Student Name: Lau Ngoc Quyen
Student ID: 104198996

TenaryTree.h

```
#pragma once

#include <stdexcept>
#include <algorithm>

using namespace std;

template<typename T>
class TernaryTreePrefixIterator;

template<typename T>
class TernaryTree
{
public:

    using TTree = TernaryTree<T>;
    using TSubTree = TTree*;

private:

    T fKey;
    TSubTree fSubTrees[3];

    // private default constructor used for declaration of NIL
    TernaryTree() :
        fKey(T())
    {
        for (size_t i = 0; i < 3; i++)
        {
            fSubTrees[i] = &NIL;
        }
    }

public:

    using Iterator = TernaryTreePrefixIterator<T>;

    static TTree NIL;           // sentinel

    // getters for subtrees
    const TTree& getLeft() const {
        return *fSubTrees[0];
```

```
    }
    const TTree& getMiddle() const {
        return *fSubTrees[1];
    }
    const TTree& getRight() const {
        return *fSubTrees[2];
    }

    // add a subtree
    void addLeft(const TTree& aTTree) { addSubTree(0, aTTree); }
    void addMiddle(const TTree& aTTree) { addSubTree(1, aTTree); }
    void addRight(const TTree& aTTree) { addSubTree(2, aTTree); }

    // remove a subtree, may through a domain error
    const TTree& removeLeft() { return removeSubTree(0); }
    const TTree& removeMiddle() { return removeSubTree(1); }
    const TTree& removeRight() { return removeSubTree(2); }

    ////////////////////////////////////////////////////////////////////////
    // Problem 1: TernaryTree Basic Infrastructure

private:

    // remove a subtree, may throw a domain error [22]
    const TTree& removeSubTree(size_t aSubtreeIndex)
    {
        if (fSubTrees[aSubtreeIndex]->empty())
        {
            throw domain_error("Subtree is NIL");
        }
        if (aSubtreeIndex > 2)
        {
            throw out_of_range("Illegal subtree index");
        }
        const TTree& index = const_cast<TTree&>(*fSubTrees[aSubtreeIndex]);
        fSubTrees[aSubtreeIndex] = &NIL;
        return index;
    }

    // add a subtree; must avoid memory leaks; may throw domain error [18]
    void addSubTree(size_t aSubtreeIndex, const TTree& aTTree)
    {
        if (empty())
        {
            throw domain_error("Operation not supported");
        }
        if (aSubtreeIndex > 2)
        {
            throw out_of_range("Illegal subtree index");
        }
        if (!fSubTrees[aSubtreeIndex]->empty())
        {
            throw domain_error("Subtree is not NIL");
        }
```

```
        fSubTrees[aSubtreeIndex] = const_cast<TTree*>(&aTTree);
    }


public:

    TernaryTree(const T& aKey) :fKey(aKey)
    {
        for (int i = 0; i < 3; i++)
        {
            fSubTrees[i] = &NIL;
        }
    }

    // destructor (free sub-trees, must not free empty trees) [14]
    ~TernaryTree()
    {
        if (!empty())
        {
            for (int i = 0; i < 3; i++)
            {
                if (!fSubTrees[i]->empty())
                {
                    delete fSubTrees[i];
                }
            }
        }
    }

    // return key value, may throw domain_error if empty [2]
    const T& operator*() const
    {
        if (empty())
        {
            throw domain_error("Tree is empty");
        }
        return fKey;
    }

    // returns true if this ternary tree is empty [4]
    bool empty() const { return this == &NIL; }

    // returns true if this ternary tree is a leaf [10]
    bool leaf() const
    {
        for (int i = 0; i < 3; i++)
        {
            if (!fSubTrees[i]->empty()) return false;
        }
        return true;
    }

    // return height of ternary tree, may throw domain_error if empty [48]
    size_t height() const
```

```
{
    if (empty())
    {
        throw domain_error("Operation not supported");
    }
    if (leaf()) return 0;
    size_t height[3] = {};

    for (int i = 0; i < 3; i++)
    {
        height[i] = fSubTrees[i]->empty() ? 0 : fSubTrees[i]->height();
    }
    return *max_element(height, height + 3) + 1;
}

////////////////////////////////////////////////////////////////////////
// Problem 2: TernaryTree Copy Semantics

    // copy constructor, must not copy empty ternary tree
TernaryTree(const TTree& aOtherTTree)
{
    for (int i = 0; i < 3; i++)
    {
        fSubTrees[i] = &NIL;
    }
    *this = aOtherTTree;
}

// copy assignment operator, must not copy empty ternary tree
// may throw a domain error on attempts to copy NIL
TTree& operator=(const TTree& aOtherTTree)
{
    if (this != &aOtherTTree)
        if (!aOtherTTree.empty())
        {
            this->~TernaryTree();
            fKey = aOtherTTree.fKey;
            for (size_t i = 0; i < 3; i++)
            {
                if (!aOtherTTree.fSubTrees[i]->empty())
                {
                    fSubTrees[i] = aOtherTTree.fSubTrees[i]->clone();
                }
                else
                {
                    fSubTrees[i] = &NIL;
                }
            }
        }
        else
        {
            throw domain_error("NIL as source not permitted.");
        }
    return *this;
```

```
    }


    // clone ternary tree, must not copy empty trees
    TSubTree clone() const
    {
        if (empty())
        {
            throw domain_error("NIL as source not permitted.");
        }
        return new TTree(*this);
    }

    ////////////////////////////////////////////////////////////////////////
    // Problem 3: TernaryTree Move Semantics

        // TTree r-value constructor
    TernaryTree(T&& aKey) : fKey(std::move(aKey))
    {
        for (int i = 0; i < 3; i++)
        {
            fSubTrees[i] = &NIL;
        }
    }

    // move constructor, must not copy empty ternary tree
    TernaryTree(TTree&& aOtherTTree)
    {
        for (int i = 0; i < 3; i++)
        {
            fSubTrees[i] = &NIL;
        }
        *this = move(aOtherTTree);
    }

    // move assignment operator, must not copy empty ternary tree
    TTree& operator=(TTree&& aOtherTTree)
    {
        if (this != &aOtherTTree)
        {
            if (!aOtherTTree.empty())
            {
                this->~TernaryTree();
                fKey = std::move(aOtherTTree.fKey);
                for (int i = 0; i < 3; i++)
                {
                    if (!aOtherTTree.fSubTrees[i]->empty()) fSubTrees[i] =
const_cast<TSubTree>(&aOtherTTree.removeSubTree(i));
                    else fSubTrees[i] = &NIL;
                }
            }
            else
            {
                throw std::domain_error("NIL as source not permitted.");
```

```
            }
        }
    }


    ////////////////////////////////////////////////////////////////////////
    // Problem 4: TernaryTree Prefix Iterator

    Iterator begin() const
    {
        return Iterator(this).begin();
    }
    Iterator end() const
    {
        return Iterator(this).end();
    }
};

template<typename T>
TernaryTree<T> TernaryTree<T>::NIL;
```

