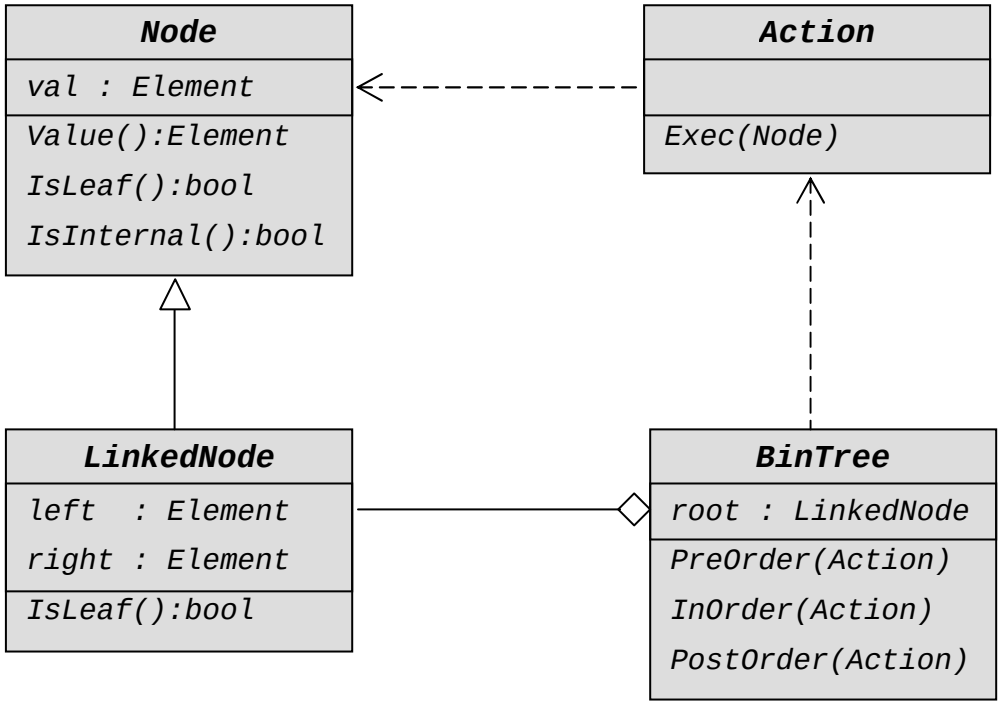
Feladat

Készítsünk egy bináris fa-típust! A típusnak támogatnia kell a fa pre-, in- és postorder bejárását! Egy bejárásnál paraméterként lehessen megadni azt a tevékenységet, ami a bejárás során a csúcsokon végre kell hajtani!

Megoldás

A feladat megoldásához több egymáshoz szorosan kapcsolódó osztály-sablont hozunk létre. A `BinTree` osztály egy láncoltan ábrázolt bináris fa adatszerkezeten alapuló típust valósít meg. A csúcsok ábrázolás független definícióját a `Node` osztály biztosítja. A fa megvalósításában az ebből az osztályból származtatott `LinkedNode` osztályt fogjuk felhasználni. A fa csúcsain elvégezhető tevékenységeket olyan objektumok testesítik meg, amelyek típusát az `Action` absztrakt osztályból tudjuk majd származtatni.



A program két részből tevődik össze. A `bintree.h` állományban helyezzük el a fent vázolt osztály-sablonok definícióját és a metódusok implementációit. A `fo.cpp` állomány tartalmazza a tesztprogramot.

Megoldás C++-ban

bintree.h

A bináris fa osztály-sablonjának publikus része:

```
template <class Element>
class BinTree{
public:
    BinTree():root(nil){srand(time(NULL));}
    ~BinTree();

    void RandomInsert(const Element& e);
    void PreOrder (Action<Element> *todo){Pre (root, todo);}
    void InOrder  (Action<Element> *todo){In  (root, todo);}
    void PostOrder(Action<Element> *todo){Post(root, todo);}

    enum Exceptions{NOROOT};

    Element RootValue()
    {
        if( root==nil ) throw NOROOT;
        return root->Value();
    }
```

A csúcsok absztrakt osztály-sablonja:

```
template <class Element>
class Node {
public:
    const Element& Value() const {return val;}
    virtual bool  IsLeaf() const = 0;
    bool IsInternal() const {return !IsLeaf();}

protected:
    Node(const Element &v): val(v){}
    Element val;
};
```

