Származtatás és objektum összetétel

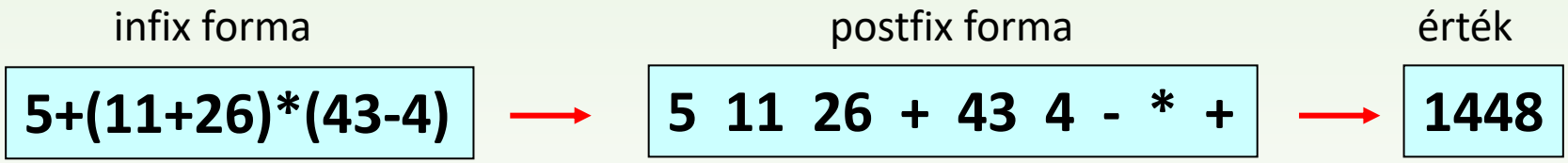
Gregorics Tibor

gt@inf.elte.hu

http://people.inf.elte.hu/gt/oep

Feladat

Alakítsunk át egy infix formájú aritmetikai kifejezést postfix formájúra (RPN), és számoljuk ki az értékét.



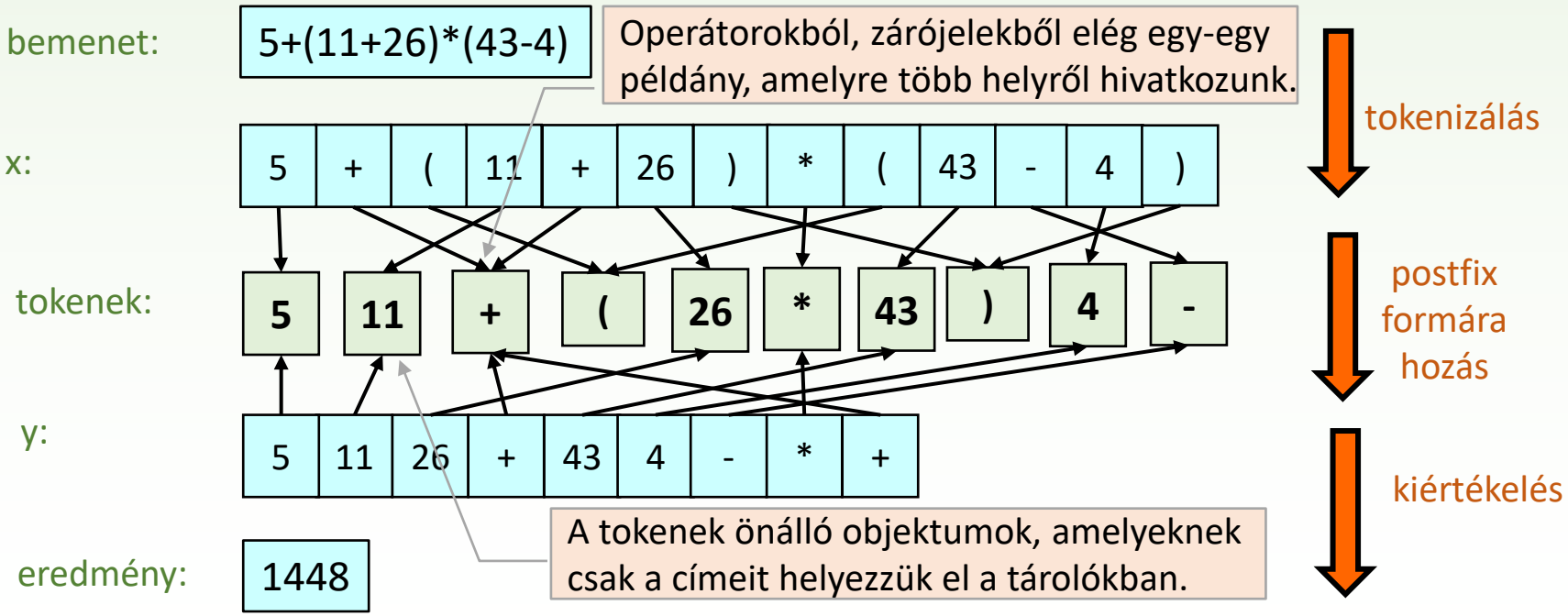
Az átalakításhoz is, és a kiértékeléshez is egy-egy vermet szoktak használni. Az elsőben műveleti jeleket és a nyitózárójeleket helyezünk el az átalakítás során, a másodikban operandusokat, illetve a részeredményeket.

Megoldási terv

1. Tokenizáljuk (operátorokra, operandusokra, nyitó- és csukózárójelekre bontjuk) a karakterláncként megadott infix formájú kifejezést és a tokenek címeit elhelyezzük egy x sorozatban.

2. Postfix formára hozzuk az x sorozatbeli kifejezést: egy verem segítségével alakítjuk át és helyezzük el egy y sorozatban.

3. Kiértékeljük az y sorozatbeli postfix formát egy verem segítségével.



A megoldás objektumai

sztring: az infix formájú kifejezés a szabványos bemeneten (fstream)

tokenek: speciális tokenek (Token), mint az operandusok (Operand), operátorok (Operator), zárójelek (LeftP, RightP)

sorozatok: tokenekre mutató pointerek gyűjteményei (vector<Token*>):
az infix formájú tokenizált kifejezést tárolja (x)
a postfix formára hozott tokenizált kifejezést tárolja (y)

vermek: tokenek címeit tároló verem (Stack<Token*>),
egész számokat tároló verem (Stack<int>)

Főprogram

```
int main() {
	char ch;
	do {
		cout << "Give me an arithmetic expression:\n";
		vector<Token*> x;
		try{
		// Tokenization
			...
		// Transforming into RPN
			vector<Token*> y;
			Stack<Token*> s
			...
		// Evaluation
			Stack<int> v;
			...
		} catch(MyException ex) { }

		deallocateToken(x);

		cout << "\nDo you continue? Y/N";
		cin >> ch;
	} while( ch!='n' && ch!='N' );
	return 0;
}
```

itt példányosodnak majd a tokenek

a folyamat során bárhol keletkezhet MyException::Interrupt kivétel

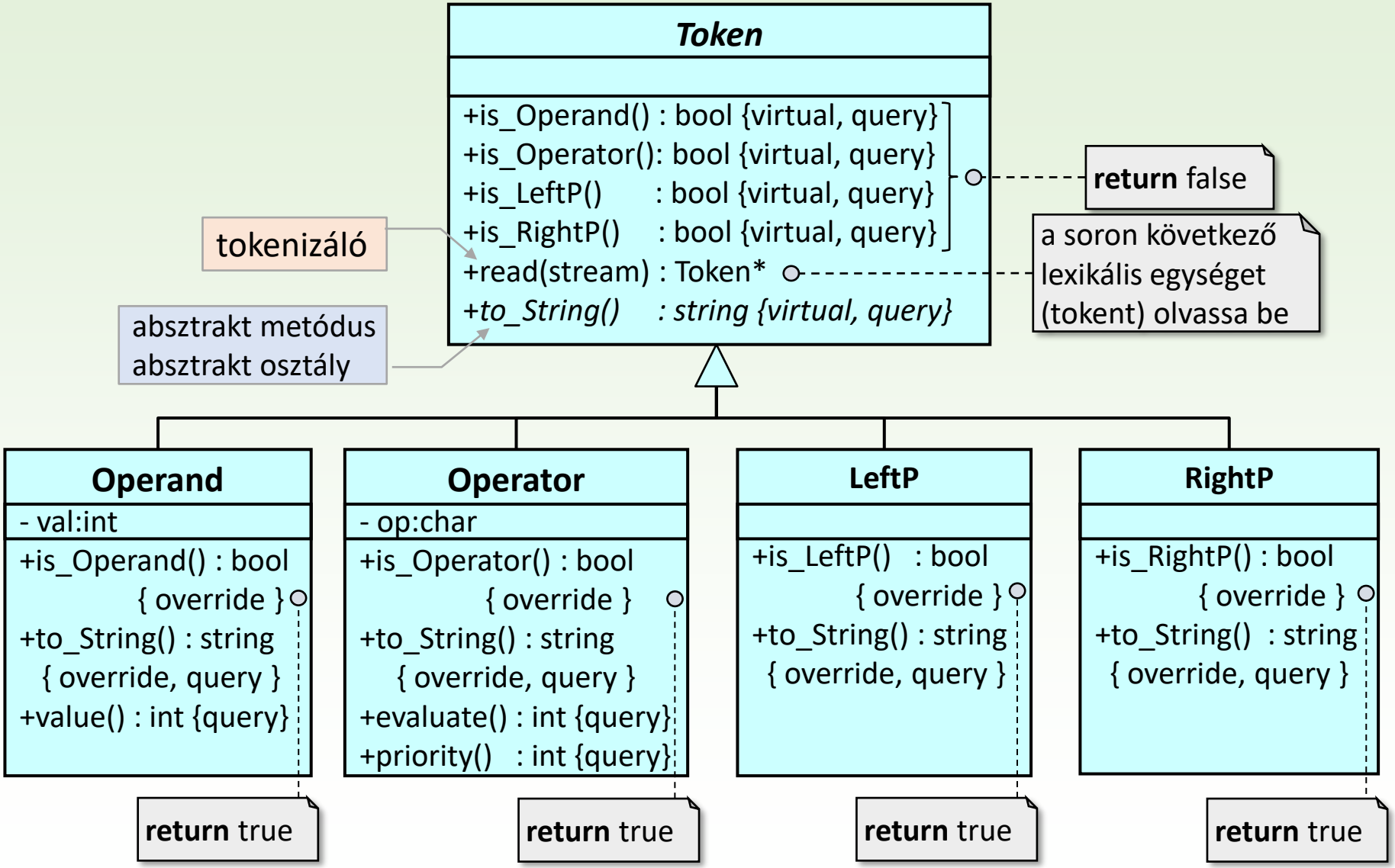
```
enum MyException { Interrupt };
```

```
void deallocateToken(vector<Token*> &x)
{
	for( Token* t : x ) delete t;
}
```

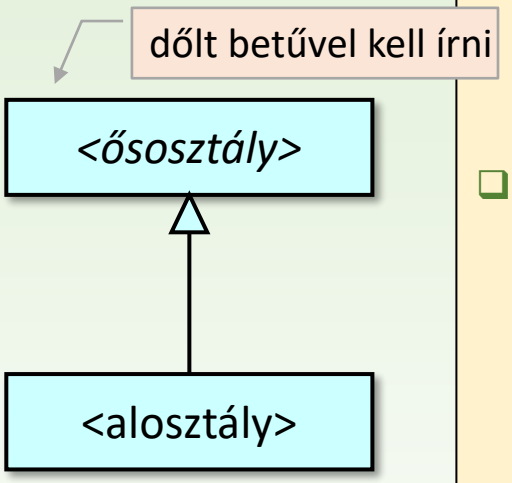
felszabadítja a tokenek helyfoglalásait

main.cpp

Token osztály és leszármazottjai

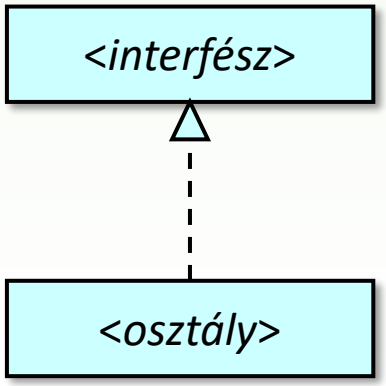


Absztrakt osztály, interfész



- Absztrakt (*abstract*) osztály az, amelyből nem példányosítunk objektumokat, kizárólag ősosztályként szolgálnak a származtatásokhoz.
- Egy osztály attól lesz absztrakt, hogy
  - konstruktorai nem publikusak,
  - vagy legalább egy metódusa absztrakt (dőlt betűvel kell írni), azaz nincs implementálva, csak származtatás során írjuk majd felül

- Interfésznek, azaz tisztán absztrakt (*pure abstract*) osztálynak nevezzük azt az osztályt, amelyiknek egyetlen metódusa sincs implementálva.
- Egy interfészből származtatott osztály, amelyik az interfész minden absztrakt metódusát implementálja, az megvalósítja az interfészt.



