```h
//:::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
//        Binary Search Tree with Templated Datatype - Joel Cressy
// NOT TESTED: ADT's in place of templated datatype.
// So far, only tested with NATIVE datatypes. (int, double, char, long, etc.)
//:::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
#pragma once
#include <iostream>
#include <vector>
using namespace std;
template <class T>
class treeNode {
public:
    treeNode();
    treeNode(T);
    T data;
    treeNode* left;
    treeNode* right;
};
template <class T>
class treetype {
public:
    enum sorttypes { PREORDER, INORDER, POSTORDER };
    //modifiers
    treetype();
    treetype(T);
    void ins(T);
    void del(T);
    void setorder(sorttypes);
    //accessors
    bool isEmpty() { return root == NULL; }
    vector<T> toVector();
    int size() { return treesize; }
    T operator[](int i) {
        vector<T> arr;
        if (i > treesize)
        {
            cout << "Index out of bounds" << endl;
            return arr[0];
        }
        treesort(root, arr);
        return arr[i];
    }

private:
    int treesize;
    treeNode<T>* getmax(treeNode<T>*);
    treeNode<T>* ins(treeNode<T>*,T);
    treeNode<T>* del(treeNode<T>*,T);
    bool treesort(treeNode<T>*,vector<T>&);
    treeNode<T> *root;
    sorttypes sortOrder;
};

template<class T>
inline treeNode<T>::treeNode()
{
    data = {};
    left, right = NULL;
}

template<class T>
inline treeNode<T>::treeNode(T newData)
{
    data = newData;
    left, right = NULL;
}

template<class T>
istream& operator >> (istream& is, treetype<T>& a);
```

```
template<class T>
ostream& operator << (ostream& os, treetype<T> a);
template<class T>
treetype<T> operator + (treetype<T> a, treetype<T> b);

template<class T>
inline treetype<T>::treetype()
{
    root = NULL;
    //default sort direction is least-to-greatest with an inorder traversal
    sortOrder = INORDER;
    treesize = 0;
}

template<class T>
inline treetype<T>::treetype(T data)
{
    root = new treeNode<T>(data);
    //default sort direction is least-to-greatest with an inorder traversal
    sortOrder = INORDER;
    treesize = 1;
}

template<class T>
inline void treetype<T>::ins(T data)
{
    if ((root = ins(root, data))) treesize++;
}

template<class T>
inline treeNode<T>* treetype<T>::ins(treeNode<T>*C,T data)
{
    if(C != NULL)
    {
        if (data < C->data)
            C->left = ins(C->left, data);
        else //(data >= C->data)
            C->right = ins(C->right, data);
    }
    else
    {
        //Current node is empty, insert directly here
        C = new treeNode<T>(data);
    }
    return C; //return the current pointer to update the links of the parent(s)
}

template<class T>
inline void treetype<T>::del(T data)
{
    if((root = del(root, data))) treesize--;
}

template<class T>
inline treeNode<T>* treetype<T>::del(treeNode<T> *C, T data)
{
    if (C != NULL)
    {
        //find the node
        if (data < C->data)
            C->left = del(C->left, data);
        else if (data > C->data)
            C->right = del(C->right, data);
        else //continue when data == C->data
        {
            //case: no children
            if (C->right == NULL && C->left == NULL)
            {
                delete C;
```

```
                C = NULL; //will return NULL to parent
            }
            else {
                //case: one child
                if (C->right == NULL) //If left contains data,
                {
                    //store current node's memory address into temp
                    treeNode<T>* temp = C;
                    //will return current node's left pointer to the parent
                    C = C->left;
                    delete temp;
                }
                if (C->left == NULL) //if right contains data,
                {
                    treeNode<T>* temp = C; //same process as above,
                    //but send the current node's right pointer to the parent
                    C = C->right;
                    delete temp;
                }
                //case: two children
                if (C->left != NULL && C->right != NULL)
                {
                    //Get the largest node of the left subtree
                    treeNode<T>* temp = getmax(C->left);
                    //Store that largest number in the current location
                    C->data = temp->data;
                    //Delete the node containing that number by starting in the left s
ubtree
                    C->left = del(C->left, temp->data);
                }
            }
        }
    }
    return C;
}

template<class T>
inline void treetype<T>::setorder(sorttypes a)
{
    //Possible arguments: treetype::PREORDER, treetype::INORDER, treetype::POSTORDER
    sortOrder = a;
}

template<class T>
inline vector<T> treetype<T>::toVector()
{
    vector<T> vec;
    treesort(root, vec);
    return vec;
}

template<class T>
inline treeNode<T>* treetype<T>::getmax(treeNode<T>*C)
{
    if (C == NULL)
        return NULL;
    //The right children will always be the highest number in a BST
    while (C->right != NULL)
        C = C->right;
    return C;
}

template<class T>
inline bool treetype<T>::treesort(treeNode<T>* C, vector<T>& arr)
{
    if (C != NULL)
    {
        if (sortOrder == PREORDER) arr.push_back(C->data);
        treesort(C->left, arr);
```

```
        if (sortOrder == INORDER) arr.push_back(C->data);
        treesort(C->right, arr);
        if (sortOrder == POSTORDER) arr.push_back(C->data);
        return true;
    }
    else
    {
        return false;
    }
}

template<class T>
inline istream & operator >> (istream & is, treetype<T>& a)
{
    T data = {};
    while (!is.eof())
    {
        is >> data;
        a.ins(data);
    }
    return is;
}

template<class T>
inline ostream & operator<<(ostream & os, treetype<T> a)
{
    for (int i = 0; i < a.size(); i++)
        os << a[i];
    return os;
}

template<class T>
inline treetype<T> operator+(treetype<T> a, treetype<T> b)
{
    treetype<T> out;
    for (int i = 0; i < a.size(); i++)
        out.ins(a[i]);
    for (int i = 0; i < b.size(); i++)
        out.ins(b[i]);
    return out;
}
```