Inheritance and data structures

Task

Transform an infix expression to a postfix expression (Reverse Polish Notation - RPN) and calculate its value.

infix expression		postfix expression		value
5+(11+26)*(43-4)	→	**5 11 26 + 43 4 - * +**	→	**1448**

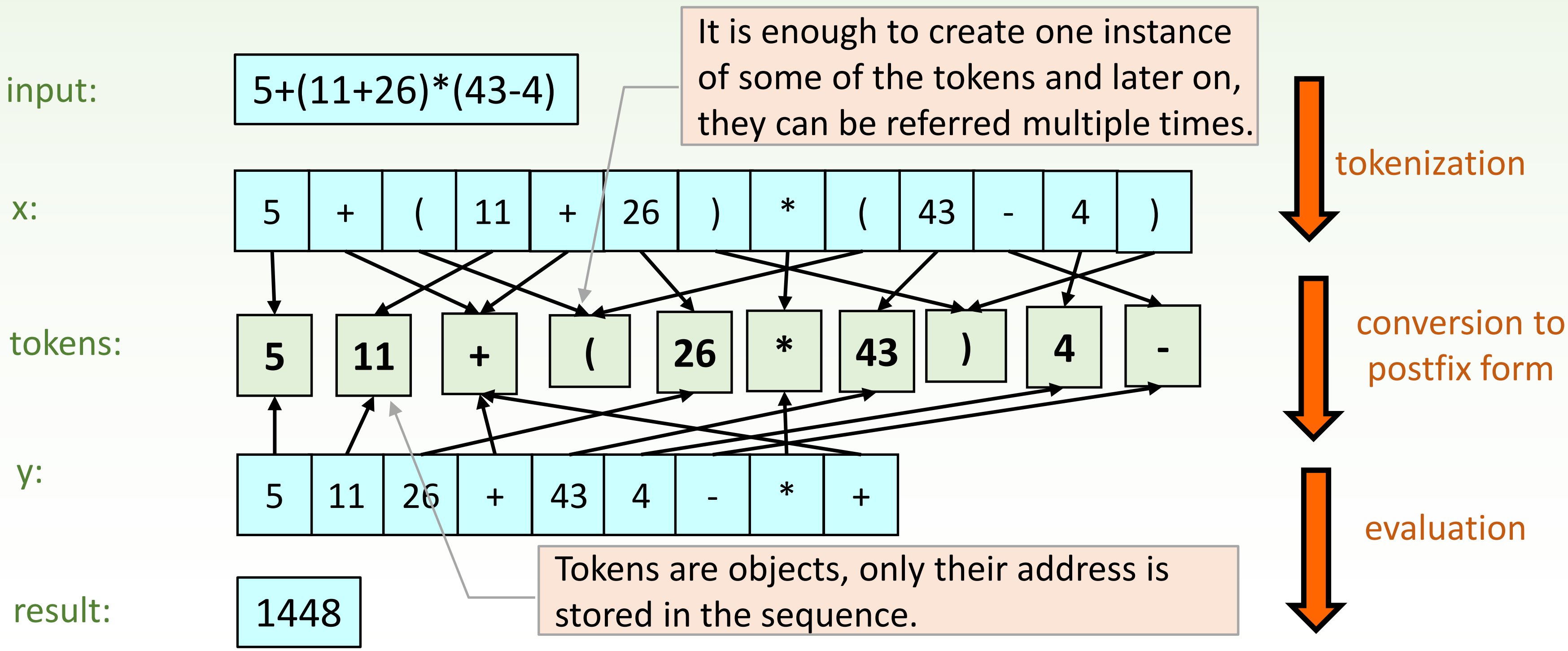
To transform an infix expression and to evaluate it, usually two stacks are needed. In the first one, the operators and the open parentheses are stored. In the second one, the operands and the partial results are put.

Plan

1. Tokenize the infix expression and put the tokens into an *x* sequence.

2. Convert to Polish notation: *x* is converted into a *y* sequence in which the tokens are in postfix form. For the conversion, a stack is used.

3. Evaluate the *y* sequence by using a second stack.



Objects to be used

string: an infix expression given in a standard input (fstream)

tokens: specific tokens (Token), as operands (Operand), operators (Operator), and parentheses (LeftP, RightP)

sequences: pointers pointing at the tokens (vector<Token*>):
for the tokenized infix expression (x),
for the tokenized postfix expression (y)

stacks: for storing the pointers of the tokens (Stack<Token*>),
for storing the numbers (Stack<int>)

Main program

main.cpp

```
int main() {
	char ch;
	do {
		cout << "Give me an arithmetic expression:\n";
		vector<Token*> x;
		try{
		// Tokenization
			...
		// Transforming into Polish notation
			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;
}
```

token instantiation

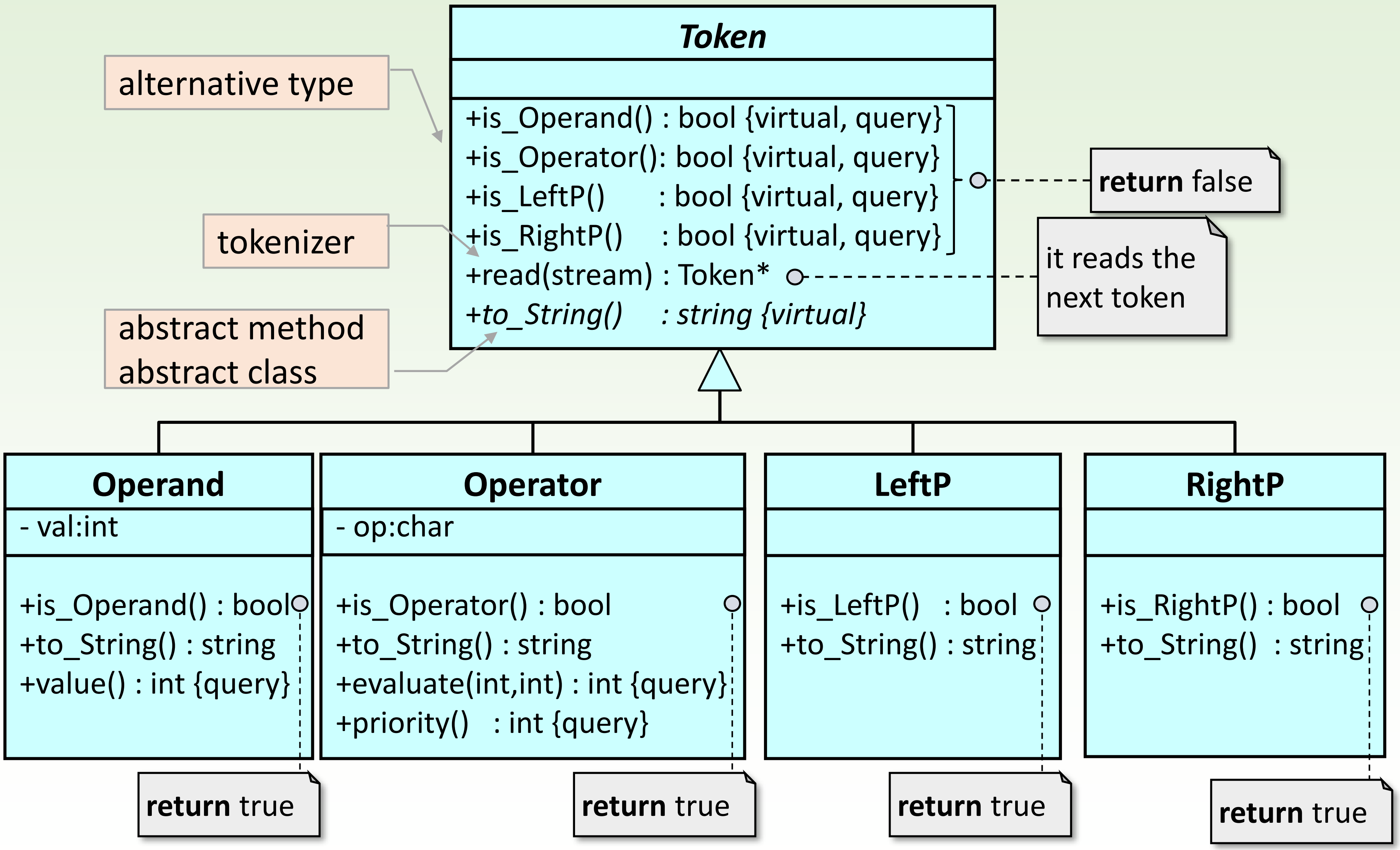
MyException::Interrupt exception may be raised anytime

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

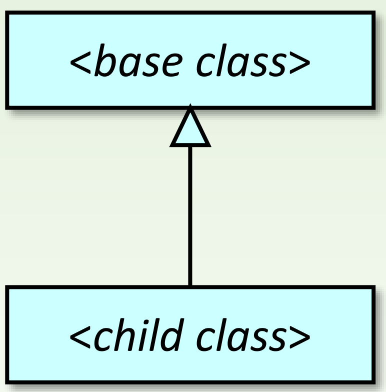
```
void deallocateToken(vector<Token*> &x)
{
	for( Token* t : x ) delete t;
}
```

frees the memory allocation of the tokens

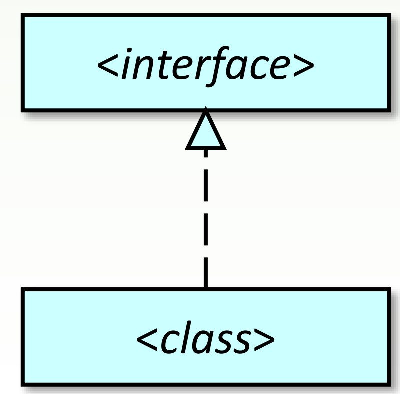
Token class and its children



Abstract class, interface



- Abstract class is never instantiated, it is used only as a base class for inheritance.
  - name of the abstract class is italic.
- A class is abstract if
  - its constructors are not public, or
  - at least one method is abstract (it is not implemented and it is overridden in a child)
    - name of the abstract method is also italic



- *Pure abstract* classes are called interfaces, none of their methods are implemented.
- When a class implements all of the abstract methods of an interface, it realizes the interface.