Token osztály

```
class Token
{
public:
        class IllegalElementException{
        private:
                char _ch;
        public:
                IllegalElementException(char c) : _ch(c){}
                char message() const { return _ch;}
        };
        virtual ~Token();
        virtual bool is_LeftP()         const { return false; }
        virtual bool is_RightP()        const { return false; }
        virtual bool is_Operand()       const { return false; }
        virtual bool is_Operator()      const { return false; }
        virtual bool is_End()           const { return false; }

        virtual std::string to_String() const = 0;

friend
        std::istream& operator>>(std::istream&, Token*&);
};
```

kivétel-kezeléshez

Mire jó a virtuális destruktor?

specifikációban még nem szerepelt

token.h

Operand osztály

```
class Operand: public Token
{
private:
	int _val;
public:
	Operand(int v) : _val(v) {}

	bool is_Operand() const override { return true; }

	std::string to_String()  const override {
		std::ostringstream ss;
		ss << _val;
		return ss.str();
	}

	int value() const { return _val; }
};
```

konverzió

token.h

LeftP osztályból elég egy példány

```
class LeftP : public Token
{
private:
    LeftP(){}
    static LeftP *_instance;
public:
    static LeftP *instance() {
        if ( _instance == nullptr ) _instance = new LeftP();
        return _instance;
    }

    bool is_LeftP()          const override { return true; }

    std::string to_String()  const override { return "("; }
};
```

legyen privát

egyetlen példányra mutat, ha az létezik

gyártó függvény

itt hívható a privát konstruktor

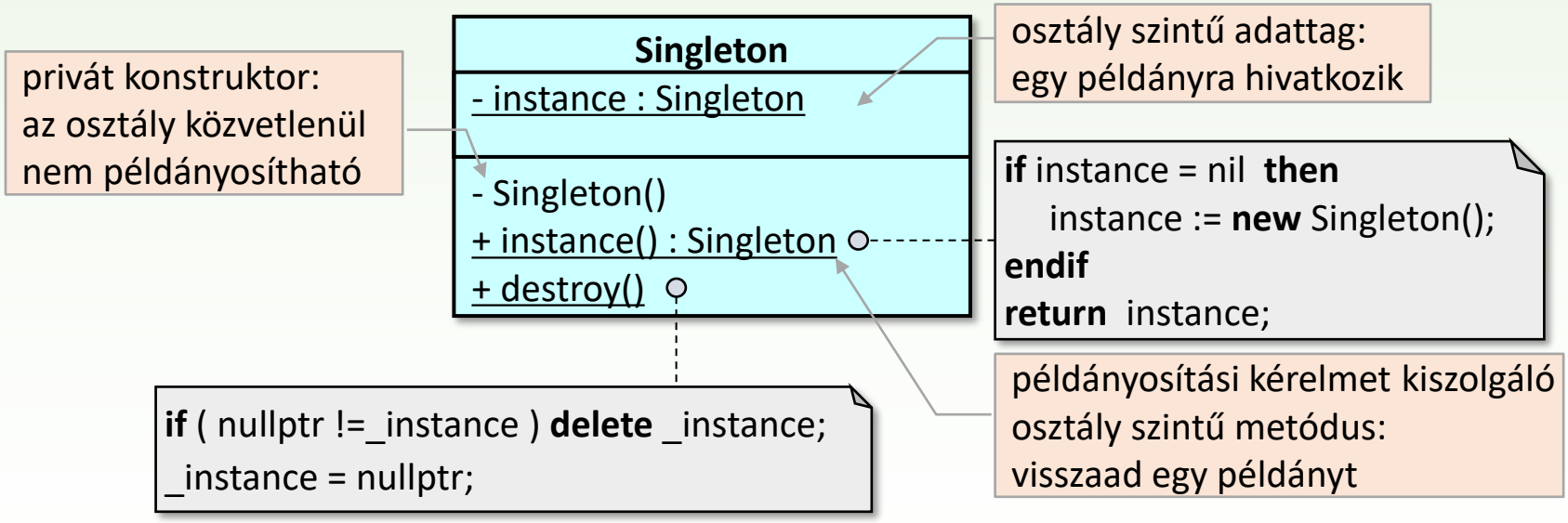
token.h

```
LeftP   *LeftP::_instance = nullptr;
```

token.cpp

Egyke (singleton) tervezési minta

- Egy osztályhoz legfeljebb egy objektumot példányosítsunk függetlenül a példányosítási kérelmek számától.



Operator osztály metódusai

```
int Operator::evaluate(int leftValue, int rightValue) const
{
	switch(_op){
		case '+':  return leftValue+rightValue;
		case '-':  return leftValue-rightValue;
		case '*':  return leftValue*rightValue;
		case '/':  return leftValue/rightValue;
		default: return 0;
	}
}

int Operator::priority() const
{
	switch(_op){
		case '+': case '-':   return 1;
		case '*' : case '/':  return 2;
		default: return 3;
	}
}
```

