Урок 6

Деревья

Построение сбалансированного дерева. Двоичные деревья поиска. Хэш-функция. MD5. Хэш-таблицы

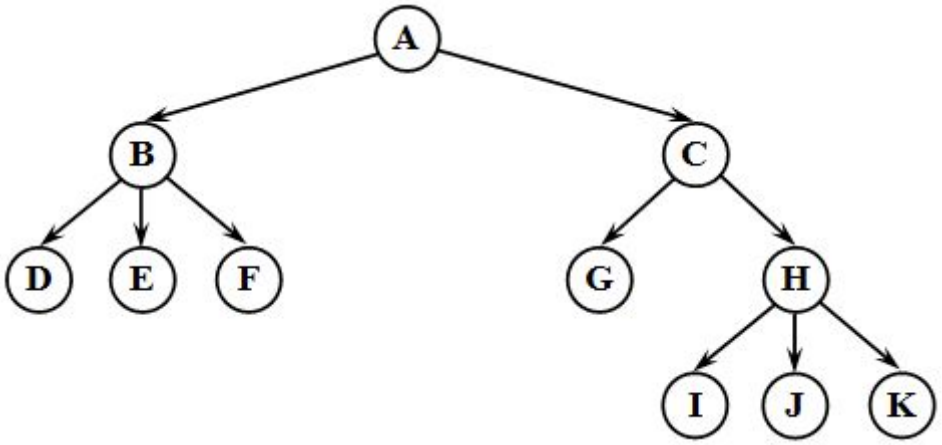
Структура данных дерево

Дерево — это структура, отражающая иерархию (отношения подчиненности, многоуровневые связи).

Дерево состоит из **узлов** и связей между ними (они называются **дугами**). Самый первый узел, расположенный на верхнем уровне — это **корень дерева**. Конечные узлы, из которых не выходит ни одна дуга, называются **листьями**. Все остальные узлы, кроме корня и листьев, — промежуточные. Из двух связанных узлов тот, который находится на более высоком уровне, называется **родителем**, а другой — **сыном**. Корень — это единственный узел, у которого нет родителя; у листьев нет сыновей.

Используются также понятия **«предок»** и **«потомок»**. Потомок некоторого узла — узел, в который можно перейти по стрелкам от узла-предка. Соответственно, предок какого-то узла — это узел, из которого можно перейти по стрелкам в данный узел.

В рисунке родитель узла F — это узел B, а предки узла F — это B и A, для которых узел F — потомок. Потомками узла A (корня дерева) являются все остальные узлы.



Высота дерева — это наибольшее расстояние (количество дуг) от корня до листа. Высота дерева на рисунке равна 3.

Формально дерево можно определить следующим образом:

1. Пустая структура — это дерево.
2. Дерево — это корень и несколько связанных с ним отдельных (не связанных между собой) деревьев.

Здесь множество объектов (дерево) определяется через само это множество на основе простого базового случая — пустого дерева. Такой прием называется рекурсией. Согласно этому определению, дерево — это рекурсивная структура данных. Поэтому можно ожидать, что при работе с деревьями будут полезны рекурсивные алгоритмы.

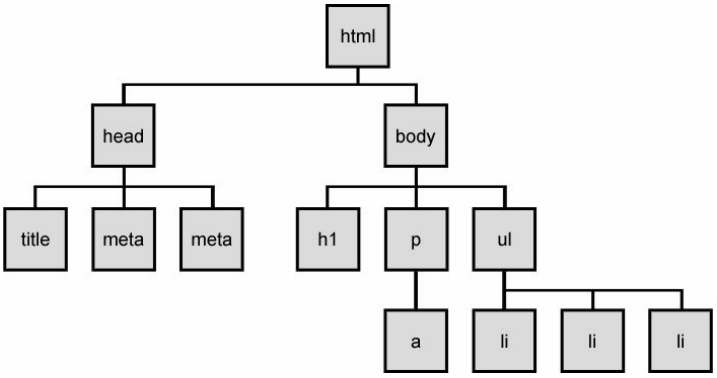
Деревья широко применяются в следующих задачах:

1. Поиск в большом массиве неменяющихся данных.

2. Сортировка данных.
3. Вычисление арифметических выражений.
4. Оптимальное кодирование данных (метод сжатия Хаффмана).

В виде дерева удобно хранить структуру организации компании или каталог частей, из которых состоит, скажем, автомобиль.

Можно особо отметить DOM (Document Object Model) в HTML. При считывании HTML-страницы браузер, анализируя HTML, выстраивает иерархию объектов в виде дерева.



