第7章 集合与搜索

**一、复习要点**

集合是最基本的抽象数据类型之一。本章讨论了集合的三种存储表示：位数组表示、有序链表表示、并查集。在本章的后半部分，讨论了与集合相关的搜索方法和简单的性能分析方法，包括适用于静态搜索表的顺序搜索和折半搜索及代表动态搜索表的二叉搜索树和AVL树。可以使用扩充的二叉搜索树描述顺序搜索和折半搜索，从而推导出估算搜索效率的公式。静态搜索表在整个程序的运行期间结构不会变化，其搜索效率随着表中对象的个数n不断增长。动态搜索表因各个对象的输入顺序不同，得到的搜索表的形态不同，典型的是二叉搜索树。在具有n个对象的二叉搜索树中，搜索效率最高的是高度最低的二叉搜索树。为确保二叉搜索树始终保持搜索效率最高，必须在输入新的对象时判断二叉搜索树是否“失去平衡”，并进行适当的平衡旋转，使二叉搜索树的高度降到最低。这就是AVL树。在AVL树的讨论中，4种平衡旋转，选择参加平衡旋转的3个结点是关键，必须加以注意。

本章复习的要点是：

**1、**基本知识点

必须理解集合及其表示方法，包括位数组表示、有序链表表示及其相关操作的实现算法集合及其表示。理解并查集实现的方法。理解搜索的概念，理解静态搜索表结构，掌握静态搜索表的顺序搜索和折半搜索算法及其性能分析方法。掌握二叉搜索树的表示、搜索、插入、删除算法及其性能分析方法，掌握AVL树的构造、插入、删除时的调整方法及其性能分析，重点是AVL树的定义、平衡化旋转、AVL树的插入和删除、AVL树的高度。

**2、**算法设计

⮚ 用有序链表表示集合时的求集合的并、交、差的算法

⮚ 并查集中的构造函数、求根及合并算法

⮚ 并查集中根据树的高度和根据树中结点个数进行合并的算法

⮚ 设置监视哨的顺序搜索算法和不设监视哨的顺序搜索算法

⮚ 有序顺序表的顺序搜索算法

⮚ 有序顺序表的折半搜索的递归算法和非递归算法

⮚ 二叉搜索树的搜索、插入和删除算法

⮚ 计算AVL树中指定结点高度的递归算法及利用此算法计算结点平衡因子的算法

**二、难点和重点**

1、集合的概念：集合的基本运算、集合的存储表示

* 用位数组表示集合时集合基本运算的实现
* 用有序链表表示集合时集合基本运算的实现

1. 并查集：并查集定义、并查集的三种基本运算的实现

3、基本搜索方法

* 对一般表的顺序搜索算法(包括有监视哨和没有监视哨)
* 对有序顺序表的顺序搜索算法，包括递归和非递归算法
* 用判定树(即扩充二叉搜索树)描述有序顺序表的顺序搜索，以及平均搜索长度(成功与不成功)的计算。
* 对有序顺序表的折半搜索算法、包括递归和非递归算法
* 用判定树(即扩充二叉搜索树)描述有序顺序表的折半搜索，以及平均搜索长度(成功与不成功)的计算。

4、二叉搜索树

* 动态搜索树与静态搜索树的特性
* 二叉搜索树的定义、二叉搜索树上的递归和非递归搜索算法
* 二叉搜索树搜索时的平均搜索长度(成功与不成功)的计算
* 二叉搜索树的插入与删除算法
* AVL树结点上的平衡因子、AVL树的平衡旋转方法
* 高度为h的AVL树上的最少结点个数与最多结点个数
* AVL树的搜索方法、插入与删除方法(不要求算法)

**三、教材中习题的解析**

7-1 设A = **{** 1, 2, 3 **}**, B = **{** 3, 4, 5 **}**，求下列结果：

(1) A + B (2) A \* B (3) A - B

(4) A.Contains (1) (5) A.AddMember (1) (6) A.DelMember (1) (7) A.Min ( )

【解答】

(1) 集合的并A + B = **{** 1, 2, 3, 4, 5 **}**

(2) 集合的交A \* B = **{** 3 **}**

(3) 集合的差A - B = **{** 1, 2 **}**

(4) 包含A.Contains (1) = 1，表示运算结果为"True"

(5) 增加A.AddMember (1)，集合中仍为**{** 1, 2, 3 **}**，因为增加的是重复元素，所以不加入

(6) 删除A.DelMember (1)，集合中为**{** 2, 3 **}**

(7) 求最小元素A.Min ( )，结果为1

7-2 试编写一个算法，打印一个有穷集合中的所有成员。要求使用集合抽象数据类型中的基本操作。如果集合中包含有子集合，各个子集合之间没有重复的元素，采用什么结构比较合适。

【解答】

集合抽象数据类型的部分内容

**template <class Type> class** Set **{**

//对象: 零个或多个成员的聚集。其中所有成员的类型是一致的, 但没有一个成员是相同的。

**int** Contains ( **const Type** x )**;** //判元素x是否集合**this**的成员

**int** SubSet ( Set <**Type>&** right )**;** //判集合**this**是否集合right的子集

**int operator** == ( Set **<Type>&** right )**;** //判集合**this**与集合right是否相等

**int** Elemtype ( )**;** //返回集合元素的类型

**Type** GetData ( )**;**  //返回集合原子元素的值

**char** GetName ( )**;**  //返回集合**this**的集合名

Set <**Type**>\* GetSubSet ( )**;** //返回集合**this**的子集合地址

Set <**Type**>\* GetNext ( )**;** //返回集合**this**的直接后继集合元素

**int** IsEmpty ( )**;** //判断集合**this**空否。空则返回1, 否则返回0

**};**

**ostream& operator** << ( **ostream&** out, Set <**Type**> t ) **{**

//友元函数, 将集合t输出到输出流对象out。

t.traverse ( out, t )**; return** out**;**

**}**

**void** traverse ( **ostream&** out, Set <**Type**> s ) {

//友元函数, 集合的遍历算法

**if** ( s.IsEmpty ( ) == 0 ) **{**  //集合元素不空

**if** ( s.Elemtype ( ) == 0 ) out << s.GetName ( ) << ‘{’**;** //输出集合名及花括号

else if ( s.Elemtype ( ) == 1 ) **{** //集合原子元素

out << s.GetData ( )**;**  //输出原子元素的值

**if** ( s.GetNext ( ) != NULL ) out << ‘,’**;**

**}**

**else {** //子集合

traverse ( s. GetSubSet ( ) )**;** //输出子集合

**if** ( s.GetNext ( ) != NULL ) out << ‘,’**;**

**}**

traverse ( s.GetNext ( ) )**;** //向同一集合下一元素搜索

**}**

**else** out << ‘}’**;**

**}**

如果集合中包含有子集合，各个子集合之间没有重复的元素，采用广义表结构比较合适。也可以使用并查集结构。

7-3 当全集合可以映射成1到N之间的整数时，可以用位数组来表示它的任一子集合。当全集合是下列集合时，应当建立什么样的映射？用映射对照表表示。

1. 整数0, 1, …, 99

(2) 从n到m的所有整数，n ≤ m

(3) 整数n, n+2, n+4, …, n+2k

(4) 字母 ‘a’, ‘b’, ‘c’, …, ‘z’

