MiniCP

A Minimalistic Educational Solver

Laurent Michel, Pierre Schaus, Pascal Van Hentenryck

@ldmbouge @pschaus @PVanHentenryck

website: https://www.info.ucl.ac.be/~pschaus/minicp

code: https://bitbucket.org/pschaus/minicp

slides http://tinyurl.com/y8n4knhx

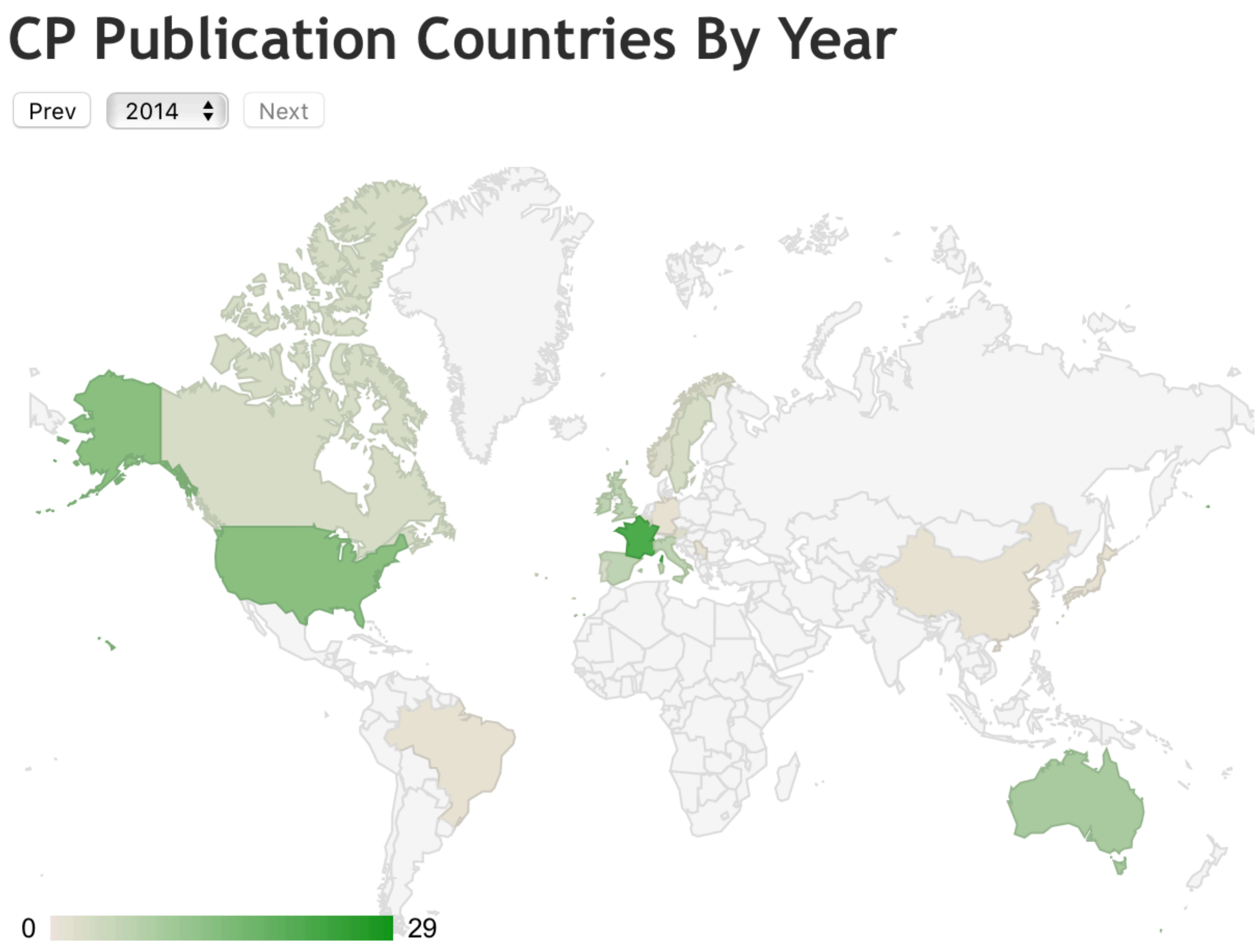
About the logo

Hummingbirds are **small, beautiful, efficient**

- the smallest birds
- rapid wing-flapping rates
    - typically around 50 times per second,
    - allowing them also to fly at speeds 54 km/h
- plumage with bright, varied coloration

The observation

- Many students in CS graduate without having ever heard about CP



http://www.a4cp.org/cparchive/countries_by_year

and outside of the community?

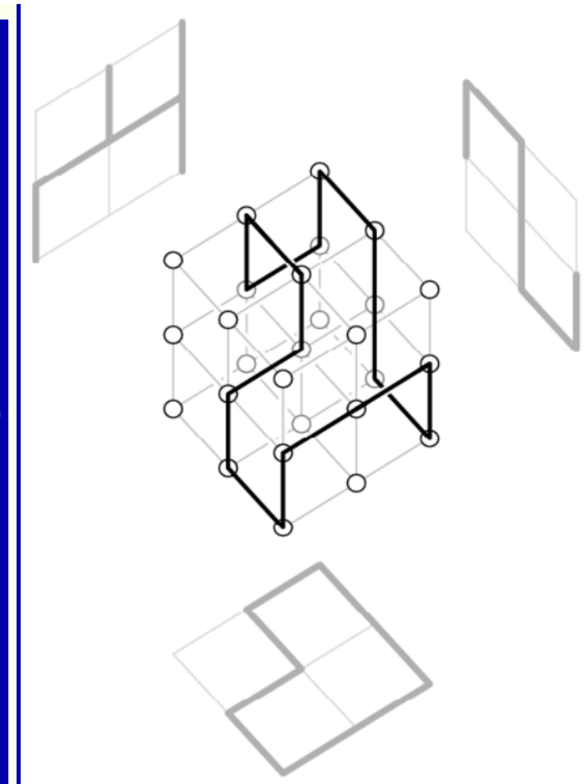
- Donald Knuth latest volume:

Donald E. Knuth

The Art of Computer Programming

Volume 4,
Combinatorial Algorithms

Links to .pdf files are uncorrected;
published versions are up-to-date.
Corresponding .ps files are on archive.org,
with links below in orange.
My balance at: The Bank of San Serriffe,
Financial Fiasco.
Somber essay: Infreq. Asked Quest.,
Letter to Rice, Cartoon.
Interviews: 2014-05-20, 2008-04-25, 2005-09-04,
2001-10-05, 2000-01-25, 1993-12-07



CP is not even mentioned

Volume 4B, Combinatorial Algorithms: Part 2				
Mathematical Preliminaries Redux	5A	5A	54	(2015-10-03)
7.2.2. Backtrack Programming	5B	5B	58	(2016-11-12)
7.2.2.1. Dancing Links	5C	5C	58	(2017-04-15)
7.2.2.2. Satisfiability	6A	6A	318	Vol 4, Fasc 6 (2015-12-18, 320)

http://www.cs.utsa.edu/~wagner/knuth/

CP2015 Workshop on Teaching (Cork)

- As a community, what can we do to improve and increase the teaching of constraint programming?

