```
class sparse_matrix
{
 private:
   .....
 public:
   sparse_matrix (int m, int n);
  ~sparse_matrix ();
   double & x(int i, int j);    /* элемент матрицы */
}
```

3.37. Добавьте в класс `sparse_matrix` из предыдущей задачи переопределения арифметических операций с матрицами (сложение, вычитание, умножение).

3.38. На основе реализации разреженной матрицы постройте электронную таблицу. Смысл задачи состоит в автоматической модификации значений некоторых элементов матрицы при изменении значений других элементов. Простейшим примером может служить числовая матрица, в которой последний столбец содержит суммы элементов соответствующих строк, а последняя строка содержит суммы элементов из соответствующих столбцов. При изменении элементов матрицы эти суммы соответствующим образом модифицируются. Для повышения универсальности реализации можно продумать способы предоставления пользователю возможности менять алгоритмы пересчета элементов матрицы.

3.3 Деревья

Прежде чем формулировать задачи, касающиеся деревьев, напомним ряд необходимых понятий.

Дерево — это структура вполне соответствующая обычному математическому понятию дерева, как связного планарного (плоского без пересечений ребер) графа, не содержащего циклов и имеющего одну выделенную вершину — корень дерева. Связь между отдельными вершинами дерева обычно описывается в терминах "родитель—потомок". Каждая вершина дерева, кроме корня, связана с одной и только одной родительской вершиной и с некоторым (возможно, нулевым) числом вершин-потомков. При этом каждая вершина является родительской для своих потомков. Корень не имеет родительской вершины.

Будем говорить, что корень составляет нулевой уровень дерева. Непосредственные потомки корня составляют первый уровень дерева. Непосредственные потомки вершин k-го уровня образуют $k+1$-й уровень дерева.

Если каждая вершина дерева имеет не более двух потомков, то это дерево называется бинарным, в противном случае — произвольным (иногда используется термин “сильно ветвящееся дерево”). Для бинарного дерева можно естественным образом ввести понятия левого и правого поддеревьев. Например, мы можем иметь дерево, представленное такой схемой:

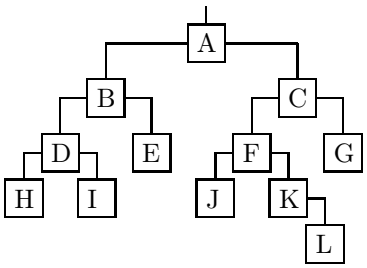

Рис. 3.3.1. Пример бинарного дерева.

Вершина дерева называется концевой (или листом), если она не имеет потомков. Длиной ветви называется количество вершин в связном участке дерева от корня до концевой вершины. Длиной (глубиной) дерева называется длина максимальной ветви в этом дереве. Дерево называется сбалансированным, если в любой его вершине длины левого и правого поддеревьев отличаются не более, чем на единицу. Дерево называется идеально сбалансированным, если длины двух любых его ветвей отличаются не более, чем на единицу.

Существует целый класс прикладных задач, где требуется обеспечить эффективный поиск и размещение данных, которые имеют между собой некоторое отношение порядка. Простейшим примером может служить набор чисел с естественными отношениями “больше” или “меньше”. Если мы рассмотрим упорядоченный числовой массив, то процедура поиска конкретного значения в таком массиве может быть реализована методом деления пополам с вычислительной сложностью (т.е. числом арифметических операций, необходимых для выполнения работы) порядка $O(\log_2 N)$, где N — количество элементов в массиве. Однако, процедура вставки элемента в упорядоченный массив требует уже $O(N)$ операций, так как при этом нужно “раздвинуть” элементы массива, чтобы освободить место для нового элемента. Добавление элемента в неупорядоченный массив выполняется всего за $O(1)$ действий (элемент просто приписывается в конец массива), но поиск уже приходится выполнять последовательным просмотром, что требует $O(N)$ действий. Эти две крайние точки в оценках трудоемкости добавления и поиска вызывают желание построить реализацию хранения и поиска набора данных с “промежуточными” характеристиками трудоемкости. Идеальным средством для этого служат бинарные деревья поиска.

Пусть данные, хранящиеся в вершинах дерева, обладают отношением порядка (т.е. их можно сравнивать между собой на “больше—меньше”). Бинарное дерево называется упорядоченным (или деревом поиска), если для любой вершины все элементы левого поддерева меньше либо равны элемента, расположенного в этой вершине, а все элемента правого поддерева строго больше элемента из этой вершины.

С деревьями обычно связаны следующие задачи.

- Формирование дерева по указанным правилам. Важным частным случаем здесь является работа с обычным или сбалансированным дере-

вом поиска с возможностью добавления, поиска и удаления заданного элемента.

- Обход дерева — требуется выполнить некоторую заданную операцию для каждой вершины дерева. Для бинарных деревьев существует три принципиально различных способа обхода — сверху-вниз (сначала обрабатывается текущая вершина, а затем ее левое и правое поддеревья), снизу-вверх (сначала обрабатываются поддеревья, а затем текущая вершина) и слева-направо (сначала обрабатывается левое поддерево, затем вершина, затем правое поддерево). Процедуры обходов можно обобщить на случай произвольных деревьев.

3.3.1 Бинарные деревья.

Для реализации бинарных деревьев (например, с элементами целого типа) удобно использовать структуру

```
struct TreeNode {
        int val;
        struct  TreeNode *left, *right;
};
```

Если в алгоритме необходимо явное перемещение по направлению к корню, то в эту структуру можно добавить ссылку на родительскую вершину: `struct  TreeNode *up;`. В рекурсивных процедурах обработки дерева подобное движение обычно реализуется автоматически за счет возврата из последовательности рекурсивных вызовов, поэтому при использовании таких алгоритмов указатель `up` не нужен.

Добавление элемента в бинарное упорядоченное дерево.

Здесь следует сделать одно замечание. Мы можем подходить к формированию дерева с двух различных позиций, а именно, дублировать или не дублировать элементы с одинаковыми ключами. В первом случае процедура добавления будет всегда включать в дерево новый элемент, во втором случае при обнаружении вершины с тем же ключем процедура будет просто завершаться (раз такой элемент уже есть — еще раз его добавлять не надо). Наш пример будет относиться к первому варианту.

