МИНИСТЕРСТВО ОБРАЗОВАНИЯ РЕСПУБЛИКИ БЕЛАРУСЬ

УЧРЕЖДЕНИЕ ОБРАЗОВАНИЯ
"БРЕСТСКИЙ ГОСУДАРСТВЕННЫЙ ТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ"

Кафедра «ЭВМ и системы»

О.В. МОРОЗ

АБСТРАКТНЫЕ ТИПЫ ДАННЫХ: РЕАЛИЗАЦИЯ НА ЯЗЫКЕ С

Рекомендовано Советом Брестского государственного технического университета в качестве пособия для студентов специальностей 1-40 02 01 «Вычислительные машины, системы и сети» и 1-36 04 02 «Промышленная электроника»

Брест 2005

УДК 004.42(07)
ББК 32.973-01р30
М80

Рецензенты:
зав. кафедрой информатики и прикладной математики БрГУ им. А.С. Пушкина, к. физ.-мат. наук, доцент **Мадорский В.М.**;
начальник управления координации деятельности ВНЗ и науки Львовской госадминистрации, к. физ.-мат. наук, доцент **Сташенко М.А.**

О.В. Мороз
М80 Абстрактные типы данных: реализация на языке С. Пособие. Брест: издательство БГТУ, 2005. - 75 с.

ISBN 985-493-022-X

В пособии рассмотрены основные типы структур данных и приемы реализации на языке С соответствующих абстрактных типов данных.

Пособие предназначено для студентов 1- и 2-го курсов специальностей I - 40 02 01 «Вычислительные машины, системы и сети» и I - 36 04 02 «Промышленная электроника», а также для студентов, освоивших вводный курс программирования на языке С и желающих расширить и углубить свои знания.

УДК 004.42(07)
ББК 32.973-01р30

ISBN 985-493-022-X

ОГЛАВЛЕНИЕ

СТРУКТУРЫ ДАННЫХ И АБСТРАКТНЫЕ ТИПЫ ДАННЫХ 4
1. СПИСКОВЫЕ СТРУКТУРЫ 7
1.1. АТД «список» 7
1.2. Линейные списки 8
1.3. Очереди и стеки 12
1.4. Иерархические списки 13
1.5. Ассоциативные списки 15
УПРАЖНЕНИЯ 16
2. ДЕРЕВЬЯ 18
2.1. АТД «дерево» 18
2.2. Деревья выражений 24
2.2.1. Создание дерева выражения 26
2.2.2. Обход дерева выражения 31
2.3. Двоичные деревья 33
2.4. Деревья двоичного поиска 36
2.5. Частично упорядоченные деревья 40
УПРАЖНЕНИЯ 42
3. МНОЖЕСТВА 44
3.1. Основные понятия теории множеств 44
3.2. АТД «множество» 46
3.3. Представление множеств посредством двоичных векторов 47
3.4. Представление множеств посредством перемешанных таблиц 50
3.4.1. Прямое хеширование 50
3.4.2. Расширенное хеширование 57
3.5. Представление множеств посредством рекурсивных структур 58
3.5.1. 2-3-деревья 58
3.5.2. B-деревья 68
3.5.3. Нагруженные деревья 70
УПРАЖНЕНИЯ 73
ЛИТЕРАТУРА 74

СТРУКТУРЫ ДАННЫХ И АБСТРАКТНЫЕ ТИПЫ ДАННЫХ

В основе современных информационных технологий лежат методы формализации знаний, относящихся к той или иной предметной области. Формализованные знания включают выявленные в результате анализа предметной области совокупности данных, под которыми понимаются предметы, факты, явления, идеи, события, процессы и т.п. и связи между ними (понятийные знания), а также описание алгоритмов выполняемых над данными действий (алгоритмические знания). В дальнейшем при построении информационных моделей среди множества взаимосвязей данных отбираются наиболее информативные, наиболее полно отражающие состояние предметной области. В конечном итоге получается упрощенная модель предметной области, в которой более простые (элементарные) данные объединяются по определенным признакам в более сложные и определяется набор выполняемых над ними действий. В программировании для представления подобных объединений данных используются структуры данных.

Структуры данных – это сложные данные, построенные из более простых или элементарных данных (возможно различных типов), объединенных с помощью определенных языковых конструкций.

Элементарные данные (элементарный объект данных) – логически неделимая поименованная порция данных.

Различают функциональные, рекурсивные и теоретико-множествен-ные структуры данных.

Функциональные структуры данных

Функциональную структуру данных (ФСД) математически можно определить как отображение (соответствие) из некоторого множества A, называемого областью расположения структуры данных, в другое множество B, называемое областью значений структуры данных. Другими словами ФСД – это множество упорядоченных пар (x, y), где x принадлежит множеству A, а y – множеству B.

Типичным примером ФСД является **многомерный массив** (регулярный тип данных) – структура данных, расположенная на целочисленной решетке Z^m размерности m и принимающая значения в диапазоне некоторого скалярного или структурного типов данных.

К ФСД относится также такая структура как **запись** (комбинированный тип данных). Областью расположения записи является упорядоченное множество полей с определенным для каждого из них типом данных T_i (i=1, 2,..., n), а областью значений – множество упорядоченных наборов (кортежей) вида $(a_1, a_2,..., a_n)$, где a_i относится к типу T_i и порядок следования значений в каждом наборе соответствует порядку следования полей в записи.

Исходя из существующей определенной аналогии между массивом и файлом, файлы также можно классифицировать как ФСД. В общем случае **файл** как совокупность записей, имеющих общую область использования, следует отнести к многоуровневым ФСД, т.е. ФСД, элементами которых являются ФСД, так же, как и массив массивов, массив записей или запись с полями регулярного или комбинированного типа.

Рекурсивные структуры данных

Рекурсивные структуры данных (РСД) определяются как структуры данных, в основе представления которых лежит понятие графа.

Граф – математическая модель связей между объектами произвольной природы, представляющая собой набор точек на плоскости (вершины графа), которые могут соединяться направленными или ненаправленными линиями (ребра графа). Если ребра графа являются направленными, т.е. для каждой вершины можно указать, какие ребра в нее входят и какие выходят, то граф называется ориентированным, в противном случае – неориентированным. Две вершины графа соединены путем, если из одной вершины можно попасть в другую, пройдя по ребрам. Граф называется связным, если любые две его вершины соединены некоторым путем. Состоящий из различных ребер замкнутый путь называется циклом. Ребро, соединяющее вершину с ней самой, называется петлей.

К РСД относятся различные списковые структуры, деревья, сетевые и фреймовые структуры и т.д.

Список можно определить как ориентированный или неориентированный граф, в любую вершину которого входит одно или два ребра.

Списковая структура данных – иерархическая система хранения данных в памяти ЭВМ, в которой данные рассматриваются как списки, элементы которых в свою очередь также могут быть списками (такие элементы называются подсписками) и т.д. Элементы списков и подсписков могут располагаться в памяти произвольно, но каждый из элементов содержит указатель на место расположения следующего и/или предыдущего элемента.

Дерево в теории графов – ориентированный или неориентированный граф без циклов с выделенной вершиной (корень дерева), из которой существует путь в любую другую вершину, причем этот путь – единственный. В каждую вершину дерева входит только одно ребро, а выходить может несколько. Различные виды деревьев широко используются в теории алгоритмов, теории оптимизации, математической статистике, теории надежности, теории распознавания образов и т.д.

Сетевая структура данных – структура, которая может быть представлена ориентированным графом, в любую вершину которого может входить более одного ребра. Сетевые структуры используются в различных приложениях для представления n-арных связей между объектами, т.е. когда данным одного типа ставится в соответствие множество данных другого типа и наоборот. Сетевая модель данных представляет собой наиболее универсальную модель структурирования фактографических данных.

Фреймовая структура данных (от англ. frame – каркас, рамка) – структура, используемая для представления знаний в ЭВМ. Фрейм можно представить в виде графа, вершины которого распределены по уровням. Верхний уровень фрейма фиксирован и содержит знания, всегда истинные для ситуаций, которые представляет данный фрейм. На нижележащих уровнях расположены так называемые слоты, которые могут заполняться подходящими данными в процессе «приспособления» фрейма к конкретной ситуации. Родственные фреймы могут связываться в систему фреймов, системы фреймов – в информационно-поисковую сеть.

Теоретико-множественные структуры данных

Теоретико-множественные структуры данных (ТМСД) – наиболее общий тип структур данных, основанный на математическом понятии множества. Понятие множества относится к фундаментальным понятиям математики, поэтому строго математического определения этого понятия не существует. В математике под множеством понимается любое объединение в одно целое некоторых определенных и различных между собой объектов, находящихся в определенных отношениях между собой или с элементами других множеств.

Различные ТМСД можно получить путем применения теоретико-множественных конструкций к уже построенным типам, включая ФСД и РСД, причем к теоретико-множественным конструкциям относятся не только специфические для этого типа структур данных – декартово произведение, множество подмножеств и др., но и конструкции, применимые для организации ФСД и РСД.

Абстрактные типы данных

В процедурных языках программирования, таких как С, достаточно широко представлены встроенные структурные типы данных, относящиеся к ФСД, при этом допустимые структуры можно представить как композиции нескольких порождающих типов структур: элемент, массив, запись и файл. Построение структур данных других типов сопряжено с разработкой соответствующих абстрактных типов данных.

Абстрактный тип данных (АТД) – тип данных, рассматриваемый независимо от способов его представления или реализации средствами языка программирования и представляющий собой математическую модель взаимосвязи объектов некоторой предметной области с определенными для них основными операциями.

АТД представляют собой обобщение встроенных типов данных в рамках концепции скрытия информации в программировании в том смысле, что отдельные категории данных совместно с процедурами их преобразования определяются в виде некоторых абстракций, допускающих их использование без знания подробностей организации данных и алгоритмов выполняемых над ними операций.

Решение любой прикладной задачи в программировании с применением того или иного АТД начинается с выбора соответствующей математической модели такой, например, как отображение, граф или множество. Далее разрабатывается АТД с подходящим набором операций, и алгоритм решения задачи описывается в терминах этого АТД. Как правило, такое описание выполняется на неформальном языке (псевдоязыке), представляющем собой смесь фраз естественного языка и базовых управляющих структур того языка программирования, на котором в конечном итоге будет представлен разрабатываемый алгоритм.

Следующий этап – реализация алгоритма на конкретном языке программирования, т.е. переход от абстрактно-логического представления алгоритма в терминах разработанного АТД к представлению в терминах типов данных, операторов и структур, допустимых в данном языке программирования. Для этого необходимо, во-первых, отобразить элементы АТД на одну из синтаксических конструкций языка и, во-вторых, реализовать на этом языке разработанные для данного АТД операции.

Для представления элементов АТД, как правило, используются структурные типы данных, а объединение элементов АТД в структуры производится путем создания поименованных групп элементов или с помощью ссылок, реализуемых посредством указателей или курсоров, когда каждый элемент АТД содержит не только определенную порцию данных, но и указание на место расположения связанных с ним элементов.

В рамках одной и той же математической модели можно рассматривать, очевидно, разные операции, поэтому любой АТД фактически определяется своим набором операций. Выбор тех или иных операций будет зависеть от поставленной задачи и способа структурирования элементов АТД, хотя в большинстве случаев можно говорить о типовых операциях, включающих операции поиска, вставки и удаления элементов.

1. СПИСКОВЫЕ СТРУКТУРЫ

1.1. АТД «список»

Список как структура данных представляет собой набор однотипных элементов, связанных друг с другом посредством ссылок.

По количеству ссылок у элементов списки делятся на **односвязные** (одна ссылка – на предыдущий или последующий элемент) и **двусвязные** (две ссылки – на предыдущий и последующий элементы). И те и другие могут быть **линейными** или **кольцевыми**. В линейном списке первый и последний элементы не связаны друг с другом, а в кольцевом списке первый элемент может ссылаться на последний и/или наоборот. В практике программирования наиболее часто используются **двусвязные кольцевые списки**, в которых последний элемент ссылается на первый как на последующий, а первый элемент – на последний как на предыдущий.

Списки, как и ФСД, можно объединять, образуя более сложные структуры того же типа, т.е. можно создавать многоуровневые структуры данных типа списка списков. Такие объединения списков имеют иерархическую структуру и поэтому называются **иерархическими списками**.

Иерархические списки применимы в тех случаях, когда взаимосвязи между реальными объектами имеют иерархическую структуру, а именно: для решения задач структурного анализа сложных систем, в частности для описания организационной структуры социальных систем, для разработки различного рода каталогов, справочных систем и т.п.

Списковые структуры представляют собой модели взаимосвязей объектов реальной среды, объединяемых по наличию некоторого общего для всех объектов свойства. Обладая множеством различных свойств, такие объекты могут одновременно входить в несколько объединений. Если представлять объекты в виде элементов списков, то это означает, что одни и те же элементы могут включаться в несколько списков одновременно. Например, на базе одного и того же множества записей, содержащих анкетные данные граждан, вставших на учет в службе занятости с целью поиска работы, можно сформировать несколько списков: по профессии, по уровню квалификации, по стажу работы в той или иной отрасли, по районам города, которым отдается предпочтение при выборе места работы, и т.д.

Избавиться от неизбежного в таких случаях избыточного дублирования данных позволяют **ассоциативные списки**. Совокупность списков, создаваемых на базе одного общего набора записей, называется **информационной сетью** и представляет собой наиболее общую форму отображения множества взаимосвязей реальных объектов.

Для представления элементов АТД «список» используются записи, в общем случае состоящие из обязательных информационной и ссылочной частей и необязательной справочной части. В информационной части содержатся поля, значения которых описывают определенные свойства элементов списка. Среди этих полей, как правило, выделяется одно или несколько полей, которые однозначно идентифицируют данный элемент. Такие поля называются ключевыми или просто – ключом. В ссылочной части указываются адреса следующего и/или предыдущего элемента списка. Справочная часть предназначена для хранения дополнительной информации об элементах списка.

В начале списка обычно располагается особый головной элемент, содержащий сведения о списке в целом, например, количество элементов, минимальное и максимальное значения ключа и т.д., и, по крайней мере, ссылку на первый элемент списка.

К основным операциям над списками относятся:

- поиск заданного элемента списка;
- включение нового элемента в список;
- исключение элемента из списка.

1.2. Линейные списки

Для определенности рассмотрим реализацию линейного односвязного списка, в котором ссылки направлены таким образом, что каждый элемент, кроме последнего, ссылается на последующий. Пусть также информационная часть элементов списка состоит из ключевого поля, тип которого определен как KEY и относится к одному из перечисленных типов, и некоторого поля, тип которого определен как VAL. Опуская справочную часть с целью сокращения объема листингов, тип элементов списка можно определить следующим образом:

```
typedef struct item
{ KEY key;          /* Ключевое поле. */
  VAL val;          /* Поле значения элемента. */
  struct item *link; /* Ссылочная часть. */
} ITEM;
```

В структуре головного элемента списка определим три поля: одно целочисленное для указания количества элементов в списке в текущий момент времени и два ссылочных поля – для ссылки на первый и последний элементы списка:

```
typedef struct
{ unsigned num;     /* Количество элементов в списке. */
  struct ITEM *first; /* Ссылка на первый элемент. */
  struct ITEM *last; /* Ссылка на последний элемент. */
} HITEM;
```

Теперь рассмотрим функции, реализующие операции над элементами списка описанной выше структуры:

1. Создание пустого списка.

Пустой список – это список, не содержащий никаких элементов, кроме головного. Таким образом, чтобы создать пустой список, надо создать его головной элемент с нулевыми ссылками (Листинг 1.1).

Листинг 1.1

```
/* Создание пустого списка.
   Параметры: нет.
   Возвращаемое значение: при успешном завершении – указатель на созданный головной элемент
   списка, иначе – NULL. */
HITEM * EmptyList()
{ HITEM * head = NULL;
  if (head = malloc(sizeof(HITEM)) != NULL)
      { head->num = 0;
        head->first = head->last = NULL; }
  return head;
}
```

2. Включение нового элемента в список.

Если список пуст, включение нового элемента сводится к определению соответствующих ссылок головного элемента, причем обе они будут иметь одно и то же значение. В противном случае сначала необходимо найти место в списке для нового элемента, т.е. найти элемент, после или перед которым необходимо вставить новый элемент. В общем случае расположение элементов в списке может быть произвольным, и тогда, чтобы вставить новый элемент в список, пользователь должен указать ключ того элемента, который будет предшествовать в списке новому элементу. Если же элементы списка упорядочены в порядке убывания или возрастания их ключей, каждый новый элемент займет вполне определенное место, а поиск элементов в таком списке будет выполняться несколько быстрее.

После того, как место для нового элемента определено, необходимо вставить его в список. Эта операция выполняется по схеме, изображенной на рис. 1.1. Здесь ключ нового элемента обозначен как NEW (новый), ключ предшествующего ему элемента обозначен как PRE, а ключ последующего элемента – как SUC. С помощью вспомогательного указателя buf создается новый элемент, в ссылочной части которого записывается адрес элемента SUC, а ссылка элемента PRE перенаправляется на вновь созданный элемент.

Если новый элемент должен стать первым элементом списка, в роли элемента PRE будет выступать головной элемент. Если новый элемент добавляется в конец списка, его ссылочная часть устанавливается в NULL-значение, поскольку элемент SUC отсутствует. И в том и в другом случае должны быть переопределены ссылочные поля головного элемента.

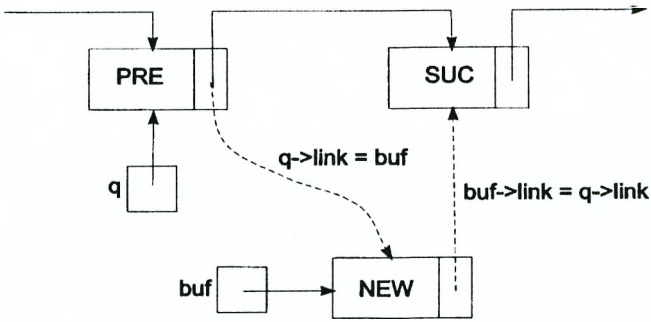


Рис. 1.1. Схема вставки элемента в линейный односвязный список.

В Листинге 1.2 представлен исходный код функции, реализующей операцию включения элемента в список, элементы которого располагаются в порядке возрастания их ключей.

Листинг 1.2

```
/* Включение элемента в упорядоченный список.
   Параметры: h – указатель на головной элемент; new_key, new_val – ключ и значение нового элемента;
   setval – указатель на функцию, которая записывает значение типа VAL по указанному адресу.
   Возвращаемое значение: при успешном завершении – указатель на новый элемент списка, иначе –
   NULL. */
ITEM * add_item(HITEM *h,
                KEY new_key,
                VAL new_val,
                void (*setval)(VAL *, VAL))
{ ITEM * q = h->first, * buf;
  if ((buf = malloc(sizeof(ITEM)) == NULL) return NULL;
  buf->key = new_key;
  (*setval)(&buf->val, new_val);
  if (h->num == 0)            /* Если список пустой. */
     { h->first = h->last = buf;
       buf->link = NULL; }
  else          /* Если список не пустой */
       if (q->key > new_key)                /* Вставка в начало списка. */
          { buf->link = q;
            h->first = buf; }
```

```
    else
        if (h->last->key < new_key)     /* Вставка в конец списка. */
            { buf->link = NULL;
              h->last->link = buf;
              h->last = buf; }
        else
            { /* Поиск места для нового элемента. */
              while (q->link->key < new_key)
                  if (q->key == new_key) return NULL;
                  else q=q->link;
              buf->link = q->link;
              q->link = buf;
            }
  h->num++;
  return buf;
}
```

3. Поиск элемента в списке.

Если список упорядочен, то для поиска элемента по заданному значению ключа нет необходимости просматривать все элементы списка. Во-первых, следует проверить, попадает ли искомое значение ключа в диапазон значений, ограниченный значениями ключей первого и последнего элементов. Во-вторых, просмотр элементов продолжается до тех пор, пока не встретится элемент, значение ключа которого больше/меньше чем искомое значение при упорядоченности элементов по возрастанию/убыванию. Реализация операции поиска для упорядоченного списка приведена в Листинге 1.3.

Листинг 2.3

```
/* Поиск элемента в упорядоченном списке.
   Параметры: h – указатель на головной элемент; search_key – ключ искомого элемента.
   Возвращаемое значение: при успешном завершении – указатель на элемент списка, иначе – NULL.
   */
ITEM * search_item(HITEM *h, KEY search_key)
{ ITEM *q=h->first;
  if ((q == NULL) || (q->key > search_key) || (h->last->key < search_key)) return NULL;   /*
      Элемента нет в списке. */
  do { if (q->key == search_key) return q;
       q =q->link; }
  while ((q != NULL) && (q->key < search_key));
  return NULL;
}
```

Поиск в неупорядоченном списке, как и поиск по значениям не-ключевых полей элементов списка, которые в общем случае не являются уникальными для всего списка, возможен лишь путем полного перебора элементов списка.

4. Исключение элемента из списка.

Для исключения некоторого элемента из списка необходимо в ссылочной части элемента, предшествующего удаляемому, указать адрес элемента, следующего за удаляемым, освободив при этом блок памяти, занимаемый удаляемым элементом (рис. 1.2).

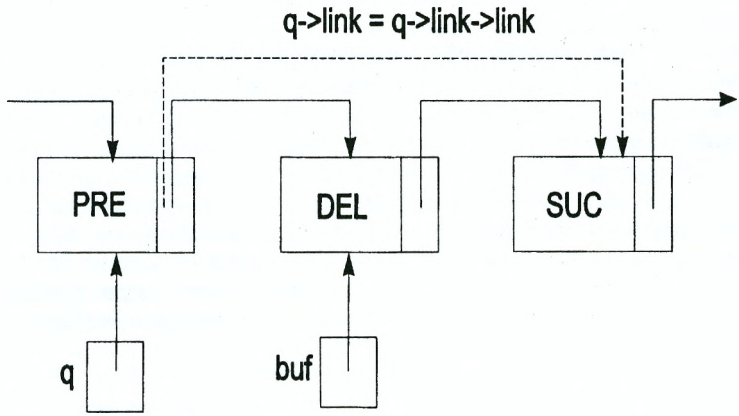


Рис. 1.2. Схема удаления элемента из линейного односвязного списка.

Исходный код функции, реализующей удаление элемента из упорядоченного списка, приведен в Листинге 1.4.

Листинг 1.4

```
/* Удаление элемента из упорядоченного списка.
   Параметры: h – указатель на головной элемент; del_key – ключ удаляемого элемента.
   Возвращаемое значение: при успешном завершении – 1, иначе – 0. */
int del_item(HITEM *h, KEY del_key))
{ ITEM *q, *p;
  q = p = h->first;
  /* Если элемента нет в списке. */
  if ((q == NULL) || (q->key > del_key) || (h->last->key < del_key)) return 0;
  while (q->key != del_key);
      { q = q->link;
        if ((q == NULL) || (q->key > del_key)) return 0;
        p = q; }
  if (p == q)   /* Если удаляемый элемент – первый. */
      h->first = q->link;
  else
      if ((p->link = q->link) == NULL) /* Если удаляемый элемент – последний. */
          h->last = p;
  free(q);
  return 1;
}
```

5. Выборка элементов списка.

Под выборкой понимают поиск и вывод на некоторое устройство вывода тех элементов списка (или другой какой-либо структуры данных), которые удовлетворяют заданному условию. Например, можно произвести выборку элементов, для которых значение

одного из полей информационной части (или нескольких полей) лежит в заданном диапазоне или равно некоторой величине.

В условии отбора элементов могут фигурировать как ключевые поля, так и не ключевые, поэтому в общем случае для реализации операции выборки необходимо пройти по списку от первого элемента до последнего и вывести значения соответствующих элементов. В однонаправленном списке такой проход возможен лишь в прямом порядке, т.е. в направлении ссылок. Для прохода по элементам списка в обратном порядке можно предварительно загрузить их во вспомогательный буфер (файл) или использовать рекурсию. Например, функция, исходный код которой представлен в Листинге 1.5, выводит ключи элементов списка, упорядоченного по возрастанию значений ключей, в порядке их убывания. Для вызова этой функции необходимо указать адрес первого элемента списка и адрес функции, которая производит вывод значения типа KEY и возвращает 1 при успешном завершении или 0 – в противном случае.

Листинг 1.5

```
/* Выборка элементов упорядоченного списка в обратном порядке.
   Параметры: q – указатель на первый элемент; outkey – указатель на функцию вывода ключевого
   поля.
   Возвращаемое значение: при успешном завершении – 1, иначе – 0. */
int select_item(ITEM *q, int (*outkey)(KEY))
{ ITEM * p;
  if (q == NULL) return 0;
  if ((p = q->link) != NULL)
      if (select_item(p, outkey) == 0) return 0;
  return (*outkey)(q->key);
}
```

Реализация основных операций для двусвязных списков аналогична. Преимуществом двусвязного списка является то, что в таком списке имеется возможность прохода по списку в обоих направлениях ссылок, поэтому в практике программирования чаще используются двунаправленные кольцевые списки.

