Programowanie Zintegrowane

Analiza przykładu - Case Study

Grażyna Mirkowska
Andrzej Salwicki
Oskar Świda

Polish-Japanese Institute of Computer Technologies,Warszawa

National Institute of Telecomunication, Warsaw

Białystok University of Technology
Department of Informatics

27.01.2009 UKSW & Concurrency, Specification and Programming - September 2008 - Groß Väter See

Plan

Bieżący rozdział

Konieczność gwarancji produktów programistycznych

Programy i moduły programów są tworzone i poddawane analizie, po to, by zapewnić iż są oprogramowaniem godnym zaufania.
Wielu klientów kupuje i stosuje oprogramowanie nie pytając o gwarancje jego jakości.
Coraz częściej jednak przemysł domaga się oprogramowania wysoce zintegrowanego, lub inaczej mówiąc oprogramowania wraz z gwarancją jego wysokiej jakości. Szczególnie dotyczy to wojskowości, lotnictwa cywilnego, przemysłu kosmicznego, bankowości i wielu innych dziedzin.
Przemysł sformułował szereg wymagań: jedno z nich (poziomu bodaj piątego) brzmi "program ma być dostarczony wraz z dowodem jego poprawności, kompilacja programu ma być dokonana przy pomocy kompilatora, którego poprawność została udowodniona". Co oznacza zdanie: "program ma być dostarczony wraz z dowodem poprawności¿'
Może okazać się, że dla 12 stronicowego programu jego dowód to ponad 100 stron druku.

test or prove?

There are two ways: either test it or prove it. Most of software companies believes in testing, some companies require proving. We are arguing that the proper methodology consists in *experimenting, observing and proving*.

Testing or proving?

- testing?

Testing or proving?

- testing? - *thesis*,
- proving?

Testing or proving?

- testing? - *thesis*,
- proving? - *antithesis*,
- experimenting, observing, proving! - *synthesis*

In the presented paper we argue[1] that the two approaches may be synthesized to a completely different scheme of practice. We propose to replace the term testing by another word *experimenting* with the program. Testing limits itself to the execution of program and the comparison of the results with the predicted, supposedly correct data. For the majority of people testing is the practice of searching bugs in software. Experimenting has larger horizons, it covers not only the search of counter-examples (aka errors) but also gathering of positive evidence. Sometimes during experimentation we begin to believe that objects obey certain rule and during the further experiments we try to find further evidence confirming our beliefs. During experiments one is going to execute program with different data, to assemble data and to present them in graphical mode, in tables, in data bases, etc. It is suggested that the experimentation were done with some plan. Next, one should analyze the gathered experience and search some regularities. It will lead to the formulation of lemmas and propositions. After this is done, there is the time of proving the hypotheses. Obviously, the process sketched above may need to be iterated

for different reasons. E.g. when our program is modified. We consider the method given below

$$\textit{experiment} \longrightarrow \textit{observe} \longrightarrow \textit{formulate hypotheses} \longrightarrow \textit{prove}.$$

as a proper approach to the production of the high integrity software.

[1] following to some extent the ideas of Georg Hegel who used to say that from a pair: thesis and antithesis we should make a synthesis

Nasza wizja

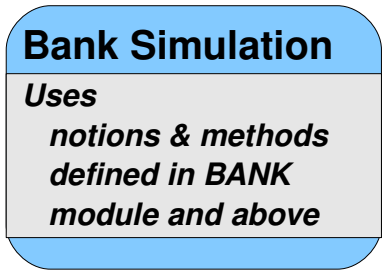
Biblioteki procedur i klas, ale także biblioteki wiedzy o tych klasach i procedurach.
Środowisko wspomagające pracę programistów:
nawigacja w bibliotekach (sieć!),
narzędzia do eksperymentów,
narzędzia wspomagające dowodzenie,

Bieżący rozdział

Zanim bank otworzy nowy oddział w ...

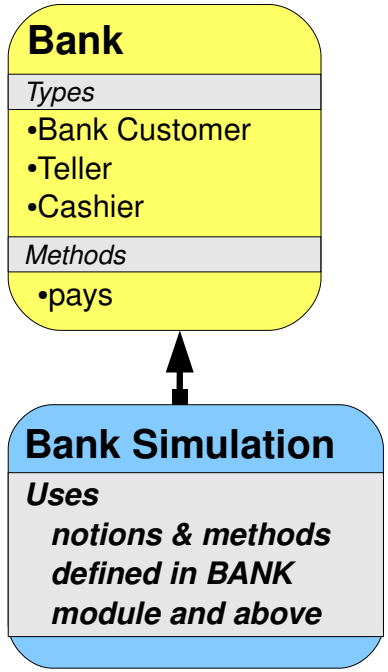
Dyrektor banku Fargo może zechcieć zbadać jak duży ma być nowy oddział, ilu ma mieć pracowników? etc.
Być może zwróci się do firmy software'owej: Proszę wykonać symulację kilku wariantów działania nowego oddziału banku w miejscowości ...