3.39. Постройте нерекурсивную процедуру добавления элемента в бинарное дерево поиска.
Решение.

```
struct TreeNode  * Tree_Add  (int x, struct TreeNode *root)
{
   strcut TreeNode  * cur, *node;
   node = Tree_CreateNode();
   if (node==NULL) return NULL;
      node->value = x;
      node->right = node->left = NULL;
   if (root==NULL) return node;
```

```
   cur = root;
   while(1){
      if ( x>cur->value ){
         if( cur->right==NULL){
           cur->right=node;
          }
          else{
           cur=cur->right;
           }
       }
       else{
         if( cur->left==NULL){
           cur->left=node;
          }
          else{
           cur=cur->left;
           }
      }
  }
  return root;
}
struct TreeNode  * Tree_CreateNode(){
 return (struct TreeNode  *)malloc(sizeof(struct TreeNode));
 }
```

3.40. Измените предложенную реализацию, объединив ветви алгоритма, отвечающие за "подклеивание" нового узла.

Указание. Возьмите за основу конструкцию следующего типа:

```
    struct TreeNode  ** pnode;
    pnode=&root;
.....................................
       while(cur!= NULL){
       if ( x>cur->value ){
           pnode = &(cur->right);
           cur=cur->right;
       }
      else{
           pnode = &(cur->left);
           cur=cur->left;
       }
    }
    (*pnode)=cur;
    return root;
```

В данном случае переменная `pnode` хранит адрес ячейки, куда по завершении цикла `while(1){...}` запишется адрес созданного узла.

Нетрудно построить и нерекурсивную процедуру, хотя выглядит она менее элегантно.

```
TreeNode  * AddElement (Type x, TreeNode *root)
```

```
{  TreeNode *pos,*old;
   int  dir;
   if ( !root ) { // пустое дерево
      pos = CreateNode();
      if ( pos ) {
         pos->value = x; pos->left = pos->right = 0;
      }
      return pos;
   }
   // для непустого дерева ищем позицию добавления
   for ( pos=root; pos; ) {
      old = pos;
      if ( x<=pos->value ) { pos=pos->left; dir=0; }
      else                 { pos=pos->right; dir=1; }
   }
   // создадим вершину
   if ( !(pos=CreateNode()) ) return 0;
   pos->value = x;
   pos->left = pos->right = 0;
   // добавим ее в дерево
   if (dir) old->right = pos;
   else     old->left = pos;
   return  root;
}
```

Указатель `old` определяет родительскую вершину по отношению к добавляемому элементу. Поскольку мы отказались от рекурсии, то нам приходится отслеживать где, слева или справа, будет присоединен новый элемент к этой вершине. Для этого вводится переменная `dir`, которая хранит направление последнего сделанного поворота при спуске по ветви (0 — налево, 1 —направо). Последующая проверка значения переменной `dir` позволяет правильно установить ссылки от родительской вершины.

3.41. Постройте рекурсивную процедуру добавления элемента в бинарное дерево поиска.

Идеи реализации. Начиная с корня, определяем, в какое поддерево должен попасть данный элемент, и рекурсивно вызываем процедуру добавления для требуемого поддерева. Возвращаемое значение функции — указатель на вершину текущего уровня, если элемент был добавлен, или `NULL`, если не удалось добавить.
Решение.

```
struct TreeNode  * Tree_Add  (int x, struct Tree *cur)
{
   if (!cur)
   {  if ( !(cur=Tree_CreateNode()) ) return NULL;
      cur>value = x;
      cur->right = cur->left = NULL;
   return cur;
   }
```

```
   else
   {  if ( x>cur->value ){
        cur->right = Tree_Add (x,cur>right);
      }
      else{
        cur->left  = Tree_Add (x,cur->left);
      }
   return cur;
   }
}
```

Относительно параметров и возвращаемого значения примем для определенности следующие решения: параметры — добавляемый элемент и указатель на корень дерева, возвращаемое значение — указатель на корень всего дерева, либо нуль, если добавить не удалось. Для создания новой вершины будем по аналогии со списками использовать некоторую функцию `CreateNode()`, возвращающую при каждом обращении указатель на созданную вершину.

```
TreeNode  * AddElement (Type x, TreeNode *root)
{  TreeNode  *pos;
   if ( !root )  {  // пустое дерево
      root = CreateNode();
      if ( root ) {
         root->value = x; root->left = root->right = 0;
      }
   } else {  // непустое дерево -> рекурсия
      if ( x<=root->value ) {
         pos = AddElement (x,root->left);
         if (pos) root->left = pos; else  return 0;
      } else {
         pos = AddElement (x,root->right);
         if (pos) root->right = pos; else  return 0;
      }
   }
   return root;
}
```

3.42. В рассмотренных выше процедурах добавления в дерево поиска элементы с одинаковыми значениями могут оказаться в дереве далеко друг от друга. Измените процедуры добавления так, чтобы элементы с одинаковыми значениями всегда оказывались рядом друг с другом (т.е. образовывали в ветви линейную цепочку).

Печать элементов дерева.

3.43. Реализуйте три упомянутые выше процедуры обхода и печати элементов бинарного дерева, предполагая, что в вершинах хранятся целые числа (`Type` есть `int`).

Решение. Пусть корень указанного дерева определяется указателем `root`. Тогда процедуры могут иметь следующий вид.

```
void Tree_PrnUpDown (struct TreeNode *root)
{
   if (!root) return;
   printf(``%d '', root->val);
   Tree_PrnUpDown(root->left);
   Tree_PrnUpDown(root->right);
   return;
}
void Tree_PrnDownUp (struct TreeNode *root)
{
   if (!root) return;
   Tree_PrnDownUp(root->left);
   Tree_PrnDownUp(root->right);
   printf(``%d '', root->val);
   return;
}
void Tree_PrnLeftRight (struct TreeNode *root)
{
   if (!root) return;
   Tree_PrnLeftRight(root->left);
   printf(``%d '', root->val);
   Tree_PrnLeftRight(root->right);
   return;
}
```

Каждая из этих функций обхода перебирает вершины дерева по-своему. Например, для дерева, изображенного на рис. ????, последовательность обработки вершин будет следующей:

для обхода сверху-вниз A B D H I E C F J K L G;
для обхода снизу-вверх H I D E B J L K F G C A;
для обхода слева-направо H D I B E A J F K L C G.