Token class

```
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*&);
};
```

for exception handling

Why is the destructor virtual?

not presented in the specification

token.h

Operand class

```
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; }
};
```

conversion

token.h

LeftP class
(one instance is enough)

```
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 "("; }
};
```

private constructor

points at the only one instance

creator method

private constructor is called here

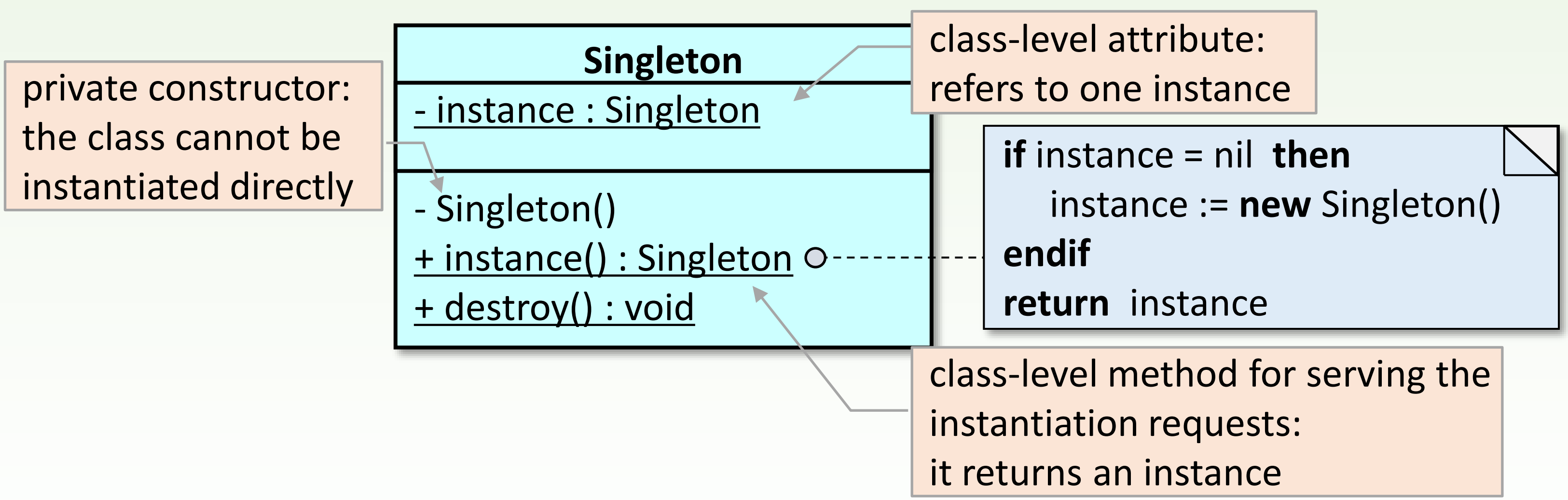
token.h

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

token.cpp

Singleton design pattern

- The class is instantiated only once, irrespectively of the number of instantiation requests.



Design patterns are class diagram patterns that help object-oriented modeling. They play a significant role in reusability, modifiability, and ensuring efficiency.

RightP, End singleton classes

```cpp
class RightP : public Token {
private:
	RightP(){}
	static RightP *_instance;
public:
	static RightP *instance() {
		if ( _instance == nullptr ) _instance = new RightP();
		return _instance;
	}
	bool is_ RightP()		const override { return true; }
	std::string to_String()	const override { return ")"; }
};
```

```cpp
RightP *RightP::_instance = nullptr;
End    *End::_instance = nullptr;
```

token.cpp

```cpp
class End : public Token {
private:
	End(){}
	static End *_instance;
public:
	static End *instance() {
		if ( _instance == nullptr ) _instance = new End();
		return _instance;
	}
	bool is_ End()		const override { return true; }
	std::string to_String()	const override { return ";"; }
};
```

token.h

Operator class

```
class Operator: public Token
{
private:
	char _op;
public:
	Operator(char o) : _op(o) {}

	bool is_Operator()       const override { return true; }
	std::string to_String()  const override {
		string ret;
		ret = _op;        conversion
		return ret;
	}
	virtual int evaluate(int leftValue, int rightValue) const;
	virtual int priority() const;
};
```

token.h

Methods of class Operator

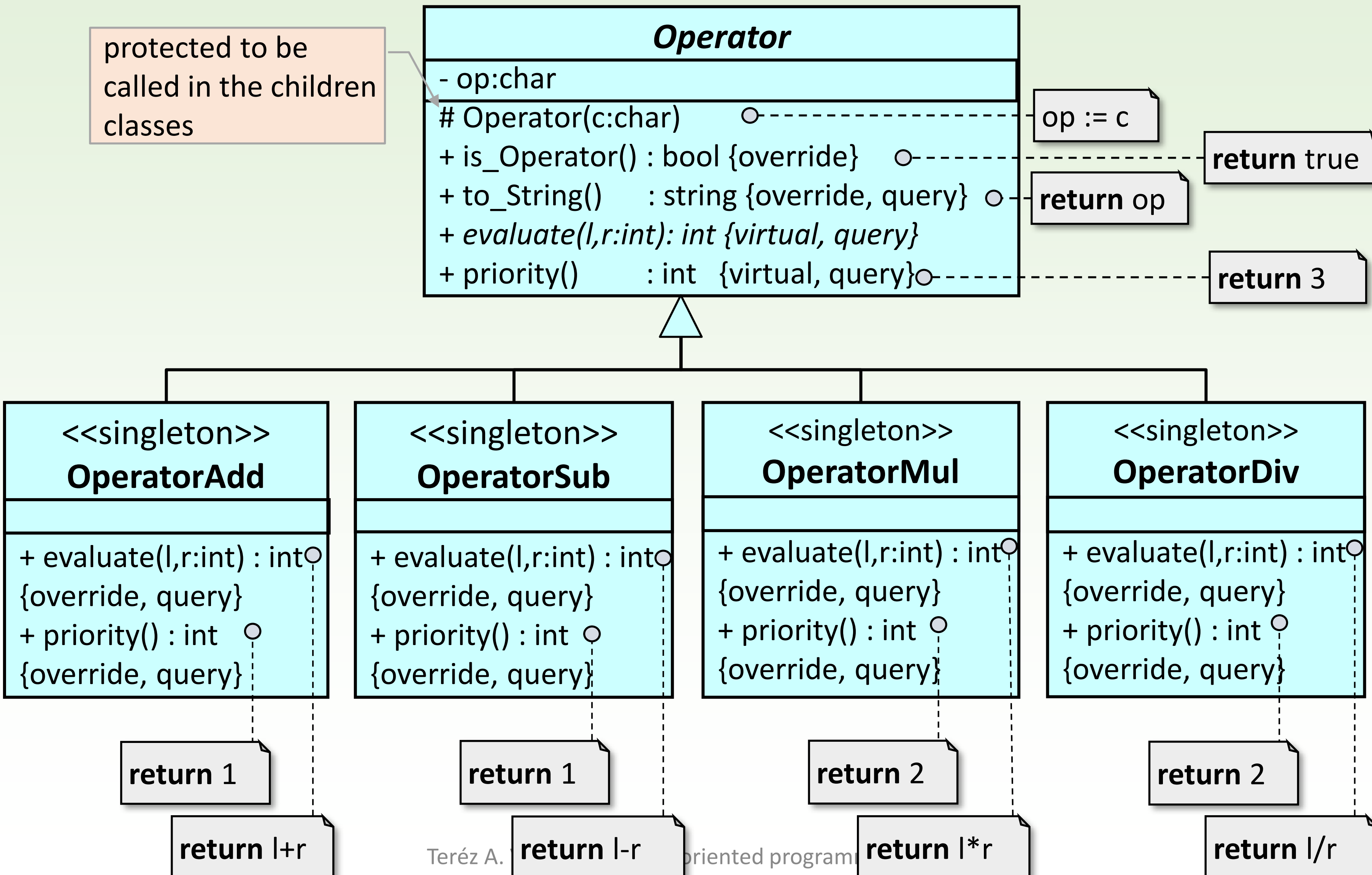
```
int Operator::evaluate(int leftValue, int rightValue) const
{
	switch(_op){
		case '+':  return leftValue+rightValue;
		case '-':  return leftValue-rightValue;
		case '*':  return leftValue*rightValue;
		case '/':  return leftValue/rightValue;
		default: return 0;
	}
}

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

token.cpp

Single responsibility
Open-Close
Liskov substitution
Interface segregation
Depedency inversion

this code does not satisfy the open-close principle

Operator classes



Abstract Operator class

