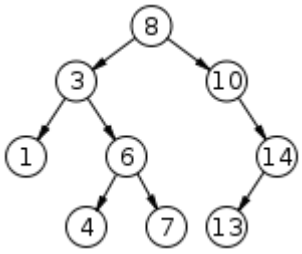
WIKIPEDIA

二元搜尋樹

維基百科，自由的百科全書

二元搜尋樹（英語：Binary Search Tree），也稱為**有序二元樹**（ordered binary tree）或**排序二元樹**（sorted binary tree），是指一棵空樹或者具有下列性質的二元樹：

1. 若任意節點的左子樹不空，則左子樹上所有節點的值均小於它的根節點的值；
2. 若任意節點的右子樹不空，則右子樹上所有節點的值均大於它的根節點的值；
3. 任意節點的左、右子樹也分別為二元搜尋樹；
4. 沒有鍵值相等的節點。



3層二元搜尋樹

二元搜尋樹相比於其他資料結構的優勢在於尋找、插入的時間複雜度較低。為$O(\log n)$。二元搜尋樹是基礎性資料結構，用於構建更為抽象的資料結構，如集合、多重集、關聯陣列等。

二元搜尋樹的尋找過程和次優二元樹類似，通常採取二元連結串列作為二元搜尋樹的儲存結構。中序遍歷二元搜尋樹可得到一個關鍵字的有序序列，一個無序序列可以通過構造一棵二元搜尋樹變成一個有序序列，構造樹的過程即為對無序序列進行尋找的過程。每次插入的新的結點都是二元搜尋樹上新的葉子結點，在進行插入操作時，不必移動其它結點，只需改動某個結點的指標，由空變為非空即可。搜尋、插入、刪除的複雜度等於樹高，期望$O(\log n)$，最壞$O(n)$（數列有序，樹退化成線性表）。

雖然二元搜尋樹的最壞效率是$O(n)$,但它支援動態查詢，且有很多改進版的二元搜尋樹可以使樹高為$O(\log n)$,如SBT,AVL樹，紅黑樹等。故不失為一種好的動態尋找方法。

目錄

二元搜尋樹的尋找演算法

在二元搜尋樹b中尋找x的過程為：

1. 若b是空樹，則搜尋失敗，否則：
2. 若x等於b的根節點的資料域之值，則尋找成功；否則：
3. 若x小於b的根節點的資料域之值，則搜尋左子樹；否則：
4. 尋找右子樹。

```
Status SearchBST(BiTree T, KeyType key, BiTree f, BiTree &p) {
    // 在根指针T所指二元查找树中递归地查找其关键字等于key的数据元素，若查找成功，
    // 则指针p指向该数据元素节点，并返回TRUE，否则指针指向查找路径上访问的最后
    // 一个节点并返回FALSE，指针f指向T的双亲，其初始调用值为NULL
    if (!T) { // 查找不成功
        p = f;
        return false;
    } else if (key == T->data.key) { // 查找成功
        p = T;
        return true;
    } else if (key < T->data.key) // 在左子树中继续查找
        return SearchBST(T->lchild, key, T, p);
    else // 在右子树中继续查找
        return SearchBST(T->rchild, key, T, p);
}
```

