**Практическая 30**

**Тема: Бинарные деревья**

**Введение**

Бинарные (двоичные) деревья являются одним из самых востребованных вариантов данной структуры данных, так как широко используются в поисковых алгоритмах и для решения других вычислительных задач.

Будут рассмотрены основные характеристики бинарных деревьев и различные операции, выполняющиеся над ними.

В практической работе кроме теоретических аспектов: таких как классификации бинарных деревьев и стандартных приемов для манипулирования ими, приведены примеры практической реализации алгоритмов для работы с бинарными деревьями на языках Си

**Классификация бинарных деревьев**

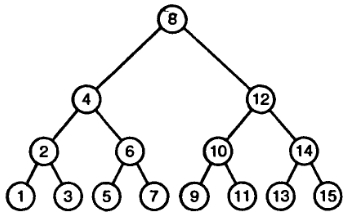
Перед тем, как перейти к обсуждению различных типов бинарных деревьев, стоит остановиться на общей классификации поисковых алгоритмов.

В случае с **линейным** поиском искомый элемент сравнивается с каждым элементом списка в ходе последовательной итерации. Скорость такого поиска зависит от размера списка: чем больше список, тем ниже скорость, т.е. скорость обратно пропорциональна размеру списка. Увеличить скорость такого поиска можно, если предварительно отсортировать список. В этом случае уже не потребуется продолжать поиск, когда искомый элемент еще не будет найден, а элементы в списке уже станут больше искомого.

Для отсортированного списка существует и более эффективный алгоритм. В начале поиска необходимо сравнить искомое число с элементом, который находится ПОСЕРЕДИНЕ уже отсортированного списка. Если серединный элемент оказывается больше искомого числа, значит искомый элемент находится в левой половине. В противном случае - справа. Затем выполняется сравнение с числом, которое находится посередине нужной половины. И так далее до тех пор, пока не будет найдено искомое число. Данный тип поиска называется **двоичным**, и он, очевидно, быстрее линейного. Необходимое количество делений списка, состоящего из **n** элементов пополам, называется логарифмом от **n** по основанию 2. Двоичный поиск является алгоритмом порядка O(log) по основанию 2.

Однако традиционные связные списки не очень подходят для двоичного поиска, и здесь на помощь приходит двоичное дерево, пример которого изображен на рисунке 1.

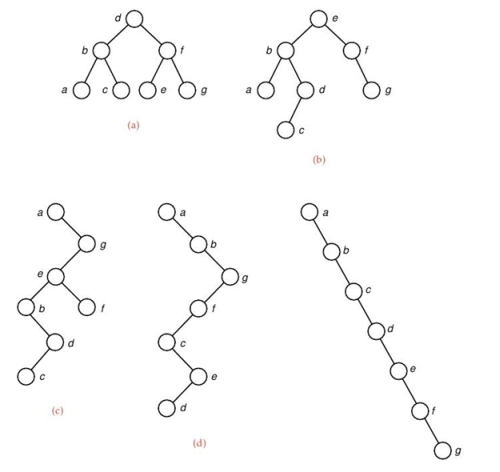
**Рисунок 1. Пример двоичного дерева**



В общем случае каждый узел в дереве может иметь неограниченное количество дочерних узлов. Отличие **бинарного** дерева от остальных типов деревьев в том, что каждый узел в нем может иметь не более двух дочерних узлов. Обычное двоичное дерево — это структура, состоящая из узлов, каждый из которых содержит некоторое значение, а также указатели на левое и правое поддеревья. Один или оба указателя на эти поддеревья могут иметь значение **NULL**. Двоичные деревья, как и связные списки, являются рекурсивными структурами.

На рисунке 2 показаны бинарные деревья, которые имеют одинаковый набор узлов, но различный порядок их следования. В последнем случае дерево вырождается в связный список.

**Рисунок 2. Различные реализации одного и того же бинарного дерева**



Существуют следующие разновидности бинарных деревьев:

1. полное бинарное дерево - каждый узел, за исключением листьев, имеет по 2 дочерних узла;
2. идеальное бинарное дерево — это полное бинарное дерево, в котором все листья находятся на одной высоте;
3. сбалансированное бинарное дерево — это бинарное дерево, в котором высота 2-х поддеревьев для каждого узла отличается не более чем на 1. Глубина такого дерева вычисляется как двоичный логарифм log(n), где n - общее число узлов;
4. вырожденное дерево - дерево, в котором каждый узел имеет всего один дочерний узел, фактически это связный список;
5. бинарное поисковое дерево (BST) - бинарное дерево, в котором для каждого узла выполняется условие: все узлы в левом поддереве меньше, и все узлы в правом поддереве больше данного узла.

