**第八章 查找**

**为什么要研究查找？**

**查找（Searching,又称检索或搜索）无论是在日常生活中，还是在计算机的系统软件和应用软件都会广泛涉及到。信息检索已经成为计算机应用领域中的一个方面。当所要查找数据量相当大时，寻找一种高效的查找方法就显得尤为重要。如果数据元素或记录项很多以至于内存都无法存放所有的数据元素或记录，则有必要把一些数据元素存储到磁盘或磁带上去，在这种情况下的查找叫做外部查找；而如果所需要的查找的数据元素都在内存中，则称这种情况下的查找为内部查找（这里只考虑内部查找，当然有些算法在外部查找中同样适用）。**

**8.1基本概念**

**8.1.1 集合**

**由任意一些可分辨的对象构成的整体，集合的元素没任何限制。为方便起见，我们只考虑类型相同的数据元素构成的集合。**

**集合中元素之间不存在逻辑关系，即元素除了同处于一个集合外，没有其他关系。**

**8.1.2查找表（Search Table）**

**查找表是由同一类型的数据元素（或记录）构成的集合。**

**关键字（Key）：是数据元素（或记录）中某个数据项的值，用它可以标识一个数据元素（或记录）。若此关键字可以唯一地标识一个记录，则称此关键字为主关键字（Primary Key）。有些关键字不能唯一地标识数据元素，则称次关键字。**

**查找（又称为检索）：根据给定的某个值，在查找表中确定一个关键字值等于给定值的记录。若表中存在这样的一个记录，则称查找成功，否则查找不成功。**

**查找操作使用较频繁，主要考虑时间和空间复杂度，由于查找只需几个辅助空间，所以主要考虑时间性能。用记录的关键字与给定的值比较次数作为衡量查找算法的优劣。**

**通常把在查找过程中对关键字需要执行的平均比较次数称为平均查找长度ASL（Average Search Length）:**

****

**其中：为查找第i个记录时的概率，且，**

**为找到第i个记录时所需要的比较次数。**

**假如一百之内的自然数放到与下标对应的数组中，即：**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **a[1]** | **a[2]** | **a[3]** | **…..** | **a[100]** |
| **1** | **2** | **3** | **…..** | **100** |

**如果顺序查找（从头到尾或从尾到头查找），假设每个元素查找概率相等，则**

****

**即要查找到一个元素平均为50.50次，显然这种方式效率较低；**

**如果按下标查找，则**

****

**即查找到一个元素只需1次，显然这种方式效率最高。**

**8.2 静态查找**

**静态查找主要适用于固定大小的线性表，对线性表不进行修改。静态查找以顺序表或线性链表作为存储。**

**顺序表的定义如下：**

**#define MAXSIZE 100**

**typedef int KeyType;**

**typedef struct**

**{**

**KeyType key;**

**……**

**}ElemtType;**

**typedef struct**

**{**

**ElemtType elem[MAXSIZE];**

**int n;**

**}SqTable;**

**8.2.1 顺序查找（Sequential Search）**

**适用于有序表或无序表，最简单和最直接的查找方法。**

**思路是：从表的一端开始，顺序扫描线性表，依次将记录的关键字与给定值比较，若某个记录的关键字与所找的值相等，则查找成功，否则查找失败。**

**顺序查找算法如下：**

**查找k（反向进行，即R.elem[n-1]，R.elem[n-2]，…，R.elem[0]）**

|  |  |
| --- | --- |
| **R.elem[0]** | **k的“监视哨”**  **查到此表示无k结束（失败）** |
| **R.elem[1]** |  |
| **R.elem[2]** |  |
| **….** | **k查到此退出（成功）** |
| **….** |  |
| **R.elem[n-1]** |  |

**int SeqSearch(SqTable R,KeyType k)**

**{ //在表R中查找k**

**R.elem[0].key=k; /\* 监视哨 \*/**

**i=R.n;**

**while(R.elem[i].key!=k)i--;**

**return i;//成功返回i,失败返回0**

**}**

**顺序查找的性能分析：**

****

**顺序查找算法比较简单，对无序表和有序表及链式存储的线性表都适用。缺点是平均查找时间较长，效率低。**

**8.2.2 二分查找（Binary Search又称折半查找）**

**适用于有序表。**

**思路是：由于待查的表中的数据元素按关键字有序（假设数据排列自左至右为升序），选择位于表中查找区间内中间位置上（下标为i）的数据元素，将其关键字与给定值k进行比较，结果可能有三种情况：**

**若k=R.elem[i].key，则查找成功；**

|  |  |  |
| --- | --- | --- |
| **左半部** | **折半元素** | **右半部** |
|  | **k =[i]** |  |

**若k<R.elem[i].key，则查找的元素位于左半部；**

|  |  |  |
| --- | --- | --- |
| **左半部** | **折半元素** | **右半部** |
| **k<[i]** |  |  |

**若k>R.elem[i].key，则查找的元素位于右半部。**

|  |  |  |
| --- | --- | --- |
| **左半部** | **折半元素** | **右半部** |
|  |  | **k>[i]** |

**若中间位置上的数据元素的关键字与给定的值不相等，则缩小查找区间并在新的区间内重复上述过程，直到查找成功或查找区间长度为0（失败）。**

**例1：设数据元素按升序是4，12，18，23，36，55，63，76，81，89，93，对关键字k=23进行二分查找（查找成功之例）。**

**在数组中存放如下：**

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **[1]** | **[2]** | **[3]** | **[4]** | **[5]** | **[6]** | **[7]** | **[8]** | **[9]** | **[10]** | **[11]** |
| **4** | **12** | **18** | **23**  **(k)** | **36** | **55** | **63** | **76** | **81** | **89** | **93** |

**进行二分查找，此时low=1，high=11，**

**折半处mid=(low+high)/2=(1+11)/2=12/2=6**

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **[1]**  **low** | **[2]** | **[3]** | **[4]**  **(k)** | **[5]** | **[6]**  **mid** | **[7]** | **[8]** | **[9]** | **[10]** | **[11]**  **high** |
| **4** | **12** | **18** | **23** | **36** | **55** | **63** | **76** | **81** | **89** | **93** |

**由于k=23<mid=55，在mid左边继续二分区间**

**low=1，high=mid-1=5，新的折半处mid=(1+5)/2=3**

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **[1]**  **low** | **[2]** | **[3]mid** | **[4]**  **(k)** | **[5]**  **high** | **[6]** | **[7]** | **[8]** | **[9]** | **[10]** | **[11]** |
| **4** | **12** | **18** | **23** | **36** | **55** | **63** | **76** | **81** | **89** | **93** |

**由于k=23>mid=18，在mid右边继续二分区间**

**low=mid+1=4，high=5，新的折半处mid=(4+5)/2=4**

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **[1]** | **[2]** | **[3]** | **[4]**  **low mid**  **(k)** | **[5] high** | **[6]** | **[7]** | **[8]** | **[9]** | **[10]** | **[11]** |
| **4** | **12** | **18** | **23** | **36** | **55** | **63** | **76** | **81** | **89** | **93** |