- Unanimous answer/observations
  - communicate better and make teaching material more broadly available
  - most CP teachers build their own teaching material without necessarily sharing it
- The ACP decided to promote the sharing of teaching material such that any university or professor who wants to propose a CP course can do it with a modest effort.

MiniCP aims to fill-in this gap

- Our hope:
  - With MiniCP any professor having a basic background in algorithmic can easily propose a CP course at his institution.

- MiniCP (will) provides teaching material, exercises, unit tests, and development projects.

Target audience

- CS students with
  - background in data-structures and algos.

- Students/Instructors interested into teaching CP modeling language should consider
  - MOOC on Minizinc by Peter Stuckey
  - Tutorial on XCSP3 format
  - User-manual of OPL, AIMMS, etc

Why not use an existing Solver?

- Existing solvers often try to balance three conflicting objectives
  - Efficiency
  - Flexibility
  - Simplicity

solvers participating to competitions

solvers focussed on real-life appli (hybridization, etc)

most important criteria in the design of MiniCP

Design of MiniCP

- Influenced by cc(fd), Comet, Objective-CP and OscaR
- Similar design in other solvers (OR-Tools, Choco, etc)
- Implemented in Java8
- MiniCP is
  - trailed-based
  - propagator centered
  - adopt the mantra

$$\boxed{\text{CP = Modeling + Search}}$$

MiniCP is small

- Code-base of +- 1500 lines of Java code

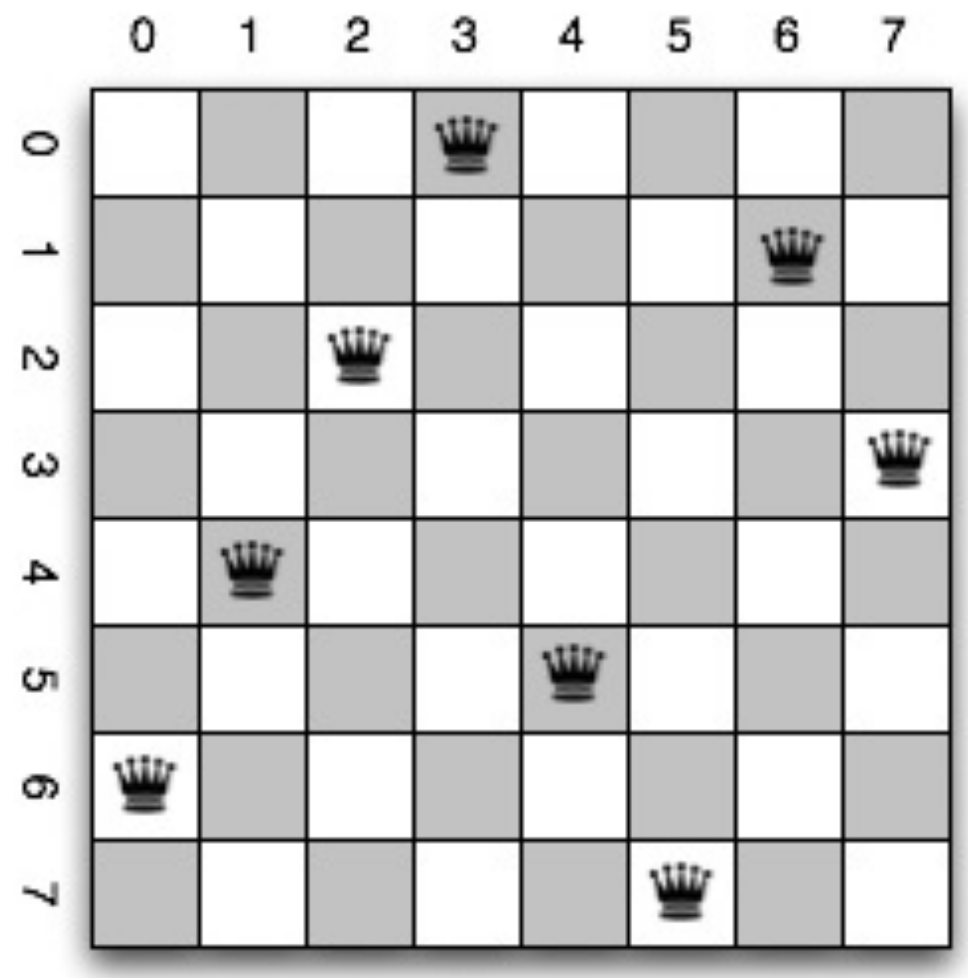
package	LOC
engine	**867**
reversible	**301**
cp	**154**
search	**148**
examples	288
tests	1316

Hello World = n-queens

- No two queens on the same line of diagonal

$$\forall\, i,j \in 0..n-1 \wedge i<j : q_i \neq q_j$$
$$\forall\, i,j \in 0..n-1 \wedge i<j : q_i \neq q_j + i - j$$
$$\forall\, i,j \in 0..n-1 \wedge i<j : q_i \neq q_j + j - i$$

q[0]=3
q[1]=6
q[2]=2
...



