МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РОССИЙСКОЙ ФЕДЕРАЦИИ
Брянский государственный технический университет

Утверждаю
Ректор университета

_____О.Н. Федонин

«____»_____2017 г.

СТРУКТУРЫ И АЛГОРИТМЫ ОБРАБОТКИ ДАННЫХ

БИНАРНЫЕ ДЕРЕВЬЯ

Методические указания
к выполнению лабораторной работы №6
для студентов очной, очно-заочной и заочной форм обучения
по направлениям подготовки
09.03.01 «Информатика и вычислительная техника»,
02.03.03 «Математическое обеспечение и администрирование информационных систем»,
09.03.04 «Программная инженерия»

Брянск 2017

УДК 006.91

Структуры и алгоритмы обработки данных. Бинарные деревья[Текст] + [Электронный ресурс]: методические указания к выполнению лабораторной работы №6 для студентов очной, очно-заочной и заочной форм обучения по направлениям подготовки 09.03.01 «Информатика и вычислительная техника», 02.03.03 «Математическое обеспечение и администрирование информационных систем», 09.03.04 «Программная инженерия». – Брянск: БГТУ, 2017. –28 с.

Разработали:
к. т. н, проф. В.К. Гулаков
к. т. н, доц. А.О. Трубаков
ст. преп. С.Н. Зимин

Рекомендовано кафедрой «Информатика и программное обеспечение» БГТУ (протокол №1 от 13.09.17)

Научный редактор В.В. Конкин
Редактор издательства Л.Н. Мажугина
Компьютерный набор В.К. Гулаков

Темплан 2017 г., п.216
Подписано в печать Формат 1/16 Бумага офсетная. Офсетная печать. Усл.печ.л. 1,63 Уч.-изд.л. 1,63 Тираж 40 экз. Заказ Бесплатно

Издательство Брянского государственного технического университета
241035, Брянск, бульвар им.50-летия Октября, 7, БГТУ, тел. 58-82-49
Лаборатория оперативной полиграфии БГТУ, ул. Институтская, 16

1. ЦЕЛЬ ЛАБОРАТОРНОЙ РАБОТЫ

Целью лабораторной работы является приобретение навыков использования рекурсивных функций и алгоритмов построения и обхода бинарных деревьев.

Продолжительность лабораторной работы – 4 часа.

2. ТЕОРЕТИЧЕСКИЕ СВЕДЕНИЯ

Дерево является одной из самых распространённых структур данных. Наиболее популярное определение дерева: – это граф (подграф), у которого все узлы связаны и у которого нет циклов. Несвязный граф, состоящий исключительно из деревьев, называется *лесом*. Каждый элемент дерева называется вершиной (узлом) дерева. Вершины дерева соединены направленными дугами, которые называют ветвями дерева или рёбрами.

Имеется большое многообразие различных деревьев. Из них популярны среди программистов две группы деревьев:

- остовные деревья (используются в основном в алгоритмах теории графов);
- корневые деревья (используются при представлении иерархических данных).

Рассмотрим корневые деревья более подробно. Начальный узел дерева называют корнем дерева, ему соответствует нулевой уровень. Листьями дерева называют вершины, в которые входит одна ветвь и не выходит ни одной ветви. В каждую вершину, кроме корня, входит одна дуга, т.е. выполняется отношение одного ко многим.

Все вершины, в которые входят ветви, исходящие из одной общей вершины, называются потомками, а сама вершина – предком. Корень дерева не имеет предка, а листья дерева не имеют потомков.

Высота (глубина) дерева определяется числом уровней, на которых располагаются его вершины. Высота пустого дерева равна нулю, высота дерева из одного корня – единице.

Поддерево – часть древовидной структуры данных, которая может быть представлена в виде отдельного дерева.

Степенью исхода вершины (ранг) в дереве называется числом дуг, которое из нее выходит. Степень дерева равна максимальной

степени вершины, входящей в дерево. При этом листьями в дереве являются вершины, имеющие степень нуль. По степени исхода различают два типа деревьев:

- бинарные – степень исхода дерева не более двух;
- *n*-арные – степень исхода дерева произвольная.

Представление деревьев Деревья являются рекурсивными структурами, так как каждое поддерево также является деревом. Таким образом, дерево можно определить как рекурсивную структуру, в которой каждый элемент является:

- либо пустой структурой;
- либо элементом, с которым связано конечное число поддеревьев.

Списочное представление деревьев основано на элементах, соответствующих вершинам дерева. Каждый элемент имеет поле данных и два поля указателей: указатель на начало списка потомков вершины и указатель на следующий элемент в списке потомков текущего уровня. При таком способе представления дерева обязательно следует сохранять указатель на вершину, являющуюся корнем дерева.

Существует большое многообразие древовидных структур данных. Выделим самые распространенные из них - бинарные (двоичные) деревья,

Бинарное (двоичное) дерево – динамическая структура данных, представляющее собой дерево, в котором каждая вершина имеет не более двух потомков. Полное бинарное дерево, когда степень исхода всех узлов равна 2, и все уровни заполнены. Фактически такое дерево можно считать фрактальной структурой (Рис. 1)

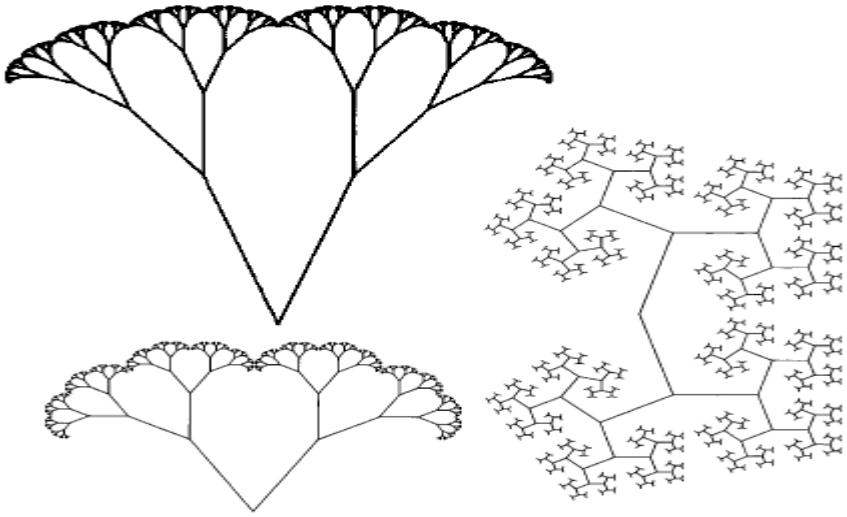


Рис. 1. Фрактальные деревья

Бинарные деревья могут применяться:

- для поиска данных в специально построенных деревьях (базы данных),
- для сортировки данных,
- для вычислений арифметических выражений,
- кодирования (метод Хаффмана)
- управление иерархией данных;
- упрощение поиска информации (см. обход дерева);
- управление сортированными списками данных;
- синтактический разбор арифметических выражений (англ. *parsing*), оптимизация программ;
- в качестве технологии компоновки цифровых картинок для получения различных визуальных эффектов;
- форма принятия многоэтапного решения и т.д.

Представление деревьев с помощью списков. Каждая вершина бинарного дерева является структурой, состоящей из четырех видов полей. Как правило, это запись. Содержимым полей этой записи будут соответственно:

- информационное поле (ключ вершины);
- служебное поле (их может быть несколько или ни одного);
- указатель на левое поддерево;
- указатель на правое поддерево.

В общем случае у бинарного дерева на k-м уровне может быть до 2^{k-1} вершин. Бинарное дерево называется полным, если оно содержит только полностью заполненные уровни. В противном случае оно является неполным.

Бинарное дерево может представлять собой пустое множество, может выродиться в список (рис.2).

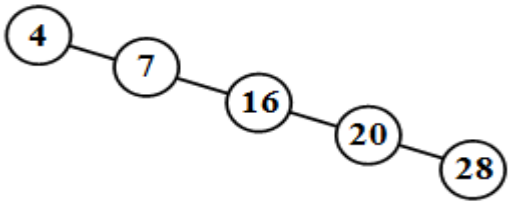
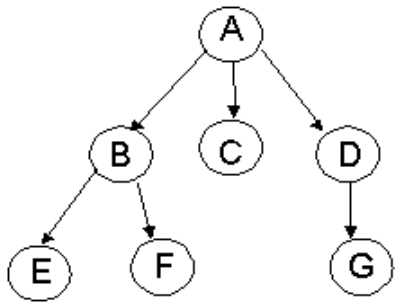


Рис. 2. Список как частный случай бинарного дерева

Представление деревьев с помощью таблицы. Представление дерева с помощью таблицы может быть рекомендовано, например, в

случае постоянства или редкой изменчивости структуры дерева. В частном случае, когда все элементы одного типа, таблица превращается в массив (Рис. 3).



A	3	2	3	4
B	2	5	6	0
C	0	0	0	0
D	1	7	0	0
E	0	0	0	0
F	0	0	0	0
G	0	0	0	0

Рис. 3. Представление деревьев с помощью таблицы.

Представление деревьев с использованием списков сыновей. Этот способ представления деревьев состоит в формировании для каждого узла списка его сыновей. Эти списки можно представить любым методом, но так как число сыновей у разных узлов может быть разное, чаще всего для этих целей применяются связанные списки.

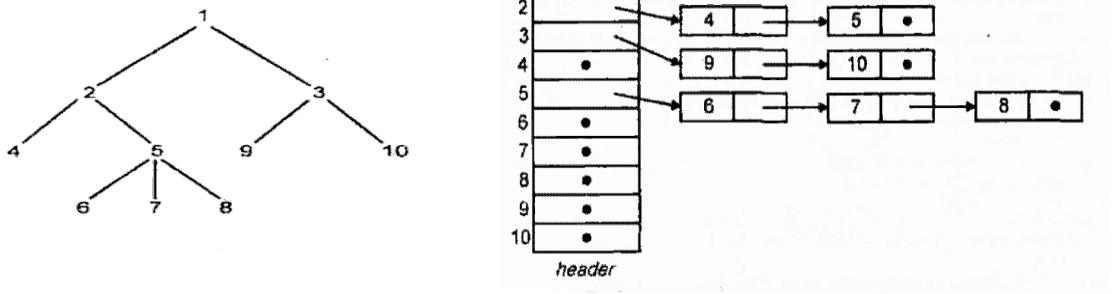


Рис 4. Представление деревьев с использованием списков сыновей.

Инверсная форма представления деревьев. Дерево представлено в инверсной форме, если отношения между узлами обратные обычным отношениям в деревьях, т. е. каждый узел ссылается на своего предшественника.

Инверсная форма представления деревьев не нашла широкого применения, поскольку при таком способе представления дерева только одна операция, операция доступа к узлу - предшественнику, реализуется просто, хотя имеется некоторая экономия памяти.

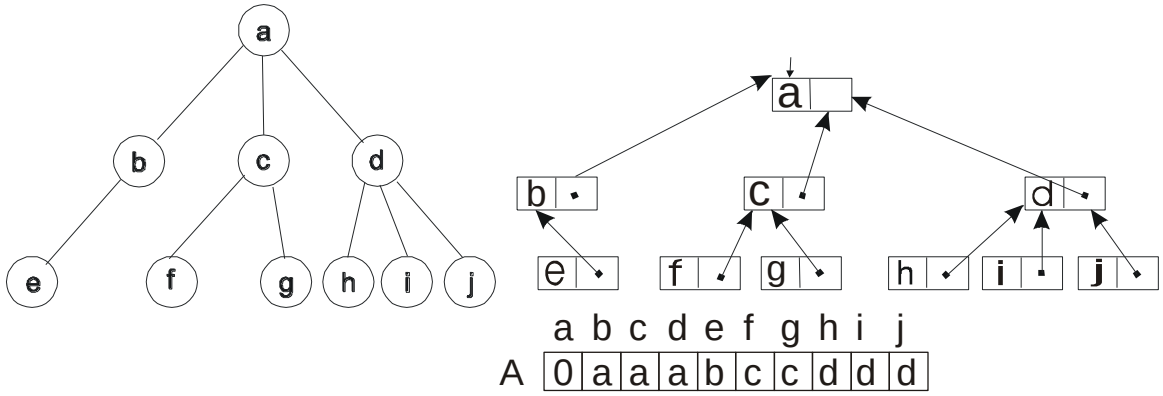


Рис 5. Инверсная форма представления деревьев.

Описание бинарного дерева выглядит следующим образом:

```
struct имя_типа {
                 информационное поле;
                 [служебное поле;]
                 адрес левого поддерева;
                 адрес правого поддерева;
                };
```

где информационное поле – поле любого объявленного или стандартного типа;

адрес левого (правого) поддерева – указатель на объект того же типа, что и определяемая структура, в него записывается адрес следующего элемента левого (правого) поддерева.

Пример

```
struct point {
              int data;//информационное поле
              int count; //служебное поле
              point *left;//адрес левого поддерева
              point *right;//адрес правого поддерева
             };
```

Основными операциями, осуществляемыми над бинарными деревьями, являются:

- создание бинарного дерева;
- печать бинарного дерева;
- обход бинарного дерева;
- вставка элемента в бинарное дерево;
- удаление элемента из бинарного дерева;
- проверка пустоты бинарного дерева;
- удаление бинарного дерева.

Дополнительные операции

- вставка нового элемента в определённую позицию;
- вставка поддерева;
- добавление ветви дерева (называется *прививкой*);
- нахождение корневого элемента для любого узла;
- нахождение наименьшего общего предка двух вершин;
- перебор элементов ветви дерева;
- поиск изоморфного поддерева;
- удаление ветви дерева (называется *обрезкой*);
- удаление поддерева;

Чтобы выполнить определенную операцию над всеми вершинами дерева необходимо все его вершины просмотреть. Такая задача называется обходом дерева.

Обход дерева – упорядоченная последовательность вершин дерева, в которой каждая вершина встречается только один раз.

При обходе все вершины дерева должны посещаться в определенном порядке. Существует несколько способов обхода всех вершин дерева. Выделим три наиболее часто используемых способа обхода дерева (рис. 6):

- прямой;
- симметричный;
- обратный.