**关键字k=[mid=4]=23找到（查找成功）。**

**例2：对关键字k=80进行二分查找（查找失败之例）。**

**此时low=1, high=11, mid=(low+high)/2=6**

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **[1]**  **low** | **[2]** | **[3]** | **[4]** | **[5]** | **[6]**  **mid** | **[7]** | **[8]** | **[9]** | **[10]** | **[11]**  **high** |
| **4** | **12** | **18** | **23** | **36** | **55** | **63** | **76** | **81** | **89** | **93** |

**由于k=80>mid=55，在mid右边继续二分区间**

**low=mid+1=7，high=11，新的折半处mid=(7+11)/2=9**

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **[1]** | **[2]** | **[3]** | **[4]** | | **[5]** | **[6]** | **[7] low** | **[8]** | **[9]**  **mid** | **[10]** | **[11]**  **high** |
| **4** | **12** | **18** | **23** | **36** | | **55** | **63** | **76** | **81** | **89** | **93** |

**由于k=80<mid=81，在mid左边继续二分区间**

**low=7，high=mid-1=8，新的折半处mid=(7+8)/2=7**

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **[1]** | **[2]** | **[3]** | | **[4]** | **[5]** | **[6]** | **[7] low mid** | **[8]**  **high** | **[9]** | **[10]** | **[11]** |
| **4** | **12** | **18** | **23** | | **36** | **55** | **63** | **76** | **81** | **89** | **93** |

**由于k=80>mid=63，在mid右边继续二分区间**

**low=mid+1=8，high=8，新的折半处mid=(8+8)/2=8**

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **[1]** | **[2]** | **[3]** | **[4]** | **[5]** | **[6]** | **[7]** | **[8]**  **low mid**  **high** | **[9]** | **[10]** | **[11]** |
| **4** | **12** | **18** | **23** | **36** | **55** | **63** | **76** | **81** | **89** | **93** |

**由于high=low=mid（查找区间长度为0）都未找到k=80，所以查找失败。**

**int BinSearch(SqTable R,KeyType k)**

**{**

**int low=1,high=R.n,mid;**

**while(low<=high)**

**{**

**mid=(low+high)/2;**

**if(R.elem[mid].key==k)return mid;**

**else if(R.elem[mid].key<k) low=mid+1;**

**else high=mid-1;**

**}**

**return 0;**

**}**

* **二分查找过程可用二叉树来描述，把当前查找区**

**间的中间位置上的结点作为根，左（右）半区间分别作为根的左（右）子树，左（右）区间再按此方法分下去，由此得到二叉树，称为二分查找的判定树(Decision Tree)。该树无论是下标和元素的值都满足二叉树的中序遍历排序序列。**

**例如上例:**

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **[1]** | **[2]** | **[3]** | **[4]** | **[5]** | **[6]** | **[7]** | **[8]** | **[9]** | **[10]** | **[11]** |
| **4** | **12** | **18** | **23** | **36** | **55** | **63** | **76** | **81** | **89** | **93** |
| **左子树** | | | | | **mid**  **根** | **右子树** | | | | |
| **4** | **12** | **18** | **23** | **36** |  | **63** | **76** | **81** | **89** | **93** |
| **左子树** | | **mid**  **根** | **右子树** | | **左子树** | | **mid**  **根** | **右子树** | |
| **4** | **12** |  | **23** | **36** | **63** | **76** |  | **89** | **93** |
| **mid**  **根** | | **mid**  **根** | | **mid**  **根** | | **mid**  **根** | |
|  | **12** |  | **36** |  | **76** |  | **93** |
| **右叶** | **右叶** | **右叶** | **右叶** |

**借助于二叉判定树，折半查找的平均查找长度：**

**第1层：结点55只需查找1次；**

**第2层：结点18和81各需查找2次；**

**第3层：结点4、23、63和89各需查找3次；**

**第4层：结点12、36、76和93各需查找4次。**

****

**当n>50时，近似结果：**

**若n=100时，≒6**

**述评：虽然二分查找效率高，但只适用于有序表，且限于顺序存储结构。**

**程序：SeqBinSearch.c对顺序表的顺序查找和二分查找（折半查找）**

**#define MAXSIZE 11**

**typedef int KeyType;**

**typedef struct**

**{**

**KeyType key;**

**}ElemtType;**

**typedef struct**

**{**

**ElemtType elem[MAXSIZE+1];**

**int n;**

**}SqTable;**

**void SeqSearch(SqTable R,KeyType k)**

**{**

**int i;**

**R.elem[0].key=k; /\* 监视哨 \*/**

**i=R.n;**

**while(R.elem[i].key!=k)i--;**

**if(i!=0)**

**printf("关键字%d顺序查找的次序是：**

**%d\n",k,i);**

**else printf("关键字%d顺序查找失败!\n",k);**

**}**

**void BinSearch(SqTable R,KeyType k)**

**{**

**int low=1,high=R.n,mid;**

**while(low<=high)**

**{**

**mid=(low+high)/2;**

**if(R.elem[mid].key==k)**

**{**

**printf("关键字%d二分查找的次序是%d\n",**

**k,mid);**

**return;**

**}**

**else if(R.elem[mid].key<k)low=mid+1;**

**else high=mid-1;**

**}**

**printf("关键字%d二分查找失败!\n",k);**

**}**

**void main( )**

**{**

**int i;**

**KeyType key;**

**SqTable a={{0,4,12,18,23,36,55,63,76,81,89,93},11};**

**printf("顺序表: ");**

**for(i=1;i<=a.n;i++)printf("%5d",a.elem[i].key);**

**printf("\n");**

**printf("输入查找关键字: ");scanf("%d",&key);**

**SeqSearch(a,key);**

**printf("\n");**

**BinSearch(a,key);**

**}**

**8.2.3 分块查找（Blocking Search）**

**又称索引表查找，为顺序查找的一种改进方法。除数据表外，还需一个索引表。索引表是有序表，所以表分块有序。**

**例1：图示的表和索引表，表中含有12个记录，可分成3个子表，分别是：（8，4，5，3）、（27，25，44，30）和（78，67，90，66），表分块有序，即第2个子表的所有记录的关键字值均大于第1个子表中的最大的关键字值；第3个子表的所有记录的关键字值均大于第2个子表中的最大的关键字值，…..。**

**索引表和顺序表**

|  |  |  |  |
| --- | --- | --- | --- |
| **块内最大关键字** | **8** | **44** | **90** |
| **块内起始位置** | **1** | **5** | **9** |

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9** | **10** | **11** | **12** |
| **8** | **4** | **5** | **3** | **27** | **25** | **44** | **30** | **78** | **67** | **90** | **66** |

