查找算法

2021年7月28日　9:54

查找算法

查找是在大量的信息中寻找一个特定的信息元素，在计算机应用中，查找是常用的基本运算，例如编译程序中符号表的查找。本文简单概括性的介绍了常见的七种查找算法，说是七种，其实二分查找、插值查找以及斐波那契查找都可以归为一类——插值查找。插值查找和斐波那契查找是在二分查找的基础上的优化查找算法。

定义

根据给定的某个值，在查找表中确定一个其关键字等于给定值的数据元素（或记录）。

分类

（1）静态查找和动态查找

在查找过程中不修改查找表的长度和表中内容的方法称作静态查找，反之称作动态查找。

（2）无序查找和有序查找

无序查找：被查找数列有序无序均可；

有序查找：被查找数列必须为有序数列。

评价指标

关键字的平均比较次数，也称平均搜索长度ASL（Average Search Length）：需要=和指定key进行比较的关键字的个数的期望值，称为查找算法在查找成功时的平均查找长度。

对于含有n个数据元素的查找表，查找成功的平均查找长度为：ASL=Pi * Ci的和。

Pi：查找表中第i个数据元素的概率。

Ci：找到第i个数据元素时已经比较过的次数。

1、顺序查找

1.1说明

顺序查找适合于存储结构为顺序存储或链接存储的线性表。

1.2基本思想

顺序查找也称为线形查找，属于无序查找算法。从数据结构线形表的一端开始，顺序扫描，依次将扫描到的结点关键字与给定值k相比较，若相等则表示查找成功；若扫描结束仍没有找到关键字等于k的结点，表示查找失败。

1.3复杂度分析

查找成功时的平均查找长度为：（假设每个数据元素的概率相等） ASL = 1/n(1+2+3+…+n) = (n+1)/2 ; 当查找不成功时，需要n+1次比较，时间复杂度为O(n); 所以，顺序查找的时间复杂度为O(n)。

1.4算法实现

```
#include<iostream>
using namespace std;
int SequenceSearch(int* arr, int size, int n) {
	if (!arr || size < 1) {
		return -1;
	}
	for (int i = 0; i != size; ++i) {
		if (n = arr[i]) {
			return i;
		}
	}
	return -1;
}
int main() {
	int arr[10] = { 0, 10, 44, 22, 25, 9, 5, 15, 40, 10 };
	int index = SequenceSearch(arr, 10, 10);
	cout << index << endl;
}
//顺序查找
void saar(int *arr, int sz, int key)    //sz: 数组大小，key: 关键字
{
    int ind = 0;    //下标索引
    bool ex = false;    //是否存在匹配
    while (ind != sz)
    {
        if (key == arr[ind])
```

```
        {
            cout<<"数组中第 "<<ind+1<<" 个数据元素与关键字 \""<<key<<"\" 相匹配"<<endl;
            ex = true;
            ++ind;
            continue;
        }
        ++ind;
    }
    if (false == ex)
    {
        cout<<"数组中不存在数据元素与关键字 \""<<key<<"\" 相匹配"<<endl;
    }
}
```

2、二分查找

2.1说明

元素必须是有序的，如果是无序的则要先进行排序操作。

2.2基本思想

也称为是折半查找，属于有序查找算法。用给定值k先与中间结点的关键字比较，中间结点把线形表分成两个子表，若相等则查找成功；若不相等，再根据k与该中间结点关键字的比较结果确定下一步查找哪个子表，这样递归进行，直到查找到或查找结束发现表中没有这样的结点。

2.3复杂度分析

复杂度分析：最坏情况下，关键词比较次数为$\log_2(n+1)$，且期望时间复杂度为$O(\log_2 n)$。

注：折半查找的前提条件是需要有序表顺序存储，对于静态查找表，一次排序后不再变化，折半查找能得到不错的效率。但对于需要频繁执行插入或删除操作的数据集来说，维护有序的排序会带来不小的工作量，那就不建议使用。——《大话数据结构》