**Вычисление путей**

Путь (англ. **path**) - это расстояние от корня дерева до какого либо узла. В расширенном бинарном дереве каждый путь оканчивается листом. Если число листьев обозначить как **S**, а число остальных узлов обозначить как **N**, то справедлива формула:

S = N +1

т.е. для расширенного дерева число листьев на единицу больше числа НЕлистьев (узлов, имеющих дочерние узлы).

Если путь от корневого узла до листа обозначить как внешний путь, а путь от корневого узла до НЕлиста обозначить как внутренний путь, тогда сумма всех внешних путей для дерева, изображенного на рисунке 3 равна:

E=3+3+2+3+4+4+3+3=25,

а сумма внутренних путей будет равна:

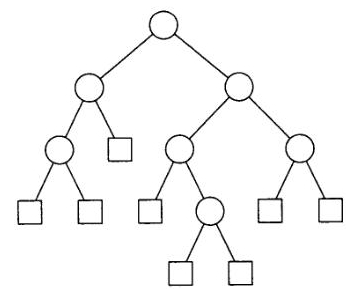
I=2+1+0+2+3+1+2=11.

и тогда будет справедлива формула:

E=I+2n

где n - число внутренних узлов (НЕлистьев).

**Рисунок 3. Пример расширенного дерева**



*Предположим,* имеется следующий набор листьев: 2,3,5,7,11,13,17,19,23,29,31,37,41

Тогда можно сформулировать следующие задачи:

* сконструировать бинарное дерево таким образом, чтобы сумма путей была минимальной, так как это сокращает время вычислений для различных алгоритмов.
* сконструировать полное расширенное бинарное дерево таким образом, чтобы сумма произведений путей от корневого узла до листьев на значение листового узла была минимальной.

**Давид Хаффман (David Huffman)** предложил алгоритм для решения этой проблемы, в котором на каждом шаге выбираются и складываются два наименьших листа, как показано в листинге 1, что приводит к дереву, изображенному на рисунке 4. Алгоритм получил название Хаффмана

**Листинг 1. Пример алгоритма Хаффмана**

2 3 5 7 11 13 17 19 23 29 31 37 41

5 5 7 11 13 17 19 23 29 31 37 41

10 7 11 13 17 19 23 29 31 37 41

17 24 17 19 23 29 31 37 41

24 34 19 23 29 31 37 41

24 34 42 29 31 37 41

42 53 65 37 41

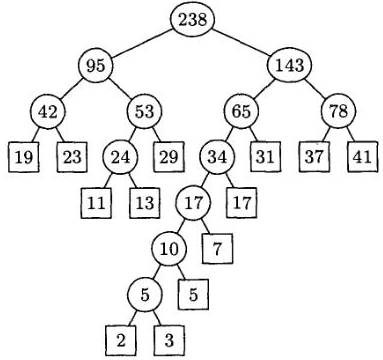
42 53 65 78

95 65 78

95 143

238

**Рисунок 4. Бинарное дерево, построенное по алгоритму Хаффмана**



**Коды Хаффмана**

Коды Хаффмана - это алгоритм, используемый для сжатия данных. Пусть имеется некое исходное текстовое сообщение, состоящее из 5 символов: **a, b, c, d, e**, и каждый символ имеет свою собственную частоту:

* **a** встречается 12 раз;
* **b** - 4 раза;
* **c** - 15 раз;
* **d** - 8 раз;
* **e** - 25 раз.

Нужно закодировать это сообщение с помощью 0 и 1 таким образом, чтобы размер результирующей строки оказался минимальным.

В таблице ниже приведен пример кодирования символов для данного сообщения:

|  |  |  |  |
| --- | --- | --- | --- |
| **символ** | **вероятность** | **код №1** | **код №2** |
| **a** | 0.12 | 000 | 000 |
| **b** | 0.40 | 001 | 11 |
| **c** | 0.15 | 010 | 01 |
| **d** | 0.08 | 011 | 001 |
| **e** | 0.25 | 100 | 10 |

Если зашифровать сообщение **bcd** с помощью кода №1, то получится - 001010011. С помощью кода №2 можно делать то же самое, но получившая строка будет короче - 1101001.

Теперь можно сформулировать саму проблему: необходимо подобрать такой алгоритм шифрования, при котором длина зашифрованной строки будет минимальной.

В первом варианте на каждый символ отводилось по 3 символа, а во втором варианте - уже 2.2, но можно сделать еще короче и получить коэффициент, равный **2.15**. Это делается с помощью алгоритма Хаффмана, в котором берутся 2 символа, имеющие наименьшую частоту, и объединяются, как два дочерних узла.