n-Queens Model

```
int n = 8;
Solver cp = makeSolver();
IntVar[] q = makeIntVarArray(
for(int i=0;i < n;i++)
    for(int j=i+1;j < n;j++) {
        cp.post(notEqual(q[i],q[j]));
        cp.post(notEqual(q[i],q[j],j-i));
        cp.post(notEqual(q[i],q[j],i-j));
    }
SearchStatistics stats = makeDfs(cp,
        selectMin(q,
                qi -> qi.getSize() > 1,
                qi -> qi.getSize(),
                qi -> {
                    int v = qi.getMin();
                    return branch(() -> equal(qi,v),
                                  () -> notEqual(qi,v));
                }
        )
).onSolution(() ->
        System.out.println("solution:"+ Arrays.toString(q))
).start();
```

create the solver

create the variables with domains 0..n-1

post the constraints

create DFS search

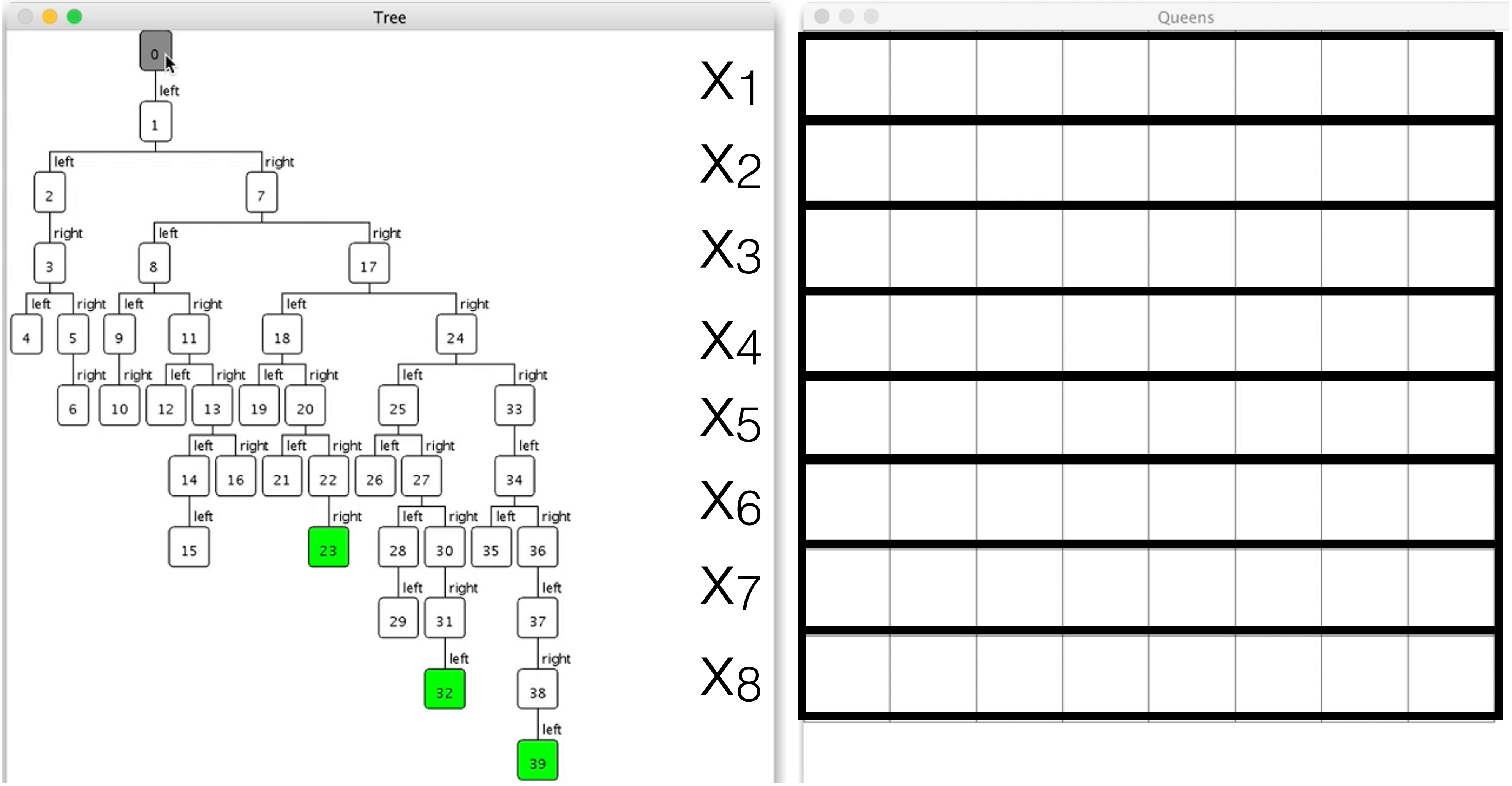
first fail heuristic

call back on solutions

effectively start the search

removed

assigned



Depth-First Search exploration letting the constraints prune the search tree

Some formalism 1/2

- A **domain** is a finite set of discrete values $D \subseteq Z$
- A **decision variable** $x \in X$ has a domain D, denoted $D(x)$
  - is instantiated (bound) when $|D(x)| = 1$,
  - inconsistent when $D(x) = \emptyset$ and free when $|D(x)| \geq 2$.
- A **constraint** $c \in C$ is a relation defined over a subset of k variables $\{x_1, \cdots, x_k\} = vars(c) \subseteq X$.
- Given a set of decision variables X, a **solution σ** is a domain D, such that $\forall x \in X : |\sigma(x)| = 1$

Some formalism 2/2

- Given decision variables X, and a constraint set C, a **feasible solution** σ is a domain D, such that

$$(\forall x \in X : |\sigma(x)| = 1) \wedge \bigwedge_{c \in C} c(\sigma)$$

- Given a CSP ⟨X,D,C⟩, the **solution set S(⟨X,D,C⟩)** is the set of all feasible solutions to ⟨X,D,C⟩.

- A **filtering algorithm F** for a constraint c ∈ C
  - removes inconsistent values from the domain (contracting)
  - consistent if it does not remove feasible solutions

$$\mathcal{S}(\langle X, D, C\rangle) = \mathcal{S}(\langle X, \mathcal{F}_c(D), C\rangle)$$

  - motonotic if $D_1 \subseteq D_2 \Rightarrow \mathcal{F}_c(D_1) \subseteq \mathcal{F}_c(D_2)$

Example of filtering rules

Whenever D(y) looses some value v from its domain, v +1 is removed from D(x)

- **x = y + 1**

$$
\begin{aligned}
v \notin D(y) &\Rightarrow v+1 \notin D(x) \\
v \notin D(x) &\Rightarrow v-1 \notin D(y) \\
|D(y)| = 1 &\Rightarrow D(x) = \{\min(D(y))+1\} \\
|D(x)| = 1 &\Rightarrow D(y) = \{\min(D(x))-1\}
\end{aligned}
$$

Fix-point computation = inference in each node

- Is the domain D solution to the fix-point equation

$$D = \bigcap_{c \in C} \mathcal{F}_c(D)$$

- In practice it is computed as an iterative procedure

Algorithm 1: Fixpoint algorithm

Data: D, C

Result: D the solution to the fixpoint equation (2)

$fix \leftarrow false$;
while $\neg fix$ **do**
    $fix \leftarrow true$;
    foreach $c \in C$ **do**
        $D' \leftarrow \mathcal{F}_c(D)$;
        if $D' \neq D$ **then**
            $D \leftarrow D'$;
            $fix \leftarrow false$;