token.cpp

Single responsibility

Open-Close

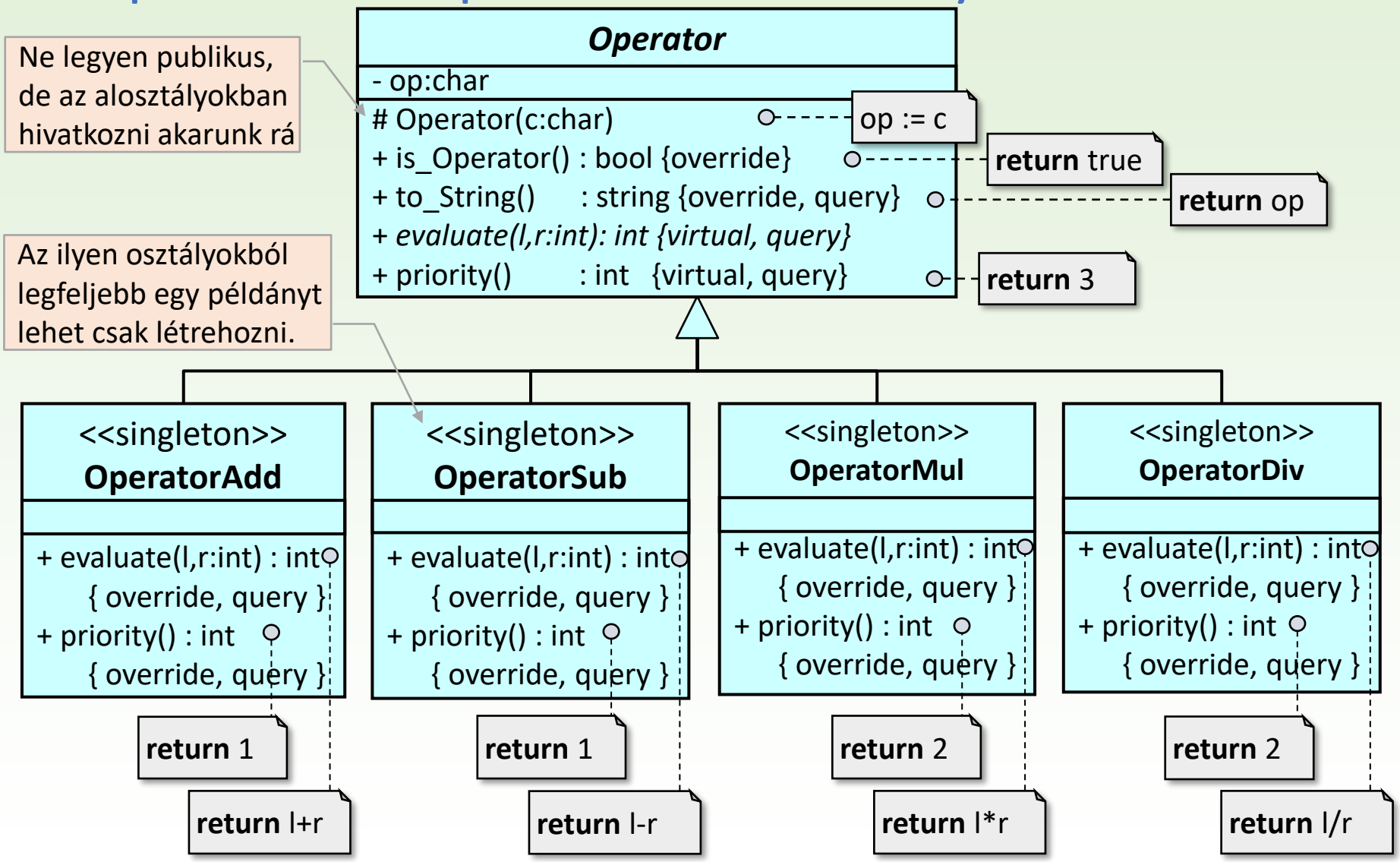
Liskov's substitution

I

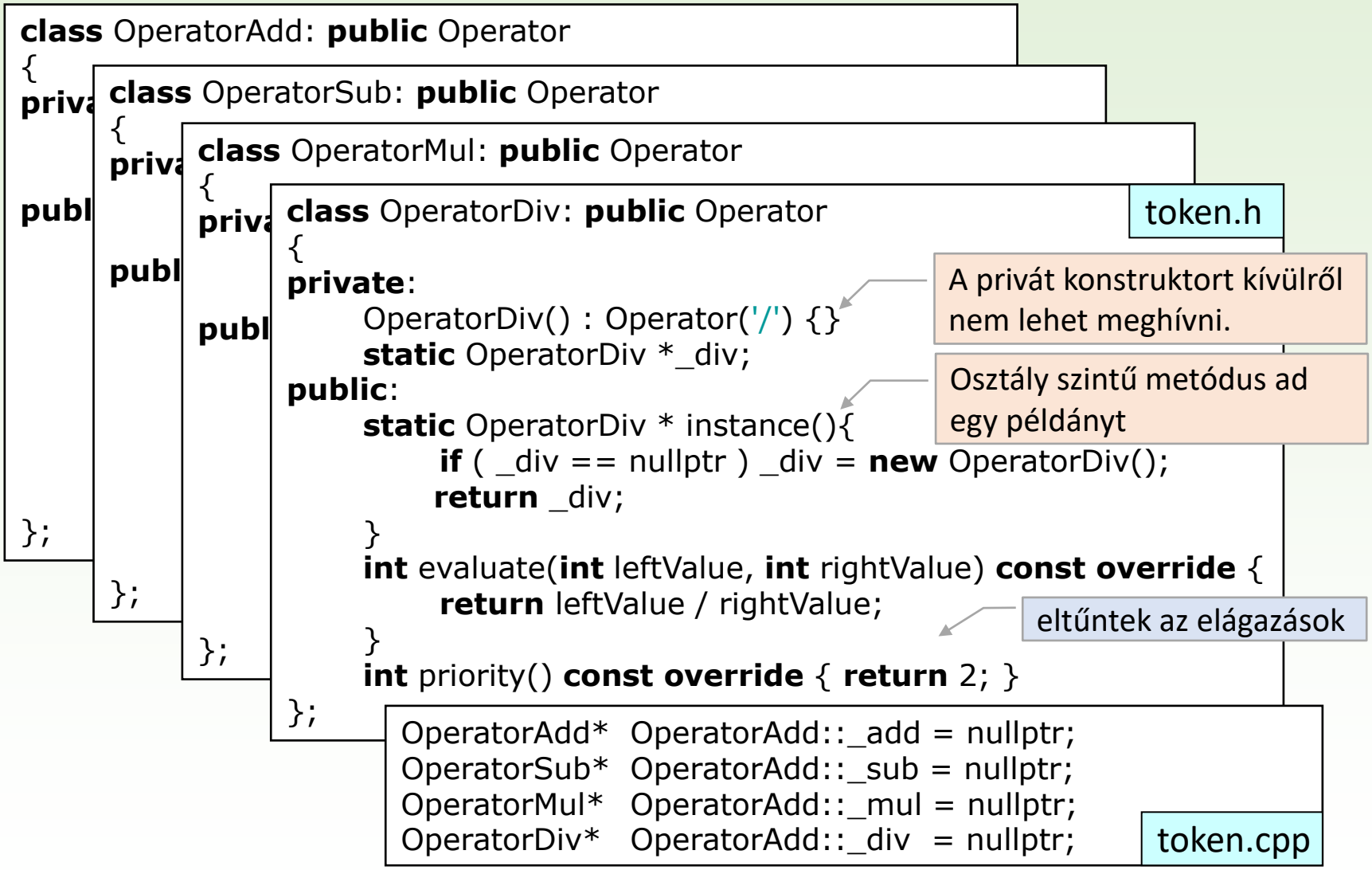
D

ez a kód nem felel meg az Open-Close elvnek

Speciális operátor osztályok



Egyke operator osztályok



Tokenizálást végző operátor

```
istream& operator>> (istream &s, Token* &t){
        char ch;
        s >> ch;
        switch(ch){
                case '0'   : case '1' : case '2' : case '3' : case '4':
                case '5'   : case '6' : case '7' : case '8' : case '9':
                             s.putback(ch);
                             int intval;
                             s >> intval;
                             t = new Operand(intval); break;
                case '+'   : t = OperatorAdd::instance(); break;
                case '-'   : t = OperatorSub::instance(); break;
                case '*'   : t = OperatorMul::instance();  break;
                case '/'   : t = OperatorDiv::instance();  break;
                case '('   : t = LeftP::instance();    break;
                case ')'   : t = RightP::instance();  break;
                case ';'   : t = End::instance();     break;
                default:  if(!s.fail()) throw new Token::IllegalCharacterException(ch);
        }
        return s;
}
```

vissza a pufferbe

egykék használata

a bemenetként megadott aritmetikai kifejezést pontosvessző zárja le

token.cpp

Kifejezés tokenizálása

```
vector<Token*> x;
// Tokenization
try{
        Token *t;
        cin >> t;
        while( !t->is_End() ){
                x.push_back(t);
                cin >> t;
        }
}catch(Token::IllegalElementException *ex){
        cout << "Illegal character: " << ex->message() << endl;
        delete ex;
        throw Interrupt;
}

if(0==x.size()) {
        cout << "Empty expression\n";
        throw Interrupt;
}
```

tokent olvas

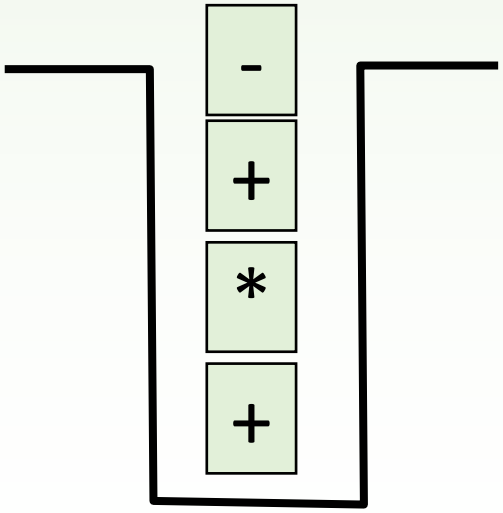
az olvasás dobja

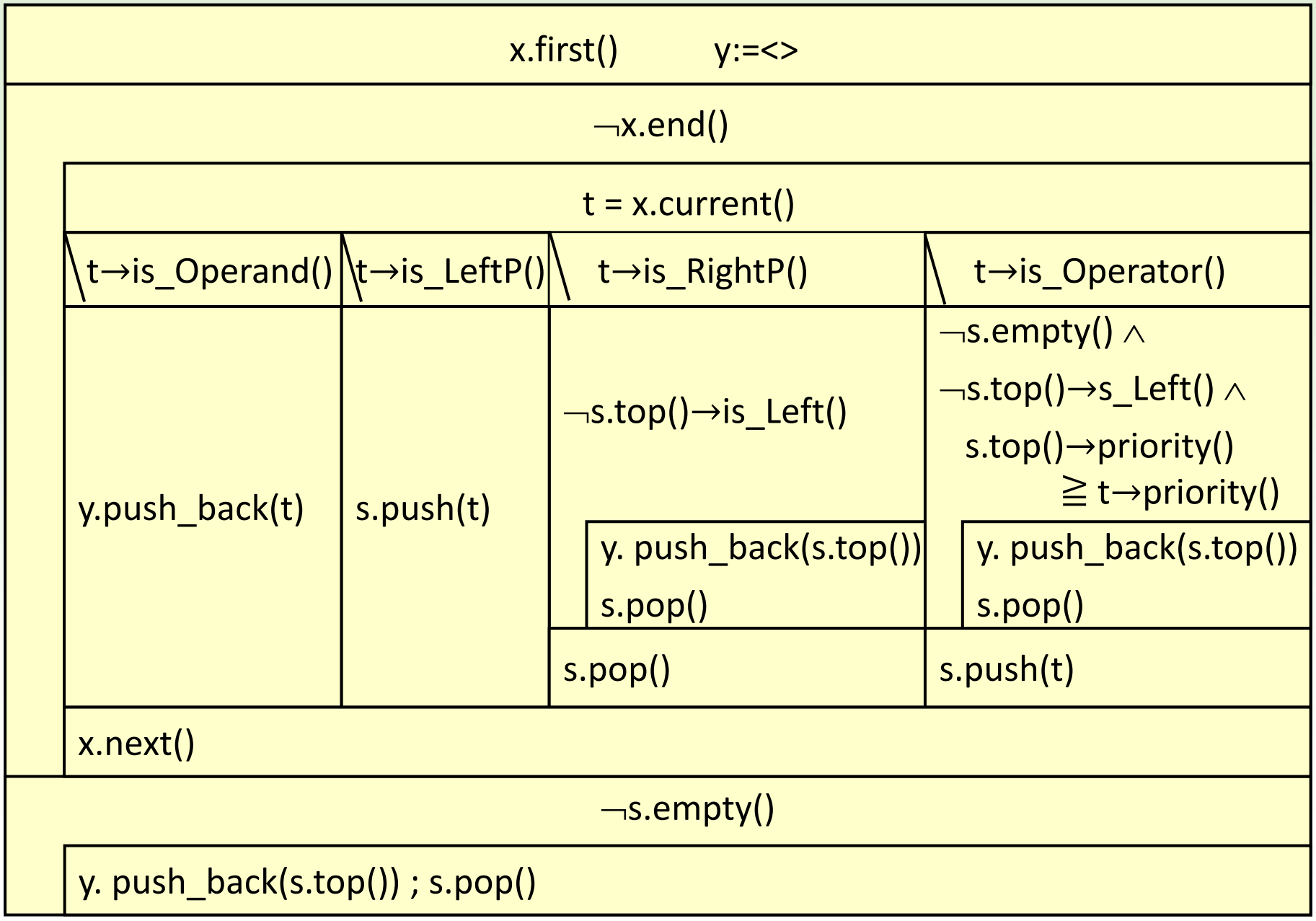
main.cpp

RPN-re hozás

A bemenő sorozat nyitó zárójeleit és műveleti jeleit egy verembe tesszük (az alacsonyabb precedenciájú műveleti jel mindig helyet cserél az alatta levő magasabb precedenciájúval), minden más jelet közvetlenül a kimenő sorozatba másolunk. Csukó zárójel olvasása esetén illetve a bemenő sorozat feldolgozásának végén kiürítjük a verem tartalmát a leg(f)első nyitózárójeléig a kimenő sorozatba.

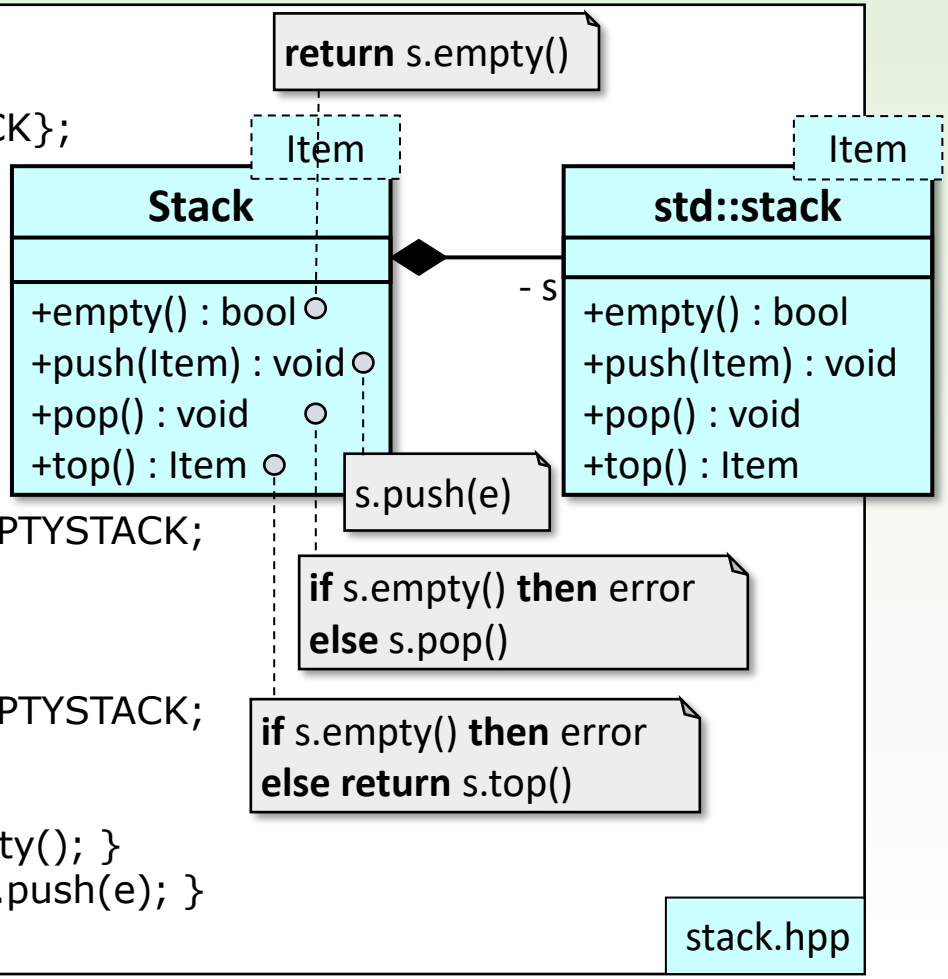
5	+	(	11	+	26	)	*	(	43	-	4	)