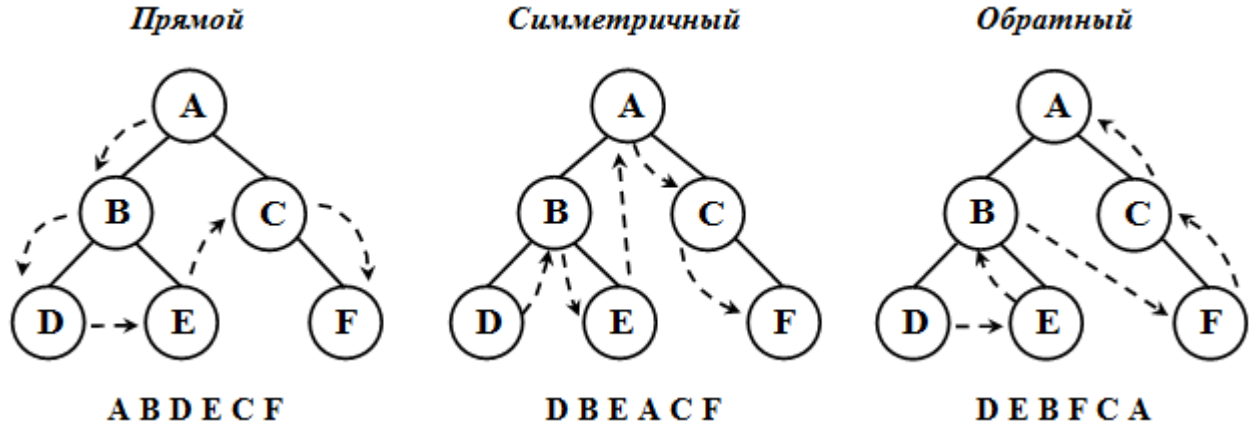


Рис. 6. Обходы деревьев

Если дерево неупорядоченно, то просмотр всех его элементов можно осуществить с помощью различных видов обхода. Обход применяется и для упорядоченных деревьев, например при сортировке.

Если узлы (вершины) упорядоченного дерева содержат ключи, то дерево называется деревом поиска.

Бинарные деревья поиска

Дерево поиска – древовидная структура данных, которая хранит объекты, каждый из которых идентифицируется ключо. Дерево имеет корень, с которого начинается поиск, а также содержит в каждом узле некоторое ключевое значение для сравнения с запрашиваемым ключом, вследствие чего мы можем проходить разные пути по дереву в зависимости от того, какой ключ больше - запрашиваемый или ключ в узле. Мы спускаемся от корня вниз по дереву до тех пор, пока не найдём узел с искомым ключом или не убедимся, что искомого ключа нет в данном дереве.

Дерево поиска является основной для многих структур данных, поскольку допускает много различных модификаций. Эти модификации можно разделить на два класса, основанных на несколько различных подходах к организации хранения записей и к установлению ограничений на ключевые значения в узлах дерева. Эти подходы называются моделями дерева.

Две модели деревьев поиска: листовые и узловые деревья. Пусть мы запрашиваем объект с ключом Q и в данный момент находимся в узле N с ключом K. Если $Q<K$, то запись с ключом Q расположена в левом поддереве узла N либо отсутствует, если у узла N нет левого сына. Аналогично если $Q>K$, то запись с ключом Q может быть только в правом поддереве, либо отсутствует.

Неоднозначность вызывает ситуация, когда искомый ключ совпадает с ключом в текущем внутреннем узле. В связи с этим и появились две модели деревьев поиска [19].

1. Листовые деревья – если $Q<K$, то переходим к левому поддереву узла N, в противном случае (в том числе при равенстве ключей) переходим к правому поддереву, и так пока не достигнет листа. Ключи во внутренних узлах служат только для сравнения, объекты могут быть только в листьях. Иначе говоря, листовые узлы полезные, а внутренние узлы вспомогательные.

2. Узловые деревья – если $Q<K$, то переходим к левому поддереву узла N, если $Q>K$, то переходим к правому поддереву. Если же $Q=K$, то запрашиваемый объект в узле N.

На рис. 6 приведён пример листового (слева) и узлового (справа) деревьев, и соответствующие схемы сравнения, где *Q* – запрашиваемый ключ, *K* – ключ в текущем внутреннем узле.

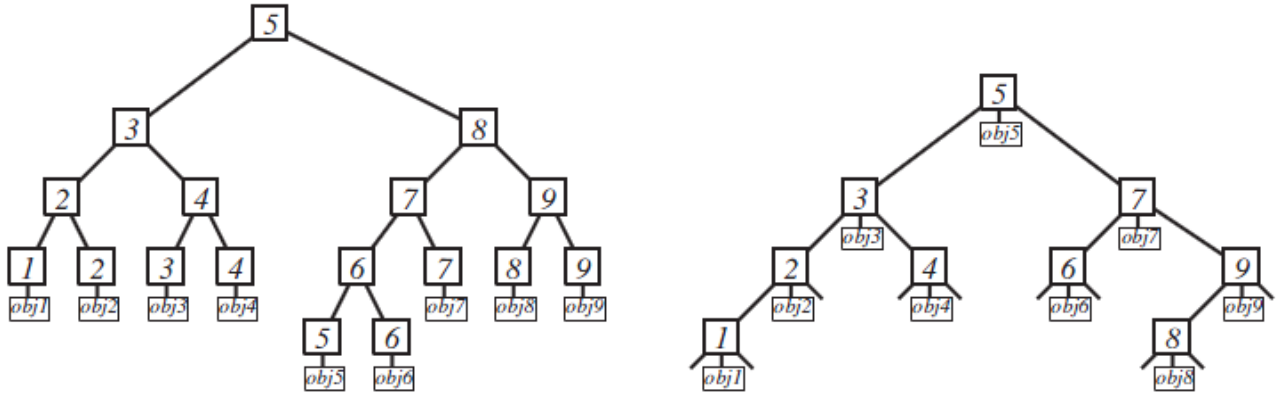


Рис. 7. Модели деревьев поиска: листовое и узловое дерево

Анализ моделей деревьев поиска. Разделение деревьев поиска на два класса даёт ряд важных следствий.

Так, листовые деревья полностью соответствуют понятию бинарных деревьев, в то время как узловые деревья в некотором смысле тернарные - у узла наряду с двумя очевидными потомками - левым и правым - есть особый - средний потомок - это объект.

Внутренние узлы листового дерева всегда имеют два поддерева, в то время как внутренний узел второй модели может не иметь левого либо правого поддерева.

В листовом дереве спуск из внутреннего узла к потомку требует одного сравнения, в то время как в узловом дереве может требоваться два сравнения: 1) верно ли, что Q<K; 2) верно ли, что Q>K.

На рис. 8 приведён пример спуска по листовому и узловому дереву. В данном случае ищется ключ 8. Легко заметить, что для поиска в узловом дереве потребовалось почти вдвое больше сравнений, чем для поиска в листовом дереве.

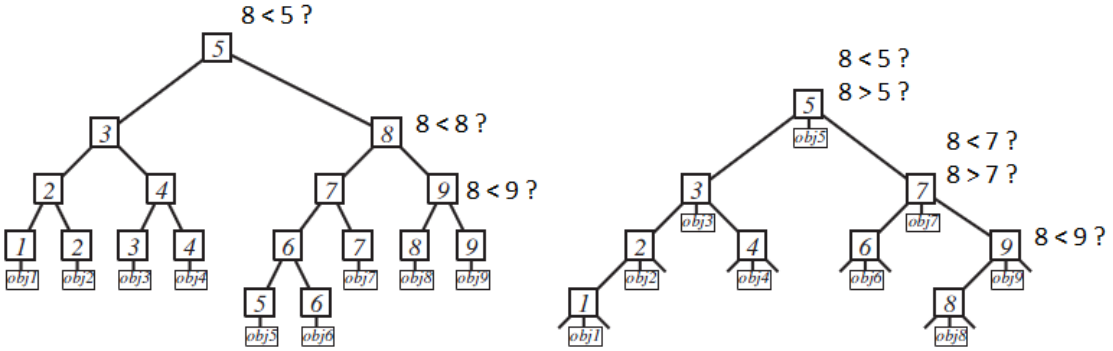


Рис. 8. Пример поиска по листовому и узловому дереву

Чтобы достичь объект, находящийся в листе, для узлового дерева в худшем случае требуется примерно вдвое больше операций сравнения, чем для листового дерева. Худший с точки зрения сравнений случай для узлового дерева - когда всегда приходится идти от узла к правому сыну.

В узловом дереве часть объектов находится на уровнях, близких к корню, и их можно достичь малым числом сравнений, но таких узлов относительно мало.

Пусть узловое дерево имеет высоту h, а все его листья находятся на одном и том же полностью заполненном уровне. Тогда средняя длина пути от корня до узла с искомым ключом составляет

$$l_h = \frac{h \cdot 2^h + (h-1) \cdot 2^{h-1} + ... + 1 \cdot 2}{2^{h+1} - 1}$$

Листовое дерево в этом случае имеет оптимальную высоту h+1, поскольку требуется хранить как 2^{h+1} – 1 объектов. Если такое листовое дерево очень хорошо сбалансированно, то почти все его листья будут на уровне h+1. Соответственно мы получаем доступ к ключу в среднем приблизительно за h+1 сравнений.

Если h велико, то можно считать, что

$$\frac{2^{h+1}}{2^{h+1} - 1} \approx 1$$

Из этого следует, что

$$l_h \approx \frac{h \cdot 2^h + (h-1) \cdot 2^{h-1} + ... + 1 \cdot 2}{2^{h+1}} = \frac{h}{2} + \frac{(h-1)}{4} + \frac{(h-2)}{8} + ... + \frac{1}{2^h}$$

Возьмём первые 2 слагаемых полученного выражения и воспользуемся тем, что все слагаемые положительные:

$$l_h > \frac{h}{2} + \frac{(h-1)}{4} = \frac{3h}{4}$$

Вероятность того, что Q<K, и вероятность обратного события примерно равны, поэтому математическое ожидание количества сравнений в одном внутреннем узле составляет примерно 1,5. Умножим полученную нижнюю оценку l_h на 1,5 и получим, что среднее количество сравнений для узлового дерева больше, чем количество сравнений для листового, несмотря на то, что некоторые объекты узлового дерева близки к корню.

Также заметим, что листовое дерево высоты h может содержать до 2^h объектов, а узловое дерево высоты h – до 2^{h+1}-1 объектов. Дело в том, что на уровне m может быть до 2^m узлов, соответственно в дереве

высоты h может быть до $1 + 2 + 4 + ... + 2^h$ узлов, откуда по формуле суммы членов геометрической прогрессии и получаем приведённый результат для узлового дерева.

В листовом дереве ключи внутренних узлов нужны только для сравнений, и один и тот же ключ может быть в дереве до 2 раз - один раз во внутреннем узле, другой раз в листе. В узловых деревьях все ключи уникальны. Для листового дерева возможна ситуация, когда ключ внутреннего узла не соответствует никакому объекту, например, это бывает, если объект был удалён. В моделях листовых деревьев вообще возможно задание ключей внутренних узлов без оглядки на соответствие ключам записей, а лишь с учётом получения хорошего разбиения пространства поиска. Под таким разбиением имеется в виду выбор ключа в узле так, чтобы вероятности добавления новых узлов в каждое из двух поддеревьев были примерно одинаковы.

Структура листового дерева поиска. В листинге 1 приведена структура узла листового дерева поиска. Как видим, полями структуры являются ключ, а также указатели на потомков. Часто в качестве дополнительных данных нужен, например, показатель сбалансированности. Могут быть и другие варианты дополнительных данных, например, какие-либо счётчики для экспериментальной оценки эффективности – чтобы собирать статистические данные.

Листинг 1. Структура узла листового дерева

```
typedef struct tr_n_t
{
      key_t              key; // ключ
      struct tr_n_t    *left; // указатель на левого потоика
      struct  tr_n_t      *right;  //  указатель  на  правого
потомка

      /* возможны дополнительные поля */
} tree_node_t;
```

На основе узлов данного типа можно построить дерево по следующему определению: каждое дерево или пустое, или является листом, или содержат особый - корневой - узел, который указывает на два непустых дерева. Пусть K - ключ корня. Тогда все ключи левого поддерева меньше K, а все ключи правого поддерева - больше или равны K.

Важно определить, как отличить листья от прочих узлов. Можно сделать следующее допущение: если узел *n - листовой, то n->right =

NULL, n->left указывает на соответствующий листу объект, а n->key хранит объект ключа.

Пусть дан узел *root - корень дерева. Если указатель left равен NULL, то дерево пустое. Если left отличен от NULL, но right равен NULL, то дерево содержит единственный объект, поскольку *root - листовой узел. Если оба указателя ненулевые, то они указывают на поддеревья дерева с корнем *root.

Теперь создать пустое дерево можно так, как показано в листинге 1.2.

Листинг 2. Создание пустого листового дерева

```
tree_node_t *create_tree(void)
{
      tree_node_t *tmp_node; // объявляем узел
      tmp_node = get_node(); // вызываем конструктор узла
      tmp_node->left = NULL; // признак пустого дерева

      return (tmp_node);
}
```

Основные возможности и преобразования. В корректно построенном дереве поиска мы можем ассоциировать каждый узел дерева с интервалом возможных ключевых значений, которые могут быть достигнуты через этот узел. Интервал, соответствующий корню, - это $(-\infty,\infty)$. Если *n является внутренним узлом, которому соответствует интервал [a, b), то $n->key$ находится в этом интервале, а потомкам соответствуют интервалы $[a,n->key)$ и $[n->key,b)$. За исключением интервала, начинающегося с $-\infty$, все интервалы полуоткрытые: они содержат левый конец и не содержат правый конец. На рис. 1.4 приведён пример такого соответствия.

Одно и то же множество пар (ключ, объект) может индексироваться с помощью многих разных корректно построенных листовых деревьев. Листья всегда одни и те же, содержащие пары (ключ, объект) в порядке возрастания ключей слева направо, но дерево, дающее доступ к листьям, может быть построено по-разному, и при этом некоторые варианты лучше остальных. Существуют две операции - левых и правых поворотов, которые преобразуют корректное дерево поиска в другое корректное дерево поиска для данного множества записей. Они используются в качестве «строительных блоков» для более сложных операций преобразований дерева, поскольку они легко реализуются и носят универсальный характер.