1.3. Очереди и стеки

Частными случаями линейного односвязного списка являются очередь, стек и дек.

Очередь – это список, в котором новый элемент добавляется в конец списка, а удаляется всегда только первый элемент. Таким образом, очередь функционирует по принципу «первым пришел – первым ушел» (FIFO – First Input First Output).

Необходимость в такой структуре возникает при моделировании различных систем обслуживания, которые выполняют заказы последовательно, один за другим. Если при поступлении заказа обслуживающее устройство свободно, заказ выполняется немедленно, в противном случае заказ ставится в конец очереди заказов, ожидающих выполнения. Каждый раз на выполнение поступает заказ, который стоит первым в очереди заказов, а после его удаления первым в очереди становится следующий за ним заказ. Если заказы поступают нерегулярно, длина очереди будет изменяться во времени.

Над элементами очереди обычно выполняются операции двух видов:

- выборка с одновременным удалением из очереди первого ее элемента;
- добавление нового элемента в конец очереди.

Одной из разновидностей очередей являются **очереди с приоритетами**. Очередь с приоритетами отличается от обычной очереди тем, что элементы имеют дополнительный параметр – приоритет, который и определяет порядок их удаления из очереди. Из очереди с приоритетами удаляется не первый элемент, а элемент, у которого в данный момент времени наивысший приоритет.

Стек (или магазин) представляет собой список, упорядоченный по времени поступления элементов таким образом, что извне доступен только последний из записанных в список элементов, который называется вершиной стека. Другими словами, стек функционирует по принципу «последним пришел – первым ушел» (LIFO – Last Input First Output).

Так как извне доступна только вершина стека, то основными операциями для стека являются:

- выборка с одновременным удалением вершины стека;
- добавление нового элемента в вершину стека.

Дек – структура, обладающая свойствами очереди и стека, т.е. очередь, в которой элементы можно добавлять как в конец, так и в начало.

1.4. Иерархические списки

Иерархический список представляет собой несколько подсписков, ранжированных по уровням, так что элементы подсписка уровня 0 являются головными элементами (или содержат ссылки на них) для подсписков уровня 1, те, в свою очередь, являются головными элементами для подсписков уровня 2 и т.д. (рис. 1.3).

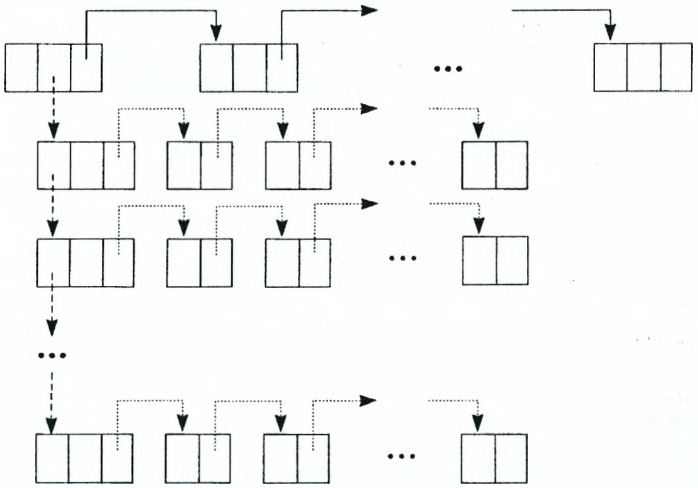


Рис. 1.3. Структура трехуровневого иерархического списка (сплошная линия – подсписок уровня 0, штриховая линия – подсписок уровня 1, пунктирная линия – подсписок уровня 2).

Таким образом, элементы иерархического списка, кроме справочной, информационной и ссылочной частей, соответствующих уровню данного подсписка, должны также содержать компоненты головного элемента подсписка вышележащего уровня (или ссылку на него).

При реализации АТД «иерархический список» особого внимания требуют две его характеристики: количество уровней и количество разных типов подсписков, расположен-

ных на одном и том же уровне. Если количество уровней велико и/или на каждом уровне могут располагаться несколько типов подсписков, это не только усложнит программный код, но и приведет к существенному увеличению времени выполнения основных операций. Однако нередко объекты, расположенные на одном уровне в иерархических моделях данных, относятся к одному и тому же типу, поэтому в таких случаях количество разных типов подсписков будет равно, очевидно, количеству уровней в иерархическом списке.

Поскольку в иерархическом списке в общем случае могут быть объединены несколько подсписков разных типов с разными типами элементов, в число полей головного элемента каждого подсписка необходимо ввести указатели на головные элементы нижележащих подсписков (у элементов данного подсписка может быть несколько типов подчиненных подсписков), а для реализации основных операций над элементами придется определить столько функций, сколько типов подсписков содержится в данном иерархическом списке.

Рассмотрим в качестве примера реализацию иерархического списка, изображенного на рис. 1.3, полагая, что все подсписки являются однонаправленными и что одноуровневые подсписки состоят из однотипных элементов. Указатели на функции поиска, вставки и удаления для каждого подсписка удобно включить в состав полей соответствующего головного элемента, тогда тип головного элемента подсписка уровня 0 и тип его элементов можно определить, например, так:

```
/* Тип головного элемента подсписка уровня 0 */
typedef struct
{ ITEM0 *first, *last,  /* Ссылки на первый и последний элементы подсписка. */
          (*srch)(HITEM0 *, KEY0), /* Указатель на функцию поиска. */
          (*add)(HITEM0 *, KEY0, VAL0); /* Указатель на функцию вставки. */
  int (* del)(HITEM0 *, KEY0);      /* Указатель на функцию удаления. */
} HITEM0;

/* Тип элементов подсписка уровня 0 */
typedef struct item
{ KEY0 key;   /* Ключевое поле. */
  VAL0 val;          /* Поле значения. */
  struct item *suc;  /* Ссылка на последующий элемент. */
  ITEM1 *sub;        /* Ссылка на головной элемент подсписка. */
} ITEM0;
```

Аналогично определяются типы HITEM1, ITEM1 и HITEM2, ITEM2 для подсписков уровней 1 и 2 с учетом того, что элементы последнего не будут иметь подсписков.

Для реализации операций вставки и удаления элементов в иерархическом списке можно использовать те же функции, что и для простых списков, но для поиска элемента уже недостаточно указать значение его ключевого поля – необходимо сначала найти подсписок, в котором находится данный элемент. Для этого надо знать путь к искомому элементу в виде последовательности значений ключей элементов, в которой начальное значение представляет собой ключ элемента подсписка уровня 0, а каждое последующее значение является ключом элемента из соответствующего нижележащего подсписка. Назовем такую последовательность ключей, включая ключ самого элемента, спецификацией элемента иерархического списка по аналогии со спецификацией файла в иерархической файловой подсистеме.

Количество значений, входящих в спецификацию элемента, зависит от того, на каком уровне расположен данный элемент, и варьируется от 1 для элементов подсписка уров-

ня 0 до величины, равной количеству уровней в иерархическом списке, для элементов подсписков, расположенных на последнем уровне. Поэтому функцию, реализующую операцию поиска элемента по его спецификации, определим как функцию с переменным числом параметров (Листинг 1.6), в качестве которых используются адреса входящих в спецификацию значений, упорядоченные по возрастанию номера уровня (0,1,2 и т.д.). При вызове функции список необязательных фактических параметров должен заканчиваться значением NULL.

Листинг 1.6

```
/* Поиск элемента в иерархическом списке.
   Параметры: h – указатель на головной элемент подсписка уровня 0; список необязательных параметров, заканчивающийся значением NULL.
   Возвращаемое значение: при успешном завершении – указатель на головной элемент подсписка, в котором содержится искомый элемент, иначе – NULL.
   Макроопределения va_start, va_arg, va_end и тип va_list определены в stdarg.h. */
void * search_item(HITEM0 * h, ...)
{ va_list ap, arg, p = h, q;
  int i = 0;
  va_start(ap, h);
  while ((arg = va_arg(ap, va_list)) != NULL)
       switch (i++)
       /* Количество меток case должно быть равно количеству уровней в иерархическом списке. */
           { case 0: { if ((q = (*h.srch)(h, * (KEY0 *) arg)) == NULL)
                              return NULL;
                       break; }
             case 1: { ITEM1 * h1 = ((ITEM1 *)q)->sub;
                       if (((p = h1) == NULL) ||
                           ((q = (*h1.srch)(h1, * (KEY1 *) arg)) == NULL))
                           return NULL;
                       break; }
             case 2: { ITEM2 * h2 = ((ITEM2 *)q)->sub;
                       if (((p = h2) == NULL) ||
                           ((*h2.srch)(h2, * (KEY2 *) arg)) == NULL))
                           return NULL;
                       break; }
             default: return NULL;          /* Слишком много параметров. */
           }
  va_end(ap);
  return p;
}
```

Функции, реализующие операции вставки и удаления элементов в иерархическом списке, читателю предлагается определить самостоятельно в качестве упражнения.

1.5. Ассоциативные списки

Информационная сеть, как было сказано выше, состоит из нескольких ассоциативных списков, которые организуются на основе одного общего или базового списка. Коль скоро одна и та же запись может входить в несколько списков одновременно, элементы базового списка должны содержать несколько ссылочных частей по количеству списков, в которые они могут включаться. Если количество ассоциативных списков фиксировано, то при реализации соответствующего абстрактного типа данных ссылочные части элементов базового

списка и головные элементы ассоциативных списков могут быть организованы в виде массивов. В противном случае следует обеспечить возможность изменения количества соответствующих ссылочных полей элементов базового списка, что возможно только в одном случае – если эти поля объединены в подсписок.

Таким образом, в общем случае информационная сеть может быть создана на базе двухуровневого иерархического списка, в котором в качестве подсписка первого уровня выступает базовый список, а подсписки второго уровня состоят из ссылочных полей ассоциативных списков. Головные элементы ассоциативных списков удобно представить также в виде элементов специально созданного для этого списка.

В отношении основных операций и их реализации ассоциативные списки ничем не отличаются от рассмотренных выше линейных или кольцевых списков, поэтому ниже приведены только примеры определения основных типов данных для информационной сети с переменным количеством ассоциативных списков:

```
/* Тип элементов базового списка. */
typedef struct item
{ KEY key;              /* Ключевое поле. */
  VAL val;              /* Поле значения. */
  struct item *pre, *suc;     /* Основные ссылочные поля. */
  PTR *psub; /*Указатель на первый элемент подсписка ссылочных полей ассоциативных списков.
  */
} BASE_ITEM;

/* Тип элементов подсписка ссылочных частей ассоциативных списков. */
typedef struct ptr
{ struct ptr *pre, *suc;
  PASS *head;       /* Ссылка на головной элемент соответствующего ассоциативного списка. */
} PTR;

/* Тип элементов списка головных элементов ассоциативных списков. */
typedef struct pass
{ struct pass *pre, *suc;
  BASE_ITEM *first, *last;   /* Ссылки на первый и последний элементы ассоциативного списка. */
} PASS;
```

УПРАЖНЕНИЯ

1. Реализовать АТД «очередь».
2. Реализовать АТД «стек».
3. Реализовать АТД «дек».
4. Реализовать АТД «двусвязный кольцевой список».
5. Дан символьный файл, в котором символы сгруппированы в слова, отделенные друг от друга знаками препинания и пробелами. Получить список слов, которые содержатся как подстроки в других словах.
6. Дана строка символов, содержащая арифметическое выражение, записанное с помощью латинских букв, цифр, знаков операций, круглых и прямоугольных скобок. Определить функцию, которая проверяет соответствие пар левых и правых скобок и в случае обнаружения несоответствия возвращает положение первой непарной скобки.
7. Дан текстовый файл, состоящий из N строк, длина которых не превышает M символов. Упорядочить строки в файле в лексикографическом порядке посредством создания кольцевого двусвязного списка с упорядоченными по возрастанию или убыванию значениями элементов.

8. Создать АТД «очередь с приоритетами», считая, что:
   а) приоритет определяется как случайное целое положительное число при включении элемента в очередь и в дальнейшем не изменяется;
   б) приоритет является функцией времени и вычисляется по формуле $P=(t - t_0)(C - a)$, где t – текущее время, t_0 – время поступления элемента в очередь, C – целая положительная константа, a – одно из полей информационной части элемента, значение которого заключено в диапазоне от 0 до C и определяется случайным образом при включении элемента в очередь.
9. Написать программу, имитирующую детскую считалку, суть которой заключается в том, что из N участников после N – 1 тура, в результате каждого из которых один из участников выбывает, выбирается «победитель». В каждом туре выбывающий определяется путем кругового перебора k (k = N, N-1, ..., 2) оставшихся участников, количество шагов в котором равно сумме k случайных натуральных чисел в диапазоне от 0 до 5. Направление отсчета, в сторону увеличения или уменьшения номеров участников, и участник, с которого начинается счет, выбираются случайно.
10. Написать программу, имитирующую работу некоторой системы массового обслуживания с фиксированным количеством обслуживающих станций. Поступивший заказ направляется на ту станцию, на которой количество заказов, стоящих в очереди наименьшее. Если таких станций несколько, выбирается первая из них. Частота поступления заказов является функцией времени f(t), а время их выполнения выбирается случайным образом, но не превышает некоторой величины T.
11. Реализовать операцию слияния двух двухуровневых иерархических списков так, чтобы в результирующем списке элементы, содержащиеся в обоих исходных списках, не повторялись ни на первом, ни на втором уровне.
12. Создать информационную сеть кафедры ВУЗа на базе списка, содержащего информацию о преподавателях: имя, фамилия, должность, количество часов учебной нагрузки по каждой из дисциплин (лекционных, семинарских, лабораторных занятий, зачетов, экзаменов и курсовых проектов). Количество ассоциативных списков определяется количеством учебных дисциплин, т.е. в один список объединяются записи о преподавателях, которые имеют учебную нагрузку по данной дисциплине. Получить распределение учебной нагрузки по кафедре в виде:
   а) №п/п <Имя> <Фамилия> <Должность>
      Лекции: <общее количество часов>
      Семинары: <общее количество часов>
      Лабораторные: <общее количество часов>
      Зачеты: <общее количество часов>
      Экзамены: <общее количество часов>
      Курс. проекты: <общее количество часов>
      Итого: <общее количество часов>
   б) №п/п <Наименование дисциплины>
      Лекции: <общее количество часов>
      Семинары: <общее количество часов>
      Лабораторные: <общее количество часов>
      Зачеты: <общее количество часов>
      Экзамены: <общее количество часов>
      Курс. проекты: <общее количество часов>
      Итого: <общее количество часов>

2. ДЕРЕВЬЯ

2.1. АТД «дерево»

Дерево представляет собой совокупность элементов (узлов), между которыми установлены иерархические отношения типа «предок-потомок» («родитель-сын»), причем каждый из этих элементов, может иметь несколько истинных потомков (сыновей) и только одного истинного предка, кроме одного элемента, называемого **корнем дерева**, у которого нет предка.

Истинный потомок или **истинный предок** – узел, являющийся потомком или предком для некоторого узла, кроме него самого. В дереве только корень не имеет истинного предка, но может быть множество узлов, не имеющих истинных потомков. Каждый из элементов дерева является одновременно и потомком и предком для самого себя, поэтому один узел также является деревом. Дерево без узлов называется нулевым или пустым деревом.

Лист дерева – узел, не имеющий истинных потомков.

Путь из одного узла дерева в другой есть упорядоченная последовательность узлов таких, что каждый последующий узел является потомком предыдущего.

Длина пути – число, на единицу меньшее количества узлов, составляющих этот путь.

Высота узла – длина самого длинного пути из этого узла до какого-либо листа дерева.

Высота дерева – высота его корня.

Глубина узла – длина пути от корня дерева до этого узла.

Неупорядоченное дерево – дерево, в котором порядок следования потомков узлов не учитывается и может быть произвольным.

Упорядоченное дерево – дерево с определенным порядком следования потомков одного узла. Обычно потомки узла упорядочиваются слева направо так, что если узлы u_1, u_2 являются прямыми потомками одного и того же узла и первый лежит левее второго, то считается, что все потомки узла u_1 лежат левее любого потомка узла u_2.

Каждый узел дерева можно сопоставить с некоторым значением, которое называется **меткой узла** и служит для его идентификации в дереве. Деревья, у которых узлы имеют метки, называются **помеченными деревьями**.

Поддерево дерева – любой элемент в дереве вместе со всеми своими потомками. Пусть в некотором дереве T узлы с метками u_1, u_2, …,u_m являются прямыми потомками узла с меткой u_0, тогда u_0 можно рассматривать как корень некоторого дерева T_0, являющегося поддеревом дерева T, а u_i (i=1,2,…,m) – как корни деревьев T_i, каждое из которых является одновременно поддеревом дерева T_0 и дерева T.

Для таких структур как деревья одной из наиболее часто выполняемых процедур является обход всех узлов дерева в некоторой последовательности и вывод полученного списка узлов. Существует **три способа обхода всех узлов дерева** – в прямом, симметричном и обратном порядках. Их рекурсивные алгоритмы можно объединить в одном описании:

1. Если дерево пустое, то занести в список обхода пустую запись и перейти к п.4.
2. Если дерево состоит из одного узла, то в список обхода занести этот узел и перейти к п.4.
3. Если дерево состоит из корня R, у которого k прямых потомков, являющихся корнями поддеревьев T_i (i=1,2, … k), то:

при **прямом обходе дерева** первым выбирается корень R, затем –последовательно в прямом порядке все узлы поддеревьев T_1, T_2 и т.д.;

при **симметричном обходе дерева** сначала выбираются все узлы поддерева T_1 в симметричном порядке, затем – корень R и далее последовательно в симметричном порядке все узлы поддеревьев T_2, T_3 и т.д.;

при **обратном обходе дерева** сначала выбираются последовательно в обратном порядке все узлы поддеревьев T_1, T_2 и т.д., а затем, т.е. в последнюю очередь, – корень R.

4. Закончить обход узлов дерева.

Графически порядок обхода дерева изображается в виде непрерывного контура, проходящего вдоль всех частей дерева, как показано на рис. 2.1, где в качестве примера изображен контур прямого обхода дерева: R, u_1, u_3, u_6, u_4, u_2, u_5, u_7, u_8, u_9.

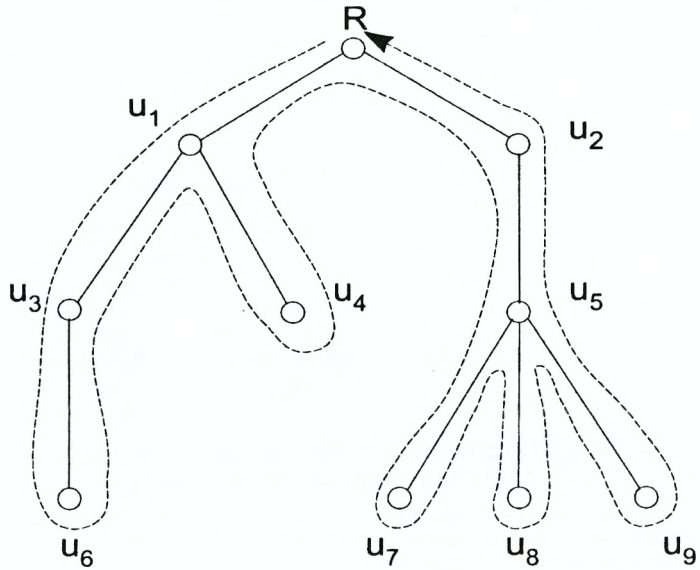


Рис. 2.1. Прямой (штриховая линия) способ обхода дерева.

При симметричном обходе дерева, изображенного на рис. 2.1, получим такую последовательность меток узлов:

u_6, u_3, u_1, u_4, R, u_7, u_5, u_8, u_9, u_2

а при обратном обходе –

u_6, u_3, u_4, u_1, u_7, u_8, u_9, u_5, u_2, R

Так же, как элементы списков, узлы деревьев представляются в виде записей, поля которых подразделяются на справочную, информационную и ссылочную части. Состав информационной и ссылочной частей может варьироваться в зависимости от типа дерева и средств реализации АТД, но метка узла, как правило, относится к информационной части.

Общепринятым является **представление деревьев посредством связанных списков потомков**, когда каждый узел является одновременно головным элементом списка своих прямых потомков (для листьев этот список будет пустым) и, за исключением корня, входит в список потомков своего прямого предка (рис. 2.2).

Такое представление деревьев равносильно представлению посредством иерархического списка, в котором подсписок уровня 0 состоит из одного узла – корня дерева, а подсписки уровня 1 и всех последующих уровней формируются из прямых потомков узлов предыдущего уровня. Количество уровней в этой иерархии определяется как максимальная глубина узлов дерева плюс 1.

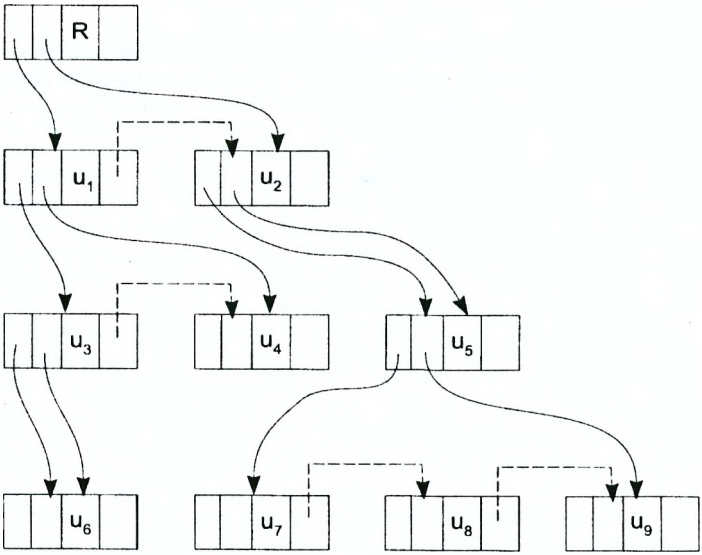


Рис. 2.2. Представление дерева, изображенного на рис. 2.1, посредством связанных списков потомков.

Каждый узел дерева при представлении посредством связанных списков потомков должен содержать, по меньшей мере, две ссылки: ссылку на следующий за ним узел в списке прямых потомков того же предка – такой узел называется правым братом – и ссылку на первый элемент списка прямых потомков. По-видимому, наиболее общим будет следующее **определение типа узлов дерева**:

```
typedef struct node
{ LABEL label;        /* Поле метки. */
  struct node *par,   /* Ссылка на прямого предка. */
              *suc;   /* Ссылка на правого брата. */
              *first; /* Ссылка на первого сына. */
              *last;  /* Ссылка на последнего сына. */
} NODE;
```

где через LABEL обозначен тип метки узлов.

Рассмотрим **основные операции** для данного АТД:

1. Обход узлов дерева в прямом, симметричном или обратном порядке.

Очевидной является реализация любого способа обхода дерева, исходя из описания их рекурсивного алгоритма, т.е. в виде рекурсивной функции. Например, функция, исходный код которой представлен в Листинге 2.1, реализует симметричный способ обхода дерева.

Листинг 2.1

```
/* Симметричный способ обхода дерева.
   Параметры: root – указатель на корень дерева; putlabel – указатель на функцию, которая выводит
   передаваемую ей метку узла в список обхода и возвращает 1 при успешном завершении или 0 в
   противном случае.
   Возвращаемое значение: количество узлов в дереве. */
```

```
int list_node(NODE * root, int (*putlabel)(LABEL))
{ int n=0;
  if (root == NULL) return 0;
  n += list_node (root->first, putlabel);
  n += (*putlabel)(root->label);
  for (root = root->first; root->suc != NULL); root = root->suc)
      n += list_node (root);
  return n;
}
```

Нерекурсивный вариант реализации операции обхода узлов дерева предполагает использование вспомогательной памяти, в которой хранились бы метки или адреса узлов, составляющих путь от корня дерева до текущего узла. Для организации такой вспомогательной памяти удобно использовать стек, элементы которого содержат ссылки на соответствующие им узлы дерева.

2. Поиск узла по его метке.

Необходимым условием корректности постановки задачи о поиске узла по его метке является уникальность меток всех узлов дерева. Если это условие не выполняется, необходимы дополнительные критерии поиска, например, метка прямого предка и/или первого сына.

В неупорядоченном дереве найти узел по его метке можно лишь в результате последовательного перебора всех узлов при обходе дерева в том или ином порядке. В Листинге 2.2 представлен исходный код рекурсивной функции, реализующей поиск узла по его метке путем полного перебора узлов дерева по схеме прямого обхода.