Początek - zadanie symulacji banku



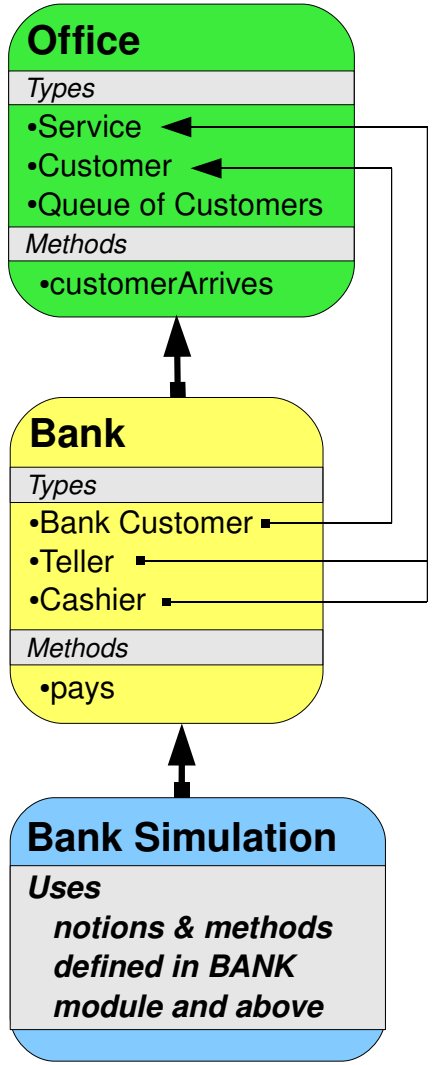
An example of application

Faktoryzacja zadania: krok 1



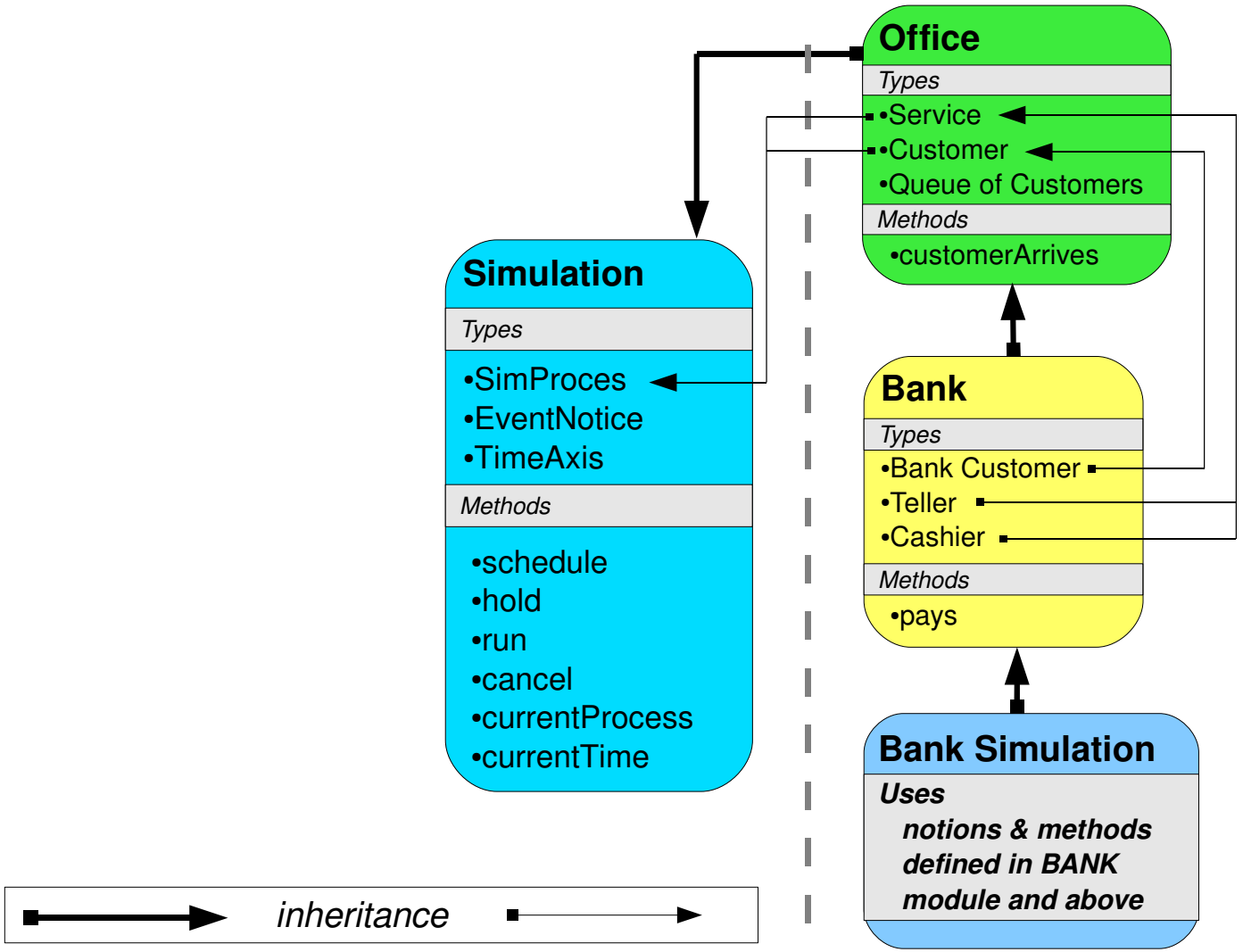
An example of application

Faktoryzacja zadania: krok 2



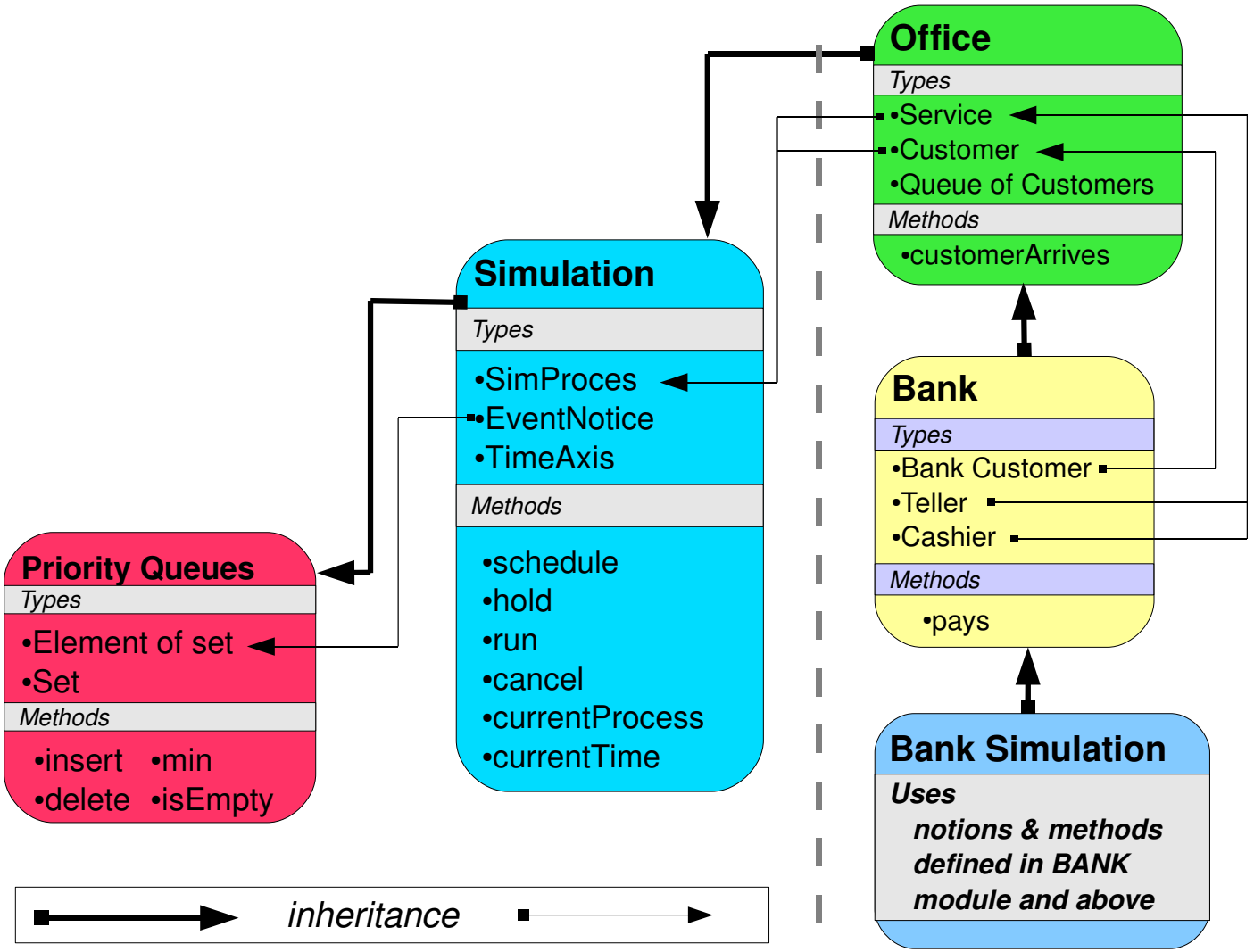
An example of application

Faktoryzacja zadania: krok 3



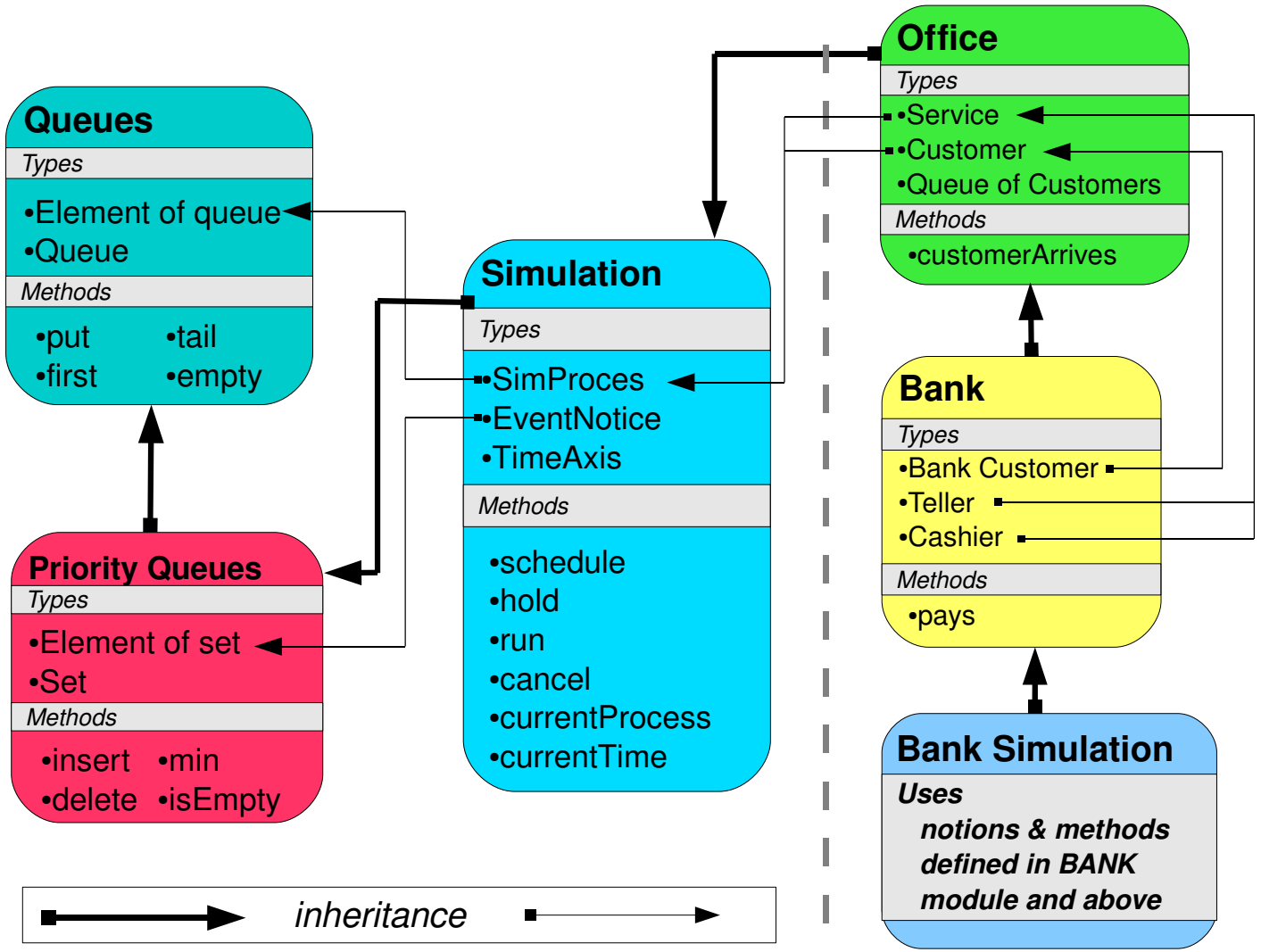
Structure of class Simulation **An example of application**

Faktoryzacja zadania: krok 4



Structure of class Simulation **An example of application**

Faktoryzacja zadania: krok 5



Structure of class Simulation **An example of application**

structure of Simulation program

```
class Queues { ...
    class QueueElem { ... }
} //end Queues
class PQS extends Queues { //system of priority queues
    class Elem {Node lab; ... } // end Elem
    private class Node { Elem el; Node up, left, right; ... } // end Node
    class PQ {
        Node root, last;
        Elem min () { ... }
        void insert(Elem e) { ... }
        void delete (Elem e) { ... }
        private void correct (Elem e, boolean upDown) { ... }
    } //end PQ
} //end PQS
class Simulation extends PQS { ...
    class Eventnotice extends Elem  { ,,,}
    PQ plan; //a heap of eventnotices
    class Simproces extends QueueElem  { ...
} // end Simulation
```

Bieżący rozdział

Signature and Axioms of the structure of FIFO **queues**

Signature	Comments
Sorts	$Universe = EQ \cup Q$
EQ	set of queueable elements
Q	set of queues
Operations	
$put : EQ \times Q \longrightarrow Q$	put element e into queue q
$out : EQ \times Q \longrightarrow Q$	delete the first element from queue q
$fr : Q \longrightarrow E$	the first element of queue q
$empty : Q \longrightarrow \{true, false\}$	is the queue q empty?
$=_{E} : EQ \times EQ \longrightarrow \{true, false\}$	the equality of elements
$=_{Q} : Q \times Q \longrightarrow \{true, false\}$	the equality of queues

Signature and Axioms of the structure of FIFO **queues**

Signature	Comments
Sorts	$Universe = EQ \cup Q$
EQ	set of queueable elements
Q	set of queues
Operations	
$put : EQ \times Q \longrightarrow Q$	put element e into queue q
$out : EQ \times Q \longrightarrow Q$	delete the first element from queue q
$fr : Q \longrightarrow E$	the first element of queue q
$empty : Q \longrightarrow \{true, false\}$	is the queue q empty?
$=_{E} : EQ \times EQ \longrightarrow \{true, false\}$	the equality of elements
$=_{Q} : Q \times Q \longrightarrow \{true, false\}$	the equality of queues
Axioms	
a1) $(\forall q \in Q)(\forall e \in EQ)\ \neg empty(put(e, q))$.	

Signature and Axioms of the structure of FIFO **queues**

Signature	Comments
Sorts	$Universe = EQ \cup Q$
EQ	set of queueable elements
Q	set of queues
Operations	
$put : EQ \times Q \longrightarrow Q$	put element e into queue q
$out : EQ \times Q \longrightarrow Q$	delete the first element from queue q
$fr : Q \longrightarrow E$	the first element of queue q
$empty : Q \longrightarrow \{true, false\}$	is the queue q empty?
$=_{E} : EQ \times EQ \longrightarrow \{true, false\}$	the equality of elements
$=_{Q} : Q \times Q \longrightarrow \{true, false\}$	the equality of queues

Axioms

a1) $(\forall q \in Q)(\forall e \in EQ)\ \neg empty(put(e, q))$.
a2) $(\forall q \in Q)(\forall e \in EQ)(empty(q) \Rightarrow (e =_{E} fr(put(e, q))))$.
a3) $(\forall q \in Q)(\forall e \in EQ)(\neg empty(q) \Rightarrow (fr(q) =_{E} fr(put(e, q))))$.