Разберите работу программ считая, что дерево поиска было построено по входной последовательности вида 8, 15, 5, 6, 2, 3, 7, 9, 13, 26, 23.

Отметим, что в этих функциях нигде не используется ссылка на родительский элемент (`up`). Действительно, эта ссылка нужна для перемещения по дереву по направлению к корню. Однако в приведенных выше процедурах это перемещение происходит "автоматически" за счет запоминания текущего локального значения параметра `pos` в системном стеке при организации последовательности рекурсивных вызовов. Таким образом, при конкретной реализации дерева указатель `up` не нужен, если обработка вершин дерева производится с помощью рекурсивных процедур. Включать или нет этот указатель в описание структуры вершины дерева зависит от решаемых задач и способов обработки вершин дерева. Следует сказать, что экономия памяти за счет исключения указателя `up` в случае очень "длинных" деревьев может вызвать определенные трудности при выполнении программы из-за переполнения системного стека в связи с большой вложенностью рекурсивных вызовов. Поэтому, несмотря на то, что рекурсивные алгорит-

мы зачастую выглядят более элегантными и короткими, в ряде случаев более эффективными могут оказаться нерекурсивные алгоритмы обработки вершин дерева.

3.44. Постройте три упомянутые выше процедуры обхода для подсчета суммы элементов бинарного дерева, предполагая, что в вершинах хранятся целые числа.

Указание. Например:

```
int  TreeSumUpDown (Tree *root)
{
   if (!root) return 0;
   return  root->val+TreeSumUpDown(root->left)+TreeSumUpDown(root->right);
}
```

Поиск элемента в бинарном упорядоченном дереве.

3.45. Постройте рекурсивную и нерекурсивную процедуру поиска элемента в бинарном дереве поиска. Функция поиска должна возвращать указатель на найденный элемент или `NULL`, если такого элемента в дереве нет.

Решение. Реализуем рекурсивную процедуру поиска данного элемента в поддереве. В качестве параметров этой процедуры разумно будет выбрать указатель на стартовую вершину поиска, сам искомый элемент, а возвращаемое значение — указатель на найденную вершину с требуемым ключем, либо `NULL`, если такая вершина отсутствует в дереве.

```
TreeNode  *RecurSearch (Type x, TreeNode *start)
{  if ( !start ) return 0;   // дерево пусто
   if ( x == start->value ) return start;   // нашли
   if ( x < start->value )  return RecurSearch(x,start->left);
   else                     return RecurSearch(x,start->right);
}
```

Нерекурсивный вариант этой процедуры выглядит еще проще.

```
TreeNode  *DirectSearch (Type x, TreeNode *start)
{  while ( start ) {
      if ( x == start->value ) break;
      start = ( x < start->value ) ?
         start->left : start->right;
   }
   return start;
}
```

3.46. Постройте подходящие процедуры обхода бинарного (но не обязательно упорядоченного) дерева с целочисленными элементами для решения следующих задач:

- Распечатайте все концевые элементы (листья).
- Вычислите количество концевых элементов.
- Распечатайте элементы, лежащие на уровне k.

- Модифицируйте функции печати из задачи 3.43 так, чтобы элементы k-го уровня распологались в k-ом столбце.
- Выведите наглядное представление дерева целых чисел в текстовой файл.
- Определите количество элементов, лежащих на уровне k.
- Найдите максимальный элемент среди всех элементов k-го уровня.
- Найдите сумму концевых элементов.
- Найдите сумму элементов, лежащих на уровне k.
- Определите глубину дерева (т.е. длину максимальной ветви).
- Определите количество ветвей, имеющих максимальную длину.
- Проверьте является ли бинарное дерево сбалансированным.
- Подсчитайте число несбалансированных вершин в бинарном дереве.
- Подсчитайте показатель сбалансированности для бинарного дерева (т.е. максимальную разницу между длинами правого и левого поддеревьев для каждой вершины).
- Распечатайте ветвь дерева с заданным значением листа.
- Найдите поддерево, для которого сумма элементов совпадает с указанным числом.
- Определите длину связного пути между двумя вершинами неупорядоченного дерева, хранящими указанные значения.

3.47. Пусть построено бинарное дерево на основе структуры

```
struct TreeNode {
        Type val;
        struct  TreeNode *left, *right;
        strruct TreeNode *next;
};
```

При этом все указатели `next` равны `NULL`. Напишите функцию, получающую на вход указатель на корень дерева, и связывающую вершины каждого уровня в однонаправленный список по указателю `next`.

Идеи реализации. Реализуйте рекурсивный алгоритм, настраивающий список (k+1)-го уровня при условии, что k-й уровень уже настроен.

Удаление элемента в бинарном упорядоченном дереве.

Теперь обсудим алгоритм удаления элемента. Если элемент является концевым или имеет только одного потомка, то для удаления достаточно перенести ссылку от родительской вершины на соответствующего потомка.

В случае с двумя потомками удалить элемент так просто не удастся, поскольку здесь уже имеются три ссылки (от родителя к вершине и от нее к двум потомкам), и их невозможно замкнуть друг на друга. Идея удаления состоит в том, что удаляемая вершина как структурный элемент дерева на самом деле остается на месте, но данные, хранящиеся в этой вершине, заменяются на другие. Поясним эту идею на примере. Пусть мы хотим удалить элементы с ключами 5 и 12 из дерева, изображенного на следующем рисунке.

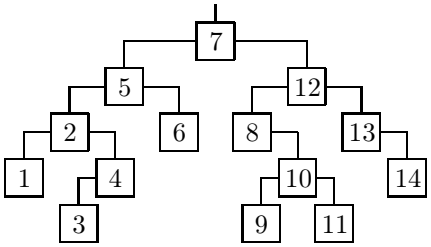

Рис. 3.3.2. Дерево до удаления элементов.