Листинг 2.2

```
/* Поиск узла.
   Параметры: root – указатель на корень дерева; label – метка искомого узла; cmplabel – указатель
   на функцию, которая сравнивает два значения типа LABEL и возвращает 1, 0 или -1, если, соответственно, первое больше второго, они равны или первое меньше второго.
   Возвращаемое значение: при успешном завершении – указатель на искомый узел, иначе – NULL. */
NODE * search_node(NODE * root,
                   LABEL label,
                   int (*cmplabel)(LABEL, LABEL))
  { NODE * q = NULL;
    if (root == NULL) return NULL;
    if ((*cmplabel)(root->label, label) == 0) return root;
    root = root->first;
    while (root != NULL)
        { if ((q = search_node(root, label, cmplabel)) != NULL) break;
          root = root->suc; }
    return q;
}
```

Таким образом, поиск в неупорядоченном дереве выполняется за время, пропорциональное количеству узлов. В упорядоченных деревьях, за счет того, что метки узлов удовлетворяют определенному соотношению, просмотр узлов обычно выполняется по правилу «один узел – один уровень», так что зависимость времени выполнения операции поиска от количества узлов имеет не линейный, а логарифмический характер.

3. Поиск первого справа соседнего узла.

Поиск первого справа соседнего узла на том же уровне глубины или, просто, правого соседа может завершиться с одним из следующих результатов:

- Правым соседом может быть правый брат данного узла.

- Если узел r замыкает список сыновей своего прямого предка, правым соседом для него будет первый сын узла, который является правым соседом для прямого предка узла r. Например, на рис. 2.2 правый сосед для узла u_4 – это узел u_5, который является первым сыном правого брата узла u_1 – прямого предка узла u_4. Налицо рекурсивная структура решаемой задачи – чтобы найти правого соседа узла, необходимо найти правого соседа для прямого предка этого узла.
- Правый сосед может отсутствовать вовсе, если, очевидно, узел является крайним справа узлом на своем уровне (узлы u_2, u_5, u_9 на рис. 2.2).

В Листинге 2.3 приведен исходный код рекурсивной функции, реализующей операцию поиска правого соседа для данного узла при условии, что его адрес последнего известен и передается в функцию посредством единственного параметра.

Листинг 2.3

```
/* Поиск первого справа соседа узла.
   Параметры: p – указатель на узел, для которого производится поиск первого справа соседа.
   Возвращаемое значение: при успешном завершении – указатель на искомый узел, иначе – NULL. */
NODE *right_neighb(NODE *p)
{ NODE *q;
  if ((p == NULL) || (p->par == NULL)) return NULL;
  if ((q = p->suc) == NULL)
     { p = p->par,
       while ((p = right_neighb(p)) != NULL) && (q == NULL))
           q = p->first;
     }
  return q;
}
```

4. Поиск крайнего слева/справа листа.

Для поиска крайнего слева/справа листа в дереве надо пройти от корня дерева по ссылкам первых/последних сыновей. Так, для дерева на рис. 2.2 крайний слева – узел u_6, а крайний справа – узел u_9.

5. Вставка узла.

Если дерево не пусто, непосредственной вставке нового узла в дерево предшествует поиск прямого предка для него. В отличие от упорядоченного дерева, в котором новый узел должен занять вполне определенное место так, чтобы упорядоченность узлов не была нарушена, для вставки узла в неупорядоченное дерево необходимо явно указать, какой узел будет прямым предком для нового узла. В функции, исходный код которой представлен в Листинге 2.4, прямой предок нового узла задается своим адресом, который передается в функцию посредством одного из параметров, причем если соответствующий фактический параметр равен NULL, то новый узел должен стать корнем дерева.

Если дерево пусто (указатель на корень дерева равен NULL), то вновь добавленный узел становится его корнем и операция успешно завершается.

Листинг 2.4

```
/* Вставка узла в дерево.
   Параметры: root – указатель на указатель на корень дерева; pnode – указатель на узел, который
   будет прямым предком нового узла; label – метка нового узла; set_label – указатель на функцию,
   которая записывает значение метки по указанному адресу.
   Возвращаемое значение: при успешном завершении – указатель на новый узел, иначе – NULL. */
NODE * add_node(NODE **root,
                NODE *pnode,
                LABEL label,
                void (*set_label)(LABEL *, LABEL))
```

```
{ NODE *p, *r = *root;
  p = malloc(sizeof(NODE));
  if ((p == NULL) || (r == NULL) && (pnode != NULL)) return NULL;
  (*set_label)(&p->label, label);
  if (r != NULL)        /* Если дерево не пусто. */
     if (pnode != NULL)     /* Если новый узел – не корень дерева. */
        { p->par = pnode;
          /* Если у прямого предка нового узла есть сыновья, то новый узел становится последним сыном, иначе – первым сыном. */
          if (pnode->first != NULL)
             { pnode->last->suc = p;
               pnode->last = p; }
          else
             pnode ->first = pnode ->last = p;
        }
     else      /* Если новый узел – корень дерева. */
        { p->par = p->next = NULL;
          p->first = p->last = r;
          r->par = p;
          *root = p; }
  else /* Если дерево пусто. */
     { p->par = p->first = p->last = p->next = NULL;
       *root = p; }
  return p;
}
```

6. Удаление узла.

В общем случае декомпозиция задачи удаления узла приводит к двум подзадачам:

- исключение удаляемого узла из списка сыновей его прямого предка;
- размещение в дереве потомков удаляемого узла.

При удалении листа или корня дерева можно, очевидно, ограничиться решением лишь одной из указанных подзадач.

Исключение удаляемого узла из списка сыновей его прямого предка выполняется в соответствии с описанными выше алгоритмами для списков и не зависит от типа дерева. Вопрос о размещении потомков удаляемого узла, напротив, решается по-разному в разных типах деревьев, но фактически всегда сводится к передаче потомков удаляемого узла другому узлу, например, прямому предку удаляемого узла или какому-нибудь другому узлу с подходящей меткой.

В неупорядоченном дереве потомки удаляемого узла могут быть переданы практически любому узлу, поэтому, как и при вставке, необходимо явно указать метку или адрес такого узла. Если при этом удаляется корень дерева, то новым корнем становится тот узел, которому передаются потомки удаляемого узла (см. Листинг 2.5).

Листинг 2.5

```
/* Удаление узла из дерева.
   Параметры: root – указатель на указатель на корень дерева; pnode – указатель на узел, которому передаются потомки удаляемого узла; node – указатель на удаляемый узел.
   Возвращаемое значение: при успешном завершении – указатель на узел, которому передаются потомки удаляемого узла, иначе – NULL. */
NODE * delete_node(NODE **root, NODE *pnode, NODE *node)
{ NODE *r = *root, *t, *u, *q;
  if ((node == NULL) || (pnode == NULL) || (r == NULL)) || (node == pnode))
     return NULL;
```

```
/* Исключить узел node, если он – не корень, или узел pnode в противном случае из списка сыновей
   соответствующего прямого предка. /
if (node->par != NULL) u = node; else u = pnode;
t = u->par;
if (u == t->first) t->first = u->suc;
else
      { for (q = t->first; q->next != u ; q = q->next);
        if ((q->next = u->next) == NULL) t->last = q; }
/* Добавить список сыновей узла node в список сыновей узла pnode */
if (node->first != NULL)
    { if (pnode->first == NULL) pnode->first = node->first;
      else pnode->last->suc = node->first;
      pnode->last = node->last; }
for (q = node->first; q !=NULL; q = q->suc) q->par = pnode;
/* Если удаляемый узел – корень, изменить адрес корня дерева */
if (node->par == NULL)
    { pnode->par = NULL;
      *root = pnode; }
free(q);
return pnode;
}
```

В общем случае количество операций, их реализация и, соответственно, время выполнения зависят от типа дерева и выбранных средств реализации соответствующего АТД. Поэтому далее рассматриваются достаточно широко используемые в практике программирования типы деревьев: деревья выражений, деревья двоичного поиска и частично упорядоченные деревья.

2.2. Деревья выражений

Дерево выражения – это дерево, в котором узлы помечены лексемами этого выражения. Рассмотрим этот тип деревьев на примере арифметических/логических выражений.

Деревья арифметических/логических выражений будем строить в соответствии со следующими правилами:

- Выражению вида $E_1\delta E_2$, где δ – знак бинарной операции, E_1 и E_2 – выражения, соответствует дерево, корень которого помечен знаком операции, а выражения-операнды образуют два его поддерева. Это означает, что если некоторый узел имеет метку в виде знака бинарной операции, например, «+», а два его прямых потомка имеют метки, например, x и y, то вместе они представляют выражение x+y.
- Выражению вида δE, где δ – знак унарной операции, E – выражение, соответствует дерево, корень которого помечен знаком или мнемокодом операции (exp, sin, sqrt и т.п.), первое поддерево состоит из одного узла, помеченного символом «0» для унарных операций «+» и «–» или «пустой» меткой для всех остальных унарных операций, а второе поддерево соответствует выражению-операнду. Различие между меткой «0» и «пустой» меткой заключается в том, что узлы с «пустыми» метками игнорируются при любом способе обхода дерева, а узлы с метками «0» – только при симметричном. При прямом и обратном обходе метки «0» позволят отличить (и корректно выполнить) унарные операции от бинарных, обозначаемых такими же знаками.
- Дерево выражения со скобками вида (E) строится на основе дерева выражения E так, что узел с меткой «(» будет прямым и единственным потомком узла, соответствующего первому операнду выражения E, а узел с меткой «)» – правым братом узла, являющегося корнем поддерева, соответствующего последнему операнду выражения E.

Таким образом, в дереве арифметического/логического выражения листья могут быть помечены идентификаторами переменных, константами и скобками, а внутренние узлы – идентификаторами унарных и бинарных операций. Например, дерево выражения (c + 2*(d – 1))*ln(a + b) будет выглядеть так, как показано на рис. 2.3.

При обходе деревьев выражений тем или иным способом получаются соответствующие формы записи этих выражений: при прямом обходе – **префиксная форма записи**, при обратном обходе – **постфиксная форма записи**, а при симметричном обходе – **инфиксная форма записи**, которая совпадает с математической формой записи. Например, используя дерево выражения, приведенное на рис. 2.3, можно легко получить префиксную

* + c * 2 – d 1 ln + a b

и постфиксную

c 2 d 1 – * + a b + ln *

формы записи этого выражения.

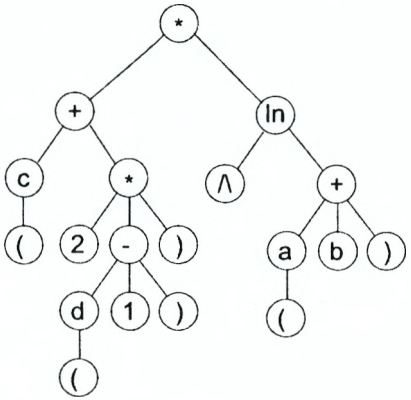


Рис. 2.3. Дерево выражения (c + 2*(d – 1))*ln(a + b) («пустая» метка обозначена символом «Λ»).

В префиксной и постфиксной формах записи выражений скобки не нужны, т.к. всегда можно корректно сформировать, причем единственным образом, выражения вида

<операция><операнд1>< операнд2>

или

<операнд1><операнд2><операция>.

Инфиксная форма записи выражения возможна без скобок и со скобками. Следует отметить, что скобки нужны лишь тогда, когда необходимо получить инфиксную форму записи выражения, в остальных случаях их можно просто опустить.

С учетом сказанного тип узлов дерева выражения можно определить как

```
typedef struct node
{ char * label;          /* Метка узла *
  struct node * s1;   /* Ссылка на первого сына*
                * s2;        /* Ссылка на второго сына*/
                * s3        /* Ссылка на третьего сына*/
} NODE;
```

Особенностью деревьев выражений является то, что для их построения, как правило, требуется предварительный синтаксический и семантический анализ всего выражения, поэтому к основным операциям над деревьями выражений относятся:

- создание дерева выражения;
- обход узлов дерева выражения в прямом, симметричном или обратном порядке.

2.2.1. Создание дерева выражения

Поскольку любой из операндов выражения также является выражением, то для создания дерева выражения можно использовать рекурсивный подход. Ниже, в Листинге 2.6, приведено описание на псевдоязыке рекурсивного алгоритма построения дерева арифметического выражения вида f(x), т.е. выражения с одной переменной, в котором над операндами выполняются унарные или бинарные операции и операнды могут быть заключены в круглые скобки.

Листинг 2.6

```
/* Создание дерева выражения.
   Параметры: str – адрес строки, в которой записано выражение.
   Возвращаемое значение: при успешном завершении – указатель на корень сформированного дерева, иначе – NULL.
   Для вывода сообщения об ошибке используется макрос
   #define ERROR { printf("Ошибка синтаксиса выражения: %s", str); return NULL; } */
NODE * build_tree(char * str)
{ NODE * R=NULL, *ptr, *f;          /* R – указатель на корень дерева */
  char s, * q, * buf, *p;
  while (str != NULL)/* Пока не конец строки */
   { s = *str;
     q = str;
_1: if (s – буква)
    { if ((s == 'x') || (s == 'X'))        /* Если переменная */
        { buf = malloc(2);
          strncpy(buf, str, 1); }
      else      /* Если идентификатор унарной операции */
        { while (*q – буква и *q != '\0') q++;
          if (*q == '(')
            { buf = malloc(q – str + 1);
              strncpy(buf, str, q – str);
              if (buf – не идентификатор унарной операции)
                  ERROR
              else
                { Найти адрес соответствующей скобки «)»;
                  if (Адрес найден)
                      Записать адрес в p;
                  else ERROR }
            }
          else ERROR
        }
```

```
    Создать новый узел с меткой buf и его адрес записать в ptr;
    if (strlen(buf) > 1)
      { Создать узел с «пустой» меткой и сделать его первым сыном узла ptr;
        buf=malloc(p – q + 2);
        strncpy(buf, q, p – q + 1);
        if ((ptr->s2 = build_tree(buf)) == NULL) return NULL; }
    if (R)
      { for (f = R; f->s2 != NULL; f = f->s2);
        if (f->s1 != NULL) ERROR
        f->s2=ptr; }
    else R=ptr;
    if (q == str) str++; else str = p + 1;
  }
_2: if (s – цифра от 0 до 9)
  { float C;
    while (*q – один из символов десятичной записи числа и *q!='\0') q++;
    buf = malloc(q – str);
    strncpy(buf, str, q – str – 1);
    if (sscanf(buf, "%d", C))
        Создать новый узел с меткой buf и его адрес записать в ptr;
    else
          ERROR
    if (R)    /* Если дерево не пусто */
      { for(f = R; f->s2 != NULL; f = f->s2);
        if (f->s1 != NULL) ERROR
        f->s2 = ptr; }
    else R = ptr;
    str = q;
  }
_3: if (s – символ «+» или «–»)
  { buf = malloc(2);
    strncpy(buf, str, 1);
    Создать новый узел с меткой buf и его адрес записать в ptr;
    if (R)    /* Если дерево не пусто */
      { for (f = R; f->s2 != NULL; f = f->s2);
        if (f->s1 != NULL) ERROR
        ptr->s1 = R;
        R = ptr; }
    else      /* Унарная операция */
        { Создать узел с меткой «0» и сделать его первым сыном узла ptr;
          R=ptr; }
    str++;
  }
_4: if (s – символ «*» или «/»)
  { Сформировать строку buf из одного символа s;
    Создать новый узел с меткой buf и его адрес записать в ptr;
    if (R)    /* Если дерево не пусто */
      { for(f = R; f->s2 != NULL; f = f->s2);
        if (f->s1 != NULL) ERROR
```

```
        if ((*R->label == '+') ||
              (*R->label == '–') && strcmp(R->s1->label, "0"))
          { ptr->s1 = R->s2;
            R->s2 = ptr; }
        else
          { ptr->s1 = R;
            R = ptr; }
      }
    else ERROR
    str++;
  }

_5: if (s – символ «(»)
  { Найти адрес соответствующей скобки «)»;
    if (Адрес найден)
        Записать найденный адрес в p;
    else ERROR
    buf = malloc(p – q);
    strncpy(buf, q + 1, p – q – 1);
    for(f = R; f->s2 != NULL; f = f->s2);
    if (f->s1 != NULL) ERROR
    ptr = f->s2 = build_tree(buf);
    if (ptr == NULL) return NULL;
    for(f = ptr; f->s1 != NULL; f = f->s1);
    Создать узел с меткой «(» и сделать его первым сыном узла f;
    for(f = ptr->s3; f->s1 != NULL; f = f->s1);
    Создать узел с меткой «)» и сделать его первым сыном узла f;
    if (R)    /* Если дерево не пусто */
      { for (f = R; (f->s2 != NULL) f = f->s2) ;
        if (f->s1 == NULL) ERROR
        f->s2 = ptr; }
    else R = ptr;
    str =p+1;
  }
 }
 return R;
}
```

Рассмотрим пошаговое выполнение описанного выше алгоритма на примере выражения 1 + 2*x*x + exp((x + 1)*(– 2 / x)):

1-ый шаг:

Считывается символ «1».

Дерево состоит из одного узла, он же является корнем дерева.

2-ой шаг:

Считывается символ «+».

Узел с меткой «+» становится новым корнем дерева, а старый корень становится его первым сыном.

3-ий шаг:

Считывается символ «2».

Узел с меткой «2» становится вторым сыном корневого узла (рис. 2.4,а).

4-ый шаг:
Считывается символ «*».
Поскольку текущий корень дерева помечен символом «+», узел с меткой «*» становится вторым сыном корневого узла (рис. 2.4,б).
5-ый шаг:
Считывается символ «x».
Поскольку у текущего корня дерева уже есть второй сын, узел с меткой «x» становится вторым сыном узла, помеченного символом «*» (рис. 2.4,в).

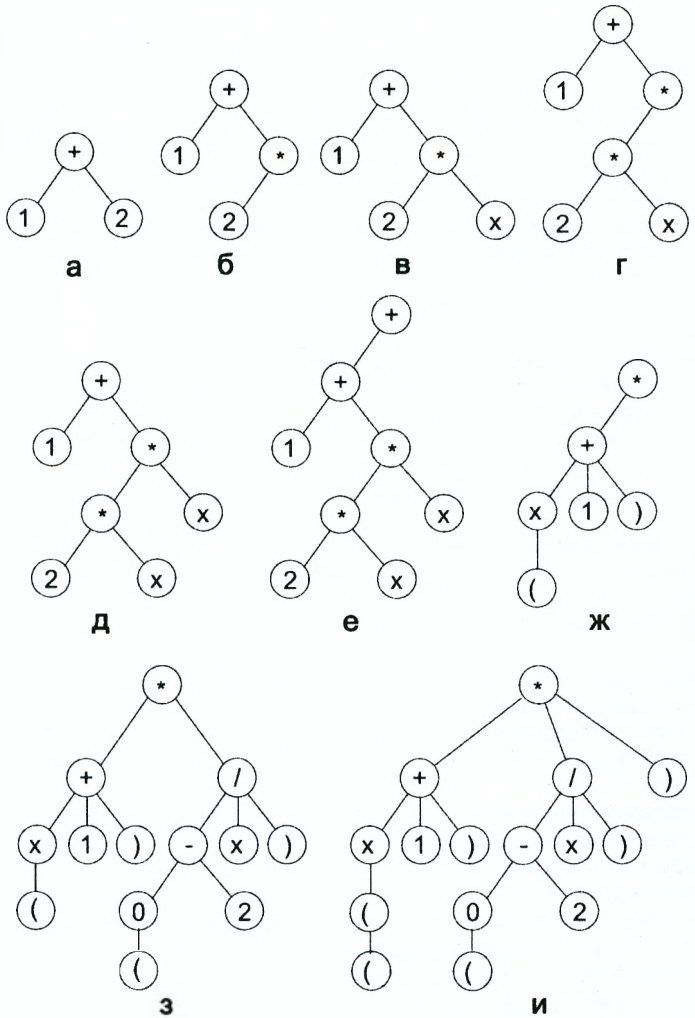


Рис. 2.4. Последовательность шагов при построении дерева выражения 1+2*x*x+exp((x+1)*(-2/x)).

6-ой шаг:
Считывается символ «*».
Узел с меткой «*» становится вторым сыном корневого узла, замещая стоящий в этой позиции узел (рис. 2.4,г).

7-ой шаг:
Считывается символ «x».
Узел с меткой «x» становится вторым сыном узла с меткой «*», вставленного в дерево на предыдущем шаге (рис. 2.4,д).
8-ой шаг:
Считывается символ «+».
Узел с меткой «+» становится корнем дерева, а старый корень – его первым сыном (рис. 2.4,е).
9-ый шаг:
Считывается строка «exp».
Считанная строка является идентификатором унарной операции, поэтому вновь созданный узел с такой же меткой становится вторым сыном текущего корня дерева. Первым сыном узла с меткой «exp» будет узел с «пустой» меткой, а вторым – корень поддерева выражения, являющегося аргументом операции (включая скобки), для построения которого производится первый рекурсивный вызов.
10-ый шаг (1-ый рекурсивный вызов):
На входе строка «((x+1)*(-2/x))».
Поскольку выражение начинается с открывающей скобки, определяется выражение, стоящее между скобками, и это выражение используется в качестве входного параметра для второго рекурсивного вызова.
11-ый шаг (2-ой рекурсивный вызов):
На входе строка «(x+1)*(-2/x)».
Аналогично предыдущему шагу производится третий рекурсивный вызов с параметром «x+1».
12-ый шаг (3-ий рекурсивный вызов):
На входе строка «x+1». Строится поддерево выражения x+1. Возврат во второй рекурсивный вызов.
13-ый шаг (продолжение 2-го рекурсивного вызова):
На входе строка «(x+1)*(-2/x)».
К поддереву выражения «x+1» добавляются узлы, помеченные скобками, и в это поддерево добавляется новый корень, помеченный символом «*» (рис. 2.4,ж). Далее считывается открывающая скобка и производится четвертый рекурсивный вызов с параметром «-2/x».
15-ый шаг (4-ый рекурсивный вызов):
На входе строка «-2/x».
Строится поддерево выражения «–2/x». Возврат во второй рекурсивный вызов.
16-ый шаг (завершение 2-го рекурсивного вызова):
На входе строка «(x+1)*(-2/x)».
К поддереву выражения «–2/x» добавляются узлы, помеченные скобками, и корень этого поддерева становится вторым сыном корня всего выражения, помеченного символом «*» (рис. 2.4,з). Возврат в первый рекурсивный вызов.
17-ый шаг (завершение 1-го рекурсивного вызова):
На входе строка «((x+1)*(-2/x))».

К поддереву выражения «(x+1)*(-2/x)» добавляются узлы, помеченные скобками (рис. 2.4,и). Завершение рекурсивных вызовов и возврат к состоянию, полученному в конце 9-го шага.

18-ый шаг:

Корень поддерева выражения, полученный после завершения рекурсивных вызовов, становится вторым сыном узла, помеченного знаком унарной операции «exp». Текущий указатель в исходной строке устанавливается на нулевой байт, и процедура завершается возвратом в вызывающую программу адреса корня построенного дерева выражения (рис. 2.5).

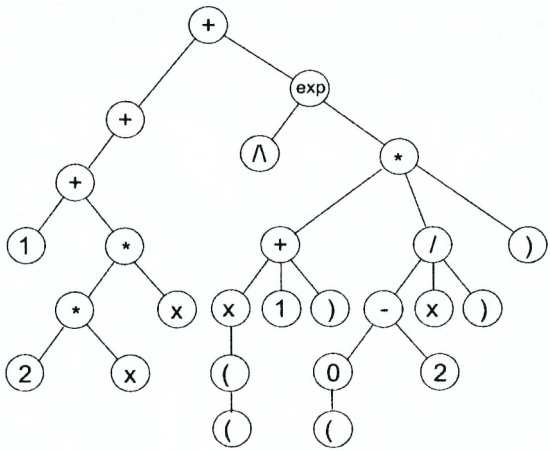


Рис. 2.5. Дерево выражения 1+2*x*x+exp((x+1)*(-2/x)).

2.2.2. Обход дерева выражения

Алгоритм симметричного способа обхода дерева выражения, как, впрочем, и два других, легко реализуется в виде рекурсивной функции, исходный код которой представлен в Листинге 2.7. Здесь же приведено определение функции, которая выполняет операцию конкатенации двух строк, одна из которых размещена в области динамического распределения памяти.