(5) 两个字母组成的字符串, 其中， 每个字母取自 ‘a’, ‘b’, ‘c’, …, ‘z’。

【解答】

(1) i → i的映射关系，i = 0, 1, 2, …, 99

(2) i → n-i 的映射关系，i = n, n+1, n+2, …, m

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 0 | 1 | 2 |  | m-n |
| n | n+1 | n+2 | … | m |

(3) i → (i-n)/2 的映射关系，i = n, n+2, n+4, …, n+2k

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 0 | 1 | 2 |  | k |
| n | n+2 | n+4 | … | n+2k |

(4) ord (c) → ord (c) - ord ('a') 的映射关系，c = 'a', 'b', 'c', …, 'z'

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 0 | 1 | 2 |  | 25 |
| 'a' | 'b' | 'c' | … | 'z' |

(5) (ord (c1) - ord ('a') )\*26 + ord (c2) - ord ('a')的映射关系，c1 = c2 = 'a', 'b', 'c', …, 'z'

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 0 | 1 | 2 |  | 675 |
| 'aa' | 'ab' | 'ba' | … | 'zz' |

7-4 试证明：集合A是集合B的子集的充分必要条件是集合A和集合B的交集是A。

【证明】

必要条件：因为集合A是集合B的子集，有A ⊆ B，此时，对于任一x ∈ A，必有x ∈ B，因此可以推得x ∈ A∩B，就是说，如果A是B的子集，一定有A∩B = A。

充分条件：如果集合A和集合B的交集A∩B是A，则对于任一x ∈ A，一定有x ∈ A∩B，因此可以推得x ∈ B，由此可得A ⊆ B，即集合A 是集合B的子集。

7-5 试证明：集合A是集合B的子集的充分必要条件是集合A和集合B的并集是B。

【证明】

必要条件：因为集合A是集合B的子集，有A ⊆ B，此时，对于任一x ∈ A，必有x ∈ B， 它一定在A∪B中。另一方面，对于那些x' ∉ A, 但x' ∈ B的元素，它也必在A∪B中，因此可以得出结论：凡是属于集合B的元素一定在A∪B中，A∪B = B。

充分条件：如果存在元素x ∈ A且x ∉ B，有x ∈ A∪B，但这不符合集合A和集合B的并集A∪B是B的要求。集合的并A∪B是集合B的要求表明，对于任一x ∈ A∪B，同时应有x ∈ B。对于那些满足x' ∈ A的x'，既然x' ∈ A∪B，也应当x' ∈ B，因此，在此种情况下集合A应是集合B的子集。

7-6 设+、\*、-是集合的或、与、差运算，试举一个例子，验证

A + B = (A - B) + (B - A) + A \* B

【解答】

若设集合A = { 1, 3, 4, 7, 9, 15 }，集合B = { 2, 3, 5, 6, 7, 12, 15, 17 }。则

A + B = { 1, 2, 3, 4, 5, 6, 7, 9, 12, 15, 17 }

又A \* B = { 3, 7, 15 }, A – B = { 1, 4, 9 }, B – A = { 2, 5, 6, 12, 17 }

则 (A – B) + (B – A) + A \* B = { 1, 2, 3, 4, 5, 6, 7, 9, 12, 15, 17 }

有 A + B = (A – B) + ( B – A ) + A \* B。

7-7 给定一个用无序链表表示的集合，需要在其上执行operator+( ), operator\*( ), operator- ( ), Contains(x), AddMember (x), DelMember(x), Min( )，试写出它的类声明，并给出所有这些成员函数的实现。

【解答】

下面给出用无序链表表示集合时的类的声明。

**template <class Type> class** Set**;** //用以表示集合的无序链表的类的前视定义

**template <class Type> class** SetNode **{**  //集合的结点类定义

**friend class** SetList<**Type>;**

**private:**

**Type** data**;** //每个成员的数据

SetNode **<Type>** \*link**;** //链接指针

**public:**

SetNode (**const Type&** item ) **:** data (item), link (NULL)**;** //构造函数

**};**

**template <class Type> class** Set **{**  //集合的类定义

**private:**

SetNode <**Type**> \*first, \*last**;** //无序链表的表头指针, 表尾指针

**public:**

SetList ( ) **{** first = last = **new** SetNode <**Type**>(0)**; }** //构造函数

~SetList ( ) **{** MakeEmpty ( )**; delete** first**; }** //析构函数

**void** MakeEmpty ( )**;** //置空集合

**int** AddMember ( **const Type&** x )**;** //把新元素x加入到集合之中

**int** DelMember ( **const Type&** x )**;** //把集合中成员x删去

Set <**Type>& operator** = ( Set <**Type>&** right )**;** //复制集合right到**this**。

Set <**Type> operator** + ( Set <**Type>&** right )**;** //求集合**this**与集合right的并

Set **<Type> operato**r \* ( Set <**Type>&** right )**;**  //求集合**this**与集合right的交

Set <**Type> operator** - ( Set <**Type>&** right )**;**  //求集合**this**与集合right的差

**int** Contains ( **const Type&** x )**;**  //判x是否集合的成员

**int operator** == ( Set <**Type>&** right )**;** //判集合**this**与集合right相等

**Type&** Min ( )**;** //返回集合中的最小元素的值

**}**

1. **operator** + ( )

**template <class Type**> Set <**Type>** Set <**Type> :: operator** + ( Set <**Type>&** right ) **{**

//求集合**this**与集合right的并, 计算结果通过临时集合temp返回，**this**集合与right集合不变。

SetNode <**Type>** \*pb = right.first->link**;** //right集合的链扫描指针

SetNode <**Type>** \*pa, \*pc**;** //**this**集合的链扫描指针和结果链的存放指针

Set <**Type>** temp**;**

pa = first->link**;**  pc = temp.first**;**

**while** ( pa != NULL ) **{** //首先把集合**this**的所有元素复制到结果链

pc->link = **new** SetNode<**Type**> ( pa->data ) **;**

pa = pa->link**;** pc = pc->link**;**

**}**

**while** ( pb != NULL ) **{** //将集合right中元素逐个拿出到**this**中查重

pa = first->link**;**

**while** ( pa != NULL **&&** pa->data != pb->data ) pa = pa->link**;**

**if** ( pa == NULL ) //在集合**this**中未出现, 链入到结果链

**{** pc->link = **new** SetNode<**Type**> ( pa->data )**;** pc = pc->link**; }**

pb = pb->link**;**

**}**

pc->link = NULL**;**  last = pc**;**  //链表收尾

**return** temp**;**

**}**

(2) **operator** \* ( )