2.4算法实现

```
#include<iostream>
using namespace std;
//非递归
int BinarySearch(int* arr, int size, int n) {
	if (!arr || size < 1) {
		return -1;
	}
	int left, mid, right;
	left = 0;
	right = size - 1;
	while (left <= right) {
		mid = (right + left) / 2;
		if (n == arr[mid]) {
			return mid;
		}
		else if (n > arr[mid]) {
			left = mid + 1;
		}
		else {
			right = mid - 1;
		}
	}
	return -1;
}
//递归
int BinarySearch(int* arr, int n, int left, int right) {
	if (!arr || left > right) {
		return -1;
	}
	int mid = left + (right - left) / 2;
	if (n == arr[mid]) {
		return mid;
	}
	if (n < arr[mid]) {
		return BinarySearch(arr, n, left, mid - 1);
	}
	if (n > arr[mid]) {
		return BinarySearch(arr, n, mid + 1, right);
	}
}
int main() {
	int arr[10] = { 0, 5, 9, 10, 10, 15, 22, 25, 40, 50};
	int index = BinarySearch(arr, 15, 0, 9);
	cout << index << endl;
}
```

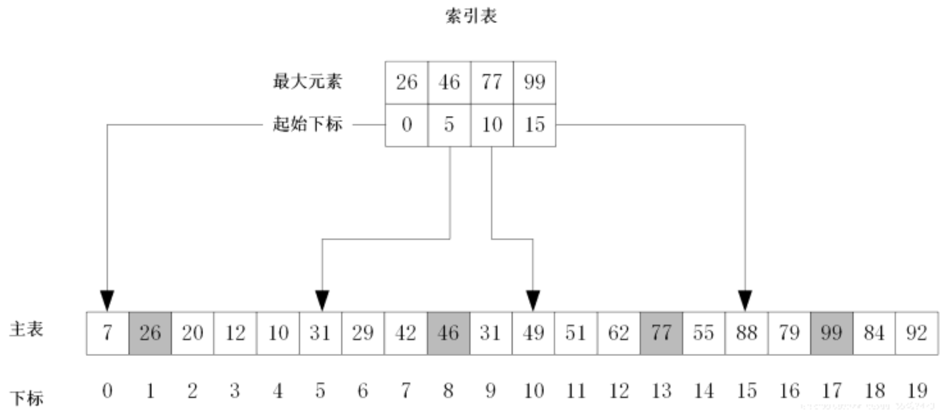
3、分块查找

3.1说明

分块查找又称索引顺序查找，它是顺序查找的一种改进方法。 是一种性能介于顺序查找和二分查找之间的查找方法。把数据分成若干块，在每一块中的数据元素的存储顺序是任意的，但要求块与块之间须按关键字值的大小有序排列，还要建立一个按关键字值递增顺序排列的索引表，索引表中的一项对应线形表中的一块，索引项包括两个内容：1) 键域存放相应块的最大关键字；2) 链域存放指向本块第一个结点的指针。分块查找分两步进行，先确定待查找的结点属于哪一块，然后在块内查找结点。

3.2基本思想

（1） 将n个数据元素"按块有序"划分为m块(m ≤ n)。每一块中的结点不必有序，但块与块之间必须"按块有序"，即前一块中任一元素的关键字都必须小于(或大于)后一块中任一元素的关键字。
（2） 根据所有块建立一张索引表，索引表中每一项由关键字字段(存放块中最大的元素的关键字)和指针字段(存放指向相应块的起始下标)组成。
（3） 查找分两个部分，先对索引表进行二分查找或顺序查找，以确定待查元素可能在哪一个块中，然后在已确定的块中顺序查找。



3.3复杂度分析

时间复杂度：O(log(m)+n/m)。

3.4算法实现

```
//索引表节点
struct iNode
{
    int idata;    //块中最大的元素
    int ipointer;    //块的起始下标
};
//分块查找
void block(iNode *index, int m, int *arr, int n, int key)        //index: 索引表数组，数组元素为索引节点
{
    //对索引表进行二分查找
    int low = 0, high = m - 1;
    int mid = 0;
    while (low <= high)
    {
        mid = (low + high) / 2;
        if (key <= index[mid].idata)
        {
            high = mid -1;
        }
        else
        {
            low = mid + 1;
        }
    }
    int aind = index[mid].ipointer;    //获得已确定块的起始下标
    bool ex =false;    //是否存在匹配
    while (arr[aind] <= index[mid].idata && aind != n)
    {
        if (key == arr[aind])
        {
            cout<<"数组中第 "<<aind+1<<" 个数据元素与关键字 \""<<key<<"\" 相匹配"<<endl;
            ex = true;
            ++aind;
            continue;
        }
        ++aind;
    }
    if (false == ex)
    {
        cout<<"数组中不存在数据元素与关键字 \""<<key<<"\" 相匹配"<<endl;
    }
}
```