Signature and Axioms of the structure of FIFO **queues**

Signature	Comments
Sorts	$Universe = EQ \cup Q$
EQ	set of queueable elements
Q	set of queues
Operations	
$put : EQ \times Q \longrightarrow Q$	put element e into queue q
$out : EQ \times Q \longrightarrow Q$	delete the first element from queue q
$fr : Q \longrightarrow E$	the first element of queue q
$empty : Q \longrightarrow \{true, false\}$	is the queue q empty?
$=_{E}: EQ \times EQ \longrightarrow \{true, false\}$	the equality of elements
$=_{Q}: Q \times Q \longrightarrow \{true, false\}$	the equality of queues

Axioms

a1) $(\forall q \in Q)(\forall e \in EQ)\ \neg empty(put(e, q))$.
a2) $(\forall q \in Q)(\forall e \in EQ)(empty(q) \Rightarrow (e =_{E} fr(put(e, q))))$.
a3) $(\forall q \in Q)(\forall e \in EQ)(\neg empty(q) \Rightarrow (fr(q) =_{E} fr(put(e, q))))$.
a4) $(\forall q \in Q)(\forall e \in EQ)(empty(q) \Rightarrow (q =_{Q} out(put(e, q))))$.
a5) $(\forall q \in Q)(\forall e \in EQ)(\neg empty(q) \Rightarrow (put(e, out(q)) =_{Q} out(put(e, q))))$

Signature and Axioms of the structure of FIFO **queues**

Signature	Comments
Sorts	$Universe = EQ \cup Q$
EQ	set of queueable elements
Q	set of queues
Operations	
$put : EQ \times Q \longrightarrow Q$	put element e into queue q
$out : EQ \times Q \longrightarrow Q$	delete the first element from queue q
$fr : Q \longrightarrow E$	the first element of queue q
$empty : Q \longrightarrow \{true, false\}$	is the queue q empty?
$=_{E}: EQ \times EQ \longrightarrow \{true, false\}$	the equality of elements
$=_{Q}: Q \times Q \longrightarrow \{true, false\}$	the equality of queues

Axioms

a1) $(\forall q \in Q)(\forall e \in EQ)\ \neg empty(put(e, q))$.
a2) $(\forall q \in Q)(\forall e \in EQ)(empty(q) \Rightarrow (e =_{E} fr(put(e, q))))$.
a3) $(\forall q \in Q)(\forall e \in EQ)(\neg empty(q) \Rightarrow (fr(q) =_{E} fr(put(e, q))))$.
a4) $(\forall q \in Q)(\forall e \in EQ)(empty(q) \Rightarrow (q =_{Q} out(put(e, q))))$.
a5) $(\forall q \in Q)(\forall e \in EQ)(\neg empty(q) \Rightarrow (put(e, out(q)) =_{Q} out(put(e, q))))$
a6) $(\forall q \in Q)$ **while** $\neg empty(q)$ **do** $q := out(q)$ **done** true.
This axiom says: for all q program halts, i.e. *the queue q is finite*

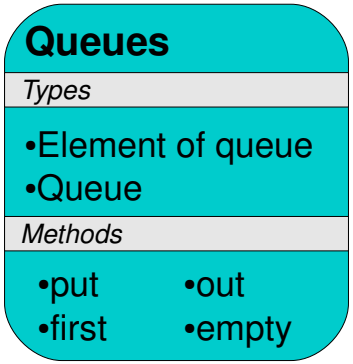
Signature and Axioms of the structure of FIFO **queues**

Signature	Comments
Sorts	$Universe = EQ \cup Q$
EQ	set of queueable elements
Q	set of queues
Operations	
$put : EQ \times Q \longrightarrow Q$	put element e into queue q
$out : EQ \times Q \longrightarrow Q$	delete the first element from queue q
$fr : Q \longrightarrow E$	the first element of queue q
$empty : Q \longrightarrow \{true, false\}$	is the queue q empty?
$=_{\boldsymbol{E}}: EQ \times EQ \longrightarrow \{true, false\}$	the equality of elements
$=_{\boldsymbol{Q}}: Q \times Q \longrightarrow \{true, false\}$	the equality of queues

Axioms

a1) $(\forall q \in Q)(\forall e \in EQ)\ \neg empty(put(e, q))$.
a2) $(\forall q \in Q)(\forall e \in EQ)(empty(q) \Rightarrow (e =_{\boldsymbol{E}} fr(put(e, q))))$.
a3) $(\forall q \in Q)(\forall e \in EQ)(\neg empty(q) \Rightarrow (fr(q) =_{\boldsymbol{E}} fr(put(e, q))))$.
a4) $(\forall q \in Q)(\forall e \in EQ)(empty(q) \Rightarrow (q =_{\boldsymbol{Q}} out(put(e, q))))$.
a5) $(\forall q \in Q)(\forall e \in EQ)(\neg empty(q) \Rightarrow (put(e, out(q)) =_{\boldsymbol{Q}} out(put(e, q))))$
a6) $(\forall q \in Q)$ **while** $\neg empty(q)$ **do** $q := out(q)$ **done** true.
This axiom says: for all q program halts, i.e. *the queue q is finite*
a7) $q =_{\boldsymbol{Q}} q' \equiv$ **begin**
$s1 := q; s2 := q'; result := true;$
while $\neg empty(s1) \wedge \neg empty(s2) \wedge result$ **do**
if $fr(s1) \neq fr(s2)$ **then** $result := false$ **fi**;
$s1 := out(s1); s2 := out(s2)$
done
end $(result \wedge empty(s1) \wedge empty(s2))$

Teraz wiemy jakie cechy ma mieć moduł QueuesFIFO



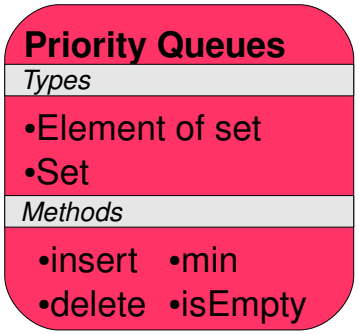
struktura klasy Queues

```
class Queues { //system zwykłych kolejek
      class ElemQ { ... } // end ElemQ
      class Q { ... } // end kolejka
           ElemQ first (Q q) { ... }
           Q insert(ElemQ e, Q q) { ... }
           Q delete (ElemQ e, Q q) { ... }
           Boolean empty (Q q) { ... }
      } //end PQ
} //end Queues
```

Pytanie:
Czy instrukcje, tu reprezentowane przez wielokropki {...}, zapewniają spełnienie własności wyliczonych wcześniej jako aksjomaty kolejekFIFO?

Bieżący rozdział

Zastanówmy się jakie cechy ma mieć klasa PQS



the structure of the class PQS

```
class PQS extends Queues { //system of priority queues
      class Elem {Node lab; ... } // end Elem
      private class Node { Elem el; Node up, left, right; ...}//end Node
      class PQ {
            Node root, last;
            Elem min () { ... }
            void insert(Elem e) { ... }
            void delete (Elem e) { ... }
            private void correct (Elem e, boolean upDown) { ... }
      } //end PQ
} //end PQS
```

Bieżący rozdział

Signature and Axioms

Signature	Comments
Sorts	$Universe = E \cup PQ$
E	set of elements
PQ	set of priority queues
Operations	
$insert : E \times PQ \longrightarrow PQ$	put element e into priority queue q
$delete : E \times PQ \longrightarrow PQ$	delete element e from q
$min : PQ \longrightarrow E$	find the minimum element of q
$empty : PQ \longrightarrow \{true, false\}$	is priority queue q empty?
$member : E \times PQ \longrightarrow \{true, false\}$	does $e \in q$?
$\leq : E \times E \longrightarrow \{true, false\}$	the ordering relation

Axioms

a1) *The set Elem is linearly ordered by the relation* $\leq$.

a2) [**while** *not empty*(q) **do** $q :=$ *delete*$(min(q), q)$ **done**] *true*
This axiom says for all q program halts, i.e. *the priority queue q is finite*

a3) $[q1 := insert(e, q)]\{member(e, q1) \wedge (\forall_{e1 \neq e}\ member(e1, q1) \Leftrightarrow member(e1, q))\}$

a4) $[q1 := delete(e, q)]\{\neg member(e, q1) \wedge (\forall_{e1 \neq e}\ member(e1, q1) \Leftrightarrow member(e1, q))\}$

a5) $empty(q) \Rightarrow (\forall_{e \in E}\ \neg member(e, q))$

a6) $\neg empty(q) \Rightarrow (\forall_{e \in E}\ member(e, q) \Rightarrow min(q) \leq e))$
The operation *min* finds the least element of the set q.

a7) $[e := min(q)]true \Leftrightarrow \neg empty(q)$
Axiom a7 says the result of expression $min(q)$ is defined iff $\neg empty(q)$

a8) $member(e, q) \Leftrightarrow$ **begin**
s1 := q; result := false;
while not *empty(s1)* **and not** *result* **do**
if *e =min(s1)* **then** *result*:=true **fi**;
s1 := delete(min(s1),s1)
done
end *result*

Teorie i moduły programu

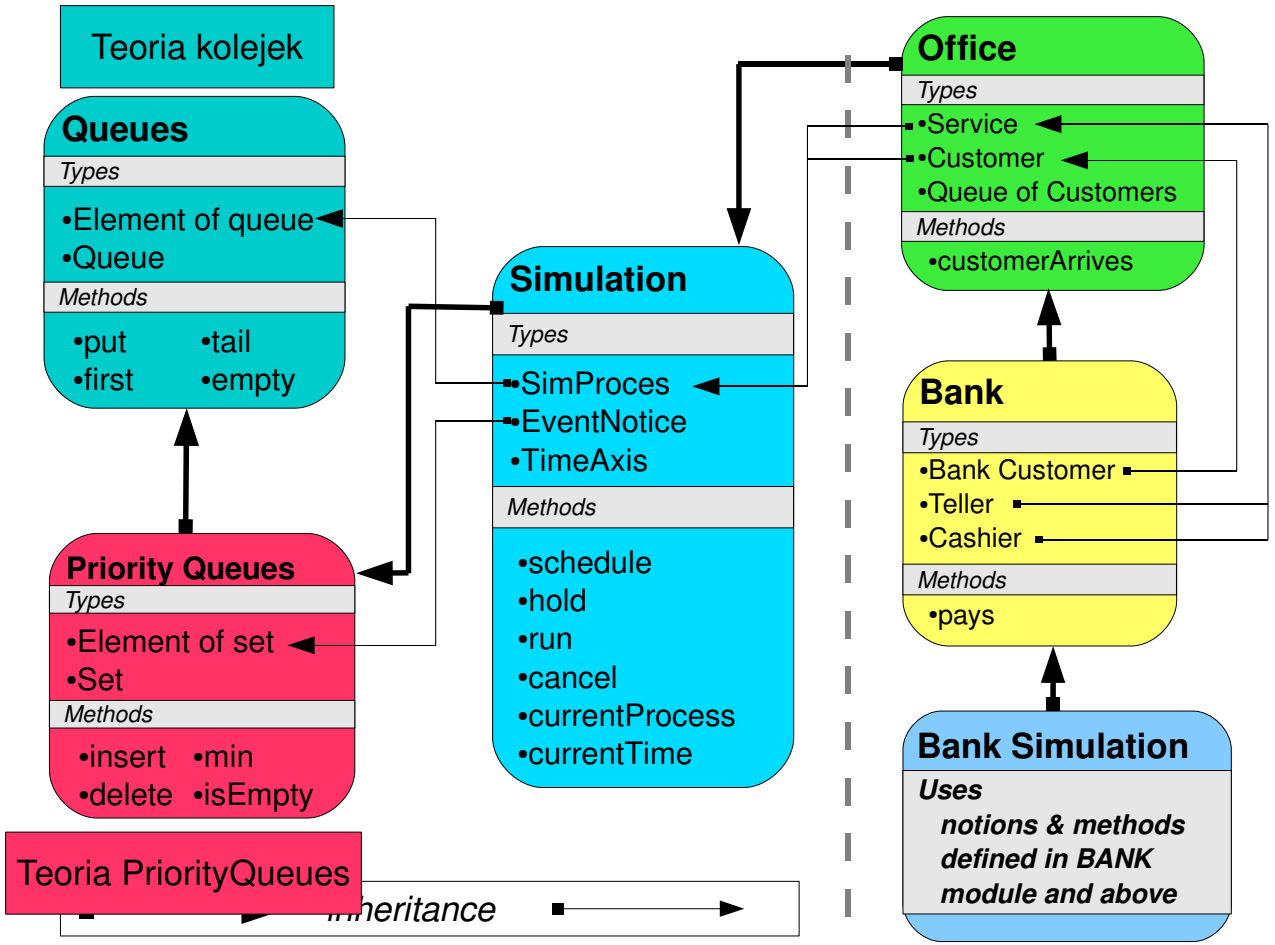


Figure: modułom programu - klasom odpowiadają teorie algorytmiczne czy klasa jest modelem teorii? czy teoria jest zupełna? rozstrzygalna? ...