```
class Operator: public Token
{
private:
	char _op;
protected:
	Operator(char o) : _op(o) {}
public:
	bool is_Operator()       const override { return true; }
	std::string to_String()  const override {
		string ret;
		ret = _op;
		return ret;
	}
	virtual int evaluate(int leftValue, int rightValue) const = 0;
	virtual int priority() const { return 3; }
};
```

token.h

Singleton operator classes

```
class OperatorAdd: public Operator
{
priv
publ
};
```

```
class OperatorSub: public Operator
{
priv
publ
};
```

```
class OperatorMul: public Operator
{
priv
publ
};
```

token.h

```
class OperatorDiv: public Operator
{
private:
        OperatorDiv() : Operator('/') {}
        static OperatorDiv *_div;
public:
        static OperatorDiv * instance(){
                if ( _div == nullptr ) _div = new OperatorDiv();
                return _div;
        }
        int evaluate(int leftValue, int rightValue) const override {
                return leftValue / rightValue;
        }
        int priority() const override { return 2; }
};
```

No need for conditionals

token.cpp

```
OperatorAdd*  OperatorAdd::_add = nullptr;
OperatorSub*  OperatorAdd::_sub = nullptr;
OperatorMul*  OperatorAdd::_mul = nullptr;
OperatorDiv*  OperatorAdd::_div  = nullptr;
```

Tokenizer operator

```
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::IllegalElementException(ch);
      }
      return s;
}
```

back to the a buffer

singletons

an expression is
ended by a semicolon

token.cpp

Tokenization

```
// 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;
}
```

token reading

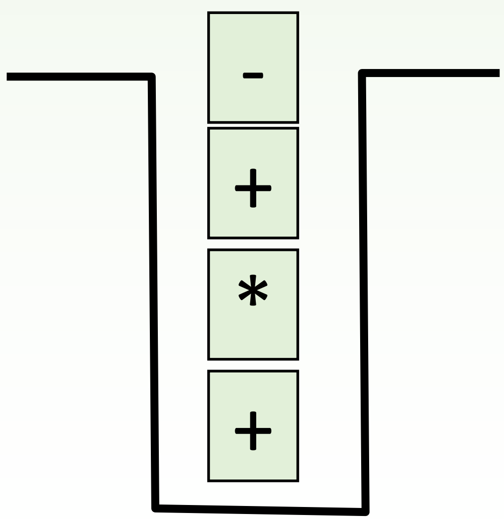
the reading throws it

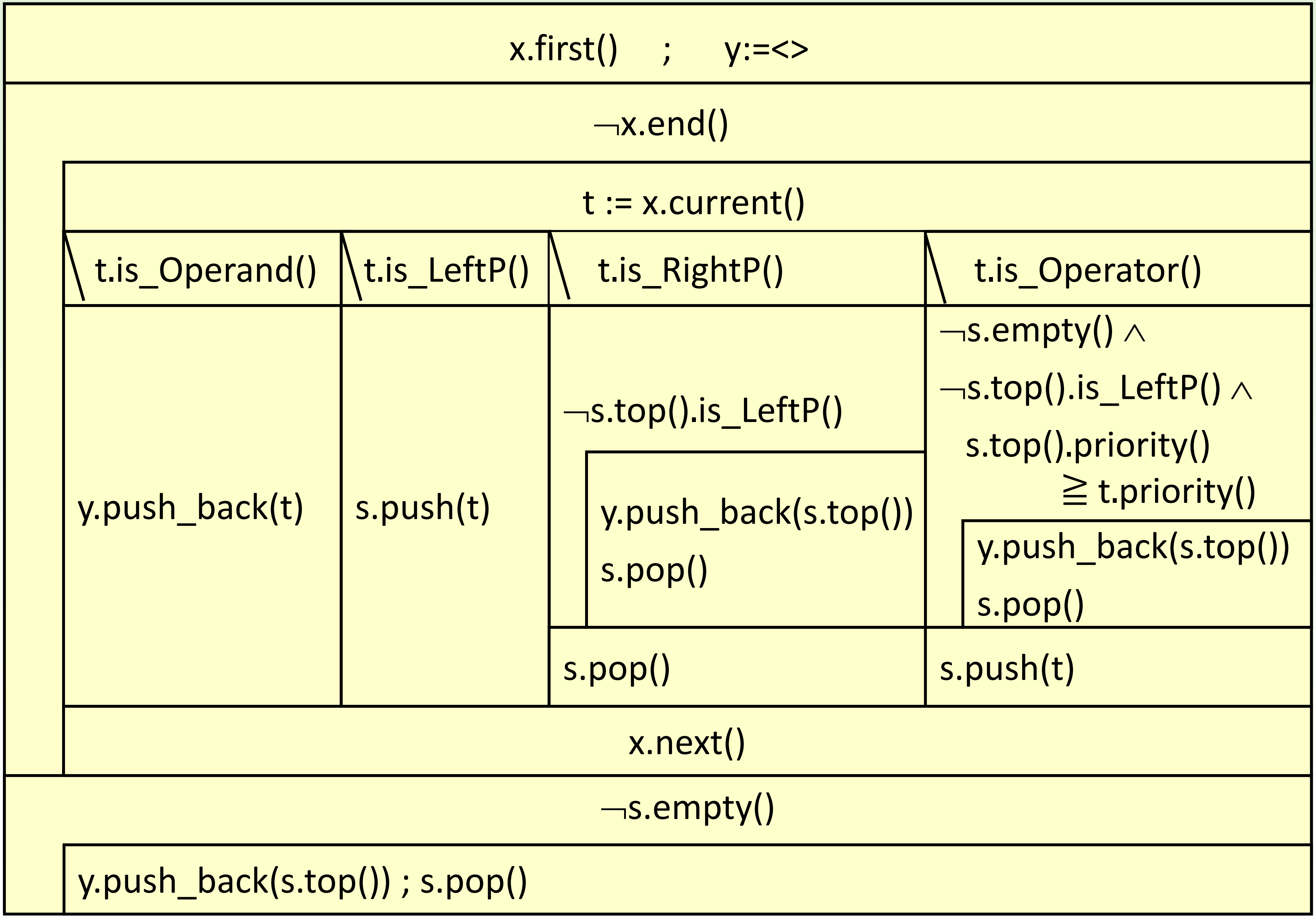
main.cpp

Convert to Polish notation

Left parentheses and operators are put into a stack. The operator with lower priority has to swap with the higher priority operators in the stack. Every other token is copied into the output sequence. In case of a right parenthesis, the content of the stack until the first left parenthesis is put into the output sequence. When we reach the end of the input sequence, the content of the stack is put into the output sequence.

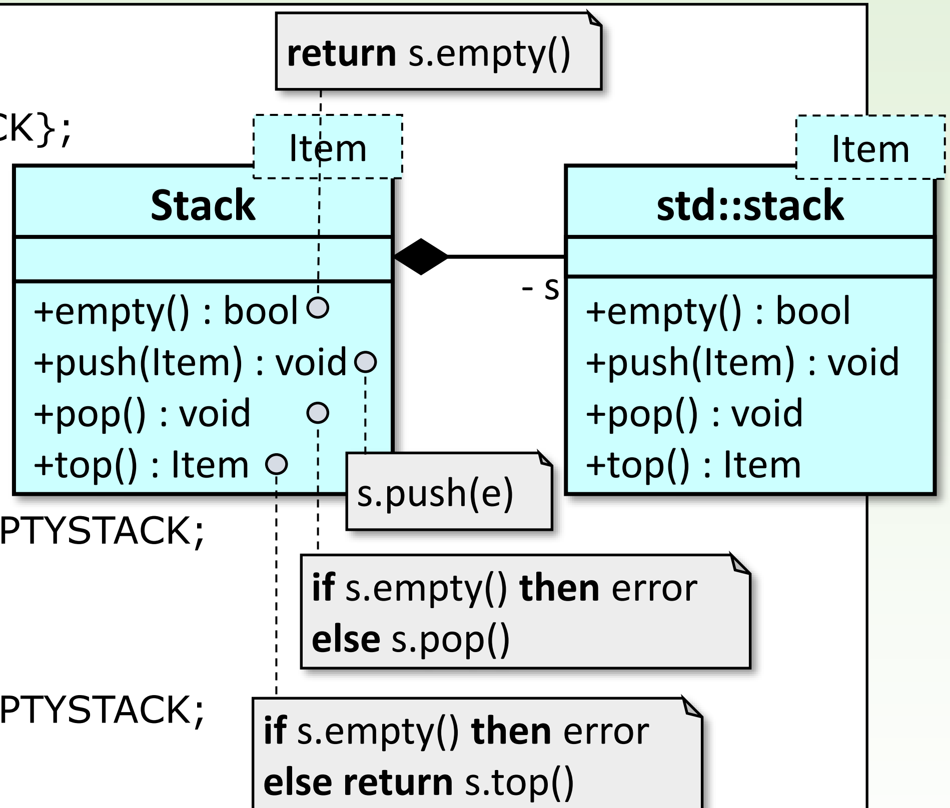
5	+	(	11	+	26	)	*	(	43	-	4	)




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_LeftP()
y.push_back(s.top())
s.pop()
s.pop()
¬s.empty() ∧
¬s.top().is_LeftP() ∧
s.top().priority()
≧ t.priority()
y.push_back(s.top())
s.pop()
s.push(t)
x.next()
¬s.empty()
y.push_back(s.top()) ; s.pop()

Template for the stack