**template <class Type>** Set <**Type>** Set <**Type> :: operator** \* ( Set <**Type>&** right ) **{**

//求集合**this**与集合right的交, 计算结果通过临时集合temp返回，**this**集合与right集合不变。

SetNode<**Type**> \*pb = right.first->link**;** //right集合的链扫描指针

Set <**Type**> temp**;**

SetNode <**Type>** \*pc = temp.first**;** //结果链的存放指针

**while** ( pb != NULL ) **{** //将集合right中元素逐个拿出到**this**中查重

SetNode<**Type>** \*pa = first->link**;**  //**this**集合的链扫描指针

**while** ( pa != NULL ) **{**

**if** ( pa->data == pb->data ) //两集合公有的元素, 插入到结果链

{ pc->link = **new** SetNode<**Type**> ( pa->data )**;** pc = pc->link**; }**

pa = pa->link**;**

**}**

pb = pb->link**;**

**}**

pc->link = NULL**;** last = pc**;**  //置链尾指针

**return** temp**;**

**}**

(3) **operator** - ( ),

**template <class Type**> Set <**Type>** Set <**Type> :: operator** - ( Set <**Type>&** right ) {

//求集合**this**与集合right的差, 计算结果通过临时集合temp返回，**this**集合与right集合不变。

SetNode<**Type**> \*pa = first->link**;** //**this**集合的链扫描指针

Set <**Type**> temp**;**

SetNode<**Type>** \*pc = temp->first**;** //结果链的存放指针

**while** ( pa != NULL ) **{** //将集合**this**中元素逐个拿出到right中查重

SetNode <**Type**> \*pb = right.first->link**;** //right集合的链扫描指针

**while** ( pb != NULL **&&** pa->data != pb->data )

pb = pb->link**;**

**if** ( pb == NULL ) //此this中的元素在right中未找到, 插入

**{** pc->link = **new** SetNode <**Type**> ( pa->data )**;** pc = pc->link**; }**

pa = pa->link**;**

**}**

pc->link = NULL**;**  last = pc**;** //链表收尾

**return** temp**;**

**}**

(4) Contains(x)

**template <class Type> int** Set <**Type> ::** Contains ( **const Type&** x ) **{**

//测试函数: 如果x是集合的成员, 则函数返回1, 否则返回0。

SetNode<**Type**> \* temp = first->link**;** //链的扫描指针

**while** ( temp != NULL **&&** temp->data != x ) temp = temp->link**;**  //循链搜索

**if** ( temp != NULL ) **return** 1**;**  //找到, 返回1

**else return** 0**;**  //未找到, 返回0

**}**

(5) AddMember (x)

**template <class Type> int** Set <**Type> ::** AddMember ( **const Type&** x ) **{**

//把新元素x加入到集合之中。若集合中已有此元素, 则函数返回0, 否则函数返回1。

SetNode<**Type**> \* temp = first->link**;** // temp是扫描指针

**while** ( temp != NULL **&&** temp->data != x ) temp = temp->link**;**  /循链扫描

**if** ( temp != NULL ) **return** 0**;**  //集合中已有此元素, 不加

last = last->link = **new** SetNode (x)**;**  //否则, 创建数据值为x的新结点, 链入

**return** 1**;**

**}**

(6) DelMember (x)

**template <class Type> int** Set <**Type> ::** DelMember ( **const Type&** x ) **{**

//把集合中成员x删去。若集合不空且元素x在集合中, 则函数返回1, 否则返回0。

SetNode<**Type**> \* p = first->link**,** \*q = first**;**

**while** ( p != NULL ) **{**

**if** ( p->data == x ) **{** //找到

q->link = p->link**;**  //重新链接

**if** ( p == last ) last = q**;** //删去链尾结点时改链尾指针

**delete** p**;**  **return** 1**;**  //删除含x结点

**}**

**else** **{** q = p**;**  p = p->link**; }** //循链扫描

**return** 0**;** //集合中无此元素

**}**

(7) Min ( )

**template <class Type>** SetNode<**Type**> \* Set <**Type> ::** Min ( ) **{**

//在集合中寻找值最小的成员并返回它的位置。

SetNode<**Type**> \* p = first->link**,** \*q = first->link**;** //p是检测指针, q是记忆最小指针

**while** ( p != NULL ) **{**

**if** ( p->data < q->data ) q = p**;** //找到更小的, 让q记忆它

p = p->link**;** //继续检测

**}**

**return** q**;**

**}**

7-8 设有序顺序表中的元素依次为017, 094, 154, 170, 275, 503, 509, 512, 553, 612, 677, 765, 897, 908。试画出对其进行折半搜索时的二叉搜索树, 并计算搜索成功的平均搜索长度和搜索不成功的平均搜索长度。

【解答】

509

154

677

017

275

553

897

094

170

503

512

612

765

908





7-9 若对有n个元素的有序顺序表和无序顺序表进行顺序搜索, 试就下列三种情况分别讨论两者在等搜索概率时的平均搜索长度是否相同?

(1) 搜索失败;

(2) 搜索成功, 且表中只有一个关键码等于给定值k的对象;

(3) 搜索成功, 且表中有若干个关键码等于给定值k的对象, 要求一次搜索找出所有对象。

【解答】

(1) 不同。因为有序顺序表搜索到其关键码比要查找值大的对象时就停止搜索，报告失败信息，不必搜索到表尾；而无序顺序表必须搜索到表尾才能断定搜索失败。

(2) 相同。搜索到表中对象的关键码等于给定值时就停止搜索，报告成功信息。

(3) 不同。有序顺序表中关键码相等的对象相继排列在一起，只要搜索到第一个就可以连续搜索到其它关键码相同的对象。而无序顺序表必须搜索全部表中对象才能确定相同关键码的对象都找了出来，所需时间就不相同了。

前两问可做定量分析。第三问推导出的公式比较复杂，不再进一步讨论。

7-10 假定用一个循环链表来实现一个有序表，并让指针head指向具有最小关键码的结点。指针current初始时等于head，每次搜索后指向当前检索的结点，但如果搜索不成功则current重置为head。试编写一个函数search(head, current, key)实现这种搜索。当搜索成功时函数返回被检索的结点地址，若搜索不成功则函数返回空指针0。请说明如何保持指针current以减少搜索时的平均搜索长度。

【解答】

current

60

50

40

30

20

10

head

相应的搜索函数可以定义为链表及链表结点类的友元函数，直接使用链表及链表结点类的私有数据成员。

**template<class Type>**

ListNode <**Type**> \* Search ( ListNode<**Type**> \* head, ListNode<**Type**> \***&** current**, Type** key ) **{**

