Министерство образования Республики Беларусь
Учреждение образования
«Белорусский государственный университет
информатики и радиоэлектроники»

Кафедра программного обеспечения информационных технологий

Л. В. Серебряная, И. М. Марина

СТРУКТУРЫ И АЛГОРИТМЫ ОБРАБОТКИ ДАННЫХ

*Рекомендовано УМО по образованию
в области информатики и радиоэлектроники
для специальности 1-40 01 01 «Программное обеспечение
информационных технологий»
в качестве учебно-методического пособия*

Минск БГУИР 2013

УДК 004.421(076)
ББК 32.973.26-018.2я73
С32

Р е ц е н з е н т ы:

кафедра веб-технологий и компьютерного моделирования
Белорусского государственного университета
(протокол №3 от 11.10.2012 г.);

заведующий кафедрой информатики учреждения образования
«Минский государственный высший радиотехнический колледж»,
кандидат технических наук, доцент Ю. А. Скудняков

Серебряная, Л. В.

С32 Структуры и алгоритмы обработки данных : учеб.-метод. пособие / Л. В. Серебряная, И. М. Марина. – Минск : БГУИР, 2013. – 51 с. : ил.
ISBN 978-985-488-941-2.

В пособии рассмотрены структуры данных и алгоритмы их обработки, которые являются основой современного программирования. Предложены семь лабораторных работ, выполнение которых поможет в выборе оптимальных способов решения задач, появляющихся при создании программного обеспечения различного назначения.

УДК 004.421(076)
ББК 32.973.26-018.2я73

ISBN 978-985-488-941-2

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

Создание качественного программного обеспечения во многом зависит от системного и научного подхода к его построению, особенно это очевидно на примере больших программ со сложными данными. Поэтому методы программирования включают в себя все варианты структурирования данных. Программы представляют собой конкретные формулировки абстрактных алгоритмов, основанные на определенных представлениях и структурах данных. Выяснилось, что решения о структурировании данных нельзя принимать без знания алгоритмов, применяемых к этим данным, и, наоборот, выбор алгоритмов существенным образом зависит от структуры данных. Поэтому строение программ и структуры данных неразрывно связаны между собой.

В пособии изложены структуры данных и алгоритмы, которые являются основой современного программирования. Для описания и реализации алгоритмов были использованы абстрактные типы данных, являющиеся удобным инструментом при разработке программ независимо от применяемого языка программирования.

Для представления алгоритмов и структур данных был выбран язык Delphi (Pascal), что объясняется его следующими особенностями:

– компактностью, делающей язык достаточно несложным для изучения;

– возможностью отражать фундаментальные концепции алгоритмов в легко воспринимаемой форме;

– способностью четко реализовывать идеи структурного программирования и осуществлять переход к объектно-ориентированному программированию.

В пособии рассматриваются динамические структуры данных и применяемые к ним алгоритмы. Работа состоит из семи частей, посвященных основным линейным и нелинейным структурам данных.

В разделах с первого по четвертый рассматриваются такие структуры данных, как списки, очереди и стеки, их разновидности и выполняемые операции. Предложен способ построения словарей с реализацией быстрого доступа к данным на основе их открытого хеширования. Рассмотрены три формы записи арифметических выражений, а также их преобразования на базе стека.

В разделах с пятого по седьмой рассмотрены структуры данных в виде деревьев и ориентированных графов. Выполнены три основных обхода бинарного дерева, введено понятие бинарного дерева поиска и показаны основные операции с данными на нем. Определено прошитое бинарное дерево, методы его прошивки и особенности выполнения операций с данными. Введена терминология ориентированных графов и способы их представления. Приведены алгоритмы решения базовых задач на ориентированных графах: поиск кратчайшего пути, поиск всех путей, центр графа.

В конце каждого раздела предложено задание для лабораторной работы, позволяющей студентам закрепить теоретические знания и получить практические навыки по изучаемой теме.

1. УКАЗАТЕЛИ И СПИСКИ

1.1. Виды памяти, указатели и работа с ними в языке Delphi (Pascal)

В любой вычислительной системе важным ресурсом является память. Поэтому управление ею – одна из важных задач программиста. Различают два способа распределения памяти. Статический – во время трансляции программы, что эффективно, поскольку в ходе его выполнения на управление памятью не расходуются ресурсы системы. Альтернативный подход – динамическое управление памятью, осуществляемое во время выполнения программы.

Если необходимый для программы объем памяти известен заранее, то можно использовать статические переменные, которые компилятор способен обработать, не выполняя программу, только на основании ее статического текста. Более рациональный подход связан с использованием динамической памяти. Это оперативная память, предоставляемая программе в ходе ее выполнения, за вычетом сегмента данных, стека и тела программы.

При экономном подходе к памяти заранее не резервируется ее максимальный объем для размещения данных, а предварительно определяется их тип, и каждый раз по мере необходимости создаются новые данные. Переменные, которые создаются и уничтожаются в процессе выполнения программы, называются *динамическими*.

Кроме экономии памяти для ряда задач заранее невозможно определить требуемый объем памяти, тогда динамическое распределение – единственно возможный подход. Динамическая память широко используется при работе с графическими и звуковыми средствами компьютера.

Для организации динамической памяти используется тип данных, называемый *указателем* или *ссылочным* типом данных. Значением указателя является адрес области памяти, содержащей переменную заранее определенного типа. В этом случае указатели называются *типизированными*. Для указателей область памяти выделяется статически, а для переменных, на которые они ссылаются, – динамически. Адреса задаются совокупностью двух 16-разрядных слов, которые называются *сегментом* и *смещением*. Сегмент – участок памяти, имеющий длину 64 Кбайт и начинающийся с адреса, кратного 16. Смещение указывает, сколько байт от начала сегмента необходимо пропустить, чтобы обратиться к нужному адресу. В результате, абсолютный адрес образуется следующим образом: *сегмент*16+смещение.*

При решении прикладных задач с использованием языков высокого уровня наиболее частые случаи, когда могут понадобиться указатели, следующие:

1. При работе с динамическими структурами данных. Память под них выделяется в ходе выполнения программы, стандартные процедуры/функции выделения памяти возвращают адрес выделенной области памяти, т. е. указатель

на нее. К содержимому динамически выделенной области памяти можно обращаться только через такой указатель.

2. При необходимости можно представить данные, принадлежащие определенной области памяти, как данные разных логических структур. В этом случае в программе вводится несколько указателей, содержащих адрес одной и той же области памяти, но имеющих разные типы. При обращении к этой области памяти по определенному указателю ее содержимое обрабатывается как данные того или иного типа.

Все без исключения модели современных процессоров аппаратно выполняют динамическую трансляцию адресов и совместно с операционными системами обеспечивают работу программ в виртуальной памяти. Программа разрабатывается и выполняется в виртуальной памяти, адреса в которой линейно изменяются от 0 до некоторого максимального значения. Виртуальный адрес представляет собой число – номер ячейки в виртуальном адресном пространстве. Преобразование виртуального адреса в реальный выполняется аппаратно при каждом обращении по виртуальному адресу. Это преобразование происходит совершенно прозрачно для программиста, поэтому структуру виртуального адреса можно считать и его же физической структурой.

В программе на языке высокого уровня указатели могут быть типизированными и нетипизированными. При объявлении типизированного указателя определяется и тип объекта в памяти, им адресуемого. Для объявления переменных ссылочного типа используется символ « ^ », после которого указывается тип динамической (базовой) переменной.

Type <имя_типа>=^ <базовый тип>;
Var <имя_переменной>: <имя_типа>; или
<имя_переменной>: ^ <базовый тип>;

Например:

```
Type ss = ^ Integer;
Var  x, yt: ss; {*Указатели на переменные целого типа *}
 z : ^Real; {*Указатель на переменную вещественного типа *}
```

Здесь переменные *x* и *y* представляют собой адреса областей памяти, в которых хранятся целые числа, а *z* – адрес области памяти, в которой хранится вещественное число. Хотя физическая структура адреса не зависит от типа и значения данных, хранящихся по этому адресу, компилятор считает указатели *x, y* и *z,* имеющие разный тип, и оператор *x:= z;* будет расценен компилятором как ошибочный. Таким образом, когда речь идет об указателях типизированных, правильнее говорить не о едином типе данных «указатель», а о целом семействе типов: «указатель на целое», «указатель на символ» и т. д. Могут быть указатели и на более сложные, интегрированные структуры данных, а также указатели на указатели.

Нетипизированный указатель (тип *pointer*) служит для представления адреса, по которому содержатся данные неизвестного типа.

Зарезервированное слово *Nil* обозначает константу ссылочного типа, которая ни на что не указывает.

Выделение оперативной памяти для динамической переменной базового типа осуществляется с помощью процедуры *New(x)*, где *x* определен как соответствующий указатель.

Обращение к динамическим переменным выполняется по правилу: <имя переменной> ^.

Например, *x*^:=15. Здесь в область памяти (два байта), адрес которой является значением указателя *x*, записывается число 15.

Процедура *Dispose(x)* освобождает память, занятую динамической переменной. При этом значение указателя *x* становится неопределенным.

Основными операциями, в которых участвуют указатели, являются *присваивание, получение адреса, выборка.*

Присваивание – это двухместная операция, оба операнда которой – указатели. Как и для других типов, операция присваивания копирует значение одного указателя в другой, в результате чего оба указателя будут содержать один и тот же адрес памяти. Типизированные указатели должны ссылаться на объекты одного и того же типа.

Операция получения адреса – одноместная, ее операнд может иметь любой тип, результатом является типизированный (в соответствии с типом операнда) указатель, содержащий адрес объекта-операнда.

Операция выборки – одноместная, где операндом является типизированный указатель, результат – данные, выбранные из памяти по адресу, заданному операндом. Тип результата определяется типом указателя-операнда.

Перечисленных операций, как правило, достаточно для решения задач прикладного программирования. К указателю можно прибавить целое число или вычесть из него целое число. Поскольку память имеет линейную структуру, прибавление к адресу числа даст адрес области памяти, смещенной на это число байт (или других единиц измерения) относительно исходного адреса. Результат операций «указатель + целое», «указатель – целое» имеет тип «указатель».

Можно вычесть один указатель из другого (оба указателя-операнда при этом должны иметь одинаковый тип). Результат такого вычитания будет иметь тип целого числа со знаком. Его значение показывает, на сколько байт (или других единиц измерения) один адрес отстоит от другого в памяти.

Необходимо отметить, что сложение указателей не имеет смысла. Поскольку программа разрабатывается в относительных адресах и при разных своих выполнениях может размещаться в разных областях памяти, сумма двух адресов в программе будет давать разные результаты при разных выполнениях. Смещение же объектов внутри программы относительно друг друга не зависит от адреса загрузки программы, поэтому результат операции вычитания указателей будет постоянным, и такая операция является допустимой.

Операции адресной арифметики выполняются только над типизированными указателями. Единицей измерения в адресной арифметике является размер объекта, который указателем адресуется. Так если переменная *x* определена как указатель на целое число, то выражение *x*+1 даст адрес, больший не на 1, а на количество байт в целом числе. Вычитание указателей также дает в результате не количество байт, а количество объектов данного типа, помещающихся в памяти между двумя адресами. Это справедливо как для указателей на простые типы, так и для указателей на сложные объекты, размеры которых составляют десятки, сотни и более байт.

Ниже приведено несколько схем, иллюстрирующих работу с указателями. Графическое представление указателей приведено на рис. 1.1.

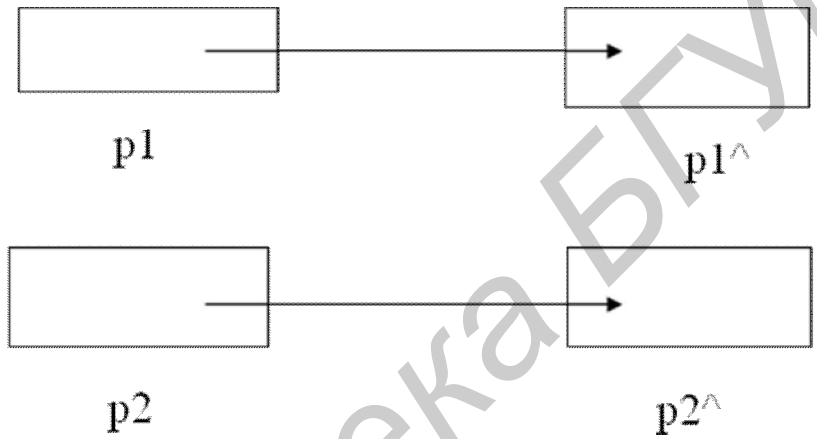


Рис. 1.1. Указатели *p*1, *p*2

В результате выполнения следующих операций присваивания (рис. 1.2) *p*1^:=2; *p*2^:=4; в выделенные участки памяти будут записаны значения 2 и 4.

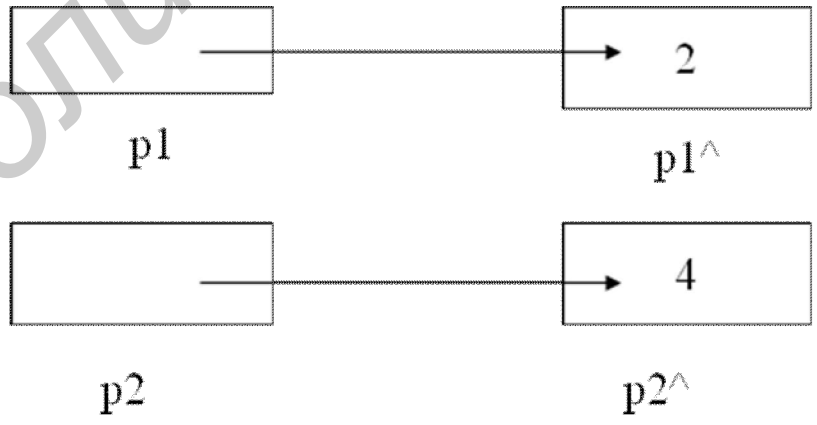


Рис. 1.2. Реализация операций присваивания

В результате выполнения оператора присваивания *p*1^:=*p*2^; в участок памяти, на который ссылается указатель *p*1, будет записано значение 4 (рис. 1.3).