How we did the verification?

We did the following:

1. Experiments - we executed methods of the class by hand and have drawn some pictures,
2. Observations - we analyzed the pictures and searched for some regularities,
3. Conclusions - we formulated several hypotheses(lemmas) and propositions,
4. Proving - we proved the lemmas.
5. Finally we obtained the correctness theorem.

Bieżący rozdział

Programs to test the class PQS

We executed several experiments:
Each experiment started by creating a new PQ object

p := new PQ;

Next, a sequence of commands

insert(e_i)

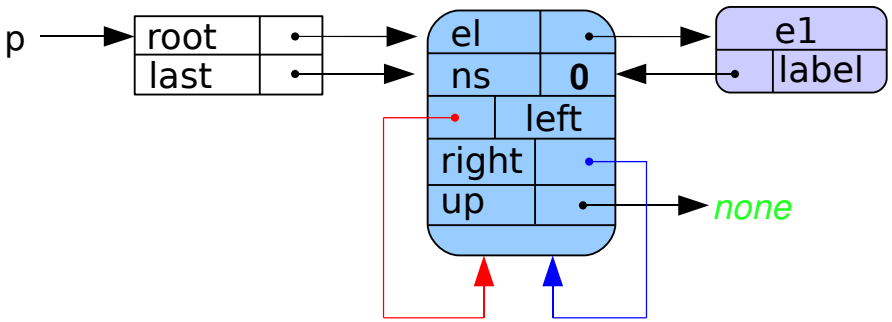
or

delete(e_j);

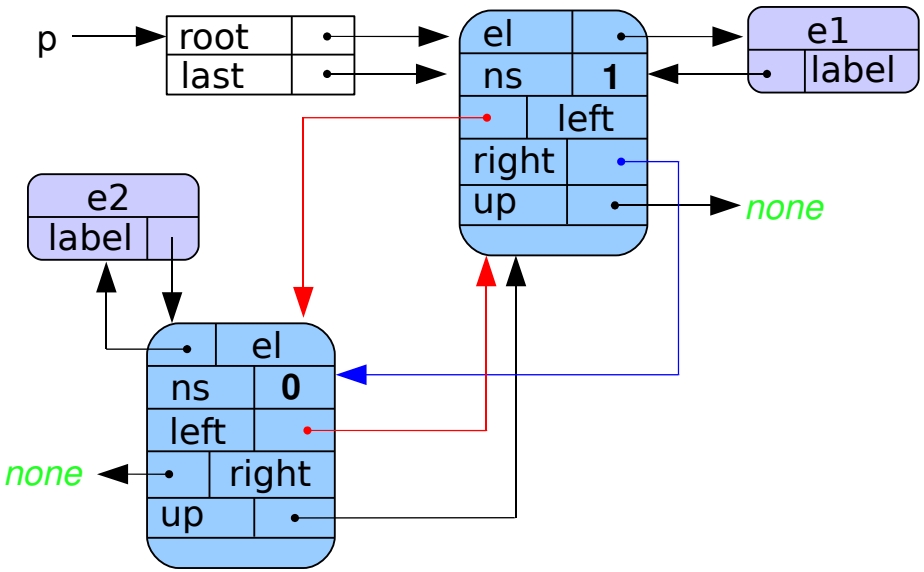
has been executed.
The trace of experiments has the form of a sequence of drawings.

p:= new PQ

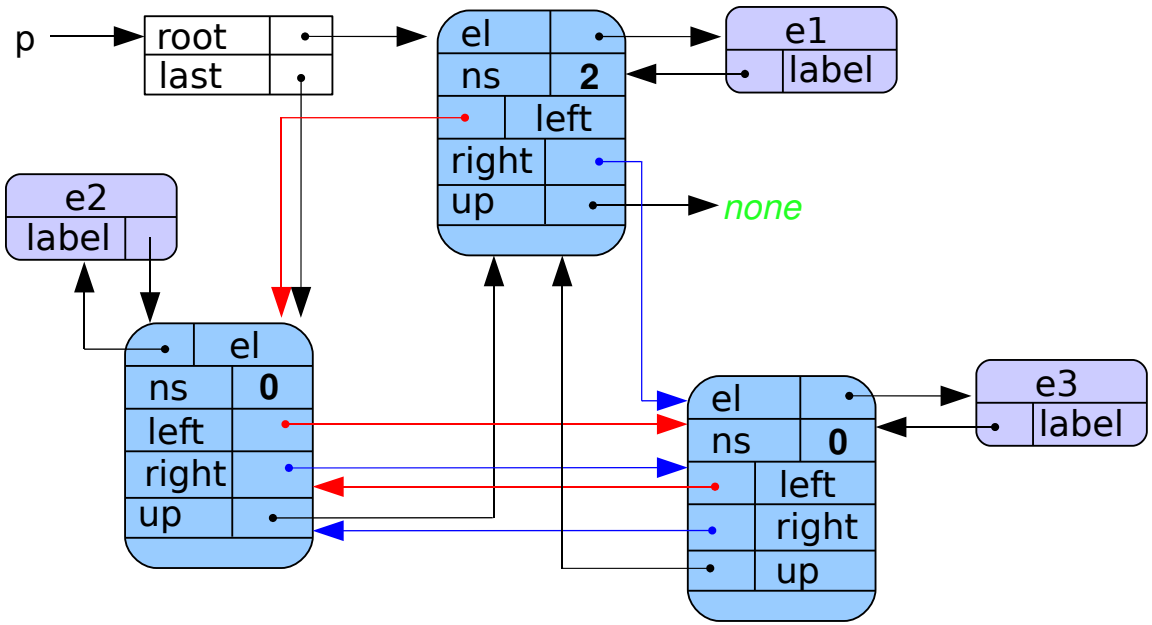
insert(e1)



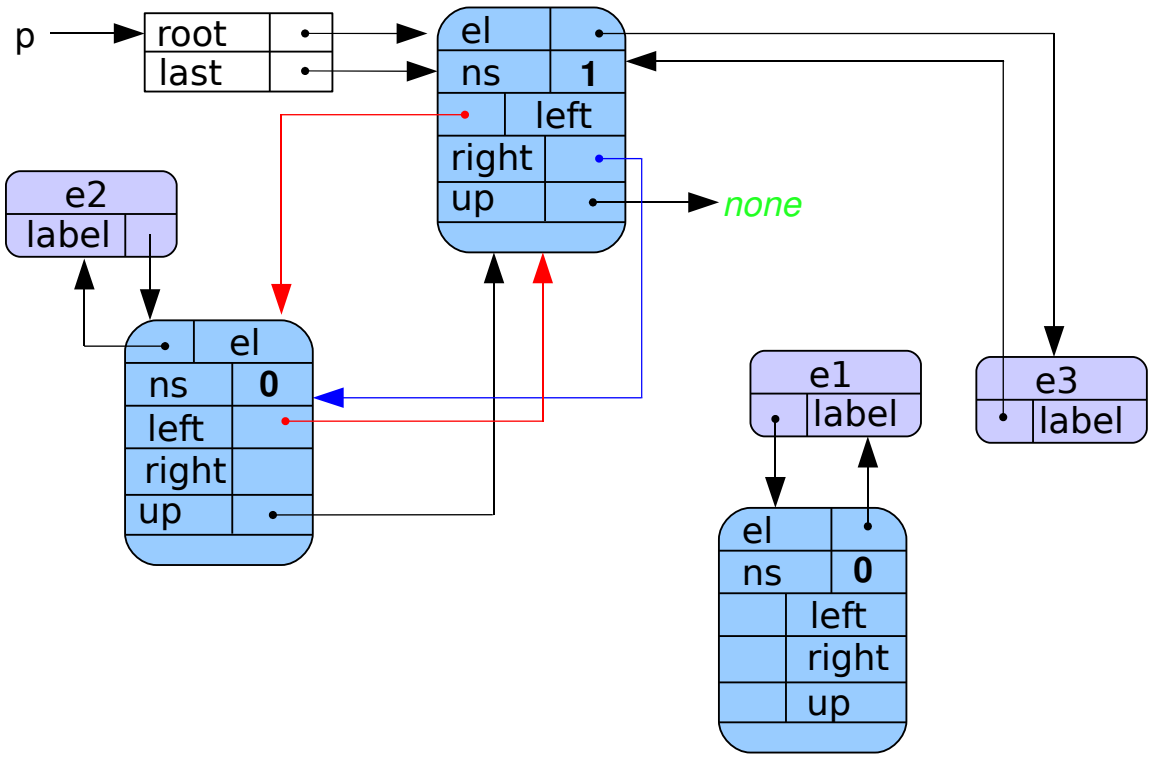
insert(e2)


p
root
last
el
ns
1
left
right
up
e1
label
none
e2
label
el
ns
0
left
right
up
none