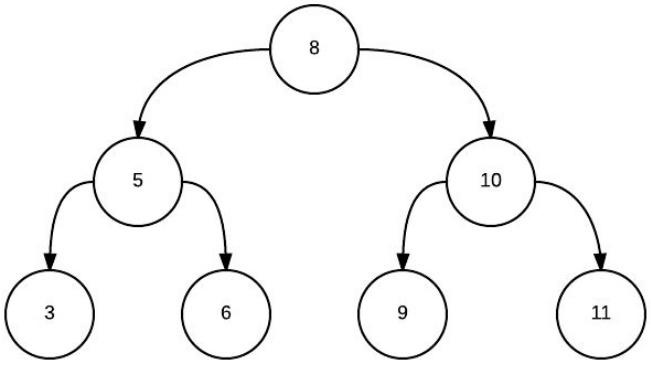
Чаще всего в информатике используются двоичные (или бинарные) деревья, т.е. такие, в которых каждый узел имеет не более двух сыновей. Их также можно определить рекурсивно.

Двоичное дерево:

1. Пустая структура — это двоичное дерево.
2. Двоичное дерево — это корень и два связанных с ним отдельных двоичных дерева (левое и правое поддеревья).

Деревья поиска

Чтобы найти заданный элемент в неупорядоченном массиве из N элементов, может понадобиться N сравнений. Теперь предположим, что элементы массива организованы в виде специальным образом построенного дерева, например, как показано на рисунке:



Значения, связанные с узлами дерева, по которым выполняется поиск, называются ключами этих узлов (кроме ключа узел может содержать множество других данных). Перечислим важные свойства дерева, показанного на рисунке:

1. Слева от каждого узла находятся узлы, ключи которых меньше или равны ключу данного узла.
2. Справа от каждого узла находятся узлы, ключи которых больше данного узла.

Дерево, обладающее такими свойствами, называется **двоичным деревом поиска.**

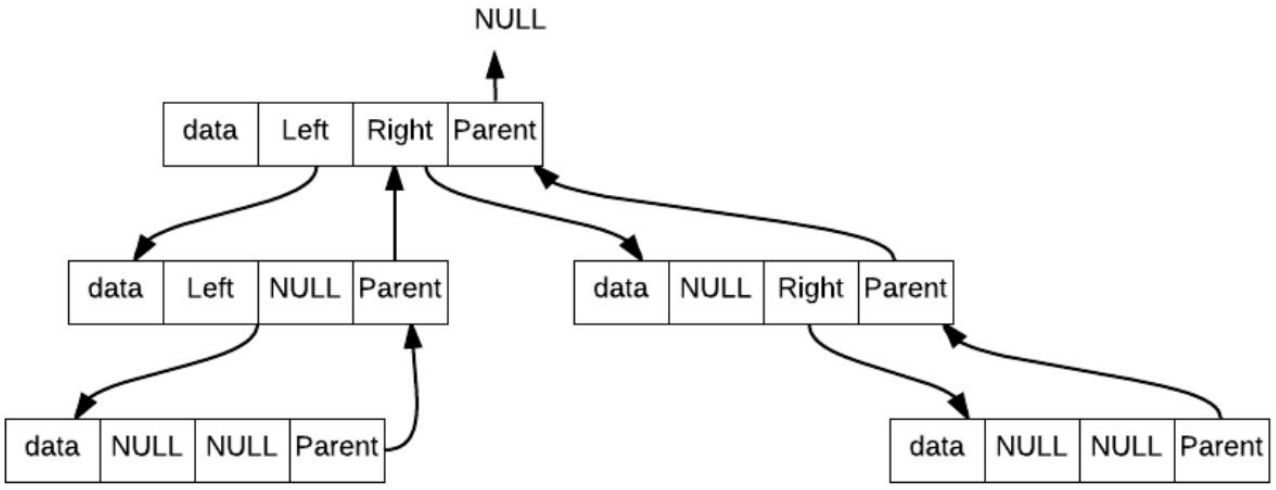
Например, пусть нужно найти узел, ключ которого равен 6. Начинаем поиск по дереву с корня. Ключ корня — 8 (больше заданного), поэтому дальше нужно искать только в левом поддереве и т.д. Если при линейном поиске в массиве за одно сравнение отсекается 1 элемент, здесь — сразу примерно половина оставшихся. Количество операций сравнения в этом случае пропорционально $\log_2 N$, т.е. алгоритм имеет асимптотическую сложность $O(\log_2 N)$. Конечно, нужно учитывать, что предварительно дерево должно быть построено. Поэтому такой алгоритм выгодно применять в тех случаях, когда данные меняются редко, а поиск выполняется часто (например, в базах данных).

Индексация базы данных как раз и есть упорядочивание элементов с использования структур типа двоичного дерева. В дальнейшем это позволяет сократить время поиска в базе данных.

Использование динамических структур данных

Поскольку двоичное дерево — это нелинейная структура данных, мы будем использовать связанные узлы. Каждый такой узел — структура, содержащая три области: область данных, ссылка на левое поддерево (указатель) и ссылка на правое поддерево (второй указатель). Также иногда добавляют указатель на предыдущий узел (Parent). У листьев нет сыновей, в этом случае в указатели будем записывать значение NULL (нулевой указатель).