Найдем в дереве вершины, содержимое которых можно подставить вместо удаляемых данных так, чтобы дерево сохранило свойство упорядоченности (осталось деревом поиска). В нашем примере этими вершинами, очевидно, будут вершины с ключами 4 (для 5) и 11 (для 12). Перенесем данные из этих вершин в "удаляемые" вершины (с ключами 5 и 12).

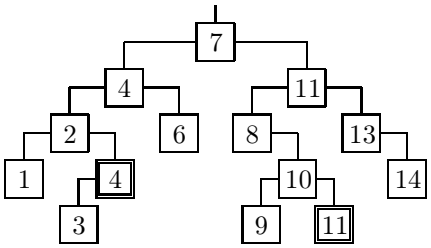

Рис. 3.3.3. Сделана замена значений, должны быть удалены помеченные вершины.

Теперь можно удалить вершины (отмеченные двойной рамкой), хранившие эти данные ранее, поскольку они имеют не более одного потомка.

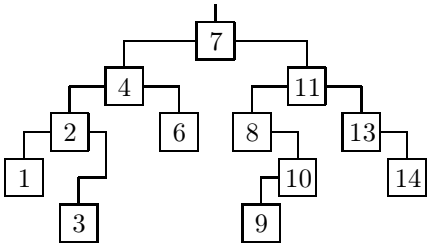

Рис. 3.3.4. Дерево после удаления помеченных вершин.

Анализируя рассмотренный пример, мы приходим к следующему неформальному алгоритму удаления.

1. Если удаляемая вершина является концевой (не имеет потомков), то достаточно освободить память, занимаемую этой вершиной, и обнулить соответствующую ссылку от родительской вершины.
2. Если удаляемая вершина имеет ровно одного потомка, то после удаления вершины из памяти ссылка от родительской вершины переставляется на этого потомка.
3. В случае двух потомков сначала в левом поддереве удаляемой вершины ищется вершина с максимальным ключем. Заметим, что эта "максимальная" вершина обязательно является либо концевой, либо имеет только одного потомка. Для того, чтобы ее найти, достаточно спускаться по самой правой ветви левого поддерева до тех пор, пока не обнаружится нулевая ссылка направо. Найдя нужную "максимальную" вершину, переносим данные из нее в "удаляемую" вершину. Теперь "максимальную" вершину можно удалить, что выполняется с помощью пунктов 1 или 2.

Заметим, что пункты 1 и 2 можно реализовать одной и той же последовательностью операторов, а пункт 3 следует рассматривать не столько как удаление элемента динамической структуры данных, сколько как удаление указанных данных из этой структуры.

Реализация рассмотренного алгоритма приводит к следующей программе.

```
TreeNode  * DelElement (Type x, TreeNode *root)
{  TreeNode *pos;
   // пустое дерево
   if ( !root )  return 0;
   // непустое дерево
   if ( x==root->value ) {  // если удаляется корень
      // не более одного потомка
      if ( !root->left ) {
         pos = root->left;  delete root;
         return pos;
      }
      if ( !root->right ) {
         pos = root->right; delete root;
         return pos;
      }
      // два потомка, ищем вершину для подмены
      for ( pos=root->left; pos->right; ) pos = pos->right;
      root->value = pos->value;
      // удаляем не нужную уже вершину для подмены
      root->left = DelElement(pos->value,root->left);
   } else {  // рекурсия, если удаляется не корень
```

```
      if ( x<root->value )
         root->left = DelElement(x,root->left);
      else
         root->right = DelElement(x,root->right);
   }
   return  root;
}
```

3.48. Реализуйте рекурсивную процедуру удаления элемента из бинарного дерева поиска.

Идеи реализации. Если удаляемая вершина имеет не более одного потомка, то удаление сводится к перестановке ссылки от родительской вершины на соответствующего потомка (возможно пустого). Если удаляемая вершина имеет два потомка, то сначала в левом поддереве ищется вершина с максимальным значением поля `val` (обозначим эту вершину через M), это значение записывается в поле `val`, удаляемой вершины и далее из дерева удаляется вершина M (докажите, что вершина M имеет не более одного потомка). Подробный анализ алгоритма удаления изложен в разделе 3.47.
Решение. Возвращаемое значение функции — указатель на корень дерева после удаления элемента.

```
struct TreeNode * TreeDel(int x, struct TreeNode * cur){
struct TreeNode * tmp;
if(cur==NULL){
  return NULL;
  }
if(cur->value == x){
  if(cur->left == NULL && cur->right == NULL){
     return NULL;
    }
if(cur->left == NULL){
return cur->right;
  }
if(cur->right == NULL){
return cur->left;
}

tmp=cur->left;
while(tmp->right != NULL){
  tmp=tmp->right;
}
 cur->value = tmp->value;
 cur->left = Tr_Del(tmp->value,cur->left);
 return cur;
}
else{
   if(x < cur->value){
   cur->left = Tr_Del(x,cur->left);
   return cur;
```

```
}
else{
 cur->right = Tr_Del(x,cur->right);
 return cur;
 }
}
}
```

3.49. Постройте нерекурсивную процедуру удаления элемента в бинарном дереве поиска.

3.50. Если в алгоритме требуется необходимо явное перемещение по направлению к корню, то в структуру стоит добавить ссылку на родительскую вершину. Постройте процедуры добавления, удаления, поиска и печати элементов для бинарного дерева поиска с элементами абстрактного типа `Type` при условии, что связи между вершинами задаются тремя указателями:

```
typedef struct _TreeNode {
        Type val;
        struct _TreeNode *left, *right, *up;
} TreeNode;
```

3.51. Пусть концевые вершины бинарного дерева содержат числа, а всем остальным вершинам дополнительно сопоставлены арифметические операции (в начальный момент числовые значения во всех остальных вершинах не определены). Назовем вычислением по дереву процедуру, записывающую в родительскую вершину значение, полученное в результате выполнения указанной арифметической операции со значениями в корнях его поддеревьев. Разработайте форму представления такого дерева и реализуйте процедуру вычисления по дереву.

3.3.2 Сильно ветвящиеся деревья.