Листинг 2.7

```
/* Симметричный способ обхода дерева выражения (рекурсивный вариант).
   Параметры: ptrnode – указатель на корень дерева; concat – указатель на функцию, которая выполняет конкатенацию двух строк.
   Возвращаемое значение: при успешном завершении – указатель на строку, содержащую соответствующую форму записи исходного выражения, иначе – NULL. */
char * list_node (NODE * ptrnode, char * (*concat)(char *, char *))
{ NODE * q;
  char * str = NULL, *label;
  if (ptrnode != NULL)
      { if ((q = ptrnode->s1) != NULL) str = list_node (q, concat);
        label = ptrnode->label;
        if ((label != NULL) && strcmp(label, "0")) /* Если метка узла – не «0» */
            if (str != NULL)
                if ((*concat)(str, label) == NULL) return NULL;
```

```
        else str = strdup(label);
        if (q = ptrnode->s2)
            if ((*concat)(str, list_node (q)) == NULL) return NULL;
        if (q = ptrnode->s3)
            if ((*concat(str, list_node (q)) == NULL) return NULL;
    }
  return str;
}
/* Конкатенация двух строк.
   Параметры: str1, str2 – указатели на строки.
   Возвращаемое значение: указатель на результирующую строку (str1).
   Строка, адрес которой содержит параметр str1, должна размещаться в области динамического
   распределения памяти. */
char * concat(char *str1, char * str2)
{ realloc(str1, strlen(str1)+ strlen(str2)+1);
  strcat(str1, str2);
  return str1;
}
```

В основе нерекурсивного варианта алгоритма симметричного способа обхода дерева выражения лежит использование вспомогательной памяти, организованной в виде стека. Если через root обозначить указатель на корень дерева, а через top – указатель на узел, соответствующий элементу, находящемуся в вершине стека, то алгоритм симметричного способа обхода дерева выражения можно представить на псевдоязыке следующим образом:

```
/* Начало процедуры */
if (root != NULL)
    Добавить root в вершину стека;
else
    Завершить процедуру;
root = root->s1;
while (top != NULL)          /* Пока стек не пустой */
      if (root != NULL)
          { Добавить узел root в вершину стека;
            root = root->s1; }
      else
          { if (root->label – не «пустая» метка)
                Вывести метку узла root в список обхода;
            if (root->s3 != NULL)
                Добавить узел root->s3 в вершину стека;
            if (root->s2 != NULL)
                root = root->s2;
            else root = NULL;
            Удалить элемент из стека; }
/* Конец процедуры */
```

Исходный код функции, реализующей описанный выше алгоритм, представлен в Листинге 2.8. Здесь тип элементов стека определен как

```
typedef struct item
{ NODE * ptrnode;   /* Указатель на узел дерева выражения *
  struct item * ptr;   /* Указатель на следующий элемент в стеке */
} ITEM;
```

а адреса функций, выполняющих операции вставки и удаления элементов стека, передаются в основную функцию посредством параметров.

Листинг 2.8

```
/* Симметричный способ обхода дерева выражения (нерекурсивный вариант).
   Параметры: root – указатель на корень дерева; concat – указатель на функцию, которая выполняет конкатенацию двух строк; add – указатель на функцию, которая выполняет операцию вставки элемента в стек; del – указатель на функцию, которая выполняет операцию удаления элемента из стека.
   Возвращаемое значение: при успешном завершении – указатель на строку, содержащую соответствующую форму записи исходного выражения, иначе – NULL. */
char * list_node2(NODE * root,
                  char * (*concat)(char *, char *),
                  ITEM * (*add)(ITEM *, NODE *),
                  ITEM * (*del)(ITEM *))
{ ITEM * top;  /* Указатель на вершину стека */
  char * str = NULL;
  if (root != NULL)
      top = (*add)(top, root);
  else return NULL;
  root = root->s1;
  while ( top != NULL)
        if (root != NULL)
            { top = (*add)(top, root);
              root = root->s1; }
        else
            { if (root->label) && strcmp(root->label,"0"))
                  if (str != NULL) concat(str, label); else str = strdup(label);
              if (root->s3 != NULL)
                  top = (*add)(top, root->s3);
              if (root->s2 != NULL)
                  root = root->s2;
              else root = NULL;
              top = (*del)(top); }
}
```

2.3. Двоичные деревья

Двоичное (бинарное) дерево – упорядоченное ориентированное дерево, у которого любой узел имеет не более двух прямых потомков, называемых левым и правым сыном (т.к. ребра, связывающие узел с его потомками, расходятся от узла вниз и влево или вниз и вправо).

К двоичным деревьям относятся рассмотренные в предыдущем пункте деревья выражений без скобок. Другим примером этого класса рекурсивных структур служат структуры данных, используемые при статистическом кодировании информационных сообщений. Рассмотрим систему кодирования информации на основе алгоритма Хаффмана.

Символы, из которых состоит сообщение, можно закодировать, используя простую семибитовую или восьмибитовую кодировку, т.е. представляя каждый символ в виде его двоичного ASCII-кода. В такой кодировке коды всех символов будут иметь одинаковую длину, но частота, а значит и вероятность, появления отдельных символов в сообщениях, составленных на естественном языке, не является инвариантом. Для русского языка, например, вероятности появления букв в текстах сообщений имеют следующие значения: А – 0,062, О – 0,09, И – 0,062, Н – 0,053, Ю – 0,006 и т.д.

В этой связи возникает проблема создания такой двоичной кодировки, чтобы средняя длина кода (в вероятностном смысле) была минимальной, поскольку, чем меньше средняя длина кода, тем короче закодированное сообщение. Поскольку длина кода является случайной дискретной величиной, ее среднее значение вычисляется как сумма

$$\sum_{i=1}^{N} s_i p_i , \qquad (2.1)$$

где s_i, p_i – длина кода и вероятность появления i-го символа из общего количества N разных символов, представленных в сообщении. Алгоритм построения такой кодировки для заданного множества символов и вероятностей их появления в сообщениях называется **алгоритмом Хаффмана**, а полученные таким образом коды – **кодами Хаффмана**.

Алгоритм Хаффмана заключается в том, что из исходной совокупности символов выбираются два символа с наименьшими вероятностями и заменяются одним условным символом, вероятность которого равна сумме вероятностей этих двух символов, и так продолжается до тех пор, пока не останется один условный символ с вероятностью 1. Если представить эту процедуру в виде графа, то такой граф будет не чем иным как двоичным деревом, листья которого помечены кодируемыми символами, а внутренние узлы соответствуют условным символам, образующимся при пошаговой замене двух символов с наименьшими вероятностями (см. рис. 2.6).

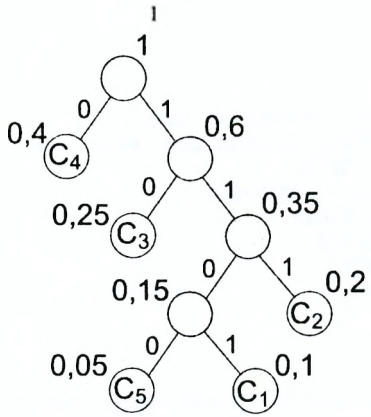


Рис. 2.6. Пример построения дерева кодов Хаффмана для алфавита из пяти символов (вероятность/код): C1 – 0,1/1101; C2 – 0,2/111; C3 – 0,25/10; C_4 – 0,4/0; C_5 – 0,05/1100. Средняя длина кода – 2,1.

Код любого символа получается как последовательность 0 и 1, формируемая при прохождении от корня дерева до соответствующего листа так, что при переходе к левому сыну любого узла в код записывается 0, а при переходе к правому – 1 (или наоборот). Заметим, что полученные таким образом коды будут уникальны, т.к. путь от корня дерева до любого листа всегда единственный.

Декодирование кодов Хаффмана производится последовательным удалением кодов символов из закодированного сообщения. Так как длины кодов символов различны, то декодирование очередного символа начинается с кода минимальной длины, т.е. из одного бита. Если символа с таким кодом нет, к нему добавляется следующий бит и проверяется, есть ли символ с полученным кодом длиной в два бита. Если такой символ в кодовой таблице есть, то производится его декодирование, иначе выбирается следующий бит и т.д.

Реализация алгоритма Хаффмана основывается на процедуре слияния двух деревьев. Действительно, в начале работы алгоритма имеем совокупность одноузловых деревьев, помеченных кодируемыми символами. При слиянии двух деревьев создается новый узел, который становится корнем объединенного дерева и соответствует псевдосимволу с вероятностью, равной сумме вероятностей узлов объединяемых деревьев (эту сумму далее будем называть весом дерева). Одно из объединяемых деревьев становится левым поддеревом, а другое – правым поддеревом объединенного дерева.

В Листинге 2.9 приведен исходный код функции, выполняющей построение двоичного дерева кодов Хаффмана с узлами, тип которых определен как

```
typedef struct node          /* Тип узлов дерева */
{ char label;  /* Метка узла */
  float prob;  /* Вес поддерева */
  struct node * left,  /* Ссылка на левого сына */
              * right;     /* Ссылка на правого сына /
} NODE;
```

Для организации совокупности объединяемых деревьев используется вспомогательный список, элементы которого содержат ссылки на корневые узлы объединяемых деревьев:

```
typedef struct item  /* Тип элементов вспомогательного списка */
{ float prob;
  struct item *suc;    /* Ссылка на следующий элемент */
  NODE * ptrnode;   /* Ссылка на корень поддерева */
} ITEM;
```

Изначально в списке должно быть столько элементов, сколько символов в кодируемом алфавите, и они должны быть упорядочены по неубыванию вероятностей символов.

Листинге 2.9

```
/* Построение дерева кодов Хаффмана.
   Параметры: h – указатель на первый элемент вспомогательного списка; del – указатель на
   функцию, которая удаляет элемент списка.
   Возвращаемое значение: при успешном завершении – указатель на корень сформированного дерева, иначе – NULL. */
NODE * build_tree(ITEM * h, ITEM * (*del)(ITEM *))
{ NODE * q;
  ITEM * p;
  float s = 0;
  unsigned size = sizeof(NODE);
  /* Проверить сумму вероятностей исходных символов */
  for(p = h; p != NULL; p = p->next) s += p->prob;
  if (s != 1) return NULL;
  while (h->suc != NULL)
      { if ((q = malloc(size)) == NULL) return NULL;
        q->left = h->ptrnode;
        q->right = h->suc->ptrnode;
        q->prob = h->prob + h->suc->prob;
        q->label = 0;
        if ((*del)(h->suc) == NULL) return NULL;
        h->ptrnode = q;
      }
  return h->ptrnode;
}
```

Для генерации кода некоторого символа с помощью дерева кодов необходимо записать по описанному выше правилу путь от корня дерева до листа, помеченного этим символом. Сгенерированные таким образом коды символов перед началом работы записываются в специальную таблицу и считываются оттуда в процессе кодирования сообщения, что позволяет избежать повторных вычислений одного и того же кода и ускорить тем самым процесс кодирования. Поскольку кодовая таблица должна обеспечивать поиск кода символа за фиксированное время, наиболее эффективной формой ее представления будет, очевидно, массив, элементы которого индексированы значениями самих символов.

2.4. Деревья двоичного поиска

Деревья двоичного поиска (ДДП) – двоичные деревья, обладающие следующими свойствами:

- для меток узлов определено отношение порядка больше/меньше;
- метки всех узлов имеют уникальные значения;
- метка любого узла (кроме листьев) больше метки любого его потомка из левого поддерева, но меньше метки любого потомка из правого поддерева.

Соотношение между метками узлов ДДП и метками их потомков приводит к тому, что список узлов ДДП, полученный при симметричном обходе, будет отсортирован в порядке возрастания меток узлов. Отсюда в частности следует, что крайний слева лист в ДДП имеет минимальную метку, а крайний справа лист – максимальную.

Рассмотрим **основные операции** для ДДП, определив тип элементов как

```
typedef struct node
        { LABEL label;          /* Метка узла */
          struct node * left,   /* Ссылка на левого сына */
                      * right;  /* Ссылка на правого сына /
        } NODE;
```

1. Поиск узла по его метке в ДДП.

Алгоритм поиска узла в ДДП строится на последовательном сравнении искомой метки с метками узлов, начиная с корня. Если искомая метка меньше, чем метка корня, то сравнение выполняется в таком же порядке с метками узлов его левого поддерева, если больше, то – с метками узлов его правого поддерева. Сравнение выполняется до тех пор, пока метка очередного узла не совпадет с искомой меткой или при необходимости перехода к левому или правому сыну соответствующая ссылка окажется равной NULL, что означает, что в дереве нет узла с указанной меткой. Исходный код функции, реализующей эту операцию, представлен в Листинге 2.10.

Листинг 2.10

```
/* Поиск узла в ДДП.
   Параметры: root – указатель на корень дерева; label – метка искомого узла; cmplabel – указатель
   на функцию, которая сравнивает два значения типа LABEL и возвращает 1, 0 или -1, если, соответственно, первое больше второго, они равны или первое меньше второго.
   Возвращаемое значение: при успешном завершении – указатель на найденный узел, иначе – NULL.
 */
NODE * search_node(NODE * root,
                   LABEL label,
                   int (*cmplabel)(LABEL, LABEL))
{ while (root != NULL)
    if ((*cmplabel)(root->label, label) == 0)
        break;
    else
        if ((*cmp)(root->label, label) == 1)
            root = root->left;
        else root = root->right;
  return root;
}
```

2. Вставка нового узла в ДДП.

Как и в любой другой связной структуре, для вставки нового элемента в ДДП необходимо создать новый узел и сформировать ссылки, позволяющие включить его в дерево. Однако в связи с упорядоченностью узлов в ДДП каждый новый узел занимает однозначно определяемую в данный момент времени позицию. Таким образом, вставка нового узла в ДДП начинается с поиска по описанному выше алгоритму истинного предка для нового узла с учетом того, что в этом качестве может выступать либо лист, либо узел с одним сыном. Исходный код функции, реализующей эту операцию, представлен в Листинге 2.11.

Листинг 2.11

```
/* Включение нового узла в ДДП.
   Параметры: root – указатель на корень дерева; label – метка нового узла; cmplabel – указатель на
   функцию, которая сравнивает два значения типа LABEL и возвращает 1, 0 или -1, если, соответ-
   ственно, первое больше второго, они равны или первое меньше второго; setlabel – указатель на
   функцию, которая записывает значение типа LABEL по указанному адресу.
   Возвращаемое значение: при успешном завершении – указатель на новый узел, иначе – NULL. */
NODE * add_node(NODE * root,
                LABEL label,
                int (*cmplabel)(LABEL, LABEL)
                void (*setlabel)(LABEL *, LABEL))
{ NODE * newnode;
  if ((newnode = malloc(size)) == NULL) return NULL;
  while (root != NULL)
      if ((*cmplabel)(label, root->label) == 0) return NULL;
      else
          if ((*cmplabel)(label, root->label) < 0)
              if (root->left != NULL) root = root->left;
              else
                  { q->left = newnode;
                    break; }
          else
              if (root->right != NULL) root = root->right;
              else
                  { root->right = newnode;
                    break; }
  (*setlabel)(&newnode->label, label);
  newnode->left = newnode->right = NULL;
  return newnode;
}
```

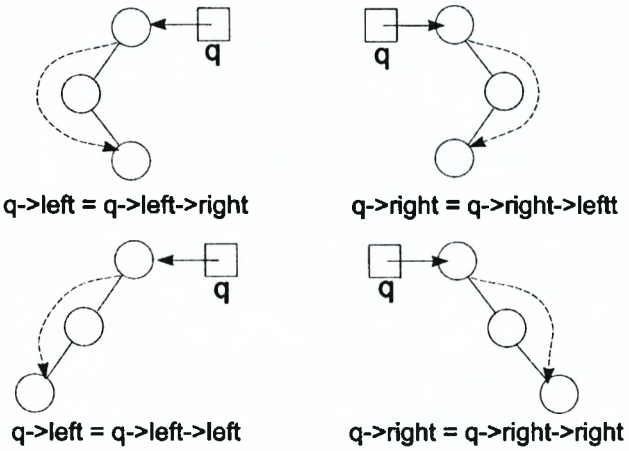
3. Удаление узла из ДДП.

Последовательность действий при выполнении операции удаления узла из ДДП зависит от того, сколько сыновей имеет удаляемый узел:

- если удаляемый узел является листом, то для его удаления достаточно обнулить соответствующую ссылку его истинного предка;
- если удаляемый узел имеет единственного сына, то последний должен стать истинным потомком прямого предка удаляемого узла (см. рис. 2.7);
- если удаляемый узел имеет двух сыновей, то его следует заменить узлом с подходящей меткой, причем у последнего должно быть не более одного сына.

Поскольку метка узла в ДДП больше метки любого его потомка из левого поддерева и меньше метки любого его потомка из правого поддерева, при удалении узла с двумя сыновьями подходящим для замены узлом будет либо узел с максимальной меткой из левого поддерева, либо с минимальной – из правого поддерева удаляемого узла. Следовательно, чтобы найти такой узел, надо перейти от удаляемого узла сначала к его левому/правому сыну, а затем – по ссылкам правого/левого сына пока не будет найден узел с нулевой правой/левой ссылкой. Информационную часть найденного таким образом узла следует сохранить во вспомогательном буфере, а сам узел удалить. Далее остается лишь восстановить содержимое буфера вместо информационной части удаляемого узла с двумя сыновьями.

Описанный алгоритм удаления узла из ДДП удобно реализовать в виде рекурсивной



функции, исходный код которой представлен в Листинге 2.12.

Рис. 2.7. Варианты удаления из ДДП узла с одним сыном (q – указатель на текущий узел).

Листинг 2.12

```
/* Удаление узла в ДДП.
   Параметры: root – указатель на указатель на корень дерева; label – метка удаляемого узла;
   cmplabel – указатель на функцию, которая сравнивает два значения типа LABEL и возвращает 1,
   0 или -1, если, соответственно, первое больше второго, они равны или первое меньше второго;
   setlabel – указатель на функцию, которая записывает значение типа LABEL по указанному адресу.
   Возвращаемое значение: при успешном завершении –1, иначе – 0. */
int del_node(NODE ** root,
             LABEL label,
             int (*cmplabel)(LABEL, LABEL)
             void (*setlabel)(LABEL *, LABEL)))
{ NODE * q = * root, * p1, * p2, *par = NULL;
  while (q != NULL) /* Поиск удаляемого узла и его прямого предка.*/
    if ((*cmplabel)(label, q->label) == 0) break;
    else
        { par = q;
```

```
        if ((*cmplabel)(label, q->label) < 0) q = q->left;
        else q = q->right; }
    if (q == NULL) return 0;
  /* Удаление листа. */
  if (((p1 = q->left) == NULL) && ((p2 = q->right) == NULL))
      if (par != NULL)
          if (par->left == q) par->left = NULL;
          else par->right = NULL;
      else
          *root = NULL;       /* Удаление корня */
  else
  /* Удаление узла с одним левым или правым сыном. */
      if ((p1 == NULL) || (p2 == NULL))
          { NODE * p;
            p = (p1 == NULL) ? p2 : p1;
            if (par != NULL)
                if (par->left == q) par->left = p; else par->right = p;
            else
              *root = p; }      /* Удаление корня.*/
  /* Удаление узла с двумя сыновьями. */
      else
          { par = q;
            for(q = q->left; q->right != NULL; q = q->right);
            (*setlabel)(&par->label, q->label);
            del_node(root, q->label, cmplabel, setlabel); }
  free(q);
  return 1;
}
```

Время выполнения алгоритма для операций поиска, вставки и удаления элементов ДДП в среднем имеет порядок $O(\log_2 N)$, где N – количество элементов. Однако такая зависимость будет иметь место только для сбалансированных ДДП.

Двоичное дерево называется **сбалансированным** тогда и только тогда, когда высоты двух поддеревьев каждой из его вершин отличаются не более чем на единицу. Деревья, удовлетворяющие такому условию, часто называют **AVL-деревьями** (по начальным буквам фамилий авторов, впервые сформулировавших указанный критерий).

Если это условие не выполняется, говорят о **вырождении ДДП**. Вырождение чаще всего является следствием манипулирования элементами ДДП, например, вставки в ДДП последовательности упорядоченных по возрастанию или убыванию данных. Вырождение может быть частичным или полным. При полном вырождении ДДП трансформируется в упорядоченный список.

С ростом степени вырождения время выполнения основных операций также увеличивается, достигая в среднем порядка $O(N)$ для полностью вырожденного ДДП. Поэтому вырожденное ДДП необходимо подвергать **процедуре балансировки**, цель которой заключается в выравнивании количества узлов в левом и правом поддеревьях для любого узла. Практически для балансировки достаточно большого по статистическим меркам ДДП, можно переписать метки его элементов в некоторый буфер, случайным образом «перемешать» и затем поэлементно загрузить во вновь созданное ДДП.

2.5. Частично упорядоченные деревья

Частично упорядоченное дерево (ЧУД) – это двоичное дерево, обладающее следующими свойствами:

- для меток узлов определено отношение порядка больше/меньше;
- метка любого узла не превышает или не меньше (одно из двух) меток всех его потомков;
- новые узлы разрешается добавлять только на самом низшем уровне глубины.

Существенным отличием ЧУД от ДДП является то, что листья в ЧУД располагаются на самом низшем или предшествующем ему уровне, поэтому такие деревья всегда будут сбалансированными. Критерий сбалансированности в данном случае можно сформулировать так: двоичное дерево является сбалансированным тогда и только тогда, когда глубина любого его листа отличается от высоты дерева не более чем на 1. Ясно, что данная формулировка эквивалентна приведенной выше.

Наиболее эффективны ЧУД для реализации очередей с приоритетами, в которых, как отмечалось выше, в любой момент времени преимущественным правом на обслуживание обладает элемент с максимальным приоритетом, а не первый, как в обычной очереди.

Рассмотрим в качестве примера очередь с приоритетами, которая возникает при распределении совместно используемых ресурсов вычислительной системы между несколькими одновременно выполняющимися процессами. Машинное время выделяется процессам квантами, поэтому, если некоторый процесс не завершился за один квант времени, обслуживание этого процесса может быть прервано, если приоритет другого процесса к этому моменту времени стал выше, чем приоритет обслуживаемого процесса.

Обычно процессы, для выполнения которых требуется меньше времени, имеют более высокие приоритеты на системные ресурсы, чем те процессы, для выполнения которых требуется значительное машинное время. Однако в таком случае более продолжительные процессы могут оказаться заблокированными, поэтому приоритет процесса определяется как функция времени:

$$P = C\, t_{exec} - t, \qquad (2.2)$$

где t_{exec} – время, уже выделенное процессу, t – время, прошедшее с момента инициализации процесса. Константа C определяет процент машинного времени, выделяемого процессам в среднем за некоторый достаточно продолжительный промежуток времени, и должна быть несколько больше числа ожидаемых активных процессов (как правило, выбирается эмпирически найденное значение C = 100).

Таким образом, приоритет процесса тем выше, чем меньше значение величины P. Если процессу в течение длительного промежутка времени не выделяется машинное время, то его приоритет растет (P становится большим отрицательным числом), и процесс, в конце концов, получает необходимые для его завершения машинные ресурсы.

Задача реализации очередей с приоритетами посредством ЧУД заключается в том, чтобы сформировать из элементов очереди с приоритетами сбалансированное двоичное дерево с частично упорядоченными узлами. Частичная упорядоченность такого дерева вытекает из требования, чтобы приоритет любого узла был не ниже приоритета его сыновей, а новый лист может быть добавлен только на самый нижний уровень справа от имеющихся на этом уровне листьев или, если этот уровень полностью заполнен, – в начало нового уровня. Тогда корень дерева будет соответствовать элементу очереди с максимальным приоритетом.

Основными операциями в случае реализации очередей с приоритетами посредством ЧУД являются:

1. Поиск крайнего справа листа на последнем уровне.

Чтобы найти крайний справа лист на последнем уровне, надо сначала найти крайний слева лист, так как этот лист в ЧУД всегда будет находиться на последнем уровне, и затем, выполняя рекурсивно операцию поиска первого справа соседа, найти лист, у которого нет соседа справа.

2. Вставка нового узла в дерево.

Эта операция выполняется в два шага.

Сначала новый узел надо установить в первую свободную позицию справа на последнем уровне (рис. 2.8,а) или крайнюю левую позицию на новом уровне, если последний уровень заполнен полностью (на полностью заполненном уровне располагается 2^i узлов, где i = 0,1,2,... – номер уровня). Если последний уровень заполнен полностью, то истинным предком для нового узла станет первый слева лист на последнем уровне глубины. В противном случае истинным предком нового узла станет либо истинный предок крайнего справа листа, либо первый справа сосед истинного предка крайнего справа листа, если крайний справа лист является левым или правым, соответственно, сыном своего истинного предка.