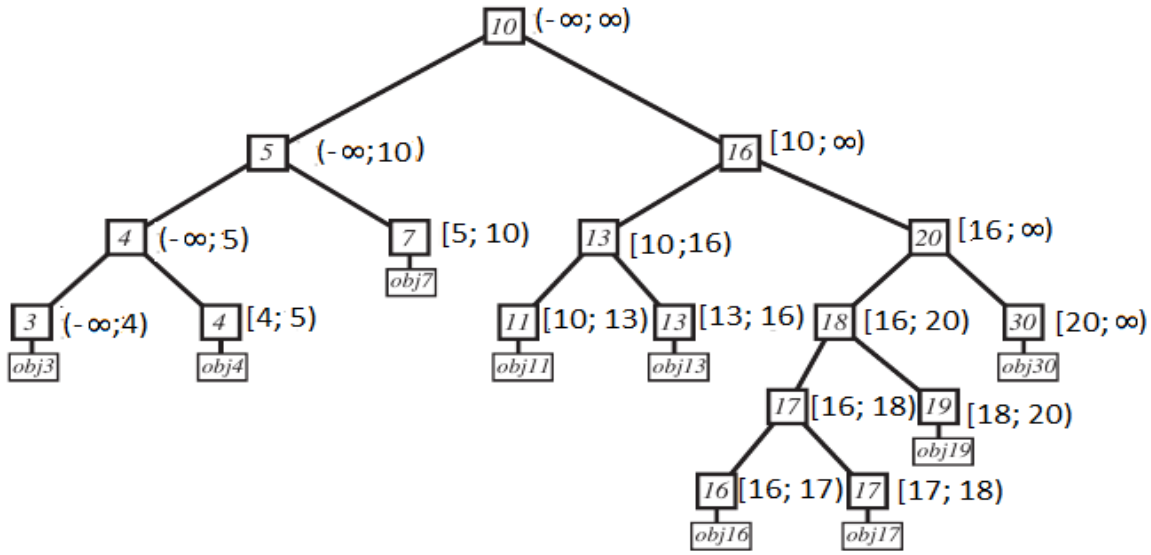


Рис. 9. Соответствие узлов листового дерева и интервалов возможных ключевых значений

Пусть *n - внутренний узел дерева и n->right также внутренний узел. Тогда трём узлам n->left, n->right->left и n->right->right соответствуют интервалы, объединение которых даст интервал *n. Теперь вместо объединения второго и третьего интервалов в одном узле n->right и затем объединения результата с интервалом n->left в узле *n мы сгруппируем первые два интервала, а результат сгруппируем с третьим интервалом. Это левый поворот, а *n является центром поворота. Это преобразование носит локальный характер и выполняется за постоянное время.

В листинге 3 приведён фрагмент исходного кода, который выполняет левый поворот вокруг некоторого узла*n, в листинге 4 – правый поворот.

Заметим, что хотя мы меняем содержимое узлов, узел *n должен по-прежнему быть корнем поддерева, поскольку узлы верхних уровней дерева прямо или косвенно ссылаются на него. Правый поворот является в точности обратной операцией относительно левого поворота (см. рис. 9).

Листинг 3. Левый поворот

```
void left_rotation(tree_node_t *n)
{
      tree_node_t *tmp_node;
      key_t tmp_key;
      tmp_node = n->left;
      tmp_key = n->key;
```

```
        n->left = n->right;
        n->key = n->right->key;
        n->right = n->left->right;
        n->left->right = n->left->left;
        n->left->left = tmp_node;
        n->left->key = tmp_key;
}
```

Листинг 4. Правый поворот

```
void right_rotation(tree_node_t *n)
{
        tree_node_t *tmp_node;
        key_t tmp_key;
        tmp_node = n->right;
        tmp_key = n->key;
        n->right = n->left;
        n->key = n->left->key;
        n->left = n->right->left;
        n->right->left = n->right->right;
        n->right->right = tmp_node;
        n->right->key = tmp_key;
}
```

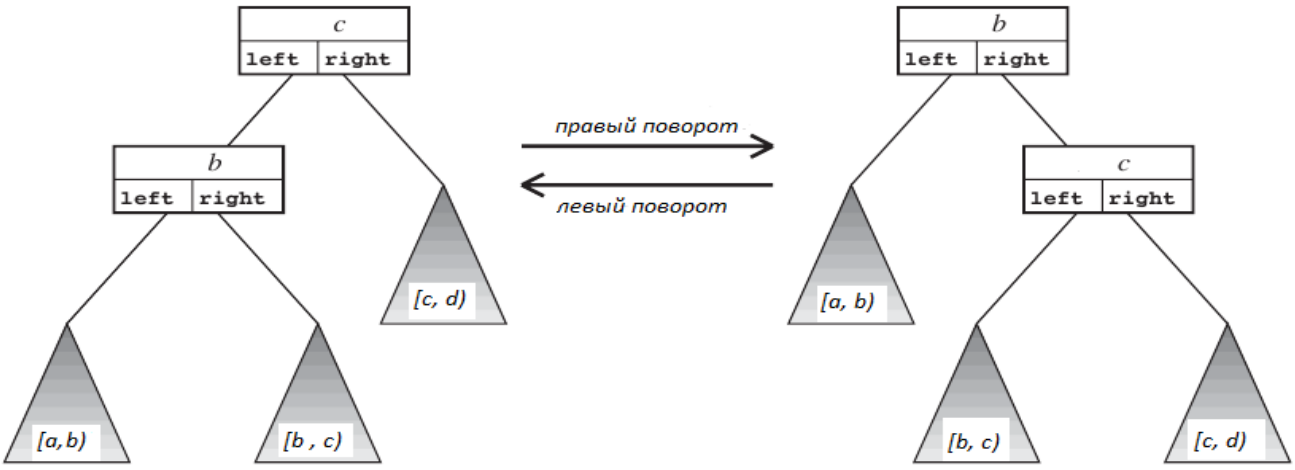


Рис. 10. Левый и правый повороты

Левый и правый повороты вокруг одного и того же узла являются взаимно обратными операциями. Левый и правый повороты преобразуют корректное дерево поиска в другое корректно дерево поиска для заданного множества пар (ключ, объект).

Огромное значение поворотов как составляющих более сложных операций над деревьями обусловлено фактом универсальности поворотов: любое корректное дерево поиска может быть

преобразовано в любое другое корректное дерево поиска для заданного множества путём применения некоторой последовательности поворотов. Но следует помнить одну важную особенность: в листовых деревьях мы можем менять значения ключей внутренних узлов без нарушения правила дерева поиска только пока не меняется отношение порядка ключей внутренних узлов и ключей объектов.

Заметим, что применяя правые повороты к дереву поиска столько, сколько это будет возможно, мы получим вырожденное дерево, в котором есть только путь, всегда идущий вправо, листья вдоль которого расположены в порядке возрастания ключей. Любое дерево поиска может быть преобразовано к этому «каноническому» дереву путём правых поворотов. Поскольку левый и правый повороты взаимно обратны, это «каноническое» дерево может быть преобразовано в любое другое путём последовательности левых поворотов.

Высота дерева поиска. Важнейшей характеристикой деревьев поиска является высота, то есть длина самого длинного пути от корня к листу, или самый нижний уровень дерева, если верхний уровень - уровень корня - принять за нулевой. Высота дерева во многом определяет сложность операций поиска в дереве и редактирования дерева. Дело в том, что для одного и того же множества записей различные правила построения дерева могут дать совершенно разные высоты.

Определим, какова минимально возможная высота дерева поиска с заданным количеством узлов. Корень дерева - 1 узел, на уровне 1 может быть максимум 2 узла, на уровне h может быть до 2^h узлов. По формуле суммы членов геометрической прогрессии получаем, что в дереве высоты h может быть до$(2^{h+1}-1)$ узлов. Это означает, что для n узлов минимально возможная высота составляет $[\log_2 n]$ узлов. Например, при n=7 это 2, а при n=8 это 3 (см. рис. 1.2). Итак, у оптимального бинарного дерева поиска высота оценивается как O(log n). При этом если дерево имеет высоту h, то листья располагаются на уровнях h и h-1 (если $n=2^{h+1}-1$, то все на высоте h).

В худшем случае дерево вырождается в линейный список (см. рис. 2). Например, пусть вставляется упорядоченная или почти упорядоченная по возрастанию последовательность записей. Первую запись вставляем в корень. Вторая соответствует правому сыну R

корня, третья - правому сыну *R* и т.д. В результате если у нас *n* записей, то число уровней оценивается как O(*n*). Соответственно сложность точного поиска в дереве становится линейной.

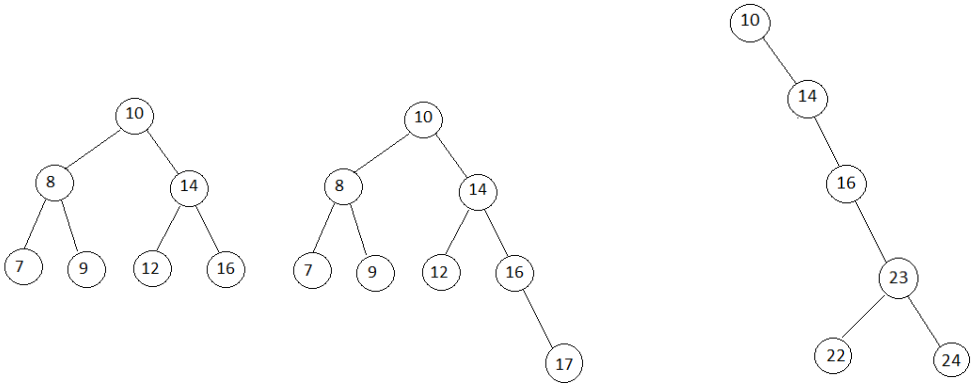


Рис. 11. Деревья поиска с наименьшими возможными высотами для 7 и 8 узлов и почти вырожденное в линейный список

Для хорошо сбалансированного дерева (то есть дерева, высота которого не сильно отличается от оптимальной), точный поиск характеризуется сложностью O(log *n*), равно как вставки и удаления записей. Для большинства реальных приложений это вполне приемлемо.

Основные операции над бинарным деревом поиска следующее:
1) вставка - добавляет в дерево заданную запись с заданным ключом;
2) удаление - уничтожает запись с заданным ключом;
3) поиск - возвращает запись с заданным ключом, если таковая есть.

Добавление записей должно выполняться так, чтобы сохранялось свойство дерева поиска.

Простейший случай - когда дерево ещё пустое. Тогда просто создаём корень и добавляем туда запись.

Теперь рассмотрим, как быть, если дерево не пустое. Ведём спуск по дереву. Пусть в данный момент мы находимся в некотором узле *N*, ему соответствует ключ *C*, а мы ищем, куда должен быть добавлен ключ *K*.

Если *C* > *K*, то по определению дерева поиска ключ должен быть в левом поддереве узла *N*. Если узел *N* не имеет левого потомка, то новый узел с ключом *K* становится левым потомком для *N*. Если же у *N* уже есть левый потомок – узел *L*, то спускаемся к узлу *L*.

Если *C* < *K*, то по определению дерева поиска ключ должен быть в правом узла *N*. Если узел *N* не имеет правого потомка, то новый узел с ключом *K* становится правым потомком для *N*. Если же у *N* уже есть правый потомок - узел *R*, то спускаемся к узлу *R*.

Если $C = K$, считаем операцию добавления неуспешной - запись, которую мы пытались добавить, уже есть в дереве. В реальных приложениях в этом случае возможен и другой вариант - меняем значения неключевых атрибутов в узле N на новые.

Приведём пример добавления записи (см. рис. 11).

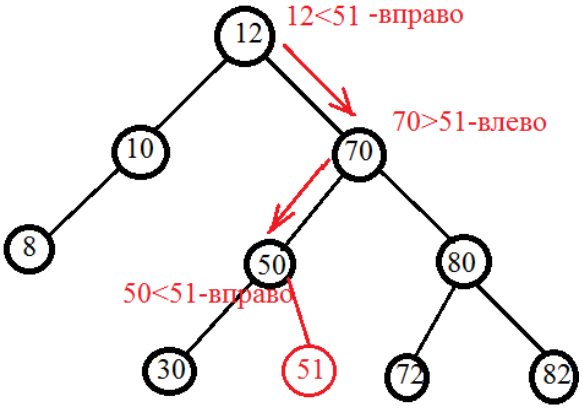


Рис. 12. Пример добавления записи

Сначала ключ 51 сравнивается с ключом корня. Поскольку ключ корня оказался меньше, переходим к правому сыну R. Его ключ 70 оказался больше добавляемого, поэтому переходим к левому потомку L узла R. Узел этого потомка имеет ключ 50 - меньше добавляемого. Правый сын у узла L отсутствует, поэтому новый узел с ключом 51 и становится правым сыном узла L. Заметим, что правило дерева поиска не нарушено.

Рассмотрим поиск в бинарном дереве.

Если дерево пустое, сразу же считаем поиск неуспешным. В противном случае ведём спуск по дереву, очень похожий на то, что мы делали при добавлении записи (при добавлении неявно вёлся поиск добавляемой записи).

Пусть в данный момент мы находимся в некотором узле N, ему соответствует ключ C, а мы ищем, куда должен быть добавлен ключ K. Если $C = K$, искомая запись найдена. Если $C > K$, то по определению дерева поиска ключ должен быть в левом поддереве узла N. Если узел N не имеет левого потомка, значит, искомой записи нет в дереве. Если у N есть левый потомок, спускаемся к этому потомку. Если $C < K$, то по определению дерева поиска ключ должен быть в правом поддереве узла N. Если узел N не имеет правого потомка, значит, искомой записи нет в дереве. Если у N есть правый потомок, спускаемся к этому потомку.

Примеры поиска приведены на рис. 13.

