```cpp
//
//  09_Day_nine_Binary_Trees.cpp
//  09_Day_nine_Binary_Trees
//
//  Created by Daniel Eftodi on 2022-01-19.
//

#include "09_Day_nine_Binary_Trees.h"

// 1. Binary Trees: Introduction
class Node
{
public:
    class Node *lchild;
    class Node *rchild;
    int data;
};

// Binary Search Tree
class BST
{
private:
    class Node *root;
public:
    BST()
    {
        root = nullptr;
    }

    int Height(class Node *p);

    void iInsert(int key);
    void InOrder(class Node *p);

    class Node *Delete(class Node *p, int key);
    class Node *InPre(class Node *p);
    class Node *InSucc(class Node *p);
    class Node *iSearch(int key);
    class Node *rSearch(class Node *p, int key);
    class Node *rInsert(class Node *p, int key);

    class Node *getRoot()
    {
        return root;
    }

};

void BST::iInsert(int key)
{
    class Node *t = root;
    class Node *p = nullptr;
    class Node *r = nullptr;

    if (root == nullptr)
    {
        p = new class Node;
        p->data = key;
        p->lchild = nullptr;
        p->rchild = nullptr;

        root = p;

        return;
    }

    while (t != nullptr)
    {
        r = t;

        if (key < t->data)
        {
            t = t->lchild;
```

```
        }
        else if (key > t->data)
        {
            t = t->rchild;
        }
        else
        {
            return;
        }
    }

    p = new class Node;
    p->data = key;
    p->lchild = nullptr;
    p->rchild = nullptr;

    if (key < r->data)
    {
        r->lchild = p;
    }
    else
    {
        r->rchild = p;
    }
}

void BST::InOrder(class Node *p)
{
    if (p)
    {
        InOrder(p->lchild);
        std::cout << p->data << ", " << std::flush;
        InOrder(p->rchild);
    }
}

Node *BST::iSearch(int key)
{
    class Node *t = root;

    while (t != nullptr)
    {
        if (t->data == key)
        {
            return t;
        }
        else
        {
            t = t->rchild;
        }
    }
    return nullptr;
}

Node *BST::rInsert(class Node *p, int key)
{
    class Node *t = nullptr;

    if (p == nullptr) {
        t = new class Node;
        t->data = key;
        t->lchild = nullptr;
        t->rchild = nullptr;
        return t;
    }

    if (key < p->data) {
        p->lchild = rInsert(p->lchild, key);

    }
    else if (key > p->data)
    {
        p->rchild = rInsert(p->rchild, key);
    }
```

```
    return p;
}

Node *BST::rSearch(class Node *p, int key)
{
    if (p == nullptr)
    {
        return nullptr;
    }

    if (p->data == key) {
        return p;
    }
    else if (key < p->data)
    {
        return rSearch(p->lchild, key);
    }
    else
    {
        return rSearch(p->rchild, key);
    }
}

Node *BST::InPre(class Node *p)
{
    while (p && p->rchild != nullptr)
    {
        p = p->rchild;
    }

    return p;
}

Node *BST::InSucc(class Node *p)
{
    while (p && p->lchild != nullptr)
    {
        p = p->lchild;
    }

    return p;
}

int BST::Height(class Node *p)
{
    int x;
    int y;

    if (p == nullptr)
    {
        return 0;
    }

    x = Height(p->lchild);
    y = Height(p->rchild);

    // return x + 1 if x is larger then y
    // else return y + 1
    return x > y ? x + 1 : y + 1;
}

Node *BST::Delete(class Node *p, int key)
{
    class Node *q = nullptr;

    if (p == nullptr)
    {
        return nullptr;
    }

    if (p->lchild == nullptr && p->rchild == nullptr)
    {
        if (p == root)
```

```
        {
            root = nullptr;
        }

        // Delete the Node where key is found
        std::cout << "Deleted: " << p->data << std::endl;
        delete p;

        return nullptr;
    }

    if (key < p->data)
    {
        p->lchild = Delete(p->lchild, key);
    }
    else if (key > p->data)
    {
        p->rchild = Delete(p->rchild, key);
    }
    else
    {
        if (Height(p->lchild) > Height(p->rchild))
        {
            q = InPre(p->lchild);