На втором шаге необходимо проверить, не нарушена ли частичная упорядоченность узлов дерева, т.е. не превышает ли приоритет нового узла приоритет его истинного предка. Если это действительно так, то новый узел и его истинный предок меняются местами (рис. 2.8,б – г). Эта процедура выполняется рекурсивно до тех пор, пока новый узел не займет место, в котором его приоритет не будет превышать приоритет его истинного предка или он не станет корнем дерева.

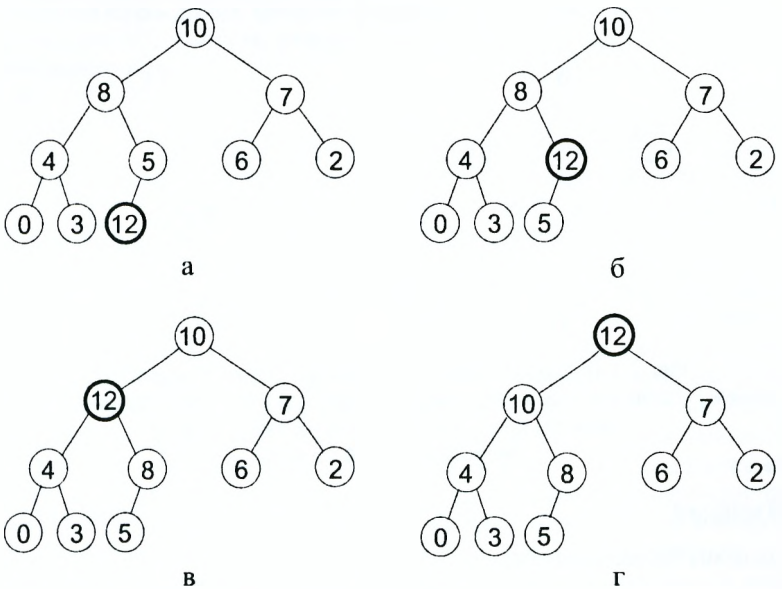


Рис. 2.8. Последовательность шагов при вставке узла в ЧУД: а – добавление нового узла на последний уровень; б,в,г – новый узел «поднимается» вверх по ЧУД пока его метка превышает метку его истинного предка.

3. Удаление корня дерева.

Чтобы удалить корень ЧУД, следует найти крайний справа узел на последнем уровне (рис. 2.9,а) и подставить его вместо корня (рис. 2.9,б). Поскольку этот узел имеет один из самых низких приоритетов, его следует поменять местами с тем из сыновей корня, у которого больший приоритет (рис. 2.9,в). Как и при вставке узла, процедура обмена узлов

выполняется рекурсивно до тех пор, пока очередной узел не станет листом или его приоритет будет не ниже, чем у его сыновей (рис. 2.9,г).

Поскольку ЧУД всегда сбалансированы, время выполнения основных операций по порядку величины оказывается не хуже $\log_2 N$, где N – количество узлов, т.е. в худшем случае, а не в среднем как для ДДП.

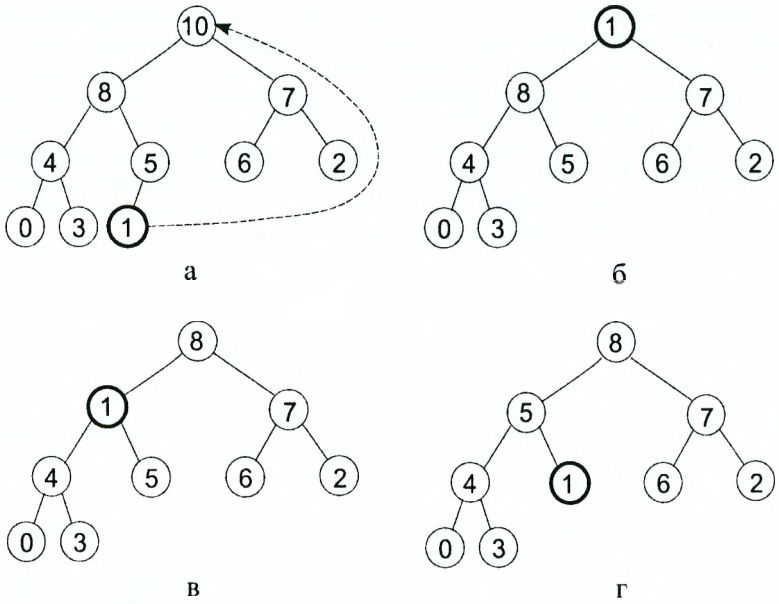


Рис. 2.9. Последовательность шагов при удалении корня ЧУД:
а – крайний справа лист на последнем уровне подставляется вместо корня ЧУД; б,в,г – новый корень «спускается» вниз пока его метка меньше меток его сыновей.

УПРАЖНЕНИЯ

1. Построить дерево выражения
   $$1 + x*(2 - \ln(x)) + 7*(x/3 - \sin(x)*(1 - 2*x))$$
   и получить его префиксную и постфиксную формы записи без скобок.
2. Определить функции, реализующие прямой и обратный способы обхода узлов дерева арифметического выражения.
3. Для вычисления значения выражения, представленного в постфиксной форме, можно использовать стек: если очередной считываемый операнд выражения представляет собой константу или переменную, то этот операнд заносится в стек; если же считываемый элемент – символ операции, то эта операция выполняется, а в вершину стека заносится результат операции. Определить функцию, в результате выпол-

нения которой вычисляется значение арифметического выражения в постфиксной форме для заданного значения аргумента плавающего типа. Исходное выражение должно передаваться в функцию в виде строки символов.

4. Определить функции для построения дерева логического выражения и обхода его узлов в прямом, симметричном и обратном порядке.
5. Дан файл, содержащий текст некоторого сообщения длиной N ($N \geq 100$) символов кириллического набора. Рассчитать вероятности появления отдельных символов в данном сообщении, построить дерево кодов Хаффмана и определить среднюю длину полученного кода. Определить функции для кодирования и декодирования произвольных сообщений, состоящих из символов того же набора.
6. Степень вырождения ДДП можно приближенно оценить как отношение количества листьев, глубина которых отличается от усредненной по всему ДДП глубины листьев более чем на 1, к общему количеству листьев в ДДП. Определить функцию, которая вычисляет степень вырождения ДДП и выполняет балансировку ДДП, если степень вырождения превышает 0,5.
7. Определить функцию, реализующую процедуру обхода узлов ДДП в порядке уровней, т.е. сначала корень дерева, затем слева направо узлы глубины 1, глубины 2 и т.д.
8. Реализовать АТД «частично упорядоченное дерево», используя в ссылочной части узлов указатели на:
   а) сыновей и прямого предка;
   б) сыновей и соседа справа.
9. Если $j=1, \ldots, 2^k$ – номер узла, расположенного в ЧУД на уровне k, то его сыновья (если таковые имеются) будут располагаться на уровне k + 1 под номерами 2j – 1 и 2j. Реализовать АТД «частично упорядоченное дерево» посредством одномерного динамического массива, в котором узлы располагаются по уровням слева направо.
10. Алгоритм турнирной (пирамидальной) сортировки заключается в следующем: элементы сортируемой последовательности разбиваются на пары, и «победитель» (меньший или больший из двух элементов в зависимости от заданного отношения порядка) выходит в следующий тур, и так далее до финального тура, «победитель» которого будет наименьшим элементом в упорядоченной последовательности. Далее всем элементам исходной последовательности с найденным значением присваивается значение абсолютного минимума для данного типа данных, а сами они переписываются во вспомогательную упорядоченную последовательность. Так продолжается до тех пор, пока в исходной последовательности не останется единственный элемент. Определить функцию, реализующую алгоритм турнирной (пирамидальной) сортировки целочисленного массива путем построения ЧУД. Время выполнения алгоритма должно быть порядка $O(N\log_2 N)$.
11. K-ой порядковой статистикой называется элемент заданной совокупности, который при сортировке этой совокупности в порядке неубывания значений элементов занял бы точно k-ую позицию. Определить функцию, которая находит k-ую порядковую статистику в совокупности целых чисел, хранящихся во внешнем файле, путем организации данных в виде ЧУД. Показать, что время выполнения алгоритма при $k<<N/\log_2 N$ имеет порядок $O(N)$.
12. Написать программу, имитирующую распределение машинного времени в вычислительной системе между процессами в соответствии с их приоритетами. Частота запуска процессов и необходимое для их выполнения машинное время выбираются случайным образом.

3. МНОЖЕСТВА

3.1. Основные понятия теории множеств

Всякое множество полностью определяется своими элементами. Утверждение, что множество A состоит из элементов $a_1, a_2, \ldots, a_n$ условно записывается так

$$A = \{a_1, a_2, \ldots, a_n\},$$

причем количество элементов множества называется **мощностью множества**.

Единичное (одноэлементное) **множество** содержит только один элемент, например, {a}.

Пустое множество (обозначается символом ∅) – множество, которое не содержит ни одного элемента. Роль пустого множества аналогична роли числа 0.

С понятием множества тесно связано **понятие отношения**. Интуитивно отношение означает связь элементов данного множества друг с другом или с элементами других множеств. Сама по себе принадлежность элементов множеству устанавливает связь между ними. Эта связь определяется отношением принадлежности, которое обозначается символом «∈», а альтернативное свойство – символом «∉», например: $a \in A$, $b \notin A$.

Бинарное отношение – отношение между парами элементов одного и того же множества или двух разных множеств. Если некоторые элементы a и b находятся в бинарном отношении R, то этот факт записывается в виде выражения aRb или $(a, b) \in R$. Ясно, что бинарное отношение полностью определяется множеством всех пар элементов, для которых это отношение устанавливает связь, поэтому любое бинарное отношение R можно рассматривать как множество упорядоченных пар $(a, b) \in R$. Примерами бинарных отношений являются отношения равенства, порядка, включения и т.д.

Два **множества** A и B **равны**, если для их элементов определено отношение равенства, т.е. тогда и только тогда, когда каждый элемент A является также и элементом B и наоборот. Равенство двух множеств A и B обозначается A = B, неравенство – $A \neq B$.

Частично упорядоченное множество – множество, для некоторых пар элементов которого определено отношение порядка (правило предшествования) $a \leq b$ такое, что $a \leq a$, из $a \leq b$ и $b \leq c$ следует $a \leq c$, из $a \leq b$ и $b \leq a$ следует $a = b$.

Линейно упорядоченное множество – множество, для любых двух элементов a и b которого определено отношение порядка $a \leq b$ или $b \leq a$, обладающее перечисленными выше свойствами.

Если все элементы множества A принадлежат и множеству B, то считается, что для A и B определено отношение включения или множество A является **подмножеством множества** B. Отношение включения обозначается $A \subset B$, т.е. «A включено в B». Любое множество A имеет всегда по крайней мере два подмножества – само это множество и пустое множество, т.е. $A \subset A$ и $\varnothing \subset A$. Эти **подмножества** называются **несобственными**, а все другие подмножества непустого множества, образуемые всевозможными сочетаниями по одному, два и т.д. элементов, – **собственными подмножествами**.

Множество, элементами которого являются все подмножества множества A, называется **множеством подмножеств**. Например, для множества {a, b, c} множество подмножеств есть множество {∅, {a}, {b}, {c}, {a,b}, {a,c}, {b,c}, {a,b,c}}.

Для конечного множества, состоящего из n элементов, его множество подмножеств состоит из 2^n элементов. Действительно, если представить подмножества n-разрядными двоичными числами, в которых 1 обозначает вхождение некоторого элемента в данное подмножество, а 0 – его отсутствие во множестве, то количество таких подмножеств будет, очевидно, равно количеству всевозможных комбинаций 0 и 1 в таком n-разрядном двоичном числе, т.е. всего 2^n.

Как правило, любое множество является подмножеством некоторой совокупности допустимых объектов, например, для множества чисел Фибоначчи совокупностью допус-

тимых объектов будет множество целых чисел. Такая совокупность допустимых объектов для всех множеств, являющихся ее подмножествами, называется **основным множеством** или **универсумом** и обозначается обычно буквой U.

Для графического представления отношений между множествами какого-либо универсума используются **круги Эйлера**. Универсум изображается прямоугольником, а содержащиеся в нем множества – кругами внутри этого прямоугольника. Например, на рис. 3.1 с помощью кругов Эйлера представлено отношение включения между множествами A и B.

Основные операции над множествами – действия, с помощью которых из одного, двух, нескольких или бесконечной совокупности множеств формируется одно множество. К основным операциям над множествами относятся: объединение (сумма), пересечение (произведение), разность, дизъюнктивная сумма (симметрическая разность) и прямое (декартово) произведение.

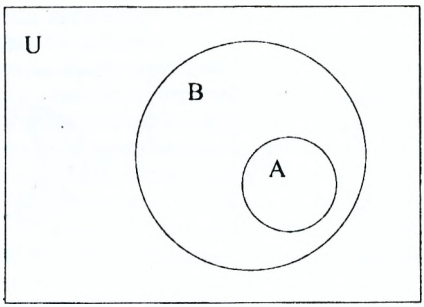


Рис. 3.1. Графическое представление отношения включения $A \subset B$ с помощью кругов Эйлера.

Объединение непустого семейства множеств A_i (i=1, 2, ..., n) – множество, состоящее из элементов, принадлежащих хотя бы одному из множеств A_i. Объединение обозначается символом «$\cup$»: $\bigcup_i^n A_i$. Объединение двух множеств A и B обозначается $A \cup B$ (рис. 3.2,а).

Пересечение непустой совокупности множеств A_i (i=1, 2, ..., n) – множество, состоящее из элементов, принадлежащих одновременно всем множествам A_i. Пересечение обозначается символом «$\cap$»: $\bigcap_i^n A_i$. Пересечение двух множеств A и B обозначается $A \cap B$ (рис. 3.2,б).

Разность множеств A и B – множество, состоящее из элементов, принадлежащих множеству A, но не принадлежащих множеству B. Разность множеств A и B обозначается $A \setminus B$ или $A - B$ (рис. 3.2,в). Разность можно рассматривать как относительное дополнение B до A. Если U есть универсум множества A, то множество $U \setminus A$ называется **абсолютным дополнением** или просто дополнением множества A.

Дизъюнктивная сумма или **симметрическая разность** множеств A и B есть множество, которое получается объединением множеств A и B за исключением тех элементов, которые принадлежат и одному и другому множеству. Дизъюнктивная сумма обозначается $A + B$ или $A \oplus B$ (рис. 3.2,г).

Декартово произведение непустой совокупности множеств A_i (i=1, 2, ..., n) представляет собой множество упорядоченных совокупностей элементов $(a_1,a_2, ..., a_n)$, называемых кортежами, таких что $a_1 \in A_1$, $a_2 \in A_2$, ..., $a_n \in A_n$. Декартово произведение обозначается символом «X», например, если $A = \{a_1, a_2, a_3\}$ и $B = \{b_1, b_2\}$, то $A \times B = \{(a_1, b_1), (a_1, b_2), (a_2, b_1), (a_2, b_2), (a_3, b_1), (a_3, b_2)\}$. Порядок следования кортежей может быть любым, но порядок следования элементов в каждом кортеже должен соответствовать порядку следования множеств в записи декартова произведения.

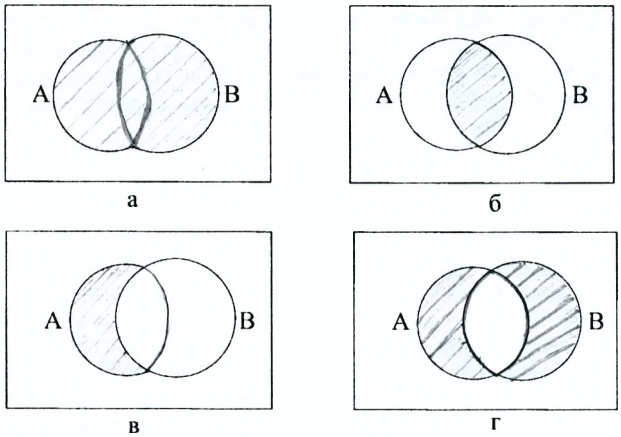


Рис. 3.2. Представление основных операций над множествами с помощью кругов Эйлера: объединение $A \cup B$ (а), пересечение $A \cap B$ (б), разность $A \setminus B$ (в), дизъюнктивная сумма $A + B$ (г). Заливкой указаны области, соответствующие результатам операций.

Множество можно задать перечислением его элементов или посредством определяющего свойства P(x) (форма от x). Обычно P(x) – это высказывание, в котором содержится некоторое утверждение об x или некоторая функция переменной x, например, «x – число Фибоначчи» или P(x) = ln(x) и т.п. Множество, заданное с помощью формы P(x), обозначается $X = \{x \mid P(x)\}$, причем $a \in X$, если значение P(a) определено, т.е. соответствующее утверждение истинно или функция определена. Например, перечисленные выше операции над множествами A и B можно определить следующим образом:

$A \cup B = \{x \mid x \in A$ или $x \in B\}$

$A \cap B = \{x \mid x \in A$ и $x \in B\}$

$A \setminus B = \{x \mid x \in A$ и $x \notin B\}$

$A + B = \{x \mid (x \in A$ и $x \notin B)$ или $(x \in B$ и $x \notin A)\}$

3.2. АТД «множество»

АТД «множество» базируется на математической трактовке этого понятия, т.е. значениями АТД в данном случае являются множества, для которых независимо от их реализации определен перечень операций с общепринятыми названиями:

- Union(A, B) – объединение множеств A и B;
- Intersection(A, B) – пересечение множеств A и B;

- Difference(A, B) – разность множеств A и B;
- Merge(A, B, C) – объединение во множестве C непересекающихся множеств A и B, т.е. если $A \cap B = \varnothing$, и результат операции не определен, если $A \cap B \neq \varnothing$;
- Member(a, A) – проверка на принадлежность элемента a множеству A, т.е. $a \in A$ или $a \notin A$;
- Insert(a, A) – вставка элемента a в множество A;
- Delete(a, A) – удаление элемента a из множества A;
- Equal(A, B) – проверка на совпадение (равенство) двух множеств, т.е. $A = B$ или $A \neq B$, соответственно;
- Assign(A, B) – присваивание множеству A значение множества B;
- MakeNull(A) – преобразование множества A в пустое множество;
- Empty(A) – проверка множества A на равенство пустому множеству, т.е. $A = \varnothing$ или $A \neq \varnothing$;
- Max(A) – поиск наибольшего элемента множества A;
- Min(A) – поиск наименьшего элемента множества A;
- Find(a, SetArray) – поиск множества из массива SetArray непересекающихся множеств, в котором содержится элемент a.

Реализация перечисленных выше операций определяется способом представления множеств, что в свою очередь имплицируется типом их элементов и мощностью.

Способы представления множеств можно условно разделить на два типа. К первому типу отнесем те методы представления множеств, для которых время выполнения операций манипулирования элементами (Insert, Delete, Member) фиксировано, т.е. не зависит от мощности множества. Эти методы основаны в основном на использовании функциональных структур данных, в которых положение элемента однозначно определяется значением этого элемента. Указанные свойства присущи представлению множеств посредством двоичных (булевых) векторов и перемешанных таблиц.

Ко второму типу отнесем те методы представления множеств, для которых характерна зависимость времени выполнения операций от мощности множества в результате того, что объекты (или их ключи), удовлетворяющие установленному для данного множества отношению, помещаются в специально созданную для этого структуру, как правило, рекурсивного типа. Указанное свойство данного подхода к представлению множеств является его основным недостатком, однако этот недостаток может компенсироваться, как будет показано ниже, более эффективным использованием памяти. Последнее обстоятельство может оказаться решающим при выборе метода реализации для больших по мощности множеств, для которых размер физической памяти, необходимой для хранения их элементов, является существенно более весомым фактором, чем возможность выполнения нескольких операций за фиксированное время.

3.3. Представление множеств посредством двоичных векторов

Представление множеств посредством двоичных векторов, по сути, заключается в построении некоторой функции, которая каждому объекту из числа допустимых для данного множества ставит в соответствие единственный элемент из двухэлементного булевского множества, т.е. множества вида {0,1}, так что значение 1 указывает на принадлежность данного объекта множеству, а значение 0 на альтернативное свойство. Другими словами, при представлении множества подмножеств некоторого конечного универсума U двоичными N-мерными векторами (массивами) каждому элементу универсума ставится в соответствие единственный элемент вектора, значение которого равно 1, если данный элемент принадлежит множеству, или 0 в противном случае. Тогда пустому множеству будет соответствовать вектор, у которого все элементы равны 0, а множест-

ву, равному всему универсуму, – вектор, у которого все элементы равны 1. Такое представление, очевидно, приемлемо только для небольших конечных универсумов.

Если тип элементов U относится к перечисляемым типам данных и a – элемент множества, представленного двоичным вектором V, то V[f(a)] = 1, иначе V[f(a)] = 0. Здесь f – некоторая частично-рекурсивная функция, называемая функцией сдвига, которая отображает U на множество положительных целых чисел M = {0, 1, 2, ..., N – 1}. В простейшем случае функция сдвига имеет вид: f(a) = a – min(U) для любого a∈U, где min(U) – минимальное из допустимых значений элементов множества. Например, для множества печатных символов ASCII-набора U = {32, 33, ..., 255} и, соответственно, функция сдвига – f(a) = a – 32.

В Листинге 3.1 приведен пример реализации АТД «множество» при представлении двоичными векторами. Рассматривая далее различные варианты реализации АТД «множество», тип его значений будем обозначать как SET, а тип элементов множества – как ITEM.

Листинг 3.1

```
/* Объявление типа SET. */
typedef char SET[N];

/* Проверка на принадлежность элемента множеству.
   Параметры: a – элемент; A – исходное множество; f – указатель на функцию, отображающую
   элементы множества на множество целых положительных чисел.
   Возвращаемое значение: 1, если a ∈ A, иначе – 0. */
int Member(ITEM a, SET A, unsigned (* f)(ITEM))
{ return A[(*f)(a)]; }

/* Вставка элемента во множество.
   Параметры: a – элемент; A – исходное множество; f – указатель на функцию, отображающую элементы
   множества на множество целых положительных чисел.
   Возвращаемое значение: 1 при успешном завершении операции, иначе – 0. */

int Insert(ITEM a, SET A, unsigned (* f)(ITEM))
{ unsigned i = (*f)(a);
  if (A[i]) return 0; else return ++ A[i]; }
}

/* Удаление элемента из множества.
   Параметры: a – элемент; A – исходное множество; f – указатель на функцию, отображающую
   элементы множества на множество целых положительных чисел.
   Возвращаемое значение: 1 при успешном завершении операции, иначе – 0. */
int Delete(ITEM a, SET A, unsigned (* f)(ITEM))
{ unsigned i = (*f)(a);
  if (A[i] == 0) return 0; else return A[i] —; }
}

/* Объединение множеств.
   Параметры: A, B – исходные множества.
   Возвращаемое значение: при успешном завершении – указатель на результирующее множество,
   иначе – NULL. */
SET * Union(SET A, SET B)
{ SET * ptrset, resset;
  unsigned i, n = sizeof(SET);
  if ((ptrset = (SET *) malloc(n)) == NULL) return NULL;
  for (i = 0, resset = *ptrset; i < n; i++) resset[i] = A[i] || B[i];
  return ptrset;
}
```

```
/* Пересечение множеств.
   Параметры: A, B – исходные множества.
   Возвращаемое значение: при успешном завершении – указатель на результирующее множество,
   иначе – NULL. */
SET * Intersection(SET A, SET B)
{ SET * ptrset, resset;
  unsigned i, n = sizeof(SET);
  if ((ptrset = (SET *) malloc(n)) == NULL) return NULL;
  for (i = 0, resset = *ptrset; i < n; i++) resset[i] = A[i] && B[i];
  return ptrset;
}
/* Разность множеств.
   Параметры: A, B – исходные множества.
   Возвращаемое значение: при успешном завершении – указатель на результирующее множество,
   иначе – NULL.. */
SET* Difference(SET A, SET B)
{ SET * ptrset, resset;
  unsigned i, n = sizeof(SET);
  if ((ptrset = (SET *) malloc(n)) == NULL) return NULL;
  for (i = 0, resset = *ptrset; i < n; i++) resset[i] = A[i] && ! B[i];
  return ptrset;
}
/* Проверка на пустое множество.
   Параметры: A – исходное множество.
   Возвращаемое значение: 1, если A = ∅, иначе – 0. */
int Empty(SET A)
{ unsigned i;
  for (i = 0; i < sizeof(SET); i++)
      if (A[i] == 1) return 0;
  return 1;
}
/* Объединение непересекающихся множеств.
   Параметры: A, B – исходные множества.
   Возвращаемое значение: указатель на результирующее множество, если A ∩ B = ∅, иначе –
   NULL. */
SET* Merge(SET A, SET B)
{ extern Empty(SET);
  extern SET * Union(SET, SET);
  extern SET * Intersection(SET, SET);
  if (Empty(Intersection(A,B))) return Union(A, B);
  else return NULL;
}
```

Основным недостатком векторного представления множеств является зависимость размера вектора не от количества элементов во множестве, а от значения максимального по величине элемента. Если это значение достаточно велико, или тип элементов множества вообще не может быть определен как перечислимый, необходимо установить взаимно однозначное соответствие между ними и множеством неотрицательных целых

чисел, т.е. определить множество пар вида (a, i), где $a \in U$, $i \in M$. В ряде случаев такое соответствие можно установить с помощью некоторой функции F(a), такой, что $F(a) \in M$ для любого $a \in U$. Однако не всегда удается подобрать функцию так, чтобы $F(a_i) \neq F(a_j)$ для любых двух элементов $a_i \neq a_j$. Поэтому для реализации множественных типов данных с фиксированным временем выполнения основных операций чаще применяют перемешанные таблицы.

