она находит в списке с помощью простого линейного поиска первый элемент, имеющий ключ k. Точнее говоря, она возвращает указатель на этот элемент или NULL, если элемента с таким ключом в списке нет.

Операция **Lins** добавляет элемент x к списку L, помещая его в голову списка. Операция **Ldel** удаляет элемент x из списка, направляя указатели «в обход» этого элемента. При этом в качестве аргумента ей передается указатель на x. Если задан ключ элемента, то перед удалением надо найти его указатель с помощью операции **LSearch**.

Выполнение этих операций показано на рисунке 3.

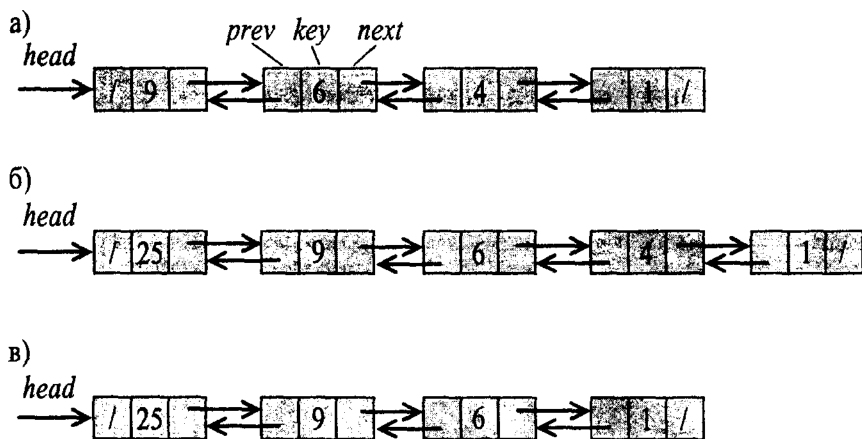


Рисунок 3. Двусторонне связанный список L

Двусторонне связанный список L содержит числа 1, 4, 9, 6; каждый элемент списка – это запись с полями для ключа и указателей на предыдущий и последующий элементы (эти указатели изображены стрелками). В поле *next* у хвоста списка и в поле *prev* у головы списка находится указатель NULL (косая черта на рисунке); *head* – указывает на голову списка.

В результате выполнения операции **Lins**, где *key* = 25, в списке появился новый элемент с ключом 25; он стал новой головой списка, а его поле *next* указывает на бывшую голову – элемент с ключом 9.

Вслед за этим была выполнена операция **Ldel**, где *x* – указатель на элемент с ключом 4.

Для того чтобы не выходить за концы списка, вводят фиктивный элемент *nil*, который имеет поля *next* и *prev*. Указатель на него играет роль значения NULL.

При этом список замыкается в кольцо: в поля *next* и *prev* запишем указатели на голову и хвост списка соответственно, а в поля *prev* у головы списка и *next* у хвоста списка занесем указатели на *nil*. При этом *nil*[*next*] – указатель на голову списка, так что *head* становится лишним.

Не следует применять фиктивные элементы без особой необходимости. Если используется много коротких списков, то применение фиктивных элементов приводит к дополнительной трате памяти.

Представление списка с помощью массива

Для хранения списка можно использовать одномерный массив, размещая в нем различные поля одной записи рядом. Указателем на элемент считают адрес ячейки, в которой хранится ключ. Адреса полей получаются сдвигом на определенные константы.

Рассмотрим массив A. Каждая запись будет занимать в нем непрерывный участок A[j...k]. Указатель на запись – это

индекс j; каждому полю записи соответствует число из промежутка 0…k – j – сдвиг. Например, при представлении списка на рисунке 3 можно задать, что полям *prev*, *key* и *next* соответствуют сдвиги 0, 1 и 2. Тогда значение *prev* – это A[i+2] (рисунок 4).