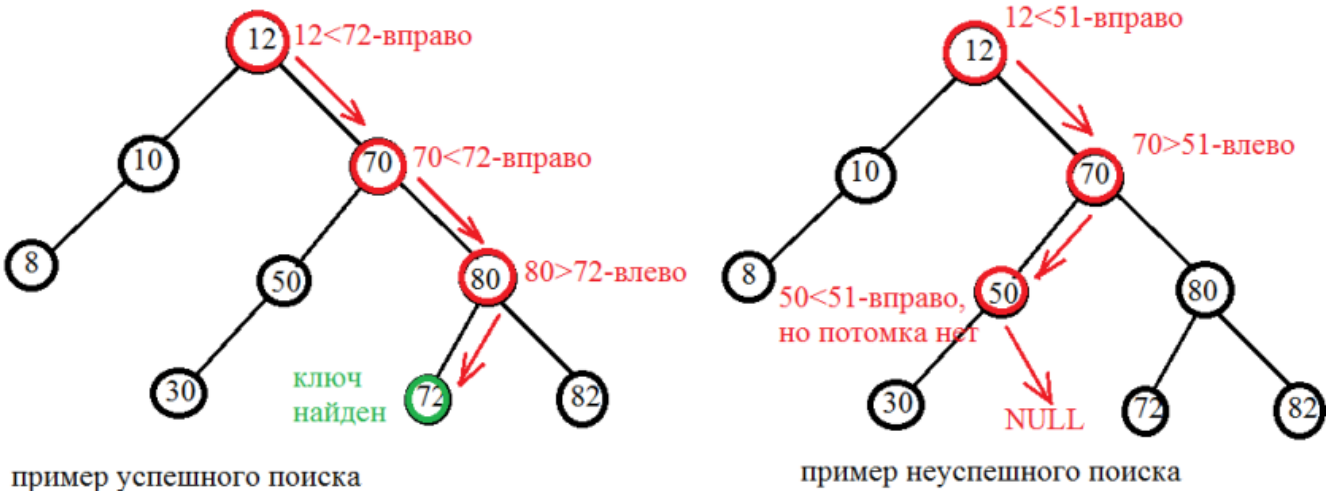


Рис. 13. Примеры поиска

Если бинарное дерево поиска с n хорошо сбалансировано, то максимально возможное количество проверяемых при поиске узлов будет близко к $\log_2 n$. Поэтому поиск в дереве часто называется логарифмическим. Заметим, что такая трудоёмкость поиска намного лучше, чем при последовательном поиске. Так, если есть 1 млн. записей, то при последовательном поиске получим в худшем случае 1 млн. проверок, в среднем - примерно 500 тыс. проверок. В этом же случае в хорошо сбалансированном бинарном дереве даже в худших ситуациях (запись находится в некотором листе) будет выполнено около 20 проверок. Ради такого ускорения и было задумано дерево поиска.

Кстати говоря, логарифмический поиск может быть выполнен не только с помощью деревьев. Бинарный поиск в отсортированном массиве также является логарифмическим. Почему же тогда используются деревья поиска? Дело в том, что для массива трудно выполнять операции редактирования. При вставке и удалении в отсортированный массив приходится выполнять сдвиг в среднем половины элементов, то есть сложность вставки и удаления составляет $O(n)$, где n - количество элементов. Кроме того, проблемой является ограниченность размера массива, в то время как дерево можно реализовать на базе списковых структур [19], тем самым оперируя с динамической структурой данных[1].

[1] Дерево может быть реализовано и на основе массива: 1) элемент с индексом 0 – корень; 2) если k – ый элемент – корень, то элементы номер 2k+1 и 2k+2 – сыновья. Такая реализация хороша только для деревьев, где уровни почти заполнены, а добавление новых записей не ожидается

Наконец, рассмотрим, как удалить запись (и соответствующий ей узел) из бинарного дерева, при этом не нарушив правила дерева поиска.

Следует различать три случая:

1) узла с заданным ключом в дереве нет;

2) узел с заданным ключом имеет не более одного сына;

3) узел с заданным ключом имеет двух сыновей.

Трудность заключается в удалении элементов с двумя потомками, поскольку мы не можем указать одной ссылкой на два направления. В этом случае удаляемый элемент нужно заменить либо на самый правый элемент его левого поддерева, либо на самый левый элемент его правого поддерева. Ясно, что такие элементы не могут иметь более одного потомка. Кроме того, выбор таких заменяющих элементов удовлетворяет правилу дерева поиска. Так, самый правый элемент левого поддерева узла N- максимальный в левом поддереве узла N. Это значит, что он больше любого другого элемента левого поддерева узла N, но по определению дерева поиска он меньше любого элемента правого поддерева узла N. Таким образом, если данный элемент заменит узел N, он окажется больше любого элемента своего левого поддерева и меньше любого элемента своего правого поддерева (см. рис. 13). Аналогично можно рассуждать для самого левого элемента правого поддерева удаляемого узла, исходя из того, что это минимальный элемент в поддереве.

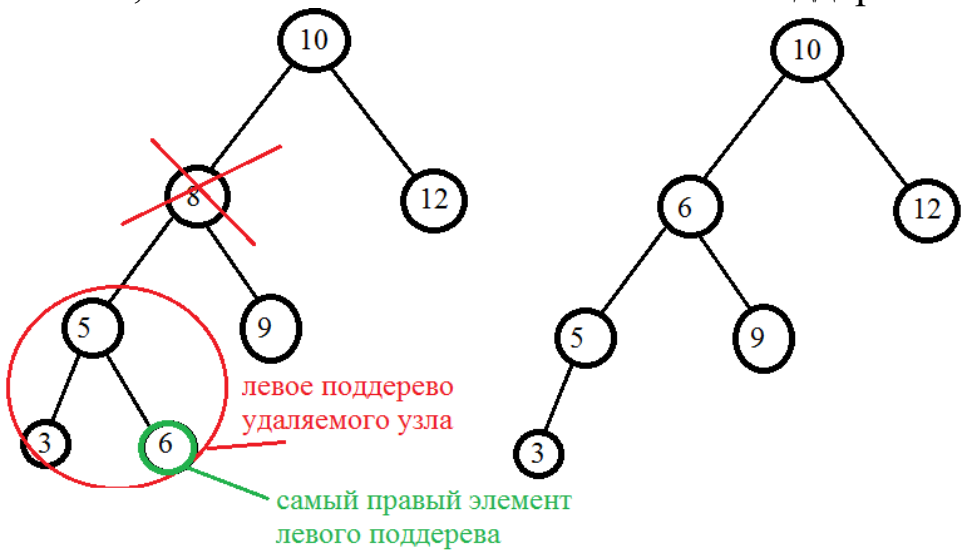


Рис. 14. Пример удаления

Возврат к корню. В большинстве случае движение по дереву при выполнении операций над ним выполняется вниз по дереву. В некоторых случаях требуется уметь выполнять обратное движение -

вверх. Например, существует такая операция, как *прошивание дерева* – установление дополнительных связей между узлами дерева, ускоряющее выполнение обхода по определённым правилам. Например, на базе коллекции записей сформировано дерево, узлы которого, как известно, придётся обходить неоднократно в порядке возрастания ключей. Тогда в каждый узел добавляем ссылку на узел, ключ которого следующий по возрастанию. При создании прошивки требуется для каждого узла найти следующий в порядке обхода, и, как можно заметить, иногда при этом придётся проходить путь с подъёмом, как например, на рис. 15, когда мы ищем последователя узла с ключом 18.

Для организации подъёма по дереву существует несколько различных способов. [7]

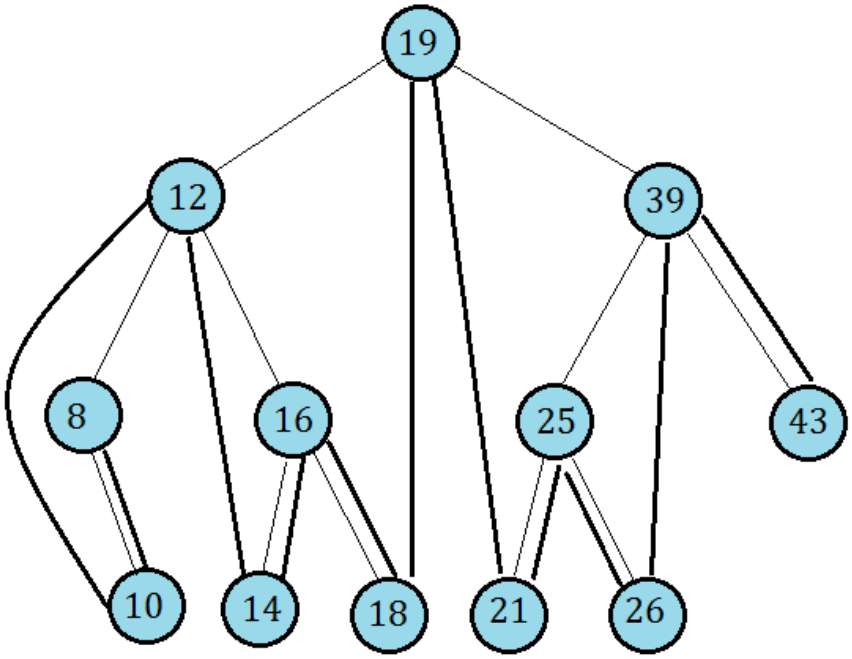


Рис. 15. Прошитое дерево

Стек. Если при спуске вниз по дереву мы используем стек, в который добавляем каждый посещённый узел, то при извлечении элементов из стека получим последовательность узлов, который нужно посетить для возврата. Такой метод не требует добавления какой-либо дополнительной информации в структуру узлов дерева - каждый узел хранит ключ записи, неключевые атрибуты и указатели на сыновей. Объём памяти, требуемой для стека, зависит от высоты дерева, то есть для хорошо сбалансированного дерева логарифмически зависит от числа записей. Такой стек вполне можно реализовать на массиве. В реальных приложениях количество

записей едва ли может достичь 2^{100}, а для хорошо сбалансированного дерева высота будет близка к 100, поэтому массива из 200 ячеек достаточно.

Дополнительный указатель. В узле дерева храним указатели не только на сыновей, но и на родителя. Это приводит к росту затрат памяти на хранение дерева, но рост несущественный - затраты памяти под отдельный узел вырастут менее, чем на 50%. Более существенная проблема - необходимость учитывать этот указатель при выполнении операций. Когда корректируется структура дерева, для некоторых узлов придётся скорректировать указатель на родителя. Также возможно введение указателя на родителя с *ленивым обновлением*: значения указателей на родителя устанавливаются в узлах только при выполнении спуска по дереву, чтобы можно было выполнить обратное движение, но при вставках и удалениях мы не трогаем эти значения. Заметим, что тогда значения дополнительного указателя может считаться верным только для тех узлов, которые были затронуты при спуске.

Обращение пути. Данный подход вообще не требует дополнительной памяти, в то время как предыдущие требовали её либо для хранения узлов усложнённой структуры, либо при выполнении спуска. При спуске по дереву меняем направления указателей на обратные (теперь указатели, по которым мы прошли путь вниз, все указывают вверх, на родительские узлы, а не на сыновей). При возврате снова обращаем направления указателей, и теперь они снова ссылаются на сыновей. Проблема такого метода в том, что его сложнее применять в сочетании с методами балансировки, нежели предыдущие.

Использование одинаковых ключей. Хотя желательно, чтобы ключ позволял однозначно идентифицировать запись, в некоторых случаях трудно подобрать атрибут, который выступил бы в роли ключа. Можно ввести искусственный атрибут, но он оказывается довольно неестественным и неудобным для пользователей (например, инвентарные номера, идентификаторы аккаунтов в соцсетях, номера документов). Соответственно в реальных приложениях иногда приходится допускать наличие одинаковых ключей. В этом случае точный поиск по ключу должен перечислить все объекты с заданным значением ключа. Тогда бинарное дерево поиска высоты h должно выполнять точный поиск за время $O(h + k)$, где k-количество

исходных записей, вставка и удаление по-прежнему должны выполняться за время O(h).

С точки зрения реализации отличие от простого дерева поиска в том, что каждому узлу соответствует связный список записей с ключом, хранящимся в данном узле [7] (см. рис. 15). Тогда при поиске по ключу мы ищем узел с этим ключом и далее бежим по списку, перечисляя хранящиеся там записи. Если вставка записи с неуникальным ключом выполняется в голову списка, то сложность вставки по-прежнему логарифмическая. Если же мы хотим поддерживать списки отсортированными, то сложность в среднем составит O($\log n + m$), где m - средняя длина списка.

иси: (10, 20), (10, 30), (15, 20), (15, 35), (16, 41), (18, 50), (19, 5), (19, 10), (19, 15)

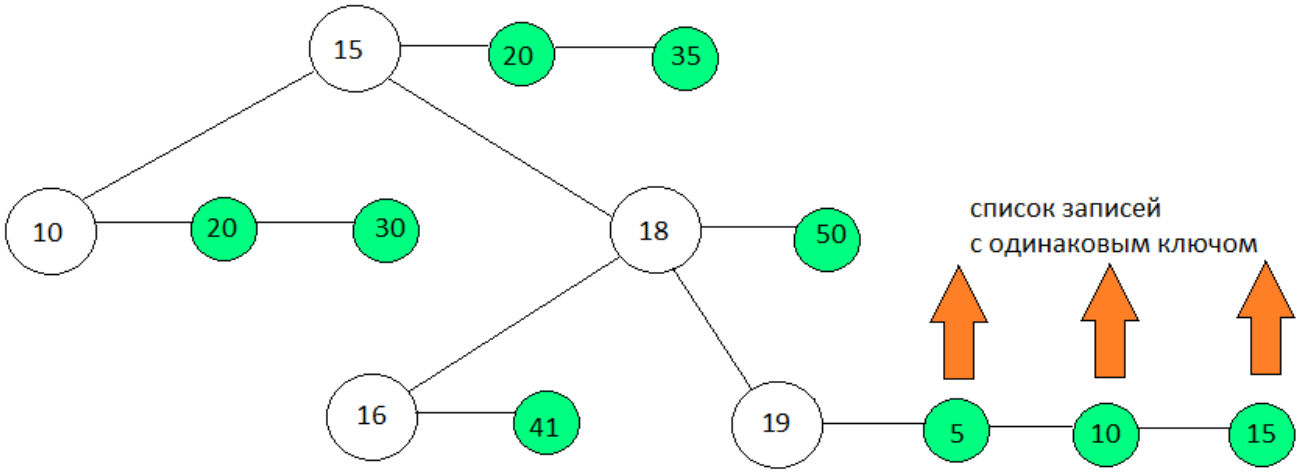


Рис. 16. Индексирование записей с одинаковыми ключами

Альтернатива – использование *многомерных деревьев*. Если не удаётся подобрать атрибут, значение которого будет для каждой записи своё, можно подобрать группу атрибутов, набор значений которых уникален для каждой записи, что в терминологии баз данных известно как *составной ключ*. Тогда индексирование выполняется на основе нескольких атрибутов, данные рассматриваются как многомерные. В этом случае можно воспользоваться, например, *K-D деревом* - обобщением идеи бинарного дерева поиска на K-мерный случай [8].