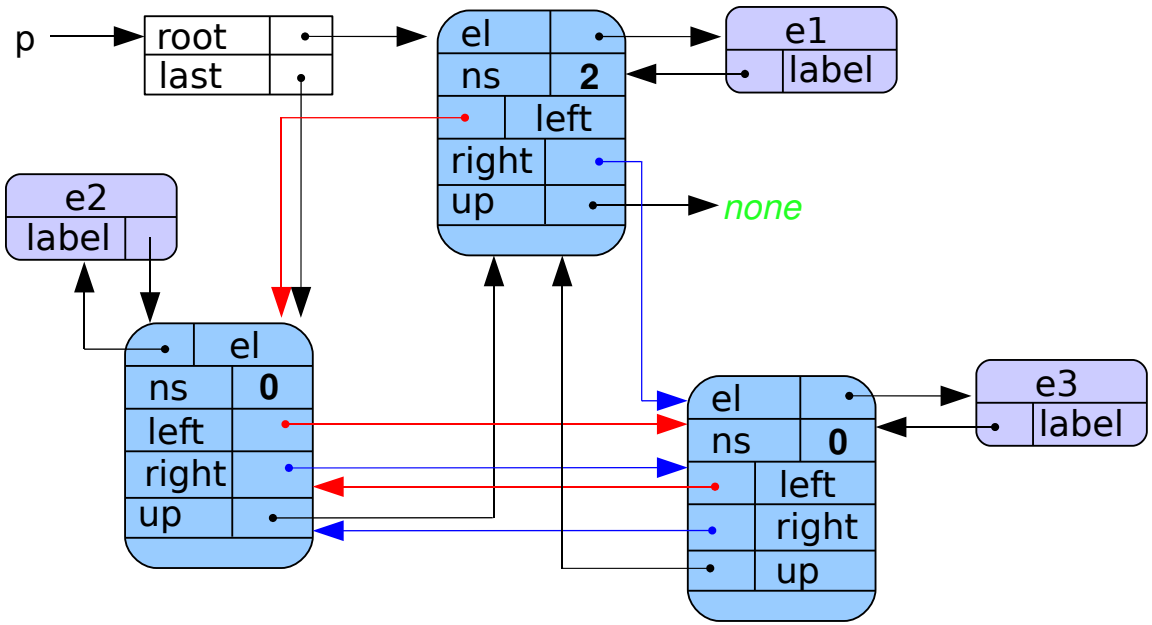
insert(e3)


p
root
last
el
ns
2
left
right
up
e1
label
none
e2
label
el
ns
0
left
right
up
el
ns
0
left
right
up
e3
label

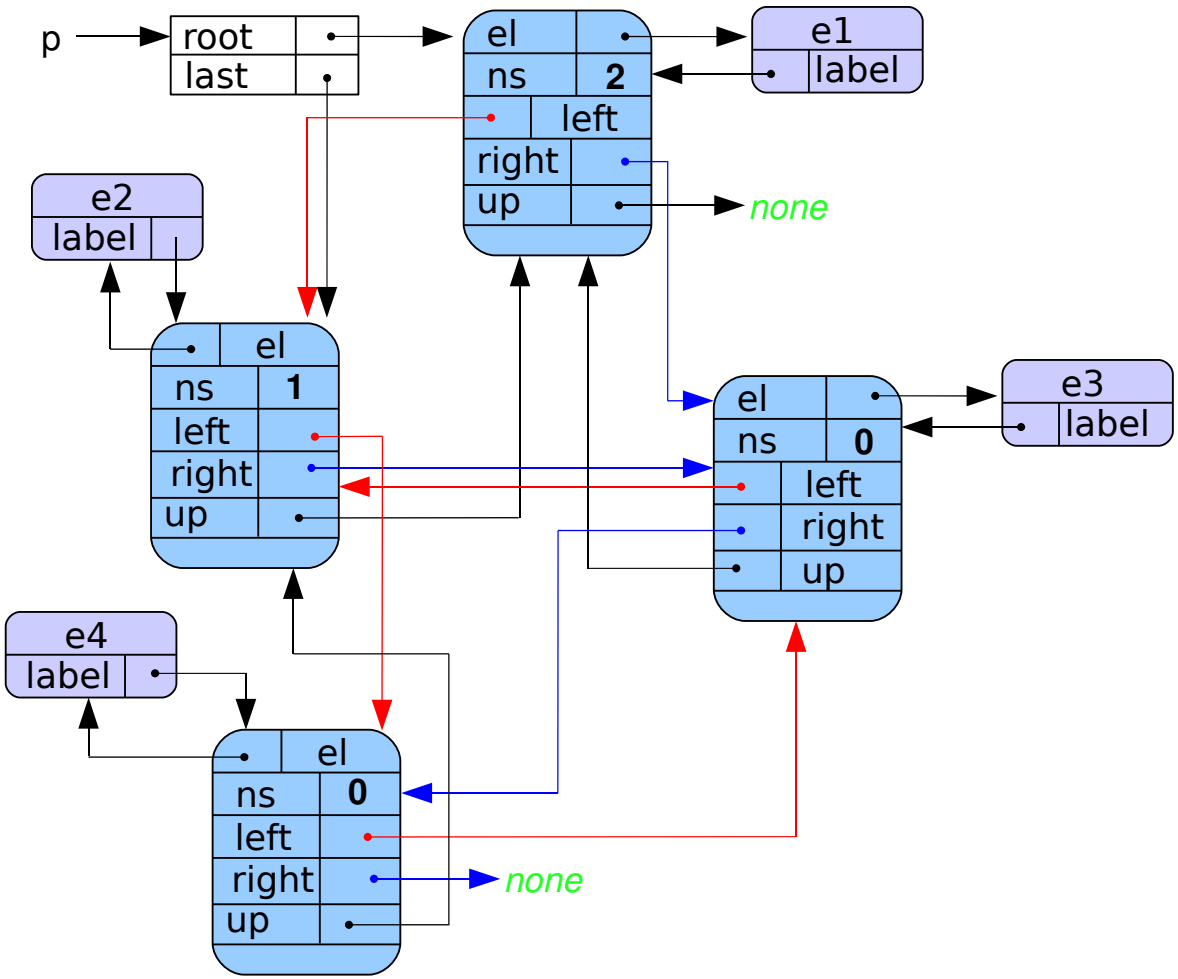
let's try: delete(e1)



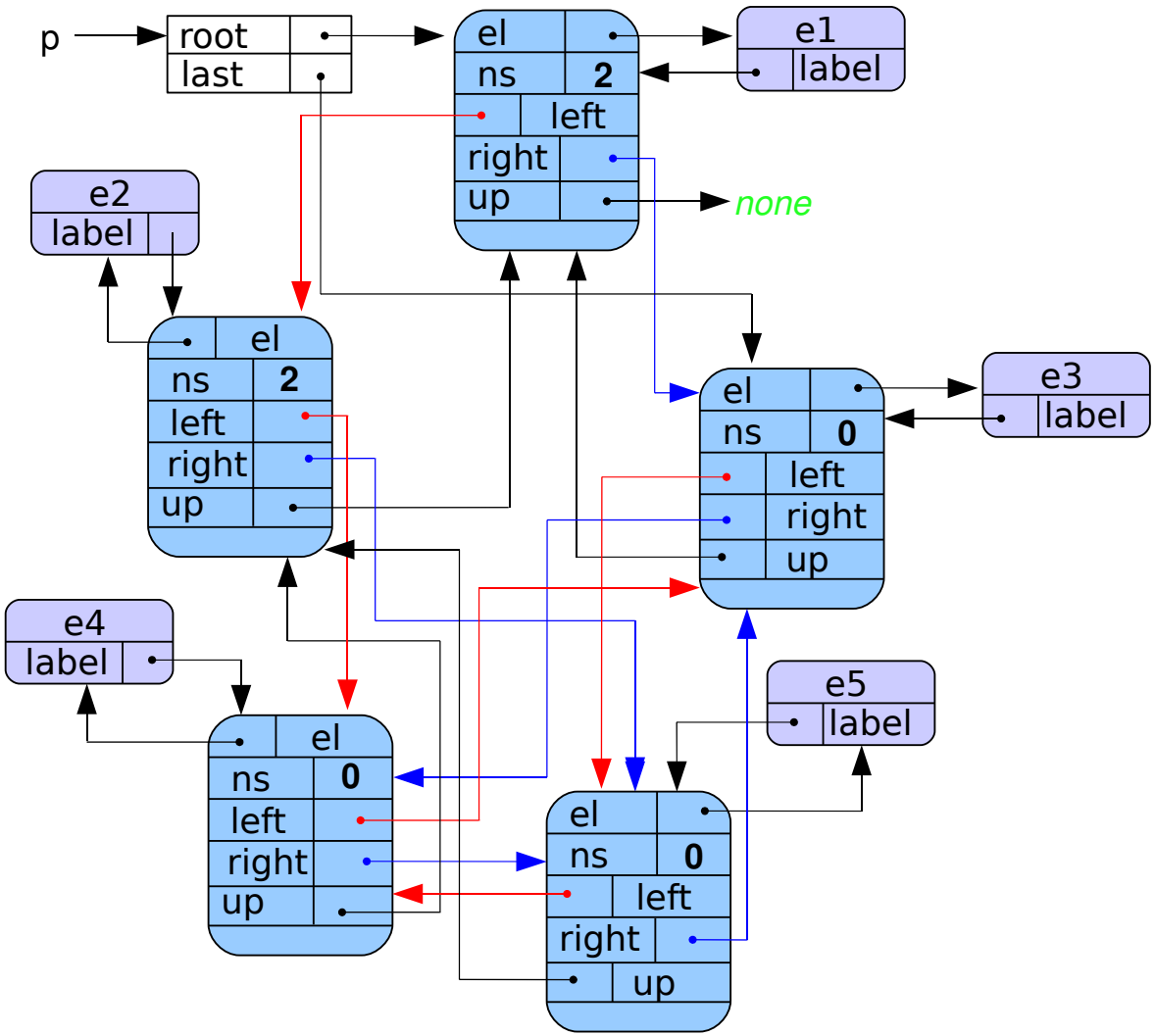
back to: insert(e3)



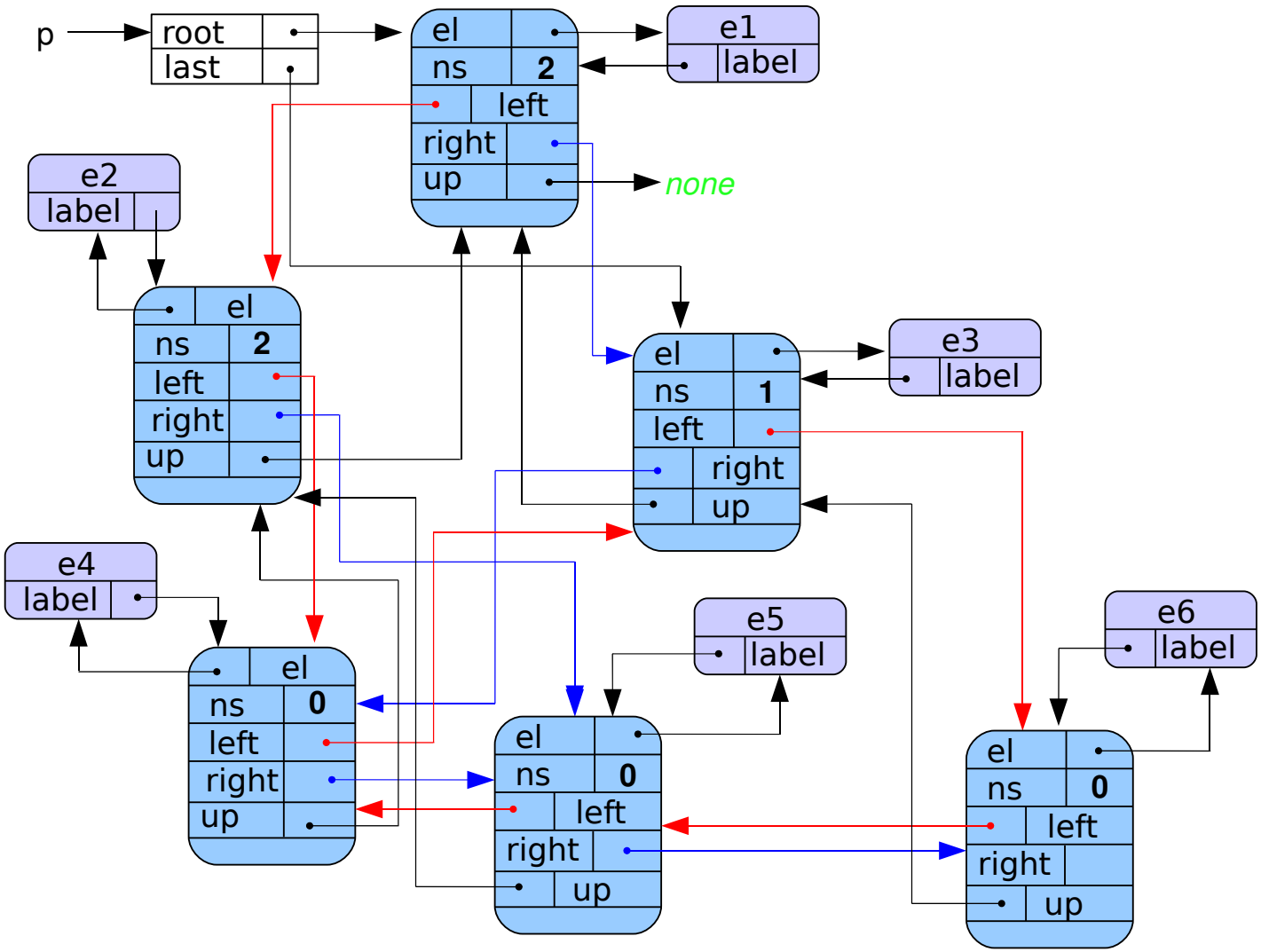
insert(e4)