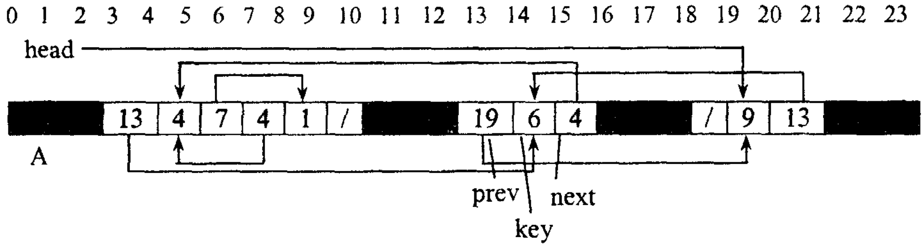


Рисунок 4. Список, реализованный на базе одномерного массива

Светло-серые записи входят в список, стрелками изображено действие указателей.

Алгоритм выполнения операции LSearch при реализации на одномерном массиве

```
int LSearch(k)
{
x=head;
while (x!=NULL && L[x]!=k)
   x=L[x+1];
return x;
}
```

Алгоритм выполнения операции Lins при реализации на одномерном массиве

```
Lins(k)
{
// найти свободное место в массиве и записать адрес в x
```

```
L[x]=k;
L[x+1]=head;
if (head!=NULL) L[head-1]=x;
head=x;
L[x-1]=NULL;
}
```

Алгоритм выполнения операции Ldel при реализации на одномерном массиве

```
Ldel(k)
{
// найти удаляемый элемент в массиве с помощью LSearch

 if (x-1==NULL)  // удаляемый элемент – голова
    {
     L[L[x+1]-1]=NULL;
     L[x+1]=NULL;
     head=L[x+1];}
 else
    {
     L[L[x-1]+1]=L[x+1];
     L[L[x+1]-1]=L[x-1];}
 }
```

Чаще, однако, элементы списка не располагают в каком-либо массиве, а просто размещают каждый по отдельности в *динамической* памяти. В качестве ссылок в этом случае используют адреса элементов в оперативной памяти.

Деревья

Фигура, которая состоит из вершин и рёбер, соединяющих вершины, называется **графом**. Примером графа является схема линий метро. Рёбра могут иметь направления, т.е. изображаться стрелочками, такие графы называются *ориентированными*.

Дерево – связный граф без циклов. Кроме того, в нем выделена одна вершина, которая называется корнем дерева. Остальные вершины упорядочиваются по длине пути, который ведет от корня дерева к данной вершине.

Двоичное дерево можно изобразить схемой, представленной на рисунке 5.

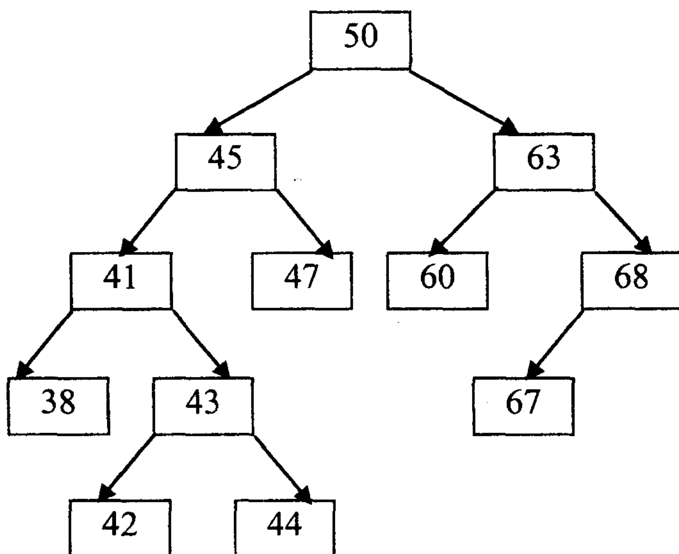


Рисунок 5. Двоичное дерево

Имеется набор вершин, соединяемых стрелками, из каждой вершины выходит не более двух стрелок (ветвей), направленных вниз влево или вниз вправо. Верхняя вершина, в которую не входит никакая стрелка, называется корнем. В остальные вершины входит по одной стрелке. Каждая вершина дерева представляет собой запись с несколькими полями. Одно из этих полей содержит ключ, как и в случае со списками. Остальные поля хранят указатели на соседние вершины.

Как показано на рисунке 6, при представлении двоичного дерева T мы используем поля ***p***, ***left*** и ***right***, в которых хранятся

указатели на *родителя* (предыдущий элемент), *левого ребенка* (меньший элемент), *правого ребенка* (больший элемент) соответственно. В поле ***k*** хранится ключ.

