准备工作

头文件

<所给的头文件>

非法输入控制

```
std::cin >> n;
while (std::cin.fail())
{
    std::cin.clear();
    std::cin.ignore();
    std::cout << "输入非法!请重新输入一个正确的数字" << std::endl;
    std::cin >> n;
}
```

考虑到二叉树类的函数基本需要递归实现,因此应当对所给的头文件进行修改

PS:如果这个函数不会对树进行修改,则可以不修改头文件

修改方法1:将函数的返回值修改为树结点的指针

修改方法2:将函数的形参修改为树结点的二级指针

函数实现

构造函数(初始化函数):

```
sorttree::sorttree() {
    root = new Node;
    root->left = NULL;
    root->right = NULL;
    length = 0;
}
```

插入函数:

```
sorttree::Node* sorttree::BST_insert(Node* p,ElemType data) {
    if (length == 0) {
        root->value = data;
        length++;
    }
    else if (p == NULL) {
        p = new Node;
        p->value = data;
        p->left = p->right = NULL;
    }
    //数值比当前结点小,放在左子系
    else if (data < p->value)
    {
            p->left=BST_insert(p->left,data);
    }
    //数值比当前结点大,放在右子系
    else if(data > p->value)
    {
            p->right=  BST_insert(p->right, data);
    }
    return p;
}
```

删除函数

思路:如果要删除的只有一个或没有族,则可以用它的族来代替顶替,如果有两个族,则用左族的最右或右族的最左与它交换位置,然后再次删除它,直到出现上述另外两种情况

```
sorttree::Node* sorttree::BST_delete(Node* p, ElemType data) {
    if (root==NULL) {
        std::cout << "请先初始化!" << std::endl;
    }
 if (p)
    {
     if (p->value == data) {
         if (p->left && p->right) {
             // 从左子树中找最大的元素填充删除结点
             Node *temp = p->left;
             Node* father = p;
             while(temp->right) {
                 temp = temp->right;
             }
             p->value = temp->value;
             // 从左子树中删除最大元素 */
             if (father->left == temp)father->left = NULL;
             else {
                 father = father->left;
                 while(father->right->right){
                     father = father->right;
                 }
                 father->right = NULL;
             }
             std::cout << "删除成功" << std::endl;
             delete temp;
         }
         else { /* 被删除结点有一个或无子结点 */
             Node*temp = p;
             if (p->left==NULL)        /* 只有右孩子或无子结点 */
                 p = p->right;
             else
                 p = p->left;
```

```
             delete temp;
             std::cout << "删除成功" << std::endl;
         }
     }
         else if (p->value > data)
             p->left = BST_delete(p->left, data);
         else if(p->value < data)
             p->right= BST_delete(p->right,data);
    }
 else if(!p) std::cout << "删除失败,树中不存在该数据!" << std::endl;
    return p;
}
```

搜索函数:

原理与插入函数类似,数小则左,数大则右

```
Status sorttree::BST_search(Node *p,ElemType data) {
    if (p) {
        if (p->value == data) {
            return true;
        }
        if (p->value < data) {
            return  BST_search(p->right, data);
        }
        if (p->value > data) {
            return  BST_search(p->left, data);
        }
    }
    return false;
}
```

三种递归遍历(前中后):

简单的递归

```
Status sorttree::BST_preorderR(Node*p ,void (*visit)(Node*)) {
    if (root == NULL || length == 0) {
        std::cout << "请确保树中含有数据" << std::endl;
        return false;
    }
    if (p){
        (*visit)(p);//printf
        BST_preorderR(p->left, visit);
        BST_preorderR(p->right, visit);
    }
    return true;
}
```

```cpp
Status sorttree::BST_inorderR(Node*p, void (*visit)(Node*)) {
    if (root==NULL || length == 0) {
        std::cout << "请确保树中含有数据" << std::endl;
        return false;
    }
    if (p) {
        BST_preorderR(p->left, visit);
        visit(p);
        BST_preorderR(p->right, visit);
    }
    return true;
}
```

```cpp
Status sorttree::BST_postorderR(Node*p,void(*visit)(Node*)){
    if (root == NULL||length == 0) {
        std::cout << "请确保树中含有数据" << std::endl;
        return false;
    }
    if(!p)return false;
    BST_postorderR(p->left,visit);
    BST_postorderR(p->right,visit);
    visit(p);
    return true;
}
```