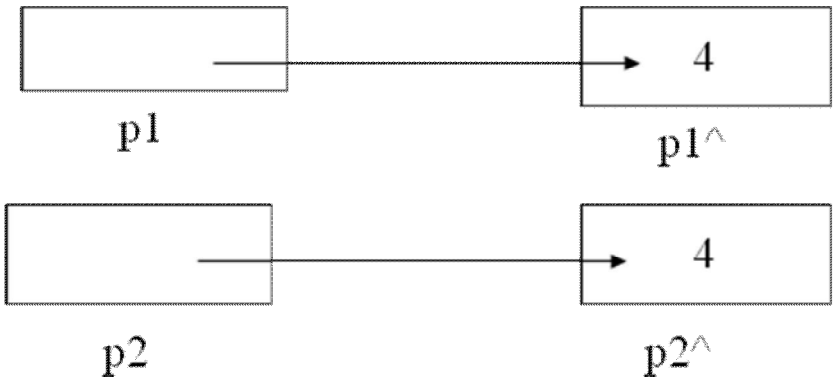


Рис. 1.3. Перезапись значения по указанному адресу памяти

После выполнения оператора присваивания *p2:=p1*; оба указателя будут содержать адрес первого участка памяти (рис. 1.4).

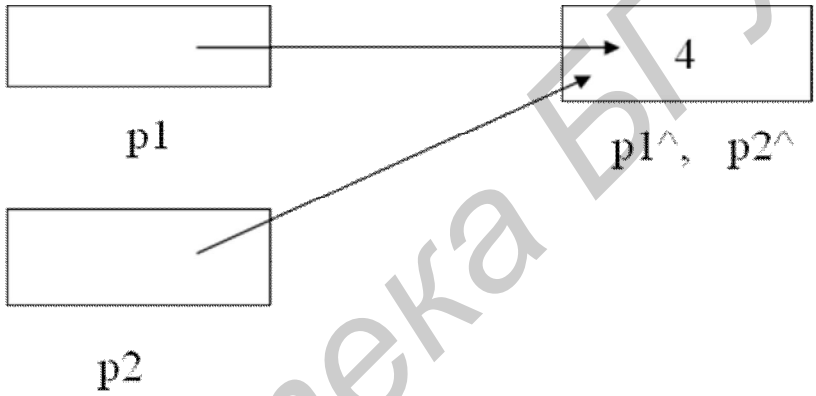


Рис. 1.4. Ссылка двух указателей на один адрес памяти

Использование динамических величин предоставляет программисту ряд дополнительных возможностей:

– увеличить объем обрабатываемых данных;

– освобождать память от ненужной информации до окончания работы программы и использовать ее для хранения актуальной информации;

– создавать структуры данных переменного размера.

При этом следует понимать, что работа с динамическими данными замедляет выполнение программы, поскольку доступ к информации происходит в два этапа: сначала ищется указатель, а затем по нему – данное.

1.2. Абстрактный тип данных «список»

В математике список представляет собой последовательность элементов определенного типа: $a_1, a_2, \ldots a_n$, где $n \geq 0$ и все a_i имеют один тип. Количество элементов, равное n, определяет длину списка, если $n \geq 1$, то a_1 – первый эле-

мент, а a_n – последний элемент списка. В случае $n = 0$ имеет место пустой список.

Списки являются чрезвычайно гибкой структурой: легко изменять их размер; их элементы доступны для вставки или удаления в любой позиции. Списки можно объединять или разбивать на части.

В случае реализации списка на основе массива его элементы располагаются в смежных ячейках памяти. Такое представление удобно для просмотра содержимого списка и добавления элементов в его конец. Однако вставка элемента в середину списка требует перемещения всех последующих элементов на одну позицию к концу массива, чтобы освободить место новому элементу. Удаление элемента из середины массива также требует перемещения части его элементов с целью заполнения освободившейся ячейки.

Рассмотрим реализацию однонаправленного списка с использованием указателей, связывающих последовательные элементы списка. Однонаправленный список представляет собой динамическую структуру данных, число элементов которой может изменяться по мере того, как данные помещаются в список или удаляются из него. Эта реализация не требует непрерывной области памяти для хранения списка и, следовательно, освобождена от перемещения элементов списка при вставке или удалении данных. Однако ценой за это удобство становится использование дополнительной памяти для хранения указателей.

При такой организации список состоит из ячеек (звеньев), каждая из которых содержит его элемент и указатель на следующую ячейку списка. Если список состоит из элементов $a_1, a_2, ...a_n$, то для i=1, 2,..., n–1 ячейка, содержащая элемент a_i, также хранит указатель на звено, содержащее элемент a_{i+1}. В ячейке, содержащей элемент a_n, значение указателя равно *nil* (нуль). В списке присутствует ячейка *header* (заголовок), которая указывает на звено, содержащее элемент a_1. Ячейка *header* не содержит элементов списка. В случае пустого списка указатель заголовка равен *nil*, т. е. он не указывает ни на какую ячейку. На рис. 1.5 показан однонаправленный список.

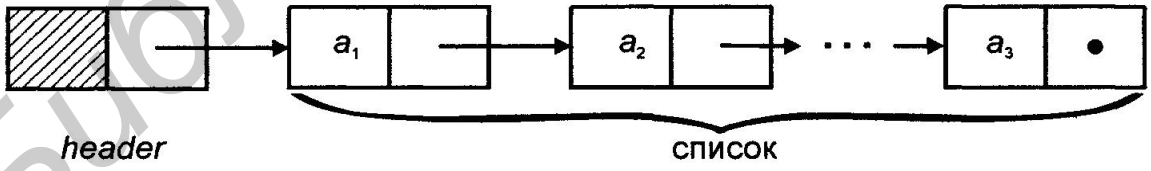


Рис. 1.5. Однонаправленный список

Для однонаправленных списков удобно использовать следующее определение позиций элементов. Здесь для i=2, 3, ..., n позиция i определяется как указатель на ячейку, содержащую элемент a_i. Позиция 1 – это указатель в ячейке заголовка, а позиция n – указатель в последней ячейке списка L. Структуру звена списка можно определить следующим образом:

```
Type pt^=elem; {*Указатель на элемент списка.*}
     elem = Record
              data: Integer;  {*Поле данных.*}
              next: pt;  {*Указатель на следующий элемент списка.*}
              end;
var first:pt;   {*Указатель на первый элемент списка.*}
```

Основные операции, выполняемые с элементами списка: просмотр, вставка, удаление.

Ниже приведен код процедуры создания списка. Здесь строится список из пяти целых чисел.

```
Procedure make(x : pt);
Var i : integer;
    Begin
        New(x);
        For i:=1 to 5   do
         Begin
            New(x^.next);
            x:=x^.next;
            Readln(x^.data);
        End;
           x^.next:=nil;
    End;
```

Процедура просмотра и вывода на печать элементов списка:

```
Procedure print (x:pt);
Begin
      while x<>nil do
        Begin
            write (x^.data, ' ');
            x:=x^.next;
       End;
End;
```

Процедура вставки элемента в заданную позицию списка.

```
   Procedure Insert ( x : integer; p:pt);
      Var
           temp: pt;
```

```
        Begin
(1)          temp:= p^.next;
(2)          New(p^.next);
(3)          p^.next^.data:= x;
(4)         p^.next^.next:=temp;
        End;
```

Механизм управления указателями в процедуре *Insert* приведен на рис. 1.6. На рис. 1.6, а показана ситуация перед выполнением процедуры вставки. Необходимо вставить новый элемент *x* перед элементом *b*, поэтому в строке (1) задается *temp* как указатель на ячейку, содержащую элемент *b*. В строке (2) листинга создается новая ячейка, и в поле *next* ячейки, содержащей элемент *a*, ставится указатель на новую ячейку. В строке (3) поле *data* вновь созданной ячейки принимает значение *x*, а в строке (4) поле *next* этой ячейки принимает значение переменной *temp*, которая хранит указатель на ячейку, содержащую элемент *b*. На рис. 1.6, б представлен результат выполнения процедуры *Insert*, где пунктирными линиями показаны новые указатели и номерами, совпадающими с номерами строк в листинге, помечены этапы их создания.

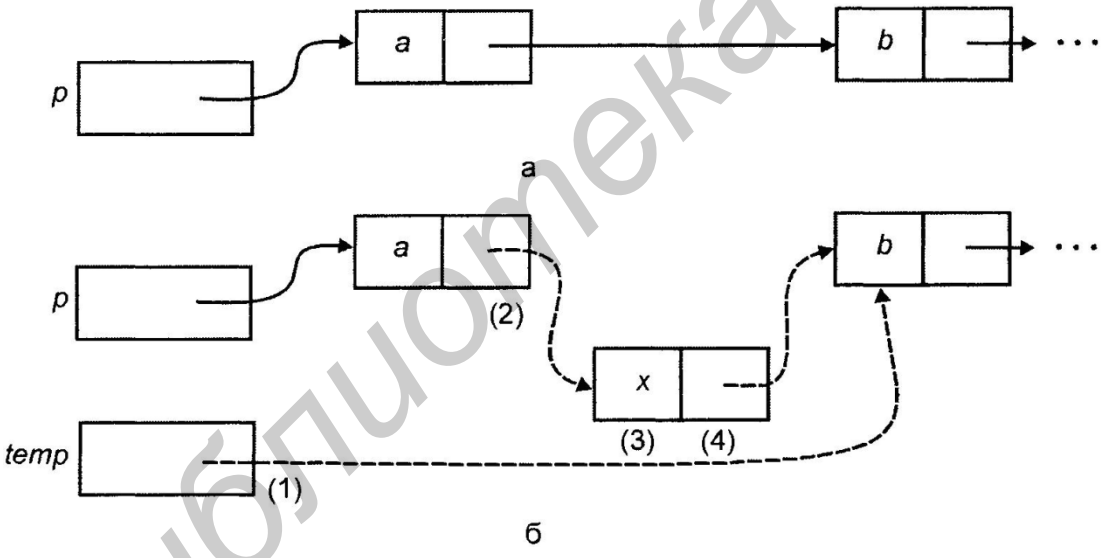


Рис. 1.6. Реализация вставки элемента в список

На рис. 1.7 показана схема манипулирования указателями в процедуре удаления элемента из списка. Старые указатели показаны сплошными линиями, а новые – пунктирной. Ниже приведен код процедуры удаления элемента.

```
Procedure Delete ( b : integer; p:pt);
      Begin
        If  p^.next^.data=b then
```

```
    p^.next:= p^.next^.next;
End;
```

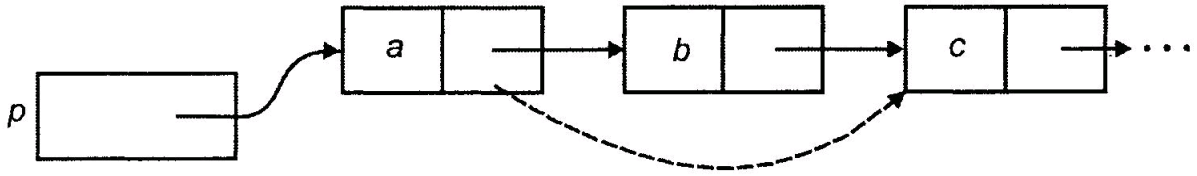


Рис. 1.7. Реализация удаления элемента из списка

1.3. Двунаправленные списки

Часто возникает необходимость организовать эффективное перемещение по списку, как в прямом, так и в обратном направлениях; или по заданному элементу нужно быстро найти предшествующий ему и последующий элементы. В таких ситуациях можно поместить в каждую ячейку указатели и на следующий, и на предыдущий элементы списка, т. е. организовать двунаправленный список. На рис. 1.8 приведен такой список.

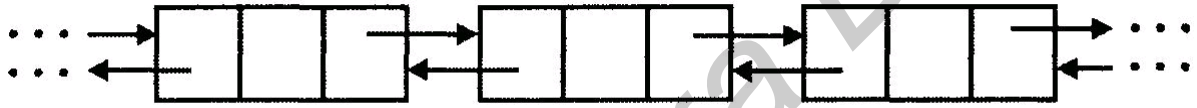


Рис. 1.8. Двунаправленный список

Важным преимуществом этого списка является то, что в нем можно использовать указатель ячейки, содержащей *i*-й элемент, для определения *i*-й позиции вместо использования указателя предшествующей ячейки. Однако при этом появляются дополнительные указатели и, следовательно, удлиняются некоторые процедуры. Ниже приведен код объявления звена двунаправленного списка.

```
Type pt=^elem; {Указатель на элемент списка}
     Elem=Record
        Data : Integer; {Поле данных}
        Next :pt; {Указатель на следующий элемент списка}
        Prev : pt; {Указатель на предыдущий элемент списка}
   End;
Var first:pt; {Указатель на первый элемент списка}
```

Рассмотрим процедуры построения списка, а также удаления элемента из него. На рис. 1.9 приведена схема удаления элемента из двунаправленного списка. Здесь предполагается, что удаляемая ячейка не является ни первой, ни последней в списке. Сплошными линиями показаны указатели до удаления, а пунктирными – после удаления элемента. Сначала с помощью указателя поля

prev определяется положение предыдущей ячейки. Затем в поле *next* этой ячейки устанавливается указатель на ячейку, предшествующую позиции *p*. Таким образом, ячейка в позиции *p* исключается из цепочек указателей и при необходимости может быть использована повторно.