Вершины произвольных деревьев удобно реализовывать с помощью структуры с двумя ссылками:

```
typedef struct _TreeNode {
        Type val;
        struct _TreeNode *down, *next;
} TreeNode;
```

В этом случае смысл указателей несколько меняется: все непосредственные потомки одной вершины связываются в однонаправленный список по указателю `next`, а указатель `down` указывает на начало списка потомков данной вершины.

Отсутствие следующей вершины в списке потомков обычно обозначается нулевым значением соответствующего указателя (`next` или `down`).

Идея состоит в следующем. Возьмем всех потомков данной конкретной вершины и свяжем их в однонаправленный список в том порядке, в котором они появляются в дереве. В родительскую вершину поместим указатель

на начало этого списка. Таким образом, в каждой вершине будут храниться два указателя: на следующий элемент в списке "братьев" и на начало списка потомков данной вершины. При необходимости прямого перемещения назад к корню, можно добавить еще один указатель на родительскую вершину. Как всегда, нулевое значение указателя соответствует отсутствию последующего элемента (т.е. концу списка потомков или концу ветви).

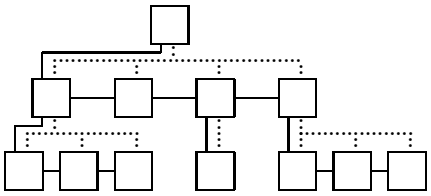

Рис. 3.3.5. Схема ссылок в произвольном дереве.

На этом рисунке пунктирными линиями изображены логические связи между вершинами исходного дерева, а сплошными линиями — связи, определяемые указателями, хранящимися в реализации дерева. Определение типа `TreeNode` здесь может выглядеть, например, так

```
class TreeNode
{   public:
    Type   value;
    TreeNode *next, *down;
};
```

Здесь указатель `next` отвечает за связь "братьев", а указатель `down` указывает на список потомков данной вершины.

Процедуры обхода произвольного дерева строятся по аналогии с бинарным деревом. Правда, обход слева-направо теряет смысл, так как неясно в какой момент (т.е. между какими потомками) следует обрабатывать родительскую вершину. Приведем для примера процедуру обхода сверху-вниз для той же задачи, что и раньше (для вычисления максимума).

```
void  Up_Down (TreeNode *pos, Type *max)
{  if (!pos) return ;
   if ( *max < pos->value ) *max=pos->value;
   pos = pos->down;
   while (pos) {
      Up_Down (pos, max);
      pos = pos->next;
   }
}
```

Отметим, что чисто топологически реализация произвольного дерева соответствует реализации некоторого бинарного дерева (так как каждый элемент имеет две ссылки на "последующие" элементы). Поэтому в тех задачах

обходов произвольных деревьев, где не требуется отслеживать принадлежность элементов определенному уровню, можно применять алгоритмы обходов бинарных деревьев.

3.52. Постройте аналоги процедур обхода сверху-вниз и снизу-вверх для произвольных деревьев.

3.53. Пусть произвольное дерево построено по следующим правилам: каждая ветвь дерева, начиная с первого уровня, соответствует некоторому слову (текстовой строке); вершины k-го уровня дерева соответствуют k-й букве в записи слова; значением вершины является пара

```
struct{char sym; char flag;},
```

где `sym` — очередная буква слова, а `flag=1`, если существует слово, оканчивающееся на этой букве, и `flag=0`, если слово продолжается дальше. Например, ветвь, задающая слово "столовая", имеет `flag=1` для первой буквы "о" и букв "л" и "я"; каждый горизонтальный список потомков одной вершины упорядочен по значению поля `sym` (по алфавиту). Требуется построить процедуры добавления, удаления и поиска слова в таком дереве.

3.3.3 Сбалансированные бинарные деревья.

Во многих задачах требуется обеспечить эффективный поиск и размещение данных, которые имеют между собой некоторое отношение порядка. Простейшим примером может служить набор чисел с естественными отношениями "больше" или "меньше". Если мы рассмотрим упорядоченный числовой массив, то, как известно (метод деления пополам), процедура поиска конкретного значения в таком массиве может быть реализована с вычислительной сложностью (т.е. числом арифметических операций, необходимых для выполнения работы) порядка $O(\log_2 N)$, где N — количество элементов в массиве. Однако, процедура вставки элемента в упорядоченный массив требует уже $O(N)$ операций, так как при этом нужно "раздвинуть" элементы массива, чтобы освободить место для нового элемента. Добавление элемента в неупорядоченный массив выполняется всего за $O(1)$ действий (элемент просто приписывается в конец массива), но поиск уже приходится выполнять последовательным просмотром, что требует $O(N)$ действий. Эти две крайние точки в оценках трудоемкости добавления и поиска вызывают желание построить реализацию хранения и поиска набора данных с "промежуточными" характеристиками трудоемкости. Идеальным средством для этого служат бинарные деревья.

Трудоемкость процедур поиска, добавления и удаления элемента в произвольном бинарном дереве поиска оценивается по порядку величиной, равной длине максимальной ветви дерева. Если дерево содержит N элементов, то в наихудшем вырожденном случае эта величина равна N, и $\log_2 N$ в наилучшем случае. На данный момент разработаны алгоритмы, позволяющие организовать работу с деревом поиска (т.е. добавление и удаление элементов) так, чтобы оно всегда оставалось сбалансированным, при этом длина

максимальной ветви не будет превосходить величины $1.5 \log_2 N$. В данном случае нам придется ограничиться деревьями, все элементы которых различны (действительно, дерево, содержащее только одинаковые элементы, не может быть одновременно упорядоченным и сбалансированным в смысле приведенных выше определений).

3.54. Докажите, что длина сбалансированного дерева с N элементами не превосходит $1.5 \log_2 N$.