Алгоритм для строки, состоящей из 5 символов, реализуется следующим образом. На первом шаге объединяются два символа - **a** и **d**, как имеющие наименьшую частоту. На втором в качестве родителя добавляется символ **c**. На третьем шаге добавляется символ **e**, и в конце - символ **b**. В результате получается следующий код:

* b – 0;
* e – 10;
* c – 110;
* a – 1111;
* d – 1110.

**Высота бинарного дерева**

Для определения высоты дерева потребуется пройти от корня сначала по левому поддереву, потом по правому, сравнить две этих высоты и выбрать максимальное значение. И не забыть к получившемуся значению прибавить единицу (корневой элемент). В листинге 2 приведена рекурсивная функция для выполнения этой задачи.

**Листинг 2. Определение высоты дерева – рекурсивная реализация на языке Си**

int height(struct node \*p)

{

struct node \*temp=p;

int h1=0,h2=0;

if(p==NULL)return(0);

if(p->left){h1=height(p->left);}

if(p->right){h2=height(p->right);}

return(max(h1,h2)+1);

}

**«Зеркальное» отражение бинарного дерева**

Когда два дерева являются зеркальным отражением друг друга, то говорится, что они симметричны. Для получения «зеркальной» копии дерева используется алгоритм, приведенный в листинге 3. Сначала выполняется проверка на наличие у корня дерева дочерних узлов, и если эти узлы есть, то они меняются местами. Потом эти же действия рекурсивно повторяются для левого и правого дочерних узлов. Если существует только один дочерний узел, тогда можно переходить на один уровень ниже по дереву и продолжать.

**Листинг 3. Реверс дерева – рекурсивная реализация на языке Си**

void reverse(struct tree \*p)

{

struct tree \* temp;

if(p)

{

if(p->left && p->right)

{

temp = p->left;

p->left = p->right;

p->right = temp;

reverse(p->left);

reverse(p->right);

}

else if (p->left && !p->right) reverse(p->left);

else if (!p->left && p->right) reverse(p->right);

}

}

**Поиск узла в бинарном дереве**

Для поиска узла в бинарном дереве по содержащимся в нем значениям можно использовать функцию **lookup**, приведенную в листинге 4:

**Листинг 4. Поиск узла в дереве – рекурсивная реализация на языке Си**

int lookup(struct node\* node, int target)

{

if (node == NULL)

{

return(0);

}

else

{

if (target == node->data) return(1);

else

{

if (target < node->data) return(lookup(node->left, target));

else return(lookup(node->right, target));

}

}

}

**Ширина бинарного дерева**

Под шириной дерева понимается максимальное количество узлов, которые расположены на одной высоте. Чтобы определить ширину дерева, достаточно просто добавить счетчик в уже рассмотренный алгоритм для определения высоты дерева.

**Листинг 5. Определение ширины дерева – рекурсивная реализация на языке Си**

int getMaxWidth(struct node\* root)

{

int maxWdth = 0;

int i;

int width = 0 ;

int h = height(root);

for( i = 1; i< h; i++)

{

width = getWidth(root, i);

if(width > maxWdth)

maxWdth = width;

}

return maxWdth;

}

int getWidth(struct node\* root,int level)

{

if (!root) return 0;

if (level == 1) return 1;

else if (level > 1)

return getWidth(root->left, level-1) + getWidth(root->right, level-1);

getWidth(root->right, level-1);

}

**Количество узлов в бинарном дереве**

Вычислить количество узлов в бинарном дереве также можно с помощью рекурсии.

**Листинг 6. Вычисление количества узлов в дереве – рекурсивная реализация на языке Си**

int size(struct node\* node)

{

if (node==NULL)

{

return(0);

}

else

{

return(size(node->left) + 1 + size(node->right));

}

}

**Сравнение бинарных деревьев**

Чтобы определить, совпадают ли два разных дерева, можно использовать алгоритм из листинга 7.

**Листинг 7. Сравнение бинарных деревьев – рекурсивная реализация на языке Си**

int sameTree(struct node\* a, struct node\* b)

{

if (a==NULL && b==NULL) return(true);

else if (a!=NULL && b!=NULL)

{

return(

a->data == b->data &&

sameTree(a->left, b->left) &&

sameTree(a->right, b->right)

);

}

else return(false);

}

**Заключение**

1. Написать программу на языке Java, использующая функции, разработанные выше, для выполнения операций над бинарным деревом.
2. Реализовать алгоритм Хаффмана

Резюме. В дпнной практической работе была выполнена классификация бинарных деревьев и рассмотрены алгоритмы для выполнения важнейших операций: определение высоты и ширины дерева, отражение дерева и поиск в нем определенного элемента. Также был рассмотрен алгоритм Хаффмана, который может использоваться для построения расширенных деревьев и, как следствие, для кодирования информации.