Запросы для ключей в заданном интервале. Простейшим видом поиска является *точный поиск* (поиск по точному совпадению) - вернуть запись, ключ которой равен Q (или все такие записи, если ключ неуникален). Часто требуется искать записи с ключами в некотором интервале [a, b]. Например, мы точно не знаем значение

искомого ключа, потому допускаем некоторую «погрешность». Другой вариант - когда критерий отбора естественным образом формулируется в виде «нужны объекты с характеристикой X не ниже Y и не выше Z». Например, «требуется компьютер стоимостью от 17 до 21 тысячи рублей».

Диапазонный поиск для бинарного дерева поиска может быть организован следующим образом. Пусть x - ключ в корне дерева. Если x лежит в диапазоне $[a, b]$, то запись в корне дерева включается в результаты поиска. Если $x \leq a$, то по свойству дерева поиска в левом поддереве нет искомых записей, поскольку ключи в узлах меньше x. Соответственно ведём поиск только в правом поддереве. Аналогично при $x \geq b$ не ведём поиск в правом поддереве (см. рис. 16).

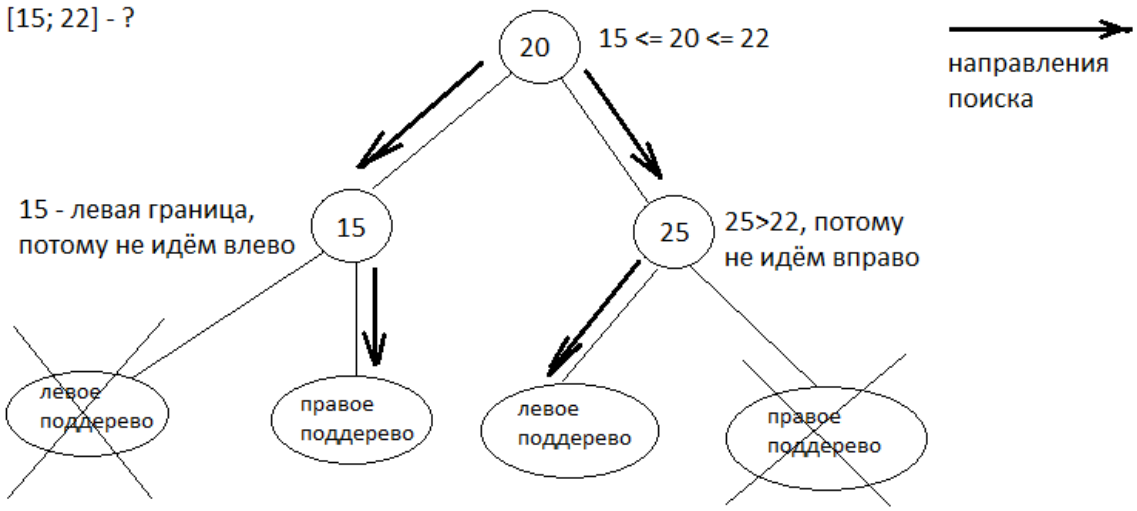


Рис. 17. Простой вариант диапазонного поиска

Самый простой способ перехода к поддеревьям - рекурсия. Можно предложить и итерационный вариант на основе стека.

1. В стек добавляем корень дерева.

2. На каждой итерации извлекаем из стека текущий узел и добавляем в стек тех сыновей, для которых есть смысл продолжить поиск. Если ключ в извлечённом узле удовлетворяет условиям поиска, соответствующая запись включается в результаты диапазонного поиска.

Ещё один вариант диапазонного поиска возникает, если создать дополнительные связи между узлами с соседними по величине ключами. Тогда можно действовать следующим образом. Ищем узел с ключом a. Поиск заканчивается в некотором узле N. Независимо от того, найден ли узел с ключом a, идём от N по дополнительным указателям в порядке возрастания значений ключей, пока не встретим

узел, в котором ключ больше b (см. рис. 18). Этот метод, очевидно, быстрее предыдущего, но несколько усложняет вставки и удаления как с точки зрения быстродействия, так и с точки зрения программирования.

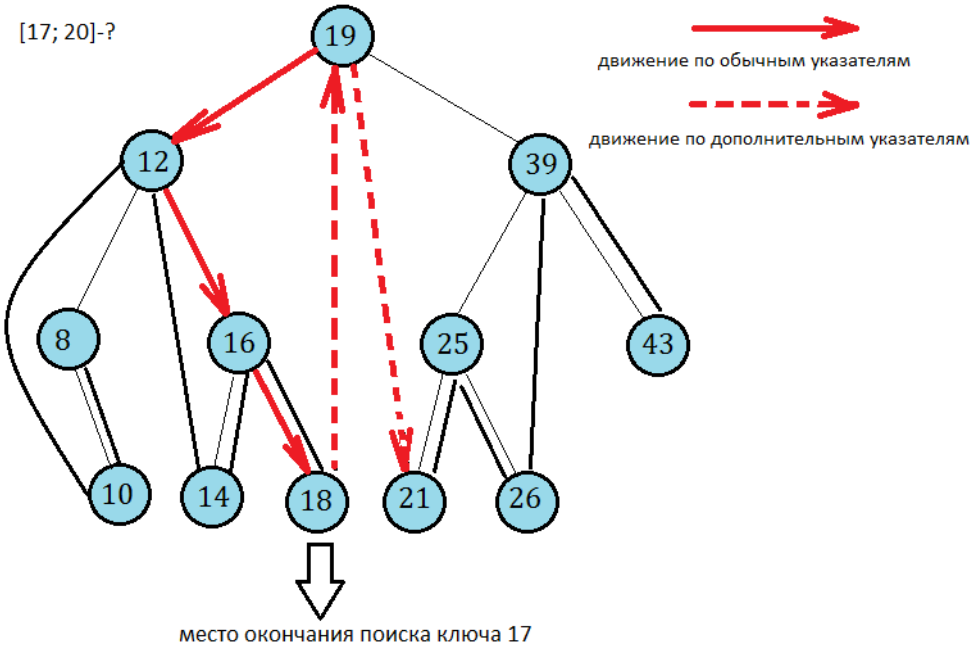


Рис. 18. Диапазонный поиск на основе дополнительных указателей

Построение оптимальных деревьев поиска. В некоторых практических приложениях множество записей часто меняется, то есть выполняется добавление, удаление записей, а в некоторых остаётся неизменным, будучи заранее заданным, или же изменения очень редки. Коллекций записей, которые редко меняются, в реальном мире встречается довольно много. Примеры - записи о государствах (изменения довольно редки, хотя иногда бывают), биологических видах, созвездиях, химических элементах. Если мы индексируем такие коллекции с помощью деревьев поиска, то полезно попытаться выполнить построение оптимального дерева по заданной коллекции. Если в бинарном дереве поиска N узлов, то высота его равна как минимум $h=[\log_2 N]$, где квадратные скобки означают целую часть. Если количество узлов в дереве имеет вид 2^h-1, то в идеале все листья находятся на уровне h (если считать уровень корня нулевым), в противном случае в идеале листья находятся на уровнях h и h-1 [1]. Допустим, $N=12$, тогда $h=3$, то есть листья должны быть на уровнях 2 и 3. Два дерева, приведённые на рис. 19, являются примером таких деревьев. Здесь листья обозначены цветом.

На рис. 20 приведены примеры деревьев, структура которых нежелательна – эти деревья плохо сбалансированы.

Пусть дерево строится на основе записей отсортированного массива или списка. Существует два основных подхода к построению: *восходящий* и *нисходящий*. [6]

Восходящее построение дерева состоит в следующем. Элементы массива или списка рассматриваются как деревья с единственным узлом. Объединяем попарно соседние деревья (первое со вторым, третье с четвёртым и т.д.). Количество деревьев сокращается вдвое. Укрупнённые деревья опять объединяем попарно. Действие повторяется, пока не останется всего одно дерево. Недостаток метода в том, что дерево может получиться далеко не лучшим с точки зрения сбалансированности.

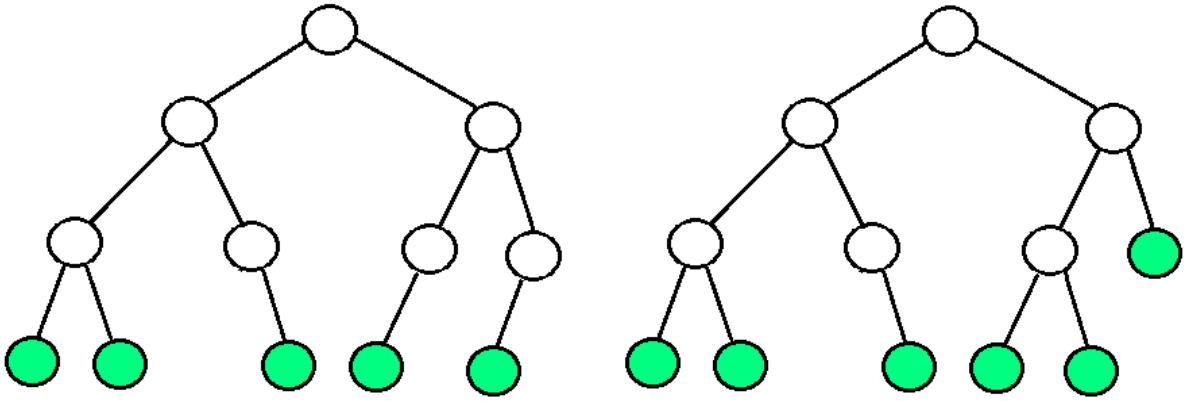


Рис. 19. Оптимальные деревья поиска

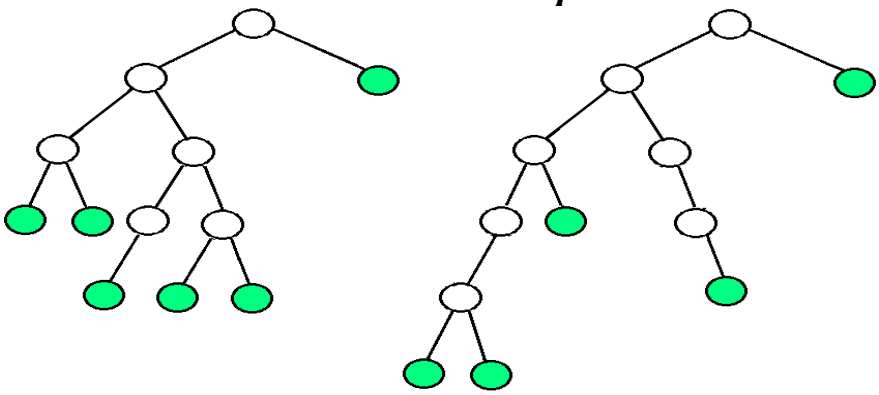


Рис. 20. Неэффективные деревья поиска

Более подробно рассмотрим здесь нисходящее построение дерева поиска. Легче всего его описать рекурсивно - берём середину отсортированной последовательности *M*, на основе подпоследовательности элементов, меньших *M*, создаём оптимальное дерево *L*, на основе другой подпоследовательности - оптимальное дерево *R*. Элемент *M* становится корнем результирующего дерева, *L* - левым поддеревом, *R* - правым поддеревом. Очевидно, что в этом

случае количество элементов в поддеревьях отличается не более, чем на 1 [3], поэтому получаем хорошо сбалансированное дерево. Проблема лишь в том, что данный метод не очень подходит для построения дерева на основе линейного списка - для каждого подсписка приходится выполнять O(p) операций при поиске середины списка, где p - длина подсписка. Очевидно, что поскольку в массиве доступ по заданной позиции выполняется за время O(1), для построения дерева по массиву нисходящий алгоритм весьма удачен. На рис. 21 приведём пример построения оптимального дерева поиска по нисходящему алгоритму.