3.4. Представление множеств посредством перемешанных таблиц

Суть метода перемешанных таблиц или хеширования (от англ. hash – перемешивать) заключается в том, что вводится некоторая функция H(x), которая по возможности равномерно отображает множество элементов мощности N на конечное множество K целых положительных чисел от 0 до некоторого K – 1. В теории множеств, как было отмечено выше, подобные отображения называются отношениями, а в теории алгоритмов – таблицами. В создаваемых таким образом таблицах элементу a множества будет соответствовать ячейка с номером H(a), т.е. расположение элементов в таблице определяется только функцией H(x) и они оказываются как бы перемешаны, поэтому такие таблицы называются перемешанными, а функция H(x) – функцией расстановки или хеш-функцией. В общем случае из того, что $x_1 < x_2$, не следует, что $H(x_1) < H(x_2)$, более того для двух различных элементов множества x_1 и x_2 может оказаться, что $H(x_1) = H(x_2)$ – такая ситуация называется коллизией.

В ячейки перемешанных таблиц обычно записываются ключи записей, объединяемых во множество, в то время как сами записи хранятся на некотором запоминающем устройстве, как правило, прямого доступа. Другими словами, в ячейку с номером i будет записан ключ k того элемента, для которого H(k) = i, и, возможно, адрес этого элемента, если необходимо обеспечить быстрый доступ к самим записям.

Существуют различные классификации методов хеширования, но обычно рассматривают **два вида хеширования – прямое** (закрытое, внутреннее) и **расширенное** (открытое, внешнее), отличающиеся способами разрешения коллизий.

3.4.1. Прямое хеширование

При прямом хешировании для разрешения коллизий вводится дополнительная функция, называемая функцией повторного хеширования и представляющая собой частично-рекурсивную функцию R(t) такую, что $R(t) \in [0, K-1]$ при $t \in [0, K-1]$. С помощью функции повторного хеширования в случае возникновения коллизии производится переопределение значения, возвращаемого хеш-функцией. Если коллизия возникла при добавлении нового элемента x в хеш-таблицу, т.е. в ячейке с номером $n_0 = H(x)$ уже содержится некоторый элемент, последовательно просматриваются ячейки, номера которых вычисляются по рекуррентной формуле $n_i = R(n_{i-1})$, $i = 1, 2, \ldots$, $n_0 = H(x)$, пока не будет найдена первая свободная ячейка и коллизия, таким образом, не будет устранена.

Аналогично производится поиск элемента x в хеш-таблице, т.е. проверка на принадлежность элемента множеству. Если в ячейке с номером $n_0 = H(x)$ содержится искомый элемент, то поиск завершается с положительным результатом, иначе проверяются ячейки с номерами, вычисляемыми по приведенной выше рекуррентной формуле, до тех пор, пока не будет найдена ячейка с таким элементом или очередная ячейка не окажется пустой. Если из хеш-таблицы элементы не удалялись, то пустая ячейка в этом случае означает отсутствие элемента во множестве и поиск также завершается, но уже с отрицательным результатом. Но если элементы удалялись, то пустая ячейка может быть одной из тех ячеек, в ко-

торых содержались элементы, удаленные после вставки искомого элемента. Поэтому при удалении элементов из хеш-таблицы следует помечать соответствующие ячейки так, чтобы при добавлении элементов эти ячейки трактовались как пустые, а при поиске – как занятые. Если же пометить ячейки непосредственно после удаления элементов нельзя, то следует применять метод расширенного хеширования.

Если значения хеш-функции равновероятны для всех элементов множества, соответствующая хеш-таблица будет заполнена равномерно и коллизий практически не будет. Но, как правило, избежать коллизий не удается, поэтому размер хеш-таблицы выбирается несколько большим, чем мощность множества. За счет этой избыточности, которая при удачном выборе хеш-функции не превышает нескольких процентов, обеспечивается размещение в хеш-таблице всех возможных элементов множества.

Характер распределения элементов в хеш-таблице также зависит и от выбора функции повторного хеширования. Часто в качестве такой функции выбирается линейная функция вида

$$R(t) = (t+1) \% K, \qquad (3.1)$$

т.е. при возникновении коллизии просматривается следующая ячейка, а если коллизия вызвана ячейкой с номером K – 1, то следующей будет ячейка с номером 0. Однако недостатком линейной функции повторного хеширования является то, что при ее применении в хеш-таблице образуются группы подряд стоящих заполненных ячеек, которые приводят к увеличению количества просматриваемых ячеек и, соответственно, замедлению выполнения операций.

Более равномерное распределение элементов в хеш-таблице можно получить при использовании случайной функции повторного хеширования, представляющей собой последовательность попарно различных случайных целых чисел из диапазона от 0 до K – 1. Такую последовательность можно сгенерировать при создании хеш-таблицы и сохранить или каждый раз воспроизводить, инициализируя генератор одним и тем же начальным значением. Однако сгенерировать последовательность случайных чисел, в которой не было бы повторяющихся значений достаточно трудно. К сожалению, генератор псевдослучайных чисел из библиотеки системы программирования C не удовлетворяет этому требованию, поэтому необходимо использовать другие методы для генерации псевдослучайных чисел. Одним из таких методов является **метод последовательных сдвигов регистра**. Если необходимо сгенерировать последовательность K случайных целых чисел в диапазоне [0, K – 1], алгоритм этого метода можно представить рекуррентной формулой:

$$a_i = (a_{i-1} 2^m - K) \wedge C, \qquad (3.2)$$

где i = 1, ...K–1, C – константа из интервала от 1 до K – 1, m – наименьшее натуральное число такое, что $a_{i-1} 2^m > K$, «^» – знак операции побитового суммирования по модулю 2. Начальное значение a_0 выбирается произвольно из того же интервала от 1 до K – 1.

Особенностью этого алгоритма является то, что, во-первых, он применим только для значений K, являющихся степенью числа 2, и, во-вторых, только для определенных значений константы C можно получить все числа от 1 до K – 1 без повторов, для остальных значений C числа в сгенерированной последовательности могут повторяться.

Выбор хеш-функции во многом определяется типом хешируемых данных. Однако в любом случае область ее значений должна покрывать диапазон от 0 до K – 1, поэтому для целых чисел, например, в качестве хеш-функции обычно используется функция

$$H(x) = |x| \% K, \qquad (3.3)$$

т.е. значение функции есть остаток от деления модуля аргумента на K. На рис. 3.3 приведена схема заполнения хеш-таблицы для множества целых чисел {77, 112, 301, 23, 12, 417, 355, 0, 99, 274} с использованием хеш-функции (3.3) и линейной функции повторного хеширования.

x	H(x)	n_1	n_2
77	5		
112	4		
301	1		
23	11		
12	0		
417	9		
355	7		
0	0	1	2
99	3		
274	10		

Рис. 3.3. Схема заполнения хеш-таблицы для множества целых чисел с использованием хеш-функции (3.3) и линейной функции повторного хеширования; N = 10, K = 12.

Для строк символов хеш-функция, как правило, параметризируется количеством символов в строке и их ASCII-кодами. Если появление различных символов в строках равновероятно, можно использовать хеш-функцию вида

$$H(s) = \left(\sum_{i=1}^{L} iC_i \right) \% K, \qquad (3.4)$$

где L – длина строки s в байтах, C_i – ASCII-коды символов. В противном случае более «удачным» выбором для хэш-функции может оказаться функция

$$H(s) = \left(\sum_{i=1}^{L} ip_iC_i \right) \% K, \qquad (3.5)$$

где p_i – вероятности символов.

Для сравнения функций (3.4) и (3.5) сгенерируем случайным образом множество строк, используя для этого пять некоторых символов с вероятностями 0,10, 0,15, 0,20, 0,25, 0,30. Выбор символов, как и длина строк, в этом случае несущественен, поэтому возьмем символы с ASCII-кодами от 100 до 104, т.е. d, e, f, g и h, а длину каждой строки ограничим 10 символами. На рис. 3.4 показана схема заполнения хеш-таблицы элементами полученного таким образом множества, состоящего из 10 строк, с использованием функций (3.4) и (3.5). В обоих случаях использовалась случайная функция повторного хеширования и стандартный генератор из библиотеки языка C, а размер хеш-таблицы, как и в предыдущем примере, был равен 12.

s	H(s)	n_1	n_2	n_3	n_4	n_5	n_6	n_7	n_8	n_9	n_{10}	n_{11}	...	n_{20}
hgdfde	0													
hg	10	9												
gg	9	9	2											
hhfhh	6													
fgfgg	5													
ghfg	9	9	2	7										
fhhdggdd	7	9	2	7	8									
hdehgg	8	9	2	7	8	4								
hgdfd	6	9	2	7	8	4	6	2	10	3				
gehhfhh	3	9	2	7	8	4	6	2	10	3	0	9	...	1

а

s	H(s)	n_1	n_2	n_3	n_4	n_5	n_6	n_7	n_8	n_9
hgdfde	11									
hg	6									
gg	4									
hhfhh	8									
fgfgg	6	9								
ghfg	4	9	2							
fhhdggdd	5									
hdehgg	10									
hgdfd	11	9	2	7						
gehhfhh	10	9	2	7	8	4	6	2	10	3

б

Рис. 3.4. Схема заполнения хеш-таблицы для множества строк с использованием функций хеширования (3.4) (а), (3.5) (б) и случайной функции повторного хеширования; N = 10, K = 12.

Как видно из рис. 3.4, если число занятых ячеек не превышает 50% от общего их числа, число коллизий невелико и в обоих случаях приблизительно одинаково, но по мере заполнения хеш-таблицы число коллизий при использовании функции (3.4) растет значительно быстрее. Число коллизий в расчете на одну ячейку при использовании функции (3.4) равно 3,66, а при использовании функции (3.5) – 1,25. Таким образом, функция (3.5) в среднем дает более равномерное распределение элементов по ячейкам хеш-таблицы чем функция (3.4).

Отношение числа занятых ячеек к общему их количеству называется **коэффициентом заполнения хеш-таблицы**. Именно эта величина, а не мощность множества, определяет среднее число коллизий и, следовательно, среднее число просматриваемых ячеек при выполнении операций над элементами хеш-таблицы.

При условии равномерного распределения элементов множества по ячейкам хеш-таблицы среднее число просматриваемых ячеек при выполнении операции вставки или поиска элемента, которого нет во множестве, определяется по формуле:

$$\frac{1}{1-\alpha}, \tag{3.6}$$

а при выполнении операции удаления или поиска присутствующего во множестве элемента – по формуле

$$-\frac{1}{\alpha}\ln(1-\alpha), \tag{3.7}$$

где α – коэффициент заполнения хеш-таблицы. Для удаления элемента из множества в среднем необходимо просмотреть больше ячеек, чем при вставке. Это объясняется тем, что при удалении просматриваются также и ячейки, из которых элементы уже были удалены, а при вставке поиск продолжается только до первой пустой ячейки независимо от того, удалялся ли ранее из этой ячейки элемент или нет.

Из формул (3.6), (3.7) видно, что чем больше α, тем больше ячеек приходится просматривать. При $\alpha > 0{,}8$ среднее число просматриваемых ячеек и при вставке, и при удалении элементов резко возрастает, поэтому, чтобы время выполнения основных операций над элементами хеш-таблиц оставалось фиксированным, при $\alpha \approx 0{,}9$ необходимо провести **реструктуризацию хеш-таблицы**. Для этого необходимо создать новую хеш-таблицу, в которой количество ячеек как минимум в два раза превышает количество ячеек в преобразуемой хеш-таблице, и затем для каждого элемента из старой хеш-таблицы выполнить операцию вставки в новую хеш-таблицу. Время, затраченное на реструктуризацию хеш-таблицы, с избытком компенсируется многократным сокращением времени выполнения операций, поскольку при реструктуризации фактор заполнения хеш-таблицы уменьшается сразу в несколько раз.

С учетом описанных выше особенностей метода прямого хеширования значения типа SET удобно определять как динамические массивы с элементами типа

```
typedef struct
{ char state;  /* Состояние ячейки хеш-таблицы. */
  KEY key;     /* Ключ хешируемой записи. */.
} CEL;
```

При этом в поле state ячеек хеш-таблицы будем записывать –1, 0 или 1, если, соответственно, данные из ячейки удалены, ячейка пустая и запись в нее не производилась, в ячейке содержится некоторое значение. Таким образом, перед записью данных в хеш-таблицу во всех ее ячейках поле state должно быть проинициализировано нулем.

В Листинге 3.2 приведены исходные коды функций, реализующих основные операции над множествами, представленными посредством перемешанных таблиц с прямым хешированием. Для простоты предполагается, что для разных множеств используются одни и те же хеш-функция и функция повторного хеширования.

Листинг 3.2

```
/* Объявление типа SET и типов указателей: ptrH – на хеш-функцию, ptrR – на функцию повторного хеширования, ptrCMP – на функцию сравнения двух значений типа KEY на равенство (возвращает 1, если значения равны, и 0 в противном случае), ptrSetVal – на функцию, записывающую значение типа KEY по указанному адресу. */
typedef CEL * SET;
typedef unsigned (*ptrH)(KEY, unsigned);
typedef unsigned (*ptrR)(unsigned, unsigned);
typedef int (*ptrCMP)(KEY, KEY);
typedef int (*ptrSetVal)(KEY *, KEY);

/* Проверка на принадлежность элемента множеству.
   Параметры: a – элемент; A – исходное множество; K – количество ячеек в хеш-таблице; H – указатель на хеш-функцию; R – указатель на функцию повторного хеширования; cmp – указатель на функцию сравнения двух значений типа KEY.
   Возвращаемое значение: индекс соответствующей ячейки в хеш-таблице, если a∈ A, и k в противном случае. */
unsigned Member(KEY a, SET A, unsigned K, ptrH H, ptrR R, ptrCMP cmp)
{ unsigned i, j;
  for (i = (*H)(a, K), j = 0; (A[i].state != 0) && (j < K); i = (*R)(i, K), j++)
      if ((*cmp)(A[i].key, a)) return i;
  return K; }
}
```

```
/* Вставка элемента во множество.
   Параметры: a – элемент; A – исходное множество; K – количество ячеек в хеш-таблице; h – указатель на хеш-функцию; r – указатель на функцию повторного хеширования; cmp – указатель на функцию сравнения двух значений типа KEY, setval – указатель на функцию, записывающую значение типа KEY по указанному адресу.
   Возвращаемое значение: 1 при успешном завершении операции, иначе – 0. */
int Insert(KEY a, SET A, unsigned K, ptrH H, ptrR R, ptrCMP cmp, ptrSetVal setval)
{ unsigned i, j;
  extern Member(KEY, SET, unsigned, ptrH, ptrR, ptrCMP);
  if (Member(a, A, K, H, R, cmp)) return 0;
  for (i = (*H)(a, K); A[i].state == 1; i = (*R)(i, K));
  (*setval)(&A[i].key, a);
  A[i].state = 1;
  return 1;
}

/* Удаление элемента из множества.
   Параметры: a – элемент; A – исходное множество; K – количество ячеек в хеш-таблице; H – указатель на хеш-функцию; R – указатель на функцию повторного хеширования; cmp – указатель на функцию сравнения двух значений типа KEY, setval – указатель на функцию, записывающую значение типа KEY по указанному адресу.
   Возвращаемое значение: 1 при успешном завершении операции, иначе – 0. */
int Delete(KEY a, SET A, unsigned K, ptrH H, ptrR R, ptrCMP cmp, ptrSetVal setval)
{ unsigned i;
  extern Member(KEY, SET, unsigned, ptrH, ptrR, ptrCMP);
  if (!Member(a, A, K, H, R, cmp)) return 0;
  for (i = (*H)(a, K); !(*cmp)(A[i].key, a); i = (*R)(i, K));
  A[i].state = –1;
  return 1; }
}

/* Объединение двух множеств.
   Параметры: A, B – исходные множества; K1 – количество ячеек в хеш-таблице A; K2 – количество ячеек в хеш-таблице B; H – указатель на хеш-функцию; R – указатель на функцию повторного хеширования; cmp – указатель на функцию сравнения двух значений типа KEY; setval – указатель на функцию, записывающую значение типа KEY по указанному адресу.
   Возвращаемое значение: при успешном завершении – указатель на результирующее множество, иначе – NULL. */
SET Union(SET A, unsigned K1, SET B, unsigned K2, ptrH H, ptrR R, ptrCMP cmp, ptrSet-
                Val setval)
{ SET ptrSET;
  unsigned i, K = K1 + K2;
  extern Member(KEY, SET, unsigned, ptrH, ptrR, ptrCMP);
  extern Insert(KEY, SET, unsigned, ptrH, ptrR, ptrCMP, ptrSetVal);
  if ((ptrSET = (SET) malloc(K * sizeof(CEL))) == NULL)
      return NULL;
  for (i = 0; i < K1; i++)
       if (A[i].state == 1)
            if (!Insert(A[i].key, ptrSET, K, H, R, cmp, setval)) return NULL;
  for (i = 0; i < K2; i++)
       if ((B[i].state == 1) && !Member(B[i].key, A, K1, H, R, cmp))
            if (!Insert(B[i].key, ptrSET, K, H, R, cmp, setval)) return NULL;
  return ptrSET;
}
```

```
/* Пересечение двух множеств.
   Параметры: A, B – исходные множества; K1 – количество ячеек в хеш-таблице A; K2 – количество
   ячеек в хеш-таблице B; H – указатель на хеш-функцию; R – указатель на функцию повторного
   хеширования; cmp – указатель на функцию сравнения двух значений типа KEY; setval – указатель
   на функцию, записывающую значение типа KEY по указанному адресу.
   Возвращаемое значение: при успешном завершении – указатель на результирующее множество,
   иначе – NULL.. */
SET Intersection(SET A, unsigned K1, SET B, unsigned K2, ptrH H, ptrR R, ptrCMP cmp,
                 ptrSetVal setval)
{ SET ptrSET, q, p;
  unsigned i, m, K = (K1 < K2) ? K1 : K2;
  extern Member(KEY, SET, unsigned, ptrH, ptrR, ptrCMP);
  extern Insert(KEY, SET, unsigned, ptrH, ptrR, ptrCMP, ptrSetVal);
  if ((ptrSET = (SET) malloc(k * sizeof(CEL))) == NULL) return NULL;
  if (K == K1)
     { m = K2; q = A; p = B; }
  else
     { m = K1; q = B; p = A; }
  for (i = 0; i < K; i++)
       if ((q[i].state == 1) && Member(q[i].key, p, m, H, R, cmp))
            if (!Insert(q[i].key, ptrSET, K, H, R, cmp, setval)) return NULL;
  return ptrSET;
}
/* Разность двух множеств.
   Параметры: A, B – исходные множества; K1 – количество ячеек в хеш-таблице A; K2 – количество
   ячеек в хеш-таблице B; H – указатель на хеш-функцию; R – указатель на функцию повторного
   хеширования; cmp – указатель на функцию сравнения двух значений типа KEY; setval – указатель
   на функцию, записывающую значение типа KEY по указанному адресу.
   Возвращаемое значение: при успешном завершении – указатель на результирующее множество,
   иначе – NULL. */
SET Difference (SET A, unsigned K1, SET B, unsigned K2, ptrH H, ptrR R, ptrCMP cmp,
                ptrSetVal setval)
{ SET ptrSET;
  unsigned i;
  extern Member(KEY, SET, unsigned, ptrH, ptrR, ptrCMP);
  extern Insert(KEY, SET, unsigned, ptrH, ptrR, ptrCMP, ptrSetVal);
  if ((ptrSET = (SET) malloc(K1 * sizeof(CEL))) == NULL) return NULL;
  for (i = 0; i < K1; i++)
       if ((A[i].state == 1) && !Member(A[i].key, B, K2, H, R, cmp))
            if (!Insert(A[i].key, ptrSET, K1, H, R, cmp, setval)) return NULL;
  return ptrSET;
}
/* Проверка на пустое множество.
   Параметры: A – исходное множество; K – количество ячеек в хеш-таблице.
   Возвращаемое значение: 1, если A = ∅; иначе – 0. */
int Empty(SET A, unsigned K)
{ unsigned i;
  for (i = 0; i < K; i++)
     if (A[i].state == 1) return 0;
  return 1;
}
```

3.4.2. Расширенное хеширование

Разрешение коллизий при расширенном хешировании осуществляется за счет размещения элементов с одинаковыми значениями хеш-функции в списках, головными элементами которых являются ячейки хеш-таблицы. Исходное множество (возможно бесконечное) разбивается на K сегментов, пронумерованных от 0 до K – 1, где K – количество ячеек в хеш-таблице. Хеш-функция H(x) выбирается так, чтобы при попадании некоторого элемента x множества в j-ый сегмент значение H(x) было равно j, т.е. x включается в список j-ой ячейки хеш-таблицы (рис. 3.5).

Размер сегментов может быть произвольным, но хеш-функция должна равномерно распределять элементы множества по сегментам. Если, например, известно, что элементами множества являются значения нормально распределенной случайной величины со средним значением Ξ и средним квадратическим отклонением σ, то хеш-функцию можно выбрать в виде

$$H(x) = \begin{cases} 0, x \le \Xi - 3\sigma \\ |x| \% (K-4) + 1, |x - \Xi| < 3\sigma \\ K-1, x \ge \Xi + 3\sigma \end{cases}$$

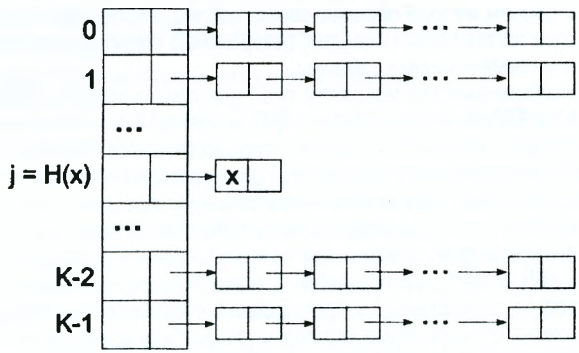


Рис. 3.5. Структура хеш-таблицы при расширенном хешировании.

Разбиение исходного множества на сегменты позволяет работать с бесконечными множествами, а объединение элементов в списки избавляет от необходимости помечать удаленные из хеш-таблицы элементы, поскольку при удалении элемента множества удаляется соответствующий элемент списка.

Вместе с тем, в отсутствие процедуры повторного хеширования, с помощью которой можно скорректировать неравномерное распределение значений хеш-функции по сегментам, повышаются требования к «качеству» хеш-функции, так как неравномерное распределение элементов приведет к появлению длинных списков в некоторых сегментах и, следовательно, к зависимости времени выполнения операций от мощности множества.

Как и при прямом хешировании, время выполнения основных операций при использовании расширенного хеширования пропорционально коэффициенту заполнения α, который в этом случае определяется как отношение количества содержащихся в хэш-таблице элементов к общему количеству ячеек (сегментов) и поэтому может, очевидно, превышать 1 за счет объединения в списки элементов, попадающих в один сегмент. Действительно, если элементы распределены в хеш-таблице равномерно, то коэффициент заполнения, есть не что иное, как среднее количество элементов в списках, сле-

довательно операции поиска, удаления или вставки элемента в хеш-таблицу будут выполняться за время порядка $O(\alpha)$, если время поиска ячейки фиксировано. Таким образом, пока $\alpha \approx 1$ время выполнения основных операций не зависит от количества элементов в хеш-таблице. С ростом мощности множества при неизменном количестве ячеек хеш-таблицы среднее количество просмотров увеличивается пропорционально α, и хеш-таблицу необходимо реструктурировать. Реструктуризация хеш-таблиц при расширенном хешировании производится при коэффициенте заполнения $\alpha \approx 2$.