```
#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); }
};
```



stack.hpp

Creating the postfix expression

```
// Transforming into polish form

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();
      }
}
```

enumeration

see the next slide

error when we have more left parentheses than right

main.cpp

Creating the postfix expression

```
if        ( t->is_Operand() )  y.push_back(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);
}
```

error when we have more right parentheses than left

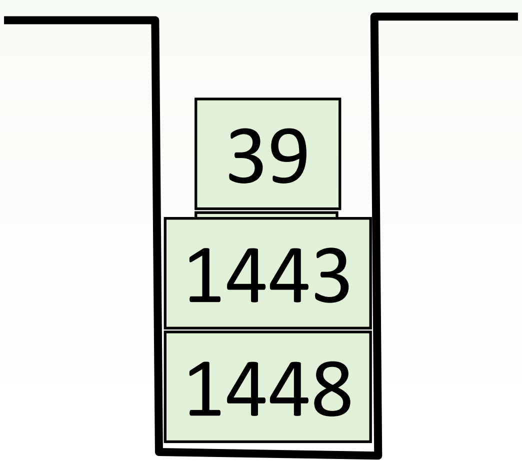
”static casting”: s.top()->priority() is not good, as in Token there is no priority().

main.cpp

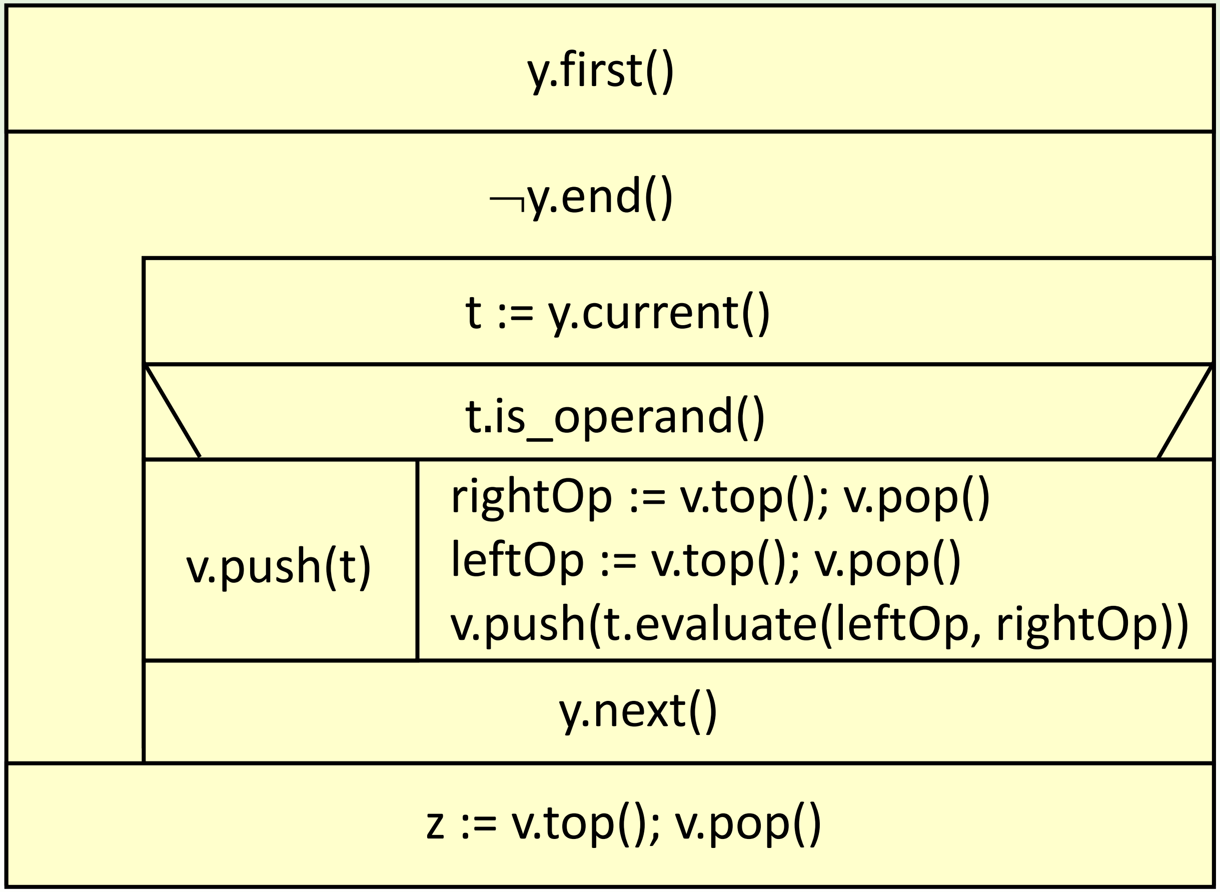
Evaluation of the postfix expression

Operands of the postfix expression (in order of reading) are put into a stack. In case of operator, the top two numbers are taken and processed according to the type of the operator. The result is put back into the stack. At the end of the process, the result can be found in the stack.

5	11	26	+	43	4	-	*	+



Evaluation


y.first()
¬y.end()
t := y.current()
t.is_operand()
v.push(t)
rightOp := v.top(); v.pop()
leftOp := v.top(); v.pop()
v.push(t.evaluate(leftOp, rightOp))
y.next()
z := v.top(); v.pop()

Evaluation