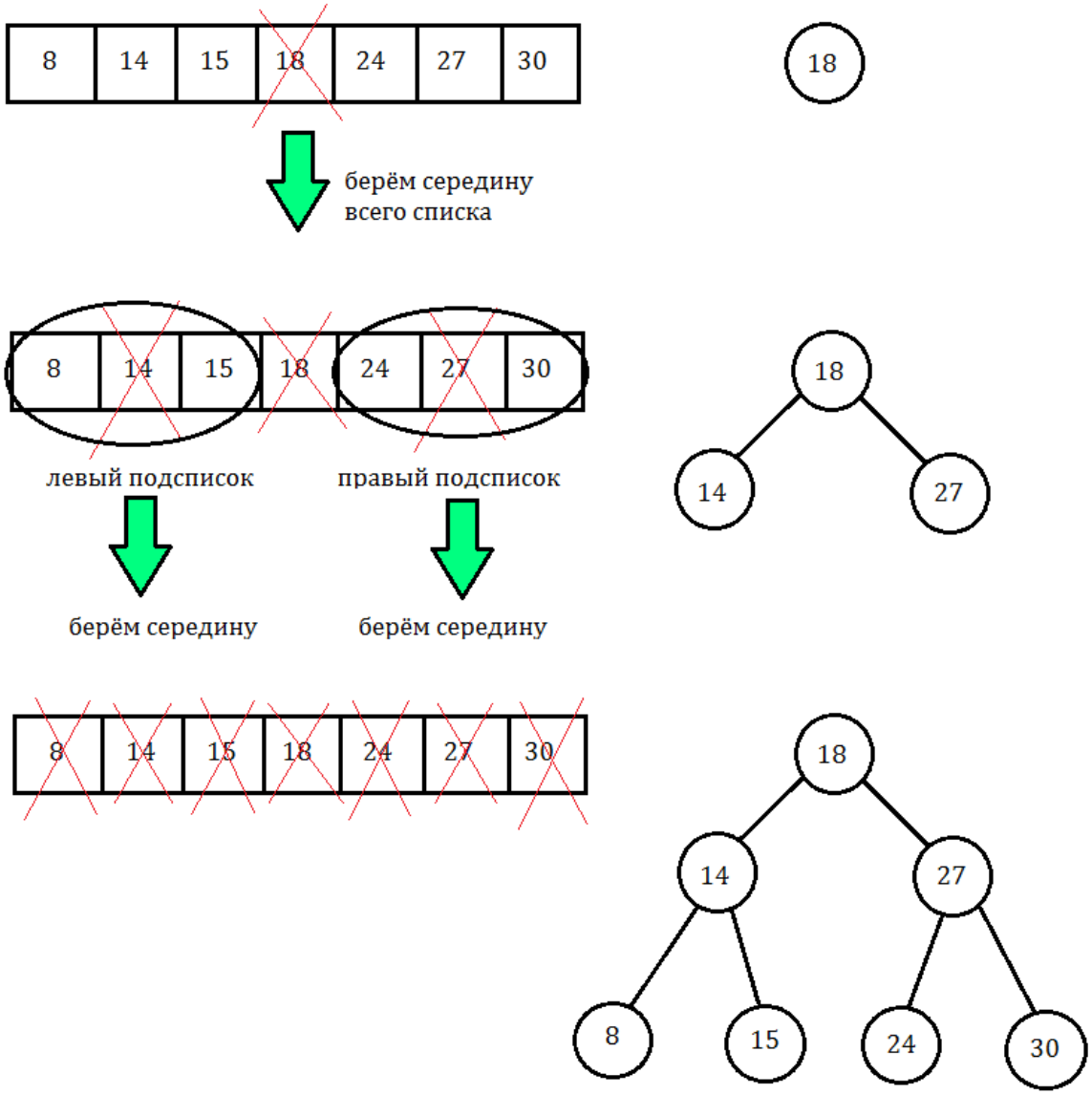


Рис. 21. Пример построения оптимального дерева поиска

Рассмотрим, как можно реализовать нисходящее построение на C++. Пусть дана структура данных «Узел дерева», полями которой являются ключ и указатели на потомков (см. листинг 1.1), а также

определён класс «Дерево», который использует такую структуру данных. Класс имеет метод Add вставки записи по правилам бинарного дерева поиска. Тогда построение дерева на основе отсортированного массива выполняется по алгоритму, приведённому в листинге 5. Заметим, что операция определения середины массива должна выполняться осторожно – если в C++ деление целых выполняется как целочисленное, то в языках, где нет явной типизации (скриптовые языки, языки функционального программирования) следует явно указать, что деление целочисленное.

Листинг 5 Построение оптимального дерева поиска

```
struct node
  {
        int data; /* ключ */
        node *left; /* ссылка на левого потомка */
        node *right; /* ссылка на правого потомка */
void Derevo::VstavkaSerial(int A[], int left, int right)
{
  if(left > right) return; // граничное условие
  int mid = (left + right) / 2; // середина подмассива

  this->Add(A[mid]);   //   вставляем   элемент   из   середины
подмассива

  /* подмассив разбивается на две части,
  для них рекурсивно вызываем нашу функцию */
  VstavkaSerial(A, left, mid-1);
  VstavkaSerial(A, mid+1, right);
}
```

Легко заметить, что за счёт рекурсии метод построения дерева оказался простым. Попытаемся разобраться, каковы затраты памяти и времени на этот алгоритм. Поскольку строится оптимальное по высоте дерево, а таковое характеризуется логарифмической оценкой высоты, то легко определить, что при наличии n записей в исходной коллекции рекурсия потребует $O(\log n)$ дополнительной памяти: расход памяти на рекурсию определяется глубиной рекурсии. Выполняется n вставок, каждая в среднем за время $O(\log n)$, поэтому время работы составляет $O(n \log n)$. Это вполне приемлемые показатели.

Преобразование деревьев в списки. Выше была рассмотрена операция получения дерева на основе заданного массива или списка. Иногда может потребоваться обратная операция - преобразование дерева в отсортированный список (см. рис. 22).

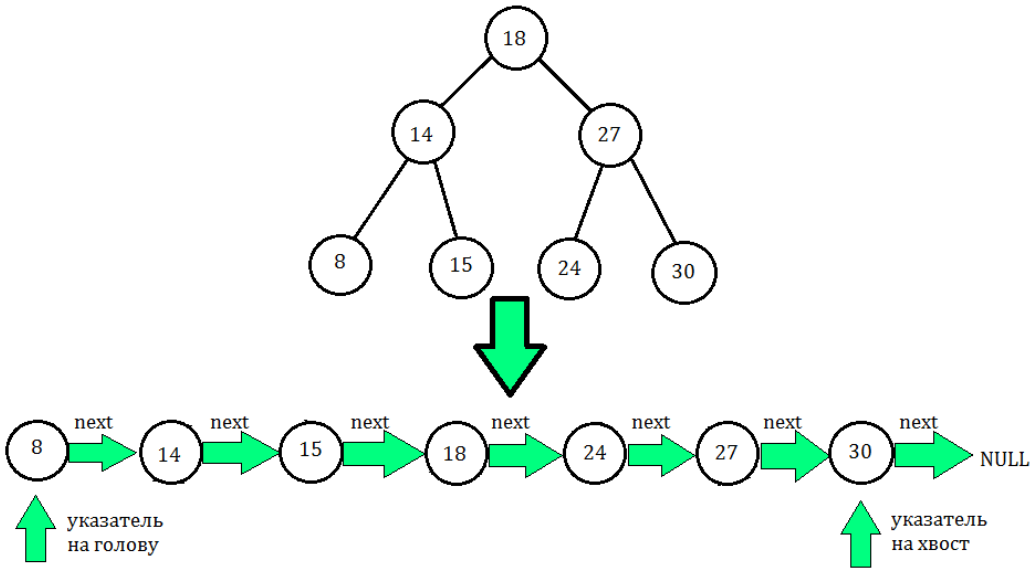


Рис. 22. Пример преобразования дерева в список

При выполнении данной операции необходимо учесть, что оценка её трудоёмкости будет определяться двумя факторами:

1) сложностью обхода дерева;

2) сложностью вставки в список.

Если заранее неизвестно, где элемент будет находиться при вставке в список, получим, что сложность вставки будет O(n), где n - количество элементов в списке, поскольку нам нужно будет пройти примерно половину списка. Если заранее известно, что вставляемый элемент больше всех имеющихся, то мы можем обеспечивать вставку за время O(1), храня указатель не только на голову, но и на хвост списка. Соответственно сложность вставки всех n записей будет O(n). Сложность такого же порядка у симметричного обхода дерева, поскольку каждый узел посещается один раз. Следует применить симметричного обхода дерева: при таком обходе записи выводятся в порядке возрастания ключей [4]. Так мы обеспечиваем построение списка за время O(n). При использовании рекурсии при симметричном обходе получим, что для хорошо сбалансированного дерева требуется O(log n) дополнительной памяти, поскольку глубина рекурсии линейно зависит от высоты дерева. Память под сам список здесь не учитываем - её объём зависит от типа списка, но не алгоритма преобразования дерева в этот список.

Пусть структура узла дерева имеет вид, как в листинге 5. В листинге 6 показано, как организуется список. Очевидно, что сначала список пустой. Реализуем вставку в хвост списка (см. листинг 7).

Листинг 6. Структура для хранения списка

```
struct SPISOK
{
 uzel * head; /* указатель на голову */
 uzel * tail; /* указатель на хвост */

 SPISOK() { head = NULL; tail = NULL; } // конструктор

 void VstavkaHvost(int key); // вставка
 void Postroenie(node * root); // построение на основе дерева
};
```

Листинг 7. Вставка в хвост списка

```
void SPISOK::VstavkaHvost(int key)
{
 // если список пустой
 if(head == NULL)
 {
       head = new uzel(key, NULL);
       tail = head; /* у одноэлементного списка голова и
хвост совпадают */
       return;
 }

 // если список непустой
 uzel * p = new uzel(key, NULL);
 tail->next = p; // добавляем в хвост
 tail = p; // обновляем указатель на хвост
}
```

Далее реализуем само построение списка на основе дерева (см. листинг 8). В данном случае предполагается, что реализованы функции работы с деревом, но здесь мы их не рассматриваем. Используем алгоритм симметричного обхода дерева: 1) обойти левое поддерево; 2) вывести корень; 3) обойти правое дерево.

Листинг 8. Построение списка на основе дерева

```
/* построение списка */
void SPISOK::Postroenie(node * root)
{
 // обходим левое поддерево
 if(root->left) Postroenie(root->left);

 // добавляем содержимое корня в список
 VstavkaHvost(root->data);

 // обходим правое поддерево
 if(root->right) Postroenie(root->right);
}
```

Если расходы памяти крайне критичны, то можно реализовать нерекурсивное построение. Например, можно использовать

прошитые деревья. Другой вариант - использовать закономерности, позволяющие определить, как из текущего узла попасть в следующий по величине. Пусть в узле дерева дополнительно храним указатель на родителя, *V* - текущий узел. Тогда верны следующие правила (*при выполнении условий ближайшего из них не проверяем остальные*).

1.Если у *V* есть правый сын, то узел, следующий после *V* получаем прохождением самого левого пути от V->right (то есть спускаемся от V->right, каждый раз направляясь влево, пока не найдём узел без левого сына).

2.Если *V* - левый сын своего родителя, то родитель узла *V* - следующий по величине узел.

3.Если *V* - правый сын своего родителя, выполняем восхождение, пока не найдём узел *M*, являющийся левым сыном своего родителя. Если мы не нашли *M*, то ключ в узле *V* - наибольший, в противном случае следующий по величине узел - родитель *M*.

Найти стартовый узел можно, спускаясь от корня всегда влево, пока не найдём узел без левого сына. Если левое поддерево пустое, то таким узлом будет корень.

По описанным правилам имеется довольно много путей длиной O(log *n*): 1. Путь, когда мы ищем стартовый узел. 2. Когда мы идём от узла, для которого корень - следующий по величине ключа. 3. Когда мы идём от узла с наибольшим ключом вверх по дереву. 4. Пути от узлов, имеющих правых сыновей и расположенных далеко от листьев.

Пусть узел расположен на некотором уровне до *n*/2, тогда оценка сложности его спуска составляет O(log *n*). Количество узлов, которые находятся на уровнях до *n*/2, оценивается как O(*n*), соответственно количество узлов, от которых мы идём по длинному пути типа 4, имеет оценку этого порядка. Разумно предположить, что такой обход характеризуется временем O(*n* log *n*) и расходом дополнительной памяти O(1) - алгоритм итерационный. Построение списка при использовании такого обхода характеризуется временем O(*n* log *n*) и дополнительным расходом памяти O(1).

Удаление деревьев Кроме многочисленных операций, которые могут быть предусмотрены для редактирования дерева и поиска в нём, обязательно должна быть операции удаления дерева, когда оно больше не требуется, поскольку в противном случае будут иметь

место существенные потери памяти. Удалить непустое дерево можно с помощью рекурсии - сначала удаляем поддеревья, затем корень (см. листинг 9). При этом если записи, хранящиеся в узлах, используют динамическую память, то нужно проследить, чтобы она была освобождена. На рис. 22 наглядно показано, как работает рекурсивный алгоритм удаления дерева и в какой последовательности удаляются узлы дерева.

Нетрудно заметить, что каждый узел посещается один раз, потому сложность удаления составляет O(n). Для хорошо сбалансированного дерева глубина рекурсии составляет O(log n).

Листинг 9. Рекурсивная процедура удаления дерева

```
void RemoveTree(node * root)
{
  // удаляем левое поддерево
  if(root->left) RemoveTree(root->left);

  // удаляем правое поддерево
  if(root->right) RemoveTree(root->right);

  // удаляем корень
  delete root;
};
```

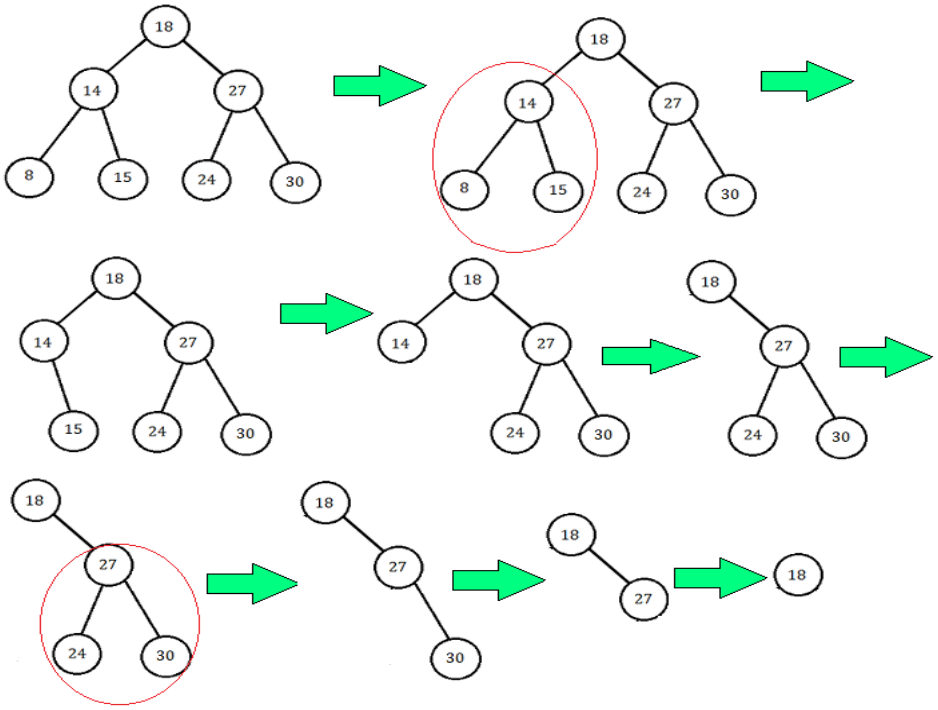


Рис. 23. Пример рекурсивного удаления дерева

Можно удалить дерево и без рекурсии, организовав стек [2]. Добавляем корень в стек, и далее действуем следующим образом.