Fix-point outcome

- Computation of the fix-point with constraints *C* on a domain D_0

$$D_1 = \mathcal{F}_C(D_0)$$

- Possible outcomes

1. failure(D_1) => no solution
2. success(D_1) => D_1 can be reported as a solution
3. not success(D_1) and not failure(D_1) => D_1 may contain a solution further splitting is necessary (divide and conquer)

Generic Search

Algorithm 2: Generic Search in MiniCP

Data: X, D, C

Result: $\mathcal{S}\langle X, D, C\rangle$

$S \leftarrow \emptyset$;

if $\mathtt{success}(D)$ **then**

  return $\{D\}$;

$Q \leftarrow \{\langle X, D, C\rangle\}$;

while $Q \neq \emptyset$ **do**

  $\langle X_0, D_0, C_0\rangle \leftarrow \mathtt{deQueue}(Q)$;

  $(c_1, \cdots, c_k) \leftarrow \mathtt{branching}(X_0, D_0)$;

  foreach $i \in 1..k$ **do**

    $D_i \leftarrow \mathcal{F}_{C \wedge c_i}(D_0)$;

    if $\mathtt{success}(D_i)$ **then**

      $S \leftarrow S \cup \{D_i\}$

    else if $\mathtt{failure}(D_i)$ **then**

      continue;

    else

      $\mathtt{enQueue}(Q, \langle X_0, D_i, C_0 \wedge c_i\rangle)$;

return S ;

Splitting of the search space
Example: x=2, x!=2

compute the fix-point
with c_i

CP mainly uses DFS so Q is
generally a stack

MiniCP

A Minimalistic Educational Solver

Laurent Michel, Pierre Schaus, Pascal Van Hentenryck

@ldmbouge **@pschaus** **@PVanHentenryck**

website: https://www.info.ucl.ac.be/~pschaus/minicp

code: https://bitbucket.org/pschaus/minicp

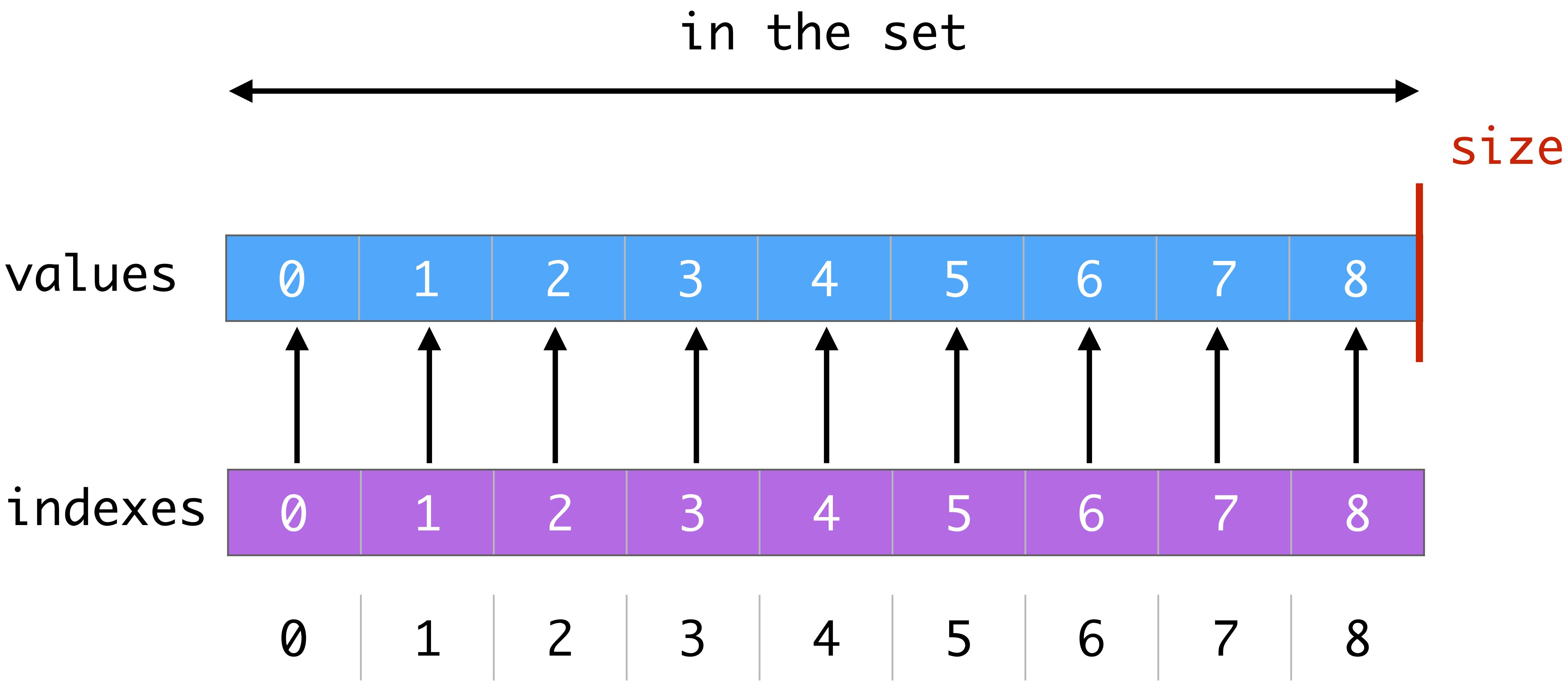
slides http://tinyurl.com/y8n4knhx

Domain Implementation

- Sparse-Set = data-structure for set implementation
  - O(1) value removal
  - O(1) remove all except one given value
  - O(1) testing if a value is present
  - Iteration in O(k), k = number of values in the set
- Sparse-Sets are convenient for domain implementation
  - easy to implement and explain