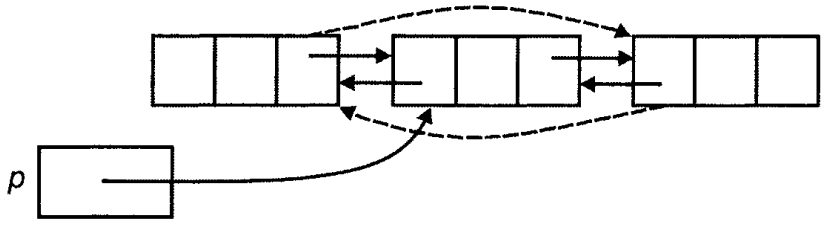


Рис. 1.9. Удаление элемента из двунаправленного списка

```
Procedure Make(var x : pt); {Построение списка из пяти элементов}
  Var  y : pt;
       i : Integer;
   Begin
      New(x);
      x^.prev:=nil;
        For i:=1 to 5 Do
        Begin
             y:=x;
            Readln(y^.data);
          If  i<>5 Then
           Begin
                New(x);
               y^.next:=x;
              x^.prev:=y;
            End;
         Else
             y^.next:=nil;
       End;
End;

Procedure Delete (p:pt);
   Begin
      If   p^.prev <> nil  then  {Удаление ячейки, не являющейся первой}
           p^.prev^.next:=p^.next;
      If   p^.next <> nil  then {Удаление ячейки, не являющейся последней}
           p^. next^. prev:=p^. prev;
 End;
```

1.4. Лабораторная работа №1

Цель работы: научиться строить однонаправленные, двунаправленные и кольцевые списки на базе указателей.

Порядок выполнения работы

1. Ознакомиться с теоретической частью лабораторной работы.
2. Выполнить практическое задание.
3. Оформить отчет по лабораторной работе.

В лабораторной работе предлагается решить несколько задач. Ниже приводятся условия каждой из них.

1.1. Многочлен $P(x) = a_n x^n + a_{n-1} x^{n-1} + ... + a_1 x + a_0$ с целыми коэффициентами можно представить в виде списка. При этом, если $a_i = 0$, то соответствующий элемент не включается в список. На рис. 1.10 показано общее представление многочлена и пример: $S(x) = -5x^6 + 3x^2 - x + 7$.

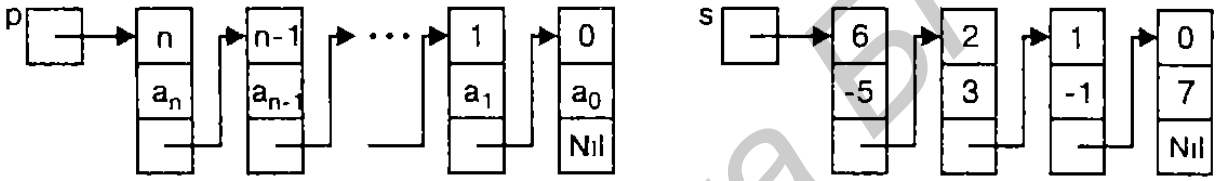


Рис. 1.10. Представление многочленов с помощью списков

Необходимо: описать тип данных, соответствующий предложенному представлению многочленов, разработать следующие функции и процедуры для работы с этими списками-многочленами:

– логическую функцию *Equality(p,q)*, проверяющую равенство многочленов *p* и *q*;

– функцию *Meaning(p,x)*, вычисляющую значение многочлена в целочисленной точке *x*;

– процедуру *Add(p,q,r)* вычисления суммы многочленов *q* и *r*, результат – многочлен *p*.

1.2. С помощью списков реализовать программу «Считалочка». *N* ребят расположены по кругу. Начав отсчет от первого, удаляют каждого *к*-ого, смыкая при этом круг. Определить порядок удаления ребят из круга.

Эту задачу необходимо исследовать для различных значений *N* от 1 до 64, составив таблицу оставшихся ребят (*t* – номер оставшегося ребенка).

1.3. Построить однонаправленный линейный список абонентов телефонной станции, упорядоченный лексикографически, содержащий ФИО и семизначный номер телефона. Составить процедуры определения:

– по номеру телефона фамилии;

– по фамилии списка номеров телефонов.

1.4. Построить двунаправленный неупорядоченный список номеров телефонов: семизначных – абонентов; трехзначных – спецслужб. Просмотреть спи-

сок справа налево и построить упорядоченный однонаправленный список, не включая в него номера телефонов спецслужб.

2. ПОСТРОЕНИЕ СЛОВАРЕЙ НА ОСНОВЕ ХЕШИРОВАНИЯ ДАННЫХ

2.1. Открытое хеширование данных

Существует множество способов повышения эффективности работы с большими объемами информации. Например, для ускорения доступа к данным в таблицах можно использовать предварительное упорядочивание таблицы в соответствии со значениями ключей. При этом могут быть использованы методы поиска в упорядоченных структурах данных, что существенно сократит время поиска данных по значению ключа. Однако при добавлении новой записи требуется переупорядочить таблицу. Потери времени на повторное упорядочивание справочника часто превышают выигрыш от сокращения времени поиска. Рассмотрим способы организации данных, лишенные указанного недостатка. Это – различные формы хеширования данных.

На рис. 2.1 показана базовая структура данных при открытом хешировании. Основная идея метода заключается в том, что множество данных (возможно, очень большое) разбивается на конечное число классов. Для B классов, пронумерованных от 0 до B–1, строится хеш-функция h такая, что для любого элемента x исходного множества функция $h(x)$ принимает целочисленное значение из интервала 0, …, B–1, соответствующее классу, которому принадлежит элемент x. Элемент x называют ключом, $h(x)$ – хеш-значением x, а классы – сегментами. Массив (таблица сегментов), проиндексированный номерами сегментов 0, 1, ..., B–1, содержит заголовки для B списков. Элемент x i-го списка – это элемент исходного множества, для которого $h(x)=i$.

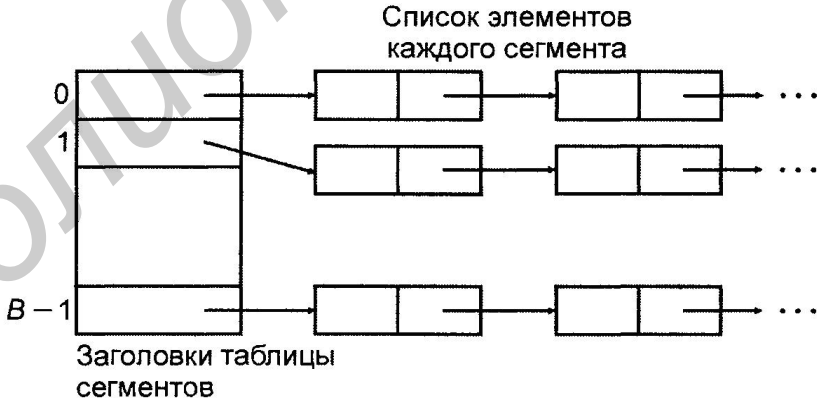


Рис. 2.1. Организация данных при открытом хешировании

Если сегменты приблизительно равны по размеру, то в этом случае списки всех сегментов должны быть наиболее короткими при данном числе сегментов. Если исходное множество состоит из N элементов, тогда средняя длина списков будет N/B элементов. Если удается оценить величину N и выбрать B как можно

ближе к этой величине, то в каждом списке будет один-два элемента. Тогда время выполнения операций с данными будет малой постоянной величиной, зависящей от *N* или от *B*. Однако не всегда ясно, как выбрать хеш-функцию *h* так, чтобы она примерно поровну распределяла элементы исходного множества по всем сегментам.

Идеальной хеш-функцией является такая, которая для любых двух неодинаковых ключей выдает неодинаковые адреса, т. е.

$$k_1 \neq k_2 \Rightarrow h(k_1) \neq h(k_2). \quad (1)$$

Однако подобрать такую функцию можно в случае, если все возможные значения ключей известны заранее. Такая организация данных носит название «совершенное хеширование». Если заранее не определено множество значений ключей, и длина таблицы ограничена, подбор совершенной функции затруднителен. Поэтому часто используют хеш-функции, которые не гарантируют выполнение условия (1).

Рассмотрим хеш-функцию, определенную на символьных строках, которая не является совершенной, однако значения *h(x)* будут «хорошими». Идея построения этой функции заключается в том, чтобы представить символы в виде целых чисел, используя для этого машинные коды символов. В языке *Delphi (Pascal)* есть встроенная функция *ord(c)*, которая возвращает целочисленный код символа *c*. Таким образом, если *x* – это ключ длиной *j* символов, тип данных ключей определен как строка символов, тогда можно использовать хеш-функцию, код которой приведен ниже. В этой функции суммируются целочисленные коды всех символов ключа, результат делится на B и берется остаток от деления, который будет целым числом из интервала от 0 до *B*–1.

Пример 1. Хеш-функция, определенная на символьных строках

```
Function h(x : string[10]; j:integer): integer;
  Var i, sum: integer;
    Begin
      sum:=0;
     For i:=1 to j do
       sum:=sum+ord(x[i]);
       h:=sum mod B; {Возвращаемое значение}
    End;
```

2.2. Лабораторная работа №2

Цель работы: научиться строить словари и справочники на базе линейных списков и открытого хеширования данных.

Порядок выполнения работы

1. Ознакомиться с теоретической частью лабораторной работы.
2. Выполнить практическое задание.

3. Оформить отчет по лабораторной работе.

Задание. На основе динамических списков необходимо реализовать словарь (справочник). Основные операции, выполняемые над данными словаря: поиск, вставка и удаление. В сочетании со списками для построения словарей удобно использовать *открытое хеширование данных*, позволяющее фиксировать время выполнения операторов над словарем, а также сделать потенциально бесконечным пространство для хранения данных. Тематическое содержание словаря может быть произвольным.

3. ПОСТРОЕНИЕ ОЧЕРЕДЕЙ И ОБРАБОТКА ДАННЫХ НА ИХ ОСНОВЕ

3.1. Абстрактный тип данных «очередь»

Очередь – это специальный тип списка. Очередью называют структуру, из которой элементы удаляются с одного ее конца, называемого началом (головой), а вставляются на противоположном конце, называемом (хвостом).

Очереди считаются списками типа *FIFO* (аббревиатура расшифровывается как *first in first out*: первым вошел – первым вышел). Две основные операции, которые определены для работы с очередью: вставка и извлечение элементов. Кроме того, используются и другие операторы:

1) *MAKENULL(Q)* очищает очередь *Q*, делая ее пустой;

2) *FRONT(Q)* возвращает первый элемент очереди *Q*;

3) *ENQUEUE(x, Q)* вставляет элемент *x* в конец очереди *Q*;

4) *DEQUEUE (Q)* удаляет первый элемент очереди *Q*;

5) *EMPTY (Q)* возвращает значение *true* (истина) только тогда, когда *Q* является пустой очередью.

Рассмотрим способ реализации очереди на базе списков, учитывая ее структурные и функциональные особенности. Например, при добавлении элемента в конец очереди вместо перемещения по списку от начала к концу можно хранить указатель на конец очереди. Указатель на начало очереди будет полезен при выполнении операторов, связанных с извлечением элементов из структуры. В качестве заголовка можно использовать динамическую переменную и поместить в нее указатель на начало очереди, что позволит удобно организовать очищение очереди.

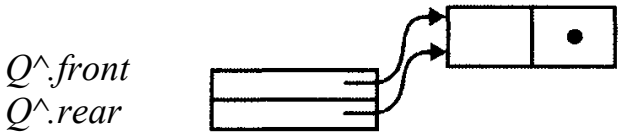
Определим список, содержащий указатели на начало и конец очереди. Первая ячейка очереди – это заголовок, в котором отсутствуют данные. Ниже определен абстрактный тип данных «очередь».

```
Type ptr = ^queue;{Указатель на элемент очереди}
   Queue = record
      data : integer; {Поле данных}
      next, front, rear : ptr;{Указатели на следующий элемент, голову и
 End;                          хвост очереди}
```

Var Q : ptr; {Указатель на заголовок очереди}

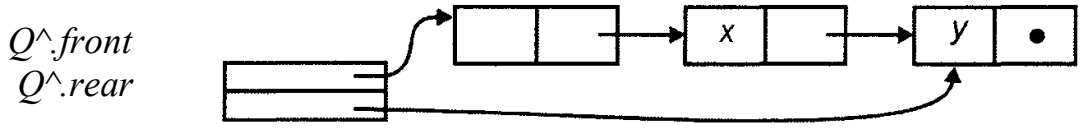
Рассмотрим программную реализацию некоторых из вышеперечисленных операторов над очередями.