ListNode <**Type**> \* p**,** \* q**;**

**if** ( key < current ) **{** p = head**;** q = current**; }** //确定检测范围, 用p, q指示

**else {** p = current**;** q = head**; }**

**while** ( p != q **&&** p->data < key ) p = p->link**;** //循链搜索其值等于key的结点

**if** ( p->data == key ) **{** current = p**; return** p**; }** //找到, 返回结点地址

**else {** current = head**; return** NULL**; }** //未找到, 返回空指针

**}**

7-11 考虑用双向链表来实现一个有序表，使得能在这个表中进行正向和反向搜索。若指针p总是指向最后成功搜索到的结点，搜索可以从p指示的结点出发沿任一方向进行。试根据这种情况编写一个函数search(head, p, key)，检索具有关键码key的结点，并相应地修改p。最后请给出搜索成功和搜索不成功时的平均搜索长度。

【解答】

p

40

70

∧

∧

60

50

30

20

10

head

**template <class Type>**

DblListNode<**Type**> \* Search ( DblListNode<**Type**> \* head, DblListNode<**Type**> \***&** p, **Type** key ) **{**

//在以head为表头的双向有序链表中搜索具有值key的结点。算法可视为双向链表类和双向链表

//结点类的友元函数。若给定值key大于结点p中的数据, 从p向右正向搜索, 否则, 从p向左反

//向搜索。

DblListNode<**Type**> \* q = p**;**

**if** ( key < p->data ) **{ while** ( q != NULL **&&** q->data > key ) q = q-> lLink**; }** //反向搜索

**else { while** ( q != NULL **&&** q->data < key ) q = q-> rLink**; }** //正向搜索

**if** ( q != NULL **&&** q->data == key ) **{** p = q**; return** p**; }**  //搜索成功

**else return** NULL**;**

**}**

如果指针p处于第i个结点(i = 1, 2, …, n)，它左边有i-1个结点，右边有n-i个结点。找到左边第i-1号结点比较2次，找到第i-2号结点比较3次，…，找到第1号结点比较i次，一般地，找到左边第k个结点比较i-k+1次(k = 1, 2, …, i-1)。找到右边第i+1号结点比较2次，找到第i+2号结点比较3次，…，找到第n号结点比较n-i+1次，一般地，找到右边第k个结点比较k-i+1次(k = i+1, i+2, …, n)。因此，当指针处于第i个结点时的搜索成功的平均数据比较次数为



一般地，搜索成功的平均数据比较次数为



如果指针p处于第i个结点(i = 1, 2, …, n)，它左边有i个不成功的位置，右边有n-i+1个不成功的位置。



一般地，搜索不成功的平均数据比较次数为



7-12 在一棵表示有序集S的二叉搜索树中，任意一条从根到叶结点的路径将S分为3部分：在该路径左边结点中的元素组成的集合S1；在该路径上的结点中的元素组成的集合S2；在该路径右边结点中的元素组成的集合S3。S = S1 ∪ S2 ∪ S3。若对于任意的a ∈ S1, b ∈ S2, c ∈ S3， 是否总有a ≤ b ≤ c？为什么？

【解答】

答案是否定的。举个反例：看下图粗线所示的路径

15

35

50

65

70

45

40

30

25

S1 = { 15 }, S2 = { 25, 30, 35, 45 }, S3 = { 40, 50, 65, 70 }

c = 40 ∈ S3，b = 45 ∈ S2，b ≤ c 不成立。

7-13 请给出下列操作序列运算的结果：Union(1, 2), Union(3, 4), Union(3, 5), Union(1, 7), Union(3, 6), Union(8, 9), Union(1, 8), Union(3, 10), Union(3, 11), Union(3, 12), Union(3, 13), Union(14, 15), Union(16, 17), Union(14, 16), Union(1, 3), Union(1, 14)，要求

(1) 以任意方式执行Union；

(2) 根据树的高度执行Union；

(3) 根据树中结点个数执行Union。

【解答】

(1) 对于union(i, j)，以i作为j的双亲

1

2

15

3

10

9

8

7

0

16

14

13

11

12

6

5

4

(2) 按i和j为根的树的高度实现union(i, j)，高度大者为高度小者的双亲；

7

8

1

2

14

3

10

6

5

4

9

16

15

13

11

12

0

(3) 按i和j为根的树的结点个数实现union(i, j)，结点个数大者为结点个数小者的双亲。

14

0

16

15

1

8

7

2

9

3

4

5

6

10

11

13

12

7-14 有n个结点的二叉搜索树具有多少种不同形态?

【解答】

7-15 设有一个输入数据的序列是 **{** 46, 25, 78, 62, 12, 37, 70, 29 **}**, 试画出从空树起，逐个输入各个数据而生成的二叉搜索树。

【解答】

46

46

46

46

46

46

78

25

25

78

25

78

25

78

25

加25

加46

空树

加78

37

12

62

12

62

62

加37

加12

加62

46

46

78

25

78

25

12

62

37

12

62

37

加29

加70

29

70

70

7-16 设有一个标识符序列{else, public, return, template}，p1=0.05, p2=0.2, p3=0.1, p4=0.05, q0=0.2, q1=0.1, q2=0.2, q3=0.05, q4=0.05，计算W[i][j]、C[i][j]和R[i][j]，构造最优二叉搜索树。

【解答】

将标识符序列简化为 { e, p, r, t }，并将各个搜索概率值化整，有

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | e |  | p |  | r |  | t |  |
|  | p1 = 1 |  | p2 = 4 |  | p3 = 2 |  | p4 = 1 |  |
| q0 = 4 |  | q1 = 2 |  | q2 = 4 |  | q3 = 1 |  | q4 = 1 |

(1) 首先构造只有一个内结点的最优二叉搜索树:

*p*1

*p*2

*p*4

*p*3

*q*4

*q*3

*q*3

*q*2

*q*2

*q*0

*q*1

*q*1

三个矩阵的内容如下:

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | 0 | 1 | 2 | 3 | 4 |  |  | 0 | 1 | 2 | 3 | 4 |  |  | 0 | 1 | 2 | 3 | 4 |
| 0 | 4 | 7 | 15 | 18 | 20 |  | 0 | 0 | 7 | 22 | 32 | 39 |  | 0 | 0 | 1 | 2 | 2 | 2 |
| 1 |  | 2 | 10 | 13 | 15 |  | 1 |  | 0 | 10 | 20 | 27 |  | 1 |  | 0 | 2 | 2 | 2 |
| 2 |  |  | 4 | 7 | 9 |  | 2 |  |  | 0 | 7 | 12 |  | 2 |  |  | 0 | 3 | 3 |
| 3 |  |  |  | 1 | 3 |  | 3 |  |  |  | 0 | 3 |  | 3 |  |  |  | 0 | 4 |
| 4 |  |  |  |  | 1 |  | 4 |  |  |  |  | 0 |  | 4 |  |  |  |  | 0 |

W[i][j] C[i][j] R[i][j]