x.first()
y:=<>
¬x.end()
t = x.current()
t→is_Operand()
t→is_LeftP()
t→is_RightP()
t→is_Operator()
y.push_back(t)
s.push(t)
¬s.top()→is_Left()
¬s.empty() ∧
¬s.top()→s_Left() ∧
s.top()→priority()
≧ t→priority()
y. push_back(s.top())
s.pop()
y. push_back(s.top())
s.pop()
s.pop()
s.push(t)
x.next()
¬s.empty()
y. push_back(s.top()) ; s.pop()

Kivételt dobó veremsablon

```
#include <stack>

enum StackExceptions{EMPTYSTACK};

template <typename Item>
class Stack
{
private:
      std::stack<Item> s;
public:
      void pop() {
            if( s.empty() ) throw EMPTYSTACK;
            s.pop();
      }
      Item top() const {
            if( s.empty() ) throw EMPTYSTACK;
            return s.top();
      }
      bool empty() {  return s.empty(); }
      void push(const Item& e) { s.push(e); }
};
```



Postfix formára hozás

```
// Transforming into RPN

vector<Token*> y;
Stack<Token*>  s;

for( Token* t : x ){
        if      ( …
        else if ( …
        else if ( …
        else if ( …
}
while( !s.empty() ){
        if( s.top()->is_LeftP() ){
                cout << "Syntax error!\n";
                throw Interrupt;
        }else{
                y.push_back(s.top());
                s.pop();
        }
}
```

felsorolás

a négy-ágú elágazás a következő dián

hiba lehetőség:
több nyitó zárójel, mint csukó

main.cpp

Postfix formára hozás (ciklusmag)

```
if      ( t->is_Operand() ) y.hiext(t);
else if ( t->is_LeftP() )   s.push(t);
else if ( t->is_RightP() ){
    try{
        while( !s.top()->is_LeftP() ) {
            y.push_back(s.top());
            s.pop();
        }
        s.pop();
    }catch(StackExceptions ex){
        if(ex==EMPTYSTACK){
            cout << "Syntax error!\n";
            throw Interrupt;
        }
    }
}else if ( t->is_Operator() ) {
    while( !s.empty() && s.top()->is_Operator() &&
        ((Operator*)s.top())->priority()>=((Operator*)t)->priority() ) {

        y.push_back(s.top());
        s.pop();
    }
    s.push(t);
}
```

hiba lehetőség: több a csukó zárójel, mint nyitó

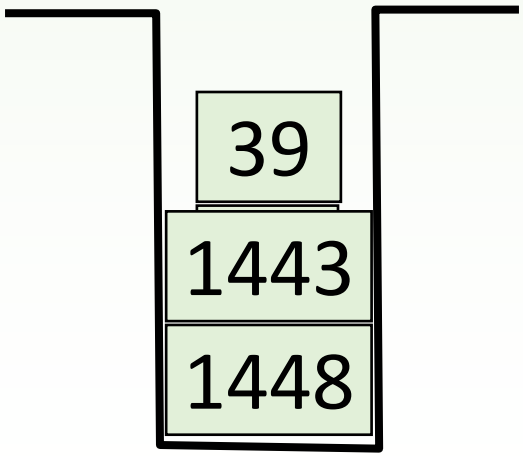
„static casting”: Az s.top()->priority() nem jó, mert Token-ben nincs priority()

main.cpp

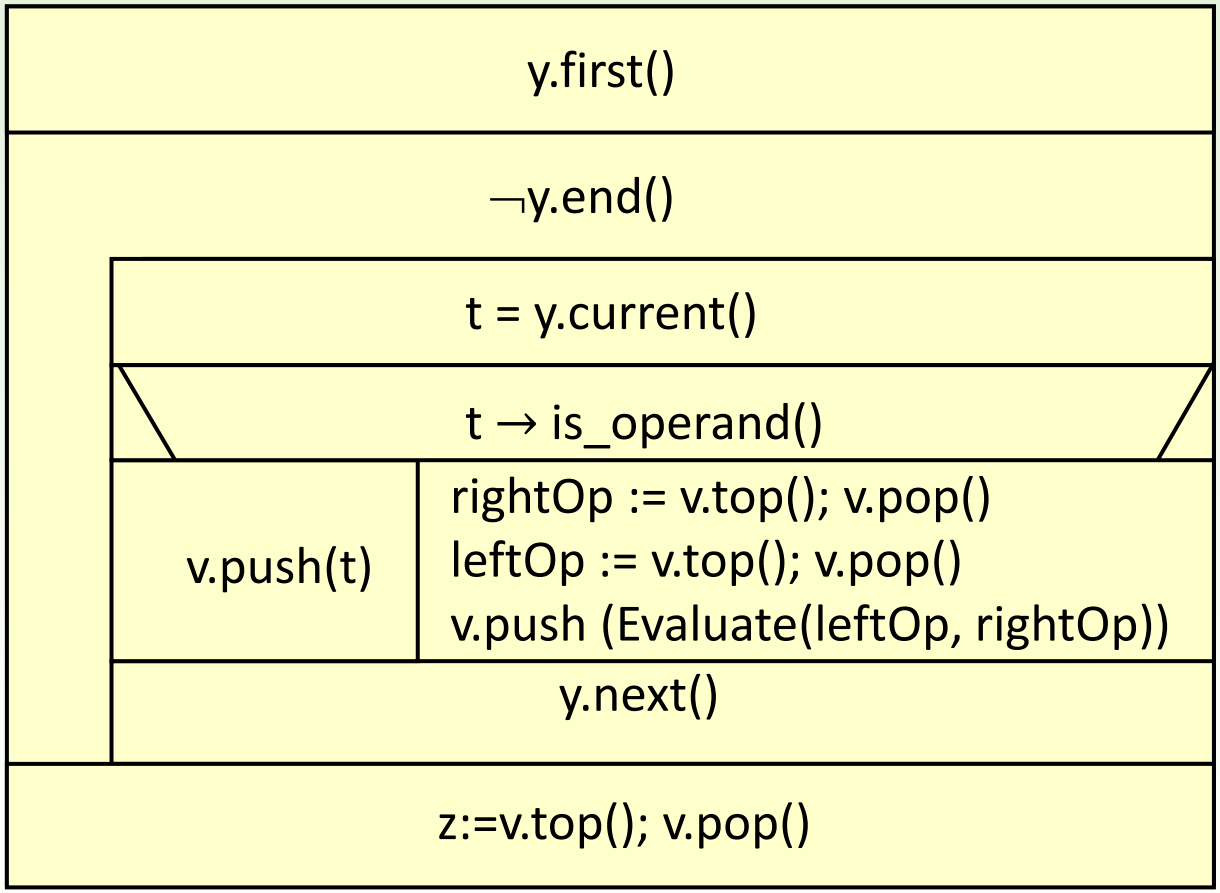
Postfix forma kiértékelése

A postfix forma operandusait (olvasásuk sorrendjében) egy verembe tesszük. Műveleti jel olvasása esetén a verem tetején levő két értéket (csak bináris műveleteink vannak) kivesszük, azokat a szóban forgó művelettel feldolgozzuk, és az eredményt visszatesszük a verembe. A feldolgozás végén a veremben találjuk kifejezés értékét.

5	11	26	+	43	4	-	*	+



Kiértékelés algoritmusa



Kiértékelés

```
// Evaluation
try{
        Stack<int> v;
        for( Token* t : y ){
                if ( t->is_Operand() ) {
                        v.push( ((Operand*)t)->value() );
                } else{
                        int rightOp = v.top(); v.pop();
                        int leftOp  = v.top(); v.pop();
                        v.push(((Operator*)t)->evaluate(leftOp, rightOp));
                }
        }
        int result = v.top(); v.pop();
        if( !v.empty() ){
                cout << "Syntax error!";
                throw Interrupt;
        }
        cout << "value of the expression: " << result << endl;
}catch( StackExceptions ex ){
        if( ex==EMPTYSTACK ){
                cout << "Syntax error! ";
                throw Interrupt;
        }
}
```