Пример дерева, состоящего из пяти таких узлов:



Структура, описывающая узел дерева:

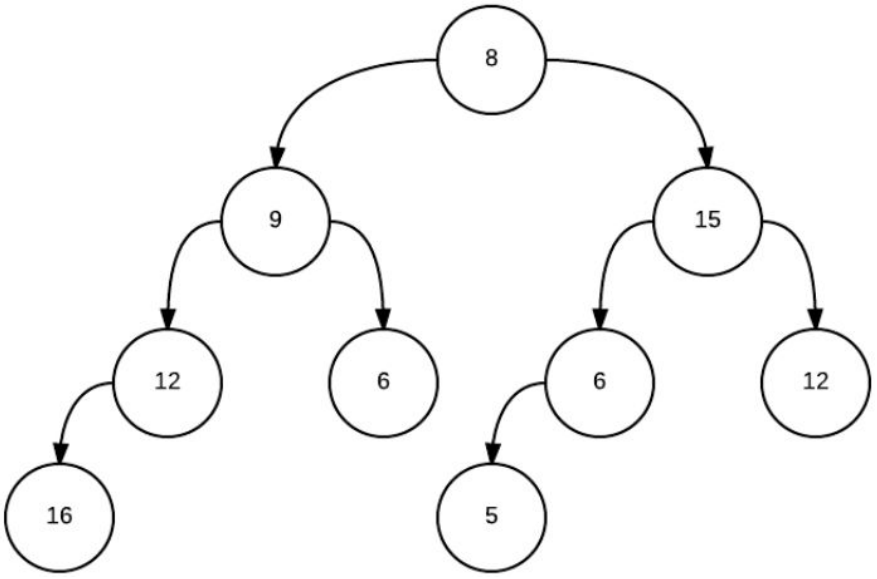
```
typedef struct Node {
    T data;
    struct Node *left;
    struct Node *right;
    struct Node *parent;
} Node;
```

Скобочная запись дерева

Так как изобразить дерево графически — довольно сложная задача, мы будем использовать скобочную запись. Например, дерево, представленное на рисунке ниже, в скобочной записи будет выглядеть следующим образом:

8(9(12(16,NULL),6),15(6(5,NULL),12))

NULL означает отсутствие узла.

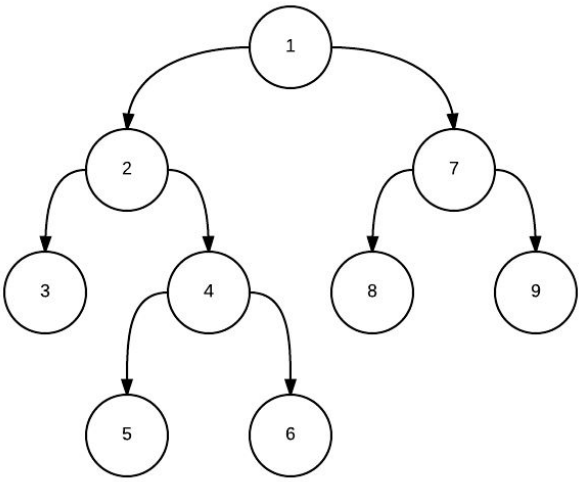



```
// Распечатка двоичного дерева в виде скобочной записи
void printTree(Node *root) {
    if (root)
    {
        printf("%d", root->data);
        if (root->left || root->right)
        {
            printf("(");

            if (root->left)
                printTree(root->left);
            else
                printf("NULL");

            printf(",");

            if (root->right)
                printTree(root->right);
            else
                printf("NULL");

            printf(")");
        }
    }
}
```

Рекурсивный обход двоичного дерева

Существует несколько способов обхода дерева:

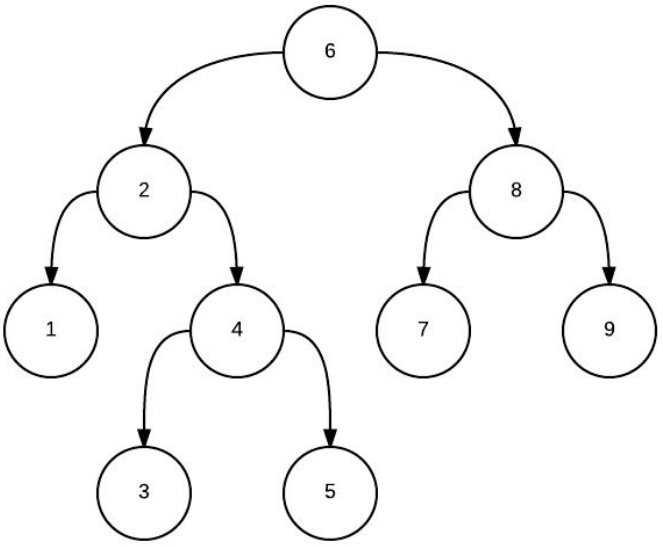
1. **КЛП — «корень–левый–правый»** (обход в прямом порядке, pre-order):
    a. Посетить корень.
    b. Обойти левое поддерево.
    c. Обойти правое поддерево.