**查找过程：第一步查索引表，确定待查的元素所在的块。由于索引表为顺序表，可以采用二分或顺序法查找；第二步在确定的块内进行顺序查找，由于子表内的元素无序，所以只能用顺序查找。**

**8.3 树表的查找**

**8.3.1二叉排序树（Binary Sort Tree，简称BST）**

**如果树非空，有下列性质的二叉树称为二叉排序树（BST）：**

**（1）若它的左子树非空，则左子树上所有的结点的值均小于它的根结点的值。**

**（2）若它的右子树非空，则右子树上所有结点的值均大于或等于它的根结点的值。**

**（3）它的左、右子树又都是一棵二叉排序树。**

**由上定义可知：二叉排序树各结点的值是一个按中序遍历的递增序列。**

**如：下列是一棵二叉排序树（BST）**

**结点的值为中序递增序列：7，25，27，39，40，**

**46，63，77，81，86**

**二叉排序树存储结构：**

**typedef struct node**

**{**

**KeyType key;**

**DataType other;**

**struct node \*lchild,\*rchild;**

**}BSTNode,\*BSTree;**

**8.3.2二叉排序树的操作**

**对于操作介绍：插入和创建、遍历、查找和删除**

* + - 1. **二叉排序树的插入和创建**

**插入的原则是：每插入一个结点之后要保证仍是一棵二叉排序树。**

**插入的算法是：**

**（1）若二叉排序树为空，则创建一个根结点；**

**（2）即将插入的结点比根结点的值小则插入根结点的左子树，否则插入右子树；**

**（3）重复进行上述的插入二叉排序树逐渐长大。**

**注意：每次插入总是从根结点进入，与根结点的值相等的情况这里规定插入右子树（当然也可以规定插入左子树），不管如何规定，生成的树可能不一样，但中序排序是一样的。**

**例1：将通过插入关键字67，24，78，53，12，99创建一棵二叉排序树**

**第1步：Φ（空树）**

**第2步：插入67（作为根）**

**第3步：插入24**

**第4步：插入78**

**第5步：插入53**

**第6步：插入12**

**第7步：插入99**

**该二叉树的中序遍历序列（升序）是：12，24，53，67，78，99**

**思考题：树的形状取决于什么？一旦给定一个关键字序列，形成的树的形状是唯一的吗？**

**二叉排序树插入算法如下：**

**void InsertBST(BSTree \*Tptr,KeyType key)**

**{**

**BSTNode \*f,\*p=\*Tptr;**

**while(p!=NULL)**

**{**

**if(p->key==key)return;**

**f=p;**

**p=(key<p->key)?p->left:p->right;**

**}**

**p=(BSTNode \*)malloc(sizeof(BSTNode));**

**p->key=key;**

**p->left=p->right=NULL;**

**if(\*Tptr==NULL)\*Tptr=p;**

**else if(key<f->key)f->left=p;**

**else f->right=p;**

**}**

**二叉排序树创建算法：**

**BSTree CreateBST(int n,KeyType R[])**

**{**

**int i;**

**BSTree T=NULL;**

**for(i=0;i<n;i++)InsertBST(&T,R[i]);**

**return T;**

**}**

**8.3.2.2二叉排序树的遍历**

**相同于以前介绍过的二叉树中序遍历，可以使用递**

**归或非递归算法，为简单起见，不妨采用递归算法。**

**void InOrder(BSTree p)**

**{**

**if(p!=NULL)**

**{**

**InOrder(p🡪left);**

**printf("%5d",p🡪key);**

**InOrder(p🡪right);**

**}**

**}**

**8.3.2.3 二叉排序树的查找**

**首先把给定值和根结点的关键字比较，若相等，则查找成功。若给定值比根结点的关键字小，则在左子树上继续查找，否则在右子树上继续查找。**

**递归算法如下：**

**BSTNode \*SearchBST(BSTree T,KeyType key)**

**{**

**if(T==NULL||key==T->key)return T;**

**if(key<T->key)return SearchBST(T->left,key);**

**else return SearchBST(T->right,key);**

**}**

**8.3.2.4 二叉排序树的删除**

**对于一般二叉树来说，删去树中某个结点是没有意义的。因为它将使已被删结点为根的子树成为森林，破坏了整棵树的结构。然而，对于二叉排序树，删去树的一个结点相当于删去一个有序序列中的一个记录，只要在删除某个结点之后依旧保持二叉排序树的特性即可。**