felsorolás

statikus konverzió

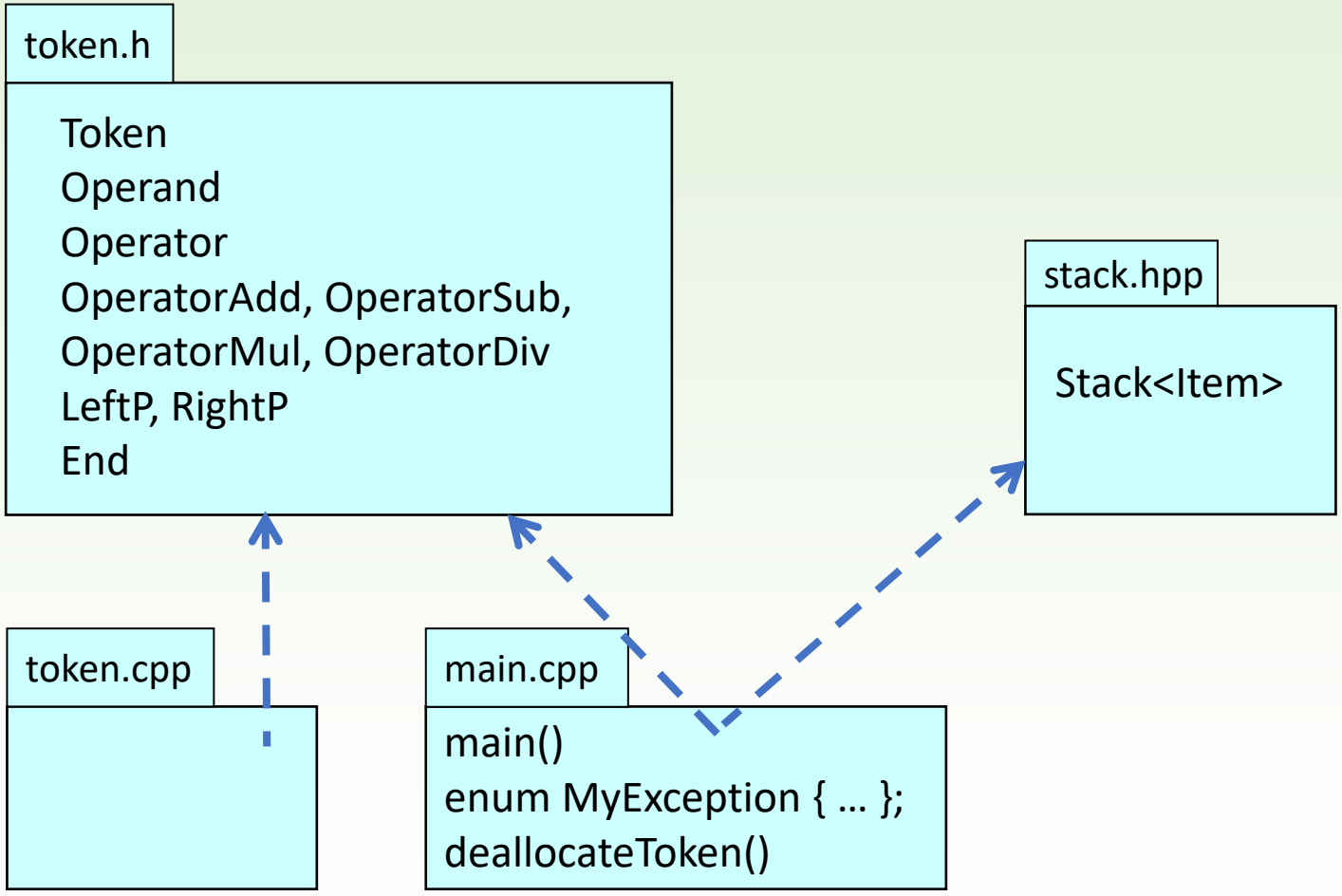
statikus konverzió

hiba lehetőség:
több operandus

hiba lehetőség:
kevés operandus

main.cpp

Csomag diagram



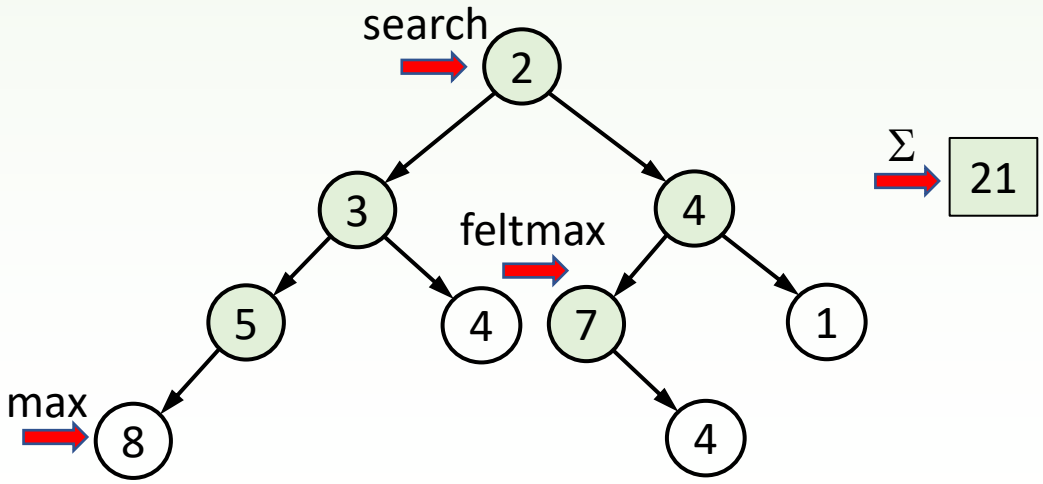
Feladat: Bináris fa bejárása

Építsünk fel véletlenszerűen egy bináris fát megadott egész számokból úgy, hogy a számok a fa csúcsaiban helyezkedjenek majd el.

Írjuk ki a csúcsokban tárolt értékeket különféle bejárási stratégiák alapján.

Oldjunk meg olyan programozási feladatokat, mint például a belső csúcsok értékeinek összege, az összes csúcs vagy csak a belső csúcsok értékeinek maximuma, az „első” páros szám keresése valamelyik bejárási szerint.

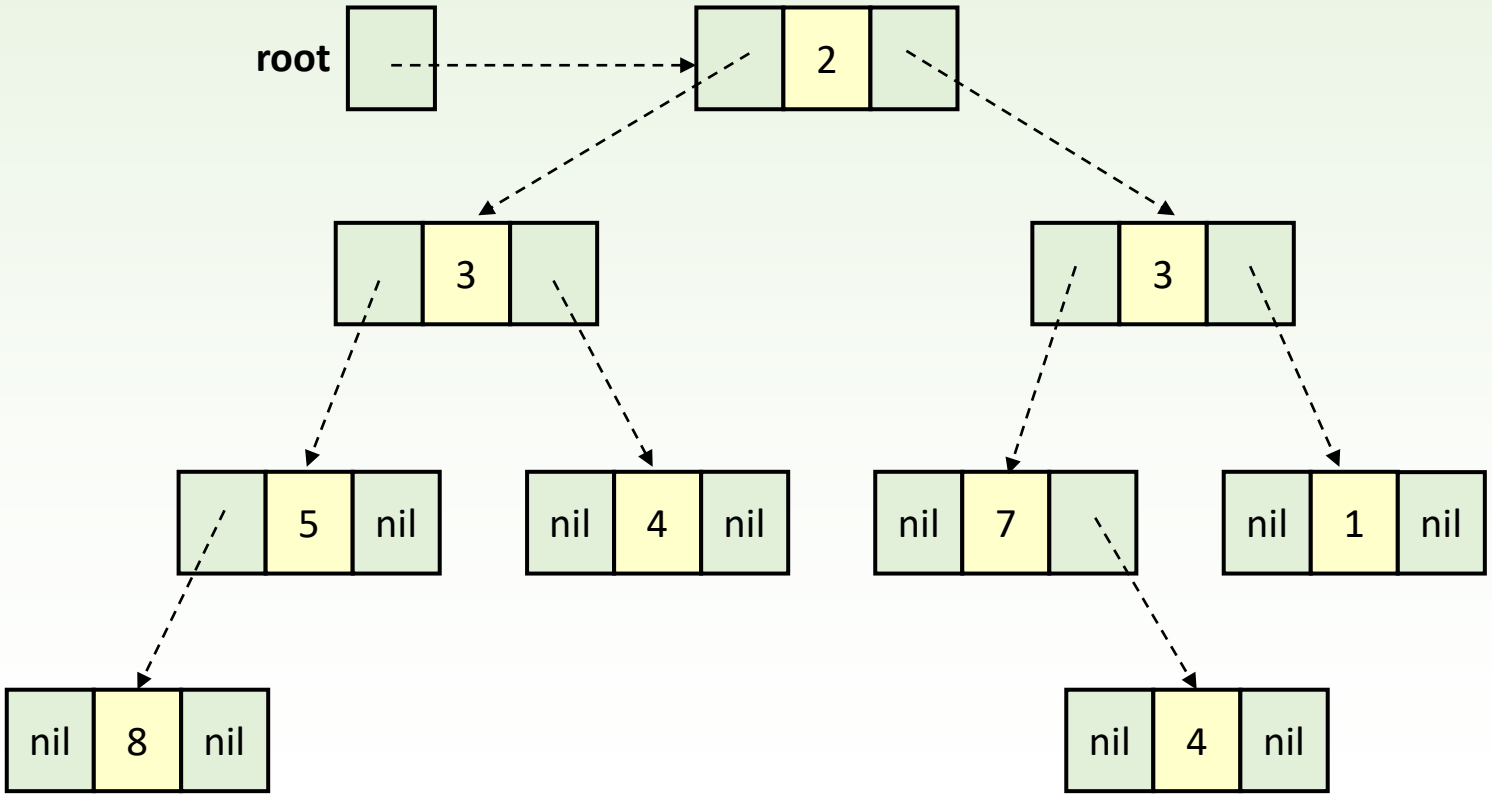
A bináris fát úgy tervezzük meg, hogy a csúcsaiban tárolt értékek típusa könnyen megváltoztatható legyen más olyan típusra, amelyen szintén értelmezhető összegzés, maximum kiválasztás, és keresés.



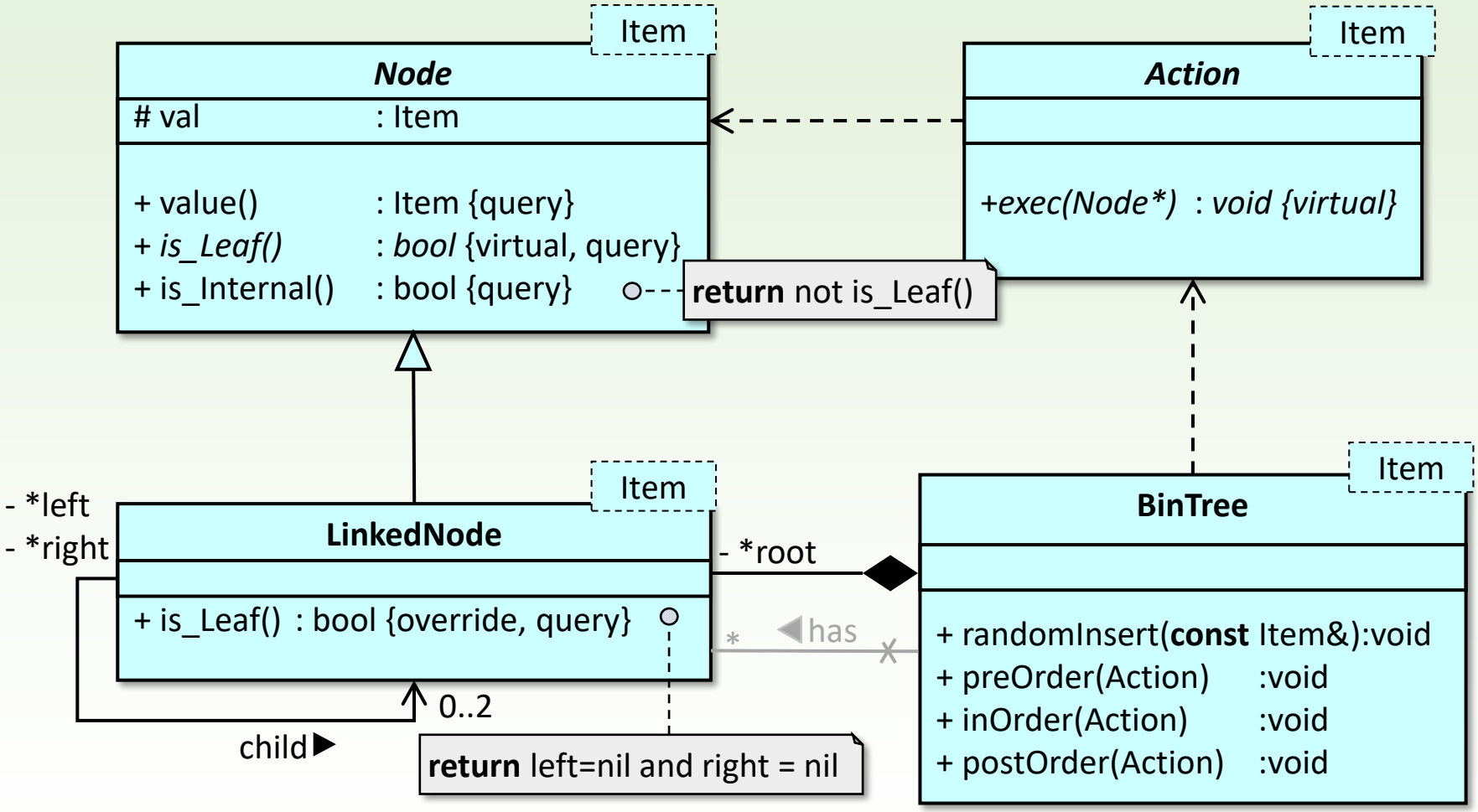
Bináris fa láncolt ábrázolása

```
class BinTree
protected:
    LinkedNode *_root;
    ...
};
```

```
template <typename Item>
struct LinkedNode {
    LinkedNode *_left;
    Item _val;
    LinkedNode *_right;
};
```