4、插值查找

4.1说明

打个比方，在英文字典里面查“apple”，你下意识翻开字典是翻前面的书页还是后面的书页呢？如果再让你查“zoo”，你又怎么查？很显然，这里你绝对不会是从中间开始查起，而是有一定目的的往前或往后翻。同样的，比如要在取值范围1 ~ 10000 之间 100 个元素从小到大均匀分布的数组中查找5， 我们自然会考虑从数组下标较小的开始查找。
经过以上分析，折半查找这种查找方式，不是自适应的（也就是说是傻瓜式的）。二分查找中查找点计算如下：
mid=(low+high)/2, 即mid=low+1/2*(high-low)；通过类比，我们可以将查找的点改进为如下：mid=low+(key-

a[low])/(a[high]-a[low])*(high-low)，也就是将上述的比例参数1/2改进为自适应的，根据关键字在整个有序表中所处的位置，让mid值的变化更靠近关键字key，这样也就间接地减少了比较次数。

4.2基本思想

基于二分查找算法，将查找点的选择改进为自适应选择，可以提高查找效率。当然，插值查找也属于有序查找。
注：对于表长较大，而关键字分布又比较均匀的查找表来说，插值查找算法的平均性能比折半查找要好的多。反之，数组中如果分布非常不均匀，那么插值查找未必是很合适的选择。

4.3复杂度分析

查找成功或者失败的时间复杂度均为O($\log_2(\log_2 n)$)。

4.4算法实现

```
//插值查找
int InsertionSearch(int a[], int value, int low, int high)
{
    int mid = low+(value-a[low])/(a[high]-a[low])*(high-low);
    if(a[mid]==value)
        return mid;
    if(a[mid]>value)
        return InsertionSearch(a, value, low, mid-1);
    if(a[mid]<value)
        return InsertionSearch(a, value, mid+1, high);
}
```

5、斐波那契查找

5.1说明

黄金比例又称黄金分割，是指事物各部分间一定的数学比例关系，即将整体一分为二，较大部分与较小部分之比等于整体与较大部分之比，其比值约为1:0.618或1.618:1。

0.618被公认为最具有审美意义的比例数字，这个数值的作用不仅仅体现在诸如绘画、雕塑、音乐、建筑等艺术领域，而且在管理、工程设计等方面也有着不可忽视的作用。因此被称为黄金分割。

大家记不记得斐波那契数列：1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89……（从第三个数开始，后边每一个数都是前两个数的和）。然后我们会发现，随着斐波那契数列的递增，前后两个数的比值会越来越接近0.618，利用这个特性，我们就可以将黄金比例运用到查找技术中。

5.2基本思想

基本思想：也是二分查找的一种提升算法，通过运用黄金比例的概念在数列中选择查找点进行查找，提高查找效率。同样地，斐波那契查找也属于一种有序查找算法。
相对于折半查找，一般将待比较的key值与第mid=（low+high）/2位置的元素比较，比较结果分三种情况：

1）相等，mid位置的元素即为所求

2）>，low=mid+1;

3）<，high=mid-1。

斐波那契查找与折半查找很相似，他是根据斐波那契序列的特点对有序表进行分割的。他要求开始表中记录的个数为某个斐波那契数小1，及n=F(k)-1;

开始将k值与第F(k-1)位置的记录进行比较(及mid=low+F(k-1)-1),比较结果也分为三种：

1）相等，mid位置的元素即为所求

2）>，low=mid+1,k-=2;

说明：low=mid+1说明待查找的元素在[mid+1,high]范围内，k-=2 说明范围[mid+1,high]内的元素个数为n-(F(k-1))= Fk-1-F(k-1)=Fk-F(k-1)-1=F(k-2)-1个，所以可以递归的应用斐波那契查找。