(2) 构造具有两个内结点的最优二叉搜索树

C[2][4] = 12

T[2][4] = 2

C[1][3] = 20

T[1][3] = 2

C[0][2] = 22

T[0][2] = 2

*p*3

*p*2

*p*2

*q*2

*p*4

*p*3

*p*1

*q*2

*q*1

*q*4

*q*3

*q*3

*q*2

*q*1

*q*0

(3) 构造具有三个内结点的最优二叉搜索树

C[0][3] = 32

T[0][3] = 2

C[1][4] = 27

T[1][4] = 2

*p*2

*p*2

*p*1

*p*3

*q*1

*p*3

*p*4

*q*2

*q*3

*q*2

*q*1

*q*0

*q*4

*q*3

(4) 构造具有四个内结点的最优二叉搜索树

C[0][4] = 39

T[0][4] = 2

*p*2

左子树 T[0][1], 其 C[0][1] = 7

右子树 T[2][4], 其 C[2][4] = 12

*p*1

*p*3

*q*2

*q*1

*q*0

*p*4

*q*4

*q*3

7-17 在二叉搜索树上删除一个有两个子女的结点时，可以采用以下三种方法：

(1) 用左子树TL上具有最大关键码的结点X顶替，再递归地删除X。

(2) 交替地用左子树TL上具有最大关键码的结点和右子树TR上具有最小关键码的结点顶替，再递归地删除适当的结点。

(3) 用左子树TL上具有最大关键码的结点或者用右子树TR上具有最小关键码的结点顶替，再递归地删除适当的结点。可随机选择其中一个方案。

试编写程序实现这三个删除方法，并用实例说明哪一个方法最易于达到平衡化。

【解答】

1. 在被删结点有两个子女时用左子树TL中具最大关键码的结点顶替的算法：

**template<class Type**> BstNode<**Type>** \* BST<**Type> ::** leftReplace ( BstNode<**Type>** \* ptr ) **{**

BstNode<**Type**> \* temp = ptr->leftChild**;**  //进到ptr的左子树

**while** ( temp->rightChild != NULL ) temp = temp->rightChild**;** //搜寻中序下最后一个结点

ptr->data = temp->data**;**  //用该结点数据代替根结点数据

**return** temp**;**

**}**

② 在被删结点有两个子女时用右子树TR中具最小关键码的结点顶替的算法：

**template<class Type>** BstNode<**Type**> \* BST<**Type> ::** rightReplace ( BstNode**<Type>** \* ptr ) **{**

BstNode<**Type**> \* temp = ptr->rightChild**;** //进到ptr的右子树

**while** ( temp->leftChild != NULL ) temp = temp->leftChild**;** //搜寻中序下最后一个结点

ptr->data = temp->data**;** //用该结点数据代替根结点数据

**return** temp**;**

**}**

(1) 用左子树TL上具有最大关键码的结点X顶替，再递归地删除X。

**template <class Type> void** BST<**Type> ::** Remove ( **Type&** x, BstNode<**Type>** \***&** ptr ) **{**

//私有函数：在以ptr为根的二叉搜索树中删除含x的结点。若删除成功则新根通过ptr返回。

BstNode<**Type**> \* temp**;**

**if** ( ptr != NULL )

**if** ( x < ptr->data ) Remove ( x, ptr->leftChild )**;** //在左子树中执行删除

**else if** ( x > ptr->data ) Remove ( x, ptr->rightChild )**;** //在右子树中执行删除

**else if** ( ptr->leftChild != NULL **&&** ptr->rightChild != NULL ) **{**

// ptr指示关键码为x的结点，它有两个子女

temp = leftReplace ( ptr )**;** //在ptr的左子树中搜寻中序下最后一个结点顶替x

Remove ( ptr->data, ptr->rightChild )**;** //在ptr的右子树中删除该结点

**}**

**else {** // ptr指示关键码为x的结点，它只有一个或零个子女

temp = ptr**;**

**if** ( ptr->leftChild == NULL ) ptr = ptr->rightChild**;** //只有右子女

**else if** ( ptr->rightChild == NULL ) ptr = ptr->leftChild**;** //只有左子女

**delete** temp**;**

**}**

**}**

(2) 交替地用左子树TL上具有最大关键码的结点和右子树TR上具有最小关键码的结点顶替，再递归地删除适当的结点。

**template <class Type> void** BST<**Type> ::** Remove ( **Type&** x, BstNode<**Type>** \***&** ptr, **int&** dir ) **{**