TenaryTreePrefixIterator.h

```
#pragma once

#include "TernaryTree.h"

#include <stack>

template<typename T>
class TernaryTreePrefixIterator
{
private:
    using TTree = TernaryTree<T>;
    using TTreeNode = TTree*;
    using TTreeStack = std::stack<const TTree*>;

    const TTree* fTTree;                // ternary tree
    TTreeStack fStack;                  // traversal stack

public:

    using Iterator = TernaryTreePrefixIterator<T>;

    Iterator operator++(int)
    {
        Iterator old = *this;
```

```
        ++(*this);

        return old;
    }

    bool operator!=(const Iterator& aOtherIter) const
    {
        return !(*this == aOtherIter);
    }

    ////////////////////////////////////////////////////////////////////////
    // Problem 4: TernaryTree Prefix Iterator

private:

    // push subtree of aNode [30]
    void push_subtrees(const TTree* aNode)
    {
        if (!(*aNode).getRight().empty())
        {
            fStack.push(const_cast<TTreeNode>(&(*aNode).getRight()));
        }
        if (!(*aNode).getMiddle().empty())
        {
            fStack.push(const_cast<TTreeNode>(&(*aNode).getMiddle()));
        }
        if (!(*aNode).getLeft().empty())
        {
            fStack.push(const_cast<TTreeNode>(&(*aNode).getLeft())); 5;
        }
    }

public:

    // iterator constructor [12]
    TernaryTreePrefixIterator(const TTree* aTTree) : fTTree(aTTree), fStack()
    {
        if (!(*fTTree).empty())
        {
            fStack.push(const_cast<TTreeNode>(fTTree));
        }
    }

    // iterator dereference [8]
    const T& operator*() const
    {
        return **fStack.top();
    }

    // prefix increment [12]
    Iterator& operator++()
    {
        TTreeNode lPopped = const_cast<TTreeNode>(fStack.top());
```

```
        fStack.pop();
        push_subtrees(lPopped);
        return *this;
    }

    // iterator equivalence [12]
    bool operator==(const Iterator& aOtherIter) const
    {
        return fTTree == aOtherIter.fTTree && fStack.size() ==
aOtherIter.fStack.size();
    }

    // auxiliaries [4,10]
    Iterator begin() const
    {
        Iterator temp = *this;
        temp.fStack = TTreeStack();
        temp.fStack.push(const_cast<TTreeNode>(temp.fTTree));
        return temp;
    }
    Iterator end() const
    {
        Iterator temp = *this;
        temp.fStack = TTreeStack();
        return temp;
    }
};
```