**删除要比插入复杂的多，因为插入的结点都是连接到树的叶子结点上，故不破坏树的结构，而删除则不同，删除的结点可能是叶子或非叶子结点，当删除非叶子结点时，就会破坏原结点的链接关系，需要重新修改指针，使得删除后仍为一棵二叉排序树。**

**下面讨论删除二叉排序树结点的三种情况：**

**(1)若删除的是叶子结点，直接删除即可。因为删除叶子结点后不会破坏原来二叉排序树的中序遍历序列。**

**如对于下列二叉排序树，可以直接删除叶子结点12，53和99**

**●删除前中序排序序列是：12，24，53，67，78，99**

**●准备删除叶子结点12**

**●删除叶子结点12后，中序排序序列是：24，53，67，78，99**

**●准备删除叶结点53**

**●删除叶子结点53后，中序排序序列是：24，67，78，99**

**●准备删除叶结点99**

**●删除叶子结点99后，中序排序序列是：24，67，78**

**述评：叶子结点被删除后，不破坏原二叉排序树的结构，即仍保持树的中序排序序列。下同。**

**(2)若被删的结点只有左（右）孩子结点，则被删的结点被其左（右）孩子结点代替即可。**

**●如准备删除下列二叉排序树中的结点24，该结点只有左孩子结点12**

**●删除结点24后，其位置被孩子结点12代替**

**●准备删除结点78，该结点只有右孩子结点99**

**●删除结点78后，其位置被孩子结点99代替**

**(3)若被删的结点左、右孩子（或子孙）都有，被删结点的位置被其直接前驱（或直接后继）结点取代即可。**

**例1：对下列二叉排序树，要被删除的结点24，其左、右孩子都有**

**其一：删除结点24后，位置被其直接前驱结点12取代**

**其二：删除结点24后，位置被其直接后继结点53取代**

**例2：对下列二叉排序树，要被删除的结点67，其有子孙**

**●其一：删除结点67后，位置被其直接前驱结点53取代**

**●其二：删除结点67后，位置被其直接后继结点78取代**

**述评：无论被删结点由其直接前驱或直接后继结点取代其位置，显然形成的两棵树是不同的，但其中序排序序列是相同的。**

**算法如下：**

**BSTNode \*DelBSTNode(BSTNode \*Tptr,KeyType key)**

**{**

**BSTNode \*p=Tptr,\*f=NULL,\*s,\*q;**

**while(p!=NULL)**

**{**

**if(p->key==key)break;**

**f=p;**

**if(p->key>key)p=p->left;**

**else p=p->right;**

**}**

**if(p==NULL)return Tptr;**

**if(p->left==NULL)**

**{**

**if(f==NULL)Tptr=p->right;**

**else if(f->left==p)f->left=p->right;**

**else f->right=p->right;**

**free(p);**

**}**

**else**

**{**

**q=p;**

**s=p->left;**

**while(s->right!=NULL){q=s;s=s->right;}**

**if(q==p)q->left=s->left;**

**else q->right=s->left;**

**p->key=s->key;**

**free(s);**

**}**

**return Tptr;**

**}**

**8.3.2.5 二叉排序树的查找分析ASL**

**在具有n个结点二叉排序树查找中，ASL不会超过二叉树的深度，最好为，最差为n。因此，最好和最坏时间复杂度分别为和O(n)。一般情形下，其时间复杂度大致可看成，比顺序查找效率要好，但比二分查找要差。**

**二叉排序树的形状取决于：结点的值和插入顺序**

**例如:下列由序列67，24，78，12，99生成的二叉排序树之一**

**ASL=(1+2\*2+3\*2)/5=11/5=2.2**

**述评：该树相对匀称和平衡，平均查找长度较小**

**又如：由序列12，24，67，78，99生成的二叉排序树之二**

**ASL=(1+2+3+4+5)/5=15/5=3.0**

**述评：该树既不匀称又不平衡，平均查找长度较大**

**由此可知，二叉排序树有缺点，树的结构事先无法预料，随意性很大，它与结点的值和插入顺序有关，往往得到的是一棵很不“平衡”的二叉树，与理想的平衡二叉树相差甚远，树的高度越高，其运算时间越长。为了克服这个缺点，需要在插入和删除结点时对树的结构进行必要的调整，使得二叉树始终处于一种平衡的状态，这就是下节所要提到的平衡二叉树（Banlanced Binary Tree）。**

**程序：SortTree.c对二叉排序树的生成、遍历、查找和删除**

**#include<stdio.h>**

**typedef int KeyType;**

**typedef struct node**

**{**

**KeyType key;**

**struct node \*left,\*right;**

**}BSTNode,\*BSTree;/\* BST 的定义 \*/**

**void InsertBST(BSTree \*Tptr,KeyType key)**

**{/\* 结点的插入 \*/**

**BSTNode \*f,\*p=\*Tptr;**

**while(p!=NULL)**

**{**

**if(p->key==key)return;**

**f=p;**

**p=(key<p->key)?p->left:p->right;**

**}**

**p=(BSTNode \*)malloc(sizeof(BSTNode));**

**p->key=key;**

**p->left=p->right=NULL;**

**if(\*Tptr==NULL)\*Tptr=p;**

**else if(key<f->key)f->left=p;**

**else f->right=p;**

**}**

**BSTree CreateBST(int n,KeyType R[])**

**{/\* 树的创建 \*/**

**int i;**

**BSTree T=NULL;**

**for(i=0;i<n;i++)InsertBST(&T,R[i]);**

**return T;**

**}**

**BSTNode \*DelBSTNode(BSTNode \*Tptr,KeyType key)**

**{/\* 树的删除 \*/**

**BSTNode \*p=Tptr,\*f=NULL,\*s,\*q;**

**while(p!=NULL)**

**{**

**if(p->key==key)break;**

**f=p;**

**if(p->key>key)p=p->left;**

**else p=p->right;**

**}**

**if(p==NULL)return Tptr;**

**if(p->left==NULL)**

**{**

**if(f==NULL)Tptr=p->right;**

**else if(f->left==p)f->left=p->right;**

**else f->right=p->right;**

**free(p);**

**}**

**else**

**{**

**q=p;**

**s=p->left;**

**while(s->right!=NULL){q=s;s=s->right;}**

**if(q==p)q->left=s->left;**

**else q->right=s->left;**

**p->key=s->key;**

**free(s);**

**}**

**return Tptr;**

**}**

**BSTNode \*SearchBST(BSTree T,KeyType key)**

**{/\* 树的查找 \*/**

**if(T==NULL||key==T->key)return T;**

**if(key<T->key)return SearchBST(T->left,key);**

**else return SearchBST(T->right,key);**

**}**

**void InOrder(BSTree p)**

**{/\* 树的遍历 \*/**

**if(p!=NULL)**

**{**

**InOrder(p->left);**

**printf("%5d",p->key);**

**InOrder(p->right);**

**}**

**}**

**#define N 6**

**void main()**

**{**

**int i;**

**KeyType data;**

**KeyType a[N]={67,24,78,53,12,99};**

**BSTree root;**

**printf("\nCreate Tree:");**

**for(i=0;i<N;i++)printf("%5d",a[i]);**

**printf("\n Sort Tree:");**

**root=CreateBST(N,a);**

**InOrder(root);**

**printf("\n\nInput Search Data: ");**

**scanf("%d",&data);**

**if(SearchBST(root,data)!=NULL)printf("%d is found!\n",data);**

**else printf("%d is not found!\n",data);**

**printf("\nInput Deleted Data: ");scanf("%d",&data);**

**printf("\nAfter Delete:");**

**root=DelBSTNode(root,data);**

**InOrder(root);printf("\n");**

**getch();**

**}**

**8.3.3 平衡二叉树(Balanced Binary Tree)**

**由上节的讨论可知，二叉排序树的查找效率取决于树的形态，而构造一棵形态均匀的二叉树与结点的值和插入的次序有关。但是结点插入的先后次序不是随意而定的，这就要求我们找到一种平衡的方法，对于任意给定的关键字序列边插入边调整构造出一棵形态匀称的二叉排序树。**

**如何构造出一棵平衡二叉树呢？前苏联的阿德尔森-威尔斯基和兰迪斯（Adelson-Velskii & Landis）提出了一个动态地保持二叉排序树平衡的方法，所以这种树又称AVL树。**

**平衡因子和平衡二叉树：一棵树某结点的左、右子树高（深）度之差，称为平衡因子（Balance Factor，简称BF）。若一棵二叉树的每个结点的平衡因子的绝对值不超过1，则称这棵树为平衡二叉树。因此，平衡二叉树的所有结点的平衡因子只能是-1，0，1。当然只要有一个结点不是这种情况就不是平衡二叉树。**

**例如下面的树是平衡二叉树（括号中的数字是其平衡因子，下同）**

**又例如下面的二叉树不是平衡二叉树**

**由特殊二叉树和平衡二叉树的特点不难看出：**

**（1）由于叶子结点没有左右子树，故叶子结点的平衡因子必为0；**

1. **完全二叉树和满二叉树是平衡二叉树且后**

**者的平衡因子都是0；**

**例如，下列完全二叉树是平衡二叉树**

**又例如，下列满二叉树是平衡二叉树且平衡因子都是0**

**(3)平衡因子都是0的二叉树是平衡二叉树，反之则不一定。**

**例如，下列平衡二叉树而平衡因子不都是0**

**●对非平衡二叉树的平衡处理：当一棵二叉排序树是平衡的，若插入某个结点后失衡，需要进行平衡处理，让其重新变成平衡状态。若多处结点发生失衡的情况，应该优先处理最小失衡子树（离插入结点最近且平衡因子的绝对值大于1的结点作为根的子树）。平衡处理原则是：处理前后二叉树的中序排序序列是不变的。**