Рекурсивное решение полностью соответствует описанию алгоритма:

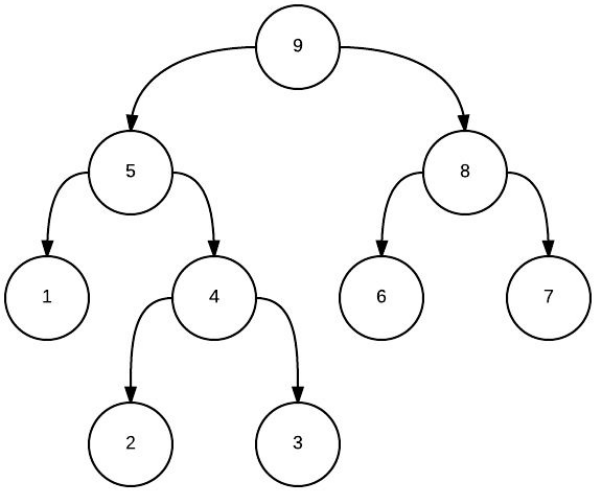
```
void preOrderTravers(Node* root) {
    if (root) {
        printf("%d ", root->data);
        preOrderTravers(root->left);
        preOrderTravers(root->right);
    }
}
```

2. **ЛКП — «левый–корень–правый»** (симметричный обход, in-order):
    a. Обойти левое поддерево.
    b. Посетить корень.
    c. Обойти правое поддерево.




```
СимметричныйОбход(Узел*)
{
   если узел не пуст, то
   {
      СимметричныйОбход(Узел->Левый)
      вывести Узел->data
      СимметричныйОбход(Узел->Правый)
   }
}
```

3. **ЛПК — «левый–правый–корень»** (обход в обратном порядке, post-order):
    a. Обойти левое поддерево.
    b. Обойти правое поддерево.
    c. Посетить корень.



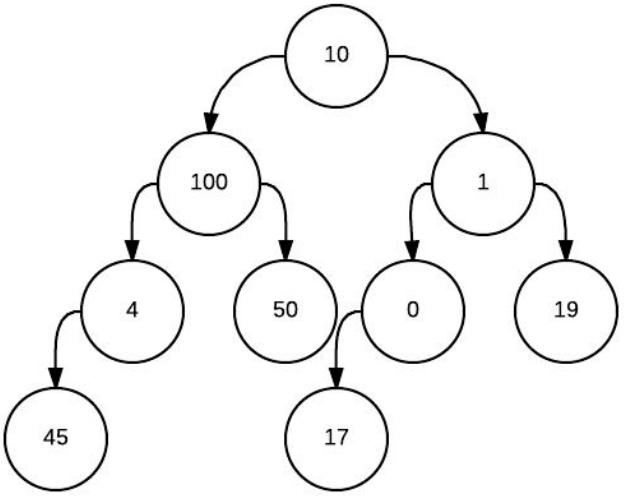
```
ОбратныйОбход(Узел*)
{
   если узел не пуст, то
   {
      ОбратныйОбход(Узел->Левый)
      ОбратныйОбход(Узел->Правый)
      вывести Узел->data
   }
}
```

Как видим, это рекурсивные алгоритмы. Они должны заканчиваться без повторного вызова, когда текущий корень — пустое дерево.

Построение сбалансированного дерева

Предположим, что нужно построить дерево, значениями в узлах которого являются n чисел, считываемых из входного файла. Чтобы сделать задачу интересней, будем строить дерево с n узлами, имеющее минимальную высоту. Чтобы получить минимальную высоту при заданном числе узлов, нужно размещать максимально возможное число узлов на всех уровнях, кроме самого нижнего. Очевидно, этого можно достичь, распределяя новые узлы поровну слева и справа от каждого узла.

Пример сбалансированного дерева на рисунке ниже:



Правило равномерного распределения при известном числе узлов n лучше всего сформулировать рекурсивно:

1. Использовать один узел в качестве корня.
2. Построить таким образом левое поддерево с числом узлов nl=n div 2.
3. Построить таким образом правое поддерево с числом узлов nr=n-nl-1.

Это правило реализуется рекурсивной процедурой, которая читает входной файл и строит идеально сбалансированное дерево. Дерево является идеально сбалансированным, если для каждого узла число узлов в левом и правом поддеревьях отличается не больше, чем на 1.

```c
#include <stdio.h>
#include <malloc.h>
#include <stdlib.h>

typedef int T;
FILE* file;

typedef struct Node {
    T data;
    struct Node *left;
    struct Node *right;
} Node;

// Построить идеально сбалансированное дерево с n узлами
Node* Tree(int n)
{
    Node* newNode;
    int x, nl, nr;
    if (n == 0)
        newNode=NULL;
    else
    {
        fscanf(file, "%d", &x);
        nl = n / 2;
        nr = n – nl - 1;
        newNode = (Node*) malloc(sizeof(Node));
        newNode->data = x;
        newNode->left = Tree(nl);
        newNode->right = Tree(nr);
    }
    return newNode;
}

// Распечатка двоичного дерева в виде скобочной записи
void printTree(Node *root) {
    if (root)
    {
        printf("%d", root->data);
        if (root->left || root->right)
        {
            printf("(");
            if (root->left)
                printTree(root->left);
            else
                printf("NULL");
            printf(",");

            if (root->right)
                printTree(root->right);
            else
                printf("NULL");
            printf(")");
        }
    }
}
```

```
int main()
{
    Node* tree = NULL;
    File = fopen("d:\\temp\\data.txt","r");
    if (file == NULL)
    {
        puts("Can't open file!");
        return 1;
    }
    int count;
    fscanf(file, "%d", &count); // Считываем количество записей
    tree = Tree(count);
    fclose(file);
    printTree(tree);
    return 0;
}
```