//私有函数：在以ptr为根的二叉搜索树中删除含x的结点。若删除成功则新根通过ptr返回。在

//参数表中有一个引用变量dir, 作为调整方向的标记。若dir = 0, 用左子树上具有最大关键码的

//结点顶替被删关键码; 若dir = 1, 用右子树上具有最小关键码的结点顶替被删关键码结点, 在调

//用它的程序中设定它的初始值为0。

BstNode<**Type>** \* temp**;**

**if** ( ptr != NULL )

**if** ( x < ptr->data ) Remove ( x, ptr->leftChild, dir )**;** //在左子树中执行删除

**else if** ( x > ptr->data ) Remove ( x, ptr->rightChild, dir )**;** //在右子树中执行删除

**else if** ( ptr->leftChild != NULL **&&** ptr->rightChild != NULL ) **{**

// ptr指示关键码为x的结点，它有两个子女

**if** ( dir == 0 ) **{**

temp = leftReplace ( ptr )**;** dir = 1**;**

//在ptr的左子树中搜寻中序下最后一个结点顶替x

Remove ( ptr->data, ptr->rightChild, dir )**;** //在ptr的右子树中删除该结点

**} else {**

temp = rightReplace ( ptr )**;** dir = 0**;**

//在ptr的右子树中搜寻中序下第一个结点顶替x

Remove ( ptr->data, ptr->leftChild, dir )**;** //在ptr的左子树中删除该结点

**}**

**}**

**else {** // ptr指示关键码为x的结点，它只有一个或零个子女

temp = ptr**;**

**if** ( ptr->leftChild == NULL ) ptr = ptr->rightChild**;** //只有右子女

**else if** ( ptr->rightChild == NULL ) ptr = ptr->leftChild**;** //只有左子女

**delete** temp**;**

**}**

**}**

(3) 用左子树TL上具有最大关键码的结点或者用右子树TR上具有最小关键码的结点顶替，再递归地删除适当的结点。可随机选择其中一个方案。

#**include** <stdlib.h>

**template <class Type> void** BST<**Type> ::** Remove ( **Type&** x, BstNode<**Type>** \***&** ptr ) **{**

//私有函数：在以ptr为根的二叉搜索树中删除含x的结点。若删除成功则新根通过ptr返回。在

//程序中用到一个随机数发生器rand( ), 产生0~32767之间的随机数, 将它除以16384, 得到0~2

//之间的浮点数。若其大于1, 用左子树上具有最大关键码的结点顶替被删关键码; 若其小于或等

//于1, 用右子树上具有最小关键码的结点顶替被删关键码结点, 在调用它的程序中设定它的初始

//值为0。

BstNode<**Type**> \* temp**;**

**if** ( ptr != NULL )

**if** ( x < ptr->data ) Remove ( x, ptr->leftChild )**;** //在左子树中执行删除

**else if** ( x > ptr->data ) Remove ( x, ptr->rightChild )**;** //在右子树中执行删除

**else if** ( ptr->leftChild != NULL **&&** ptr->rightChild != NULL ) **{**

// ptr指示关键码为x的结点，它有两个子女

**if** ( (**float**) ( **rand** ( ) / 16384 ) > 1 ) **{**

temp = leftReplace ( ptr )**;** //在ptr的左子树中搜寻中序下最后一个结点顶替x

Remove ( ptr->data, ptr->rightChild )**;** //在ptr的右子树中删除该结点

**} else {**

temp = rightReplace ( ptr )**;** //在ptr的右子树中搜寻中序下第一个结点顶替x

Remove ( ptr->data, ptr->leftChild )**;** //在ptr的左子树中删除该结点

**}**

**}**

**else {** // ptr指示关键码为x的结点，它只有一个或零个子女

temp = ptr**;**

**if** ( ptr->leftChild == NULL ) ptr = ptr->rightChild**;** //只有右子女

**else if** ( ptr->rightChild == NULL ) ptr = ptr->leftChild**;** //只有左子女

**delete** temp**;**

**}**

**}**

7-18 (1) 设T是具有n个内结点的扩充二叉搜索树, I是它的内路径长度, E是它的外路径长度。试利用归纳法证明 E = I + 2n, n ≥ 1。

(2) 利用(1)的结果, 试说明: 成功搜索的平均搜索长度Sn与不成功搜索的平均搜索长度Un之间的关系可用公式

Sn = ( 1 + 1/n ) Un - 1, n ≥ 1 表示。【解答】

(1) 用数学归纳法证明。当n = 1时，有1个内结点(I = 0)，2个外结点(E = 2)，满足E = I+2n。设n = k 时结论成立，Ek = Ik + 2k。则当n = k + 1时，将增加一个层次为l的内结点，代替一个层次为l的外结点，同时在第l+1层增加2个外结点，则Ek+1 = Ek – l + 2\*(l+1) = Ek + l +2，Ik+1 = Ik + l，将Ek = Ik + 2k代入，有Ek+1 = Ek + l +2 = Ik + 2k + l +2 = Ik+1+ 2(k +1)，结论得证。

1. 因为搜索成功的平均搜索长度Sn与搜索不成功的平均搜索长度Un分别为



其中，ci是各内结点所处层次，cj是各外结点所处层次。因此有

 (n+1)Un = En = In + 2n = nSn – n +2n = nSn +n



7-19 求最优二叉搜索树的算法的计算时间为O(n3)，下面给出一个求拟最优二叉搜索树的试探算法，可将计算时间降低到O(nlog2n)。算法的思想是对于关键码序列{ keyl, keyl+1, …, keyh }， 轮流以keyk为根，k = l, l+1, …, h，求使得 | W[l-1][k-1] - W[k][h] | 达到最小的k，用keyk作为由该序列构成的拟最优二叉搜索树的根。然后对以keyk为界的左子序列和右子序列，分别施行同样的操作，建立根keyk的左子树和右子树。要求：

(1) 使用7.17题的数据，执行这个试探算法建立拟最优二叉搜索树，该树建立的时间代价是多少？

(2) 编写一个函数，实现上述试探算法。要求该函数的时间复杂度应为O(nlog2n)。

【解答】

(1) 各个关键码的权值为 { p1, p2, p3, p4 }，利用题7-17中的W矩阵，轮流让 k = 1, 2, 3, 4为根，此时，下界l = 1, 上界h = 4。有

min⏐W[l-1][k-1] – W[k][h]⏐ = ⏐W[0][1] – W[2][4]⏐ = 2

求得 k = 2。则根结点为2，左子树的权值为 { p1 }，右子树的权值为 { p3, p4 }。

因为左子树只有一个结点，所以，权值为p1的关键码为左子树的根即可。对于右子树 { p3, p4 }，采用上面的方法，轮流让 k = 3, 4为根，此时，下界l = 3，上界h = 4。有

min⏐W[l-1][k-1] – W[k][h]⏐ = ⏐W[2][2] – W[3][4]⏐ = 1

求得 k = 3。于是以权值为p3的关键码为根，其左子树为空，右子树为 { p4 }。这样，得到拟最优二叉搜索树的结构如下：

*p*2

*p*3

*p*1

*p*4

建立该拟最优二叉搜索树的时间代价为O(4+1+2+1) = O(8)

(2) 建立该拟最优二叉搜索树的算法

**void** nearOptiSrchTree ( **int** W[n+1][n+1], **int** n, **int** left, **int** right ) **{**

**if** ( left > right ) **{ cout** << “Empty Sequence! “ << **endl; return; }**

**if** ( left == right ) **{ cout** << left**; return; }**

**int** p = 0**; int** k**;**

**for** ( **int** j = left**;** j <= right**;** j++ )

**if** ( p > **abs** ( W[left-1][j-1] – W[j][right] )

**{** p = **abs** ( W[left-1][j-1] – W[j][right] )**;** k = j**; }**

**cout** << k**;**

**if** ( k == left ) nearOptiSrchTree ( W[n+1][n+1], n, k+1, right )**;**

**else if** ( k == right ) nearOptiSrchTree ( W[n+1][n+1], n, left, k-1 )**;**

**else {** nearOptiSrchTree ( W[n+1][n+1], n, left, k-1 )**;**

nearOptiSrchTree ( W[n+1][n+1], n, k+1, right )**;**

**}**

**}**

7-20将关键码DEC, FEB, NOV, OCT, JUL, SEP, AUG, APR, MAR, MAY, JUN, JAN 依次插入到一棵初始为空的AVL树中，画出每插入一个关键码后的AVL树，并标明平衡旋转的类型。

【解答】

加JUL

加OCT

加NOV

加FEB

加DEC

**+2**

FEB

FEB

DEC

左单旋

DEC

DEC

DEC

NOV

DEC

NOV

DEC

FEB

FEB

FEB

JUL

OCT

OCT

NOV

NOV

**+2**

NOV

FEB

FEB

加AUG

左单旋

FEB

OCT

NOV

DEC

NOV

DEC

JUL

SEP

DEC

OCT

JUL

OCT

JUL

加SEP

AUG

SEP

SEP

NOV

NOV

NOV

加APR

加MAR

FEB

OCT

OCT

FEB

右单旋

FEB

OCT

**-2**

JUL

SEP

AUG

JUL

AUG

SEP

SEP

DEC

JUL

MAR

APR

DEC

APR

DEC

APR

AUG

加MAY

NOV

NOV

FEB

OCT

FEB

OCT

**+2**

AUG

左单旋

SEP

JUL

AUG

SEP

JUL

MAR

MAY

DEC

DEC

APR

MAR

APR

MAY

**-2**

加JUN

MAR

NOV

NOV

FEB

FEB

OCT

JUL

OCT

AUG

AUG

左右双旋

MAR

SEP

SEP

JUN

MAY

DEC

APR

DEC

JUL

MAY

APR

JUN

MAR

加JAN

FEB

NOV

AUG

JUL

OCT

MAY

JUN

JAN

DEC

APR

SEP

7-21 从第7-20题所建立的AVL树中删除关键码MAY，为保持AVL树的特性，应如何进行删除和调整? 若接着删除关键码FEB，又应如何删除与调整?