3）<，high=mid-1,k-=1。

说明：low=mid+1说明待查找的元素在[low,mid-1]范围内，k-=1 说明范围[low,mid-1]内的元素个数为F(k-1)-1个，所以可以递归 的应用斐波那契查找。

5.3复杂度分析

复杂度分析：最坏情况下，时间复杂度为O($\log_2 n$)，且其期望复杂度也为O($\log_2 n$)。

5.4算法实现

```
#include<iostream>
using namespace std;
const int max_size = 20; //斐波那契数组的长度
/*构造一个斐波那契数组*/
void Fibonacci(int* F) {
	F[0] = 0;
	F[1] = 1;
	for (int i = 2; i < max_size; ++i) {
		F[i] = F[i - 1] + F[i - 2];
	}
}
/*定义斐波那契查找法*/
int FibonacciSearch(int* a, int n, int key) { //a为要查找的数组,n为要查找的数组长度,key为要查找的关键字
	int low = 0;
	int high = n - 1;
	int F[max_size];
	Fibonacci(F); //构造一个斐波那契数组F
	int k = 0;
	while (n > F[k] - 1) //计算n位于斐波那契数列的位置
		++k;
	int* temp; //将数组a扩展到F[k]-1的长度
	temp = new int[F[k] - 1];
	memcpy(temp, a, n * sizeof(int));
	for (int i = n; i < F[k] - 1; ++i)
		temp[i] = a[n - 1];
	while (low <= high) {
		int mid = low + F[k - 1] - 1;
		if (key < temp[mid]) {
			high = mid - 1;
			k -= 1;
		}
		else if (key > temp[mid]) {
			low = mid + 1;
			k -= 2;
		}
		else {
			if (mid < n)
				return mid;
			else
				return n - 1;
		}
	}
	delete[] temp;
	return -1;
}
int main()
{
	int a[] = { 0,16,24,35,47,59,62,73,88,99 };
	int key = 16;
	int index = FibonacciSearch(a, sizeof(a) / sizeof(int), key);
	cout << key << " is located at:" << index;
	return 0;
}
```

6、数表查找

6.1最简单的数表查找算法——二叉树查找算法

6.1.1基本思想

基本思想：二叉查找树是先对待查找的数据进行生成树，确保树的左分支的值小于右分支的值，然后在就行和每个节点的父节点比较大小，查找最适合的范围。这个算法的查找效率很高，但是如果使用这种查找方法要首先创建树。

二叉查找树（BinarySearch Tree，也叫二叉搜索树，或称二叉排序树BinarySort Tree）或者是一棵空树，或者是具有以下性质的二叉树：

（1） 若任意节点的左子树不空，则左子树上所有结点的值均小于它的根结点的值；；

（2） 若任意节点的右子树不空，则右子树上所有结点的值均大于它的根结点的值；

（3） 任意节点的左、右子树也分别为二叉查找树。

二叉查找树性质：对二叉查找树进行中序遍历，即可得到有序的数列。

6.1.2复杂度分析

复杂度分析：它和二分查找一样，插入和查找的时间复杂度均为O(logn)，但是在最坏的情况下仍然会有O(n)的时间复杂度。

6.2平衡查找树之2-3查找树（2-3 Tree）

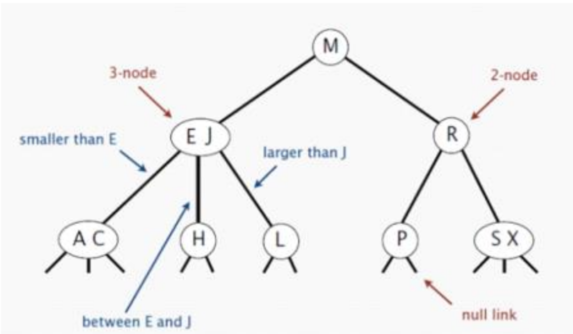
6.2.1定义

2-3查找树定义：和二叉树不一样，2-3树运行每个节点保存1个或者两个的值。对于普通的2节点(2-node)，他保存1个key和左右两个子节点。对应3节点(3-node)，保存两个Key，2-3查找树的定义如下：

（1）要么为空，要么：

（2）对于2节点，该节点保存一个key及对应value，以及两个指向左右节点的节点，左节点也是一个2-3节点，所有的值都比key要小，右节点也是一个2-3节点，所有的值比key要大；