Двоичное дерево поиска

Дан файл с числами произвольной длины. Считать файл с диска и построить по данным файла двоичное дерево поиска. Реализовать функции считывания данных из файла в двоичное дерево и поиска заданного числа в двоичном дереве.

Создание двоичного дерева поиска:

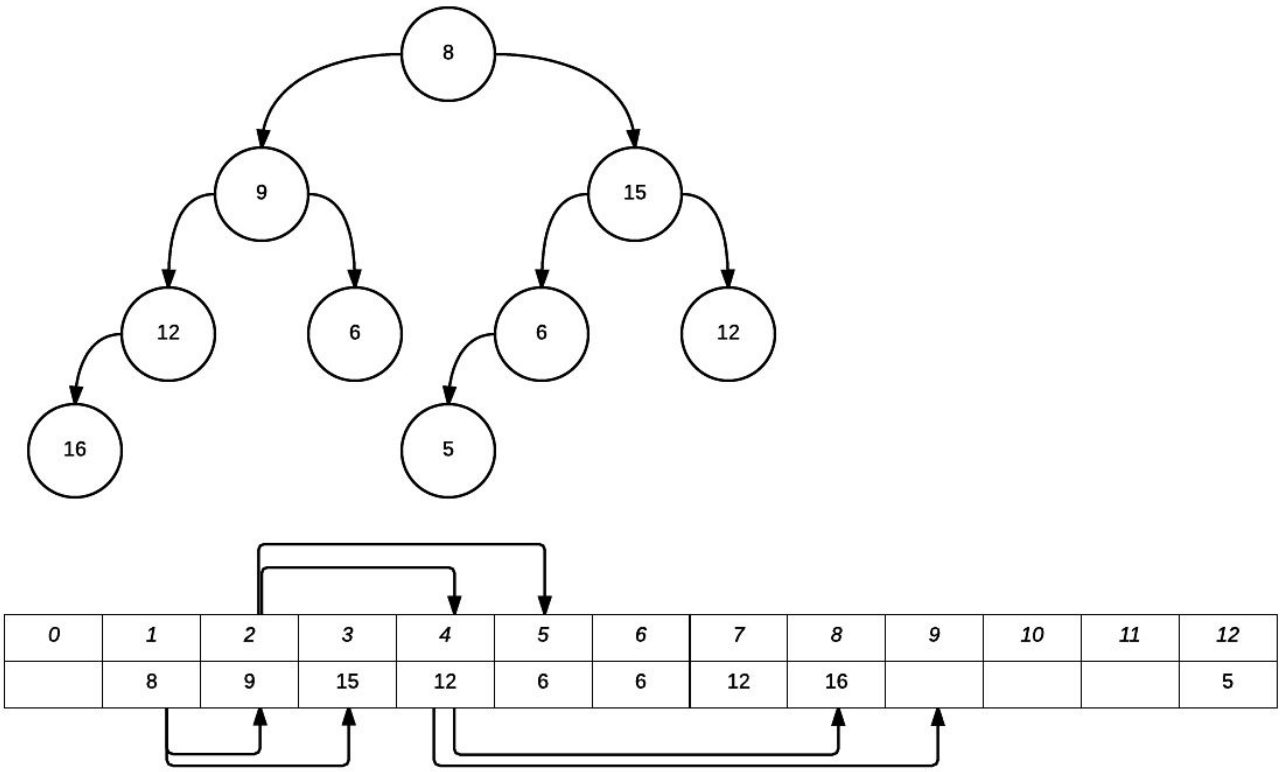
```
#include <stdio.h>
#include <malloc.h>
#include <stdlib.h>

typedef int T;
typedef struct Node {
    T data;
    struct Node *left;
    struct Node *right;
    struct Node *parent;
} Node;
// Распечатка двоичного дерева в виде скобочной записи
void printTree(Node *root) {
    if (root)
    {
        printf("%d",root->data);
        if (root->left || root->right)
        {
        printf("(");

        if (root->left)
            printTree(root->left);
        else
            printf("NULL");
        printf(",");

        if (root->right)
            printTree(root->right);
        else
            printf("NULL");
```

```
        printf(")");
        }
    }
}

// Создание нового узла
Node* getFreeNode(T value, Node *parent) {
    Node* tmp = (Node*) malloc(sizeof(Node));
    tmp->left = tmp->right = NULL;
    tmp->data = value;
    tmp->parent = parent;
    return tmp;
}

// Вставка узла
void insert(Node **head, int value) {
    Node *tmp = NULL;
    if (*head == NULL)
    {
        *head = getFreeNode(value, NULL);
        return;
    }

    tmp = *head;
    while (tmp)
    {
        if (value> tmp->data)
        {
            if (tmp->right)
            {
                tmp = tmp->right;
                continue;
            }
            else
            {
                tmp->right = getFreeNode(value, tmp);
                return;
            }
        }
        else if (value< tmp->data)
        {
            if (tmp->left)
            {
                tmp = tmp->left;
                continue;
            }
            else
            {
                tmp->left = getFreeNode(value, tmp);
                return;
            }
        }
        else
        {
            exit(2);                        // дерево построено неправильно
        }
    }
}
```

```
void preOrderTravers(Node* root) {
    if (root) {
        printf("%d ", root->data);
        preOrderTravers(root->left);
        preOrderTravers(root->right);
    }
}


int main()
{
    Node *Tree = NULL;
    FILE* file = fopen("data.txt", "r");
    if (file == NULL)
    {
        puts("Can't open file!");
        exit(1);
    }
    int count;
    fscanf(file, "%d", &count);          // Считываем количество записей
    int i;
    for(i = 0; i < count; i++)
    {
        int value;
        fscanf(file, "%d", &value);
        insert(&Tree, value);
    }
    fclose(file);
    printTree(Tree);
    printf("\nPreOrderTravers:");
    preOrderTravers(Tree);
    return 0;
}
```