**平衡处理的一般方法分为三步：**

**(1)先找最小失衡子树，若不存在则处理结束，否则转入下一步；**

**(2)找出最小失衡子树涉及到的三个结点，并将其关键字升序排序（设中序遍历为A，B，C），并按中序重新构造一棵二叉树（结点B为根，结点A和C及其各自的子孙分别为B的左、右子树）；**

**(3)参照处理前树各结点的中序升序序列，将A，B，C的原各子树的结点按中序升序插入到新二叉树的适当位置，转入(1)。**

**平衡处理分四种情况：LL（左左）型、RR（右右）型、LR（左右）型和RL（右左）型**

**分别讨论如下：**

**(1)LL（左左）型**

**●下图插入前原树是平衡的**

**其中：[h]为子树的高（深）度，下同**

**●插入结点BL+后失衡，由于失衡因子在A上，显然最小失衡子树为：A-B-BL+，所以应调整它。由于A-B为左子树，B-BL+也是左子树，故称LL型，类似分析下同。**

**●调整：先将最小失衡子树为A-B-BL+，所涉及的BL+，B，A按中序升序规律形成的二叉树是**

**再比照调整前的原树中序升序序列，将BR和AR搭配到适当的位置，看下表和下图。**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **树的失衡类型** | **LL（左左）** | | | | |
| **调整前树(升序)** | **BL**  **1** | **B**  **2** | **BR**  **3** | **A**  **(根)**  **4** | **AR**  **5** |
| **最小失衡树(升序)** | **BL+**  **1** | **B**  **(根)**  **2** |  | **A**  **(BR)**  **4** |  |
| **调整后树(升序)** | **BL+**  **1** | **B**  **(根)**  **2** | **BR**  **(AL)**  **3** | **A**  **4** | **AR**  **5** |

**例１：最简单的LL型**

**●插入前是平衡的**

**●插入结点1后失衡（形成LL型）**

**●调整后变平衡**

**例2：插入前是平衡的**

**●插入结点1后失衡，需要调整5-3-2（LL型）**

**●调整后变平衡**

**(2)RR（右右）型**

**●下图插入前原树是平衡的**

**●插入BR+后失衡**

**●调整：最小失衡子树为A-B-BR+，形成RR型，**

**涉及的A， B，BR+，形成的二叉树是**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **树的失衡类型** | **RR（右右）** | | | | |
| **调整前树(升序)** | **AL**  **1** | **A**  **(根)**  **2** | **BL**  **3** | **B**  **4** | **BR**  **5** |
| **最小失衡树(升序)** |  | **A**  **(BL)**  **2** |  | **B**  **(根)**  **4** | **BR+**  **5** |
| **调整后树(升序)** | **AL**  **1** | **A**  **2** | **BL**  **(AR)**  **3** | **B**  **(根)**  **4** | **BR+**  **5** |

**例１：最简单的RR型**

**●插入前是平衡的**

**●插入结点3后失衡（形成RR型）**

**●调整后变平衡**

**例2：插入前是平衡的**

**●插入结点6后失衡，需要调整2-4-5（RR型）**

**●调整后变平衡**

**(3)LR（左右）型**

**●下图插入前原树是平衡的**

**●插入CL+后失衡**

**●调整：最小失衡子树为A-B-C，形成LR型，涉及的B，C，A，形成的二叉树是**

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **树的失衡类型** | **LR（左右）** | | | | | | |
| **调整前树(升序)** | **BL**  **1** | **B**  **2** | **CL**  **3** | **C**  **4** | **CR**  **5** | **A**  **(根)**  **6** | **AR**  **7** |
| **最小失衡树(升序)** |  | **B**  **(CL)**  **2** |  | **C**  **(根)**  **4** |  | **A**  **(CR)**  **6** |  |
| **调整后树(升序)** | **BL**  **1** | **B**  **2** | **CL+**  **(BR)**  **3** | **C**  **(根)**  **4** | **CR**  **(AL)**  **5** | **A**  **6** | **AR**  **7** |

**例1：最简单的LR型**

**●插入前平衡**

**●插入结点2后失衡（形成LR型）**

**●调整后变平衡**

**例2：●插入结点3后失衡，需要调整5-2-4（LR型）**

**●调整后变平衡**

**(4)RL（右左）型**

**●插入前原树是平衡的**

**●插入CR+后失衡**

**●调整：最小失衡子树为A-B-C，形成RL型，涉及的A，C，B，形成的二叉树是**

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **树的失衡类型** | **RL（右左）** | | | | | | |
| **调整前树(升序)** | **AL**  **1** | **A**  **(根)**  **2** | **CL**  **3** | **C**  **4** | **CR**  **5** | **B**  **6** | **BR**  **7** |
| **最小失衡树(升序)** |  | **A**  **(CL)**  **2** |  | **C**  **(根)**  **4** |  | **B**  **(CR)**  **6** |  |
| **调整后树(升序)** | **AL**  **1** | **A**  **2** | **CL**  **(AR)**  **3** | **C**  **(根)**  **4** | **CR+**  **(BL)**  **5** | **B**  **6** | **BR**  **7** |

**例1：最简单的RL型**

**●插入前是平衡的**

**●插入结点2后失衡（形成RL型）**

**●调整后平衡**

**例2：插入结点3后，2-6-4失衡（形成RL型）**

**●调整后平衡**

**例：设给定关键字序列为：4，5，7，2，1，3，6，试生成一棵平衡二叉树（AVL）**

**●首先插入4，显然是平衡的**

**●插入5后还是平衡的**

**●插入7后变成不平衡的，4-5-7（形成RR型）**

**●处理RR型后，变成平衡的**

**●插入2后还是平衡的**

**●插入1后变成不平衡的，最小失衡树选5-4-2形成LL型（尽量多选含非平衡因子的结点），也可选4-2-1。**

**●处理LL型5-4-2后，变成平衡的**

**●插入3后，是平衡的**

**●插入6后，变成不平衡的，5-7-6形成RL型**

**●处理RL型后，变成平衡的，结束。**

**平衡二叉树（AVL）的查找及性能分析：平衡二叉树本身就是一棵二叉排序树，故它的查找与二叉排序树（BST）完全相同，但它的查找性能优于二叉树，不像二叉排序树，会出现最坏的时间复杂度，它的时间复杂度与二叉排序树的最好时间复杂度相同，都为。**