Пусть V - вершина стека. В цикле выполняем следующее. Если у узла V нет потомков, удаляем его (если V - корень, то удаление дерева завершено, а стек становится пустым). В противном случае добавляем сыновей в стек, удаляем V и переходим к следующей итерации цикла.

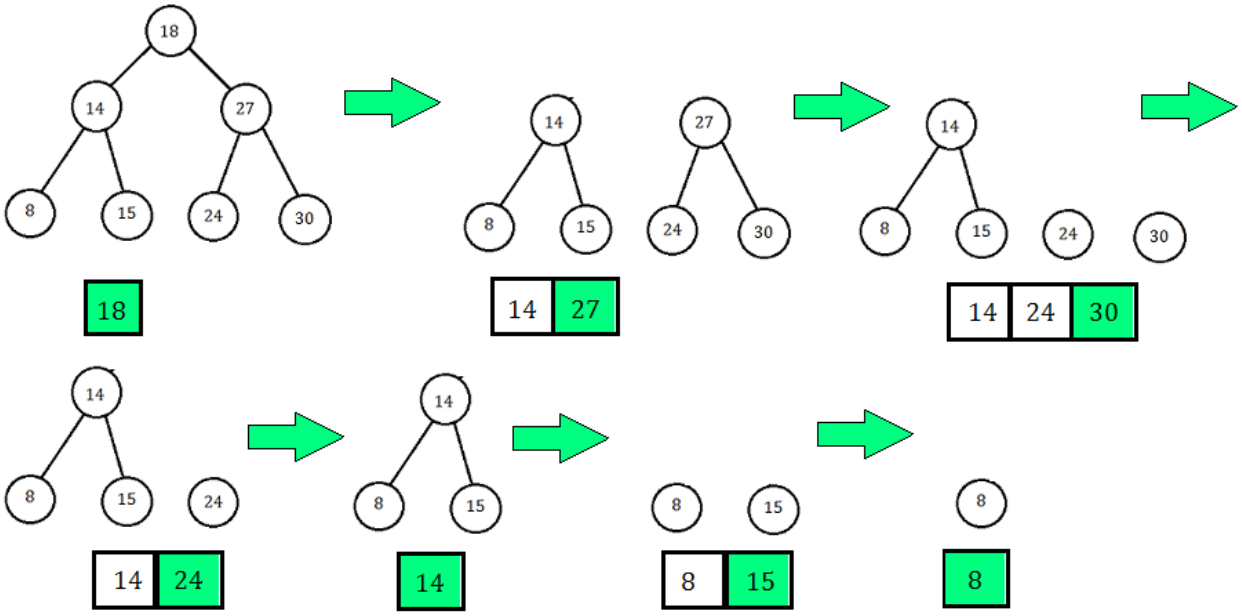


Рис. 24. Пример удаления дерева с помощью стека

Сложность алгоритма составляет O(n), поскольку каждый узел посещается лишь один раз. Максимальный размер L стека при n=1 составляет 1 и, как можно заметить при внимательном подробном рассмотрении работы на различных деревьях, для хорошо сбалансированных деревьев при добавлении ещё одного заполненного уровня он увеличивается в среднем на 1. Учитывая, что с ростом высоты хорошо сбалансированного дерева на 1 число узлов примерно удваивается, получаем, что в среднем $L(n) = 1 + L(n/2)$, то есть требуется O(log n) памяти под стек.

Основные понятия

Бинарное (двоичное) дерево – дерево, в котором каждая вершина имеет не более двух потомков.

Вершина (узел) дерева – каждый элемент дерева.

Ветви дерева – направленные дуги, которыми соединены вершины дерева.

Высота (глубина) дерева – число уровней, на которых располагаются его вершины.

Дерево – структура данных, представляющая собой совокупность элементов и отношений, образующих иерархическую структуру этих элементов.

Корень дерева – начальный узел дерева, ему соответствует нулевой уровень.

Листья дерева – вершины, в которые входит одна ветвь и не выходит ни одной ветви.

Неполное бинарное дерево – дерево, уровни которого заполнены не полностью.

Нестрогое бинарное дерево – дерево, у которого вершины имеют степень ноль (у листьев), один или два (у узлов).

Обход дерева – упорядоченная последовательность вершин дерева, в которой каждая вершина встречается только один раз.

Поддерево – часть древообразной структуры данных, которая может быть представлено в виде отдельного дерева.

Полное бинарное дерево – дерево, которое содержит только полностью заполненные уровни.

Потомки – все вершины, в которые входят ветви, исходящие из одной общей вершины.

***Почти сбалансированное дере*во** – дерево, у которого длины всевозможных путей от корня к внешним вершинам отличаются не более, чем на единицу.

Предок – вершина, из которой исходят ветви к вершинам следующего уровня.

Сбалансированное дерево – дерево, у которого длины всех путей от корня к внешним вершинам равны между собой.

Степень вершины – число дуг, которое выходит из этой вершины.

Степень дерева – максимальная степень вершин, входящих в дерево.

Строгое бинарное дерево – дерево, у которого вершины имеют степень ноль (у листьев) или два (у узлов).

Упорядоченное дерево – дерево, у которого ветви, исходящие из каждой вершины, упорядочены по определенному критерию.

Уровень вершины – число дуг от корня дерева до вершины.

Резюме

1. Бинарные деревья являются одними из наиболее широко распространенных структур данных в программировании, которые представляют собой иерархические структуры в виде набора связанных узлов.

2. Каждое дерево обладает следующими свойствами: существует узел, в который не входит ни одной дуги (корень); в каждую вершину, кроме корня, входит одна дуга.

3. С понятием дерева связаны такие понятия, как корень, ветвь, вершина, лист, предок, потомок, степень вершины и дерева, высота дерева.

4. Списочное представление деревьев основано на элементах, соответствующих вершинам дерева.

5. Дерево можно упорядочить по указанному ключу.

6. Просмотреть с целью поиска все вершины дерева можно с помощью различных способов обхода дерева.

7. Наиболее часто используемыми обходами являются прямой, симметричный, обратный.

8. В программировании при решении большого класса задач используются бинарные деревья.

9. Бинарные деревья по степени вершин делятся на строгие и нестрогие, по характеру заполнения узлов – на полные и неполные, по удалению вершин от корня – на сбалансированные и почти сбалансированные.

10. Основными операциями с бинарными деревьями являются: создание бинарного дерева; печать бинарного дерева; обход бинарного дерева; вставка элемента в бинарное дерево; удаление элемента из бинарного дерева; проверка пустоты бинарного дерева; удаление бинарного дерева.

11. Бинарные деревья могут применяться для поиска данных в специально построенных деревьях (базы данных), сортировки данных, вычислений арифметических выражений, кодирования.

3. ТЕХНИКА БЕЗОПАСНОСТИ ПРИ ВЫПОЛНЕНИИ ЛАБОРАТОРНОЙ РАБОТЫ

Меры безопасности при работе с электротехническими устройствами соответствуют мерам безопасности, принимаемым при эксплуатации установок с напряжением до 1000 В и разработанным в соответствии с «Правилами техники безопасности при эксплуатации электроустановок потребителей», утвержденными Главгосэнергонадзором 21 декабря 1984 г.

Студенту не разрешается приступать к выполнению лабораторной работы, если замечены какие-либо неисправности в лабораторном оборудовании.

Студент не должен прикасаться к токоведущим элементам электрооборудования, освещения и электропроводке, открывать двери электрошкафов и корпусов системных блоков, мониторов.

Студенту запрещается прикасаться к неизолированным или поврежденным проводам и электрическим устройствам, наступать на переносные электрические провода, лежащие на полу, самостоятельно ремонтировать электрооборудование и инструмент.

Обо всех замеченных неисправностях электрооборудования студент должен немедленно поставить в известность преподавателя или лаборанта.

При выполнении лабораторной работы необходимо соблюдать осторожность и помнить, что только человек, относящийся серьезно к своей безопасности, может быть застрахован от несчастного случая.

4. УКАЗАНИЯ К ВЫПОЛНЕНИЮ ЛАБОРАТОРНОЙ РАБОТЫ

Каждую задачу необходимо решить в соответствии с изученными методами формирования, вывода и обработки данных в виде бинарных деревьев. Обработку бинарных деревьев выполнить на основе базовых алгоритмов: поиск, вставка элемента, удаление элемента, удаление всей динамической структуры. При объявлении списков выполните комментирование используемых полей. Программу для решения каждой задачи необходимо разработать методом процедурной абстракции, оформив комментарии к коду.

Каждую задачу реализовать в соответствии с приведенными этапами:

- изучить словесную постановку задачи, выделив при этом все виды данных;
- сформулировать математическую постановку задачи;

- выбрать метод решения задачи, если это необходимо;
- разработать графическую схему алгоритма;
- записать разработанный алгоритм на языке С++;
- разработать контрольный тест к программе;
- отладить программу;
- составить отчет о лабораторной работе.

5. ЗАДАЧИ К ЛАБОРАТОРНОЙ РАБОТЕ

1. Задано бинарное дерево. Построить прошитое дерево, соответствующее обходу слева направо. Составить программу удаления поддерева, висящего на заданной вершине. Конечное дерево должно оставаться прошитым. Выдать информацию о нитях исходного и конечного деревьев.

2. Генеалогическое дерево человека строится следующим образом. Корень дерева соответствует данному человеку. Сыновья каждой вершины описывают родителей соответствующего вершине человека. Имеется два генеалогических дерева. Требуется определить, являются ли эти люди:

двоюродными братьями или сестрами;

сводными братьями или сестрами.

3. Теннисный турнир проходит по олимпийской системе с выбыванием. Третий призер определяется в матче игроков, проигравших полуфинальные встречи. Известен рейтинг каждого игрока. Турнир записан с помощью дерева. Участникам соответствуют листья дерева. Известно, что турнир прошел в полном соответствии с рейтингом игроков. Требуется выдать результаты по турам и определить первых трех призеров.

4. Имеется массив записей, расположенных по возрастанию некоторого ключевого поля. Одинаковых ключей нет. Построить из элементов массива бинарное дерево такое, что для каждой вершины ключ левого сына меньше ключа отца, а ключ отца меньше ключа правого сына. Составить программу быстрого поиска элемента с заданным ключом, возвращающую номер элемента массива, если есть элемент с этим ключом. Если такого элемента нет, выдать соответствующее сообщение.

5. Для хранения информации о ключевых словах некоторой программы имеется таблица перекрестных ссылок. Для каждого

слова задан список номеров строк, где оно встречается, а для каждой строки - число образцов данного слова в строке. Ключевые слова расположены по алфавиту. Построить бинарное дерево для поиска ключевых слов. Составить программу определения общего количества вхождений заданного ключевого слова.

6. Теннисный турнир проходит по олимпийской системе с выбыванием. Известен рейтинг каждого игрока. Результаты турнира записаны с помощью дерева. Участникам соответствуют листья дерева. Выдать список сенсаций турнира, когда побеждал игрок с низшим рейтингом. Определить самый сенсационный результат по максимальной разнице рейтингов.

7. В листьях дерева, соответствующего некоторой конструкции, указаны минимально возможные значения массы. Задана предельно допустимая масса конструкций. Требуется определить максимально возможные значения массы для каждого узла, соответствующего листу дерева.

8. В листьях бинарного дерева указаны идентификаторы переменных, в других вершинах - знаки арифметических операций или функций SIN, COS, TG, CTG, LOG, EXP. Возможны одноместные операции типа '+' или '-'. В этом случае требуется только один операнд. Значения переменных известны. Проверить синтаксическую правильность идентификаторов. Выдать на экран выражение в инфиксной форме со скобками. Определить значение выражения.

Пример: ((-(((a1*bar) +c))- (((SIN (dors))-e)))).

9. Задано бинарное дерево. Построить прошитое дерево, соответствующее обходу сверху вниз. Составить программу удаления поддерева, висящего на заданной вершине. Конечное дерево должно оставаться прошитым. Выдать информацию о нитях исходного и конечного дерева.

10. Разработать рекурсивный алгоритм и программу, преобразующую лес в эквивалентное ему бинарное дерево.

11. Разработать алгоритм и программу для определения подобия двух бинарных деревьев на основе методов обхода.

12. Составить процедуру подсчета числа узлов заданного бинарного дерева.

13. В некоторой файловой системе справочник файлов организован в виде упорядоченного двоичного дерева. Каждой

вершине соответствует некоторый файл, здесь содержится имя файла и дата последнего обращения к нему, закодированная целым числом. Напишите программу, которая обходит дерево и удаляет все файлы, последнее обращение к которым происходило до некоторой определенной даты.

14. В некоторой древовидной бинарной структуре опытным путем измеряется частота обращения к каждому из элементов. Для этого с любым из элементов связан счетчик обращений. По прошествии определенного периода времени дерево реорганизуется. Для этого оно просматривается и с помощью соответствующей программы строится новое дерево, в которое ключи включаются в порядке убывания счетчиков частот обращения. Напишите программы, выполняющие такую реорганизацию. Будет ли средняя длина пути в новом дереве равна, больше или даже значительно меньше длины пути в оптимальном дереве?

15. Компонентами файла *number* являются целые числа. Во входном файле расположена последовательность целых чисел. Вывести те числа из файла *number*, которые не входят в данную последовательность. Указание: для быстрой проверки факта вхождения числа из файла *number* в данную последовательность построить дерево, содержащее все различные числа из входного файла.

16. Во входном файле расположена последовательность целых чисел. Выяснить есть ли в этой последовательности совпадающие числа. Решить задачу с помощью дерева.

17. Написать программу построения бинарного дерева с помощью связных структур и поиска в дереве при обратном порядке обхода его.

18. Написать программу построения бинарного дерева с помощью связных структур и поиска в дереве при прямом порядке обхода его.

19. Написать программу построения бинарного дерева с помощью связных структур и поиска в дереве при симметричном порядке обхода его.

20. Написать программу построения бинарного дерева с помощью массивов и поиска в дереве при обратном порядке обхода его.

21. Написать программу построения бинарного дерева с помощью массивов и поиска в дереве при прямом порядке обхода его.

22. Написать программу построения бинарного дерева с помощью массивов и поиска в дереве при симметричном порядке обхода его.

23. Напишите программу, которая формирует двоичное дерево, узлам которого прописаны имена, и печатает эти имена в префиксном, инфиксном и постфиксном порядке.