Хранение двоичного дерева в массиве

Двоичные деревья можно хранить в массиве почти так же, как и списки. Вопрос в том, как сохранить структуру (взаимосвязь узлов). Если нумерация элементов начинается с 1, то сыновья элемента a[i] — это a[2*i] и a[2*i+1]. Каждому числу 1, 2, 3, 4 будет соответствовать пара 1 — (2,3), 2 — (4,5), 3 — (6,7) и т.д. На рисунке ниже показано, как можно сохранить часть двоичного дерева в массиве. Обратите внимание, что некоторые элементы остались пустыми, это значит, что их родитель — лист дерева.



Хэш-функция

Часто при скачивании торрентов в их описании можно увидеть что-то наподобие «A9990CD6D22117C1D4E3C5E9CF2FC5AA20305143», нередко с припиской «MD5». Это хэш-код — результат, который выдаёт хэш-функция после обработки входящих данных. Расшифровать хэш MD5 очень сложно, можно сказать, практически невозможно. Данная функция предназначена для преобразования входящих данных сколь угодно большого размера в результат фиксированной длины. Сам процесс такого преобразования называется хэшированием, а результат — хэшем или хэш-кодом.

Порой ещё используют слова «отпечаток» или «дайджест сообщения», но на практике они встречаются намного реже. Существует масса различных алгоритмов того, как можно превратить любой массив данных в некую последовательность символов определённой длины. Наибольшее распространение получил алгоритм под названием md5, который был разработан ещё в 1991 году. Несмотря на то, что на сегодняшний день md5 является несколько устаревшим и к использованию не рекомендуется, он до сих пор всё ещё в ходу, и часто вместо слова «хэш-код» на сайтах просто пишут md5 и указывают сам код.

MD5 (Message Digest)

Для функции, описанной выше, если известен результат, можно легко найти параметр, для которого будет такой же результат. А вот для функции MD5 это сделать не так-то просто. Т.е. если у нас есть только результат функции MD5, то мы не сможем найти параметр, для которого функция выдаст этот же результат (речь даже не идёт про однозначное восстановление параметра). Например, MD5 используют для хранения паролей. Приведем пример, когда хранение паролей в открытом виде опасно. Предположим, у вас есть сайт и база данных с логинами и паролями для доступа к нему. У вас возникла необходимость сделать локальную копию этого сайта, и нужно перенести логины и

пароли пользователей. Можно просто перенести базу данных с логинами и паролями, но, получается, будет две базы, что снижает безопасность. А можно посчитать для каждого пароля хэш-значение и перенести базу с ними.

Пароли хранятся в зашифрованном виде (при помощи MD5). После того, как пользователь введёт свой пароль, от него вычисляется хэш-функция MD5. Результат сравнивается со значением, хранящимся в базе. Если значения равны, то пароль верен.

Также MD5 можно использовать в качестве контрольной суммы. Предположим, необходимо куда-то скопировать файл или скачать из Интернета. Причём нет никаких гарантий, что файл будет доставлен без повреждений. Перед отправкой можно посчитать MD5 от содержимого файла и передать результат вместе с ним. Затем посчитать MD5 от принятого файла и сравнить два результата. Если результаты различные, то это означает, что файл или результат были испорчены при передаче.