1) Procedure Makenull (var q:ptr);
```
Begin
    new(q); {Создание ячейки заголовка}
    q^. front:=q;
    q^. front^.next:=nil;
    q^. rear:=q^.front;
 End;
```



2) Function Empty (q:ptr):boolean;
```
 var z:boolean;
   Begin
     If
       q^.front= q^.rear then     {Проверка пустоты очереди}
            z:=true
    Else    z:=false;
End;
```

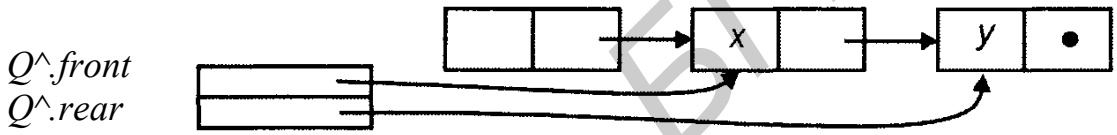
3)Procedure Make (var q:ptr);
```
 Var i: integer;
 begin
     for i:=1 to 2 do  {Добавление в очередь двух элементов }
        begin
          new( q^.rear^.next);
          q^.rear:= q^.rear^.next;
         readln(q^.rear^.data);
      end;
      q^.rear^.next:=nil;
end;
```



4) Procedure Print(var q:ptr); {Вывод на печать элементов очереди}
```
 begin
writeln('Вывод элементов очереди');
```

```
q^.front:=  q^.front^.next;
      while q^.front<> nil do
  begin
     writeln (q^.front^.data);
     q^.front:=  q^.front^.next;
    end;
end;
```

```
5) Procedure Dequeue (var q:ptr); {Удаление первого элемента очереди}
   begin
        if empty (q) then
            writeln('Очередь пуста')
       else
            q^.front:=q^.front^.next;
  end;
```



3.2. Разновидности очередей

Интересной разновидностью очередей являются *многопоточные очереди*. Элементы в нее, как обычно, добавляются в хвост, но очередь имеет несколько потоков или голов. В общем случае многопоточные очереди более эффективны, чем несколько однопоточных очередей.

Циклическая очередь. Некоторые программы можно улучшить, используя циклические очереди. Организуя очереди на основе массива, в случае достижения его предела можно вернуть указатели вставки в очередь и удаления из нее на начало массива. Тогда в очередь можно добавлять и удалять из нее любое число элементов. Такая очередь называется циклической, поскольку массив используется не как линейный список, а как циклический.

В действительности очередь становится заполненной только в том случае, когда указатель свободного места совпадает с указателем выборки следующего элемента. В начале программы индекс выборки должен устанавливаться не в нулевое значение, а на значение максимального числа событий, иначе первое обращение к процедуре вставки в очередь приведет к появлению сообщения о заполнении структуры. Следует помнить, что максимально очередь должна содержать на один элемент меньше, чем может в нее поместиться, поскольку указатели выборки и вставки всегда должны отличаться хотя бы на единицу (в противном случае нельзя будет понять заполнена очередь или она пустая).

Наиболее широко циклические очереди применяются в операционных системах при буферизации информации, которая считывается или записывается на

дисковые файлы или консоль. Другое широкое применение эти очереди нашли в решении задач реального времени, когда, например, пользователь может продолжать ввод с клавиатуры во время выполнения другой задачи. Имеется короткий промежуток времени, когда набранная на клавиатуре информация не выводится на экран до окончания другого процесса. Для достижения такого эффекта в программе должна предусматриваться постоянная проверка ввода с клавиатуры в ходе выполнения другого процесса. При вводе некоторого символа его значение ставится в очередь, а процесс продолжается. По завершении процесса набранные символы выбираются из очереди и обрабатываются нужным образом.

Приоритетные очереди. Каждый элемент такой очереди имеет связанный с ним приоритет. Если программе нужно удалить элемент из очереди, она выбирает элемент с наивысшим приоритетом. Форма хранения элементов в приоритетной очереди не имеет значения, если всегда можно найти элемент с наивысшим приоритетом.

Некоторые операционные системы используют приоритетные очереди для планирования заданий. В операционной системе UNIX все процессы имеют разные приоритеты. Когда процессор освобождается, выбирается готовый к исполнению процесс с наивысшим приоритетом. Процессы с более низким приоритетом должны ждать завершения или блокировки процессов с более высокими приоритетами.

Название «очередь с приоритетом» предполагает, что объекты, требующие обработки, ставятся в очередь, а извлекаются из нее не в порядке занесения, а согласно приоритету.

Алгоритмы приоритетного обслуживания очень популярны во многих областях вычислительной техники, в частности – в операционных системах, когда одним приложениям нужно отдать предпочтение перед другими при их обработке в мультипрограммной смеси. Весь трафик разбивается на небольшое количество классов, каждому из которых присваивается приоритет. Приоритетное обслуживание обычно применяется для класса трафика, чувствительного к задержкам, имеющего небольшую интенсивность. Тогда обслуживание этого класса не слишком ущемляет остальные классы. Например, голосовой трафик (чувствителен, но его интенсивность обычно не превышает 8–16 Кбит/с).

Альтернативой приоритетному обслуживанию являются *взвешенные очереди.* Они гарантируют всем классам трафика определенный минимум пропускной способности. Под весом понимается процент предоставляемой классу трафика пропускной способности от полной пропускной способности выходного интерфейса. С каждой очередью связывается процент пропускной способности ресурса, гарантируемый ему при перегрузках этого ресурса. Совмещение достоинств приоритетных и взвешенных очередей удается получить в *комбинированных алгоритмах.* Обычно в них используется одна приоритетная очередь для чувствительного трафика, а остальные обслуживаются в соответствии с взвешенным алгоритмом. Им выделяется часть интенсивности ресурса, оставшегося от приоритетной очереди.

3.3. Лабораторная работа №3

Цель работы: научиться строить очереди, применять различные алгоритмы для их обслуживания, а также выполнять заданные операции с элементами очередей.

Порядок выполнения работы

1. Ознакомиться с теоретической частью лабораторной работы.
2. Выполнить практическое задание.
3. Оформить отчет по лабораторной работе.

Задание. Необходимо реализовать очередь на базе списков, применяя комбинированный алгоритм для ее обслуживания. Затем продемонстрировать выполнение основных операций с элементами очереди: поиск, добавление, удаление.

4. ПОСТРОЕНИЕ РАЗЛИЧНЫХ ФОРМ ПРЕДСТАВЛЕНИЯ ВЫРАЖЕНИЙ С ИСПОЛЬЗОВАНИЕМ СТЕКА

4.1. Абстрактный тип данных «стек»

Стек – специальный тип списка, в котором все вставки и удаления выполняются только на одном конце, называемом вершиной (*top*). В англоязычной литературе для обозначения стеков используется аббревиатура *LIFO* (*last-in-first-out* – последний вошел и первый вышел).

Удобным вариантом представления стека является способ на основе списка, в котором добавление новых элементов и извлечение имеющихся происходит с одного конца списка. Значением указателя, представляющего стек, является ссылка на вершину стека. Каждое звено стека содержит поле данных и указатель на следующее звено. Абстрактный тип данных «стек» можно представить следующим образом:

```
Type ptr = ^stack; {Указатель на элемент стека}
   Stack = record
    data : integer; {Поле данных}
    next : ptr;{Указатели на следующий элемент стека}
  End;
Var st : ptr;
```

Если стек пуст, то значение указателя *st* равно *nil*.

Рассмотрим программную реализацию нескольких наиболее часто выполняемых операторов с элементами стека:

1) процедура записи элементов в стек содержит один параметр: указатель на начало стека. Эта процедура выполняется аналогично вставке элемента в начало списка;

2) процедура извлечения элементов из стека содержит один параметр: указатель на начало стека. Сначала печатается элемент из вершины стека, а затем изменяется указатель на вершину стека.

```
1)  Procedure WriteStack (var st: ptr);
    Var i :integer;
        x :ptr;
     Begin
        For i:=1 to 5 Do {В стек заносится пять элементов}
         Begin
            New(x);
           Readln(x^.data);
           x^.next:=st;
          st:=x;
         End;
    End;

2) Procedure Print(var st:ptr);
    Begin
       Writeln('Вывод элементов стека');
        While st<> nil do
          Begin
            Writeln (st^.data);
             st:=  st^.next;
          End;
    End;
```

4.2. Различные формы записи выражений

Введем понятие формы записи выражений. *A+B* – инфиксная: знак операции находится между операндами; *+AB* – префиксная (польская): знак операции расположен перед операндами; *AB+* – постфиксная (обратная польская): знак операции находится после операндов.

Хотя префиксная и постфиксная формы записи, на первый взгляд, кажутся не очень наглядными, они чаще инфиксной используются в вычислительной технике для обработки выражений.

Большую часть решаемых с помощью программирования задач составляют те, в которых широко применяются методы вычислительной математики, а в них входят арифметические и логические выражения. Поэтому трансляцией

выражений занимались очень многие исследователи и разработчики. Сейчас классическим стал метод трансляции выражений, основанный на использовании промежуточной обратной польской записи. Она названа в честь польского математика Яна Лукашевича, впервые использовавшего эту форму представления выражений в математической логике.

В ходе преобразования выражений из инфиксной в постфиксную и префиксную формы нужно учитывать правила приоритетности операций. Операции с высшим приоритетом преобразуются первыми, а после преобразования операция рассматривается как один операнд. Общепринятую приоритетность операций можно изменить при помощи скобок. При просмотре строки, не содержащей скобок, вычисления выполняются слева направо для операций с одинаковым приоритетом, за исключением случая возведения в степень, когда вычисления выполняются справа налево. Ниже приведены примеры различных форм записи выражений.

Инфиксное представление	Постфиксное представление
A+B-C	*AB+C-*
(A+B)(C-D)*	*AB+CD-**
*A^B*C-D+E/F/(G+H)*	*AB^C*D-EF/GH+/+*
*A-B/(C*D^E)*	*ABCDE^*/-*

Инфиксное представление	Префиксное представление
A+B-C	*- +ABC*
(A+B)(C-D)*	**+AB-CD*
*(A+B)*C-(D-E)^(F+G)*	*^ -*+ABC-DE+FG*
*A-B/(C*D^E)*	*-A/B*C^DE*

Рассмотрим суть обратной польской записи. В ней отсутствуют скобки, операнды располагаются в том же порядке, что в исходном выражении, а знаки операций при просмотре записи слева направо встречаются в порядке, необходимом для выполнения соответствующих действий. Отсюда вытекает основное преимущество обратной польской записи перед обычной записью выражений со скобками: *выражение можно вычислить в процессе однократного просмотра слева направо.*

4.3. Построение выражений в обратной польской записи

Правило вычисления выражения в обратной польской записи состоит в следующем. Обратная польская запись просматривается слева направо. Если очередной элемент – операнд, то происходит переход к следующему элементу. Если текущий элемент – знак операции, то над операндами, записанными левее знака операции, выполняется данная операция. Ее результат записывается вместо первого (самого левого) операнда, участвовавшего в операции. Остальные элементы (операнды и знак операции), участвовавшие в операции, вычеркива-

ются из записи. Просмотр продолжается. В результате последовательного выполнения изложенного правила будут выполнены все операции в выражении, и запись сократится до одного элемента – результата вычисления выражения.

Для управления порядком выполнения операций над операндами используется скобочная запись выражений. При вычислении сначала вычисляется часть выражения, скобочный уровень вложенности которой самый высокий. При наличии нескольких операторов с наивысшим скобочным уровнем, они вычисляются слева направо. При вычислении выражений с операциями, которым присвоен приоритет, или частично скобочных выражениях инфиксной формы записи, необходимо повторное сканирование слева направо. Повторное сканирование можно исключить, если инфиксное выражение преобразовать в постфиксную или префиксную форму записи.

Основные правила выполнения постфиксного выражения:

1) найти в выражении крайний левый оператор;

2) выбрать два операнда, стоящих непосредственно слева от найденного оператора;

3) выполнить операцию;

4) заменить оператор и операнды результатом;

5) повторять указанные действия, пока не будут обработаны все операнды.

Постфиксное и префиксное выражения корректны тогда и только тогда, когда ранг выражения равен 1, а ранг любой правой головы польской формулы больше (меньше) или равен 1. Ранг корректного выражения равен 1.

Алгебраическое преобразование инфиксного выражения в обратное польское основано на приоритетах операторов и предлагает использование стека. Обратное польское выражение хранится в виде выходной строки, используемой в дальнейшем при генерации объектного кода.

В ходе преобразования инфиксного выражения в обратное польское порядок всех переменных и констант не меняется, а порядок операторов выходной строки соответствует их приоритетам (табл. 4.1).

Таблица 4.1

Символ	Приоритет	Ранг
+,–	1	–1
*, /	2	–1
A,b,…, z	3	1
Дно стека	0	–

Рассмотрим пример преобразования инфиксного выражения *a+b*c–d/e*h* в обратное польское (табл. 4.2).

Таблица 4.2

Сканируемый символ	Содержание стека	Обратная польская запись	Ранг
	|–		
A	|– *a*		
+	|– +	*a*	1
b	|– +*b*	*a*	1
*	|– +*	*ab*	2
c	|– +**c*	*ab*	2
–	|– –	*abc*+	1
d	|– –*d*	*abc*+	1
/	|– – /	*abc*+*d*	2
e	|– – / *e*	*abc*+*d*	2
*	|– – *	*abc*+*de*/	2
h	|– – * *h*	*abc*+*de*/	2
|–	|–	*abc*+*de*/*h**–	1

Алгоритм преобразования.

1. В стек помещается признак пустого стека.

2. Значение приоритета очередного входного символа сравнивается с приоритетом верхнего элемента стека.

3. Если приоритет символа больше приоритета верхнего элемента стека, то символ помещается в стек, выбирается следующий входной символ.

4. Если приоритет входного символа меньше или равен приоритету верхнего элемента стека, то этот элемент удаляется из стека и помещается в формируемую строку, после чего сравниваются приоритеты очередного символа и нового верхнего символа.

Каждый раз при изменении обратной польской записи пересчитывается ранг результирующего выражения.

4.4. Преобразование скобочных выражений в обратную польскую запись

Для преобразования скобочных выражений воспользуемся способом голландского ученого Дейкстры. Его метод основан на использовании *стека с приоритетами*, позволяющего изменить порядок следования знаков операций в выражении так, что получается обратная польская запись. Этот метод применим к арифметическим и логическим выражениям, содержащим переменные, знаки арифметических и логических операций, знаки операций отношения и круглые скобки. Каждому символу, за исключением операндов, присваивается приоритет, приведенный в табл. 4.3.