Если ***p*** =NULL, то вершина ***x*** – *корень*. Если ***left*** =NULL или ***right***=NULL, то у вершины ***x*** нет левого или правого ребенка; если оба этих указателя нулевые, то вершина ***x*** называется *терминальной*, или *листом*.

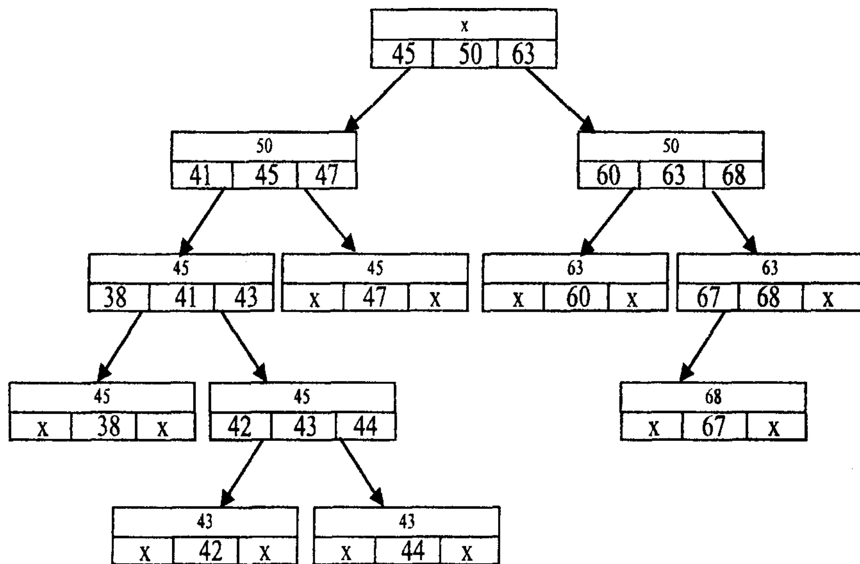


Рисунок 6. Представление двоичного дерева.

Поиск записи с заданным ключом ***K*** начинаем с корня дерева. Если ***K*** не совпадает с ключом ***k*** рассматриваемой записи, то в случае $K<k$ идем по левой ветви, в случае $K>k$ – по правой и так далее.

Например, для дерева, изображенного на рисунке 6, нужно найти запись с ключом ***K***=47. Мы сравниваем значение ***K*** с ключом ***k***=50 из корневого звена. Поскольку ***K***<50, то мы идем

по левой ветви и сравниваем ***K*** с ключом ***k***=45. Теперь ***K***>***k***, мы идем по правой ветви и приходим к записи с искомым ключом 47. Если на каком-то этапе поиска оказалось, что ветвь, по которой нам следовало бы двигаться, отсутствует, то искомой записи нет в дереве.

Если требуется **добавить** в дерево новую запись с ключом ***K***, то выполняется аналогичный поиск звена с этим ключом. Если такая запись не найдется, то присоединяем к этому звену ветвь с новой записью, причем новая ветвь направлена влево или вправо в зависимости от выполнения условия ***K***<***k*** или ***K***>***k***.

Представление двоичного дерева с помощью одномерного массива

Для хранения дерева будем использовать одномерный массив T. Каждое звено этого дерева состоит из четырёх элементов:

T[i] – ссылка на «родителя»;

T[i+1] – ключ записи;

T[i+2] – ссылка на левого «ребенка» (индекс звена по левой ветви);

T[i+3] – ссылка на правого «ребенка» (индекс звена по правой ветви).

Если из данной вершины не выходит левая или правая ветвь, то соответствующий указатель T[i+2] или T[i+3] равен нулю. Два начальных элемента массива T отводятся для информации:

T[0] – ссылка на звено, являющееся корнем дерева;

T[1] – ссылка на начало свободного места в массиве T.

Поиск и включение записей в дереве

Рассмотрим операцию **поиска** по ключу звена в двоичном дереве.