```
// 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 << "The value of the expression: " << result << endl;
}catch( StackExceptions ex ){
      if( ex==EMPTYSTACK ){
            cout << "Syntax error! ";
            throw Interrupt;
      }
}
```

enumeration

static casting

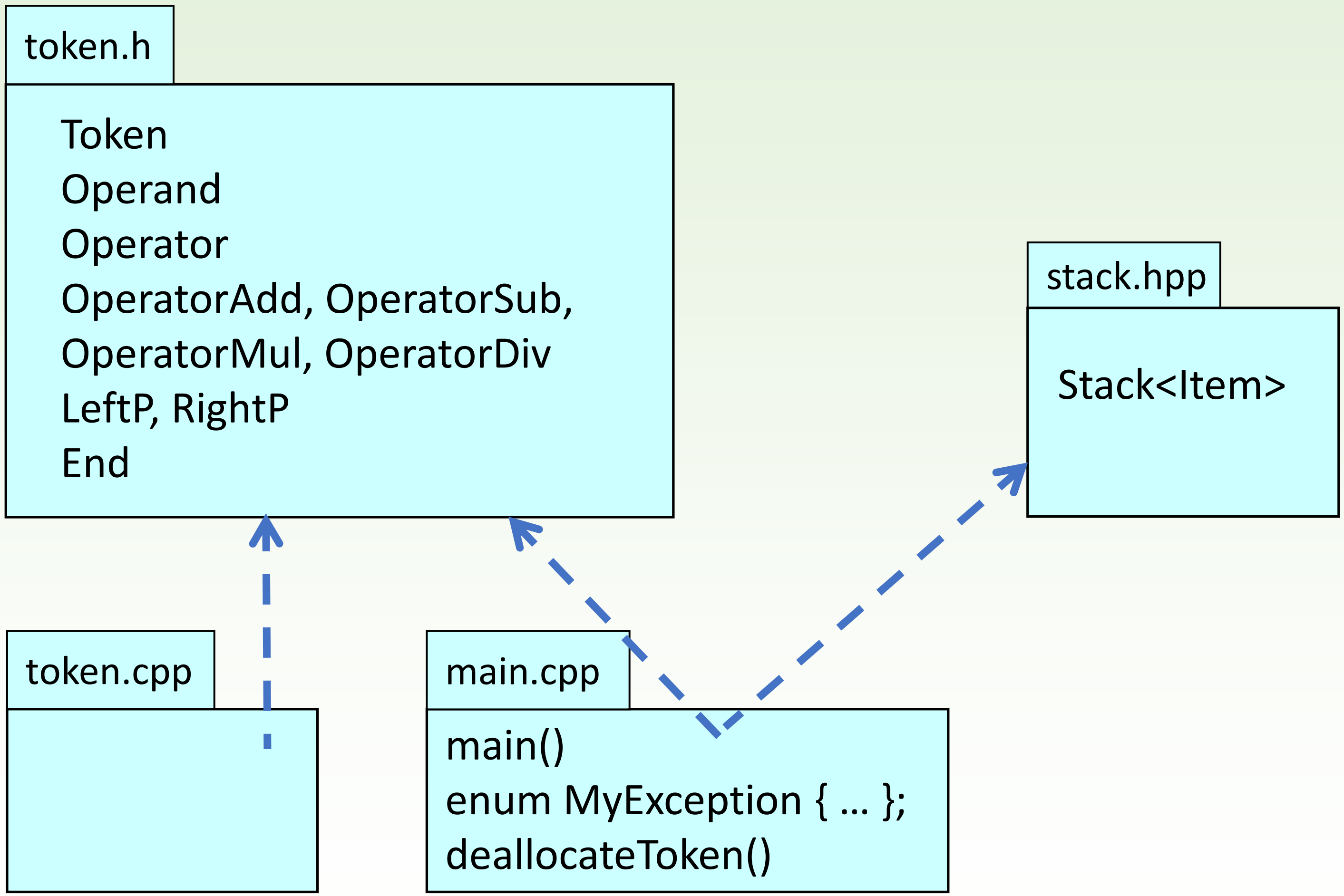
static casting

error when we have too many operands

error when we do not have enough operands

main.cpp

Package diagram

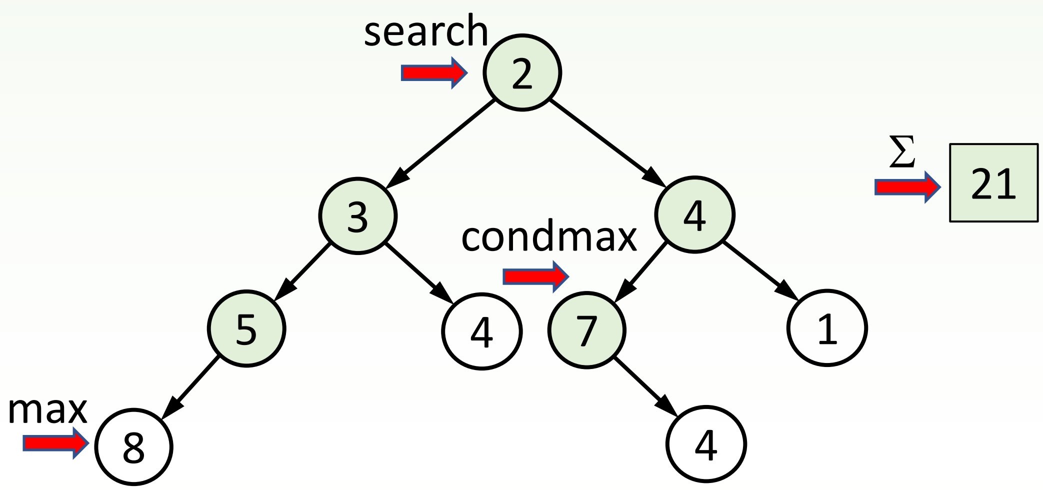


Task: traversal of a binary tree

Read some numbers from a standard input and build randomly a binary tree from them. Then print the tree’s elements to a standard output based on different traversal strategies. Finally,

- sum up the internal nodes,
- find the maximums of the internal nodes and all of the nodes,
- find the ”first” even element!

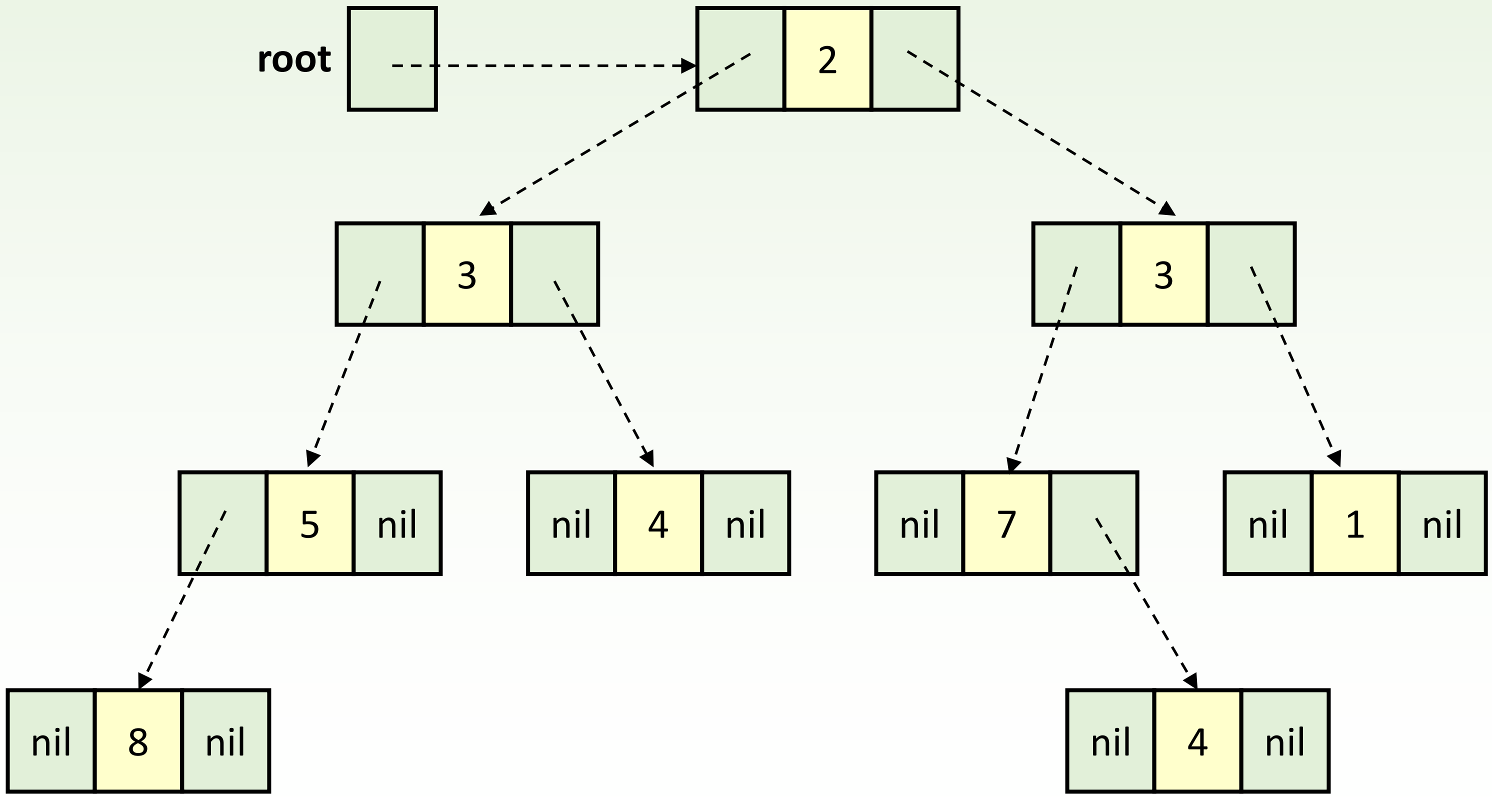
The tree is planned so that the summation, maximum search, and linear search are implemented easily. To be able to change the type of the nodes, the tree becomes a template.



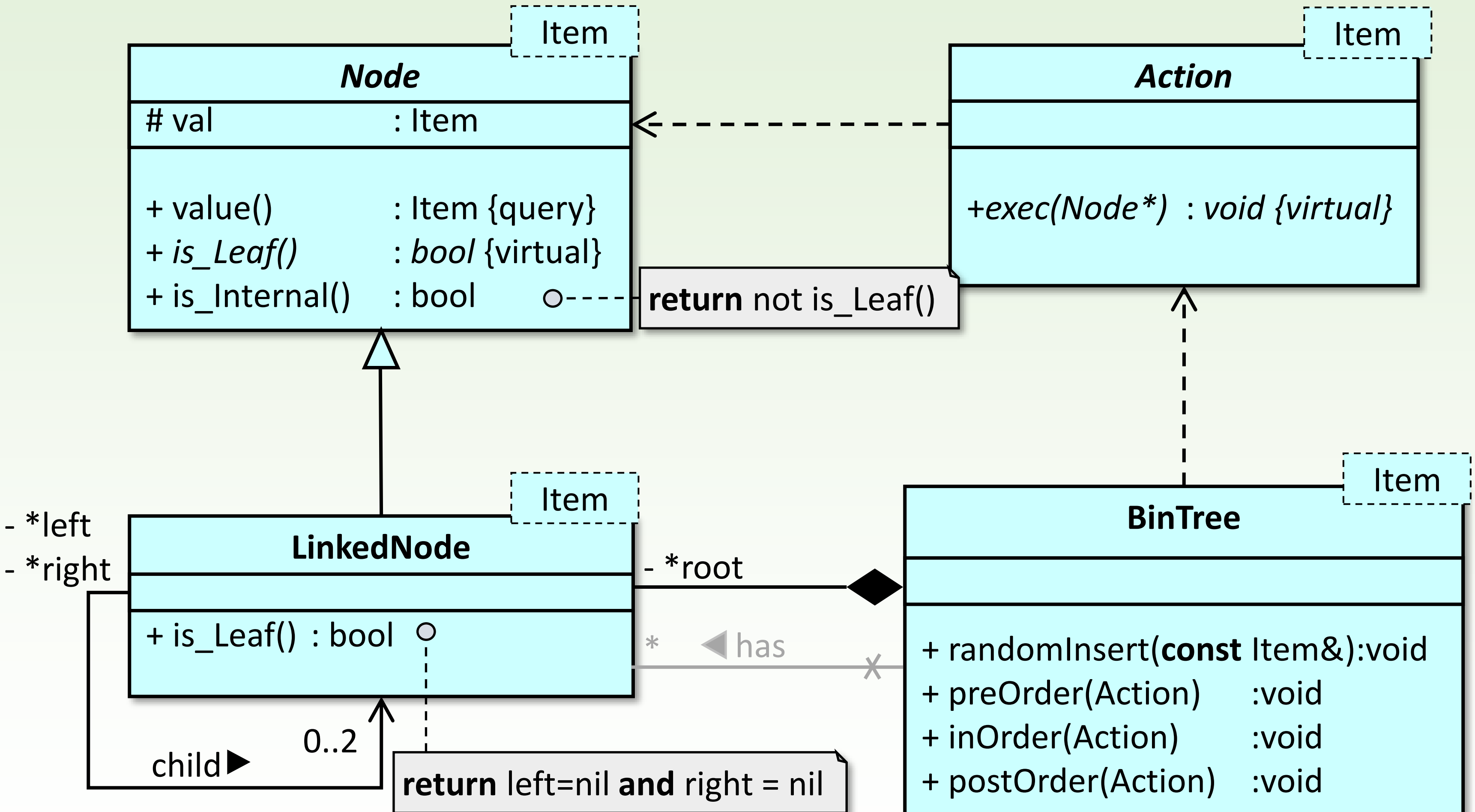
Chained binary tree

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

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



Class diagram



Template class of a node