Таблица 4.3

Операция	Приоритет
(	0
)	1
+,–	2
*, /	3
^	4

Арифметическое или логическое выражение рассматривается как входная строка символов. Операнды переписываются в выходную строку, а знаки операций помещаются сначала в стек операций. Если приоритет входного знака равен нулю или больше приоритета знака, находящегося в вершине стека, то новый знак добавляется к вершине стека. В противном случае из стека «выталкивается» и переписывается в выходную строку знак, находящийся в вершине, а также следующие за ним знаки с приоритетами, большими или равными приоритету входного знака. После этого входной знак добавляется к вершине стека. Особенности имеет лишь обработка скобок. Открывающая круглая скобка, имеющая приоритет нуль, записывается в вершину стека и не выталкивает ни одного знака. В то же время ее не может вытолкнуть ни один знак, кроме закрывающей скобки. Ее появление вызывает выталкивание всех знаков до ближайшей открывающей скобки включительно. Скобки взаимно уничтожаются и в выходную строку не переносятся. По завершении просмотра всех символов входной строки происходит выталкивание оставшихся в стеке символов и добавление их к выходной строке.

Ниже приведены два примера преобразования инфиксных выражений в обратные польские.

Пример 1.

((A–(B+C))*D)^(E+F)

N п/п	Входная строка	Стек	Постфиксная строка
1	(	(	
2	(	((	
3	A	((	A
4	-	((-	A
5	(	((-(	A
6	B	((-(	AB
7	+	((-(+	AB
8	C	((-(+	ABC
9	)	((-	ABC+
10	)	(	ABC+-
11	*	(*	ABC+-
12	D	(*	ABC+-D
13	)	пусто	ABC+-D*
14	^	^	ABC+-D*
15	(	^(	ABC+-D*
16	E	^(	ABC+-D*E
17	+	^(+	ABC+-D*E
18	F	^(+	ABC+-D*EF
19	)	^	ABC+-D*EF+
20			ABC+-D*EF+^

Пример 2.

a+b*c–d/(a+b)

Входная строка	Стек				Выходная строка										
a					a										
+	+				a										
b	+				a	b									
×	+	×			a	b									
c	+	×			a	b	c								
-	-				a	b	c	×	+						
d	-				a	b	c	×	+	d					
/	-	/			a	b	c	×	+	d					
(	-	/	(		a	b	c	×	+	d					
a	-	/	(		a	b	c	×	+	d	a				
+	-	/	(	+	a	b	c	×	+	d	a				
b	-	/	(	+	a	b	c	×	+	d	a	b			
)	-	/			a	b	c	×	+	d	a	b	+		
					a	b	c	×	+	d	a	b	+	/	-

Алгоритм преобразования выражения из инфиксной формы в префиксную запись рассмотрим на примере выражения *a+b/(c–d)*.

1. Необходимо переписать выражение справа налево: *(d–c)/b+a*;

2. Воспользовавшись алгоритмом постфиксной трансляции, получим: *dc–b/a+;*

3. Полученную строку требуется записать справа налево, в результате чего получается выражение в префиксном виде: *+a/b–cd*.

4.5. Лабораторная работа №4

Цель работы: научиться строить выражения в префиксной, постфиксной и инфиксной формах записи, а также преобразовывать выражения из одной формы в другую.

Порядок выполнения работы

1. Ознакомиться с теоретической частью лабораторной работы.
2. Выполнить практическое задание.
3. Оформить отчет по лабораторной работе.

Задание.

1. Используя стек, реализовать алгоритм преобразования алгебраического выражения из инфиксной формы записи в постфиксную форму представления.

2. Используя стек, реализовать алгоритм преобразования алгебраического выражения из инфиксной формы записи в префиксную форму представления.

Для обоих алгоритмов предусмотреть вхождение операций с различными приоритетами, а также наличие скобок в инфиксных выражениях.

5. ПОСТРОЕНИЕ БИНАРНОГО ДЕРЕВА ПОИСКА. ОБХОДЫ ДЕРЕВА И РЕАЛИЗАЦИЯ ОПЕРАЦИЙ С ДАННЫМИ

5.1. Бинарные деревья

Дерево представляет собой иерархическую структуру какой-либо совокупности элементов (узлов) и отношений, образующих иерархическую структуру узлов. Один из узлов определен как корень. Узлы дерева могут быть элементами любого типа и обычно изображаются буквами, строками или числами.

Формально дерево можно определить следующим образом.

Один узел является деревом, он же является корнем этого дерева.

Пусть n – это узел, а $T_1, T_2, \ldots T_k$ – деревья с корнями $n_1, n_2, \ldots n_k$ соответственно. Можно построить новое дерево, сделав n родителем узлов $n_1, n_2, \ldots n_k$. В этом дереве n будет корнем, а $T_1, T_2, \ldots T_k$ – поддеревьями этого корня. Узлы $n_1, n_2, \ldots n_k$ называются сыновьями узла n.

В древовидной структуре число потомков вершины называется ее степенью. Максимальное значение этих степеней называется степенью дерева. Наибольшую популярность в программировании и вычислительной технике получили *бинарные (двоичные) деревья*, у которых степень дерева равна двум. В этом случае вершина дерева может иметь не более двух потомков, называемых *левым* и *правым* сыновьями. В отдельный подкласс *бинарных деревьев* выделены *деревья поиска*. Они характеризуются тем, что значение информационного поля, связанного с вершиной дерева, больше любого соответствующего значения из левого поддерева и меньше, чем содержимое любого узла его правого поддерева. Описание вершины дерева имеет следующий вид:

```
Type pt=^node;
   node=record
      data:integer; {Информационное поле}
      left,right:pt; {Указатели на левого и правого потомков}
      End;
 Var  root  :pt; {Указатель на корень дерева}
```

К основным операциям, выполняемым с деревом, относятся поиск элемента, вставка элемента, удаление элемента, обход дерева.

Создание бинарного древа можно реализовать на основе операции вставки элемента. Ниже приведена соответствующая процедура.

```
Procedure Insert (var x : pt;  y:integer);  {x – указатель на корень дерева, y –
    Begin                                    вставляемая вершина}
      If x=nil then
         Begin
            new(x);
            x^.data:=y;
            x^.left:=nil;
            x^.right:=nil;
          End
      Else If y<= x^.data then Insert (x^.left,y)
           Else Insert (x^.right,y);
  End;
```

Реализация удаления элемента из дерева выполнятся чуть сложнее. Если узел имеет одного потомка, то в поле ссылки родителя удаляемого элемента записывается ссылка, не равная *nil*. В том случае, когда у удаляемого элемента два потомка, для сохранения структуры дерева поиска на место этого элемента необходимо записать или самый правый элемент левого поддерева, или самый левый элемент правого поддерева. На рис. 5.1 приведено бинарное дерево, из которого необходимо удалить узел со значением 50. На рис. 5.2 показано дерево после удаления из него заданного элемента по первому правилу.

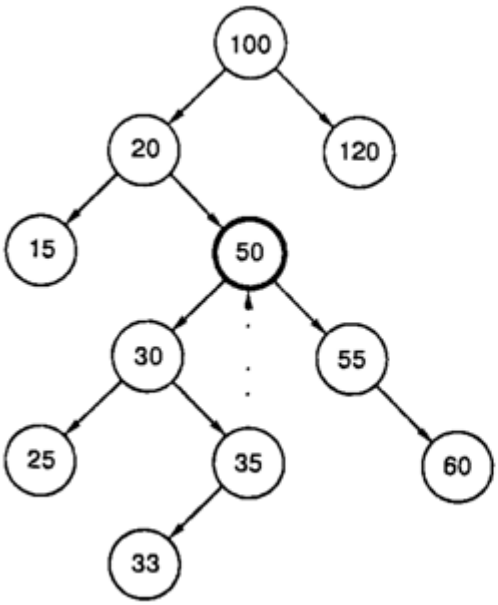


Рис. 5.1. Исходное дерево

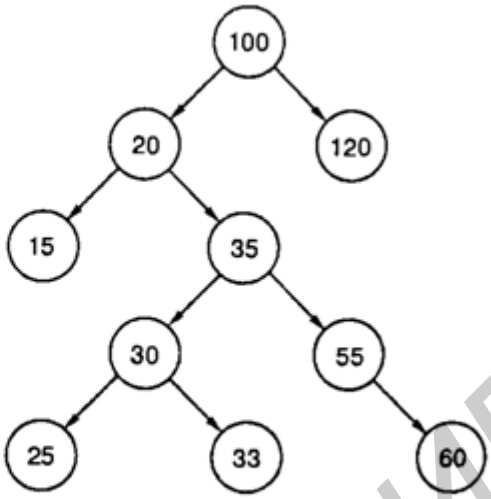


Рис. 5.2. Дерево после удаления вершины

```
{Удаление элемента из дерева}
procedure Del_tree (var x : pt;  y:integer);
  Var q:pt;
{Поиск самого правого элемента в дереве}
  procedure Del (var w : pt);
    Begin
      If w^.right<>nil then Del(w^.right)
        Else  Begin
          q:=w; {Запоминается адрес для освобождения места в «куче» }
          x^.data:=w^.data;
          w:=w^.left;
        End;
    End;
       Begin
        If x<>nil then
         If y<x^.data then  Del_tree (x^.left, y)
           Else if y>x^.data then  Del_tree (x^.right, y)
            Else Begin
                  q:=x;
                  {Правого поддерева нет}
                 If x^.right=nil Then x:=x^.left
                   {Левого поддерева нет}
                    Else If x^.left=nil Then x:=x^.right
                       Else del(x^.left); {Поиск самого правого элемента в левом
                 dispose(q);              поддереве}
                 End;
    End;
```

5.2. Обходы дерева

В ходе решения прикладных задач с применением структур в виде деревьев очень часто выполняются различные обходы дерева. Существует несколько способов обхода (прохождения) всех узлов дерева. Наиболее популярны три следующих способа обхода дерева: прямой (сверху вниз), обратный (снизу вверх), слева направо (симметричный).

Все три способа обхода дерева можно определить рекурсивно следующим образом.

Если дерево T является нулевым деревом, то в список обхода заносится пустая запись.

Если дерево T состоит из одного узла, то в список обхода записывается этот узел.

Далее, пусть T – дерево с корнем n и поддеревьями $T_1, T_2, ..., T_k$, тогда для различных способов обхода имеем следующее:

а) при прохождении в прямом порядке (т. е. при прямом упорядочивании) узлов дерева T сначала посещается корень n, а затем узлы поддерева T_1, далее все узлы поддерева T_2 и т. д. Последним посещаются узлы поддерева T_k;

б) при симметричном обходе узлов дерева T сначала посещаются в симметричном порядке все узлы поддерева T_1, далее корень, n, затем последовательно в симметричном порядке все узлы поддеревьев $T_2, ..., T_k$;

в) в случае обхода в обратном порядке сначала посещаются в обратном порядке все узлы поддерева T_1, затем последовательно посещаются все узлы поддеревьев $T_2, ..., T_k$ также в обратном порядке, последним посещается корень n.

Для бинарного дерева, изображенного на рис. 5.3, рассмотрим рекурсивные процедуры, реализующие три способа обхода дерева.

После каждой процедуры приводится последовательность узлов дерева, выделенная жирным шрифтом, соответствующая указанному способу обхода. Подчеркиванием помечены узлы, в которые возвращается процедура в ходе рекурсивного вычислительного процесса.

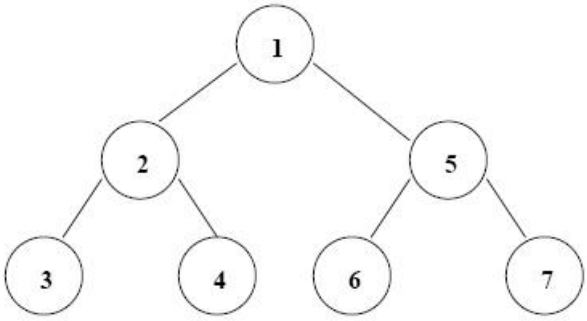


Рис. 5.3. Дерево обхода

Прямой обход дерева:

```
Procedure prym_print(var x:pt);
 Begin
  if  x<> nil then
    Begin
     write (x^.data);
     prym_print(x^.left);
     prym_print(x^.right);
  End;
 End;
```

Последовательность обработки: **1, 2, 3,** 2, **4**, 2, 1, **5, 6,** 5, **7,** 5, 1

Симметричный обход дерева:

```
Procedure sim_print(var x:pt);
  Begin
    if  x<> nil then
      Begin
      sim_print(x^.left);
      write (x^.data);
      sim_print(x^.right);
    End;
End;
```

Последовательность обработки: 1, 2, **3, 2, 4**, 2, **1**, 5, **6, 5, 7,** 5, 1

Обратный обход дерева:

```
 Procedure obr_print(var x:pt);
  Begin
    if  x<> nil then
      Begin
        obr_print(x^.left);
        obr_print(x^.right);
        write (x^.data);
     End;
  End;
```

Последовательность обработки: 1, 2, **3,** 2**, 4**, **2**, 1, 5, **6,** 5, **7, 5, 1**

5.3. Бинарные деревья поиска

Бинарные деревья чаще всего применяются для представления множеств данных, элементы которых ищутся по уникальному, только им присущему ключу. Если бинарное дерево организовано таким образом, что для каждого узла t_i все ключи в левом поддереве меньше ключа t_i, а ключи в правом поддереве больше ключа t_i, то это дерево называется *бинарным деревом поиска*. В дереве поиска можно найти место каждого ключа, двигаясь, начиная от корня и переходя на левое или правое поддерево каждого узла в зависимости от значения ключа. Таким образом, места элементов в дереве определяются как значениями ключей, так и последовательностью их поступления. Определяющим фактором является значение ключа, от последовательности поступления элементов зависит степень сбалансированности дерева. При случайном распределении ключей в исходной последовательности получается почти сбалансированное дерево. Если же исходная последовательность упорядочена по возрастанию или убыванию ключей, то дерево вырождается в последовательный список. Высота такого дерева равна числу элементов дерева, уменьшенному на 1.