Хэш-функции и хэш-таблицы

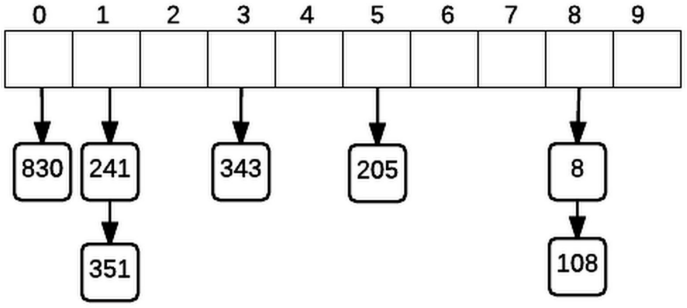
Хэш-таблицы и хэш-функции позволяют быстро искать значения, поскольку не хранят их в специальном отсортированном списке, а организуют особым образом информацию в хэш-таблицу, и с помощью хэш-функций из этой таблицы информация извлекается.

Предположим, у вас есть небольшая компания из 50 сотрудников с идентификационными номерами (ID) длинной в 4 цифры. Для этого можно организовать массив из 100 элементов, в котором элемент массива будет соответствовать ID % 100. Например, сотрудник с ID 3468 будет размещен в позиции 68, а с ID 4677 — в позиции 77. Чтобы найти определенного работника, достаточно воспользоваться хэш-функцией **ID % 100** и выбрать элемент со значением возвращаемой функцией.

В реальности, если сотрудников окажется довольно много, то среди них найдутся несколько, у которых ID будет соответствовать одному и тому же значению. Скажем, ID 4465 и 3165 должны будут занять в таблице позицию 65. Этот случай называется коллизией, и при работе с хэш-таблицами есть алгоритмы обработки коллизий.

Прямое связывание (хэширование с цепочками)

В хэш-таблицах с прямым связыванием значения ключей хранятся в специальных наборах записей, называемых блоками. Каждый из них является вершиной связного списка, в котором находятся привязанные к блоку элементы.

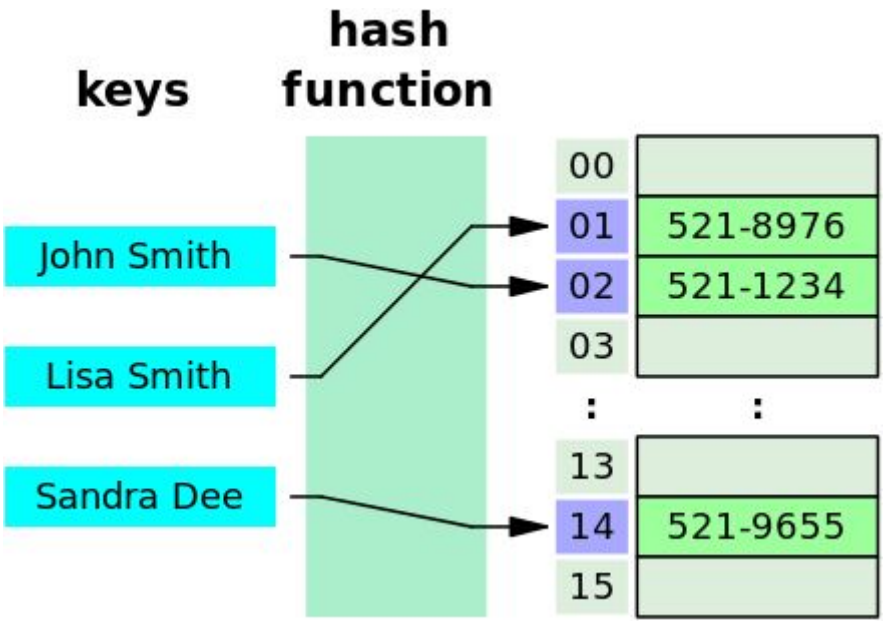


Обычно блоки расположены в массиве таким образом, что можно использовать простую функцию хэширования для определения их ключа. Например, если у вас N блоков, а ключи численные, свяжите ключ K с блоком под номером K % N. Для рисунка K % 10.

В практике пример, реализующий прямое связывание на языке C.

Хэширование с открытой адресацией

В случае метода открытой адресации (или по-другому — закрытого хэширования) все элементы хранятся непосредственно в хэш-таблице без использования связанных списков.



В отличие от хэширования с цепочками, при использовании метода открытой адресации может возникнуть ситуация, когда хэш-таблица окажется полностью заполненной, так что будет невозможно добавлять в неё новые элементы. Решением при этом может быть динамическое увеличение размера хэш-таблицы с одновременной её перестройкой.