```
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 override
                { return _left==nullptr && _right==nullptr; }
private:
        LinkedNode *_left;
        LinkedNode *_right;
};
```

BinTree class is defined after LinkedNode. To avoid circular reference, BinTree is just indicated here.

LinkedNode allows BinTree to see its private attributes and methods.

bintree.hpp

Template class of the binary tree

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

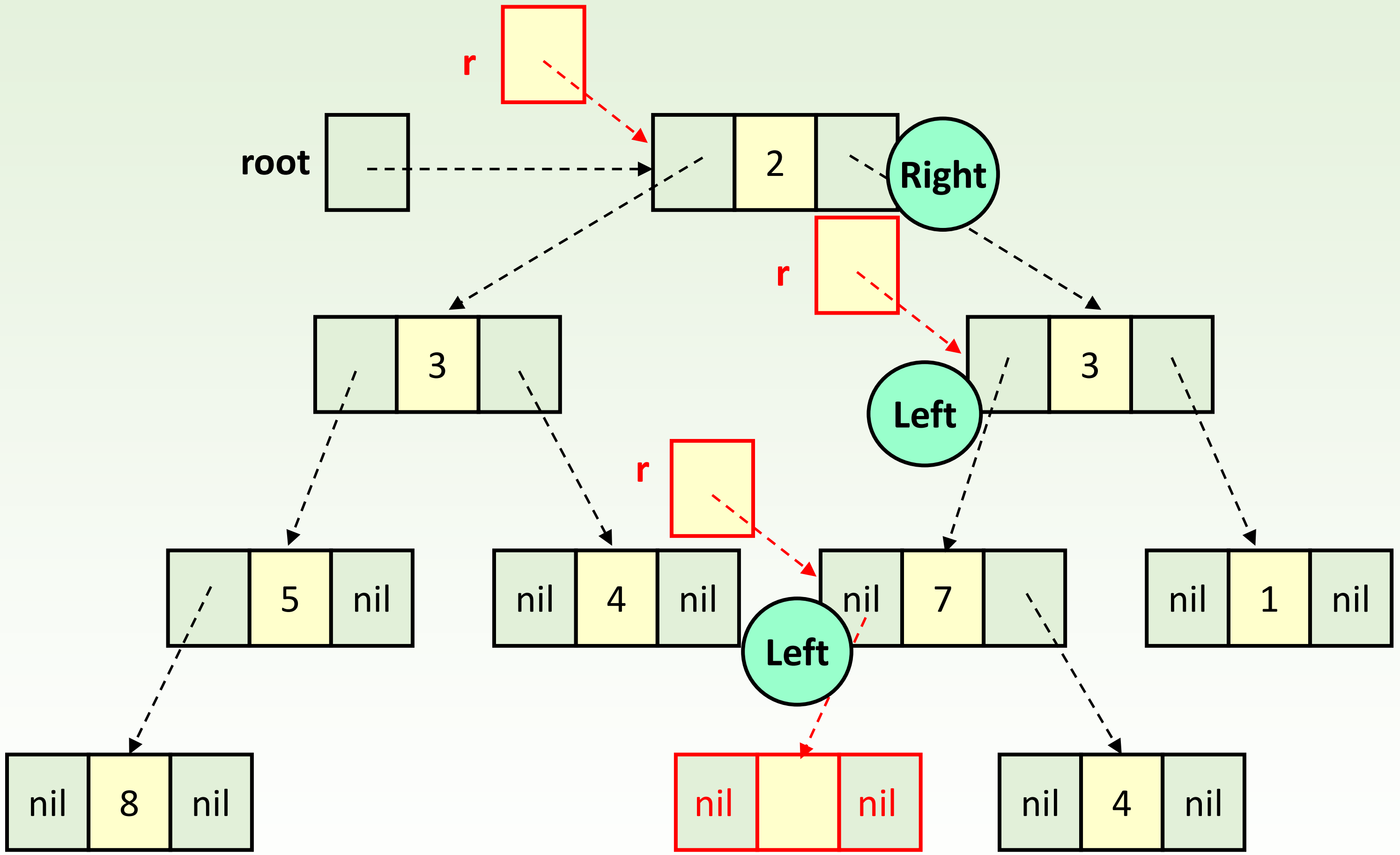
```
template <typename Item>
class BinTree{
public:
    BinTree():_root(nullptr) {srand(time(nullptr));}
    ~BinTree();

    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);
};
```

initialization of the random generator :
#include <time.h>
#include <cstdlib>

with a given action, starting from the root, they traverse the nodes

bintree.hpp

Inserting a new node into the tree



Inserting a new node into the tree

```
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);
    }
}
```

random generator

Is the last bit of the random number 1?

bintree.hpp

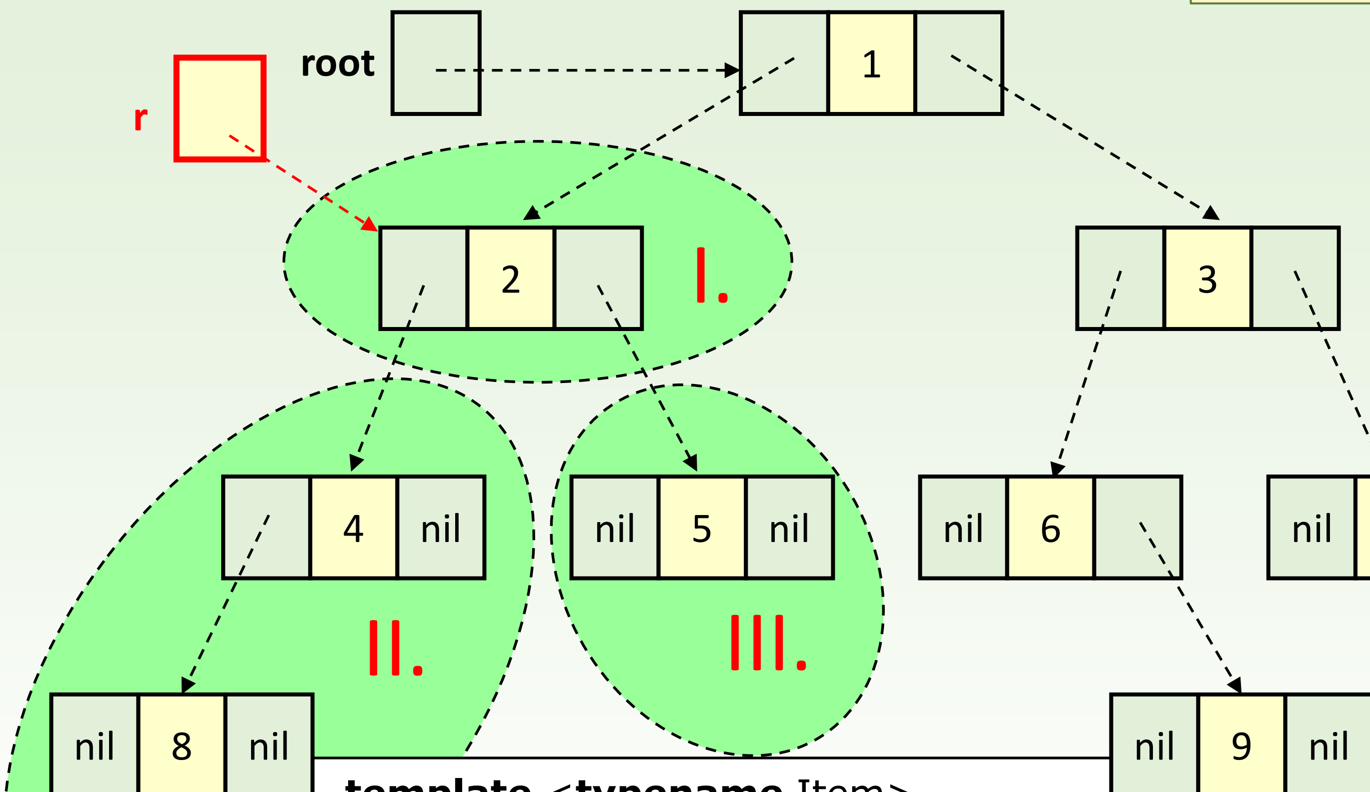
Building the tree

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

using namespace std;

int main()
{
	BinTree<int> t;
	int i;
	while(cin >> i){
		t.randomInsert(i);
	}

	return 0;
}
```