在二元搜尋樹插入節點的演算法

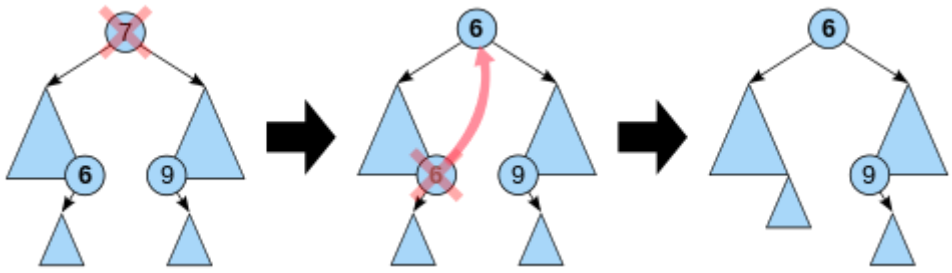
向一個二元搜尋樹b中插入一個節點s的演算法，過程為：

1. 若b是空樹，則將s所指節點作為根節點插入，否則：
2. 若s->data等於b的根節點的資料域之值，則返回，否則：
3. 若s->data小於b的根節點的資料域之值，則把s所指節點插入到左子樹中，否則：
4. 把s所指節點插入到右子樹中。（新插入節點總是葉子節點）

```
/* 当二元搜寻树T中不存在关键字等于e.key的数据元素时，插入e并返回TRUE，否则返回 FALSE */
Status InsertBST(BiTree *T, ElemType e) {
    if (!T) {
        s = new BiTNode;
        s->data = e;
        s->lchild = s->rchild = NULL;
        T = s; // 被插節点*s为新的根结点
    } else if (e.key == T->data.key)
        return false;// 关键字等于e.key的数据元素，返回錯誤
    if (e.key < T->data.key)
        InsertBST(T->lchild, e);  // 將 e 插入左子樹
    else
        InsertBST(T->rchild, e);  // 將 e 插入右子樹
    return true;
}
```

在二元搜尋樹刪除結點的演算法

在二元搜尋樹刪去一個結點，分三種情況討論：



刪除一個有左、右子樹的節點

1. 若*p結點為葉子結點，即PL（左子樹）和PR（右子樹）均為空樹。由於刪去葉子結點不破壞整棵樹的結構，則只需修改其雙親結點的指標即可。
2. 若*p結點只有左子樹PL或右子樹PR，此時只要令PL或PR直接成為其雙親結點*f的左子樹（當*p是左子樹）或右子樹（當*p是右子樹）即可，作此修改也不破壞二元搜尋樹的特性。
3. 若*p結點的左子樹和右子樹均不空。在刪去*p之後，為保持其它元素之間的相對位置不變，可按中序遍歷保持有序進行調整，可以有兩種做法：其一是令*p的左子樹為*f的左/右（依*p是*f的左子樹還是右子樹而定）子樹，*s為*p左子樹的最右下的結點，而*p的右子樹為*s的右子樹；其二是令*p的直接前驅（in-order predecessor）或直接後繼（in-order successor）替代*p，然後再從二元搜尋樹中刪去它的直接前驅（或直接後繼）。

在二元搜尋樹上刪除一個結點的演算法如下：

```
Status DeleteBST(BiTree *T, KeyType key) {
    // 若二叉查找树T中存在关键字等于key的数据元素时，则删除该数据元素，并返回
    // TRUE；否则返回FALSE
    if (!T)
        return false; //不存在关键字等于key的数据元素
    else {
        if (key == T->data.key)   //    找到关键字等于key的数据元素
            return Delete(T);
        else if (key < T->data.key)
            return DeleteBST(T->lchild, key);
        else
            return DeleteBST(T->rchild, key);
    }
}

Status Delete(BiTree *&p) {
    // 该节点为叶子节点，直接删除
    BiTree *q, *s;
    if (!p->rchild && !p->lchild) {
        delete p;
        p = NULL;  // Status Delete(BiTree *&p) 要加&才能使P指向NULL
    } else if (!p->rchild) { // 右子树空则只需重接它的左子树
        q = p->lchild;
        /*
        p->data = p->lchild->data;
        p->lchild=p->lchild->lchild;
        p->rchild=p->lchild->rchild;
        */
        p->data = q->data;
        p->lchild = q->lchild;
        p->rchild = q->rchild;
        delete q;
    } else if (!p->lchild) { // 左子树空只需重接它的右子树
        q = p->rchild;
        /*
        p->data = p->rchild->data;
        p->lchild=p->rchild->lchild;
        p->rchild=p->rchild->rchild;
        */
        p->data = q->data;
        p->lchild = q->lchild;
        p->rchild = q->rchild;
        delete q;
    } else { // 左右子树均不空
        q = p;
        s = p->lchild;
        while (s->rchild) {
            q = s;
            s = s->rchild;
        } // 转左，然后向右到尽头
        p->data = s->data;  // s指向被删结点的"前驱"
        if (q != p)
            q->rchild = s->lchild;  // 重接*q的右子树
        else
            q->lchild = s->lchild;  // 重接*q的左子树
        delete s;
    }
    return true;
}
```