3.5. Представление множеств посредством рекурсивных структур

Основной недостаток рекурсивных структур как инструментального средства для представления множеств выражается в зависимости времени выполнения основных операций от мощности множества. Однако этот недостаток может компенсироваться более эффективным использованием памяти, поскольку часть занимаемого перемешанными таблицами объема памяти не используется даже при расширенном хешировании, не говоря уже о прямом хешировании с его хеш-таблицами явно избыточного размера. Необходимость в своевременном проведении реструктуризации хеш-таблиц и проблема генерации неповторяющихся серий псевдослучайных целых чисел также следует отнести к «слабым» сторонам метода перемешанных таблиц. Более того, в некоторых случаях, в частности при организации словарей, рекурсивные структуры оказываются более эффективными, чем перемешанные таблицы.

При представлении множества мощности N в виде списка время выполнения операций Member, Insert и Delete имеет порядок $O(N)$, а время выполнения операций Union или Intersection для двух множеств, мощность каждого из которых равна N, будет иметь уже порядок $O(N^2)$. Для сравнения заметим, что при представлении множеств посредством перемешанных таблиц время выполнения таких операций как Union или Intersection для двух множеств мощности N каждое имеет порядок $O(N)$, так как время выполнения операций Member, Insert и Delete фиксировано. Тем не менее, если необходимо многократно выполнять операции MIN и MAX, т.е. производить поиск минимального или максимального элемента во множестве, эффективность перемешанных таблиц снижается до уровня неупорядоченного списка, в то время как подобные операции над упорядоченным списком выполняются за фиксированное время. Таким образом, о практическом использовании списков для представления множеств имеет смысл говорить лишь в случае представления линейно упорядоченных множеств упорядоченными списками.

Для реализации теоретико-множественных структур деревья двоичного поиска очевидно более эффективны чем списки, но поскольку деревья двоичного поиска требуют дополнительных временных затрат на поддержание их в сбалансированном состоянии, более подходящими для представления множеств чаще оказываются так называемые 2-3-деревья. 2-3-деревья относятся к сбалансированным деревьям и обеспечивают выполнение операции проверки на вхождение элемента во множество за время порядка $O(\log_2 N)$, но не в среднем, как в случае деревьев двоичного поиска, а в худшем случае.

3.5.1. 2-3-деревья

2-3-дерево – это дерево, в котором*):

- глубина всех листьев одинакова в любой момент времени;
- элементы множества содержатся только в листьях, и каждый лист содержит 1 или 2 элемента, причем элемент с меньшим значением всегда расположен левее;

*) Существует несколько разновидностей 2-3-деревьев.

- каждый внутренний узел имеет 2 или 3 сына и содержит, соответственно, 1 или 2 значения, которые представляют собой наименьшие значения среди элементов, потомков второго и третьего сыновей данного узла, причем значения элементов, потомков 2-го сына, превышают значения элементов, потомков 1-го сына, а значения элементов, потомков 3-го сына, превышают значения элементов, потомков 2-го сына;
- корень дерева либо является листом, либо, в противном случае, имеет не менее двух сыновей.

Для наглядности узлы 2-3-дерева будем изображать в виде прямоугольников, разделенных на два или пять полей для листьев и внутренних узлов, соответственно (рис. 3.6). В прямоугольнике, представляющем внутренний узел, в двух полях (обозначим их a_1 и a_2) будем записывать наименьшие значения элементов среди потомков второго и третьего сыновей, а остальные три поля (обозначим их p_1, p_2 и p_3) использовать для ссылки на 1-го, 2-го и 3-го сыновей. При отсутствии второго элемента у листа или третьего сына у внутреннего узла соответствующие поля будем помечать коротким тире.

p_1	a_1	p_2	a_2	p_3

Рис. 3.6. Представление внутренних узлов 2-3-дерева.

Как следует из определения 2-3-дерева, минимальное количество элементов при котором в 2-3-дереве появляется хотя бы один внутренний узел, равно 3, при этом элементы могут быть распределены между двумя или тремя листьями. На рис. 3.7,а изображено 2-3-дерево, состоящее из одного внутреннего узла, являющегося корнем, и двух листьев, один из которых содержит элементы 3 и 4, а другой – элемент 6. В такое дерево можно добавить еще три элемента – количество листьев при этом возрастет до трех. Соответствующий алгоритм фактически сводится к вставке нового элемента в упорядоченный по возрастанию список элементов, содержащихся в листьях. Если, например, добавить элементы 5, 9 и 10, то содержимое первого листа не изменится, второй лист будет содержать элементы 5, 6, а третий лист – элементы 9, 10, в результате чего в поле a_1 корня появится 5, а в поле a_2 – 9 (рис .3.7,б).

При попытке вставки еще одного элемента в 2-3-дерево, в котором уже содержится 6 элементов, это дерево придется преобразовать, поскольку листья 2-3-дерева не могут содержать более двух элементов, а внутренние узлы не могут иметь более трех сыновей. Суть преобразования заключается в расщеплении корня на два узла и перераспределении между ними листьев, в том числе и только что добавленного, так, что два листа с меньшими элементами становятся сыновьями старого корня, а два листа с большими элементами – сыновьями вновь созданного внутреннего узла. Затем создается новый корень 2-3-дерева, его сыновьями становятся внутренние узлы, полученные при расщеплении старого корня, и устанавливаются соответствующие значения полей a_1 всех трех внутренних узлов. Таким образом, в результате подобных преобразований количество узлов в 2-3-дереве сразу увеличивается до семи (три внутренних узла и четыре листа) и добавляется один уровень.

Если в 2-3-дерево, изображенное на рис. 3.7,б, добавить элемент 7, то в соответствии с описанным выше алгоритмом листья с элементами 3, 4 и 5, 6 останутся сыновьями старого корня, а листья с элементами 7 и 9, 10 станут сыновьями вновь созданного

внутреннего узла. Установив значения полей a_1 (поля a_2 при этом остаются пустыми) для всех трех внутренних узлов, получаем 2-3-дерево, изображенное на рис. 3.7,в.

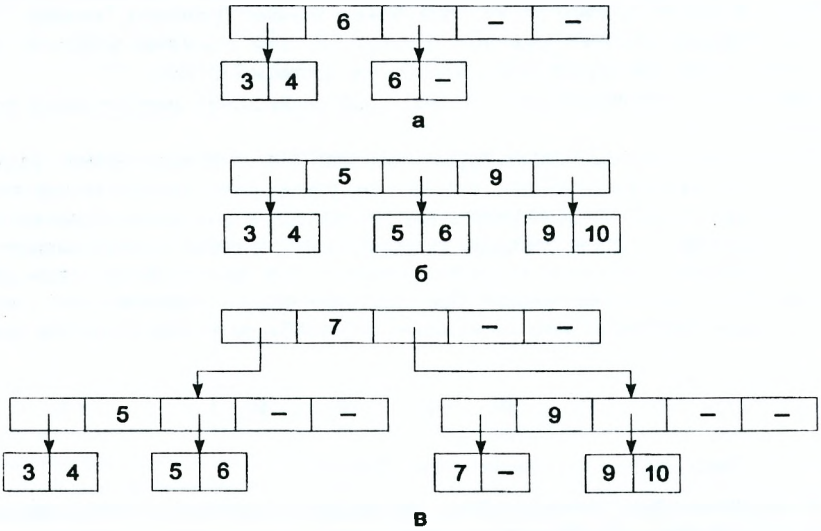


Рис. 3.7. 2-3-дерево с тремя элементами (а) и его вид после последовательной вставки элементов 5, 9, 10 (б) и после вставки элемента 7 (в).

Рассмотрим алгоритмы операций поиска, вставки и удаления элементов в 2-3-дереве:

1) Поиск элемента в 2-3-дереве

В общем случае для поиска элемента в 2-3-дереве необходимо пройти от его корня до последнего уровня, на котором расположены листья. Однако если для какого-то внутреннего узла обнаружено совпадение искомого значения со значением в поле a_1 или a_2 этого узла, просмотр узлов прекращается и процедура завершается с кодом возврата, соответствующим положительному результату поиска. Приведем описание алгоритма этой операции на псевдоязыке:

```
/* Начало процедуры */
Присвоить указателю q адрес корня 2-3-дерева;
Присвоить переменной V значение искомого элемента;
while (Узел q не является листом)
    { if (Значение одного из полей a1 или a2 узла q равно V)
          Завершить процедуру с положительным результатом поиска;
      if (Если V меньше значения поля a1)
          Перейти к первому сыну узла q;
      else
          if (Если у узла q нет третьего сына или третий сын есть, но V меньше значения
              в поле a2)
              Перейти ко второму сыну узла q;
          else
              Перейти к третьему сыну узла q;
    }
```

if (Узел q содержит элемент, значение которого равно V)
    Завершить процедуру с положительным результатом;
else
    Завершить процедуру с отрицательным результатом;
/ Конец процедуры */*

2) Вставка элемента в 2-3 дерево

Операция вставки элемента в 2-3-дерево выполняется за три шага. Первый шаг – поиск листа, в который мог бы быть помещен новый элемент. Поиск производится по описанному выше алгоритму, причем в качестве искомого значения выступает значение нового элемента. Если в результате поиска устанавливается, что элемент с таким же значением имеется в 2-3-дереве, процедура завершается с кодом возврата, соответствующим неуспешному завершению операции. В противном случае последующие действия определяются количеством элементов, содержащихся в найденном листе (для удобства описания алгоритма обозначим его L): если в L содержится один элемент, то новый элемент добавляется к L и процедура завершается с кодом возврата, соответствующим успешному завершению операции. Если же в L содержатся два элемента, но в поддереве, корнем которого является прямой предок L (обозначим его R), содержится менее шести элементов, то все элементы, включая новый, распределяются между L и его братьями (если третьего сына у узла R нет, то он может быть создан) и процедура также успешно завершается, иначе – переход ко второму шагу.

На втором шаге производится поиск места для «лишнего» элемента среди потомков узлов, являющихся братьями R, причем предпочтение следует отдать, очевидно, тому из этих узлов, у которого нет третьего сына. Если это будет узел, расположенный слева от R, то в роли «лишнего» выступит элемент с минимальным значением среди элементов, потомков R, и нового элемента, а если – справа, то с максимальным значением среди тех же элементов. В случае результативного поиска производится вставка «лишнего» элемента, и процедура успешно завершается, иначе – переход к третьему шагу.

На третьем шаге создается новый лист, в нем размещается элемент с наибольшим значением среди трех элементов – двух, содержащихся в L, и нового элемента – и R расщепляется на два узла по описанной выше схеме преобразования корня 2-3-дерева с тремя сыновьями. Прямым предком вновь созданного листа при этом станет либо R, либо вновь созданный внутренний узел (обозначим его S). Далее, поскольку прямой предок R (обозначим его P) имеет трех сыновей, он преобразуется по той же схеме. Это преобразование выполняется рекурсивно для всех вышележащих родительских узлов, включая корень, до тех пор, пока не встретится узел с двумя сыновьями или не будет создан новый корень. В завершение операции производится корректировка полей a_1 и a_2 всех внутренних узлов, для которых произведенные преобразования 2-3-дерева могли повлиять на наименьшее значение среди элементов, потомков второго и/или третьего сына.

В качестве примера рассмотрим последовательную вставку элементов 8, 11, 12, 13, 14 и 15 в 2-3-дерево, изображенное на рис. 3.7,в (для удобства описания узлы, соответствующие первому и второму сыну корня в этом дереве, обозначим A и B, соответственно). Сначала элементы 8, 11, 12 будут размещены в листьях, являющихся потомками узла B, причем для элементов 11 и 12 будет создан новый лист, который станет третьим сыном узла B. Поиск листа для элемента 13 приведен опять же к третьему сыну узла B, но поскольку в листьях, являющихся сыновьями узла B, свободного места нет, лист с элементами 7, 8 будет передан узлу A и станет его третьим сыном, а листья с элементами 9, 10; 11, 12 и 13 станут, соответ-

ственно, первым, вторым и третьим сыном узла B. В результате в поле a_1 корня будет записано значение 9, а в поля a_1 и a_2 узла B – значения 11 и 13. После того, как элемент 14 будет вставлен в лист, являющийся третьим сыном узла B, дальнейшая вставка элементов без преобразования внутренних узлов 2-3-дерева будет невозможна, и вставка элемента 15 повлечет за собой расщепление узла B и балансировку листьев между узлом B и вновь созданным внутренним узлом, который станет третьим сыном корня. После корректировки полей a_1 и a_2 корня и его второго и третьего сыновей 2-3-дерево примет вид, как показано на рис. 3.8.

3) Удаление листа из 2-3-дерева

Эта операция также начинается с поиска листа (обозначим его L, а его прямого предка R, как и при описании предыдущего алгоритма), в котором содержится удаляемый элемент. Если в L такого элемента нет, то процедура завершается с кодом возврата, соответствующим неуспешному завершению операции, иначе элемент удаляется. Если удаляемый элемент не единственный в L, то процедура также завершается, но уже с кодом возврата, соответствующим успешному завершению операции. В противном случае недостающий элемент может быть заимствован у непосредственных соседей L слева или справа, причем в качестве внешнего «донора» может выступать либо первый сын правого брата R, либо второй или третий сын левого брата R. В ситуации, когда и R, и его братья имеют лишь по два сына, в каждом из которых содержится по одному элементу, заимствование невозможно, поэтому L удаляется, а единственный сын R передается левому или правому брату последнего, после чего R также удаляется.

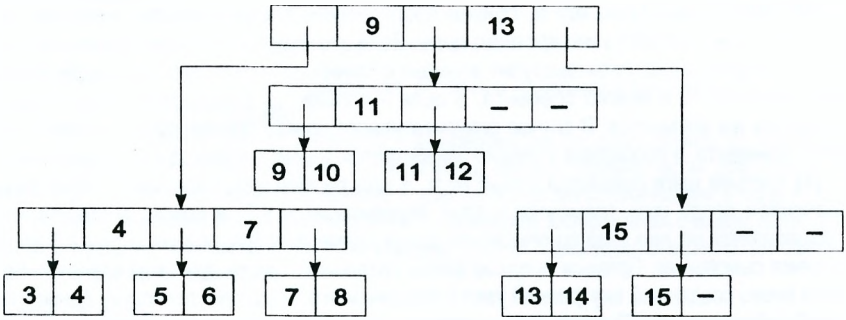


Рис. 3.8. Состояние 2-3-дерева, изображенного на рис. 3.7,в, после последовательной вставки элементов 8, 9, 10, 11, 12, 13, 14, 15.

При удалении прямого предка узла в 2-3-дереве последний, очевидно, может быть передан одному из братьев удаляемого узла, хотя в случае альтернативы предпочтение следует отдать, по-видимому, соседу слева, поскольку для соседа справа передаваемый узел автоматически становится первым сыном, что влечет за собой поиск соответствующего вышележащего внутреннего узла, для которого в результате указанного преобразования изменится наименьшее значение среди потомков 2-го или 3-го сына.

Удаление внутренних узлов повторяется рекурсивно до тех пор, пока прямой предок очередного удаленного узла не будет иметь двух сыновей или это будет корень 2-3-дерева. И, наконец, если корень 2-3-дерева останется с одним сыном, он также удаляется, а его единственный сын становится новым корнем дерева.

Операция удаления элемента из 2-3-дерева, как и операция вставки, завершается корректировкой полей a_1 и a_2 всех внутренних узлов, задействованных в операции.

Рассмотрим, например, последовательное удаление элементов 3, 4, 5, 6, 11, 12, 13 и 10 из 2-3-дерева, изображенного на рис. 3.8, обозначая, как и в предыдущем примере, первого и второго сыновей корня A и B, а третьего сына – C. После удаления элементов 3, 4, 5, 6, 11, 12 и 13 и перераспределения оставшихся элементов среди потомков узлов A, B и C, у каждого из этих узлов останется по два листа, содержащих по одному элементу, поэтому последующее удаление элемента 10 повлечет за собой удаление B, в результате чего единственный сын B, лист с элементом 9, будет передан A, и после корректировки меток внутренних узлов 2-3-дерево принимает вид, как показано на рис. 3.9.

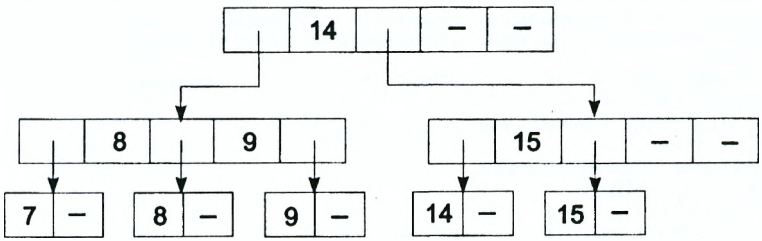


Рис. 3.9. Состояние 2-3-дерева, изображенного на рис. 3.8, после последовательного удаления элементов 3, 4, 5, 6, 11, 12, 13, 10.

Рассмотрев алгоритмы основных операций манипулирования узлами 2-3-дерева, перейдем к реализации АТД «множество» посредством 2-3-дерева.

Определим тип SET:

```
typedef NODE ** SET;
typedef struct node
{ ITEM a1,    /* Наименьшая метка среди потомков 2-го сына. */
          a2; /* Наименьшая метка среди потомков 3-го сына. */
  struct node * p[3]; /*Массив ссылок на сыновей. */
  struct node * par; /* Ссылка на прямого предка. */
} NODE;
```

Если узел содержит один элемент, в поле a2 будем записывать некоторое константное значение empty (относящееся, очевидно, к типу ITEM) и ссылке на третьего сына, а для листьев всем трем ссылкам, присваивать значение NULL.

В Листинге 3.3 приведены исходные коды функций, реализующих операции Member и Insert на основе вышеприведенного определения типа SET. В данной реализации предполагается, что тип ITEM, т.е. тип элементов множества, относится к перечислимому типу (в противном случае для сравнения и присваивания значений типа ITEM придется определить соответствующие функции), а также используются несколько вспомогательных функций:

```
void SetMin(NODE *ptrleaf);
ITEM GetMin(NODE *ptrnode);
int LocalSearch(NODE *ptrnode);
int LocalInsert(NODE *ptrnode, ITEM item);
```

Функция SetMin производит поиск соответствующего вышележащего внутреннего узла и корректировку его поля наименьшего элемента среди потомков 2- или 3-го сына при

изменении наименьшего значения среди элементов, содержащихся в листе, адрес которого задается параметром ptrleaf.

Функция GetMin находит и возвращает значение наименьшего элемента среди потомков узла, адрес которого задается параметром ptrnode.

Функция LocalSearch производит поиск свободного места для вставки элемента в один из листьев, являющихся истинными потомками узла, адрес которого задается параметром ptrnode. Возвращаемое значение: 1, если поиск результативен, и 0 - в противном случае.

Функция LocalInsert выполняет вставку нового элемента в один из листьев, являющихся истинными потомками узла, адрес которого задается параметром ptrnode. Возвращаемое значение: 1 при успешном завершении операции и 0 - в противном случае.

Читателю предлагается в качестве упражнения самостоятельно определить функции SetMin, GetMin, LocalSearch, LocalInsert.

Листинг 3.3

```
/* Операция Member для множества, представленного посредством 2-3-дерева.
   Входные параметры: a – значение, проверяемое на принадлежность множеству; A – исходное
   множество.
   Возвращаемое значение: 1, если a принадлежит множеству, и 0 – в противном случае. */
int Member(ITEM a, SET A)
{ extern ITEM empty; /* Значение «пусто» для поля a2. */
  NODE *L = *A;
  /* Поиск листа, содержащего элемент a. */
  while(L->p[1] != NULL)
      { ITEM a1 = L->a1, a2 = L->a2;
        NODE *s1 = L->p[1], *s2 = L->p[2], *s3 = L->p[3];
        if ((a1 == a) || (a2 == a)) return 1;
        if (a1 > a) L = s1;
        else
            if ((a2 == empty) || (a2 > a)) L = s2;
            else L = s3;
      }
  return (L->a1 == a) || (L->a2 == a);
}
/* Операция Insert для множества, представленного посредством 2-3-дерева.
   Входные параметры: a – значение добавляемого во множество элемента, A – указатель на указатель на корень 2-3-дерева.
   Возвращаемое значение: 1, если операция выполнена успешно, и 0 – в противном случае. */
int Insert(ITEM a, SET A)
{ extern void SetMin(NODE *);
  extern ITEM GetMin(NODE *);
  extern LocalSearch(NODE *);
  extern LocalInsert(NODE *, ITEM);
  const extern ITEM empty; /* Значение «пусто» для поля a2. */
  NODE *R, *P, *news, *newp, *L = *A;
  unsigned i, j;
  if (L == NULL) return 0;
  /* Поиск листа для вставки нового элемента. */
  while(L->p[1] != NULL)
      { ITEM a1 = L->a1, a2 = L->a2;
        NODE *s1 = L->p[1], *s2 = L->p[2], *s3 = L->p[3];
        if ((a1 == a) || (a2 == a)) return 0;
        if (a1 > a) L = s1;
        else
            if ((a2 == empty) || (a2 > a)) L = s2;
            else L= s3;
      }
```

```
if ((L->a1 == a) || (L->a2 == a)) return 0;
if (L->a2 == empty)
    { if (L->a1 < a)
        { L->a2 = a;
          return 1; }
      else
        { L->a2 = L->a1;
          L->a1 = a;
          if (L->par == NULL) return 1; }
if ((R = L->par) != NULL);
    { j = R->p[0] == L ? 0 : R->p[1] == L ? 1 : 2;
      if (L->a1 == a)
        { switch (j)
            { case 0: SetMin(L); break;
              case 1: R->a1 = a; break;
              case 2: R->a2 = a; }
          return 1; }
      /* Поиск свободного места среди потомков узла R и его братьев. */
      if (LocalSearch(R)) return LocalInsert(R, a);
      if ((P = R->par) != NULL)
        { int k = -1, m;
          i = P->p[0] == R ? 0 : P->p[1] == R ? 1: 2;
          for (m = 2; m >= 0; m - -)
            { NODE *q = P->p[m];
              if ((m != i) && (q != NULL))
                { if ((k < 0) && LocalSearch(q)) k = m;
                  if (q->p[2] == NULL)
                    { k = m;
                      break; } }
          if ((m >= 0) || (k*m <= 0))
            { ITEM min, max, x, y;
              min = (x = R->p[0]->a1) > a ? a : x;
              max = (x = R->p[2]->a2) < a ? a : x;
              x = (k < i) ? min : max;
              if (x == min)
                { if ((y = L->a1 > a ? a : L->a1) != a)
                    { if (L->a2 > a) L->a1 = a;
                      else
                        { L->a1 = L->a2;
                          L->a2 = a; }
                      switch (j)
                        { case 0: SetMin(L); break;
                          case 1: R->a1 = L->a1; break;
                          case 2: R->a2 = L->a1; } }
                  for ( j - -; j >= 0; j - -)
                    { NODE *q = R->p[j];
                      x = q->a1;
                      q->a1 = q->a2;
                      q->a2 = y;
                      switch (j)
                        { case 0: SetMin(L); break;
                          case 1: R->a1 = q->a1; break;
                          case 2: R->a2 = q->a1; }
                      y = x; }
                  for (i - -; i != k; i - -)
                    { NODE **q = P->p[i]->p;
```

```
                        x = q[0]->a1;
                        for (j = 0; j < 3; j + +)
                            { q[j]->a1 = q[j]->a2;
                              if (j < 2) q[j]->a2 = q[j + 1]->a1;
                              else q[j]->a2 = y; }
                        y = x; }
                    }
                else
                    { if ((y = L->a2 < a ? a : L->a2) != a)
                        { if (L->a1 < a) L->a2 = a;
                          else
                              { L->a2 = L->a1;
                                L->a1 = a; }
                          switch (j)
                              { case 0: SetMin(L); break;
                                case 1: R->a1 = L->a1; break;
                                case 2: R->a2 = L->a1; } }
                      for ( j + +; j < 3; j + +)
                          { NODE *q = R->p[j];
                            x = q->a2;
                            q->a2 = q->a1;
                            q->a1 = y;
                            switch (j)
                              { case 0: SetMin(L); break;
                                case 1: R->a1 = q->a1; break;
                                case 2: R->a2 = q->a1; }
                            y = x; }
                      for (i + +; i != k; i + +)
                          { NODE **q = P->p[i]->p;
                            x = q[2]->a2;
                            for (j = 2; j >= 0; j – –)
                                { q[j]->a2 = q[j]->a1;
                                  if (j > 0) q[j]->a1 = q[j – 1]->a2;
                                  else q[j]->a1 = y; }
                            y = x; }
                    }
                return LocalInsert(P->p[i], y);
            }
        }
    }
/* Создать лист и сбалансировать элементы между ним и листом L. */
if ((news = malloc(sizeof(NODE))) == NULL) return 0 ;
for (k = 0; k < 3; k++) news->p[k] = NULL;
news->a2 = empty;
if ((news->a1 = L->a2 < a ? a : L->a2) != a)
    if (L->a1 < a) L->a2 = a;
    else
        { L->a2 = L->a1;
          L->a1 = a;
          switch (j)
              { case 0: SetMin(L); break;
                case 1: R->a1 = a; break;
                case 2: R->a2 = a; } }
if (R != NULL)
    {
```

```
    /* Расщепить узел R на два узла. Если прямой предок узла R имеет трех сыновей, расщепление внутренних узлов продолжать рекурсивно до тех пор, пока очередной родительский
       узел не будет иметь двух сыновей или будет достигнут корень дерева. */
    do
      { if ((newp = malloc(sizeof(NODE))) == NULL) return 0;
        switch (j)
          { case 0: newp->p[0] = R->p[1];
                    R->a1 = GetMin(R->p[1] = news);
                    newp->p[1] = R->p[2];
                    break;
            case 1: newp->p[0] = news;
                    newp->p[1] = R->p[2];
                    break;
            case 2: newp->p[0] = R->p[2];
                    newp->p[1] = news; }
        newp->a1 = GetMin(newp->p[1]);
        newp->a2 = empty;
        R->p[2] = NULL;
        R->a2 = empty;
        if (P != NULL)
          { R = P;
            news = newp;
            j = i;
            if ((P = P->par) != NULL)
              { if (P->p[2] == NULL) break;
                i = P->p[0] == R ? 0 : P->p[1] == R ? 1: 2; }
          }
      while (P != NULL);
      if (P != NULL)
        { if (j == 0)
            { R->a2 = GetMin(R->p[2] = R->p[1]);
              R->a1 = GetMin(R->p[1] = news); }
          else R->a2 = GetMin(R->p[2] = news);
          return 1; }
    }
  else
    { R = L;
      newp = news;
    }
  /* Создать новый корень 2-3-дерева и сделать узлы R и newp его сыновьями. */
  if ((news = malloc(sizeof(NODE))) == NULL) return 0;
  news->p[0] = R;
  news->a1 = GetMin(news->p[1] = newp);
  news->a2 = empty;
  news->p[3] = NULL;
  *A = news;
  return 1;
}
```

В заключение оценим время выполнения операции Member для множества, представленного 2-3 деревом. Для выполнения этой операции необходимо, очевидно, просмотреть столько узлов, сколько уровней в 2-3-дереве. Аналитически определить количество уровней при заданном числе N листьев невозможно, но можно указать нижнюю и верхнюю границы. Ясно, что число уровней в 2-3-дереве будет минимальным в том случае, если все внутрен-

ние узлы имеют по три сына, т.е. количество уровней ограничено снизу величиной $1 + \log_3 N$. Если у каждого узла будет по два сына, то количество уровней при том же количестве листьев будет максимальным, а это означает, что количество уровней ограничено сверху величиной $1 + \log_2 N$. Следовательно время поиска листа в 2-3 дереве в худшем случае имеет порядок $O(\log_2 N)$, а в лучшем случае – $O(\log_3 N)$.