Sparset-Set

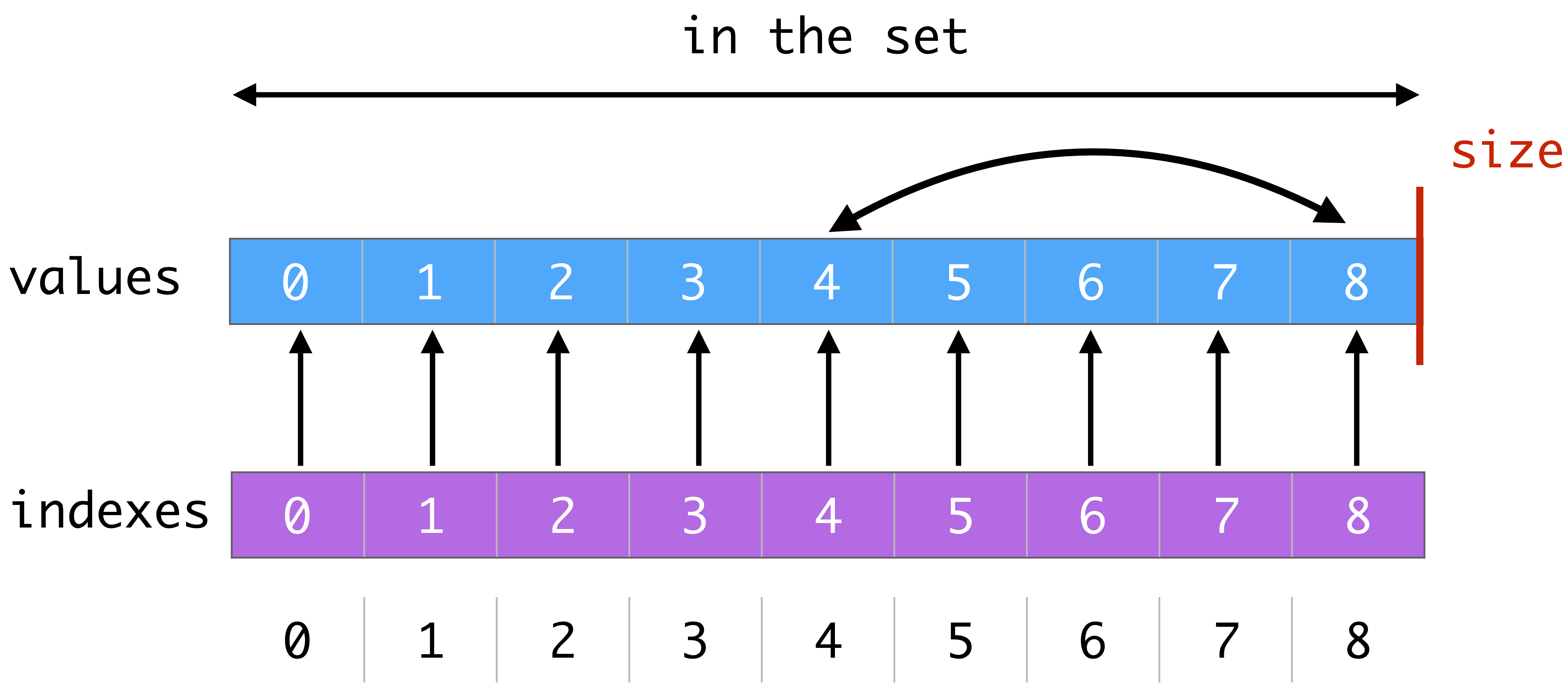
- Initialization for {0,1,2,3,4,5,6,7,8}