A bináris fa reprezentációjára használt láncolt csúcs-típus, amelyet a bináris fa osztályába ágyazunk be:

```
class LinkedNode: public Node<Element>{
friend class BinTree;
public:
    LinkedNode(const Element& v, LinkedNode *l, LinkedNode *r):
          Node<Element>(v), left(l), right(r){}
    virtual bool IsLeaf()const{return left==nil && right==nil;}

private:
    LinkedNode *left;
    LinkedNode *right;
};
```

A bináris fát véletlenszerűen felépítő metódus implementációja (a véletlenszám generátorhoz a <cstdlib> könyvtárt kell használni, kezdeti értékbeállításához pedig srand() hívást el kell helyezni a bináris fa konstruktorában):

```
void BinTree<Element>::RandomInsert(const Element& e)
{
    if(root==nil)root = new LinkedNode(e,nil,nil);
    else {
        LinkedNode *r = root;
        int d = rand();
        while(d&1 ? r->left!=nil : r->right!=nil){
            if(d&1) r = r->left;
            else    r = r->right;
            d = rand();
        }
        if(d&1) r->left  = new LinkedNode(e,nil,nil);
        else    r->right = new LinkedNode(e,nil,nil);
    }
}
```

A bináris fa osztály-sablonjának rejtett része:

```
protected:
    LinkedNode *root;
    void Pre (LinkedNode *r, Action<Element> *todo);
    void In  (LinkedNode *r, Action<Element> *todo);
    void Post(LinkedNode *r, Action<Element> *todo);
};
```

A tevékenység objektumok absztrakt osztály-sablonja:

```
template <class Element>
class Action{
public:
    virtual void Exec(Node<Element> *node)=0;
};
```

A bináris fa bejáró műveleteinek implementációja:

```
template <class Element>
void BinTree<Element>::Pre(LinkedNode *r,Action<Element> *todo)
{
    if(r==nil) return;
    todo->Exec(r);
    Pre(r->left, todo);
    Pre(r->right, todo);
}
```

```
template <class Element>
void BinTree<Element>::In(LinkedNode *r,Action<Element> *todo)
{
    if(r==nil) return;
    In(r->left, todo);
    todo->Exec(r);
    In(r->right, todo);
}

template <class Element>
void BinTree<Element>::Post(LinkedNode *r,Action<Element> *todo)
{
    if(r==nil) return;
    Post(r->left, todo);
    Post(r->right, todo);
    todo->Exec(r);
}
```

A bináris fa destruktorának implementációja:

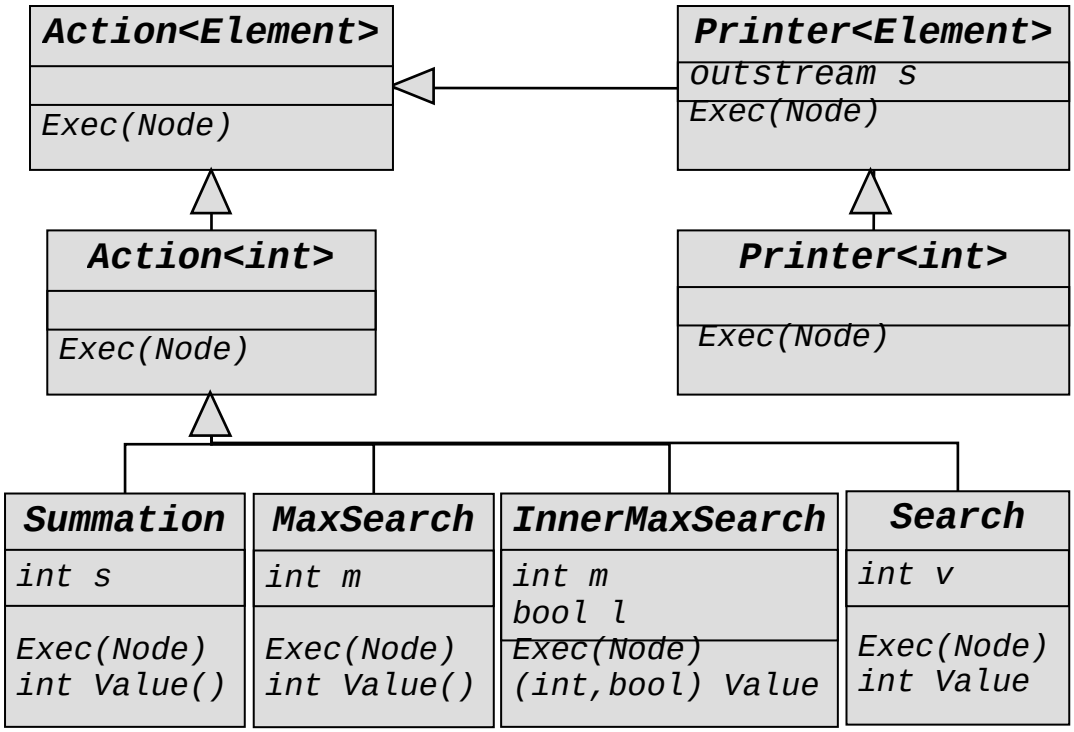
```
template <class Element>
BinTree<Element>::~BinTree()
{
    DelAction del;
    Post(root, &del);
}
```