在C語言中有些編譯器不支援為`struct Node`節點分配空間，聲稱這是一個不完全的結構，可使用一個指向該Node的指標為之分配空間。

- 如：`sizeof( Probe )`，Probe作為二元樹節點在typedef中定義的指標。

Python實現：

```
def find_min(self):   # Gets minimum node (leftmost leaf) in a subtree
    current_node = self
    while current_node.left_child:
        current_node = current_node.left_child
    return current_node

def replace_node_in_parent(self, new_value=None):
    if self.parent:
        if self == self.parent.left_child:
            self.parent.left_child = new_value
        else:
            self.parent.right_child = new_value
    if new_value:
        new_value.parent = self.parent

def binary_tree_delete(self, key):
    if key < self.key:
        self.left_child.binary_tree_delete(key)
    elif key > self.key:
        self.right_child.binary_tree_delete(key)
    else: # delete the key here
        if self.left_child and self.right_child: # if both children are present
            successor = self.right_child.find_min()
            self.key = successor.key
            successor.binary_tree_delete(successor.key)
        elif self.left_child:   # if the node has only a *left* child
            self.replace_node_in_parent(self.left_child)
        elif self.right_child:  # if the node has only a *right* child
            self.replace_node_in_parent(self.right_child)
        else: # this node has no children
            self.replace_node_in_parent(None)
```

二元搜尋樹的遍歷

中序遍歷（in-order traversal）二元搜尋樹的Python程式碼：

```
def traverse_binary_tree(node, callback):
    if node is None:
        return
    traverse_binary_tree(node.leftChild, callback)
    callback(node.value)
    traverse_binary_tree(node.rightChild, callback)
```

排序（或稱構造）一棵二元搜尋樹

用一組數值建造一棵二元搜尋樹的同時，也把這組數值進行了排序。其最差時間複雜度為$O(n^2)$。例如，若該組數值經是有序的（從小到大），則建造出來的二元搜尋樹的所有節點，都沒有左子樹。自平衡二元搜尋樹可以克服上述缺點，其時間複雜度為O($n$log n)。一方面，樹排序的問題使得CPU Cache效能較差，特別是當節點是動態記憶體分配時。而堆積排序的CPU Cache效能較好。另一方面，樹排序是最佳的增量排序（incremental sorting）演算法，保持一個數值序列的有序性。

```
def build_binary_tree(values):
    tree = None
    for v in values:
        tree = binary_tree_insert(tree, v)
    return tree

def get_inorder_traversal(root):
    '''
    Returns a list containing all the values in the tree, starting at *root*.
    Traverses the tree in-order(leftChild, root, rightChild).
    '''
    result = []
    traverse_binary_tree(root, lambda element: result.append(element))
    return result
```

二元搜尋樹效能分析

每個結點的C_i為該結點的層次數。最壞情況下，當先後插入的關鍵字有序時，構成的二元搜尋樹蛻變為單支樹，樹的深度為n，其平均尋找長度為$\frac{n+1}{2}$（和順序尋找相同），最好的情況是二元搜尋樹的形態和折半尋找的判定樹相同，其平均尋找長度和$\log_2(n)$成正比（$O(\log_2(n))$）。

二元搜尋樹的最佳化

請參見主條目平衡樹。

1. Size Balanced Tree(SBT)
2. 加權平衡樹(WBT)
3. AVL樹
4. 紅黑樹
5. Treap(Tree+Heap)

這些均可以使尋找樹的高度為$O(\log(n))$

參見

- 跳躍列表

外部連結

- Literate implementations of binary search trees in various languages (http://en.literateprograms.org/Category:Binary_search_tree) on LiteratePrograms
- Binary Tree Visualizer (http://btv.melezinek.cz) (JavaScript animation of various BT-based data structures)
- Kovac, Kubo. Binary Search Trees (Java applet). Korešponden?ný seminár z programovania.
- Madru, Justin. Binary Search Tree. JDServer. 18 August 2009.（原始內容存檔於28 March 2010）. C++ implementation.
- Binary Search Tree Example in Python (http://code.activestate.com/recipes/286239/)
- References to Pointers (C++). 微軟開發者網路. 微軟. 2005. Gives an example binary tree implementation.