Построение дерева поиска заключается в следующем. Первый элемент образует корень дерева. Для последующих элементов осуществляется поиск места включения по ветвям дерева до тех пор, пока не будет найден подходящий узел с нулевым указателем, туда и добавляется элемент. Для каждого узла запрашивается динамическая память, ее адрес заносится в указатель узла-предка, данные элемента помещаются в узел, и обнуляются левый и правый указатели нового узла.

Дерево поиска, построенное из последовательности ключей 9, 17, 20, 16, 12, 21, 6, 3, 11, 4, 19, 14, 13, 1, 5, 2, 8, 18, 7, 10, 15, имеет вид, приведенный на рис. 5.4.

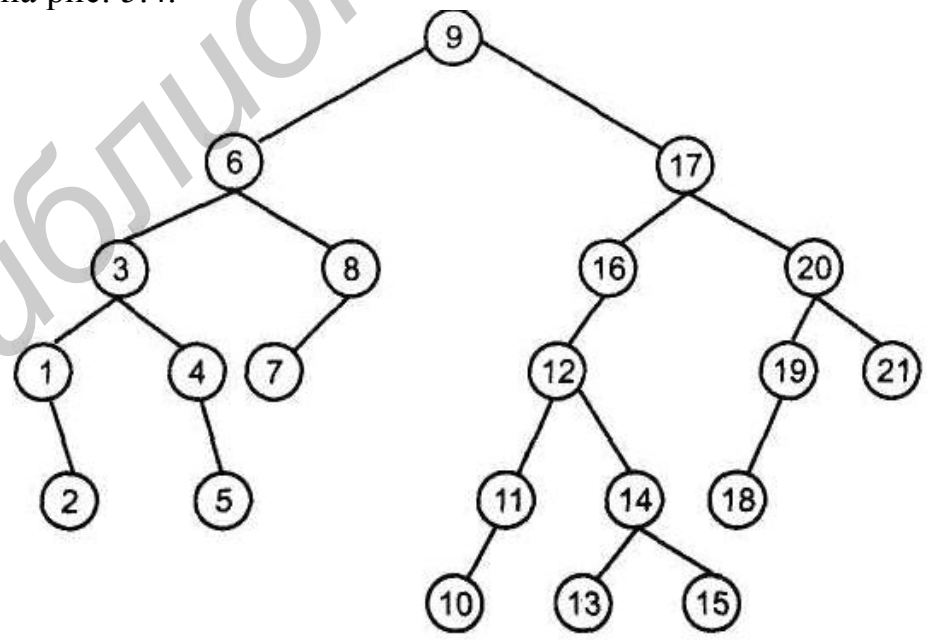


Рис. 5.4. Бинарное дерево поиска

5.4. Лабораторная работа №5

Цель работы: научиться строить и обходить бинарные деревья, а также выполнять на них различные операции с данными.

Порядок выполнения работы

1. Ознакомиться с теоретической частью лабораторной работы.
2. Реализовать практическое задание.
3. Оформить отчет по лабораторной работе.

Задание. С помощью указателей построить бинарное дерево поиска. Обойти его в прямом, симметричном и обратном порядке. Реализовать процедуры поиска, вставки и удаления элементов в бинарное дерево поиска.

6. ПРОШИТЫЕ БИНАРНЫЕ ДЕРЕВЬЯ. ОБХОДЫ И РЕАЛИЗАЦИЯ ОСНОВНЫХ ОПЕРАЦИЙ С ДАННЫМИ

6.1. Прошитые бинарные деревья

В случае представления бинарного дерева в виде узлов, содержащих информационное поле и два поля связи, количество полей связи, имеющих значения *nil*, всегда больше числа связей, указывающих на реально существующие узлы. Поэтому часто такой способ хранения деревьев оказывается неэффективным с точки зрения использования памяти, особенно, если размер информационного поля сопоставим с размером указателя.

Эффективность прохождения дерева рекурсивными и нерекурсивными алгоритмами может быть увеличена, если использовать пустые указатели на отсутствующие поддеревья для хранения в них адресов узлов преемников, которые надо посетить при заданном порядке прохождения бинарного дерева. Такой указатель называется *нитью*. Его следует отличать от указателей в дереве, которые используются с левым и правым поддеревьями. На рис. 6.1 показано дерево, *симметрично прошитое справа.* На нем пунктирными линиями обозначены прошивочные нити.

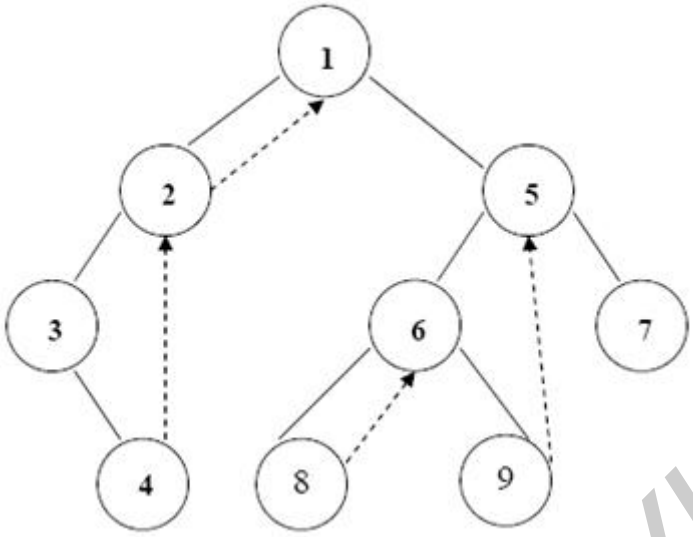


Рис. 6.1. Симметрично прошитое справа бинарное дерево

Операция, заменяющая пустые указатели на нити, называется *прошивкой*. Она может выполняться по-разному. Если нити заменяют пустые указатели в узлах с пустыми правыми поддеревьями при просмотре в симметричном порядке, то бинарное дерево называется *симметрично прошитым справа.* Похожим образом может быть определено бинарное дерево, *симметрично прошитое слева:* дерево, в котором каждый пустой левый указатель изменен так, что он содержит нить – связь к предшественнику данного узла при просмотре в симметричном порядке. *Симметрично прошитое* бинарное дерево – то, которое симметрично прошито слева и справа. Однако левая прошивочная нить не дает тех преимуществ, что правая прошивочная нить.

Также используются бинарные деревья, *прямо прошитые* справа и слева. В них пустые правые и левые указатели узлов заменены соответственно на их преемников и предшественников при прямом порядке просмотра. Как и в случае с бинарными деревьями, симметрично прошитыми слева, левая прошивка при прямом обходе дерева не имеет преимуществ правосторонней прошивки.

Прошитые деревья эффективно проходятся без использования стека.

Поскольку нужно каким-то образом отличать обычную связь от прошивочной нити, каждому узлу добавляется два однобитовых (логических) поля тэга: *ltag* и *rtag*. Если значение тэга *true*, соответствующее поле связи является обычной связью, в случае значения *false* – прошивочной нитью.

В этой связи узел обычного бинарного дерева представляется структурой:

```
Type ptr = ^node;
     node = record
     info : integer;   {Информационное поле}
     left, right : pt;  {Указатели на левое и правое поддеревья}
 end;
```

Узел прошитого бинарного дерева имеет иную структуру:

```
Type ptr = ^node;
     node = record
     info : integer;    {Информационное поле}
     ltag, rtag : boolean; {Тэги прошивочных нитей}
   left, right : pt;  {Указатели на левое и правое поддеревья}
end;
```

Логические поля в прошитом дереве могут принимать следующие значения:

1) *ltag=true*, следовательно, *left* представляет собой обычную связь;
2) *ltag=false*, следовательно, *left* указывает на узел-предшественник;
3) *rtag=true*, следовательно, *right* представляет собой обычную связь;
4) *rtag=false*, следовательно, *right* указывает на узел-преемник.

Рассмотрим вставку новой вершины слева от заданной в симметрично прошитое бинарное дерево (рис. 6.2). На рис. 6.3 показано результирующее дерево.

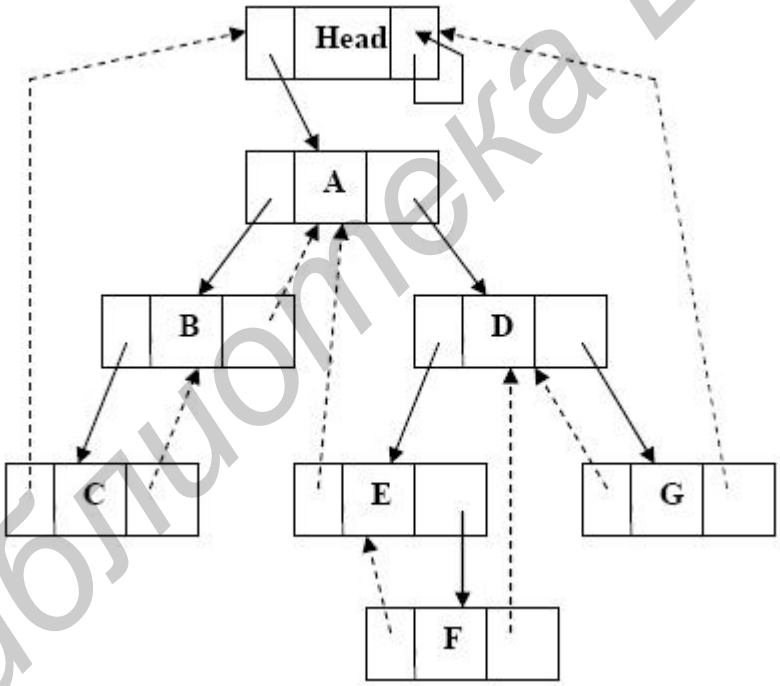


Рис. 6.2. Исходное симметрично прошитое дерево

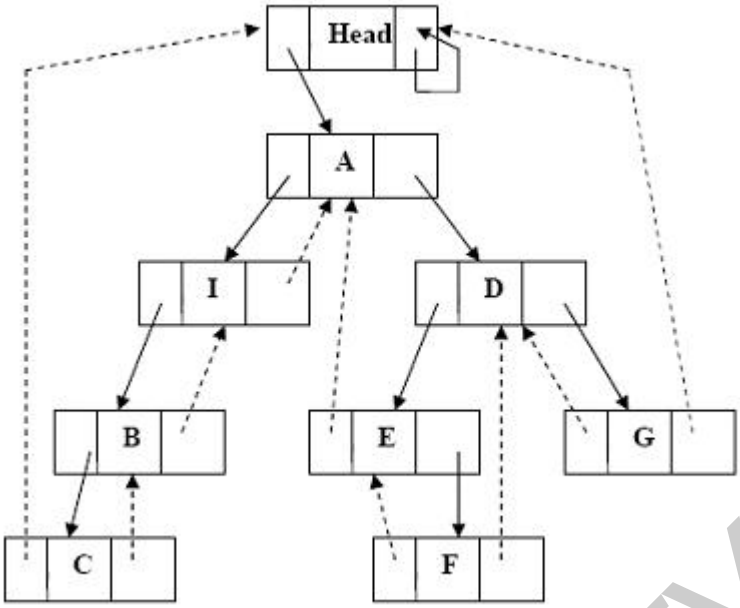


Рис. 6.3. Прошитое дерево после вставки в него нового элемента

Здесь требовалось вставить вершину *I* в качестве левого поддерева вершины *A*, если *A* не имеет левого поддерева. В противном случае, новая вершина вставляется между *A* и ее левым сыном.

Для удобства создания и обхода дерева используется дополнительная головная вершина *Head*, которая служит при симметричном обходе предшественником его первой вершины и приемником всех его концевых вершин.

Наряду с преимуществами прошитых деревьев – быстрый обход, отсутствие необходимости в стеке, можно определить предшественника и преемника вершины – существуют недостатки. Включение новой вершины в дерево занимает больше времени, т. к. необходимо поддерживать два типа связей: структурные и по нитям. Поэтому прошитые деревья целесообразно использовать в тех задачах, где изменения в деревьях происходят редко, а обходы выполняется часто.

6.2. Алгоритмы прошивки и обхода прошитых деревьев

Рассмотрим алгоритм правосторонней симметричной прошивки бинарного дерева.

1. Строится бинарное дерево. При этом поля *ltag* и *rtag* создаваемых узлов дерева остаются неопределенными, а *left* и *right* соответственно указывают на левое или правое поддеревья, либо равны *nil*. На корень построенного дерева указывает *root*.

2. Создается головной узел, *left* которого указывает на корень дерева, а *right* на сам головной узел:

```
new(HEAD);
HEAD^.left := root;
```

```
HEAD^.right := HEAD;
```

Информационное поле и поля тэгов головного узла можно оставить неопределенными.

3. Прошивка правых связей. Вводится дополнительный глобальный указатель *y* (указатель на узел, предшествующий текущему узлу). Указатель на текущий узел *p* устанавливаем на корень дерева.

Выполняется симметричный обход дерева. При обработке каждого узла проверяется: если *p^.right <>nil*, то *p^.rtag:=true;* иначе *p^.rtag:=false; y:=p;* (указатель на предшественника).