【解答】

删除关键码MAY， 调整

MAR

MAR

OCT

FEB

FEB

NOV

SEP

NOV

JUL

AUG

JUL

MAY

AUG

OCT

JAN

JAN

JUN

DEC

APR

SEP

JUN

DEC

APR

删除关键码FEB， 不用调整

MAR

JAN

OCT

NOV

SEP

JUL

AUG

JUN

DEC

APR

7-22 将关键码1, 2, 3, …, 2k-1依次插入到一棵初始为空的AVL树中。试证明结果树是完全平衡的。

【解答】

所谓“完全平衡”是指所有叶结点处于树的同一层次上，并在该层是满的。此题可用数学归纳法证明。

当k = 1时，21-1 = 1，AVL树只有一个结点，它既是根又是叶并处在第0层，根据二叉树性质，应具有20 = 1个结点。因此，满足完全平衡的要求。

设k = n时，插入关键码1, 2, 3, …, 2n-1到AVL树中，恰好每一层(层次号码i = 0, 1, …, n-1)有2i个结点，根据二叉树性质，每一层达到最多结点个数，满足完全平衡要求。则当k = n+1时，插入关键码为1, 2, 3, …, 2n-1, 2n, …, 2n+1-1，总共增加了从2n到2n+1-1的2n+1-1-2n +1 = 2n个关键码，使得AVL树在新增的第n层具有2n个结点，达到该层最多结点个数，因此，满足完全平衡要求。

7-23 对于一个高度为h的AVL树，其最少结点数是多少？反之，对于一个有n个结点的AVL树, 其最大高度是多少? 最小高度是多少?

【解答】

设高度为h（空树的高度为 -1）的AVL树的最少结点数为Nh，则Nh = Fh+3 – 1。Fh是斐波那契数。又设AVL树有n个结点，则其最大高度不超过3 / 2 \* log2 (n + 1)，最小高度为 ⎡log2 ( n+1)⎤ -1。

12 个结点的最大高度

12 个结点的最小高度

**四、其他练习题**

7-24 供选择的答案中选择与下面有关搜索算法的叙述中各括号相匹配的词句，将其编号填入相应的括号内。

(1) 对线性表进行折半搜索时，要求线性表必须（ A ）。

(2) 采用顺序搜索算法搜索长度为n的线性表时，元素的平均搜索长度为（ B ）。

(3) 采用折半搜索算法搜索长度为n的有序表时，元素的平均搜索长度为（ C ）。

(4) 采用折半搜索算法搜索长度为n的有序表时，元素的平均搜索长度应（ D ）对应判定树的最大层次数。

(5) 折半搜索与二叉搜索树（即二叉排序树）的时间性能（ E）。

(6) 顺序搜索算法适合于存储结构为（ F ）的线性表。

供选择的答案

A：① 以数组方式存储 ② 以数组方式存储且结点按关键码有序排列

③ 以链接方式存储 ④ 以链接方式存储且结点按关键码有序排列

B：① n/2 ② n ③ (n+1)/2 ④ (n-1)/2

C：① O(n2) ② O(nlog2n) ③ O(log2n) ④ O(n)

D：① 小于 ② 大于 ③ 等于 ④ 大于等于

E：① 相同 ② 完全不同 ③ 有时不相同

F：① 散列存储 ② 顺序存储或链接存储

③ 压缩存储 ④ 索引存储

【解答】

A．② B．③ C．③ D．④ E．③ F. ②

7-25 填空题

(1) 以折半搜索方法从长度为12的有序表中搜索一个元素时，平均搜索长度为\_\_\_\_\_\_\_\_。

(2) 以折半搜索方法搜索一个线性表时，此线性表必须是\_\_\_\_\_\_\_存储的\_\_\_\_\_\_\_表。

(3) 从有序表（12, 18, 30, 43, 56, 78, 82, 95）中依次折半搜索43和56元素时，其搜索长度分别为\_\_\_\_\_\_\_和\_\_\_\_\_\_\_\_。

(4) 对于折半搜索所对应的判定树，它既是一棵\_\_\_\_\_\_\_\_，又是一棵\_\_\_\_\_\_\_。

(5) 假定对长度n = 50的有序表进行折半搜索，则对应的判定树高度为\_\_\_\_\_\_\_，判定树中前5层的结点数为\_\_\_\_\_\_\_，最后一层的结点数为\_\_\_\_\_\_\_\_。

【解答】

(1) 37/12 (2) 顺序，有序 (3) 1，3

(4) 二叉搜索树，理想平衡树 (5) 5，31，19

7-26 判断题

(1) 任一棵二叉搜索树的平均搜索时间都小于用顺序搜索法搜索同样结点的顺序表的平均搜索时间。

(2) 对于同一组待输入的关键码集合，虽然各关键码的输入次序不同，但得到的二叉搜索树都是相同的。

(3) 对于两棵具有相同关键码集合而形状不同的二叉搜索树，按中序遍历它们得到的序列的各元素的顺序是一样的。

(4) 在二叉搜索树上插入新的结点时，不必移动其他结点，仅需改动某个结点的指针，使它由空变为非空即可。

(5) 在二叉搜索树上删除一个结点时，不必移动其他结点，只要将该结点的双亲结点的相应的指针域置为空即可。

(6) 最优二叉搜索树的任何子树都是最优二叉搜索树。

(7) 在所有结点的权值都相等的情况下，只有最下面两层结点的度数可以小于2，其他结点的度数必须等于2的二叉搜索树才是最优二叉搜索树。

(8) 在所有结点的权值都相等的情况下，具有平衡特性的二叉搜索树一定是最优二叉搜索树。

(9) 最优二叉搜索树一定是平衡的二叉搜索树。

【解答】

(1) × (2) × (3) √ (4) √ (5) × (6) √ (7) √ (8) × (9) √

7-27设有序顺序表中的元素依次为017, 094, 154, 170, 275,503, 509, 512, 553, 612, 677, 765, 897, 908。试画出对其进行顺序搜索时的判定树, 并计算搜索成功的平均搜索长度和搜索不成功的平均搜索长度。

【解答】

509

154

677

275

553

897

094

170

503

512

612

765

908

017





7-28 试仿照折半搜索方法编写一个Fibonacci搜索算法，并针对n = 12情况，画出Fibonacci算法的判定树。

【解答】

Fibonacci数列为F0 = 0, F1 = 1, F2 = 1, F3 = 2, F4 = 3, F5 = 5, F6 = 8, F7 = 13, F8 = 21, F9 = 34, …。Fibonacci搜索就是利用Fibonacci数列来划分搜索区间的搜索方法。下图是一个 n = 12 的Fibnacci数列的树形结构，Fibonacci搜索就是利用Fibonacci树进行搜索的。