（3）对于3节点，该节点保存两个key及对应value，以及三个指向左中右的节点。左节点也是一个2-3节点，所有的值均比两个key中的最小的key还要小；中间节点也是一个2-3节点，中间节点的key值在两个根节点key值之间；右节点也是一个2-3节点，节点的所有key值比两个key中的最大的key还要大；



6.2.2性质

（1）如果中序遍历2-3查找树，就可以得到排好序的序列；

（2）在一个完全平衡的2-3查找树中，根节点到每一个为空节点的距离都相同。（这也是平衡树中“平衡”一词的概念，根节点到叶节点的最长距离对应于查找算法的最坏情况，而平衡树中根节点到叶节点的距离都一样，最坏情况也具有对数复杂度）。

6.2.3复杂度分析

2-3树的查找效率与树的高度是息息相关的。

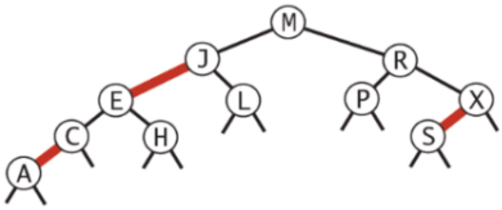
（1）在最坏的情况下，也就是所有的节点都是2-node节点，查找效率为lgN。

（2）在最好的情况下，所有的节点都是3-node节点，查找效率为$\log_3 N$约等于0.631lgN。

6.3平衡查找树之红黑树（Red-Black Tree）

6.3.1基本思想

红黑树的思想就是对2-3查找树进行编码，尤其是对2-3查找树中的3-nodes节点添加额外的信息。红黑树中将节点之间的链接分为两种不同类型，红色链接，他用来链接两个2-nodes节点来表示一个3-nodes节点。黑色链接用来链接普通的2-3节点。特别的，使用红色链接的两个2-nodes来表示一个3-nodes节点，并且向左倾斜，即一个2-node是另一个2-node的左子节点。这种做法的好处是查找的时候不用做任何修改，和普通的二叉查找树相同。



6.3.2红黑树的定义

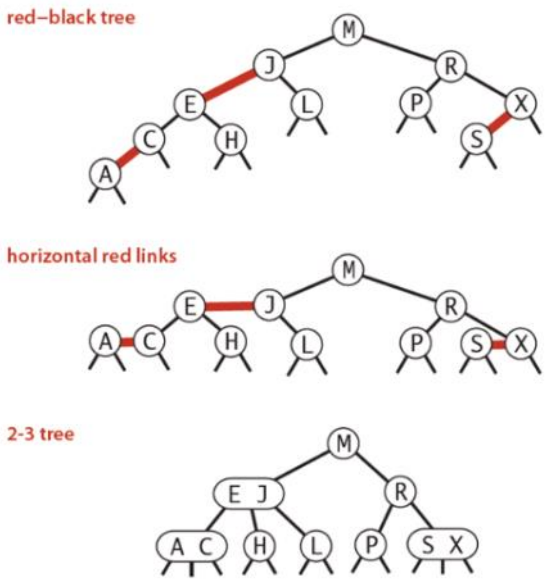
红黑树是一种具有红色和黑色链接的平衡查找树，同时满足：

（1）红色节点向左倾斜；

（2）一个节点不可能有两个红色链接；

（3）整个树完全黑色平衡，即从根节点到所有叶子节点的路径上，黑色链接的个数都相同。

下图可以看到红黑树其实是2-3树的另外一种表现形式：如果我们将红色的连线水平绘制，那么他链接的两个2-node节点就是2-3树中的一个3-node节点了。



6.3.3红黑树的性质

整个树完全黑色平衡，即从根节点到所以叶子结点的路径上，黑色链接的个数都相同（2-3树的第2）性质，从根节点到叶子节点的距离都相等）。

6.3.4复杂度分析

最坏的情况就是，红黑树中除了最左侧路径全部是由3-node节点组成，即红黑相间的路径长度是全黑路径长度的2倍。

implementation	worst-case cost (after N inserts)			average case (after N random inserts)			ordered iteration?	key interface
	search	insert	delete	search hit	insert	delete		
sequential search (unordered list)	N	N	N	N/2	N	N/2	no	equals()
binary search (ordered array)	lg N	N	N	lg N	N/2	N/2	yes	compareTo()
BST	N	N	N	1.39 lg N	1.39 lg N	?	yes	compareTo()
2-3 tree	c lg N	c lg N	c lg N	c lg N	c lg N	c lg N	yes	compareTo()
red-black BST	2 lg N	2 lg N	2 lg N	1.00 lg N *	1.00 lg N *	1.00 lg N *	yes	compareTo()