**对上例给定的关键字序列4，5，7，2，1，3，6，对得到的二叉排序树（BST）和平衡二叉树（AVL）进行ASL分析。**

**二叉排序树（BST）如下：**

**ASL(BST)=(1+2\*2+3\*3+4)/7=18/7=2.57**

**平衡二叉树（AVL）如下：**

**ASL(AVL)=(1+2\*2+3\*4)/7=17/7=2.43**

**述评：在二叉排序树的插入和删除中，平衡二叉树的优点是：使树的结构变好，从而提高查找的速度。缺点是：使插入和删除变得复杂，从而降低速度，这是因为不但要插入删除，还要检查最小失衡子树。因此，采用平衡树，适合那种二叉排序树一经建立就很少进行插入删除的情况而主要是查找的应用场合。**

**8.4 哈希（Hashing散列）查找**

**8.4.1 哈希查找和哈希表**

**以前介绍的查找方法都是基于待查关键字与表中元素比较实现查找。理想的情况是，一般总想在得知给定值后，不经查找，马上得到其在表中的位置。有没有这种手段呢？散列法可以部分实现。散列查找是通过构造散列函数来得到待查关键字的地址，理论上讲不需要比较的一种方法。**

**哈希查找又称散列查找，它既是一种查找方法，又是一种存储方法，称为散列存储。散列存储的内存存放形式也称为散列表（又称哈希表）。**

**例如要找关键字为k的元素，则只需求出函数值H(k)，H(k)为给定的散列函数，代表关键字k在存储区中的地址，而存储区为一块连续的内存单元，可用一维数组或链表来表示。**

**例1：设一线性表的关键字集合为{2，4，1，6，3，8，9，7}，假设以一维数组H来存储该线性表，数组的长度为10，则可以按下述方法进行存储，由于线性表的关键字均为一位数字，给定的哈希函数（又称散列函数）H(key)=key，该线性表对应的哈希值如下表：**

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **H(key)** | **0** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9** |
| **key** |  | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9** |

**例2：设关键字集合为{18，75，60，43，54，90，46}，给定的哈希函数（又称散列函数）H(k)=k%13，存储内存从0~15，散列地址计算如下：**

**H(18)=18%13=5**

**H(75)=75%13=10**

**H(60)=60%13=8**

**H(43)=43%13=4**

**H(54)=54%13=2**

**H(90)=90%13=12**

**H(46)=46%13=7**

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **H(key)** | **0** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9** | **10** | **11** | **12** | **13** | **14** | **15** |
| **key** |  |  | **54** |  | **43** | **18** |  | **46** | **60** |  | **75** |  | **90** |  |  |  |

**上述散列表是一种比较理想的情况，即每一个关键字对应一个唯一的地址。**

**例3：已知一个线性表的关键字集合为省、直辖市和自治区的英文拼写名字(“BeiJing”,“ShangHai”,“TianJin”,“HeBei”,“ShanDong”,“XinJiang”,“ShanXi”),用名字的第一个字母的英文序号作为哈希地址值。**

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **key** | **Bei**  **Jing** | **Shang**  **Hai** | **Tian**  **Jin** | **He**  **Bei** | **Shan**  **Dong** | **Xin**  **Jiang** | **Shan**  **Xi** |
| **H(key)** | **2** | **19** | **20** | **8** | **19** | **24** | **19** |

**很明显，H(ShangHai)=H(ShanDong)=H(ShanXi)=19，出现不同的关键字对应同一个地址，即当key1≠key2时，H(key1)=H(key2)，导致关键字无法存储，称这种现象叫冲突（Collision）。**

**所以构造哈希函数的重要考虑应尽量减少这种冲突。**

**8.4.2哈希函数（散列函数）的构造方法**

**构造函数应符合以下条件：**

**(1)能快速计算；**

**(2)具有均匀性。当key关键字集中随机选择时，则H(key)将等概率地分布，可以减少冲突。**

**常用的哈希函数构造有以下几种：**

**8.4.2.1 直接定址法**

**H(k)=a\*k+b, 其中a和b均为常数。**

**这种方法计算简单，不会发生冲突；但关键字分布不连续时，会出现很多空闲单元，将造成存储单元大量浪费。**

**8.4.2.2数字选择法**

**对关键字变化多的、频度大的作为散列函数地址。**

**如：在身份证中，取3位作为地址（第9，15，16位），组成哈希函数:**

**H(110108195201222211)= 522**

**H(110108197812209350)= 793**

**H(110108198510018451)= 884**

**H(110108194202076300)= 463**

**H(110108199907234511)= 945**

**8.4.2.3 平方取中法**

**一个数平方后的中间几位数和数的每一位都相关，所以，可以使用随机分布的关键字得到函数地址。**

**如：取平方的第2到4位作为函数的地址**

|  |  |  |
| --- | --- | --- |
| **关键字** | **关键字的平方** | **函数地址** |
| **0100** | **0010000** | **010** |
| **1100** | **1210000** | **210** |
| **1200** | **1440000** | **440** |
| **1160** | **1370400** | **370** |
| **2061** | **4310541** | **310** |