Таким образом, в отличие от хеш-таблиц, для которых время выполнения основных операций возрастает по мере увеличения коэффициента заполнения, время выполнения операции поиска в 2-3 дереве остается неизменным, если число уровней не изменяется, а коэффициент заполнения 2-3-дерева определяется как отношение количества имеющихся листьев к их максимально возможному количеству, которое для k-уровневого 2-3-дерева равно 3^{k-1}. Однако с ростом N среднее число просматриваемых элементов в хеш-таблице не изменится, если при этом пропорционально увеличить ее размер, оставляя неизменным коэффициент заполнения, тогда как среднее число просматриваемых узлов в 2-3-дереве возрастет, по меньшей мере, как $\log_3 N$.

3.5.2. B-деревья

Если мощность множества достаточно велика, объем оперативной памяти может быть не достаточен для размещения узлов 2-3-дерева. В этом случае, по крайней мере, часть узлов 2-3-дерева, например, листья, должны храниться во внешней памяти, а их ссылочные поля – содержать адреса блоков не оперативной памяти, а внешней. Однако при размещении узлов в дисковом файле возникает проблема сокращения количества обращений к этому файлу. Действительно, для поиска элемента во множестве, состоящем всего из 1000 элементов, понадобится просмотреть порядка $\log_2 2^{10} = 10$ узлов в 2-3-дереве, т.е. сделать 10 обращений к диску. Эта проблема решается с помощью B-деревьев, которые являются обобщением 2-3-деревьев и используются для организации крупномасштабных деревьев поиска данных во внешней памяти.

B-дерево порядка m – это дерево, в котором:

- внутренние узлы имеют не менее [m/2] + 1*), но не более m сыновей;
- каждый узел содержит не менее [m/2], но не более m – 1 элементов;
- корень дерева либо является листом, либо имеет не менее двух сыновей;
- все листья имеют одинаковую глубину.

Из данного определения вытекает, что 2-3-дерево есть B-дерево порядка 3, т.к. внутренние узлы, включая корень, имеют от 2 до 3 сыновей, и каждый узел в 2-3-дереве содержит от 1 до 2 элементов.

Так же, как и внутренние узлы 2-3-деревьев, внутренние узлы B-деревьев удобно изображать в виде прямоугольников, разделенных на 2m – 1 полей (рис.3.10), среди которых m полей занимают указатели на сыновей p_i, $0 \le i \le m-1$, и m – 1 полей – упорядоченные элементы $a_1 < a_2 < a_3 \dots a_{m-2} < a_{m-1}$. Соотношение между элементами в поддереве, на которое ссылается указатель p_k, и смежными для этого указателя элементами в данном узле зависит от k:

- если k = 0, то все элементы в поддереве меньше a_1;
- если k = m – 1, то все элементы в поддереве больше a_{m-1};
- если $1 \le k < m-1$, то все элементы в поддереве не меньше a_k и меньше a_{k+1}, другими словами, a_k представляет собой минимальный элемент в поддереве, на которое указывает p_k.

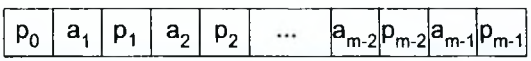


Рис. 3.10. Структура внутренних узлов B-дерева.

*) Прямоугольные скобки обозначают взятие целой части.

Внутренние узлы могут храниться как во внешней, так и во внутренней памяти, кроме корневого узла, который должен, очевидно, постоянно располагаться во внутренней памяти.

Листья B-дерева, как правило, хранятся в файле в виде блоков фиксированного размера, равного максимальному количеству элементов m – 1. Элементы в листьях также упорядочены по возрастанию, поэтому, если все элементы, содержащиеся в листьях B-дерева, отобразить на одномерную последовательность, обходя их слева направо, начиная с крайнего слева листа, то эта последовательность будет упорядочена по возрастанию значений элементов. Такое распределение элементов по узлам B-дерева обусловливает, как и для 2-3-деревьев, выполнение операций манипулирования элементами.

1. Поиск элемента в B-дереве.

Для поиска некоторого элемента x в B-дереве надо пройти от корня дерева до листа, в котором может содержаться искомый элемент. Для каждого внутреннего узла на этом пути выбирается то поддерево, в котором содержится x, для этого необходимо найти такой указатель p_s, что $x < a_s$ при $s = 0$, $a_s < x < a_{s+1}$ при $0 < s < m - 1$ или $x > a_s$ при $s = m - 1$. Если при этом обнаруживается совпадение x с одним из элементов внутреннего узла, означающее, что искомый элемент имеет наименьшее значение среди потомков s-го сына данного внутреннего узла, то дальнейший поиск, естественно, прекращается.

После того, как найден лист, в котором возможно содержится искомый элемент, соответствующий блок считывается во внутреннюю память и выполняется линейный или двоичный, если m достаточно велико, поиск среди его элементов.

2. Вставка элемента в B-дерево.

Как и в случае 2-3-дерева, операция вставки элемента в B-дерево начинается с поиска листа, в котором должен содержаться новый элемент. При этом если количество элементов в каком-либо узле превысит m – 1, также выполняется разделение этого узла на два узла и их балансировка, в результате которой m элементов распределяются поровну между преобразуемым узлом и вновь созданным его правым братом. Расщепление узлов выполняется снизу вверх и может достигнуть корня, в результате преобразования которого появляется новый корень с двумя сыновьями. Таким образом, корень B-дерева – это единственный узел, который может иметь менее [m/2] элементов.

3. Удаление элемента из B-дерева.

Если в результате удаления элемента количество оставшихся в данном листе элементов станет менее допустимого, то недостающие элементы могут быть позаимствованы у братьев данного узла слева или справа, но при условии их балансировки. Если же заимствование невозможно, выполняется слияние узлов, которое, как и разделение узлов при вставке элементов, выполняется снизу вверх вплоть до корня.

Время выполнения операций с элементами B-дерева функционально зависит от максимально допустимого количества элементов в узлах, т.е. от порядка B-дерева. Действительно, чем больше высота B-дерева, тем больше внутренних узлов надо просмотреть на пути от корня до любого его листа и тем больше время выполнения операций. Высота B-дерева при данном количестве элементов N имеет наибольшее значение в том случае, когда каждый внутренний узел имеет минимальное количество сыновей [m/2] + 1 или приближенно m/2 при достаточно большом m. Тогда, учитывая, что максимальное количество листьев равно 2N/m, находим верхнюю границу для высоты B-дерева:

$$H = \log_{m/2} \frac{2N}{m} = \log_{m/2} N - 1 \overset{m \ll 1}{=} \frac{\log_2 N}{\log_2 m} - 1, \qquad (3.8)$$

и из условия $H \geq 1$ – верхнюю границу для m:

$$m \leq 2\sqrt{N} \qquad (3.9)$$

Из (3.8) и (3.9) следует, что чем больше m, тем меньше внутренних узлов B-дерева придется просматривать, но m не может быть сколь угодно большим, т.к. с ростом m увеличивается и время поиска внутри самого узла.

Поскольку зависимость от m времени, которое требуется для того, чтобы в процессе поиска некоторого элемента в B-дереве пройти от корня до соответствующего листа, считав при

этом порядка H блоков из внутренней или внешней памяти и выполнив двоичный поиск среди порядка m элементов внутри каждого из этих блоков, можно представить в виде функции

$$T(m) \cong \log_2 N \left\{ \frac{\alpha + \beta m}{\log_2 m} + \gamma \right\}, \qquad (3.10)$$

где α, β, γ – константы, то нетрудно показать, что существует значение m, для которого это время минимально.

Если внутренние узлы B-дерева расположены во внешней памяти, то на m накладывается дополнительное условие, вытекающее из того, что во внутренней памяти придется разместить порядка H блоков, в каждом из которых может содержаться до m – 1 элементов и m указателей.

3.5.3. Нагруженные деревья

Для представления множеств, состоящих из символьных строк, альтернативой перемешанным таблицам служат такие рекурсивные структуры, как **нагруженные деревья**[*], которые обладают следующими свойствами:

- все узлы, кроме листьев, могут иметь столько сыновей, сколько символов в используемом алфавите;
- внутренние узлы помечаются отдельными символами алфавита, а листья – специальными концевыми метками, которые необходимы, чтобы можно было выделять более короткие строки, являющиеся префиксами более длинных строк;
- путь от корня к какому-либо листу, записанный в виде последовательности меток пройденных узлов, соответствует одной строке из множества.

Если корень дерева пометить одним из символов используемого алфавита, то все строки, очевидно, будут начинаться с одного и того же символа. Поэтому в ситуации, когда необходимо создать множество строк с разными префиксами, можно просто в качестве начального символа строк использовать не корень, а один из его сыновей, тогда корень нагруженного дерева будет выполнять единственную функцию точки входа в структуру и может быть помечен произвольным символом, причем не обязательно из используемого алфавита.

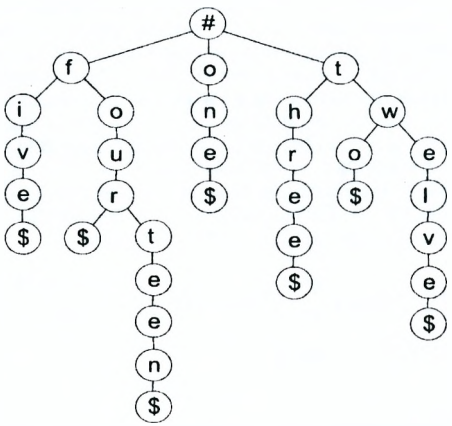


Рис. 3.11. Пример нагруженного дерева.

[*] Этому термину в англоязычной литературе соответствует термин «TRIE» от англ. reTRIEval, т.е. поиск, выборка.

На рис. 3.11 показано нагруженное дерево для множества строк {one, two, three, four, five, twelve, fourteen}. Строки состоят из символов латинского алфавита, корень дерева помечен символом «#», а в качестве концевых меток используется символ «$». Заметим, что благодаря наличию общих для нескольких строк префиксов (f, four, t и tw) количество узлов в нагруженном дереве меньше, чем суммарное количество символов во всех строках.

В нагруженном дереве время выполнения основных операций определяется тем, как представлены прямые потомки узлов. Для этого можно использовать упорядоченные списки или массивы фиксированной длины, поскольку количество сыновей любого узла, кроме листьев, ограничено количеством символов алфавита.

При представлении прямых потомков узлов посредством массивов тип узлов нагруженного дерева можно определенный как:

```
typedef struct node
{ unsigned char chr; /* Метка узла. */
  struct node * succ[M]; /* Массив указателей на сыновей. */
} TRIE;
```

где M – количество символов в алфавите. При таком представлении время прохождения каждого узла фиксировано, поэтому данная реализация АТД обеспечивает наибольшее быстродействие операторов. Однако суммарный объем памяти, занимаемый узлами нагруженного дерева, может значительно превышать объем памяти, который потребовался бы для последовательного размещения всех строк множества.

При представлении прямых потомков узлов посредством упорядоченных списков память используется более эффективно, но требуется дополнительное время на поиск узла в каждом списке на пути от корня до листа. Узлы нагруженного дерева в этом случае удобно определить как объекты типа:

```
typedef struct node
{ unsigned char character;
  struct node * next; /* Указатель на правого брата. */
  struct node * first; /* Указатель на первый элемент списка сыновей. */
  struct node * last; /* Указатель на последний элемент списка сыновей. */
} TRIE;
```

Проведем сравнительную оценку эффективности перемешанных таблиц при расширенном хешировании и нагруженных деревьев для представления множеств, состоящих из строк переменной длины. В данном случае под эффективностью структур понимается время выполнения основных операций и требуемый для реализации объем памяти.

Для нагруженных деревьев такие операции, как Member, Insert и Delete, выполняются за время, пропорциональное длине строки-аргумента, поэтому даже при представлении узлов связанными списками сыновей временные показатели нагруженных деревьев и хеш-таблиц оказываются сравнимыми, а при использовании массивов для представления прямых потомков узлов эффективность нагруженных деревьев оказывается выше, так как ко времени, требуемому для определения номера сегмента в хеш-таблице, необходимо добавить время на обработку коллизий, т.е. поиск элемента в списке, длина которого может быть сравнима с длиной строки, а также временные затраты на реструктуризацию хеш-таблицы. Тем не менее, если поиск, вставка и удаление строк производятся многократно, как при выполнении операций Union и Intersection, временные показатели для хеш-таблиц и для нагруженных деревьев сопоставимы лишь в случае представления узлов нагруженных деревьев с помощью массивов.

Сравнение по требуемому объему памяти имеет смысл, очевидно, при представлении узлов нагруженных деревьев связанными списками сыновей. При таком представлении количество узлов в нагруженном дереве определяется соотношением между количеством строк и количеством префиксов (но не количеством символов в них). Если обозначить через ψ долю узлов от их максимально возможного количества, равного суммарному количеству

символов в строках, то объем памяти, необходимый для размещения узлов нагруженного дерева, будет приближенно равен

$$\psi N \lambda (3p+1) \qquad (3.11)$$

где N – количество строк (мощность множества), λ – средняя длина строк, p – размер указателя в используемой модели памяти. Чем меньше префиксов в строках, тем больше значение коэффициента ψ и тем больше узлов в нагруженном дереве. В предельном случае, когда все строки различны и не имеют общих префиксов, ψ = 1. И, наоборот, с ростом количества префиксов значение ψ уменьшается, но не имеет нулевого предела, если даже количество префиксов достигает своего максимального значения.

Объем памяти, требуемый для организации множества строк переменной длины по методу расширенного хеширования, складывается из двух частей, определяемых размером самой хеш-таблицы и суммарной длиной строк. Если в K сегментах хеш-таблицы содержится по одному указателю на первый элемент списка, а узлы списков состоят из указателя на следующий элемент и указателя на соответствующую строку, то результирующий объем памяти определяется выражением:

$$Kp + 2pN + N\lambda = N\lambda\left(1 + \frac{p}{\lambda}\,\frac{2+\alpha}{\alpha}\right), \qquad (3.12)$$

где α – коэффициент заполнения хеш-таблицы.

Как следует из выражений (3.11 и (3.12), при ψ = 1 размер нагруженного дерева в несколько раз превосходит размер хеш-таблицы и размер строк вместе взятых, однако при

$$\psi = \frac{p(2+\alpha)+\lambda\alpha}{(3p+1)\lambda\alpha} \overset{\alpha=1}{\approx} \frac{3p+\lambda}{(3p+1)\lambda} \qquad (3.13)$$

требуемые объемы памяти в обоих случаях становятся приблизительно одинаковыми.

Таким образом, чем больше общих префиксов в строках, т.е. чем меньше коэффициент ψ в выражении (3.11), тем более экономно используется память в нагруженном дереве. Это объясняет тот факт, что нагруженные деревья наиболее эффективны в представлении словарей, представляющих собой множества слов, разбитых на группы с одинаковыми префиксами.

На рис. 3.12 изображен фрагмент словаря, в котором содержится 8 слов, средняя длина которых λ ≈ 5,75. В представляющем этот фрагмент нагруженном дереве 30 узлов, поэтому в данном случае имеем ψ ≈ 0,77.

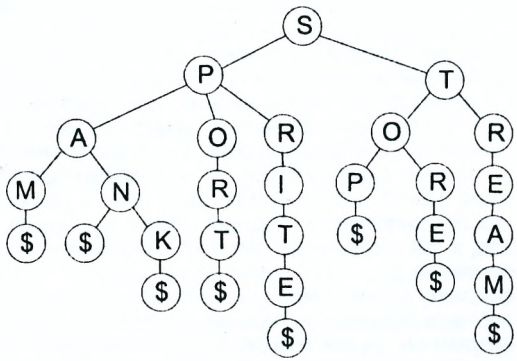


Рис. 3.12. Фрагмент словаря, представленного посредством нагруженного дерева.

УПРАЖНЕНИЯ

1. Представить посредством двоичного вектора:
   а) множество целых чисел $X=\{x \mid n \le x \le m,\ m,n \in Z\}$;
   б) множество целых чисел $X=\{x \mid x = n+2i,\ i=0,1,\ldots,m,\ n,m \in Z\}$;
   в) множество двухсимвольных строк, составленных из символов 'a','b',...,'z';
   г) множество кодов символов, полученных с помощью дерева кодов Хаффмана, высота которого равна h.
2. Написать функции, реализующие операции Makenull, Equal, Assign, Max, Min, Find над множествами в векторном представлении.
3. Определить подходящую структуру данных и написать функции, реализующие операции Delete, Insert, Member и Restruct (реструктуризация) для множества, представленного посредством хеш-таблицы с прямым хешированием и случайной функцией повторного хеширования. Для генерации случайных чисел использовать метод последовательных сдвигов регистра.
4. Даны два текстовых файла, состоящих из попарно различных строк фиксированной длины. Количество строк и их длина равны соответственно N_1, L_1 и N_2, L_2. Используя метод расширенного хеширования, создать множество, являющееся результатом декартова произведения двух множеств, элементами которых являются строки исходных файлов.
5. Даны две последовательности целых чисел. Получить общую для них подпоследовательность наибольшей длины. Считать, что любая подпоследовательность получается из исходной путем выборки элементов (не обязательно соседних) с сохранением порядка их следования.
6. Написать функцию, реализующую операцию удаления элемента из 2-3-дерева.
7. Для двух множеств, представленных посредством 2-3-деревьев, написать функции, реализующие операции Union, Intersection и Merge. Считать, что мощности множеств различны.
8. Дан символьный файл, состоящий из N символов. Используя структуру нагруженного дерева, определить содержащиеся в файле различные 2-, 3-,... (N – k)-символьные подстроки, где $k \le N$ – заданное целое число, и сколько раз каждая из них встречается в исходном файле. Оценить объем памяти, необходимый для хранения узлов нагруженного дерева, и сравнить полученную величину с объемом памяти, которая потребовалась бы для организации хранения всех этих подстрок в куче с помощью хеш-таблицы.
9. Для файла, организованного в виде B-дерева и состоящего из записей с ключами, написать функции, реализующие операции Delete, Insert, Member, а также операцию Range, при выполнении которой получается список записей исходного файла, ключи которых попадают в заданный диапазон значений.
10. Дан файл, содержащий текст программы на языке C. Создать словарь ключевых слов языка C, реализованный посредством нагруженного дерева, и с его помощью получить список всех пользовательских идентификаторов, определенных в программе, и определить их среднюю длину.
11. Дана действительная матрица A размера $n \times 2$ и квадратная целочисленная матрица B порядка n, элементы которой равны 0 или 1. Считая, что пары элементов в строках матрицы A задают на плоскости координаты n точек, а матрица B является матрицей соединений между точками ($b_{ij} = 1$, если i-ая и j-ая точки соединены отрезком), определить, можно ли попасть из одной точки в другую, пройдя по соединительным линиям, и, если можно, найти путь минимальной длины.

ЛИТЕРАТУРА

1. Ахо Альфред В., Хопкрофт Джон, Ульман Джеффри Д. Структуры данных и алгоритмы. : Пер. с англ. – М. : Издательский дом «Вильямс», 2001. – 384 с.
2. Кнут Д.Э. Искусство программирования, том 1: Основные алгоритмы, 3-е изд. : Пер. с англ. : Уч. пос. – М.: Издательский дом «Вильямс», 2000. – 720 с.
3. Кнут Д.Э. Искусство программирования, том 3: Сортировка и поиск, 2-е изд. : Пер. с англ. : Уч. пос. – М.: Издательский дом «Вильямс», 2000. – 832 с.
4. Вирт Н. Алгоритмы и структуры данных: Пер. с англ. – М.: Мир, 1989. – 360 с.
5. Томас Х. Кормен, Чарльз И. Лейзерсон, Рональд Л. Ривест. Алгоритмы: построение и анализ. 2-е издание. – М.: МЦМНО, 2004. – 960 с.
6. Брайан Керниган, Роб Пайк. Практика программирования. – К.: Издательский дом «Вильямс», 2004. – 288 с.
7. Керниган Б. и др. Язык программирования Си. 3-е изд. – С-Пб.: Невский диалект, 2001. – 352 с.
8. Давыдов В. Программирование и основы алгоритмизации. – М.: Высшая школа, 2003. – 447 с.
9. Катков В.Л., Любимский Э.З. Программирование: Учеб. пособие для вузов. – Мн. : Выш. шк., 1992. – 295 с.
10. Касаткин А.И., Вальвачев А.Н. Профессиональное программирование на языке Си: От Turbo C к Borland C++. – Мн.: Выш. шк., 1992. – 240 с.
11. Касаткин А.И. Профессиональное программирование на языке Си: Управление ресурсами. – Мн.: Выш. шк., 1992. – 432 с.
12. Сигорский В.П. Математический аппарат инженера. Изд. 2-е, стереотип. – К.: Издательство «Техніка», 1977. – 768 с.

Учебное издание

Мороз Олег Васильевич

АБСТРАКТНЫЕ ТИПЫ ДАННЫХ: РЕАЛИЗАЦИЯ НА ЯЗЫКЕ С

Рекомендовано Советом Брестского государственного технического университета в качестве пособия для студентов специальностей 1-40 02 01 «Вычислительные машины, системы и сети» и 1-36 04 02 «Промышленная электроника»

ISBN 985-493-022-X

9 789854 930220

Ответственный за выпуск: ***Мороз О.В.***
Редактор: ***Строкач Т.В.***
Компьютерная верстка: ***Боровикова Е.А.***
Корректор: ***Никитчик Е.В.***

Издательство Учреждения образования «Брестский государственный технический университет».
Лицензия № 02330/0133017 от 30.04.2004 г.
Подписано к печати 10.10.2005 г. Формат 60x84 1/16. Бумага писчая. Гарнитура Arial Narrow.
Усл. п. л. 4,4. Уч.-изд. л. 4,75. Тираж 150 экз. Заказ № 990. Отпечатано на ризографе Учреждения образования «Брестский государственный технический университет».
224017, Брест, ул. Московская, 267.
Лицензия № 02330/0148711 от 30.04.2004 г.