p
root
last
el
ns
2
left
right
up
e1
label
none
e2
label
el
ns
1
left
right
up
el
ns
0
left
right
up
e3
label
e4
label
el
ns
0
left
right
up
none

insert(e5)


p
root
last
el
ns
2
left
right
up
e1
label
none
e2
label
el
ns
2
left
right
up
el
ns
0
left
right
up
e3
label
e4
label
el
ns
0
left
right
up
e5
label
el
ns
0
left
right
up

insert(e6)


p
root
last
el
ns
2
left
right
up
e1
label
none
e2
label
el
ns
2
left
right
up
el
ns
1
left
right
up
e3
label
e4
label
el
ns
0
left
right
up
e5
label
el
ns
0
left
right
up
e6
label
el
ns
0
left
right
up

Bieżący rozdział

observable states

Following the intuition gained from the experiments we shall introduce the notion of observable states. The initial state s_0 is the graph consisting of exactly one object o of type `PQ`, $s_0 = \{new\ \texttt{PQ}\}$ and no edges.

Definition 1

The set $\mathcal{S}$ of observable states is the least set which contains the initial state s_0 and which is closed with respect to the operations insert and delete and creation of `new Elem()` objects.

Each state consist of a set of objects and the edges connecting them. The examples of states are presented on figures shown earlier in the section Experimenting.

algebraic structure PQS I

The class `PQS` may be viewed as a definition of the relational structure PQS. The universe U of the structure consists of the objects of the inner classes of the class `PQS`. The attributes of objects of U define functions between objects. The set of objects of type `Node` will be denoted Node. Similarly for the set of objects of type `Elem` and of type `PQ`:

$$\begin{aligned} \text{Node} &= \{n : \ n \text{ instanceof } \texttt{Node}\} \\ \text{Elem} &= \{e : \ e \text{ instanceof } \texttt{Elem}\}. \\ \text{PQ} &= \{q : \ q \text{ instanceof } \texttt{PQ}\}. \end{aligned}$$

algebraic structure PQS II

Definition 2

The class PQS determines an algebraic structure

$$\mathrm{PQS} = \langle U, .up, .left, .right, .el, .lab. \rangle,$$

where U is a subset of the union $\mathrm{Node} \cup \mathrm{Elem} \cup \mathrm{PQ}$. The functions of the structure PQS are defined as follows

$$\begin{aligned}
.up &\stackrel{df}{=} \{\langle n, n' \rangle : \ n, n' \in \mathrm{Node}, \ n' = n.up\}, \\
.left &\stackrel{df}{=} \{\langle n, n' \rangle : \ n, n' \in \mathrm{Node}, \ n' = n.left\}, \\
.right &\stackrel{df}{=} \{\langle n, n' \rangle : \ n, n' \in \mathrm{Node}, \ n' = n.right\}, \\
.el &\stackrel{df}{=} \{\langle n, e \rangle : \ n \in \mathrm{Node}, e \in \mathrm{Elem}, \ e = n.el\}, \\
.lab &\stackrel{df}{=} \{\langle e, n \rangle : \ e \in \mathrm{Elem}, n \in \mathrm{Node}, \ n = e.lab\}
\end{aligned}$$

Observation

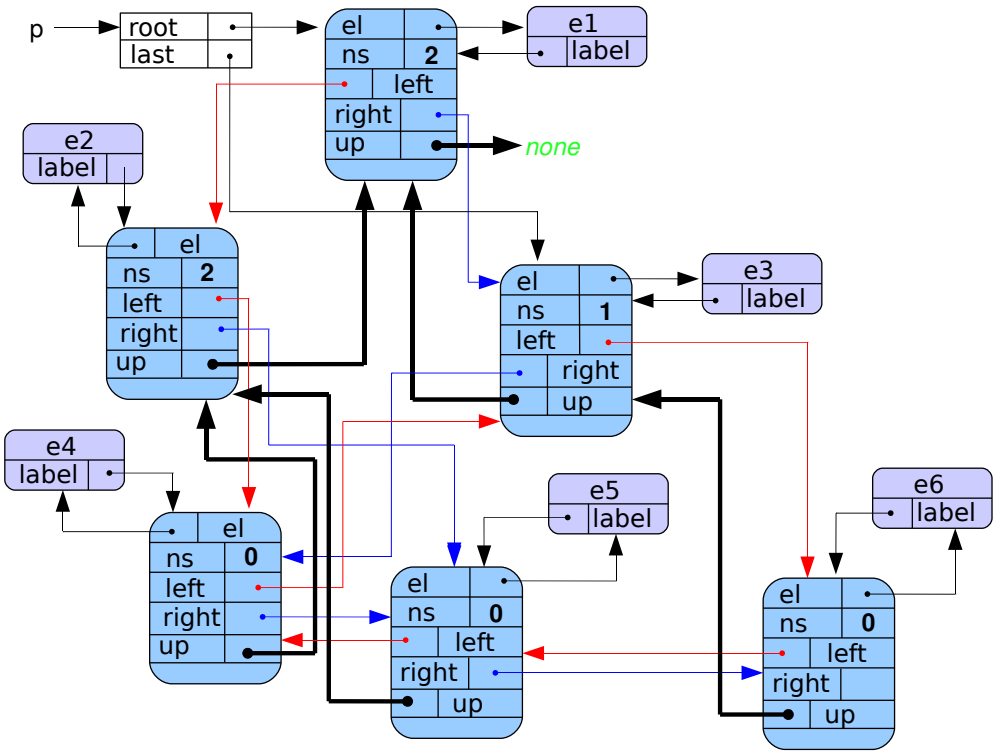
In any observable state the following condition hold

$$(\forall_{n \in \mathsf{Node}})\ (\forall_{e \in \mathsf{Elem}})\ n.el = e \Leftrightarrow e.lab = n.$$

This so because when a new Elem object is created, its companion object of type Node will be created too. See the examples of the previous section and the constructor of the class Elem. $\square$

nodes form a tree I

Our first observation is that if we abstract from (i.e. we forget about) the arrows *.left* and *.right* then for each state *s* its graph shows a tree.



nodes form a tree II

Definition 3

Let s be an observable state, consider the objects of type `Node` in this state. Let T_s be the set of these object of type `Node` that access the object *.root* by a path composed from *.up* arrows only.

$$T_s = \{o \in \text{Node} \cap s : \textit{there is a path composed from .up arrows only, leading from object } o \textit{ to object .root}\}$$

Lemma 4

In any observable state s the pair $\langle T_s, .up\rangle$ is a tree.

sons and leafs

Definition 5

We say that a node n is a son of a node f in the tree T_s if and only if $n.up = f$. A node n is said to be a leaf of the tree T_s if and only if it has no sons.

Our next observation is

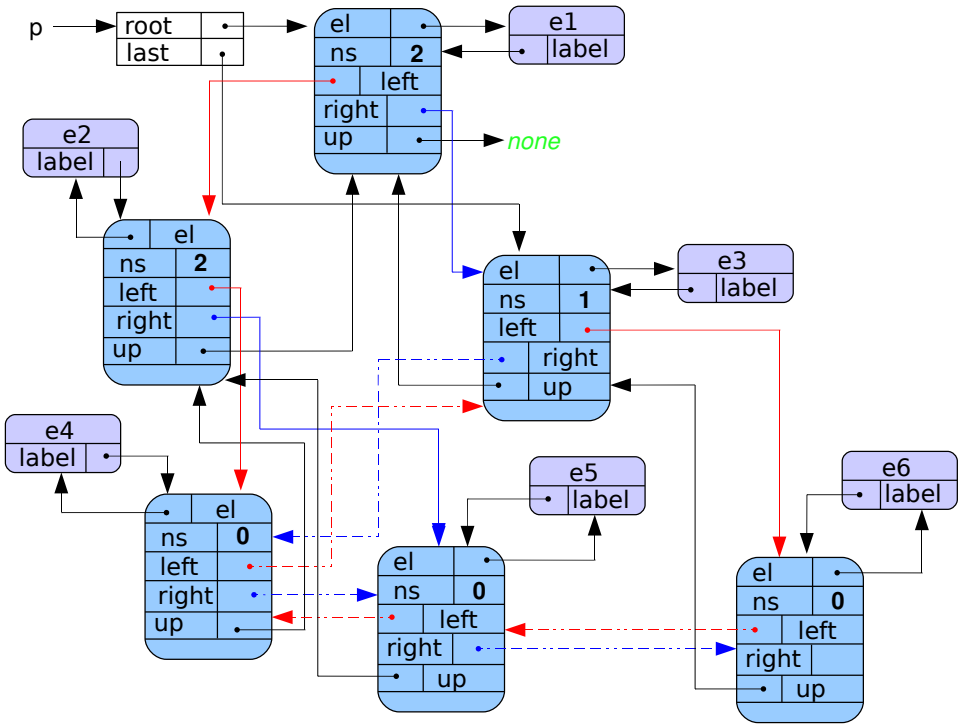
Observation 6

For every $o \in T_s$, $o.ns$ = number of sons of o. □

dotted & solid arrows I

Definition 7

We say that an arrow *.left* from the node n is solid iff $n.ns > 0$. We say an arrow *.right* from the node n is solid iff $n.ns = 2$. Otherwise the arrows are said weak, or dotted.



dotted & solid arrows II

Proposition 8

For every state s, the tree T_s with solid arrows only, forms a binary tree.