bintree.hpp

Preorder traversal

1 2 4 8 5 3 6 9 7

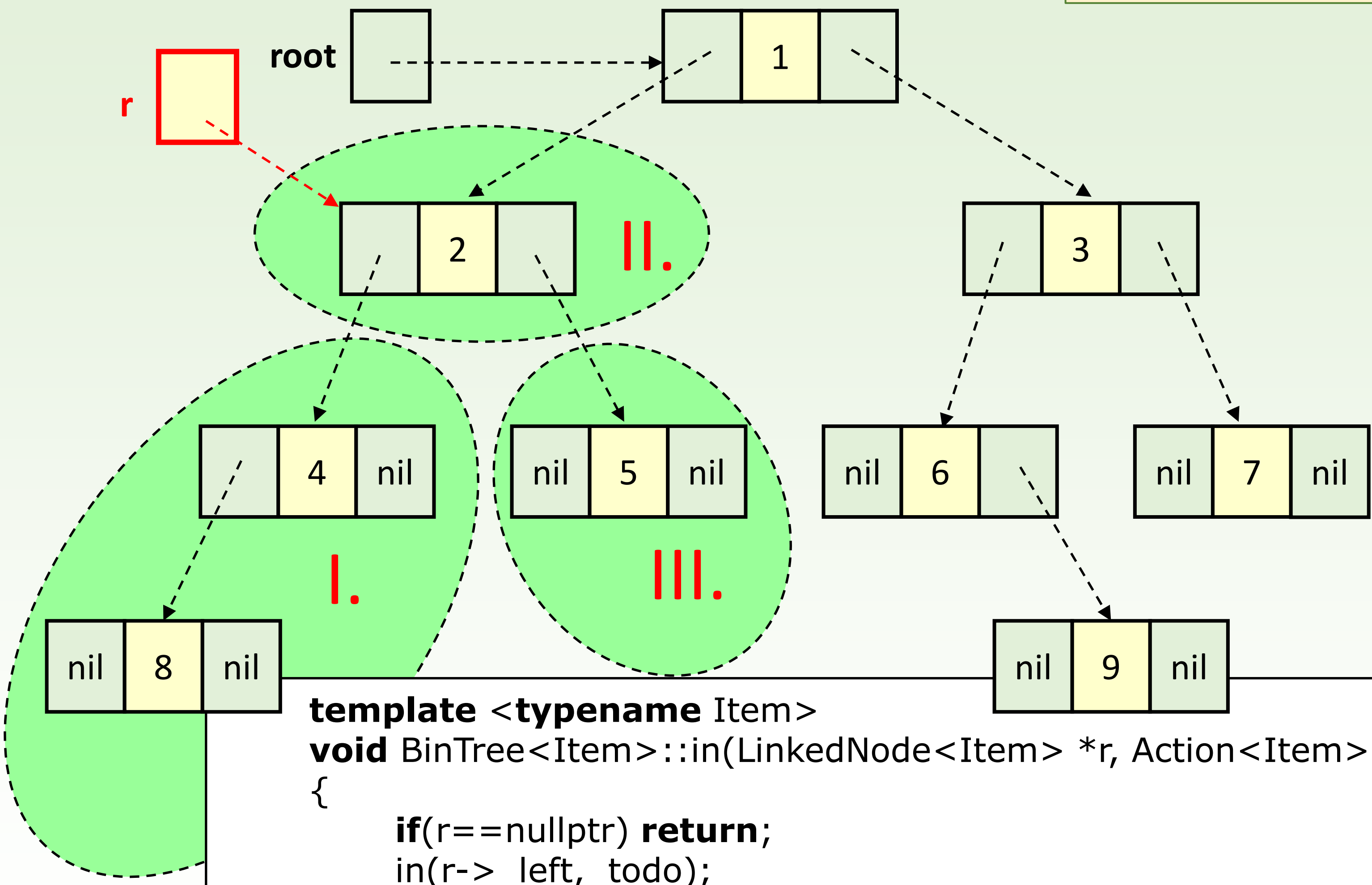



```
template <typename Item>
void BinTree<Item>::pre(LinkedNode<Item> *r, Action<Item> *todo)
{
        if(r==nullptr) return;
        todo->exec(r);
        pre(r->_left,  todo);
        pre(r->_right, todo);
}
```

bintree.hpp

Inorder traversal

8 4 2 5 1 6 9 3 7

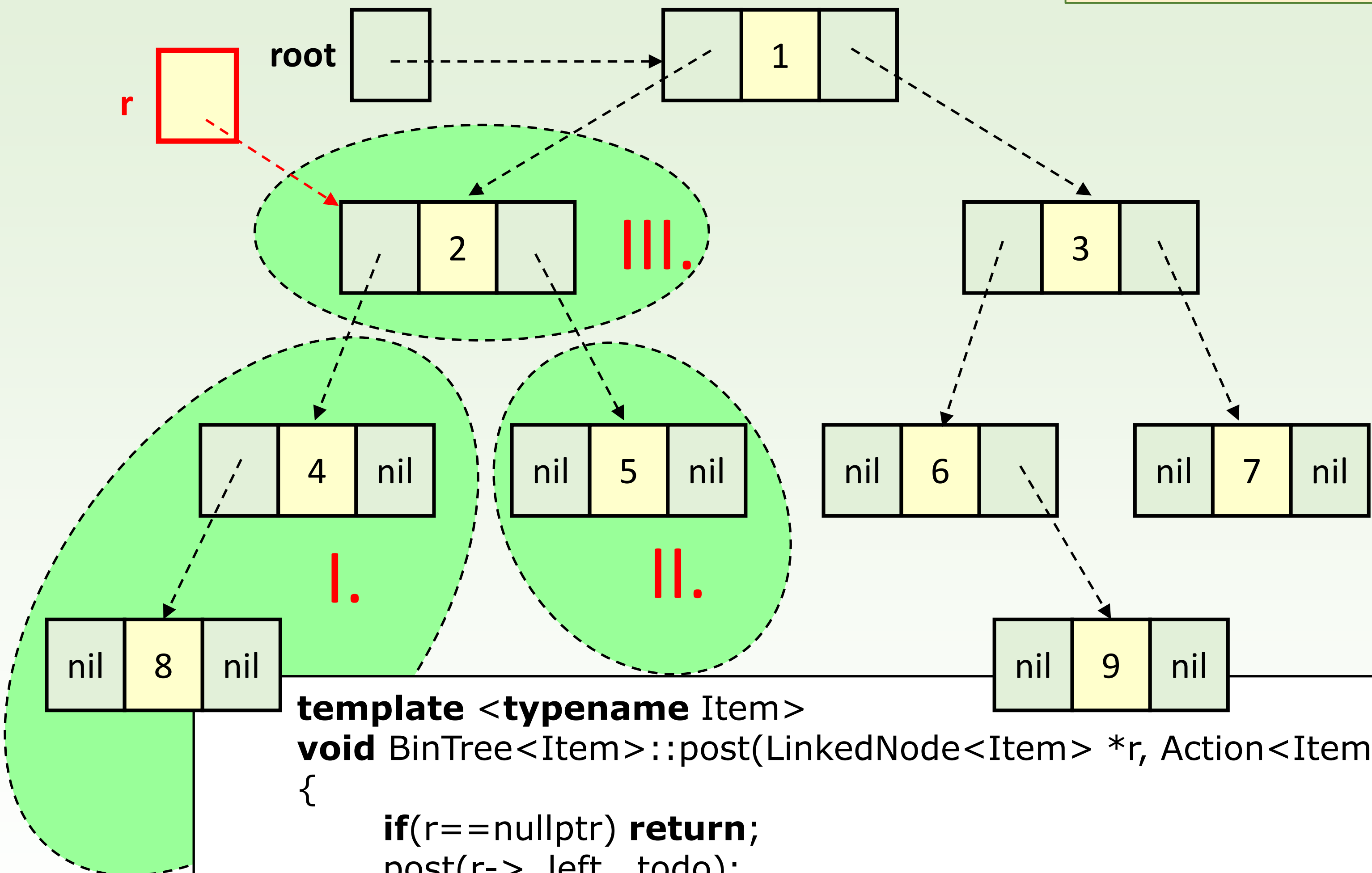



```
template <typename Item>
void BinTree<Item>::in(LinkedNode<Item> *r, Action<Item> *todo)
{
	if(r==nullptr) return;
	in(r->_left,  todo);
	todo->exec(r);
	in(r->_right, todo);
}
```