6.3.5应用

红黑树这种数据结构应用十分广泛，在多种编程语言中被用作符号表的实现，如：

- Java中的java.util.TreeMap,java.util.TreeSet;
- C++ STL中的：map,multimap,multiset;
- .NET中的：SortedDictionary,SortedSet 等。

红黑树这种数据结构应用十分广泛，在多种编程语言中被用作符号表的实现，如：

- Java中的java.util.TreeMap,java.util.TreeSet;
- C++ STL中的：map,multimap,multiset;
- .NET中的：SortedDictionary,SortedSet 等。

6.4 B树和B+树

6.4.1定义

维基百科对B树的定义为“在计算机科学中，B树（B-tree）是一种树状数据结构，它能够存储数据、对其进行排序并允许以O(log n)的时间复杂度运行进行查找、顺序读取、插入和删除的数据结构。B树，概括来说是一个节点可以拥有多于2个子节点的二叉查找树。与自平衡二叉查找树不同，B树为系统最优化大块数据的读和写操作。B-tree算法减少定位记录时所经历的中间过程，从而加快存取速度。普遍运用在数据库和文件系统。

B树定义：B树可以看作是对2-3查找树的一种扩展，即允许每个节点有M-1个子节点。

（1）根节点至少有两个子节点；

（2）每个节点有M-1个key，并且以升序排列；

（3）位于M-1和M key的子节点的值位于M-1和M key对应的value之间；

（4）其他节点至少有M/2个子节点。

下图是一个M=4 阶的B树：

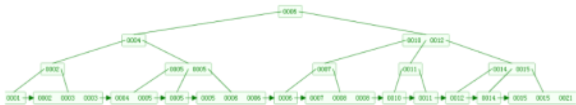
往B树中依次插入： 6 10 4 14 5 11 15 3 2 12 1 7 8 8 6 3 6 21 5 15 15 6 32 23 45 65 7 8 6 5 4。

B+树定义：B+树是对B树的一种变形树，它与B树的差异在于：

（1）有k个子节点的节点必然有k个关键码；

（2）非叶节点仅具有索引作用，跟记录有关的信息均存放在叶节点中；

（3）树的所有叶节点构成一个有序链表，可以按照关键码排序的次序遍历全部记录。

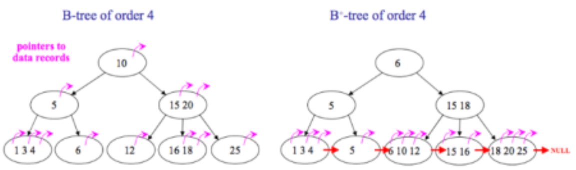


6.4.2 B和B+树区别

B和B+树的区别在于，B+树的非叶子节点只包含导航信息，不包含实际的值，所有的叶子节点和相连的节点使用链表相连，便于区间查找和遍历。

B+ 树的优点在于：

- 由于B+树在内部节点上不含数据信息，因此在内存页中能够存放更多的key。 数据存放的更加紧密，具有更好的空间局部性。因此访问叶子节点上关联的数据也具有更好的缓存命中率。
- B+树的叶子结点都是相链的，因此对整棵树的便利只需要一次线性遍历叶子结点即可。而且由于数据顺序排列并且相连，所以便于区间查找和搜索。而B树则需要进行每一层的递归遍历。相邻的元素可能在内存中不相邻，所以缓存命中性没有B+树好。
- 但是B树也有优点，其优点在于，由于B树的每一个节点都包含key和value，因此经常访问的元素可能离根节点更近，因此访问也更迅速。