Программный код процедур симметричного обхода *sim_print* и прошивки правых связей *rightsew* будет выглядеть так:

```
procedure  sim_print(var x:pt);
     procedure  rightsew( var p:pt);
       begin
        if y <> nil then  {значение y присваивается в вызывающей программе}
           begin
             if y^.right=nil then
               begin
                  y^.rtag := false;
                  y^.right := p;
               end
           else    y^.rtag := true;
         end;
        y:=p;
      end;
   begin
     if  x<> nil then
       begin
       sim_print(x^.left);
       rightsew(x);
       sim_print(x^.right);
     end;
   end;
```

Рассмотрим алгоритм обхода прошитого бинарного дерева. Пусть *HEAD* – указатель на головной узел прошитого дерева, *p* – указатель на текущий узел. Тогда алгоритм симметричного обхода прошитого дерева можно сформулировать следующим образом:

1. Переход к корню дерева (*p := HEAD^.left*);

2. До тех пор, пока *p^.left<>nil*, повторять: *p := p^.left*, то есть идти по левой ветви до самого левого узла;

3. Обработка узла *p*, например, печать *p^.info*;

4. Если *p^.rtag* равен *false*, то *p := p^.right* и переход к шагу 3 (к преемнику). Иначе *p:= p^.right* и переход к шагу 2.

Алгоритм заканчивает работу, когда *p* станет равным *HEAD*.

Программный код процедуры обхода прошитого дерева.

```
procedure obxod_proshiv(var x:pt);
 begin
    while x<>head do
     begin
        while x^.left<>nil   do x:=x^.left;
           begin
             writeln(x^.data);
             while x^.rtag = false do begin
            x:=x^.right;
           if x=head then exit;
          writeln(x^.data);
                                    end;
       x:=x^.right;

         end;
  end;
 end;
```

6.3. Лабораторная работа №6

Цель работы: изучить различные способы прошивки деревьев с последующим их обходом и выполнением операций с данными.

Порядок выполнения работы

1. Ознакомиться с теоретической частью лабораторной работы.
2. Выполнить практическое задание.
3. Оформить отчет по лабораторной работе.

Задание.

1. Выполнить симметричную прошивку бинарного дерева поиска. Обойти его согласно симметричному порядку следования элементов. Реализовать вставку и удаление элементов из симметрично прошитого бинарного дерева.

2. Выполнить прямую прошивку бинарного дерева поиска. Обойти его согласно прямому порядку следования элементов. Реализовать вставку и удаление элементов из прямо прошитого бинарного дерева.

7. ПОИСК МАРШРУТОВ НА ОРИЕНТИРОВАННЫХ ГРАФАХ

7.1. Основные определения ориентированных графов

Ориентированный граф (или орграф) *G=(V, E)* состоит из множества вершин *V* и множества дуг *E*. Вершины также называют узлами, а дуги – ориентированными ребрами. Дуга представляется в виде упорядоченной пары вершин *(v, w)*, где вершина *v* называется началом, а *w* – концом дуги. Дугу *(v, w)* часто записывают как $v \to w$ и изображают, как показано на рис. 7.1.

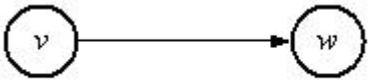


Рис. 7.1. Дуга (*v*, *w*)

Кроме того, дуга $v \to w$ ведет от вершины *v* к вершине *w*, а вершина *w* смежная с вершиной *v*. На рис. 7.2 показан орграф с четырьмя вершинами и пятью дугами.

Вершины орграф можно использовать для представления объектов, а дуги – для отношений между объектами. Например, вершины орграфа могут представлять города, а дуги – маршруты рейсовых полетов самолетов из одного города в другой. В виде орграфа может быть представлена блок-схема потока данных в компьютерной программе. В последнем примере вершины соответствуют блокам операторов программы, а дугам – направленное перемещение потоков данных.

Путем в орграфе называется последовательность вершин $v_1, v_2, ..., v_{n-1}, v_n$, для которых существуют дуги $v_1 \to v_2, v_2 \to v_3, ..., v_{n-1} \to v_n$. Этот путь начинается в вершине v_1 и, проходя через вершины $v_2, ..., v_{n-1}$, заканчивается в вершине v_n. Длина пути – количество дуг, составляющих путь, в данном случае длина пути равна *n*–1. Одна вершина рассматривается как путь длины 0 от вершины *v* к этой же вершине *v*.

Путь называется простым, если все вершины на нем, за исключением, может быть, первой и последней, различны. Цикл – это простой путь длины не менее 1, который начинается и заканчивается в одной и той же вершине. На рис. 7.2 вершины 3, 2, 4, 3 образуют цикл длины 3.

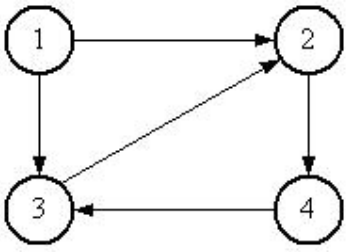


Рис. 7.2. Пример орграфа

Во многих приложениях удобно к вершинам и дугам присоединить какую-либо информацию. Для этих целей используется помеченный граф, т. е. орграф, у которого каждая дуга и/или каждая вершина имеет соответствующие метки. Меткой может быть имя, вес или стоимость (дуги), или значение данных какого-либо заданного типа.

7.2. Представление ориентированных графов

Для представления ориентированных графов можно использовать различные структуры данных. Выбор структуры данных зависит от операторов, которые будут применяться к вершинам и дугам орграфа.

Одним из часто используемых способов представления орграфа *G=(V, E)* является *матрица смежности*. Предположим, что множество вершин орграфа *V={1, 2, ..., n}*, тогда матрица смежности графа *G* – это матрица *A* размера $n \times n$ со значениями булевого типа, где *A[i, j]=true* тогда и только тогда, когда существует дуга из вершины *i* в вершину *j*. Часто в матрице смежности значение *true* заменяется на 1, а значение *false* – на 0. Время доступа к элементам матрицы смежности зависит от размеров множества вершин и множества дуг. Представление орграфа в виде матрицы смежности удобно применять в тех алгоритмах, в которых надо часто проверять существование данной дуги.

С помощью матрицы смежности можно представлять и помеченные орграфы. В этом случае элемент *A*[*i, j*] равен метке дуги $i \to j$. Если дуги от вершины *i* к вершине *j* не существует, то значение *A*[*i, j*] может рассматриваться как пустая ячейка. На рис. 7.3 изображен помеченный орграф и соответствующая ему матрица смежности.

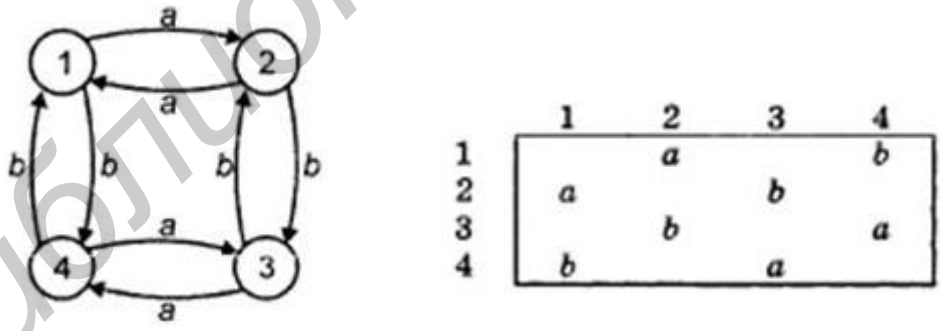


Рис. 7.3. Помеченный орграф и соответствующая ему матрица смежности

Основной недостаток матриц смежности заключается в том, что она требует объема памяти, равного n^2, даже если дуг значительно меньше, чем n^2. Поэтому для чтения матрицы или нахождения в ней необходимого элемента требуется время порядка n^2, что не позволяет создавать алгоритмы со временем *n* для работы с орграфами, имеющими порядка *n* дуг.

Поэтому вместо матриц смежности могут использовать представления орграфов с помощью *списков смежности*. Списком смежности для вершины *i* называется список всех вершин, смежных с вершиной *i*, причем упорядоченный определенным образом. Следовательно, орграф *G* можно представить с помощью массива *HEAD*, чей элемент *HEAD[i]* является указателем на список смежности вершины *i*. Представление орграфов посредством списков смежности требует для хранения объем памяти, пропорциональный сумме количества вершин и количества дуг. Если количество дуг имеет порядок *n*, то общий объем необходимой памяти имеет такой же порядок. Но и для списков смежности время поиска определенной дуги может иметь порядок *n*, т. к. определенная вершина может иметь количество дуг такого же порядка. На рис. 7.4 показана структура данных, представляющая орграф с рис. 7.2 посредством связанных списков смежности. Если дуги имеют метки, то их можно хранить в ячейках связанных списков. Для вставки и удаления элементов в списках смежности необходимо иметь массив *HEAD*, содержащий указатель на ячейки заголовков списков смежности, но не сами смежные вершины.

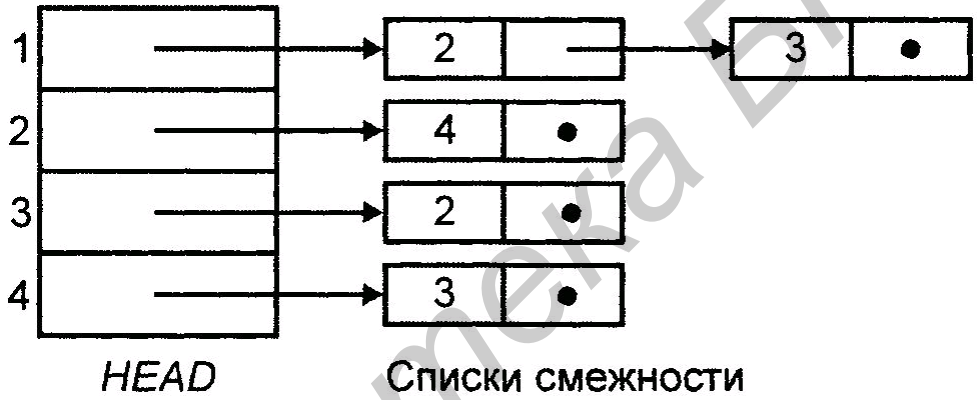


Рис. 7.4. Структура списков смежности для орграфа

7.3. Нахождение кратчайшего пути на ориентированном графе

Пусть есть ориентированный граф *G = (V, E)*, у которого все дуги имеют неотрицательные метки, а одна вершина определена как источник. Задача состоит в нахождении стоимости кратчайших путей от источника ко всем другим вершинам граф *G*. Длина пути определяется как сумма стоимостей дуг, составляющих путь. Эта задача часто называется задачей нахождения кратчайшего пути с одним источником. При этом длина пути может измеряться в любых единицах измерения, например, оцениваться временем.

Для решения поставленной задачи будем использовать алгоритм Дейкстры. Алгоритм строит множество *S* вершин, для которых кратчайшие пути от источника уже известны. На каждом шаге к множеству *S* добавляется та из вершин графа, расстояние до которой от источника меньше, чем для других оставшихся вершин. Если стоимости всех дуг неотрицательны, то кратчайший путь от источника к конкретной вершине проходит только через вершины мно-

жества *S*. Такой путь называют *особым.* В ходе выполнения алгоритма заполняется массив *D*, в который записываются длины кратчайших особых путей для каждой вершины. Когда множество *S* будет содержать все вершины орграфа, т. е. для всех вершин будут найдены особые пути, тогда массив *D* будет содержать длины кратчайших путей от источника к каждой вершине. Ниже приведен фрагмент программного кода алгоритма Дейкстры. *FL* – массив типа *Boolean*. Если значение его элемента равно *false*, то соответствующий ему элемент массива *D* должен проверяться на предмет поиска кратчайшего пути, в противном случае этот элемент больше не участвует в рассмотрении.

```
Program Deikstra;
{ Объявление переменных и ввод данных}
...
Begin
For m:=1 to n do
  P[m]:=1; {P – массив, используемый для построения кратчайших путей}
    For j:=1 to n do
     D[j]:=C[i,j]; {C -  матрица цен}
       For j:=2 to n do
    Begin
          m:=2;
            While fl[m]= true do m:=m+1;
            w:=D[m];
           s[j]:=m;
         For k:=m+1 to n do
          if fl[k]=false then
             if D[k]< w then
               begin
                 w:=D[k];
                 s[j]:=k;
              end;
         v:=s[j];
         fl[v]:=true;
         for i:=1 to n do begin
          if D[i]> D[v]+C[v,i] then P[i]:=v;
          D[i]:=min(D[i], D[v]+C[v,i]);
        end;
end;
```

Здесь предполагается, что в орграфе *G* вершины поименованы целыми числами, т. е. множество вершин *V*= {*1, 2, ..., n*}, причем вершина 1 является источником. Массив *C* – это двумерный массив стоимостей, где элемент *C*[*i, j*] равен стоимости дуги $i \to j$. Если дуги $i \to j$ не существует, то *C*[*i, j*] присваи-

вается значение, большее любой фактической стоимости дуг. Здесь оно обозначено как ∞. На каждом шаге $D[i]$ содержит длину текущего кратчайшего особого пути к вершине i.

Применим алгоритм Дейкстры для ориентированного графа, показанного на рис. 7.5.