Практика

Хэш-таблицы

Написать программу создания хэш-таблицы, используя структуру «односвязный список». Реализовать функции вставки, удаления и поиска узла.

```
#include <stdio.h>
#include <stdlib.h>

/* Установите вместо int нужный вам тип данных */
typedef int T;                  /* тип сохраненных данных */
typedef int hashTableIndex;     /* индекс в хэш-таблице */
#define compEQ(a, b) (a == b)   // Определили директиву сравнения двух значений

struct Node_ {
        T data;                 /* данные сохраненные в узле */
        struct Node_ *next;     /* следующий узел */
};
typedef struct Node_ Node;

Node **hashTable;

int hashTableSize;
```

```
hashTableIndex hash(T data)
{
	/**********************************
	*хэш-функция применяемая к данным *
	**********************************/
	return (data % hashTableSize);
}

Node *insertNode(T data)
{
	Node *p, *p0;
	hashTableIndex bucket;
	/************************************************
	*распределим узел для данных и вставим в таблицу*
	************************************************/
	/* Вставка узла в начало таблицы */
	bucket = hash(data);	// рассчитываем номер блока
	p = (Node*)malloc(sizeof(Node));
	if (p == 0) {
		// В стандартный поток ошибок выводим сообщение о нехватке памяти
		fprintf(stderr, "out of memory (insertNode)\n");
		exit(1);
	}
	// Запоминаем текущее значение указателя найденного блока
	p0 = hashTable[bucket];
	// В найденный блок записываем новый элемент
	hashTable[bucket] = p;
	// Связываем новый элемент со старым
	p->next = p0;
	// Записываем данные в новый элемент
	p->data = data;
	return p;
}

/*Удаление узла*/
void deleteNode(T data) {
	Node *p0, *p;
	hashTableIndex bucket;
	/*****************************************
	* удаляем узел содержащие данные из таблицы  *
	*****************************************/
	/* находим узел*/
	p0 = 0;
	bucket = hash(data);
	p = hashTable[bucket];
	while (p && !compEQ(p->data, data)) {
		p0 = p;
		p = p->next;
	}
	if (!p)
		return;
	/* p найденный узел для удаления, удаляем его из списка */
	if (p0)
		// не первый, p0 указывает на предыдущий
		p0->next = p->next;
	else
		// первый в цепочке
```

```
                hashTable[bucket] = p->next;
        free(p);
}
Node *findNode(T data) {
        Node *p;
        /**************************************
        *нахождение узла, содержащего данные *
        **************************************/
        p = hashTable[hash(data)];
        while (p && !compEQ(p->data, data))
                p = p->next;
        return p;
}
void printTable(int size) {
        Node *p;
        for (int i = 0; i < size; i++)
        {
                p = hashTable[i];
                while (p)
                {
                        printf("%5d", p->data);
                        p = p->next;
                }
                printf("\n");
        }
}
int main(int argc, char **argv) {
        int i, *a, maxnum, random;
        if (argc < 2) {
                printf("incorrect command line\ncommand line maxnum hashTableSize
[random]\nSample:\n");
                printf("hashTable 2000 100\n");
                printf("Create 2000 records, hastTable=100, fill sequense numbers\n");
                printf("or 4000 200 r\n");
                printf("Create 4000 records, hastTable=200, fill random numbers\n");
                exit(0);        // Выход без ошибки
        }
        maxnum = atoi(argv[1]);
        hashTableSize = atoi(argv[2]);
        random = argc > 3;
        if ((a = (int*)malloc(maxnum * sizeof(*a))) == 0)
        {
                fprintf(stderr, "out of memory (a)\n");
                exit(1);
        }
        if ((hashTable = (Node**)calloc(hashTableSize , sizeof(Node *))) == 0)
        {
                fprintf(stderr, "out of memory (hashTable)\n");
                exit(1);
        }
        if (random)
        { /* random */
          // заполняем "a" случайными числами
                for (i = 0; i < maxnum; i++)
                        a[i] = rand()%100;
                printf("ran ht, %d items, %d hashTable\n", maxnum, hashTableSize);
        }
        else
```

```
        {                       // заполняем последовательными данными
                for (i = 0; i<maxnum; i++)
                        a[i] = i;
                printf("seq ht, %d items, %d hashTable\n", maxnum, hashTableSize);
        }
        for (i = 0; i < maxnum; i++)
                insertNode(a[i]);

        printTable(hashTableSize);
        for (i = maxnum - 1; i >= 0; i--)
                findNode(a[i]);

        for (i = maxnum - 1; i >= 0; i--)
                deleteNode(a[i]);

        getchar();
        return 0;
}
```

Домашнее задание

1. Реализовать простейшую хэш-функцию. На вход функции подается строка, на выходе сумма кодов символов.
2. Переписать программу, реализующее двоичное дерево поиска.
    а) Добавить в него обход дерева различными способами;
    б) Реализовать поиск в двоичном дереве поиска;
    в) *Добавить в программу обработку командной строки с помощью которой можно указывать из какого файла считывать данные, каким образом обходить дерево.
3. *Разработать базу данных студентов из двух полей “Имя”, “Возраст”, “Табельный номер” в которой использовать все знания, полученные на уроках.
Считайте данные в двоичное дерево поиска. Реализуйте поиск по какому-нибудь полю базы данных (возраст, вес)

Дополнительные материалы

1. Двоичное дерево поиска
2. Хэширование

Используемая литература

Для подготовки данного методического пособия были использованы следующие ресурсы:

1. Род Стивенс, Алгоритмы. Теория и практическое применение. Издательство «Э», 2016.
2. Никлаус Вирт, Алгоритмы и структуры данных. Новая версия для Оберона, ДМК, Москва, 2010.
3. Пол Дейтел, Харви Дейтел. C для программистов. C введением в C11, ДМК, Москва, 2014.