- ### 6.4.3应用
- B/B+树常用于文件系统和数据库系统中，它通过对每个节点存储个数的扩展，使得对连续的数据能够进行较快的定位和访问，能够有效减少查找时间，提高存储的空间局部性从而减少IO操作。它广泛用于文件系统及数据库中，如：
- （1）Windows：HPFS文件系统；
- （2）Mac：HFS，HFS+文件系统；
- （3）Linux： ResiserFS，XFS，Ext3FS，JFS文件系统；
- （4）数据库：ORACLE，MYSQL，SQLSERVER等中。

8、哈希查找

8.1说明

- 什么是哈希表?
- 使用一个下标范围比较大的数组来存储元素。可以设计一个函数（哈希函数，也叫做散列函数），使得每个元素的关键字都与一个函数值（即数组下标）相对应，于是用这个数组单元来存储这个元素。也可以简单的理解为，按照关键字为每个元素“分类”，然后将这个元素存储在相应“类”所对应的地方。但是，不能够保证每个元素的关键字与函数值是一一对应的，因此，极有可能出现对于不同的元素，却计算出了相同的函数值，这样就产生了冲突，换句话说，就是把不同的元素分在了相同的“类”中。
- 什么是哈希函数?
- 哈希函数的规则是：通过某种转换关系，使关键字适度的分散到指定大小的顺序结构中，越分散，则以后查找的时间复杂度越小，空间复杂度越高。

8.2算法思想

- 算法思想：哈希的思路很简单，如果所有的键都是整数，那么就可以使用一个简单的无序数组来实现：将键作为索引，值即为其对应的值，这样就可以快速访问任意键的值。这是对于简单的键的情况，我们将其扩展到可以处理更加复杂的类型的键。

8.3算法流程

- （1）用给定的哈希函数构造哈希表；
- （2）根据选择的冲突处理方法解决地址冲突（常见的解决冲突的方法：拉链法和线性探测法）；
- （3）在哈希表的基础上执行哈希查找。
- 哈希表是一个在时间和空间上做出权衡的经典例子。如果没有内存限制，那么可以直接将键作为数组的索引。那么所有的查找时间复杂度为O(1)；如果没有时间限制，那么我们可以使用无序数组并进行顺序查找，这样只需要很少的内存。哈希表使用了适度的时间和空间来在这两个极端之间找到了平衡。只需要调整哈希函数算法即可在时间和空间上做出取舍。

8.4复杂度分析

- 单纯论查找复杂度：对于无冲突的Hash表而言，查找复杂度为O(1)（注意，在查找之前我们需要构建相应的Hash表）。
- 使用Hash，付出了什么?
- 我们在实际b编程中存储一个大规模的数据，最先想到的存储结构可能就是map，也就是我们常说的KV pair，经常使用python可能会体会到，使用map的好处就是，在后续处理数据时，我们可以根据数据的key快速地查询到对应的value值。map的本质就是Hash表，那么我们在获取超高查找效率的基础上，付出了什么?
- Hash是一种典型以空间换时间的算法，比如原来一个长度为100的数组，对其查找，只需要遍历且匹配相应记录即可，从空间复杂度来看，假如数组存储的是byte类型数据，那么该数组占用100byte空间。现在我们采用哈希算法，我们前面说的哈希必须有一个规则，约束键与存储位置的关系，那么就需要一个固定长度的Hash表，此时，仍然是100byte的数组，假设我们需要的100byte用来记录键与位置的关系，那么总的空间为200byte，而且用于记录规则的表大小会根据规则，大小可能是不定的。

implementation	worst-case cost (after N inserts)			average case (after N random inserts)			ordered iteration?	key interface
	search	insert	delete	search hit	insert	delete		
sequential search (unordered list)	N	N	N	N/2	N	N/2	no	equals()
binary search (ordered array)	lg N	N	N	lg N	N/2	N/2	yes	compareTo()
BST	N	N	N	1.38 lg N	1.38 lg N	?	yes	compareTo()
red-black tree	2 lg N	2 lg N	2 lg N	1.00 lg N	1.00 lg N	1.00 lg N	yes	compareTo()
separate chaining	lg N *	lg N *	lg N *	3-5 *	3-5 *	3-5 *	no	equals()
linear probing	lg N *	lg N *	lg N *	3-5 *	3-5 *	3-5 *	no	equals()

来自 <https://blog.csdn.net/jerry_yiu/article/details/9228035>