            p->data = q->data;
            p->lchild = Delete(p->lchild, q->data);
        }
        else
        {
            q = InSucc(p->rchild);
            p->data = q->data;
            p->rchild = Delete(p->rchild, q->data);
        }
    }

    return p;
}

int one_main()
{
    class BST bst;

    // Iterative Insert
    bst.iInsert(10);
    bst.iInsert(5);
    bst.iInsert(20);
    bst.iInsert(8);
    bst.iInsert(30);

    // Inorder Traversal
    bst.InOrder(bst.getRoot());
    std::cout << std::endl;

    // Iterative search
    class Node *temp = bst.iSearch(20);

    if (temp != nullptr) {
        std::cout << "iSearch found: " << temp->data << std::endl;
    }
    else
    {
        std::cout << "iSearch: element not found" << std::endl;
    }

    // Recursive search
    temp = bst.rSearch(bst.getRoot(), 20);
    if (temp != nullptr)
    {
        std::cout << "rSearch found: " << temp->data << std::endl;
    }
    else
    {
        std::cout << "rSearch: element not found" << std::endl;
    }
```

```cpp
    // Recursive insert
    bst.rInsert(bst.getRoot(), 50);
    bst.rInsert(bst.getRoot(), 70);
    bst.rInsert(bst.getRoot(), 10);

    // Sort
    bst.InOrder(bst.getRoot());
    std::cout << "\n" << std::endl;

    // Pre and Successor
    BST bs;
    bs.iInsert(5);
    bs.iInsert(2);
    bs.iInsert(8);
    bs.iInsert(7);
    bs.iInsert(9);
    bs.iInsert(1);

    temp = bs.InPre(bs.getRoot());
    std::cout << "InPre: " << temp->data << std::endl;

    temp = bs.InSucc(bs.getRoot());
    std::cout << "InSucc: " << temp->data << std::endl;
    bs.InOrder(bs.getRoot());
    std::cout << "\n" << std::endl;

    // Delete
    bs.Delete(bs.getRoot(), 7);
    bs.InOrder(bs.getRoot());
    std::cout << "\n" << std::endl;

    return 0;
}


// 2. Binary Trees:
int two_main()
{

    return 0;
}


// 3. Binary Trees:

int three_main()
{
    return 0;
}

/* INIT - BEGIN */
int main(int argc, char ** argv){

    int key_pressed = '\0';

    // 1. Binary Trees: Introduction
    printf("[01] Binary Trees: Introduction\n");
    one_main();
    printf("[01] Press any key: ");
    key_pressed = c_getch();
    key_pressed = c_getch();
    printf(" [%d] \n\n", key_pressed);

    // 2. Binary Trees:


    // 3. Binary Trees:



    //Endnig with an extra new line
```

```
    printf("\n");

    return 0;
}
/* INIT - END */


/* IMPLEMENTATION OF ALL FUNCTIONS - BEGIN */

void DBG_LOG(std::string sText,
          std::string sVarA,
          std::string sVarB,
          std::string sVarC,
          std::string sVarD,
          std::string sVarE,
          std::string sVarF,
          std::string sVarG,
          std::string sVarH,
          std::string sVarI,
          std::string sVarJ,
          std::string sVarK,
          std::string sVarL,
          std::string sVarM)
{
#if DEBUG_LOGGING
 printf("%s%s%s%s%s%s%s%s%s%s%s%s%s%s\n",
        sText.c_str(),
        sVarA.c_str(),
        sVarB.c_str(),
        sVarC.c_str(),
        sVarD.c_str(),
        sVarE.c_str(),
        sVarF.c_str(),
        sVarG.c_str(),
        sVarH.c_str(),
        sVarI.c_str(),
        sVarJ.c_str(),
        sVarK.c_str(),
        sVarL.c_str(),
        sVarM.c_str());
#endif
}

/* Read 1 character without echo */
int c_getch(void)
{
    struct termios old, char_new;
    int ch;

    tcgetattr(0, &old);

    char_new = old;
    char_new.c_lflag &= ~ICANON;
    char_new.c_lflag &= ~ECHO;

    tcsetattr(0, TCSANOW, &char_new);

    ch = getchar();

    tcsetattr(0, TCSANOW, &old);

    return ch;
}

/* Read 1 character with echo */
int c_getche(void)
{
    struct termios old, char_new;
    int ch;

    tcgetattr(0, &old);

    char_new = old;
    char_new.c_lflag &= ~ICANON;
```

```
//    char_new.c_lflag &= ~ECHO;

    tcsetattr(0, TCSADRAIN, &char_new);

    ch = getchar();

    tcsetattr(0, TCSADRAIN, &old);
    return ch;
}

/* IMPLEMENTATION OF ALL FUNCTIONS - END */
```