Osztálydiagram



Csúcs osztálysablonja

```
template <typename Item>
class Node {
public:
        Item value() const { return _val; }
        virtual bool is_Leaf() const = 0;
        bool is_Internal() const { return !is_Leaf(); }
        virtual ~Node(){}
protected:
        Node(const Item& v): _val(v){}
        Item _val;
};

template <typename Item> class BinTree;

template <typename Item>
class LinkedNode: public Node<Item>{
public:
        friend class BinTree;
        LinkedNode(const Item& v, LinkedNode *l, LinkedNode *r):
                Node<Item>(v), _left(l), _right(r){}
        bool is_Leaf()const
                { return _left==nullptr && _right==nullptr; }
private:
        LinkedNode *_left;
        LinkedNode *_right;
};
```

A Bintree definíciójának megelőlegezése.
A definíció majd a LinkedNode után jön.

A LinkedNode megengedi a Bintree-nek, hogy hivatkozhasson a privát tagjaira.

bintree.hpp

Bináris fa osztálysablonja

```
template <typename Item>
class Action{
public:
      virtual void exec(Node<Item> *node) = 0;
      virtual ~Action(){}
};
```

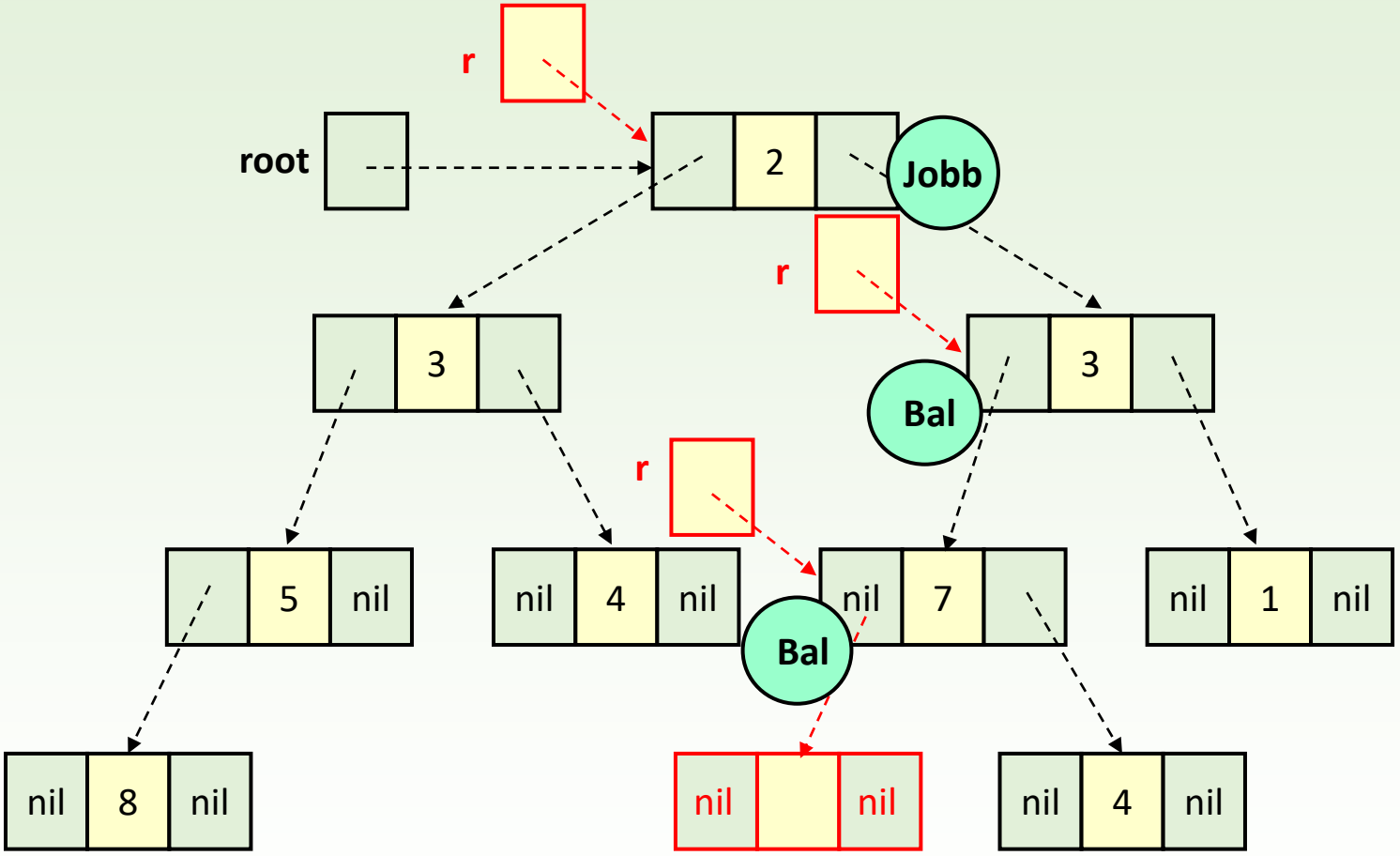
```
template <typename Item>
class BinTree{
public:
      BinTree():_root(nullptr){srand(time(nullptr));}
      ~BinTree();

      void randomInsert(const Item& e);

      void preOrder (Action<Item>*todo){ pre  (_root, todo); }
      void inOrder   (Action<Item>*todo){ in   (_root, todo); }
      void postOrder(Action<Item>*todo){ post(_root, todo); }

protected:
      LinkedNode<Item>* _root;

      void pre (LinkedNode<Item>*r, Action<Item>*todo);
      void in   (LinkedNode<Item>*r, Action<Item>*todo);
      void post(LinkedNode<Item>*r, Action<Item>*todo);
};1
```

véletlenszám-generátor inicializálása:
#include <time.h>
#include <cstdlib>

adott tevékenység-objektummal bejárják a fa csúcsait a gyökértől kezdődően

bintree.hpp

Új csúcs beszúrása a bináris fába



Új csúcs beszúrása a bináris fába

```
void BinTree<Item>::randomInsert(const Item& e)
{
    if(_root==nullptr) _root = new LinkedNode<Item>(e, nullptr, nullptr);
    else {
        LinkedNode<Item> *r = _root;
        int d = rand();
        while(d&1 ? r->_left!=nullptr : r->_right!=nullptr){
            if(d&1) r = r->_left;
            else    r = r->_right;
            d = rand();
        }
        if(d&1) r->_left  = new LinkedNode<Item>(e, nullptr, nullptr);
        else    r->_right= new LinkedNode<Item>(e, nullptr, nullptr);
    }
}
```

véletlenszám generálása

A d szám utolsó bitje vajon 1-e?

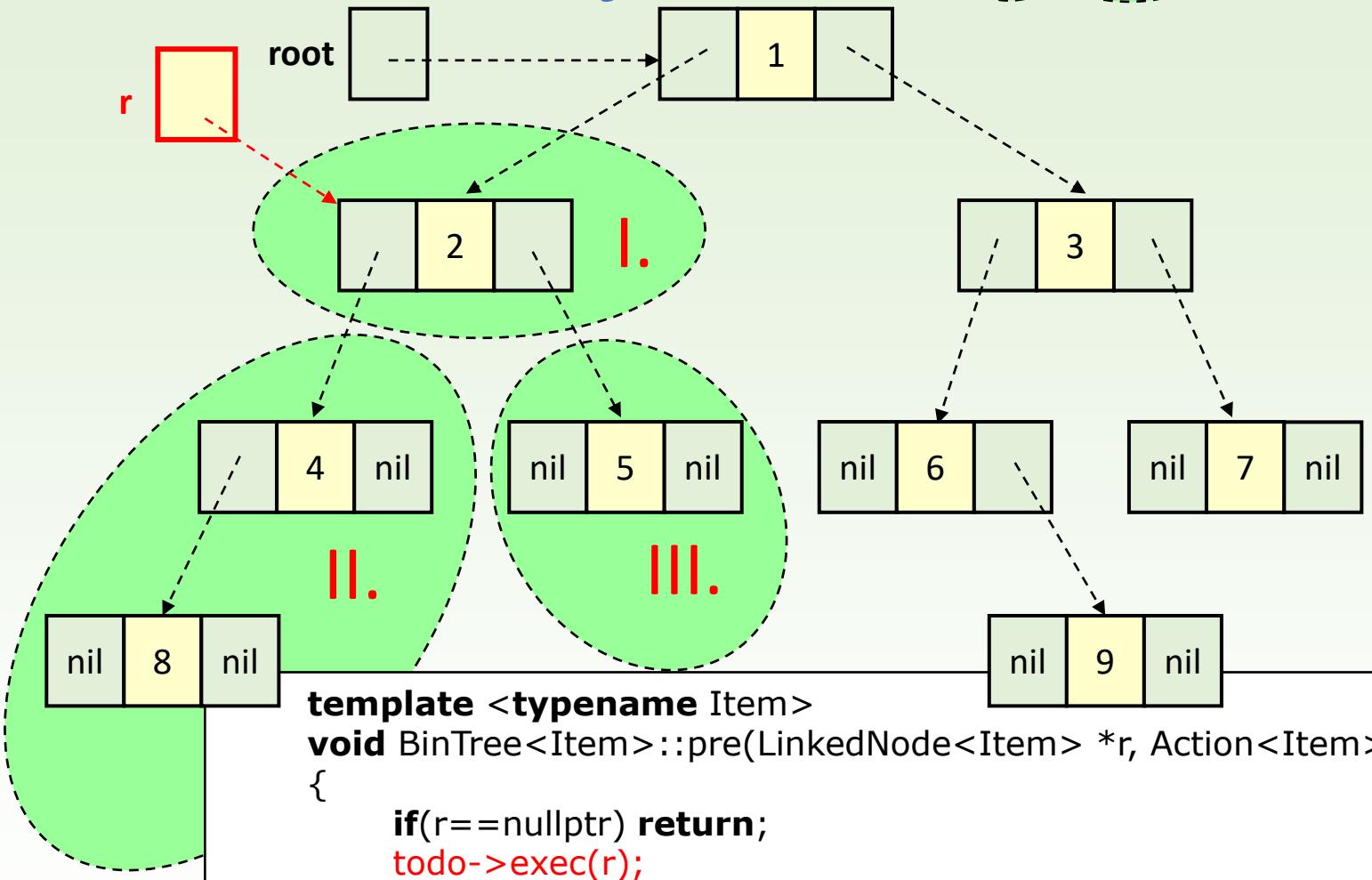
bintree.hpp

Bináris fa felépítése

```
#include <iostream>
#include "bintree.hpp"

BinTree build()
{
	BinTree<int> t;
	int i;
	while(std::cin >> i){
		t.randomInsert(i);
	}
	return t;
}
```