Для реализации сбалансированных деревьев нам придется модифицировать структуру его вершины. Назовем балансом вершины разность между значениями длин его правого и левого поддеревьев. Будем хранить значение баланса каждой вершины вместе с остальной информацией данной вершины, т.е. будем использовать, например, следующее определение типа вершины:

```
typedef struct _TreeNode {
        Type val;
        int  balance;
        struct _TreeNode *down, *next;
} TreeNode;
```

Для сбалансированного дерева значения поля `balance` в каждой вершине могут быть $-1, 0, 1$. Пусть мы имеем сбалансированное дерево поиска. Требуется организовать добавление и удаление элементов в это дерево так, чтобы сохранить сбалансированность дерева. Заметим, что если при добавлении элемента в некоторое поддерево длина этого поддерева не возросла, то баланс в вершине, родительской по отношению к этому поддереву, не изменился и, следовательно, балансировка не нужна. Если же длина поддерева возросла, то, возможно, придется перестраивать дерево для восстановления утраченного баланса. Алгоритмы балансировки деревьев при добавлении и удалении элементов описаны в [2], и мы их здесь не будем приводить.

3.55. Реализуйте функцию, которая проверяет является ли данное бинарное дерево а) идеально сбалансированным; б) просто сбалансированным.

3.56. Реализуйте функцию, которая обходит бинарное дерево (не обязательно сбалансированное) и записывает в поле `balance` значение баланса каждой вершины.

3.57. Реализуйте процедуры добавления и удаления элементов в сбалансированном бинарном дереве поиска элементов абстрактного типа `Type`.

3.3.4 B–деревья.

Встречаются ситуации, когда обрабатываемые наборы данных невозможно полностью разместить в оперативной памяти. В этом случае их частично приходится хранить на диске и по мере необходимости считывать в память, предварительно освобождая место и записывая обратно на диск не

используемые в данный момент данные. В такой постановке задачи самыми неэффективными по времени являются именно операции обмена с диском. Таким образом, возникает проблема минимизации количества обращений к диску, возможно, за счет некоторого увеличения накладных расходов по памяти и времени доступа для данных, уже находящихся в оперативной памяти.

Для достижения этой цели удобным средством являются так называемые B-деревья. Дадим определение этой структуры.

B-*деревом порядка* n называется древовидная структура, удовлетворяющая следующим условиям:

- вершиной дерева является массив, способный вместить $2n$ элементов данных;
- в каждой вершине элементы данных расположены в массиве вершины в порядке возрастания их ключей.
- каждая вершина, кроме корневой, содержит не менее n и не более $2n$ элементов данных;
- вершина, содержащая k $(n \le k \le 2n)$ элементов данных, имеет ровно $k+1$-го потомка, либо является концевой, т.е. не имеет потомков;
- если вершина имеет k элементов, то ключи всех элементов поддерева i-го потомка меньше ключа i-го элемента и больше ключа $(i-1)$-го элемента родительской вершины (для 0-го и k потомков рассматриваются только 0-й и $(k-1)$-й элементы, соответственно);
- все концевые вершины лежат на одном уровне дерева;

B-дерево можно рассматривать как обобщение идеально сбалансированного дерева поиска на случай множественного числа потомков.

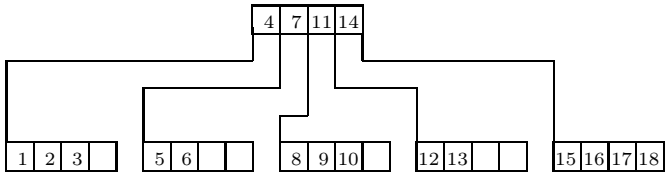

Рис. 3.3.6. Числовое B-дерево порядка 2, длины 2.

Рассмотрим для простоты ситуацию, когда порядок дерева n является некоторым фиксированным числом (например, 2048). В этом случае реализацию B-дерева порядка n можно построить на основе структуры

```
#define   n  2048
class BTreeNode
{  public:
   Type  value [2*n];        // значения элементов
   int   k;                  // количество элементов
   BTreeNode *child[2*n+1];  // ссылки на потомков
};
```

где массив `value` предназначен для хранения элементов данных, значение `k` есть фактическое количество элементов данных в этой вершине, массив `child` определяет ссылки на потомков.

При размещении B-дерева в памяти мы вынуждены резервировать место для $2n$ элементов в каждой вершине. Следовательно, возможны потери памяти, которые в наихудшем случае сравнимы с количеством размещенных элементов данных. Нетрудно подсчитать, что длина B-дерева, содержащего N элементов данных, не превосходит $O(\log_n N)$, что при $n > 2$ существенно меньше, чем у бинарного дерева. B-деревья обычно применяются в тех случаях, когда данные для каждой вершины B-дерева имеют значительный объем и изначально хранятся на диске. При работе с таким деревом его вершины считываются с диска по мере необходимости. Таким образом, доступ к любому элементу требует прохождения по ветви дерева от корня до требуемого элемента, что можно осуществить за $O(\log_n N)$ обращений к диску, считывая каждый раз массив данных длины $2n$.

Поскольку реализация B-дерева должна обеспечивать возможность считывания с диска данных для вершин-потомков, в структуру `BTreeNode` можно ввести еще массив ссылок на место размещения данных этих вершин-потомков на диске, например, в виде смещений от начала соответствующего дискового файла:

```
#define   n  2048
class BTreeNode
{  public:
   Type  value [2*n];        // значения элементов
   int   k;                  // количество элементов
   BTreeNode *child[2*n+1];  // ссылки на потомков
   unsigned long  file_offset[2*n+1]; // смещения в файле
};
```

В этом случае ненулевой указатель `child[k]` определяет местоположение соответствующей вершины-потомка в памяти, а при нулевом значении `child[k]` мы прочитаем вершину из файла по смещению `file_offset[k]`, разместим ее в памяти, а адрес этого размещения занесем в `child[k]`. Для концевой вершины можно считать, что все значения массива `file_offset` равны нулю.

Поиск элемента. *Опишем процедуру поиска элемента данных в B-дереве. Так как по условию построения B-дерева массив* `val` *в каждой вершине является упорядоченным, то мы можем методом деления пополам либо найти искомый элемент x, либо определить позицию, где x можно вставить в этот массив. Для такой позиции j имеем* `val[j-1]<x<val[j]` *(либо* `x<val[0]`*, либо* `val[k-1]<x`*). В этом случае мы рекурсивно переадресуем поиск на поддерево, определяемое указателем* `down[j]` *(соответственно* `down[0]`*, или* `down[k]`*). Если очередное поддерево пусто, то элемент отсутствует.*

Опишем алгоритм поиска конкретного элемента `x`. Будем считать, что в нашем распоряжении есть процедура бинарного поиска элемента в заданной вершине B-дерева

```
Type * BNodeSearch (Type x, BTreeNode *node, int *ind);
```

которая возвращает указатель на найденный элемент или 0 в случае неудачи, а по указателю `ind` записывается индекс найденного элемента в массиве в случае успеха, либо индекс потомка, в котором нужно далее искать `x`, т.е.