Proof: For every two nodes n and f of the tree T_s the following properties hold:

- if a solid .left arrow leads from node f to node n then $n.up = f$ $(f.left = n \wedge f.ns > 0\ \ ) \Rightarrow n.up = f$,
- if a solid .right arrow leads from node f to node n then $n.up = f$ $(f.right = n \wedge f.ns = 2\ \ ) \Rightarrow n.up = f$,
- $n.up = f \Leftrightarrow (f.left = n \vee f.right = n)$.

Hence T_s is a *binary* tree.

last

Our next observation can be stated as follows:

Proposition 9

There exists at most one node n in T_s such that $n.ns = 1$. If it is the case then $last = n$. □

tree is balanced

Now, we observe that the leaves of tree T_s are on two levels only.

Proposition 10

For every state s, there exists a natural number $k(T_s)$ such that every leaf of the tree T_s is on the level $k(T_s)$ or $k(T_s) - 1$.

The number $k(T_s)$ is equal the length of the path composed from *.up* arrows leading from the object *last.left* to the *root* object. It is equal 0 if *root* = *none*. □

the role of non-solid arrows

Proposition 11

The object referenced by the variable last in the tree T_s is the leftmost node on the level $k(T_s) - 1$ which has less than two sons, or it is the leftmost leaf on the level $k(T_S)$. □

Let us return to the pictures of experiments and observe the following facts:

Remark 12

A) If a node n has two sons then its left brother has also two sons.
B) If a node n has one son then it is its left son.
C) If a node n has one son then its brother from the left has two sons and its brother from the right is a leaf. □

Proposition 13

The value of the variable last is a head of a list of leaves linked together via .right (weak) arrows. □

cyclic list of leaves

Proposition 14

The value of the variable last is a head of a cyclic list of leaves linked by (weak) .left arrows. □

We see that all leaves on the level $k(T_s)$ are grouped to the left.

Proposition 15

Tree T_s is a perfect binary tree i.e. all the levels are completely filled with an eventual exception on the deepest level, in this case all the leaves are grouped to the left. □

correctness of insert

The following four lemmas have similar form $(\alpha \implies I\beta)$, where I is either instruction *insert*(e) or instruction *delete*(e), α is a precondition and β is a postcondition of the instruction I.

Lemma 16

Let I : *insert*(e) *and*
$\alpha_1 : \{last = o \wedge o.left = n1 \wedge o.right = k \wedge o.ns = 0 \wedge e.lab = n \wedge n.el = e\}$,
$\beta_1 : \{last = o \wedge o.ns = 1 \wedge o.left = n \wedge o.right = k \wedge n.up = o \wedge n.left = n1\}$.
The instruction I *is correct with respect to the precondition* α_1 *and postcondition* β_1 *in the structure* PQS

$$PQS \models (\alpha_1 \implies I\beta_1).$$

Proof I

The lemma states that in any state s, if the precondition α_1 is satisfied by s, then the execution of instruction *insert*(e) will succesfully lead to certain state s' and the postcondition β_1 will be satisfied by s'. We draw the meaning of the precondition α_1.

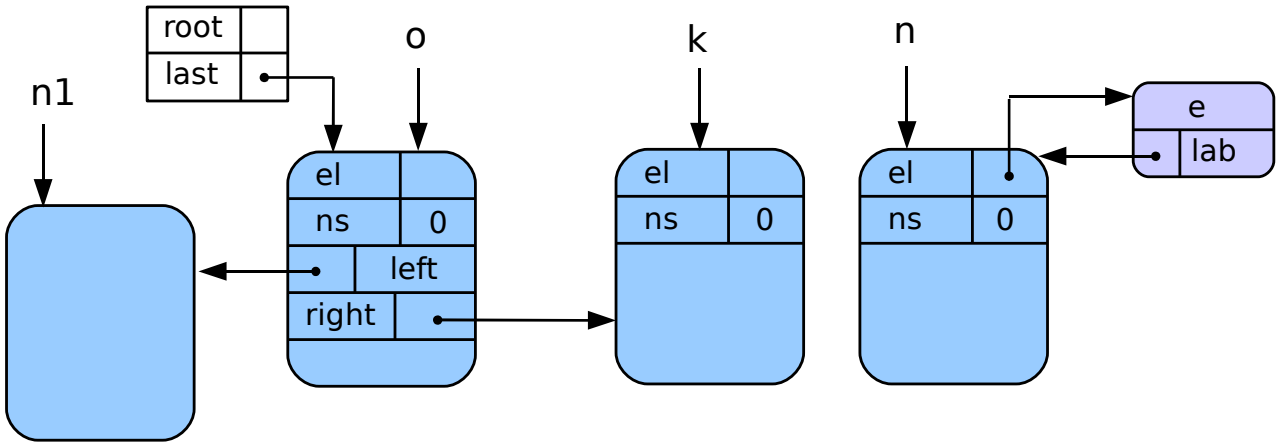


Fig. 2 Precondition α_1.

$\{last = o \wedge o.left = n1 \wedge o.right = k \wedge o.ns = 0 \wedge e.lab = n \wedge n.el = e\}$

Proof II

We check that the configuration of objects drawn on Fig. 2 satisfies the precondition α_1, The equality $last = o$ is satisfied since both variables point to the same object. The variable $last.left$ points to the object pointed by $n1$. $last.right$ point to the object pointed by k. The objects e and n are linked together, $e.lab = n$ and $n.el = e$.
Next, we can follow step by step the execution of the command $q.insert(e)$ with the text of method *insert* in hand. We start observing that the precondition α implies $last.ns = 0$ hence the instructions executed by *insert* are:
x:=e.lab; last.ns := 1; z := last.left; last.left := x; x.up := last;

x.left := z; z.right := x;last:=z;

Now we can draw modifications to the picture following the instructions.