Эта операция по заданному ключу ***K*** находит индекс *i* в

массиве Т звена с данным ключом, а также находит индекс *p* предшествующего звена (содержащего ссылку на искомое звено). Если искомого звена не окажется в дереве, то идентификатору *i* присваивается значение 0. Если искомой записью оказывается корень дерева, то выдаются значения *i*=T[0] и *p*=0.

Работа начинается с того, что устанавливаются значения *p*=0 и *i*=T[0], соответствующие корню дерева. Затем выполняется цикл поиска по дереву нужного звена. Для каждого очередного звена с индексом *i* ключ этого звена T[i+1] сравнивается с искомым ключом *K*. В случае их равенства рассматриваемая запись оказывается искомой. Если же эти ключи не равны, то присваиваем *p*=*i* – делаем рассматриваемое звено кандидатом в предшественники искомого звена и ищем среди его потомков звено с ключом *K*. Если *K*<T[i+1], то ключ может оказаться у звена с левой ветви дерева. Это звено указывается ссылкой из T[i+2], и мы устанавливаем значение *i*=T[i+2]. Иначе нужно проверять правую ветвь, и тогда мы присваиваем *i*=T[i+3]. Если получилось новое значение *i*=0, то у предыдущей записи не оказалось соответствующего потомка. Это означает, что искомой записи нет в дереве. Цикл выполняется, пока не будет найдено звено с ключом T[i+1]=*K* или пока мы не достигнем конечного звена поиска, т.е. пока не получим очередное значение *i*=0.

```
int TSearch(k)
{
p=0;
i=T[0];
while (T[i+1]!=k || i!=0)
   {
    p=i;
    if (k<T[i+1] ) i=T[i+2];
          else i=T[i+3];
   }
```

```
return i;
}
```

Заметим, что дерево может оказаться пустым и тогда T[0]=0. В этом случае работа функции заканчивается сразу после начального присваивания *i*=T[0], так как условие цикла i≠0 оказывается невыполненным. Если даже в дереве не оказалось записи с ключом *K*, все равно после окончания работы функции **TSearch()** параметр *p* будет содержать индекс звена, которое должно предшествовать звену с ключом *K*, если оно появится в дереве. Этот факт используется в функции **Tins()**.

Рассмотрим теперь операцию **включения** записи в двоичное дерево. Функция Tins() обращается к TSearch(), если в результате поиска получится значение i≠0, то в дереве уже содержится звено с ключом *K*, и добавлять уже ничего не надо.

В противном случае нужно сформировать новое звено с ключом *K*. Это звено занесется в массив T на свободное место, указываемое значением *n*= T[1]. К указателю T[1] начала свободного места прибавляется длина 4 формируемого звена. У нового звена нет потомков, поэтому на места ссылок заносятся нули T[i+2]=T[i+3]=0. В элемент T[n+1] заносится ключ *K*. В результате выполнения операции поиска функцией TSearch() в переменной *p* содержится индекс звена, которое предшествует (или должно предшествовать) в дереве звену с ключом *K*. Если *p*=0, то новое звено будет корнем дерева, и в указатель корня T[0] заносим значение индекса этого звена *n*. Иначе нужно выяснить, по левой или по правой ветви звено с ключом *K* является потомком звена с индексом *p*, ключ которого находится в элементе T[p+1]. Для этого сравниваем значения *K* и T[p+1] и в зависимости от результата сравнения заносим значение указателя *n* на место левой ссылки T[p+2] или правой ссылки T[p+3].

```
Tins(k)
{
// найти свободное место в массиве с помощью LSearch
if (i!=0) return 0;
   else {
          n=T[1];
          T[1]=T[1]+4;
          T[n+2]=T[n+3]=0;
          T[n+1]=k;
        }
if (p==0) S[0]=n;
   else {
          if (k<S[p+1]) S[p+2]=n;
               else S[p+3]=n;
        }
}
```

Удаление элемента из дерева

Наметим этапы алгоритма удаления элемента из дерева:

1) поиск соответствующего звена;
2) его исключение.

На этапе исключения возможны три варианта, в зависимости от того, сколько потомков имеется у исключаемого элемента (ни одного, один или два).

Вариант 1. Удаляемый элемент не имеет потомков. Тогда удаление сводится к уничтожению ссылки на этот элемент (рисунок 7).