0, если `x<pos->value[0]`;
k, если `x>pos->value[pos->k-1]`;
i, если `pos->value[i-1]<x<pos->value[i]` для `0<i<pos->k`.

Для чтения вершины с диска и размещения ее в памяти у нас будет процедура `ReadNodeFromDisk`, которая имеет параметром требуемое смещение и возвращает адрес, по которому она разместила в памяти прочитанную вершину.

Процедура поиска элемента, которую мы собираемся построить, получает в качестве параметров искомый элемент `x`, стартовую вершину поиска `node` и возвращает указатель на найденный элемент, или 0 при отсутствии такового.

```
Type  * BTSearch (Type x, BTreeNode *node)
{
   int   ind;
   Type *find;

 //  для пустого дерева ничего не делаем
   if ( !node )  return 0;

 // пытаемся найти требуемый элемент
   if ( find = BNodeSearch(x,node,&ind) )
     // нашли в корневой вершине
      return find;
   else
   { // переходим к поиску в потомке,
      if ( !node->child[ind] )
      {  // потомка нет в памяти, надо считать его с диска
         if ( !node->offset[ind] )
            // концевая вершина
            return 0;
         else
         { // пытаемся прочитать с диска
            node->child[ind]
              = ReadNodeFromDisk (node->offset[ind]);
           // проверим, удалось или нет
            if ( !node->child[ind] ) return 0;
         }
      }
     // теперь потомок в памяти, ищем в нем
      return BTSearch(x,node->child[ind]);
   }
}
```

Не принимая во внимание действий по определению файла с данными и различных проверок корректности, использование этой процедуры может выглядеть, например, так:

```
BTreeNode * root;
Type * find;
root = ReadNodeFromDisk(0);
find = BTSearch(x,root);
```

(Здесь предполагается, что данные корневой вершины записаны в самом начале дискового файла.)

Нетрудно подсчитать, что сложность поиска в B-дереве порядка n с N элементами, в наихудшем случае сводится к проходу от корня до концевой вершины с выполнением бинарного поиска в вершинах каждого уровня, что составляет $O(\log_2(2n)\log_n N) \sim O(\log_2 N)$ арифметических операций и сравнений.

Добавление элемента. *Опишем схему добавления элемента в B-дерево поиска. Сначала отыскивается концевая вершина, в которую попадает добавляемый элемент. Если в массиве* `val` *этой вершины есть свободное место (т.е. $k < 2m$), то элемент добавляется в массив так, чтобы сохранить упорядоченность. В противном случает требуется перестройка узла и, возможно, всего дерева.*

Первый способ перестройки заключается в выталкивании элемента `val[m]` *в родительскую вершину и расщеплении исходной вершины на две части: с элементами* `val[i]` *для $0 \le i < m$ и элементами для $m+1 \le i < 2m$. Массив данных и массив ссылок на поддеревья в родительской вершине корректируются соответствующим образом. Если при этом происходит переполнение родительской вершины, то ее расщепляем аналогичным образом, и т.д. (возможно, вплоть до корня). При расщеплении корня создается новый корень.*

Второй способ заключается в попытке перемещения части "лишних" данных в соседнюю правую (либо левую) вершину. При этом необходимо модифицировать родительскую вершину. Если соседние вершины заполнены, то производится расщепление первым способом.

Третий способ заключается в объединении данных двух соседних вершин и их расщеплении в три новых вершины. При этом необходимо модифицировать родительскую вершину. Если у расщепляемой вершины нет соседних, то это — корень. Корень расщепляется на два узла. Обсудим теперь идею алгоритма добавления элемента. Сначала отыскивается концевая вершина, в которую можно добавить требуемый элемент в соответствии с упорядоченностью существующих элементов. Если добавление элемента в массив этой вершины не превышает допустимый размер (напомним, что это $2n$), то выполняется вставка в упорядоченный массив, и на этом процесс завершается. Если в концевой вершине (обозначим ее A) уже размещено $2n$ элементов, то, добавляя новое значение, мы получаем упорядоченный набор из $2n+1$ элементов. По определению B-дерева такое количество элементов не может быть размещено в вершине. Создается новая концевая вершина

(обозначим ее B), и первые n элементов набора размещаются в старой вершине, последние n — во вновь созданной вершине, а средний элемент набора (обозначим его y) вставляется в родительскую вершину. При этом справа от y в родительской вершине вставляется ссылка на нового потомка B. Если прямое добавление y в родительскую вершину невозможно (она уже имела $2n$ элементов), то опять выполняется ее разделение на две вершины, а средний элемент переносится в вершину выше по ветви. При полной занятости всех вершин в ветви этот процесс переноса среднего элемента распространяется до корня. Если корень также заполнен, то он аналогично разделяется на две вершины, а средний элемент образует новый корень. В отличие от бинарных деревьев поиска, B-дерево растет "снизу-вверх". Самым трудоемким шагом в процедуре добавления является процесс разделения вершин, который требует "раздвигания" массивов данных для вставки нового элемента. Таким образом, наилучшая оценка трудоемкости составляет в среднем $O(\log_n N \log_2 n + n)$ операций (спуск по ветви и одна вставка), а наихудшая оценка — $O(n \log_n N)$ (происходит разделение вершин во всей ветви). Отметим, что последняя оценка не является такой уж обременительной, поскольку появляется много вершин, заполненных лишь наполовину, и последующие добавления уже долго не будут вызывать разделения вершин.