三种非递归遍历(前中后):

采用了栈先进后出的特点,再根据三种遍历的特点对出入栈顺序修改

```cpp
Status sorttree::BST_preorderI(Node*p, void (*visit)(Node*)){
    if (root == NULL||length==0) {
        std::cout << "请确保树中含有数据"<<std::endl;
        return false;
    }
    //首先创建一个栈
    std::stack<Node*> s;
    s.push(p);//将节点入栈
    while (!s.empty()){
        //将当前节点输出
        Node* head = s.top();
        visit(s.top());
        s.pop();
        //由于使用的是栈结构，是一个先进后出的结构，所以将右树先入栈，将左树后入栈，这样输出的就是先输出的就是左树
        if (head->right != NULL) {
            s.push(head->right);
        }
        if (head->left != NULL) {
            s.push(head->left);
        }
    }
    return true;
}
```

```cpp
Status sorttree::BST_inorderI(Node*p, void (*visit)(Node*)){
    if (root == NULL || length == 0) {
        std::cout << "请确保树中含有数据" << std::endl;
        return false;
    }
    //首先创建一个栈
    std::stack<Node*> s;
    while (p != NULL || !s.empty()) {
        if (p != NULL) {
            //先一直入队到左子树完;
            s.push(p);
            p = p->left;
        }
        else {
            //如果当前curr为空，说明左边已经递归完成，出栈，开始输出根节点，
            Node* head = s.top();
            //输出当前元素并出栈
            visit(s.top());
            s.pop();
            //寻找head的右子树，遍历右子树
            //如果head的右子树为空，继续出栈.
            p = head->right;
        }
    }
    return true;
}
```

```cpp
Status sorttree::BST_postorderI(Node*p, void (*visit)(Node*)){
    if (root == NULL||length == 0){ ... }
    //首先创建一个栈
    std::stack<Node*> s;
    Node* head;                          //当前结点
    Node*last = NULL;                    //记录前一次出栈的结点
    s.push(p);
    //如果P不存在左和右，则可以直接出栈它；
    //P存在左或者右，但是其左和右都已被出栈过了，则同样可以直接访问该结点。
    //若非上述两种情况，则将P的右和左依次入栈，
  while (!s.empty())
   {
        head = s.top();
        if (
            (head->left == NULL && head->right == NULL) //该结点为叶结点
            ||
           (last != NULL && (last == head->left || last == head->right))//上一个结点为该结点的右或左
            )
        {
            visit(head);
             s.pop();
            last=head;
        }
        else
        {
             if (head->right != NULL)
                s.push(head->right);
            if (head->left != NULL)
                 s.push(head->left);
        }
   }
    return true;
}
```

层序遍历:

使用队列,以根结点为一开始的队头

将当前队头的左和右入队,之后出队,一直循环直到队列为空:

```cpp
Status sorttree::BST_levelOrder(Node* p, void (*visit)(Node*)) {
    if (root == NULL||length==0) {
        std::cout << "请确保树中含有数据" << std::endl;
        return false;
    }
    std::deque<Node*> line;
    line.push_back(p);
        while (!line.empty()) {
            auto head = line.begin();
            Node* temp = *head;
            visit(temp);
            line.pop_front();
            if (temp->left)line.push_back(temp->left);
            if (temp->right)line.push_back(temp->right);
        }
        return true;
}
```

主交互界面

采用switch case的外围包裹一层do while,实现菜单的多次出现,同时采用清屏函数确保控制台的整洁.

```
do {
    system("pause");
    system("cls");
    { ... }
    std::cin >> n;
    while (std::cin.fail())
    {
        std::cin.clear();
        std::cin.ignore();
        std::cout << "输入非法!请重新输入一个正确的数字" << std::endl;
        std::cin >> n;
    }
    switch (n) {
    case 1: { ... }
    case 2: { ... }
    case 3: { ... }
    case 4: { ... }
    case 5: { ... }
    case 6: { ... }
    case 7: { ... }
    case 8: { ... }
    case 9: { ... }
    case 10: { ... }
    case 0:
        break;
    default:
        std::cout << "请输入一个正确的数字" << std::endl;
        break;
    }
} while (n != 0);
```