Proof III

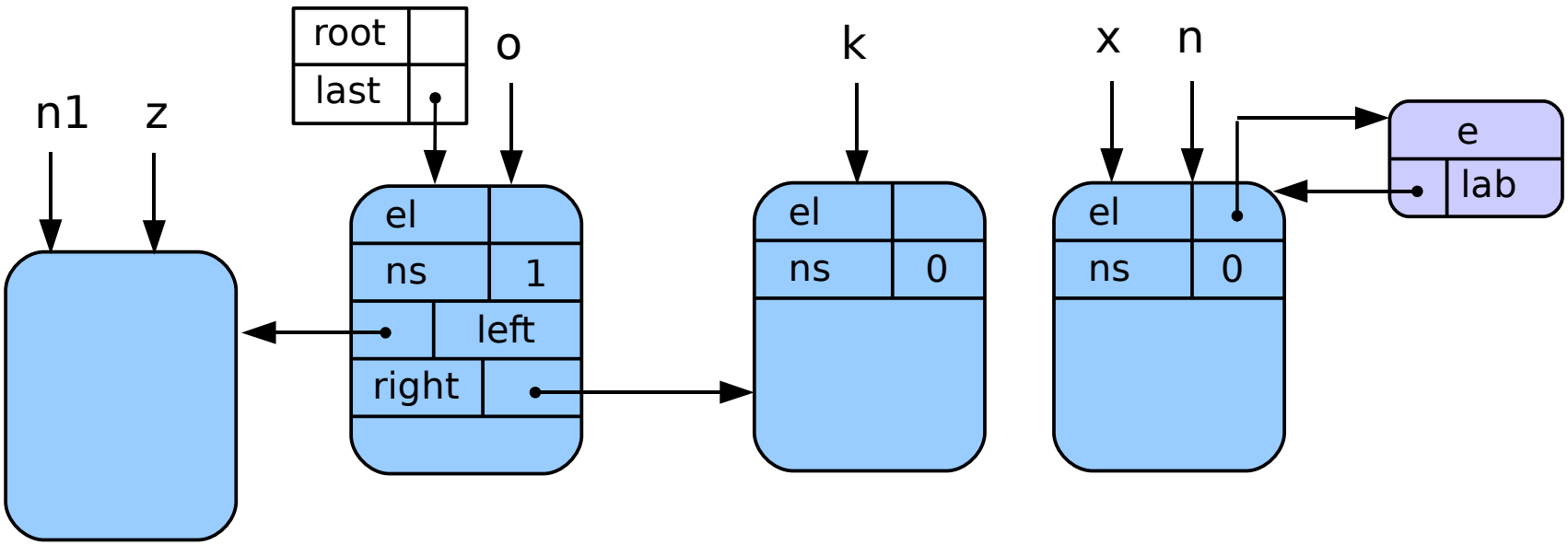


Fig. 3 Snapshot after 3 instructions of insert's body

Proof IV

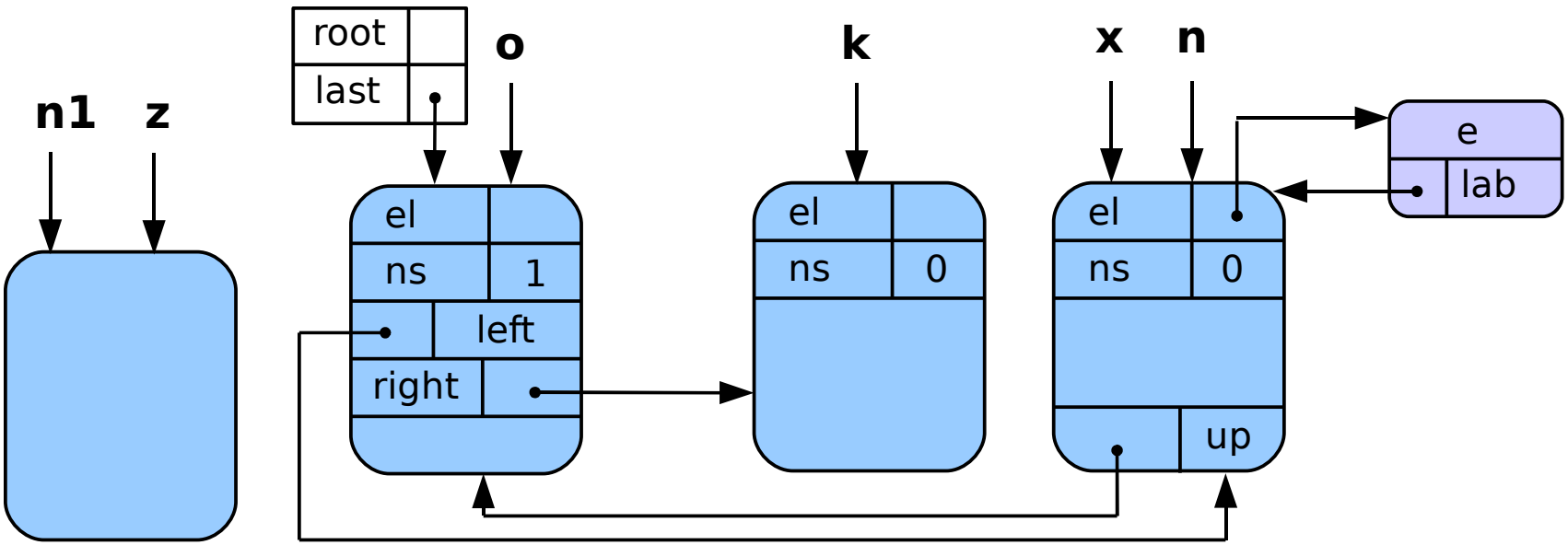


Fig. 4 Snapshot after 5 instructions of insert's body

Proof V

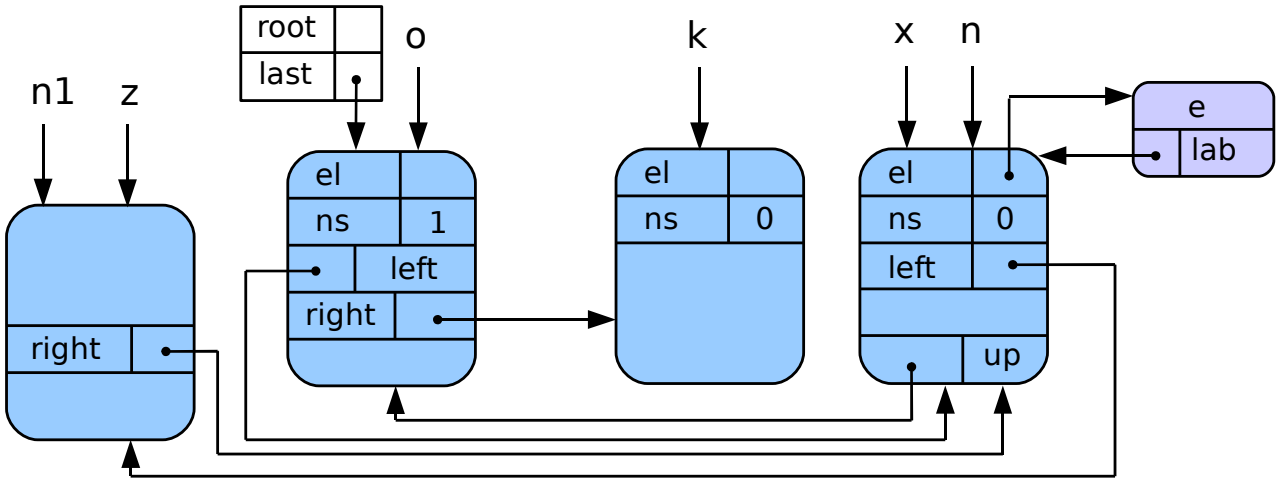


Fig. 5 Snapshot after instruction insert

The state shown on Figure 5 can be described by the following formula $\gamma : \{last = o \wedge o.ns = 1 \wedge o.left = n = n1.right = e.lab \wedge o.right = k \wedge n.up = o \wedge n.left = n1 \wedge n.el = e\}$. It is easy to observe that the formula γ implies the formula β_1. Note that if the initial state s satisfies the precondition α_1 then at the end of the execution of the instruction $insert(e)$ in PQS we obtain the state s' satisfying the postcondition β_1. □

correctness of insert - second lemma

Lemma 17

Let I : $insert(e)$ *and*
α_2:{$last{=}o \wedge o.left{=}n1 \wedge o.right{=}n2 \wedge o.ns{=}1 \wedge e.lab{=}n$},
β_2: {$last{=}n2 \wedge o.ns{=}2 \wedge o.left{=}n1 \wedge o.right{=}n \wedge n.up{=}o \wedge n.left{=}n1 \wedge n1.right{=}n$}.
In every state s *the following implication holds* $(\alpha_2 \implies I\beta_2)$.

proof of second lemma

If *last.ns* $= 1$ then the actions executed by *insert* are presented in Table 2 below.

Table 2. Proof of lemma 3.

Precondition: α_2:(last=o$\wedge$o.left=n1$\wedge$o.right=n2$\wedge$o.ns=1 $\wedge$e.lab = n)	
Instruction	Effect
x := e.lab	x=n, *since precondition says* e.lab=n
last.ns:=2	o.ns=2, *since* last=o
z := last.right	z=n2, *because* last.right=n2
last.right:=x	o.right=n, *because* last=o and x=n
x.right := z	n.right=n2, *because* x=n
x.up:= last	n.up=o, *because* last=o and x=n
z.left:=x	n2.left=n, *because* z=n2
last.left.right:=x	n1.right=n, *because* last.left=n1
x.left:=last.left	n.left =n1, *because* last.left=n1 and x=n
last :=z	last=n2, *because* z=n2
Postcondition: β':(o.ns=2$\wedge$o.right=n$\wedge$n.right=n2$\wedge$n.up=o$\wedge$ n2.left=n$\wedge$n1.right=n$\wedge$ n.left=n1$\wedge$last=n2$\wedge$o.left=n1$\wedge$e.lab=n)	

The postcondition β' collects the facts enlisted in the column Effect extended by the formulas e.lab=n and o.left=n1. The formula β' is stronger than β_2.

two observations

Proposition 18

For every two nodes x, y in the tree T_s, $x.up = y \Rightarrow y.less(x)$. □

This sequence of observations leads to the following:

Lemma 19

In each observable state s the tree T_s is a heap. □

main theorem

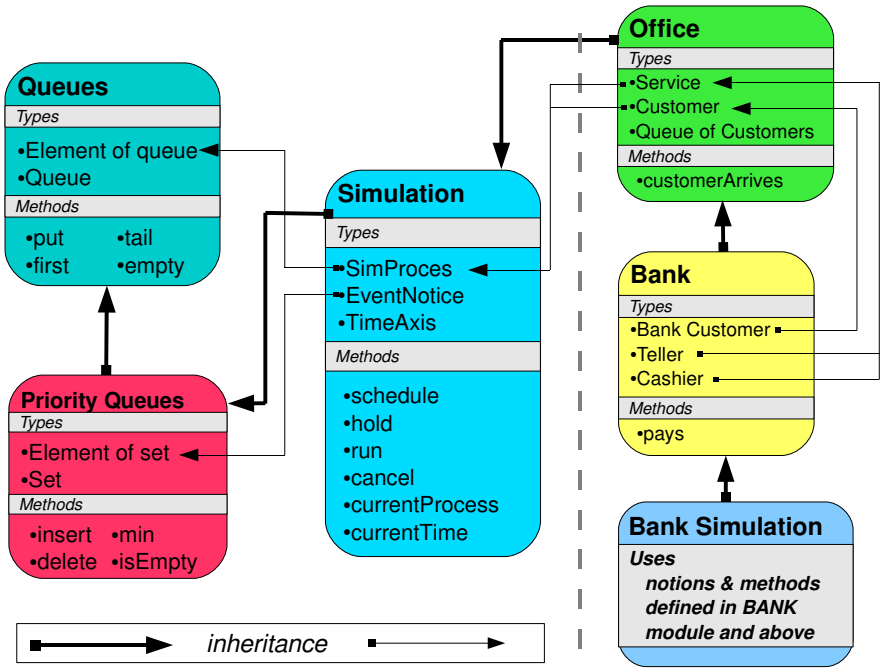
Now, we are ready to verify that all axioms of ATPQ theory hold in the structure defined by the class `PQS`.

Theorem 20

The structure $\mathcal{PQS}$ of elements and `PQ` *objects implemented by the class* `PQS` *is a priority queue.* □

Application

One immediate application of the class PQS is in the class Simulation.



A simulation needs many operations on the plan of simulation. A plan is the set of eventnotices ordered by the time of eventnotice. Operations *insert*, *delete* and *min* should be executed efficiently. And the theorem asserts that they are implemented in a correct way.

Bieżący rozdział

Final remarks

We gave you a sample where you could observe:

- specification

Final remarks

We gave you a sample where you could observe:

- specification
- experiments

Final remarks

We gave you a sample where you could observe:

- specification
- experiments
- analysis of experiments & formulation of hypotheses

Final remarks

We gave you a sample where you could observe:

- specification
- experiments
- analysis of experiments & formulation of hypotheses
- proving

One may remark:

- specification is much shorter than program

One may remark:

- specification is much shorter than program
- it is a criterion of correctness of an implementation

One may remark:

- specification is much shorter than program
- it is a criterion of correctness of an implementation
- it is helpful in programming an application (here class Simulation)

One may remark:

- specification is much shorter than program
- experiments need not limit to testing, they are needed to enable:

One may remark:

- specification is much shorter than program

- experiments need not limit to testing, they are needed to enable:
- analyzing experiments & formulating hypotheses

One may remark:

- specification is much shorter than program

- experiments need not limit to testing, they are needed to enable:
- analyzing experiments & formulating hypotheses
- proving is not so difficult (once we know what to prove)

Thank you!

Appendix - the class PQS I

The full text of the class PQS as written by W.M. Bartol and D. Szczepańska. It is a part of bigger program of simulation of bank department.

```
unit PQS : class; (* priority queues system *)
    unit PQ: class;
        var last,root:node;
        unit min: function: elem;
        begin
            if root=/= none then result:=root.el else throw new Undefined() f
        end min;
        unit insert: procedure(r:elem);
            var x,z:node;
        begin
            x:= r.lab;
            if last=none then
```

Appendix - the class PQS II

```
            root:=x; root.left, last:=root
        else
            if last.ns=0 then
                last.ns:=1; z:=last.left; last.left:=x;
                x.up:=last; x.left:=z; z.right:=x;
            else
                last.ns:=2; z:=last.right; last.right:=x;
                x.right:=z; x.up:=last; z.left:=x;
                last.left.right:=x; x.left:=last.left; last:=z;
            fi
        fi;
        call correct(r,false)
    end insert;
    unit delete: procedure(r: elem);
        var x,y,z:node;
    begin
```

Appendix - the class PQS III

```
x:=r.lab; z:=last.left;
if last.ns =0 then
    y:= z.up;
    if y=none then root:=none else y.right:= last fi;
    last.left:=y; last:=y;
else
    y:= z.left; y.right:= last; last.left:= y;
fi;
z.el.lab:=x; x.el:= z.el; last.ns:= last.ns-1;
r.lab:=z; z.el:=r;
(* the following three instructions were added after experimenting
z.left.right :=none; z.ns:=0; z.left, z.right, z.up := none;
if x.less(x.up) then
    call correct(x.el,false)
else
    call correct(x.el,true)
```

Appendix - the class PQS IV

```
        fi;
    end delete;
    unit correct: procedure(r:elem, down:boolean);
        for the body of correct consult [?].
    end correct;
end PQ;
unit node: class (el:elem);
    var left,right,up: node, ns:integer;
    unit less: function(x:node): boolean;
    begin
        if x= none then
            result:=false
        else
            result:=el.less(x.el)
        fi;
    end less;
```

Appendix - the class PQS V

```
    end node;
    unit elem: class(prior:real);
        var lab: node;
        unit virtual less: function(x:elem):boolean;
        begin
            if x=none then
                result:= false
            else
                result:= prior ≤ x.prior
            fi;
        end less;
    begin
        lab:= new node(this elem);
    end elem;
    class Undefined extends Exception { }
end PQS (* priority queues system *);
```