A bináris fa destruktorában használt tevékenység objektum osztályának definíciója, amelyet a bináris fa osztály-sablonjába ágyazunk:

```
class DelAction: public Action<Element>{
public:
    void Exec(Node<Element> *node){delete node;}
};
```

test.cpp

Többféle tevékenység-objektumnak definiáljuk az osztályát.



Egy csúcs értékét írja ki a `Printer` osztály:

```
template <class Element>
class Printer: public Action<Element>{
public:
    Printer(ostream &o): s(o){};
    void Exec(Node<Element> *node)
        {s << '['<< node->Value() << ']';}
private:
    ostream& s;
};
```

A csúcsokban tárolt egész számok összegét számítja ki a `Summation` tevékenység-osztály:

```
class Summation: public Action<int>{
public:
    Summation(): s(0){}
    void Exec(Node<int> *node){s+=node->Value();}
    int Value(){return s;}
private:
    int s;
};
```

A csúcsokban tárolt egész számok maximumát a `MaxSearch` tevékenység-osztály számolja:

```
class MaxSearch: public Action<int>{
public:
    MaxSearch(int s): m(s){}
    void Exec(Node<int> *node)
        {m = m>node->Value() ? m : node->Value();}
    int Value(){return s;}
private:
    int m;
};
```

A belső csúcsokban tárolt egész számok maximumát az `InnerMaxSearch` tevékenység-osztály adja:

```
class InnerMaxSearch: public Action<int>{
public:
    struct Result {
        int m;
        bool l;
    };
    InnerMaxSearch(){r.m = 0; r.l = false;}
    void Exec(Node<int> *node)
    {
        if(node->IsInternal()){
            if(!r.l){
                r.l = true;
                r.m = node->Value();
            }else{
                if(node->Value()>r.m){r.m = node->Value();}
            }
        }
    }
    Result Value(){return r;}
private:
    Result r;
};
```

A csúcsokban tárolt értékek közül egy páros számot kereső `Search` tevékenység osztály:

```
class Search: public Action<int>{
    int v;
public:
    enum Exceptions{FOUND};
    Search(){};
    void Exec(Node<int> *node)
    {
        if(node->Value()%2==0){
            v = node->Value();
            throw FOUND;
        }
    }
    int Value(){return v;}
};
```

A típusok szemléltetése céljából beolvasunk számokat a standard bemenetről és véletlenszerűen felépítünk belőlük egy bináris fát. Ezután kiírjuk a standard kimenetre különféle bejárási stratégiák mellett a csúcsok értékeit, majd meghatározzuk a csúcsokban tárolt értékek összegét, maximumát, a belső csúcsok maximumát, végül keresünk a csúkban egy páros számot!

```
int main()
{
    BinTree<int> t;
    int i;
    cin >> i;
    while(i!=0){
        t.RandomInsert(i);
        cin >> i;
    }

    Printer<int> print(cout);
    cout << "Preorder bejárás:";
    t.PreOrder(&print);
    cout << endl;
    cout << "Inorder bejárás:";
    t.InOrder(&print);
    cout << endl;
    cout << "Postorder bejárás:";
    t.PostOrder(&print);
    cout << endl;

    Summation sum;
    t.PreOrder(&sum);
    cout << "Fa elemeinek összege:" << sum.Value() << endl;
    try{
        MaxSearch max(t.RootValue());
        t.PreOrder(&max);
        cout << "Maximális érték:" << max.Value();
    }catch(BinTree::Exceptions e){
        if( e==BinTree::NOROOT ) cout << "Nincs elem a fában!";
    }
    cout << endl;
    InnerMaxSearch inmax;
    t.PreOrder(&inmax);
    cout << "A fa belső csúcsainak maximális értéke:";
    if(inmax.Value().l) cout << inmax.Value().m << endl;
    else                cout << "Nincs belső csúcs" << endl;
    Search se;
    try{
        t.PreOrder(&se);
        cout << "Nincs páros szám!" << endl;
    }catch(Search::Exceptions e){
        cout << "Első páros szám:" << se.Value() << endl;
    }

  return 0;
}
```