**8.4.2.4折叠法**

**将关键字分割成位数相同的几部分（最后一部分的位数可以不同），然后取这几部分的叠加和（舍去进位）作为哈希函数地址，称为折叠法。**

**如：某人的身份证号码是：110108195201222211，**

**分成4位一组进行叠加，有：**

**1101+0819+5201+2222+11=9354，则有：**

**H(110108195201222211)=9354为该关键字的哈希函数地址。**

**8.4.2.5余数法**

**假设哈希表的表长为m，将关键字除以p(p<=m)后的余数作为哈希表的地址。f(key)=key%p使用这种方法时，p的选择非常重要，如果选不好，容易发生冲突。通常选p为不大于且最接近m的质（素）数。**

**求余法不仅可以直接对关键字取余，也可以对关键字进行其它运算之后取余。**

**8.4.2.6随机数法**

**选择一个随机函数，取关键字的随机函数值作为他的哈希地址，f(key)=random(key)。**

**8.4.3 处理冲突的方法**

**哈希函数选的再理想，也难免有冲突。处理冲突的方法有开放地址法和链地址法。**

**8.4.3.1开放定址法**

**按照一定的次序，将元素存储到散列表中，若发生冲突再按某种规则寻找下一个空闲的存储单元，以期解决存储问题。**

**在开放定址法中，从发生冲突的散列地址为d的单元起进行查找有多种方法，每一种都对应着一定的查找次序或查找路径，都产生一个确定的探查序列。在多种方法中主要有：线性探查法、平方探查法和双散列函数探查法等。**

**(1)线性探查法**

**最简单的一种探查方法，它从发生冲突的d单元**

**起，依次探查下一个单元（当达到下标为m-1的表尾单元时还未找到空闲单元，下一个探查的单元是下标为0的表首单元，即把散列表看做首尾相接的循环表），直到碰到一个空闲单元或探查完所有单元为止。**

**探查的的序列是: d，d+1，d+2，……，m-1，0，1，2，……，d-1**

**用公式表示为：(d+i)%m (0 ≤ i ≤ m-1)**

**探查的步长为1。**

**递推公式：**

****

**述评：优点是总会找到位置，i在最坏的情况才会取到m-1,一般探查几个单元就会找到空闲位置。缺点是容易在元素冲突附近发生“堆积”现象。**

**例1：将十个关键字序列18，16，24，1，78，19，55，10，11，89，使用线性探查法，依次填入哈希表中。**

**其中：哈希表长度为m=13（0~12），哈希函数为H(key)=key%13,**

**算出哈希函数的值：**

**第1步：H(18)=18%13=5（直接填入）**

**第2步：H(16)=16%13=3（直接填入）**

**第3步：H(24)=24%13=11（直接填入）**

**第4步：H(1)=1%13=1（直接填入）**

**第5步：H(78)=78%13=0（直接填入）**

**第6步：H(19)=19%13=6（直接填入）**

**第7步：H(55)=55%13=3（地址3中已有16，下一地址4中无数据，填入55；冲突1次）**

**第8步：H(10)=10%13=10（直接填入）**

**第9步：H(11)=11%13=11（地址11中已有24，下一地址12中无数据，填入11，冲突1次）**

**第10步：H(89)=89%13=11（地址11中已有24，地址12中已有11，地址0中已有78，地址1中已有1， 地址2中无数据，填入89；冲突发生4次）。**

**关键字和哈希地址对照表：**

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **关键字(key)** | **18** | **16** | **24** | **1** | **78** | **19** | **55** | **10** | **11** | **89** |
| **哈希地址H(key)=key%13** | **5** | **3** | **11** | **1** | **0** | **6** | **3** | **10** | **11** | **11** |

**哈希表形成如下：现场填**

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **哈希地址** | **0** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9** | **10** | **11** | **12** |
| **关键字** |  |  |  |  |  |  |  |  |  |  |  |  |  |
| **查找次数** |  |  |  |  |  |  |  |  |  |  |  |  |  |
| **冲突次数** |  |  |  |  |  |  |  |  |  |  |  |  |  |

**对比**

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **哈希地址** | **0** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9** | **10** | **11** | **12** |
| **关键字** | **78**  **89** | **1**  **89** | **89** | **16**  **55** | **55** | **18** | **19** |  |  |  | **10** | **24**  **11**  **89** | **11**  **89** |
| **查找次数** | **1** | **1** | **5** | **1** | **2** | **1** | **1** |  |  |  | **1** | **1** | **2** |
| **冲突次数** | **0** | **0** | **4** | **0** | **1** | **0** | **0** |  |  |  | **0** | **0** | **1** |

**ASL=16/10=1.60**

**Hashlist.c**

**#define KeyN 10**

**#define Hlen 13**

**#define Max 9999**

**#define h1 key[i]%13**

**typedef int Elemtype;**

**void creatHash(Elemtype key[KeyN],Elemtype HF[Hlen],Elemtype poli[Hlen])**

**{**

**int i,d,PoliTime;**

**for(i=0;i<KeyN;i++)**

**{**

**d=h1;**

**PoliTime=0;**

**while(HF[d]!=Max){d=(d+1)%Hlen;PoliTime++;}**

**HF[d]=key[i];poli[d]=PoliTime;**

**}**

**}**

**void main()**

**{**

**int i,s=0;**

**Elemtype KEY[KeyN]={18,16,24,1,78,19,55,10,11,89};**

**Elemtype HF[Hlen],POLI[Hlen]={0};**

**printf(" Key:");**

**for(i=0;i<KeyN;i++)printf("%4d",KEY[i]);**

**printf("\n\nAddres:");**

**for(i=0;i<Hlen;i++){HF[i]=Max;printf("%4d",i);}**

**printf("\n Hash:");**

**creatHash(KEY,HF,POLI);**

**for(i=0;i<Hlen;i++)**

**if(HF[i]!=Max)printf("%4d",HF[i]);**

**else printf(" ");**

**printf("\n Poli:");**

**for(i=0;i<Hlen;i++)**

**if(HF[i]!=Max){printf("%4d",POLI[i]);s+=POLI[i]+1;}**

**else printf(" ");**

**printf("\n\nASL=%.2f\n\n",(float)s/KeyN);**

**}**

**(2)平方探查法**

**若在地址d发生冲突，下一次探查的地址是:**

**探查序列：d+**

**实际探查序列：d+1 , d+4 , d+9 ,……**

**探查的步长是变动的，即：2i-1（i=1,2,3,……）。**

**递推公式：**

****

**述评：优点是跳跃式散列处理冲突较好避免“堆积”**

**现象。缺点是不能探查到散列表上的所有单元，但至少可以探查到一半单元。**

**例2：将十个关键字序列12，13，25，23，38，34，6，84，91，92，设哈希表长度为m=14，哈希函数H(key)=key%11,使用平方探查法，依次填入哈希表中。**

**关键字和哈希地址对照表：**

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **关键字(key)** | **12** | **13** | **25** | **23** | **38** | **34** | **6** | **84** | **91** | **92** |
| **哈希地址H(key)=key%11** | **1** | **2** | **3** | **1** | **5** | **1** | **6** | **7** | **3** | **4** |

**哈希表形成如下：现场填**

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **存储**  **地址** | **0** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9** | **10** | **11** | **12** | **13** |
| **关键字** |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| **查找**  **次数** |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| **冲突**  **次数** |  |  |  |  |  |  |  |  |  |  |  |  |  |  |

**对比**

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **存储**  **地址** | **0** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9** | **10** | **11** | **12** | **13** |
| **关键字** |  | **12**  **23**  **34** | **13**  **23**  **34** | **25**  **91** | **91**  **92** | **23**  **38**  **34**  **92** | **38**  **6** | **6**  **84** | **84**  **92** |  | **34** |  |  | **92** |
| **查找**  **次数** |  | **1** | **1** | **1** | **2** | **3** | **2** | **2** | **2** |  | **4** |  |  | **4** |
| **冲突**  **次数** |  | **0** | **0** | **0** | **1** | **2** | **1** | **1** | **1** |  | **3** |  |  | **3** |

**ASL=22/10=2.20**

**Hashsqrt.c**

**#define KeyN 10**

**#define Hlen 14**

**#define Max 9999**

**#define h1 key[i]%11**

**typedef int Elemtype;**

**void creatHash(Elemtype key[KeyN],**

**Elemtype HF[Hlen],Elemtype poli[Hlen])**

**{**

**int i,j,d,PoliTime;**

**for(i=0;i<KeyN;i++)**

**{**

**d=h1;**

**PoliTime=0;**

**j=1;**

**while(HF[d]!= Max)**

**{d=(d+2\*j-1)%Hlen;**

**PoliTime++;**

**j++;**

**}**

**HF[d]=key[i];poli[d]=PoliTime;**

**}**

**}**

**void main()**

**{**

**int i,s=0;**

**Elemtype KEY[KeyN]={12,13,25,23,38,34,6,84,91,92};**

**Elemtype HF[Hlen],POLI[Hlen]={0};**

**printf(" Key:");**

**for(i=0;i<KeyN;i++)printf("%5d",KEY[i]);**

**printf("\nAddres:");**

**for(i=0;i<Hlen;i++){HF[i]= Max;printf("%5d",i);}**

**printf("\n Hash:");**

**creatHash(KEY,HF,POLI);**

**for(i=0;i<Hlen;i++)**

**if(HF[i]!= Max)printf("%5d",HF[i]);**

**else printf(" ");**

**printf("\n Poli:");**

**for(i=0;i<Hlen;i++)**

**if(HF[i]!= Max){printf("%5d",POLI[i]);s+=POLI[i]+1;}**

**else printf(" ");**

**printf("\n ASL=%.2f\n\n",(float)s/KeyN);**

**}**

**(3)双散列函数探查法**

**使用两个函数h1和h2，其中h1和前面的h(key)一样，以关键字为自变量，产生一个[0,m-1]之间的数作为散列地址；h2也以关键字为自变量，产生一个[1,m-1]之间的并和m互素的数（即m不能被该数整除）作为探查序列的地址增量（即步长）。**