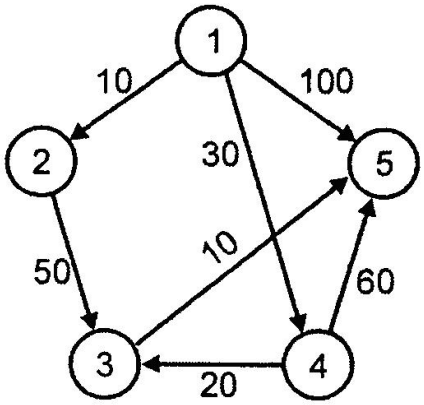


Рис. 7.5. Помеченный орграф, к которому применен алгоритм Дейкстры

Вначале $S=\{1\}$, $D[2]=10$, $D[3]=\infty$, $D[4]=30$, $D[5]=100$. На первом шаге цикла $w=2$, т. е. вершина 2 имеет минимальное значение в массиве D. Затем вычисляется $D[3]=\min(\infty, 10+50)=60$. $D[4]$ и $D[5]$ не изменяются, т. к. не существует дуг, исходящих из вершины 2 и ведущих к вершинам 4 и 5. Последовательность значений элементов массива D после каждой итерации цикла показана в табл. 7.1.

Таблица 7.1

Итерация	S	w	$D[2]$	$D[3]$	$D[4]$	$D[5]$
Начало	{1}	–	10	∞	30	100
1	{1, 2}	2	10	60	30	100
2	{1, 2, 4}	4	10	50	30	90
3	{1, 2, 4, 3}	3	10	50	30	60
4	{1, 2, 4, 3, 5}	5	10	50	30	60

В рассмотренный алгоритм можно внести изменения, которые позволят определить кратчайший путь для любой вершины графа. Для этого нужно ввести еще один массив P, где $P[v]$ содержит вершину, непосредственно предшествующую вершине v в кратчайшем пути. Вначале $P[v]=1$ для всех $v \neq 1$. В листинге алгоритма Дейкстры при выполнении условного оператора $D[v]+C[v,i]<D[i]$, элементу $P[i]$ присваивается значение v. После выполнения алгоритма кратчайший путь к каждой вершине можно найти с помощью обратного прохождения по предшествующим вершинам массива P.

Для рассмотренного орграфа массив P имеет следующие значения: $P[2]=1$, $P[3]=4$, $P[4]=1$, $P[5]=3$. Для определения кратчайшего пути, например, от вершины 1 к вершине 5, надо отследить в обратном порядке предшествующие

вершины, начиная с вершины 5. Таким образом, кратчайший путь из вершины 1 в вершину 5 составляет последовательность вершин: 1, 4, 3, 5.

7.4. Нахождение кратчайших путей между парами вершин

Пусть дан орграф *G=(V, E)*, и необходимо определить кратчайшие пути между всеми парами вершин орграфа. Каждой дуге $v \to w$ этого графа сопоставлена неотрицательная стоимость *C[v, w]*. Общая задача нахождения кратчайших путей заключается в нахождении для каждой упорядоченной пары вершин (*v, w)* любого пути от вершины *v* в вершину *w*, длина которого минимальна среди всех возможных путей от *v* к *w*.

Для решения поставленной задачи применим алгоритм Флойда. Для этого пронумеруем вершины графа последовательно от 1 до *n*.

Алгоритм Флойда использует матрицу *A* размера $n \times n$, в которой вычисляются длины кратчайших путей. Вначале *A[i, j]= C[i, j]* для всех $i \neq j$. Если дуга $i \to j$ отсутствует, то *C[i, j]*=∞. Каждый диагональный элемент матрицы *A* равен 0.

Над матрицей *A* выполняется *n* итераций. После *к*-ой итерации *A*[*i, j*] содержит значение наименьшей длины путей из вершины *i* в вершину *j*, которые не проходят через вершины с номером, большим *к*, т. е. между концевыми вершинами пути из *i* в *j* могут находиться только вершины, номера которых меньше или равны *к*. На *к*-ой итерации для вычисления матрицы A применяется следующая формула:

$$A_k[i,j] = \min(A_{k-1}[i,j], A_{k-1}[i,k] + A_{k-1}[k,j]).$$

Нижний индекс *к* обозначает значение матрицы *A* после *к*-ой итерации. Графическая интерпретация приведенной формулы показана на рис. 7.6.

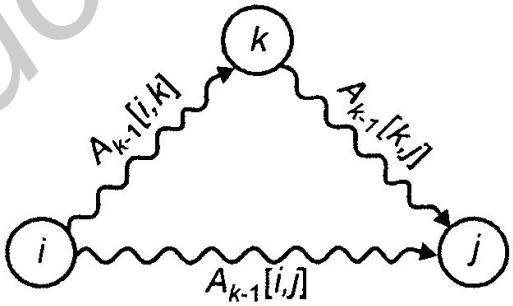


Рис. 7.6. Включение вершины *к* в путь от вершины *i* к вершине *j*

Для вычисления $A_k[i,j]$ проводится сравнение величины $A_{k-1}[i,j]$ с величиной $A_{k-1}[i,k] + A_{k-1}[k,j]$. Если путь через вершину *к* «дешевле», чем $A_{k-1}[i,j]$, то величина $A_k[i,j]$ изменяется.

На рис. 7.7 приведен помеченный орграф, а на рис. 7.8 – значения матрицы *A* после трех итераций.

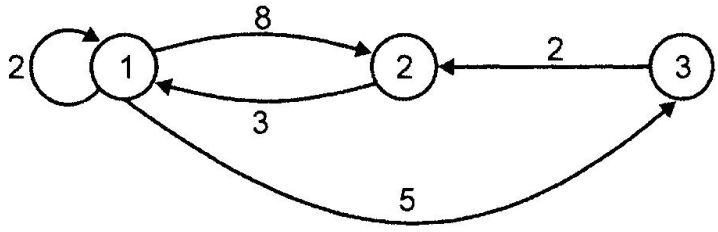


Рис. 7.7. Помеченный орграф

$A_0[i, j]$

	1	2	3
1	0	8	5
2	3	0	∞
3	∞	2	0

$A_1[i, j]$

	1	2	3
1	0	8	5
2	3	0	8
3	∞	2	0

$A_2[i, j]$

	1	2	3
1	0	8	5
2	3	0	8
3	5	2	0

$A_3[i, j]$

	1	2	3
1	0	7	5
2	3	0	8
3	5	2	0

Рис. 7.8. Последовательные значения матрицы *A*

Равенства $A_k[i,k] = A_{k-1}[i,k]$ и $A_k[k,j] = A_{k-1}[k,j]$ означают, что на *к*-ой итерации элементы матрицы *A*, стоящие в *к*-ой строке и *к*-ом столбце, не изменяются. Программа, реализующая алгоритм Флойда, представлена ниже.

```
Program Floid;
var  n,i,j,k : integer;
     C : array[1..10, 1..10] of integer; {Массив цен}
     A : array[1..10, 1..10] of integer;
begin
   writeln('Введите количество вершин графа');
   readln(n);
  writeln('Введите матрицу цен');
       For i:=1 to n do
         For j:=1 to n do
          readln(C[i,j]);
              For i:=1 to n do
                 For j:=1 to n do
                   A[i,j]:=C[i,j];
       For k:=1 to n do
         For i:=1 to n do
           For j:=1 to n do
             if A[i,k]+ A[k,j]< A[i,j] then
            A[i,j]:=  A[i,k]+ A[k,j];
      end.
```

Время выполнения этой программы имеет порядок n^3. Поскольку алгоритм Дейкстры с использованием матрицы смежности находит кратчайшие пути от одной вершины за время порядка n^2, то в случае применения алгоритма Дейкстры для нахождения всех кратчайших путей потребуется время порядка n^3, т. е. получается такой же временной порядок, как и в алгоритме Флойда. Если e, количество дуг в орграфе, значительно меньше, чем n^2, то рекомендуют применять алгоритм Дейкстры со списками смежности. Тогда время нахождения кратчайших путей имеет порядок $ne * \log_2 n$, что значительно лучше алгоритма Флойда, хотя бы для больших разреженных графов.

7.5. Нахождение центра ориентированного графа

Определим понятие *центральной вершины* орграфа. Пусть v – произвольная вершина орграфа $G=(V, E)$. Эксцентриситет (максимальное удаление) вершины v определяется как

max {минимальная длина пути от вершины w до вершины v }.

Центром орграфа G называется вершина с минимальным эксцентриситетом, т. е. это вершина, для которой максимальное расстояние (длина пути) до других вершин минимально.

Рассмотрим помеченный орграф, показанный на рис. 7.9.

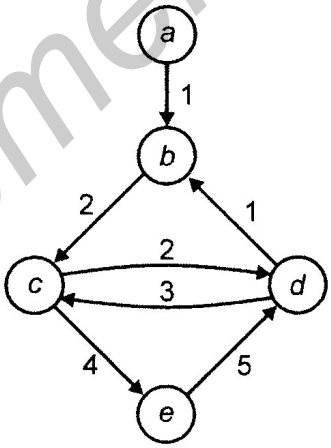


Рис. 7.9. Помеченный орграф

В этом графе вершины имеют следующие эксцентриситеты.

Вершина	Эксцеитриситет
a	∞
b	6
c	8
d	5
e	7

Откуда видно, что центром данного орграфа является вершина *d*.

Пусть *C* – матрица стоимостей для орграфа *G*. Тогда центр орграфа можно найти, применив следующий алгоритм:

1. Применить алгоритм Флойда к матрице *C* для вычисления матрицы *A*, содержащей все кратчайшие пути орграфа *G*;

2. Найти максимальное значение в каждом столбце *i* матрицы *A*. Это значение равно эксцентриситету вершины *i*;

3. Найти вершину с минимальным эксцентриситетом. Она и будет центром графа *G*.

Матрица всех кратчайших путей для орграфа из рис. 7.9 представлена на рис. 7.10. Максимальные значения в каждом столбце приведены под матрицей.

	a	*b*	*c*	*d*	*e*
a	0	1	3	5	7
b	∞	0	2	4	6
c	∞	3	0	2	4
d	∞	1	3	0	7
e	∞	6	8	5	0
max	∞	6	8	5	7

Рис. 7.10. Матрица кратчайших путей

7.6. Лабораторная работа №7

Цель работы: научиться представлять и анализировать ориентированные графы, определять в них центр и находить заданные маршруты.

Порядок выполнения работы

1. Ознакомиться с теоретической частью лабораторной работы.
2. Выполнить практическое задание.
3. Оформить отчет по лабораторной работе.

Задание. Представить помеченный орграф с помощью матрицы смежности или списков смежности. Указать вершину-источник и результирующую вершину на графе, а затем найти:

– кратчайший путь между заданными вершинами;
– самый длинный путь между заданными вершинами;
– все пути между заданными вершинами, упорядочив их по возрастанию;
– центр орграфа.

ЗАКЛЮЧЕНИЕ

Цель учебно-методического пособия – сформировать представления о многообразии компьютерных структур данных, способах описания объектов и алгоритмизации процессов в различных предметных областях, а также влиянии выбранных структур данных на функции обработки и эффективность программных средств.

В результате изучения курса «Структуры и алгоритмы обработки данных» с помощью предложенного пособия студенты смогут получить теоретические сведения и практические навыки по выбору и разработке структур данных для представления конкретных объектов разработки и преобразования их из одной формы в другую.

Пособие разработано согласно рабочей учебной программе дисциплины «Структуры и алгоритмы обработки данных» и содержит материал, относящийся к выполнению лабораторных работ. Пособие опирается на понятия, рассматриваемые в курсах «Основы алгоритмизации и программирования», «Основы вычислительной техники». В свою очередь, работа создает теоретическую и практическую базу для таких дисциплин, как «Методы и алгоритмы принятия решений», «Базы данных, знаний и экспертные системы», «Объектно-ориентированное программирование», «Информационное обеспечение финансовых структур», «Технологии разработки программного обеспечения».

ЛИТЕРАТУРА

1. Лэнгсам, Й. Структуры данных для персональных ЭВМ / Й. Лэнгсам, М. Огенстайн, А. Тененбаум. – М. : Мир, 1989.

2. Кинг, Д. Создание эффективного ПО / Д. Кинг. – М. : Мир, 1991.

3. Кнут, Д. Э. Искусство программирования. Т. 1. / Д. Э. Кнут. – 3-е изд. – М. : Изд. Дом «Вильямс», 2001.

4. Ахо, А. В. Структуры данных и алгоритмы / А. В. Ахо. – М. : Изд. Дом «Вильямс», 2000.

5. Вирт, Н. Алгоритмы + Структуры данных = Программы / Н. Вирт. – М. : Изд. Дом «Вильямс», 2002.

6. Кормен, М. Алгоритмы, построение и анализ / М. Кормен. – 2-е изд.– М. : Мир, 2005.

7. Седжвик, Р. Фундаментальные алгоритмы на С++ / Р. Седжвик. – М. : Изд. Дом «Вильямс», 2001.

Св. план 2013, поз. 30

Учебное издание

Серебряная Лия Валентиновна
Марина Ирина Михайловна

СТРУКТУРЫ И АЛГОРИТМЫ ОБРАБОТКИ ДАННЫХ

УЧЕБНО-МЕТОДИЧЕСКОЕ ПОСОБИЕ

Редактор *И. В. Ничипор*
Корректор *Е. И. Герман*
Компьютерная правка, оригинал-макет *В. М. Задоля*

Подписано в печать 25.04.2013. Формат 60x84 1/16. Бумага офсетная. Гарнитура «Таймс».
Отпечатано на ризографе. Усл. печ. л. 3,14. Уч.-изд. л. 3,0. Тираж 100 экз. Заказ 634.

Издатель и полиграфическое исполнение: учреждение образования
«Белорусский государственный университет информатики и радиоэлектроники»
ЛИ №02330/0494371 от 16.03.2009. ЛП №02330/0494175 от 03.04.2009.
220013, Минск, П. Бровки, 6