Удаление элемента. *Алгоритм удаления можно получить обращением описанной процедуры. Если при удалении элемента из массива* `val` *количество данных в нем станет меньше* m*, то необходима перестройка, которая может быть выполнена перемещением данных из соседних вершин или слиянием двух соседних вершин в одну новую. Более подробно алгоритмы работы с B-деревьями описаны в [5].*

Алгоритм удаления элемента из B-дерева также имеет общие черты с удалением из бинарного дерева. Если нужный элемент лежит в концевой вершине, то удаление не составляет труда. Если же этот элемент расположен в другой вершине, то сначала он заменяется на максимальный элемент из поддерева, берущего свое начало от "левой" ссылки удаляемого элемента. Нетрудно понять, что этот элемент лежит в концевой вершине. Теперь можно удалить этот максимальный элемент. В результате удаления элемента из концевой вершины (обозначим ее A) может оказаться, что количество элементов в ней станет равным $n-1$, т.е. нарушится одно из условий B-дерева. В этом случае производится балансировка, состоящая в объединении элементов вершины A, элемента y родительской вершины, для которого "левая" ссылка указывает на A и элементов соседней концевой вершины B (определяемой правой ссылкой от y). Полученный набор данных делится на равные (с точностью до одного элемента) "левую" и "правую" части, разделяемые "средним" элементом. Средний элемент заменяет y в родительской вершине, левая часть записывается в A, а правая — в B. Если в вершине B только n элементов, то такое слияние вершин невозможно (опять нарушится условие B-дерева). В этом случае весь объединенный набор (его длина равна $2n$) записывается в A, вершина B удаляется, и из родительской вершины также удаляется элемент y вместе со ссылкой на B. При удалении y из родительской вершины, число элементов в ней также может уменьшиться до $n-1$. В

этом случае аналогично перераспределяются элементы двух соседних вершин данного уровня (либо вершины сливаются в одну). Описанный процесс может распространиться до корня. Если количество элементов в корне при этом станет равным нулю (а это возможно только при слиянии двух его потомков), то корень удаляется, а новым корнем становится единственная объединенная при последнем слиянии вершина. Глубина дерева при этом уменьшается на единицу. Легко подсчитать, что сложность удаления совпадает со сложностью добавления и составляет от $O(\log_n N \log_2 n + n)$ до $O(n \log_n N)$ операций в зависимости от текущей заполненности вершин.

3.58. Реализуйте стандартные процедуры добавления, поиска, удаления для работы с B-деревом целых чисел для случая m=2, т.н. (2-3) дерево.

3.59. Реализуйте стандартные процедуры добавления, поиска, удаления для работы с B-деревом порядка m, содержащим элементы некоторого типа `Type`.

3.60. Разработайте формат файла для сохранения B-дерева на диске и реализуйте процедуры добавления, удаления и поиска элементов некоторого типа `Type` на базе B-дерева, при условии, что все дерево не помещается в оперативную память.

Идеи реализации. Пронумеруем все вершины дерева последовательными натуральными числами. В начале файла должна содержаться таблица, которая устанавливает соответствие между номером вершины и смещением от начала файла, определяющим местоположение этой вершины. Поскольку количество вершин в дереве может меняться, то следует предусмотреть способы для удлинения этой таблицы. За этой таблицей размещаются вершины B-дерева. Прочитав данную таблицу в оперативную память, мы получаем возможность считывания любой вершины по ее номеру.

Для хранения вершины B-дерева возьмем структуру следующего вида

```
typedef struct _BT {
   Type val[2*m];    // массив элементов
   int  down[2*m+1]; // массив номеров вершин след. уровня
   int  k;           // текущее количество элементов
   int  number;      // номер данной вершины
} BTree;
```

Пусть мы имеем возможность разместить в памяти N вершин. Создадим список (массив), в котором будем хранить номер вершины, указатель на начало ее размещения в памяти, смещение в файле для данной вершины, количество обращений к элементам данной вершины. Если нужная нам вершина уже расположена в памяти, то указанная информация обеспечивает возможность доступа к ее элементам (при этом следует корректировать поле, относящееся к количеству обращений), а также запись вершины из памяти обратно в файл. Если вершины в памяти нет, то она считывается с диска и информация о ней записывается в свободную ячейку этого списка (массива). Если список уже содержит N значений, то мы должны освободить один из его элементов, переписав одну из вершин обратно на диск. При этом можно воспользоваться одной из следующих стратегий:

1) освобождается та вершина, которая имела наименьшее количество обращений (освобождается наименее используемая вершина);

2) освобождается та вершина, которая имела наибольшее количество обращений (возможно, элементы этой вершины уже обработаны и далее не потребуются);

3) подсчитывается общее число m обращений ко всем элементам B-дерева; при достижении числом m некоторого порогового значения, количество обращений к каждой отдельной вершине обнуляется; освобождается та вершина, которая на данный момент имеет наименьшее количество обращений (т.е. последнее время реже всего использовалась).

3.61. Постройте полные реализации описанных выше деревьев в виде параметризованных классов на языке C++.

3.62. Для каждого из деревьев реализуйте итератор по его элементам. Итератором обычно называется набор функций, позволяющих последовательно перебрать все элементы из заданного множества. Например, можно предложить следующий интерфейс для итератора:

```
void StartIteration();
   начать итерирование с начала
Type * GetNextValue();
   получить указатель на очередное значение
   (NULL, если больше нет)
```

Для такого итератора пример печати всех значений из множества при помощи некоторой функции `Print` может выглядеть, например, так

```
Type * v;
for (StartIteration(); v=GetNextValue(); ) {
    Print(v);
}
```

Реализация итератора для деревьев отличается от процедур обходов. Обходы используют рекурсию и выполняются "за один прием". Функции итератора вызываются независимо друг от друга и, следовательно, реализация должна хранить где-то внутри состояние обхода, например, в виде стека посещенных вершин в текущей ветви дерева.

3.4 Графы

Граф — это пара (V, E), где V — множество вершин, а E — множество ребер (т.е. пар вершин). Каждой вершине и каждому ребру могут быть дополнительно сопоставлены некоторые наборы данных, например, некоторые ребра могут быть ориентированными (имеется связь только в одном направлении), либо содержать длину пути между соответствующими вершинами. В этом случае говорят о нагруженном графе, а его ребра называют дугами. Для описания графа из N вершин необходимо каким-то образом указать множества V и E. Вершины обычно нумеруют последовательностью целых