Preorder bejárás

1 2 4 8 5 3 6 9 7

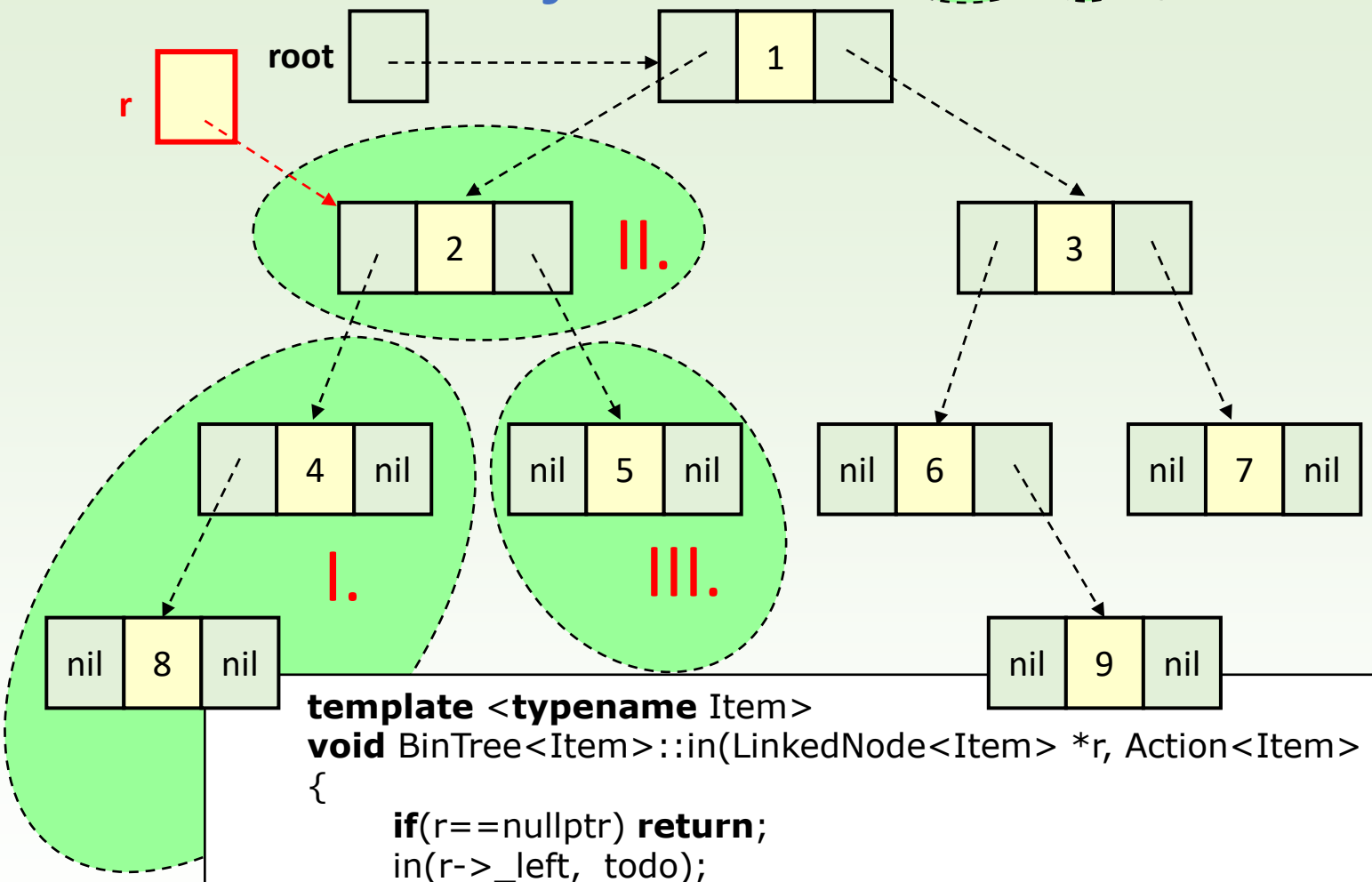



```
template <typename Item>
void BinTree<Item>::pre(LinkedNode<Item> *r, Action<Item> *todo)
{
        if(r==nullptr) return;
        todo->exec(r);
        pre(r->_left,  todo);
        pre(r->_right, todo);
}
```

bintree.hpp

Inorder bejárás

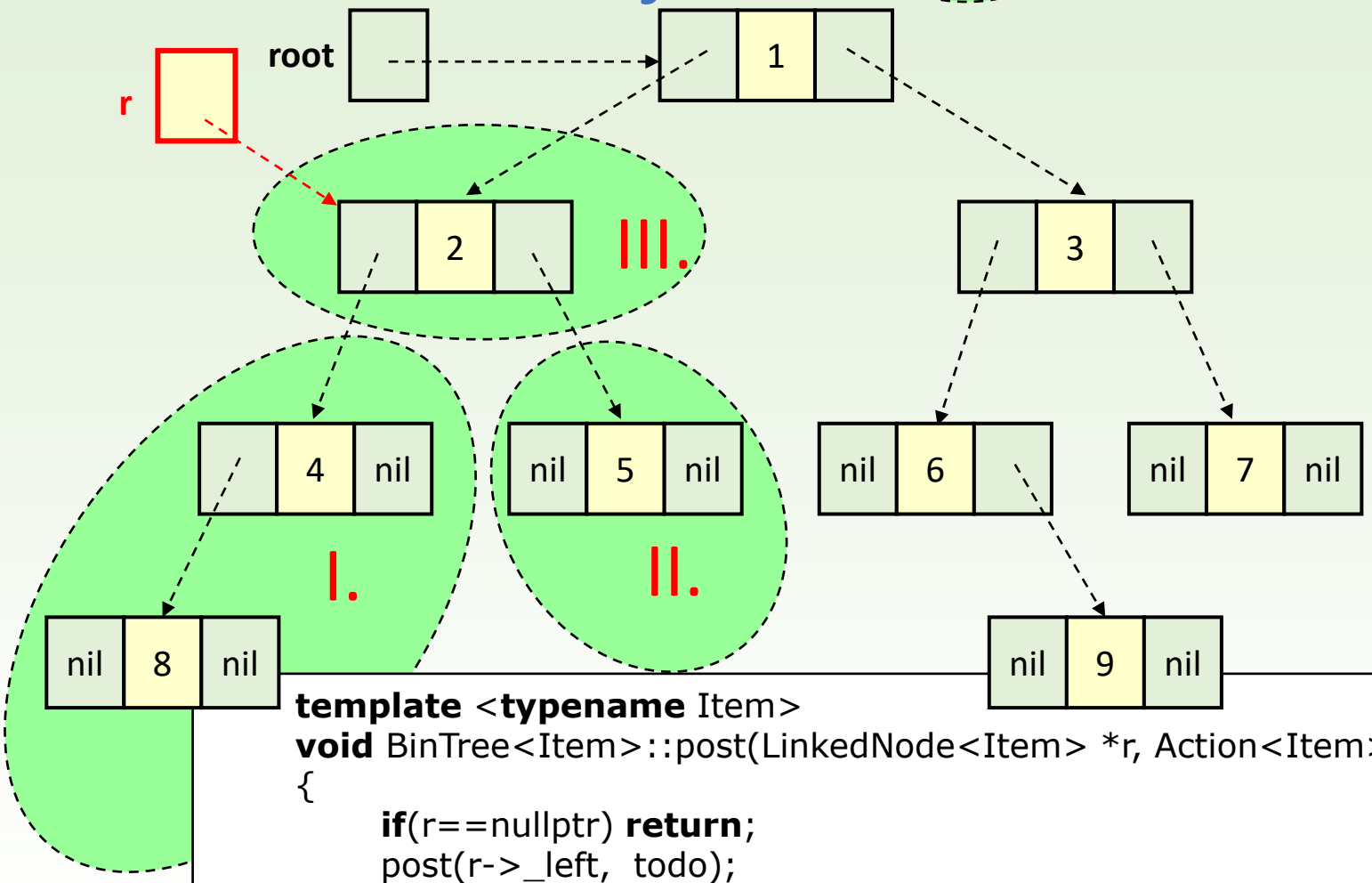
8 4 2 5 1 6 8 3 7




```
template <typename Item>
void BinTree<Item>::in(LinkedNode<Item> *r, Action<Item> *todo)
{
        if(r==nullptr) return;
        in(r->_left,  todo);
        todo->exec(r);
        in(r->_right, todo);
}
```

bintree.hpp

Postorder bejárás 8 4 5 2 9 6 7 3 1



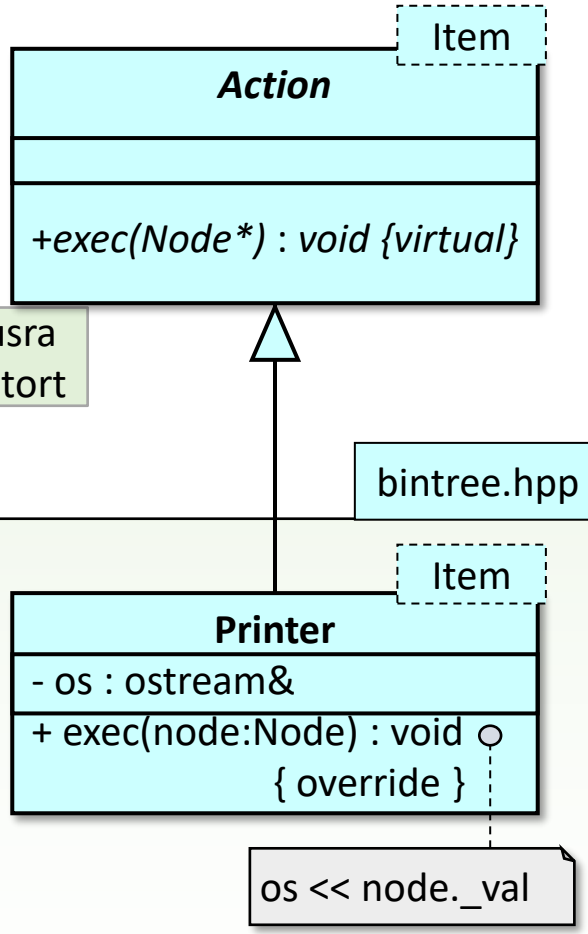

```
template <typename Item>
void BinTree<Item>::post(LinkedNode<Item> *r, Action<Item> *todo)
{
        if(r==nullptr) return;
        post(r->_left,  todo);
        post(r->_right, todo);
        todo->exec(r);
}
```