Problem 5 **(50 marks)**

Answer the following questions in one or two sentences:

a. How can we construct a tree where all nodes have the same degree? [4]

5a)

Set the max number of nodes:
- Max number of nodes at x levels = (2 ^ x) when x >= 0
- Max number of nodes in a tree of y height = ((2 ^ 2) * y) - 1

b. What is the difference between l-value and r-value references? [6]

5b)

- An l-value denotes a memory location that identifies an object
- An r-value denotes a temporary value that does not have a presistent memory location.

c. What is a key concept of an abstract data types? [4]

5c)

- An object's behavior can be described by a set of values and actions, known as an abtract data type (ADT). An ADT specifies the actions to be performed without detailing how they are implemented. It abstracts away the algorithms and data structures used, providing an implementation-independent view. This abstraction allows for flexibility in software design.

d. How do we define mutual dependent classes in C++? [4]

5d)

We can define mutual dependent classes in C++ by using forward declarations. Forward declairations are used to delcare a class before it is defined.This allow us to define classes that depend on each other in circular manner.

For example:

```
Class B;
Class A{
   B* b;
};
Class B{
   A* a;
};
```

e. What must a value-based data type define in C++? [2]

5e)

A value-based data types in C++ must define:
- Copy Constructor: to create a new object as a copy of an existing object
- Copy Assignment Operator: To assign the value Of one object to another existing object
- Destructor: To properly clean up resources when an object is destroyed

f. What is an object adapter? [6]

5f)

An object adapter is design pattern that allows incompatible interfaces to work together by wrapping an object. It acts as a bridge between two interfaces, enabling them to communicate

For example:

```cpp
#include <bits/stdc++.h>
using namespace std;
typedef long long ll;


// Old interface
class OldSystem {
public:
    void oldMethod() {
        std::cout << "Old system method called" << std::endl;
    }
};

// New interface
class NewSystem {
public:
    virtual void newMethod() = 0;
};

// Adapter class
class Adapter : public NewSystem {
private:
    OldSystem& oldSystem;
public:
    Adapter(OldSystem& os) : oldSystem(os) {}
    void newMethod() override {
        oldSystem.oldMethod();
    }
};

int main() {
    OldSystem oldSystem;
    Adapter adapter(oldSystem);
    adapter.newMethod(); // Calls oldMethod() on oldSystem
    return 0;
}
```

In this example, the Adapter class allows the OldSystem to be used where the NewSystem interface is expected by adapting the oldMethod() to the newMethod() interface.

g. What is the difference between copy constructor and assignment operator and how do we guarantee safe operation? [8]

5g)

-Copy constructor is used to create a new object as a copy of a existing object. It is called when a object is initialized from an existing object, either through direct initialization or copy initialization

-Assgnment operator is called when an already initialized object is assigned a new value from another existing object. It does not allocate a new memory block; it updates the existing object's state. It can be overloaded and if it not, a default bitwise copy is performed by the complier

-To ensure safe operation, both should perform a deep copy to handle dynamic memory coorectly.

h. What is the best-case, average-case, and worse-case for a lookup in a binary tree? [6]

5h)

Best Case O(1): if the node is the root node
Avarage Case O(log n): where n is the number of node in the tree
Worst Case O(n): if the tree is unbalanced and the node is the leaf node

i. What are reference data members and how do we initialize them? [2]

5i) A reference data member is a reference that must be initialized in the constructor's initialization list, cannot be NULL and cannot be rebound to another object after initialization. It must be bound to a valid object at the time of oject contruction and remains referring to the same object throghout the object's lifetime.

j. You are given n-1 numbers out of n numbers. How do we find the missing number n_k, $1 \leq k \leq n$, in linear time? [8]

5j)

For example:

```cpp
#include <bits/stdc++.h>
using namespace std;
typedef long long ll;


int sum_expected(int n, int sum){
    for(int i = 0; i <= n; i++){
        sum += i;
    }
    return sum;
}


int main(){
    vector<int> v = {1, 2, 3, 5};
    int n = 5;
    int sum_Expected = 0;
    int sum_temp = 0;
    for(auto it : v){
        sum_temp += it;
    }
    sum_Expected = sum_expected(n, sum_Expected);
    int missing = sum_Expected - sum_temp;
    cout << missing << endl;
// OUTPUT: 4 ( the missing number )
}
```

Explaination:
First implement a sum_expected function to caculate the 'n' sum that not missing any value

Second, we have a vector contain a number from 1 to 5 but missing 1 number is 4

We range-based for loop sum up all of the value in the vector

Finally, we minus the sum_expect with sum_temp to have the missing number.

The example code only use two loop with is O(n)=> linear time