24. Рассмотрим задачу кодирования непустой последовательности целых чисел. Пусть дана последовательность целых чисел $a_1,a_2,\ldots,a_n$ и функция целочисленного аргумента F(x), принимающая целые значения. Значение F(x) будем называть кодом числа x. Требуется закодировать данные числа, т.е. вычислить значения $F(a_1),F(a_2),\ldots,F(a_n)$, при условии, что в последовательности $a_1,a_2,\ldots,a_n$ имеются частые повторения значений элементов, а способ вычисления F(x) достаточно сложен (т.е. нужно избежать повторных вычислений одного и того же значения). Для этого следует по ходу кодирования данных чисел строить таблицу (справочник) уже найденных кодов. Организация этой таблицы должна быть такой, чтобы в ней можно было быстро находить элемент с данным значением (или устанавливать факт отсутствия такого элемента в таблице), а также, чтобы без особых усилий добавлять в таблицу новые элементы. Этим требованиям удовлетворяет представление таблицы в виде двоичного дерева, в вершинах которого расположены различные элементы из данной последовательности и коды. Составить программу, решающую эту задачу в случае F(k)=k.

25. Компонентами файла number являются целые числа. Во входном файле расположена последовательность целых чисел. Вывести те числа из файла number, которые входят в заданную последовательность.

Указание. Для быстрой проверки факта вхождения числа из

файла number в данную последовательность построить дерево, содержащее все различные числа из входного файла

26. Во входном файле расположена последовательность целых чисел. Выяснить, есть ли в этой последовательности совпадающие

числа. Для этого построить дерево, содержащее все различные числа из входного файла.

27. В некоторой файловой системе справочник файлов организован в виде упорядоченного двоичного дерева. Каждой вершине соответствует некоторый файл, здесь содержится имя файла и, кроме всего прочего, дата последнего обращения к нему, закодированная целым числом. Напишите программу, которая обходит дерево и удаляет все файлы, последнее обращение к которым происходило до некоторой определенной даты.

28. Напишите программу, которая печатает отдельные слова, поступающие на ввод, отсортированными в порядке частоты встречаемости. Входные данные представить в виде двоичного дерева, узел которого представляет запись, включающую указатель на слово, счетчик слов (сколько раз встречается в тексте данное слово), указатель на левый потомок, указатель на правый потомок.

29. Составить процедуру подсчета числа листьев заданного бинарного дерева.

30. Написать программу поиска по случайному дереву при прямом порядке обхода его. Дерево представлено с помощью массива.

31. Написать программу построения случайного дерева с помощью связных структур и поиска в дереве при прямом порядке обхода его

32. Написать программу построения случайного дерева с помощью связных структур и поиска в дереве при обратном порядке обхода его.

33. Реализуйте программу, в которой выполняются все основные операции с бинарным деревом.

34. Найдите количество четных элементов бинарного дерева. Укажите эти элементы и их уровни.

35. Найдите сумму элементов бинарного дерева, находящихся на уровне k.

36. Оператор мобильной связи организовал базу данных абонентов, содержащую сведения о телефонах, их владельцах и используемых тарифах, в виде бинарного дерева. Составьте программу, которая:

обеспечивает начальное формирование базы данных в виде бинарного дерева;

производит вывод всей базы данных;

производит поиск владельца по номеру телефона;

выводит наиболее востребованный тариф (по наибольшему числу абонентов).

37. Описать логическую функцию same (T), определяющую, есть ли в бинарном дереве T хотя бы 2 одинаковых элемента.

38. Во внешнем текстовом файле *PROG* записана (без ошибок) некоторая программа на алгоритмическом языке. Известно, что в этой программе каждый идентификатор (служебное слово или имя) содержит не более 9 латинских букв и/или цифр. Напечатать в алфавитном порядке все различные идентификаторы этой программы, указав для каждого из них число его вхождений в текст программы. (Учесть, что в идентификаторах одноименные прописные и строчные буквы отождествляются, что внутри литерных значений, строк-констант и комментариев последовательности из букв и цифр не являются идентификаторами и что в записи вещественных чисел может встретиться буква *E* или *e.)*

39. Для хранения идентификаторов использовать дерево поиска, элементами которого являются пары—идентификатор и число его вхождений в текст программы.

40. Из случайного множества целых чисел (n>100) построить упорядоченное бинарное дерево. С помощью этого дерева найти максимальное и минимальное число.

41. Из случайного множества целых чисел (n>100) построить упорядоченное бинарное дерево. С помощью этого дерева найти количество заданных элементов.

42. Из случайного множества целых чисел (n>100) построить упорядоченное бинарное дерево. С помощью этого дерева найти среднее арифметическое всех элементов дерева.

43. Из случайного множества целых чисел (n>100) построить упорядоченное бинарное дерево. С помощью этого дерева найти число вершин на заданном уровне.

44. Из случайного множества целых чисел (n>100) построить упорядоченное бинарное дерево. С помощью этого дерева найти высоту дерева.

45. Проверить в двух заданных деревьях наличие одинаковых элементов.

46. Проверить на равенство двух заданных деревьев.

47. Написать программу поиска по случайному дереву при обратном порядке обхода его. Дерево представлено с помощью массива

48. В упорядоченном бинарном дереве с целочисленными ключами возведите в квадрат корневой элемент. Предложите алгоритм восстановления его упорядоченности и реализуйте его.

49.

6. УКАЗАНИЯ К ВЫПОЛНЕНИЮ ЛАБОРАТОНОЙ РАБОТЫ

Выполнение работы следует начать с изучения алгоритмов основных операций над сбалансированным бинарным деревом. При объявлении сбалансированных бинарных деревьев выполните комментирование используемых полей.

Каждую задачу реализовать в соответствии с приведенными этапами:

- изучить словесную постановку задачи, выделив при этом все виды данных;
- сформулировать математическую постановку задачи;
- выбрать метод решения задачи, если это необходимо;
- разработать графическую схему алгоритма;
- записать разработанный алгоритм на алгоритмическом языке;
- разработать контрольный тест к программе;
- отладить программу;
- составить отчет о лабораторной работе.

7. ПРИМЕР РЕШЕНИЯ ЗАДАЧИ

В листьях бинарного дерева указаны цифры либо идентификаторы переменных, заданные строчными латинскими буквами, в других вершинах знаки арифметических операций. Значения переменных известны. Выдать на экран выражение в

префиксной и постфикской формах. Определить значение выражения.

```
Program TreeCount;
    Uses crt;
    Const
    zn=['+', '-', '/', '*', '^'];
    let={'a'., 'z'}; {буква}
    dig=['0'., '9']; {цифры}
    smv=zn+let+dig; {допустимые символы при вводе} Type
    ukaz=^uzel;
    uzel=record {структура вершины дерева}
    key: char; {знак, цифра или буква}
    left, right: ukaz {сыновья}
    end; Var
    root, kon: ukaz;
    a: array ['a'..'z'] of real; {массив значений идентификатора дерева} s: set of char {множество введенных букв} m, k char; Procedure Sozd (t: ukaz) ; {рекурсивная процедура создания исходного дерева} {t-указатель на корень}

- 7 Begin
    if t<>nil then
    begin
    Repeat
    Write ('Введите значение вершины') ;
    Readln (k);
    if not (k in smv) then
Writeln ('Неправильный символ, повторите ввод') ;
    Until k in smv;
    t^.key: =k;
    if not (k in zn) then {к-буква или цифра}
    begin
t^.left: =nil;
t^.right: =nil;
if k in let then {к-буква}
s: =s+[k] {добавление в множество}
    end
    else
    begin
Writeln ('Переходим к левому сыну вершины,
t^, key) ;
New (kon) ;
```

```
t^, left: =kon
    end;
    Sozd (t^.left) ;
    if t^.left<>nil then {t^.key-буква или цифра}
begin
Writeln ('Переходим к правому сыну вершины',
t^.key) ;
New (kon) ;
t^.right: =kon
end;
    Sozd (t^.right)
    end
    End; Procedure Pechpo (t:ukaz) ; {вывод на экран
выражения в постфиксной форме} Begin
    if t<>nil then
    begin
    PecgPo (t^.left) ;
    PechPo (t^.right) ;
    Write (t^.key, ' ')
    end End; Procedure PechPr (t:ukaz) ; {вывод на экран
выражения в префиксной форме} Begin
    if t<> nil then
    begin
    Write (t^.key, ' ') ;
    PechPr (t^.left) ;
    PechPr (t^.right) ;
    end End; Function f1 (t:ukaz) :real; {расчет значения
выражения, заданного бинарным деревом} Begin
    if t^.left-nil then {лист: в t^.key цифра или буква}
    if t^.key in dig then {цифра}
    a1: =ord (t^.key) -ord ('0') {числовое значение}
    else {буква-идентификатор}
    f1: =a[t^.key] {значение идентификатора} else case
t^.key of {не лист: в t^.key знак}
    '+': f1: =f1 (t^.left) +f1 (t^.right) ;
    '-': f1: =f1 (t^.left) -f1 (t^.right) ;
    '*': f1: =f1 (t^.left) *f1 (t^.right) ;
    '/': f1: =f1 (t^.left) /f1 (t^.right) ;
    '^': f1: =exp (f1 (t^.right) *ln (f1 (t^.left) )) )
    end End: Begin
    s: =[]; {пустое множество}
    Clrscr;
    New (root) ;
    Sozd (root);
    Writeln ('Ввод закончен l') ;
```

```
    Readln; {пауза}
    PechPo (root) ;
    Writeln ('- постфикская форма') ;
    Readln;
    PechPr (root) ;
    Writeln ('- префиксная форма') ;
    Readln;
    For n: ='a' to 'z' do {ввод значений идентификаторов}
    if n in s then
    begin
    Write (n, '='_;
    Readln (a[n]) ;
    end;
Writeln ('Значение выражения: ' ', f1 (root) :1: 3) ;
 Readln
End.
```

8. КОНТРОЛЬНЫЕ ВОПРОСЫ

1. С чем связана популярность использования деревьев в программировании?
2. Можно ли список отнести к деревьям? Ответ обоснуйте.
3. Какие данные содержат адресные поля элемента бинарного дерева?
4. Может ли бинарное дерево быть строгим и неполным? Ответ обоснуйте.
5. Может ли бинарное дерево быть нестрогим и полным? Ответ обоснуйте.
6. Каким может быть почти сбалансированное бинарное дерево: полным, неполным, строгим, нестрогим? Ответ обоснуйте.
7. Куда может быть добавлен элемент в бинарное дерево в зависимости от его вида (полное, неполное, строгое, нестрогое)? Вид дерева при этом должен сохраниться.
8. Куда может быть добавлен элемент в сбалансированное бинарное дерево? Вид дерева при этом должен сохраниться.
9. Чем отличаются, с точки зрения реализации алгоритма, прямой, симметричный и обратный обходы бинарного дерева?

9. СПИСОК РЕКОМЕНДУЕМОЙ ЛИТЕРАТУРЫ

1. Кнут, Д. Искусство программирования для ЭВМ Т.1,

Основные алгоритмы / Д. Кнут – М.: Вильямс 2000. – 215 с.

2. Вирт, Н. Алгоритмы и структуры данных / Н. Вирт – М.: МИР, 1989. – 363 с.

3. Пападимитриу, Х. Комбинаторная оптимизация. Алгоритмы и сложность: пер. с англ. / Х. Пападимитриу, К.М. Стайглиц. – М.: Мир 1985. – 512 с.

4. Топп, У. Структуры данных в С ++: пер. с англ. / У. Топп, У. Форд. – М.: БИНОМ, 1999. – 816 с.

5. Кормен, Т. Алгоритмы: построение и анализ / Т. Кормен, Ч. Лейзерсон, Р. Ривест. – М: МЦНМО, 1999. –960 с.

6. Гэри, М. Вычислительные машины и труднорешаемые задачи / М. Гэри, Д. Джонсон. – М.: Мир, 1982. – 416 с.

7. Ахо, А. В. Структуры данных и алгоритмы / А. В. Ахо, Д. Э. Хопкрофт, Д. Д. Ульман. – М.: Вильямс, 2000. – 384 с.

8. Кубенский, А. А. Создание и обработка структур данных в примерах на Java / А. А.Кубенский. – СПб.: БХВ-Петербург, 2001. – 336 с.

9. Седжвик, Р. Фундаментальные алгоритмы на С++. Анализ. Структуры данных. Сортировка. Поиск: пер. с англ. /Р. Седжвик. – К.: ДиаСофт, 2001. – 688 с.

10. Хэзфилд, Р. Искусство программирования на С. Фундаментальные алгоритмы, структуры данных и примеры приложений. Энциклопедия программиста: пер. с англ. / Р. Хэзфилд, Л. Кирби [и др.]. – К.: ДиаСофт, 2001. – 736 с.

11. Мейн, М. Структуры данных и другие объекты в С++: пер. с англ. / М .Мейн, У. Савитч. – 2-е изд.– М.: Вильямс, 2002. – 832 с.

12. Хусаинов, Б.С. Структуры и алгоритмы обработки данных. Примеры на языке Си (+ CD): учеб. Пособие / Б.С. Хусаинов. – М.: Финансы и статистика, 2004. – 464 с.

13. Макконнелл, Дж. Основы современных алгоритмов / Дж. Макконнелл. – 2-е изд. – М.: Техносфера, 2004. – 368с.

14. Гудман, С.Введение в разработку и анализ алгоритмов / С. Гудман, С.Хидетнием. – М.: Мир, 1981. – 206 с.

15. Ахо, А. Построение и анализ вычислительных алгоритмов / А. Ахо, Дж.Хопкрофт, Дж.Ульман. – М.: МИР, 1979. – 329 с.

16. Сибуя, М. Алгоритмы обработки данных / М. Сибуя, Т. Яматото. – М.:МИР, 1986. – 473 с.

17. Лэнгсам, Й.Структуры данных для персональных ЭВМ / Й.

Лэнгсам, М. Огенстайн, А. Тененбаум. – М.: МИР, 1989. – 327 с.

18. Гулаков, В.К. Деревья: алгоритмы и программы / В.К. Гулаков. – М.: Машиностроение-1, 2005. – 206 с.

19. BRASS, P., Advanced Data Structures, New York: Cambridge University Press, 2008. – 474 p.