1

2

3

4

5

8

6

7

9

10

11

12

设有n个已经排序好的数据，每次搜索必须先找出Fibonacci树的根root和距离值。其原则如下：

第1步，计算满足Fib(a) <= n+1的Fibonacci数Fib(a)。在上例中，n = 12，由于Fib(7) <= 12+1 = 13，所以，a = 7。

第2步，求得树的根root = Fib(a-1) = Fib(6) = 8。与左子树根的距离dictance\_1 = Fib(a-2) = Fib(5) = 5，与右子树根的距离distance\_2 = Fib(a-3) = Fib(4) = 3。

第3步，进行迭代，直到找到满足给定值x的数据或距离等于0为止：

(1) 若根结点的数据值等于给定值x，搜索成功，返回root所指位置，算法结束。

(2) 若根结点的数据值小于给定值x，则必须搜索第Fib(a-1)个数据之后的数据，新的Fibonacci树是原树的右子树。root = root + distance\_2，修改distance\_1和distance\_2：

distance\_1 = distance\_1 – distance\_2，distance\_2 = distance\_2 – distance\_1。

(3) 若根结点的数据值大于给定值x，则必须搜索第Fib(a-1)个数据之前的数据，新的Fibonacci树是原树的左子树。root = root – distance\_1，修改distance\_1和distance\_2：

temp = distance\_1，distance\_1 = distance\_2，distance\_2 = temp – distance\_2。

下面给出非递归的Fibonacci搜索算法：

#**include** "dataList.h"

**template <class Type> int** dataList**<Type> : :** Fib\_Search ( **const Type&** value ) **{**

**int** i = 1, root, dis1, dis2**;**

**while** ( Fib ( i ) < CurrentSize ) i++**;** //确定Fibnacci数的阶数

root = Fib ( i-1 )**;** dis1 = Fib ( i-2 )**;** dis2 = Fib ( i-3 )**;**

**while** ( dis2 != 0 ) **{**

**switch** ( compare ( Element[root].key, value ) ) **{**

**case** '=' **:** **return** root**;** //搜索成功, 返回满足要求的位置

**case** '>' **:** root = root - dis1**; //**左缩搜索区间

temp = dis1**;** dis1 = dis2**;** dis2 = temp – dis2**;**

**break;**

**case** ‘<’ **:** root = root + dst2**;** //右缩搜索区间

dis1 = dis1 – dis2**;** dis2 = dis2 – dis1**;**

**break;**

**}**

**}**

**return** –1**;**

**}**

**template <class Type> char** dataList **<Type> ::** compare ( **Type** a**, Type** b ) **{**

**if** ( a – b == 0 ) **return** ‘=’**;**

**else if** ( a – b < 0 ) **return** ‘<’**;**

**else return** ‘>’**;**

**}**

7-29 假设二叉树存放于二叉链表中，树中结点的关键码互不相同。试编写一个算法，判别给定的二叉树是否二叉搜索树。

【解答】

判断给定二叉树是否二叉搜索树，可以采用递归算法。对于树中所有结点，检查是否左子树上结点的关键码都小于它的关键码，右子树上结点的关键码都大于它的关键码。相应算法如下：

**template <class Type>**

**void** BinaryTree <**Type> ::** binSearchTree ( BinTreeNode**<Type>** \* t, **int&** bs ) **{**

//在以t为根的子树中递归判断该子树是否二叉搜索树。是，bs返回1，否则bs返回0

**if** ( t != NULL ) **{**

**if** ( ( t->leftChild == NULL || t->data > t->leftChild->data ) **&&**

( t->rightChild == NULL || t->data < t->rightChild->data ) ) **{**

bs = 1**;**

binSearchTree ( t->leftChild, bs )**;** //递归到左子树判断

**if** ( bs ) binSearchTree ( t->rightChild, bs )**;** //递归到右子树判断

**}**

**else** bs = 0**;**

**}**

**}**

7-30 回答下列问题：

(1) 直接在二叉搜索树中搜索关键码为key的对象与从中序遍历输出的有序序列中顺序搜索关键码为key的对象，其效率是否相同？

(2) 输入关键码有序序列来构造一棵二叉搜索树，然后对此树进行搜索，试分析其效率。

【解答】

(1) 效率不相同。在二叉搜索树中平均搜索效率高于有序表的顺序搜索。

(2) 其效率等同于有序表的顺序搜索。因为按关键码有序序列构造出来的二叉搜索树是一棵向右倾斜的单支树，失去了二叉搜索树的优点。

7-31 设在一棵二叉搜索树的每个结点中，含有关键码key域和统计相同关键码结点个数的count域，当向该树插入一个元素时，若树中已存在与该元素的关键码相同的结点，则就使该结点的count域增1，否则就由该元素生成一个新结点而插入到树中，并使其count域置为1，试按照这种插入要求编写一个算法。

【解答】

**template <class Type>**

**void** BST **<Type> ::** Insert1 ( **const Type&** value ) **{**

//向二叉搜索树中插入一个元素value, 若树中存在该元素, 则将匹配结点中的count域

//的值加1即可

BstNode**<Type>** \*p = root, \*pr = NULL**; //**p是检测指针, pr是其双亲指针

**while** ( p != NULL ) **{ //**在树中搜索关键码为value的结点

pr = p**;**

**if** ( value == p->data ) **break;**

**else if** ( value < p->data ) p = p->leftChild**;**

**else** p = p->rightChild**;**

**}**

**if** ( p != NULL ) p->count++**;**  //若元素已存在，将其count域的值增1

**else {** //否则建立新结点并插入到合适位置

BstNode**<Type>** \*s = **new** BTreeNode**;**

s->data = value**;** s->count = 1**;**

s->leftChild = s->rightChild = NULL**;**

**if** ( pr == NULL ) root = s**;**  //空树情形

**else if** ( value < pr->data ) pr->leftChild = s**; //**非空树情形, 接在双亲下面

**else** pr->rightChild = s**;**

**}**

**}**

7-32 设有一个关键码的输入序列 **{** 55, 31, 11, 37, 46, 73, 63, 02, 07 **}**,

(1) 从空树开始构造平衡二叉搜索树, 画出每加入一个新结点时二叉树的形态。若发生不平衡, 指明需做的平衡旋转的类型及平衡旋转的结果。

(2) 计算该平衡二叉搜索树在等概率下的搜索成功的平均搜索长度和搜索不成功的平均搜索长度。

【解答】

(1) 构造平衡二叉搜索树的过程

37

46

31

11

55

73

37

46

31

11

55

31

11

55

左右旋

左旋

02

63

11

55

73

37

46

31

07

右旋

73

37

46

31

11

55

63

02

63

55

73

37

46

31

07

11

右左旋

左右旋

(2) 计算在等概率下的搜索成功的平均搜索长度和搜索不成功的平均搜索长度。

ASLsucc = (1/9)\*(1+2\*2+3\*4+4\*2)=25/9

ASLunsucc = (1/10)\*(3\*6+4\*4)=17/5