Вариант 2. Удаляемый элемент имеет одного потомка. Тогда изменение ссылок можно изобразить схемой, показанной на рисунке 8.

Вариант 3. Удаляемый элемент имеет двух потомков. Тогда нужно найти элемент с таким ключом, чтобы было удобно подставить его на место исключаемого, и заменить в структуре

дерева удаляемый элемент этим найденным элементом. Такой элемент существует всегда. Чтобы найти его, нужно идти по дереву от исключаемого элемента один раз налево и потом все время направо (можно и наоборот), и как только окажется, что идти некуда, то этот элемент (из которого нет нужной ветви) годится для подстановки.

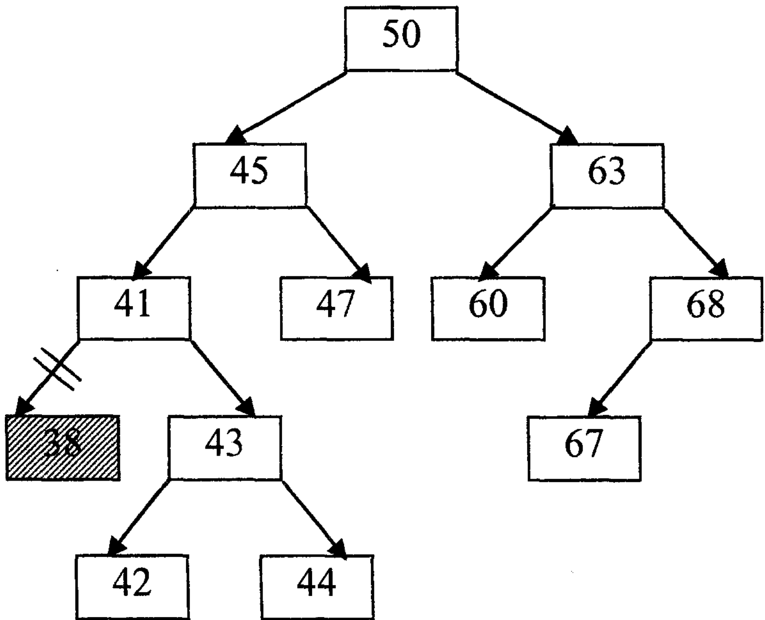


Рисунок 7. Удаление из двоичного дерева элемента, не имеющего потомков

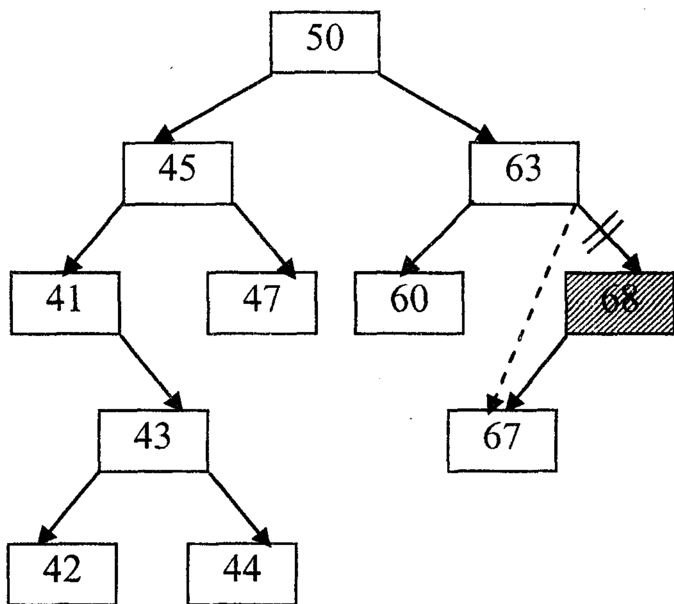


Рисунок 8. Удаление из двоичного дерева элемента, имеющего одного потомка

Таким образом, когда требуется удалить элемент, у которого два потомка, находят и подставляют на его место элемент, у которого не более одного потомка. При этом нужно исключить этот подставляемый элемент из того места, где он находился раньше. Такое исключение сводится к уже рассмотренному варианту 1 (если у подставляемого элемента нет потомков) или варианту 2 (если есть один потомок).