bintree.hpp

Postorder traversal

8 4 5 2 9 6 7 3 1



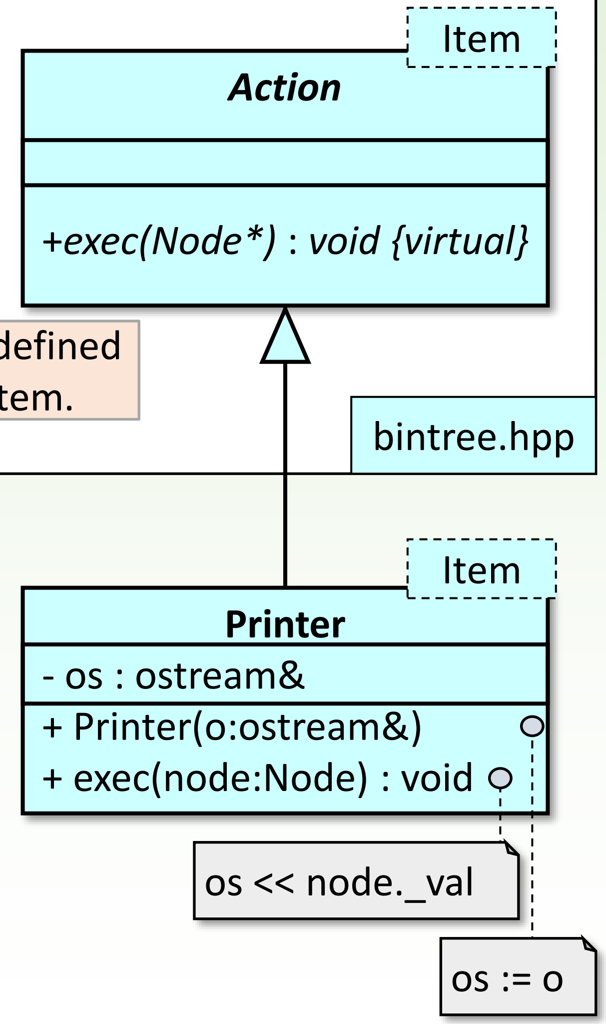

```
template <typename Item>
void BinTree<Item>::post(LinkedNode<Item> *r, Action<Item> *todo)
{
        if(r==nullptr) return;
        post(r->_left,  todo);
        post(r->_right, todo);
        todo->exec(r);
}
```

bintree.hpp

Template class of printing

```
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;
};
```

Output operator has to be defined for the datatype replacing Item.



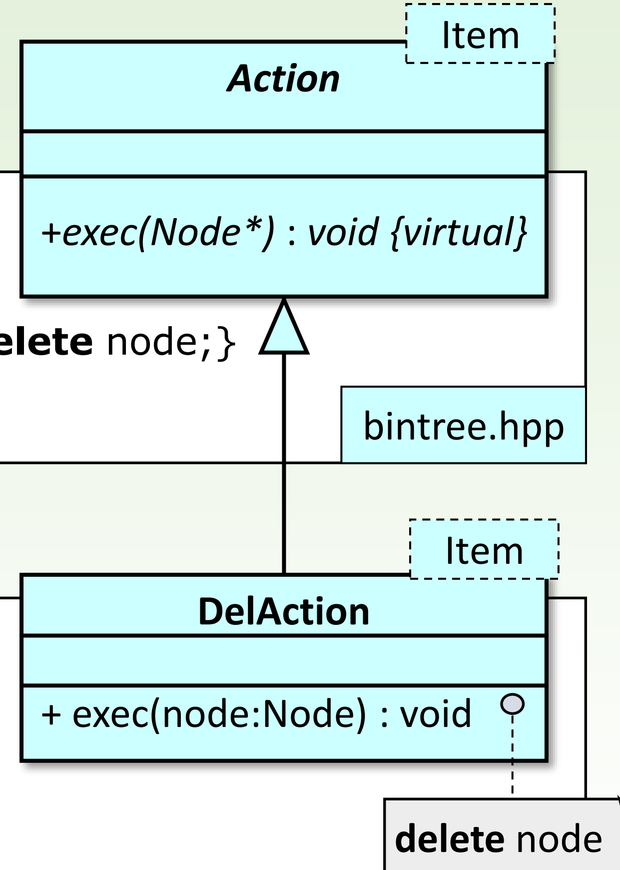

```
BinTree<int> t = build();

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;
```

Destructor: traversal with delete

in BinTree<Item>'s protected part.

```
protected:
  class DelAction: public Action<Item>{
  public:
        void exec(Node<Item> *node) override {delete node;}
  };
```

bintree.hpp

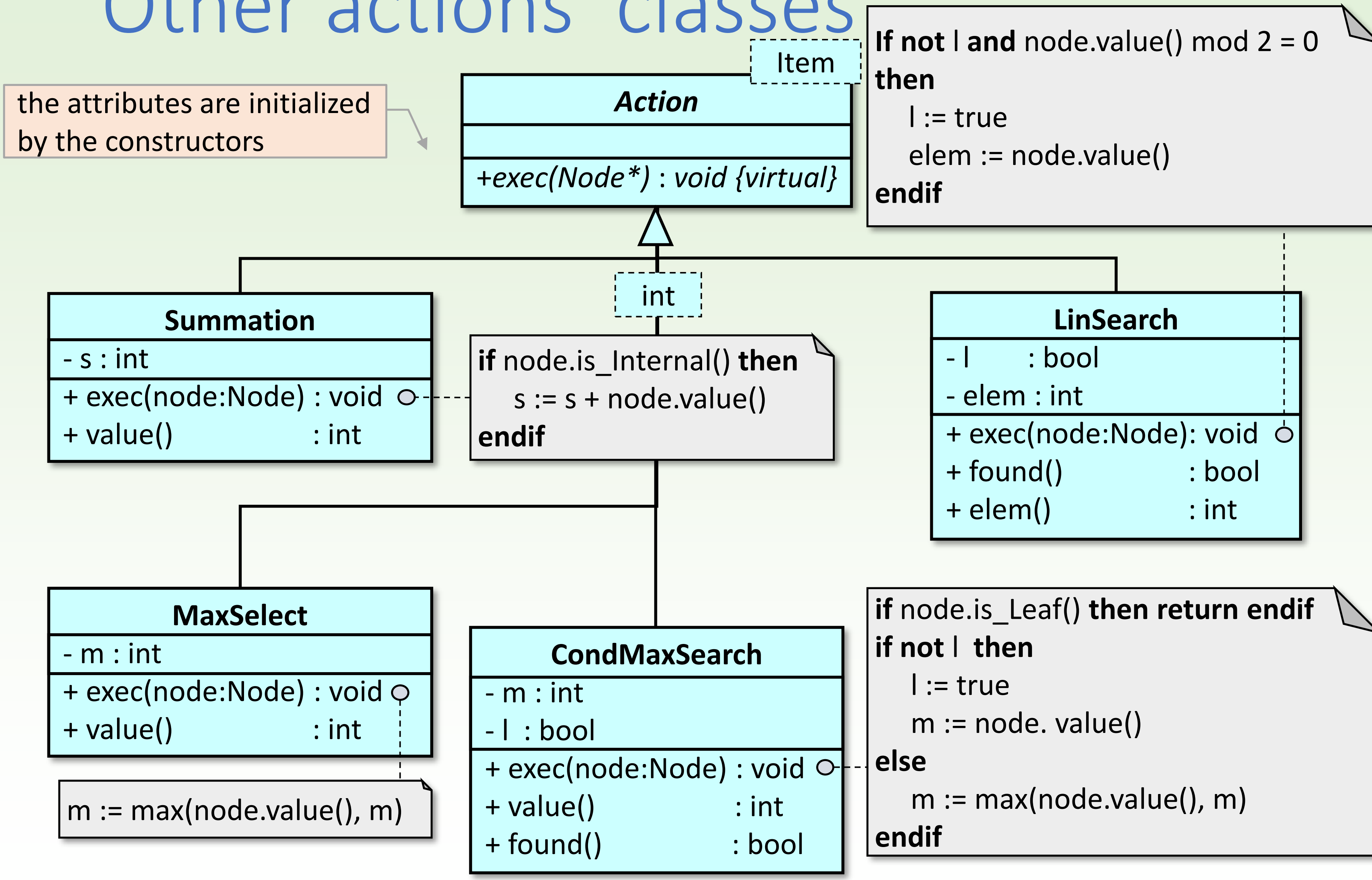



```
template <typename Item>
BinTree<Item>::~BinTree()
{
        DelAction del;
        post(_root, &del);
}
```

It works only with postorder!

bintree.hpp

Other actions’ classes



Summation

```
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;
```

Linear search

```
class LinSearch: public Action<int>{
public:
	void LinSearch():_l(false){}
	void exec(Node<int> *node) override {
		if (!l && node->value ()%2==0){
		   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 in the tree.";
```

Maximum search

```
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();
```

It shoud raise an exception if the tree is empty (without root)

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

bintree.hpp

Conditional maximum search

```
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(max.value().l) cout << max.value().m << endl;
else              cout << "none" << endl;
```