bintree.hpp

Kiírás tevékenység osztálysablonja

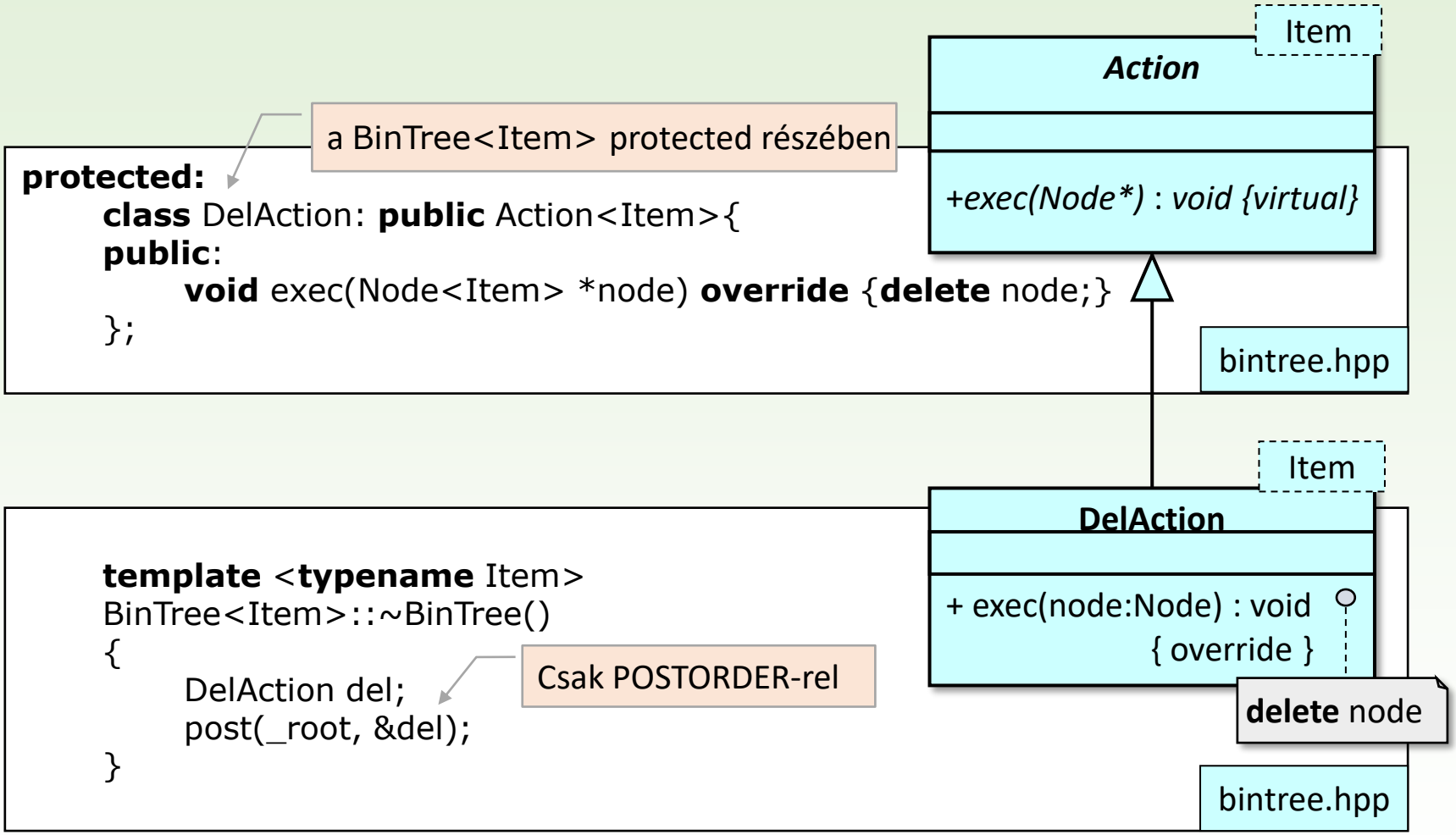
```
template <typename Item>
class Printer: public Action<Item>{
public:
        Printer(ostream &o): _os(o){};
        void exec(Node<Item> *node) override {
                _os << '[' << node->value() << ']';
        }
private:
        ostream& _os;
};
```

Item paraméter helyébe írt típusra legyen értelmezve a kiíró operátort

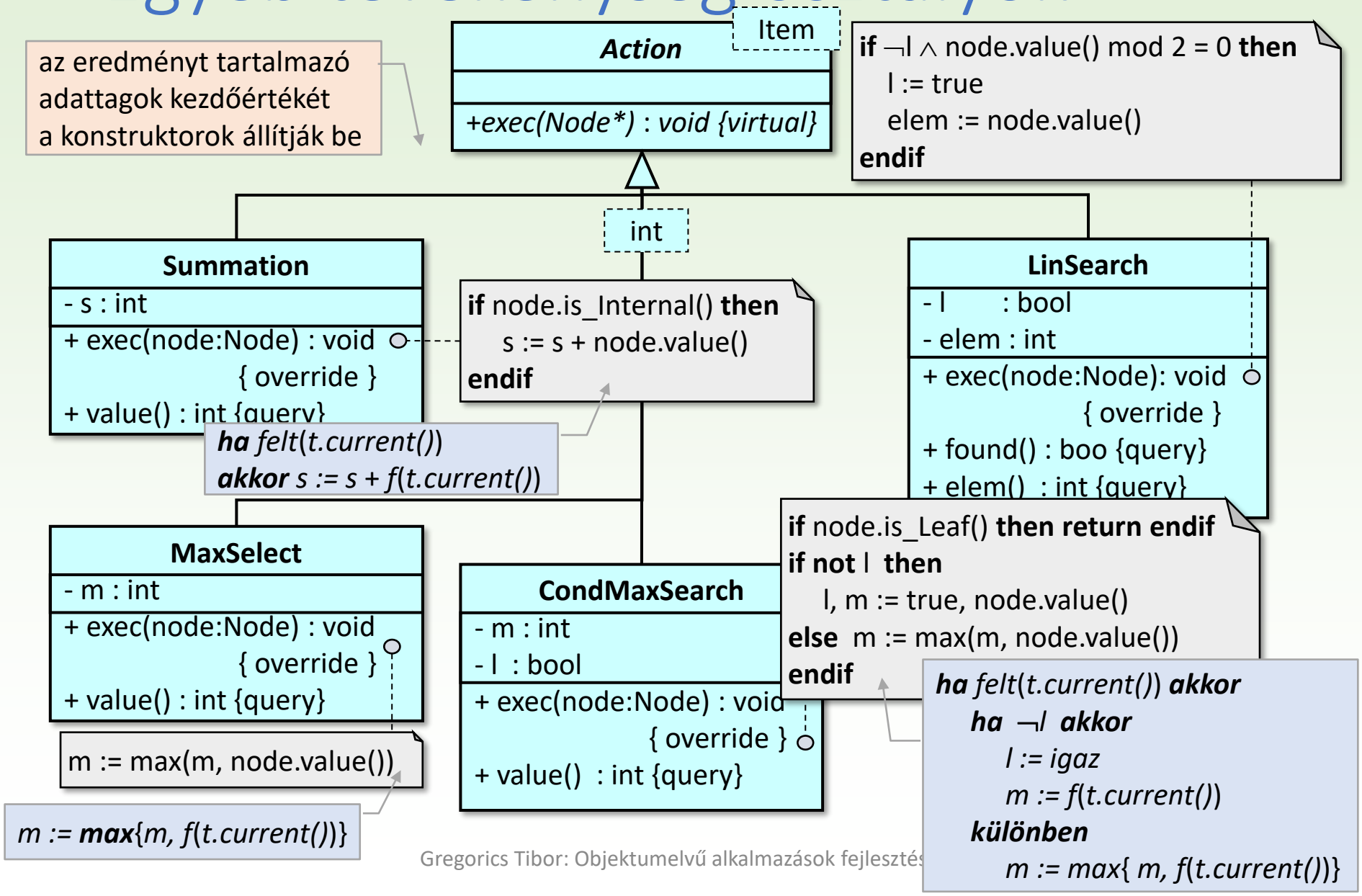



```
BinTree<int> t = build();

Printer<int> print(cout);

cout << "Preorder traversal :";
t.preOrder(&print);     cout << endl;

cout << "Inorder traversal :";
t.inOrder(&print);      cout << endl;

cout << "Postorder traversal :";
t.postOrder(&print);    cout << endl;
```

Destruktor: bejárás törléssel



Egyéb tevékenység osztályok



Összegzés tevékenység osztálya

```
class Summation: public Action<int>{
public:
	void Summation():_s(0){}
	void exec(Node<int> *node) override {
		{ if(node->Is_Internal())_s+=node->value();}
	int value() const {return _s;}
private:
	int _s;
};
```

```
BinTree<int> t = build();

Summation sum;
t.preOrder(&sum);

cout << "Sum of the elements of the tree: "
	<< sum.value() << endl;
```

Lineáris keresés

```
class LinSearch: public Action<int>{
public:
    void LinSearch():_l(false){}
    void exec(Node<int> *node) override {
        if (!l && 0==node->value())%2 ){
            l = true;
            _elem = node->value();
        }
    }
    bool  found() const {return _l;}
    int   elem()  const {return _elem;}
private:
    bool _l;
    int  _elem;
};
```

```
BinTree<int> t = build();

LinSearch search;
t.preOrder(&search);

if (search.found()) cout << search.elem() << " is an";
else cout << "There is no";
cout << " even element of the tree.";
```

Maximum kiválasztás

```
class MaxSelect: public Action<int>{
public:
        MaxSelect(int &i) : _m(i){}
        void exec(Node<int> *node) override
          {_m = max( _m, node->value() ); }
        int value() const {return _m;}
private:
        int _m;
};
```

```
BinTree<int> t = build();

MaxSelect max(t.rootValue());
t.preOrder(&max);

cout << "Maxima of the elements of the tree: " << max.value();
```

Dobjon kivételt, ha a bináris fa üres, azaz még gyökércsúcs sincs

```
template <class Item> class BinTree {
…
public:
        enum Exceptions{NOROOT};
        Item rootValue() const {
                if( _root==nullptr ) throw NOROOT;
                return _root->value();
        }
```

bintree.hpp

Feltételes maximum keresés

```
class CondMaxSearch: public Action<int>{
public:
    struct Result {
        int m;
        bool l;
    };
    CondMaxSearch(){_r.l = false;}
    virtual void exec(Node<int> *node) {
        if(node->is_Leaf()) return;
        if(!_r.l){
            _r.l = true;
            _r.m = node->value();
        }else{
            _r.m = max( _r.m, node->value() );
        }
    }
    Result value(){return _r;}
private:
   Result _r;
};
```

```
BinTree<int> t = build();

CondMaxSearch max;
t.preOrder(&max);

cout << "Maxima of the elements of the tree: " << endl;
if(max2.value().l) cout << max2.value().m << endl;
else               cout << "none" << endl;
```