Например, если требуется исключить запись с ключом 46 из дерева, изображенного на рисунке 9, то вместо исключаемой записи в ту же вершину дерева можно подставить запись с ключом 47 или 45. При любой из этих замен получится правильное дерево, в котором для любой вершины все ее прямые

и косвенные потомки по левой ветви имеют меньшие ключи, а по правой ветви – большие ключи, чем ключ этой вершины.

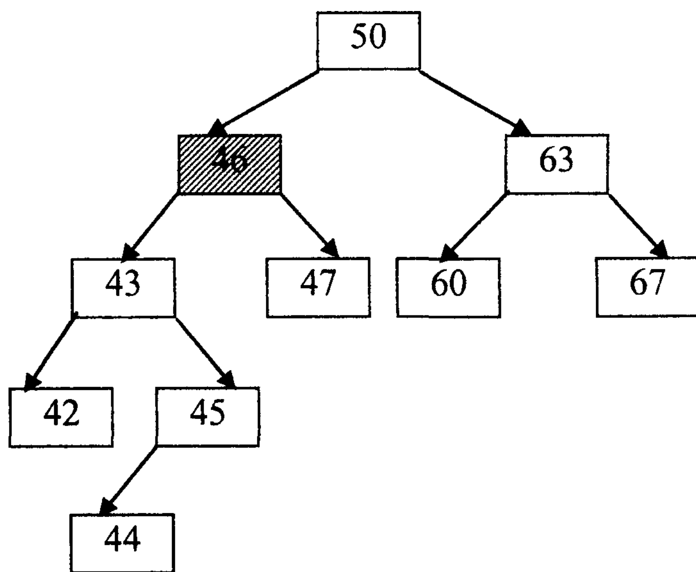
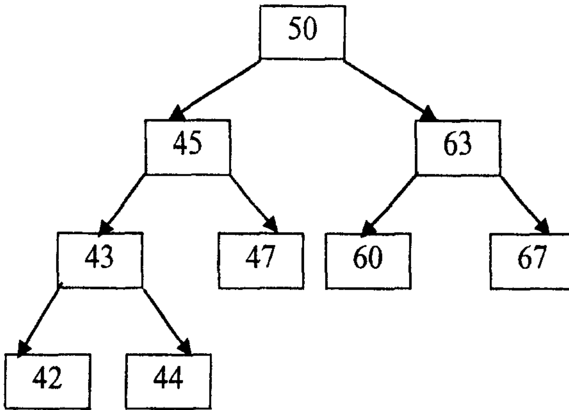


Рисунок 9. Удаление из двоичного дерева элемента, имеющего двух потомков

Если подставить запись с ключом 45, то в результате преобразования получится дерево, показанное на рисунке 10, а. Бывшие потомки исключенной записи (с ключами 43 и 47) становятся потомками подставленной записи. В результате исключения из дерева записи с ключом 46 бывший ее косвенный потомок (с ключом 44) становится прямым потомком записи с ключом 43.

Если вместо удаляемой записи подставить запись с ключом 47, то в результате преобразования получится дерево, показанное на рисунке 10, б.

а)



б)

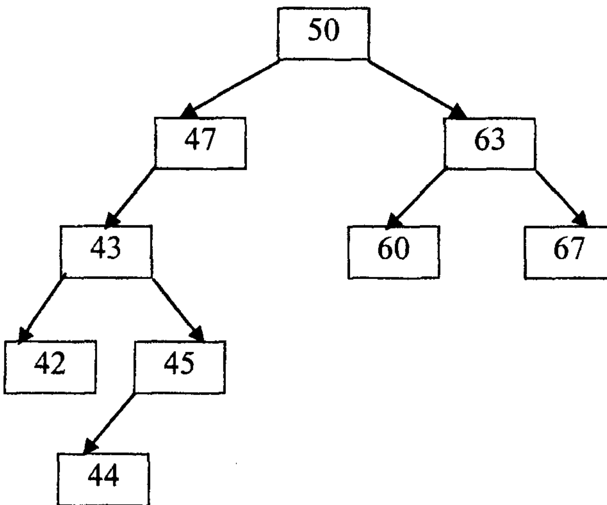


Рисунок 10. Результат удаления из двоичного дерева элемента, имеющего двух потомков

Описанная выше структура двоичного дерева позволяет существенно сократить время при поиске элементов.