**双散列函数探查序列递归公式是：**

****

**步长是：h2(k)**

**例：上例h1(key)=key%11, h2(key)=key%9+1**

**注意：11和9互（质）素**

**述评：优点是散列处理冲突较好避免“堆积”现象。缺点是计算时间增加。**

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **关键字(key)** | **12** | **13** | **25** | **23** | **38** | **34** | **6** | **84** | **91** | **92** |
| **哈希地址H1(key)=key%11** | **1** | **2** | **3** | **1** | **5** | **1** | **6** | **7** | **3** | **4** |
| **步长H2(key)=key%9+1** | **4** | **5** | **8** | **6** | **3** | **8** | **7** | **4** | **2** | **3** |

**哈希表形成如下：现场填**

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **存储**  **地址** | **0** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9** | **10** | **11** | **12** | **13** |
| **关键字** |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| **查找**  **次数** |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| **冲突**  **次数** |  |  |  |  |  |  |  |  |  |  |  |  |  |  |

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **存储**  **地址** | **0** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9** | **10** | **11** | **12** | **13** |
| **关键字** |  | **12**  **23**  **34** | **13** | **25**  **91** | **92** | **38**  **91** | **6** | **23**  **84**  **91** |  | **3491** |  | **84**  **91** |  | **91** |
| **查找**  **次数** |  | **1** | **1** | **1** | **1** | **1** | **1** | **2** |  | **2** |  | **2** |  | **6** |
| **冲突**  **次数** |  | **0** | **0** | **0** | **0** | **0** | **0** | **1** |  | **1** |  | **1** |  | **5** |

**ASL=18/10=1.80**

**Hashdouble.c**

**#define KeyN 10**

**#define Hlen 14**

**#define Max 9999**

**#define H1 key[i]%11**

**#define H2 key[i]%9+1**

**typedef int Elemtype;**

**void creatHash(Elemtype key[KeyN],Elemtype HF[Hlen],Elemtype poli[Hlen])**

**{**

**int i,d,h2,PoliTime;**

**for(i=0;i<KeyN;i++)**

**{**

**d=H1;**

**PoliTime=0;**

**h2=H2;**

**while(HF[d]!=Max)**

**{**

**d=(d+h2)%Hlen;PoliTime++;h2++;**

**}**

**HF[d]=key[i];poli[d]=PoliTime;**

**}**

**}**

**void main()**

**{**

**int i,s=0;**

**Elemtype KEY[KeyN]=**

**{12,13,25,23,38,34,6,84,91,92};**

**Elemtype HF[Hlen],POLI[Hlen]={0};**

**printf(" Key:");**

**for(i=0;i<KeyN;i++)printf("%5d",KEY[i]);**

**printf("\n\nAddres:");**

**for(i=0;i<Hlen;i++){HF[i]=Max;printf("%5d",i);}**

**printf("\n Hash:");**

**creatHash(KEY,HF,POLI);**

**for(i=0;i<Hlen;i++)**

**if(HF[i]!=Max)printf("%5d",HF[i]);**

**else printf(" ");**

**printf("\n Poli:");**

**for(i=0;i<Hlen;i++)**

**if(HF[i]!=Max){printf("%5d",POLI[i]);**

**s+=POLI[i]+1;}**

**else printf(" ");**

**printf("\n\nASL=%.2f\n\n",(float)s/KeyN);**

**}**

**9.4.3.2 链地址法**

**拉链法解决冲突的方法是:哈希表中不存放关键字信息，将所有哈希地址相同的关键字拉成一个单链表，将各单链表的头指针存放在哈希表的各项中。**

|  |
| --- |
| **0** |
| **1** |
| **2** |
| **3** |
| **4** |
| **5** |
| **6** |
| **7** |
| **8** |
| **9** |
| **10** |
| **11** |
| **12** |

78

1

55

16

18

19

10

89

11

24

**例1：将关键字序列18，16，24，1，78，19，55，10，11，89，设哈希表长度为m=13，哈希函数H(key)=key%13,使用用链地址法存储并查找关键字。**

**Link.c**

**#define KeyN 10**

**#define ListN 13**

**#define H1 Key[i]%13**

**typedef int ElemType;**

**#include<stdio.h>**

**typedef struct Lnode**

**{**

**ElemType data;**

**struct Lnode \*next;**

**}LN;**

**void creat(int m,ElemType Key[],LN \*HT[])**

**{**

**int i,j;**

**ElemType k;**

**LN \*s;**

**for(i=0;i<m;i++)HT[i]=NULL;**

**for(i=0;i<m;i++)**

**{**

**j=H1;**

**s=(LN \*)malloc(sizeof(LN));**

**s->data=Key[i];**

**s->next=HT[j];**

**HT[j]=s;**

**}**

**}**

**void find(LN \*HT[],ElemType k)**

**{**

**int j=k%13;**

**LN \*p=HT[j];**

**while(p!=NULL&&p->data!=k)p=p->next;**

**if(p!=NULL)printf(" is found!\n");**

**else printf(" isn't found!\n");**

**}**

**void main()**

**{**

**ElemType k;**

**int i;**

**LN \*ht[ListN];**

**ElemType KEY[KeyN]=**

**{18,16,24,1,78,19,55,10,11,89};**

**printf("\nKeys:");**

**for(i=0;i<KeyN;i++)printf("%5d",KEY[i]);**

**creat(ListN,KEY,ht);**

**printf("\n\nInput Key: ");**

**scanf("%d",&k);printf("%d",k);**

**find(ht,k);**

**}**

**9.4.4 哈希表的查找与分析**

**哈希查找的过程与哈希表的构造过程基本一致。根据给定关键字key的值，按照构造哈希表时的哈希公式地址，若表中为空，则查找不成功；若不为空，则和给定的关键字比较，若相等，则查找成功；否则根据构造哈希表时的处理冲突的方法计算出下一地址，直至哈希表中某个位置为“空”（查找不成功）或者表中所填的关键字与给定的值相等（查找成功）。**