Задания:

1. Реализовать стек на базе массива целых чисел из 6 элементов. Процедуры **Push** и **Pop** оформить в виде функций. Предусмотреть проверку на случай пустоты и переполнения.

2. Реализовать стек на базе массива символов из 8 элементов. Процедуры **Push** и **Pop** оформить в виде функций. Предусмотреть проверку на случай пустоты и переполнения.

3. Реализовать 2 стека на базе одного массива суммарной длины не больше 12 элементов. Процедуры **Push** и **Pop** оформить в виде функций. Предусмотреть проверку на случай пустоты и переполнения.

4. Реализовать очередь на базе массива целых чисел из 8 элементов. Процедуры **Inq** и **Deq** оформить в виде функций. Предусмотреть проверку на случай пустоты и переполнения.

5. Реализовать очередь на базе массива символов из 8 элементов. Процедуры **Inq** и **Deq** оформить в виде функций. Предусмотреть проверку на случай пустоты и переполнения.

6. Реализовать 2 очереди на базе одного массива суммарной длины не больше 16 элементов. Процедуры **Inq** и **Deq** оформить в виде функций. Предусмотреть проверку на случай пустоты и переполнения.

7. Реализовать однонаправленный список на базе массива целых чисел из 30 элементов. Процедуры **поиска**, **добавления** и **удаления** оформить в виде функций. Предусмотреть проверку на случай пустоты и переполнения.

8. Реализовать однонаправленный кольцевой список на базе массива из 36 элементов (без фиктивных элементов). Процедуры **поиска**, **добавления** и **удаления** оформить в виде функций. Предусмотреть проверку на случай пустоты и переполнения.

9. Реализовать двунаправленный список на базе массива целых чисел из 24 элементов. Процедуры **поиска**, **добавления** и **удаления** оформить в виде функций.

10. Реализовать двунаправленный упорядоченный список на базе массива целых чисел из 30 элементов. Процедуры **поиска**, **добавления** и **удаления** оформить в виде функций.

11. Реализовать кольцевой двунаправленный упорядоченный список на базе массива целых чисел из 24 элементов (без фиктивных элементов). Процедуры **поиска**, **добавления** и **удаления** оформить в виде функций.

12. Реализовать кольцевой двунаправленный список на базе массива целых чисел из 30 элементов (без фиктивных элементов). Процедуры **поиска**, **добавления** и **удаления** оформить в виде функций. Предусмотреть проверку на случай пустоты и переполнения.

13. Создать и вывести на экран дерево (можно в виде таблицы), элементы которого вводятся с клавиатуры и имеют целый тип.

14. Реализовать дерево на базе одномерного массива вещественных чисел. Процедуры **поиска**, **добавления** и **удаления** оформить в виде функций.

15. Реализовать дерево на базе одномерного массива целых чисел. Элементы вводить с клавиатуры. Процедуры **поиска, добавления** и **удаления** оформить в виде функций.

16. Создать и вывести на экран дерево (в виде таблицы), элементы которого считываются из файла и имеют целый тип. Реализовать поиск и удаление в этом дереве.

Литература

1. Кормен Т., Лейзерсон Ч., Ривест Р. Алгоритмы: построение и анализ. М.: МЦНМО, 1999. – 960 с.
2. Любимский Э.З., Мартынюк В.В., Трифонов Н.П. Программирование. – М.: Наука, 1980. – 608 с.
3. Семакин И.Г., Хеннер Е.К. Информатика. – М.: Бином, 2002. – 144 с.
4. Аванта+. Том 22. Информатика / под ред. Е.А. Хлебалина. – М.:Аванта+, 2003. – 624 с.

Учебно-методическое издание

Владимир Юрьевич Смирнов
Ольга Владимировна Смирнова

СТРУКТУРЫ ДАННЫХ
Методические указания к лабораторным работам

Подписано к печати - 09.12.05. Формат 60x90 1/16 Тираж 100.
Усл. – печ. л. 1,75
Изд. № - 55-05 заказ № 695.

Типография МИИТа, 127994, Москва, ул. Образцова, 15.