$$values[indexes[v]] = v, \forall v \in \{0..n-1\}$$

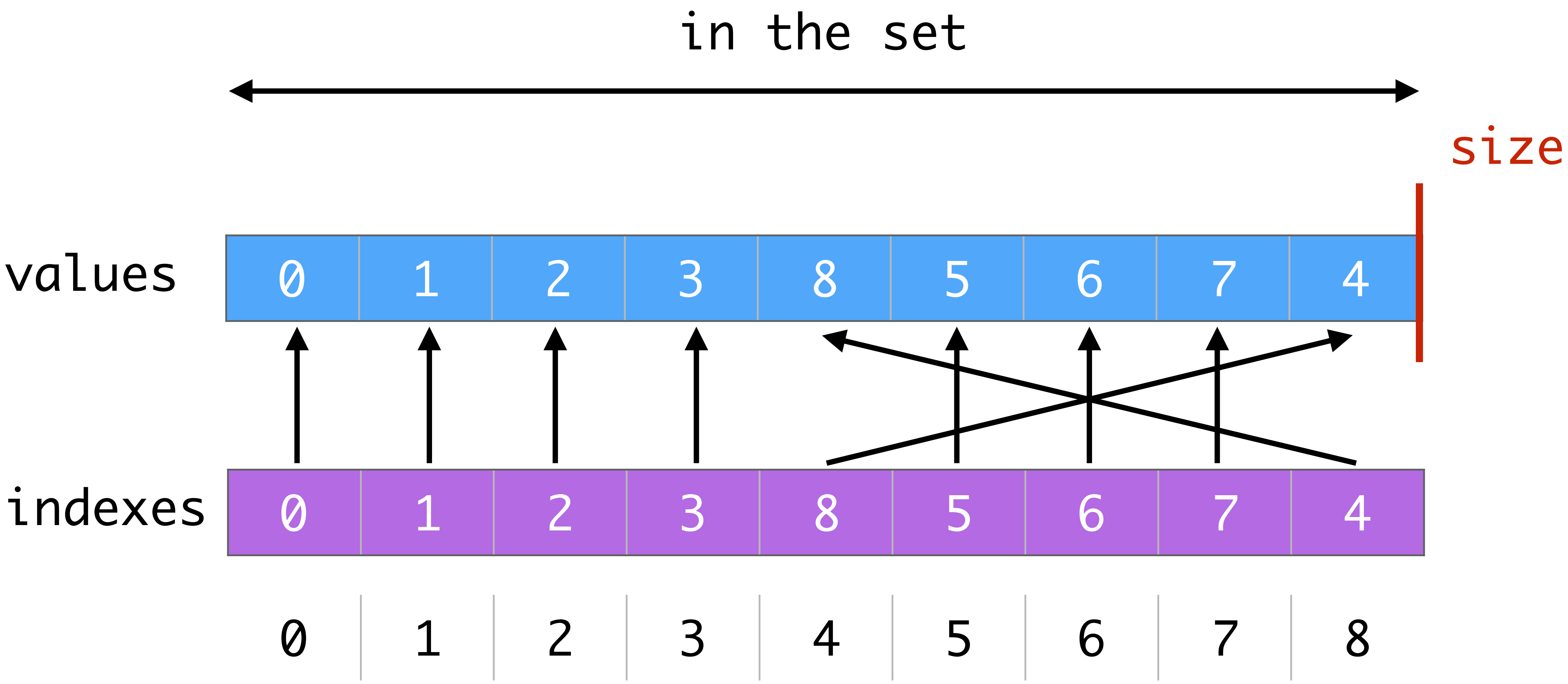
Removal operation

- Remove 4



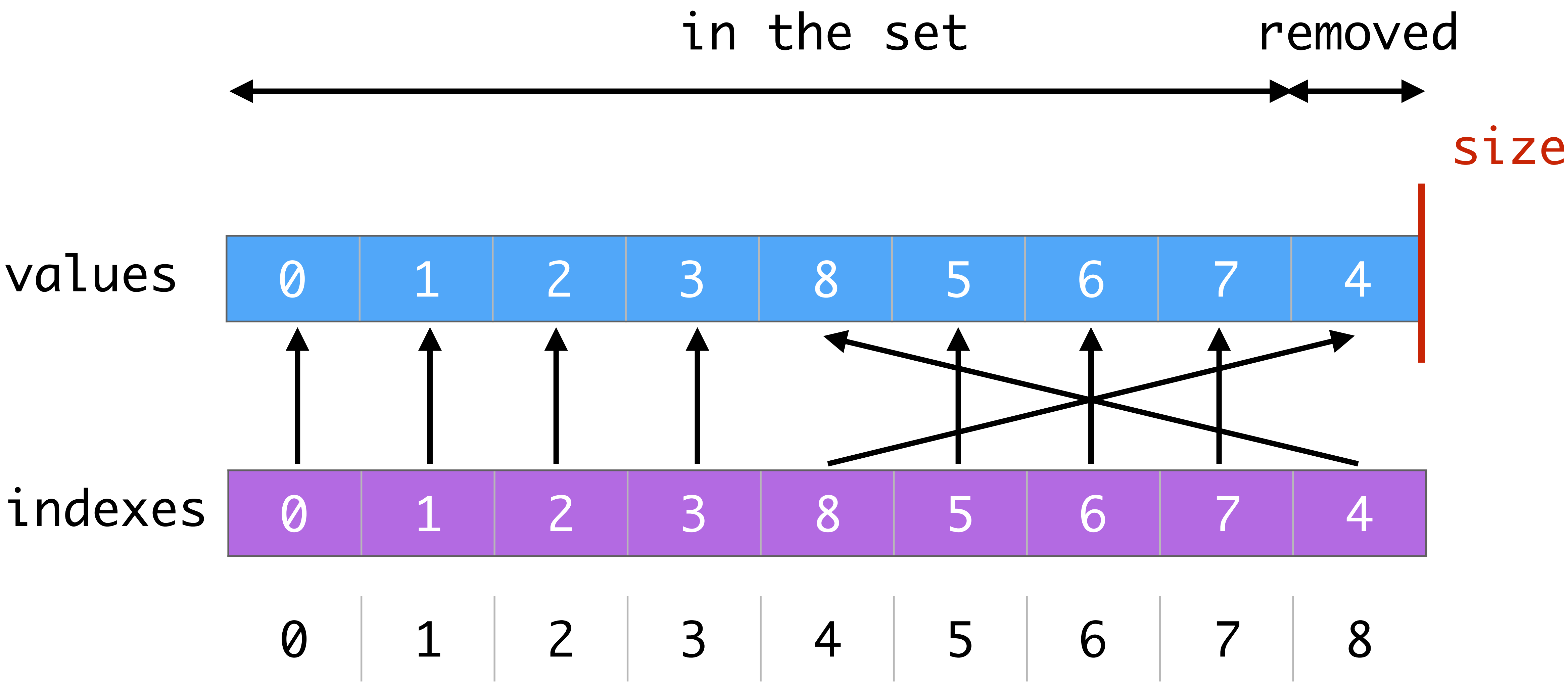
Removal operation

- Remove 4



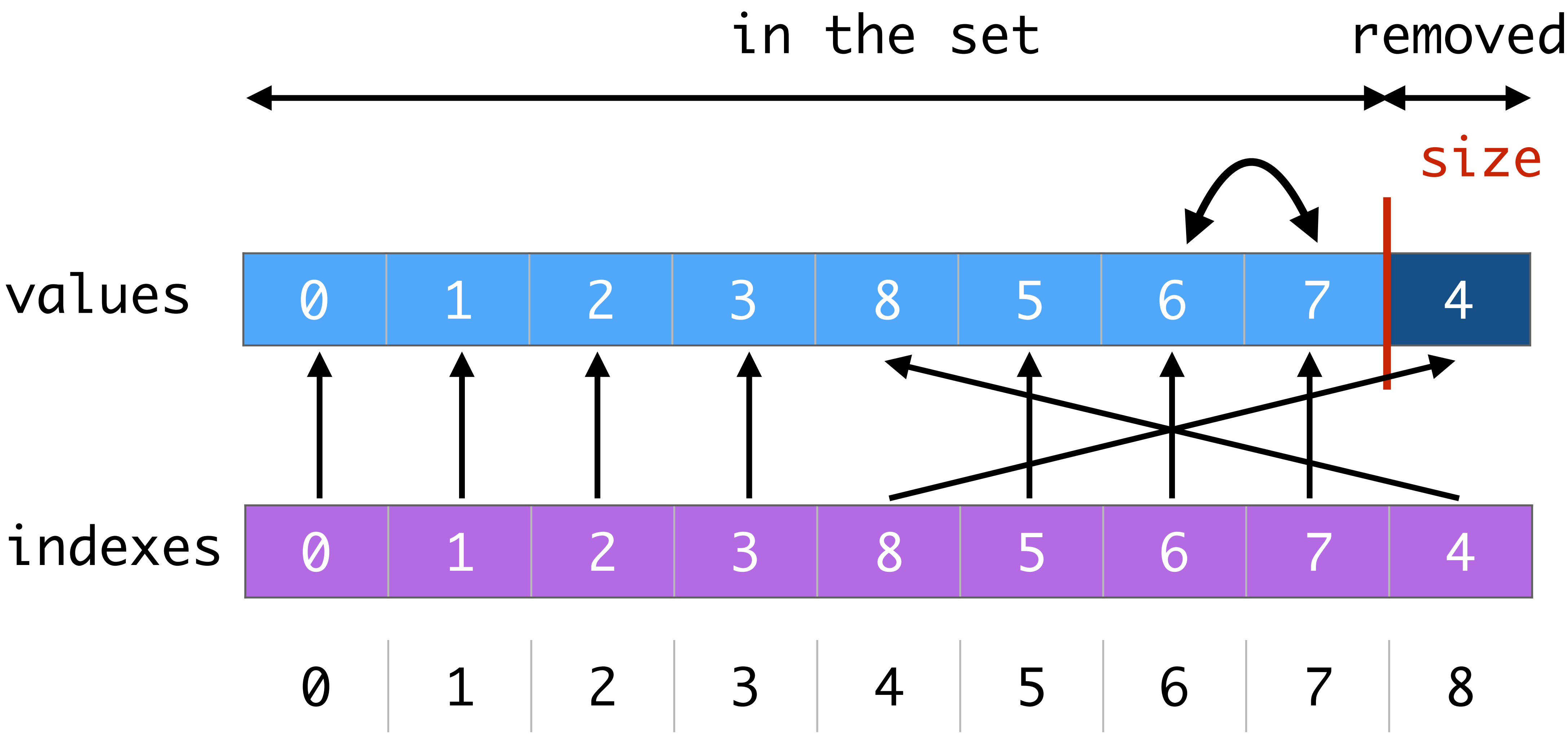
Removal operation

- Remove 4



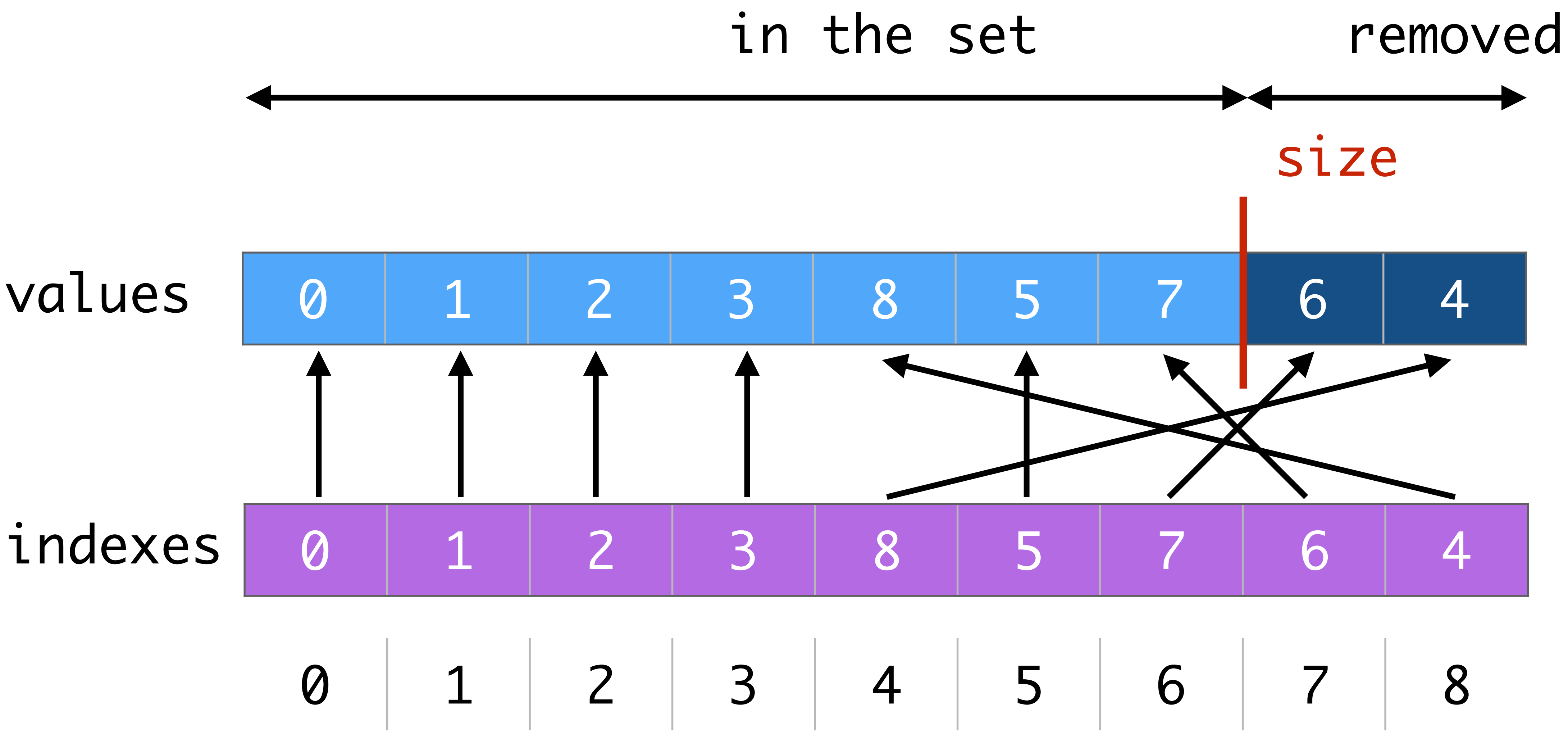
Removal operation

- Remove 6



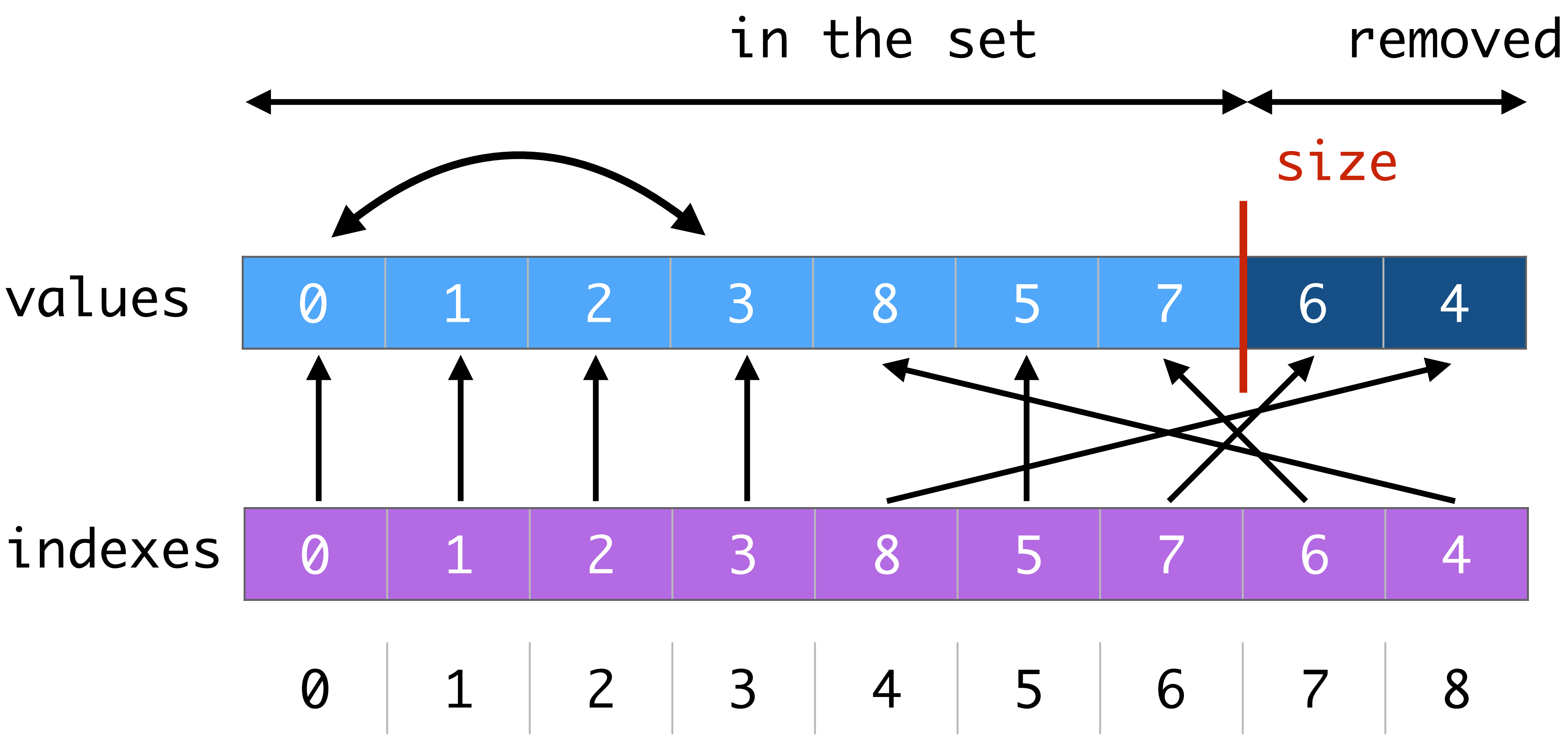
Removal operation

- Remove 6



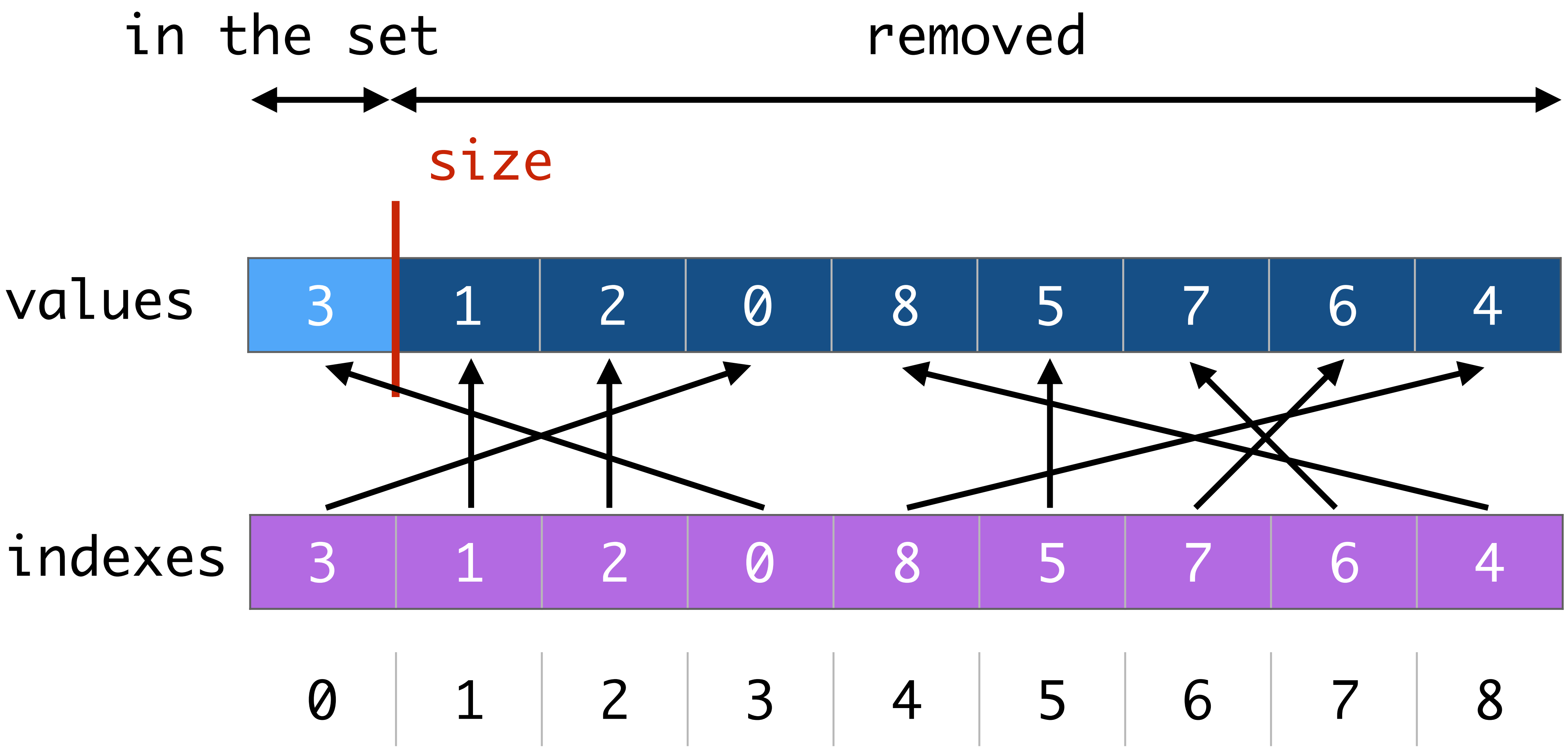
Removal operation

- Assignment operation: only keep 3



Removal operation

- Assignment operation: only keep 3 in the set



Sparset-Set API

```java
public class SparseSet {

    private int [] values;

    private int [] indexes;

    private int size;

    private int n;

    public boolean remove(int val) {...}

    public void removeAllBut(int v) {...}

    public boolean contains(int val) {
            return indexes[val] < size;
    }

}
```

Domain Listener

```java
public interface DomainListener {

    void bind();

    void change(int domainSize);

    void removeBelow(int domainSize);

    void removeAbove(int domainSize);
}
```

D(X)

Domain API

```java
public abstract class IntDomain {

    public abstract int getMin();

    public abstract  int getMax();

    public abstract  int getSize();

    public abstract  boolean contains(int v);

    public abstract  boolean isBound();

    public abstract  void remove(int v, DomainListener x)
             throws InconsistencyException;

    public abstract  void removeAllBut(int v, DomainListener x)
             throws InconsistencyException;

    public abstract  int removeBelow(int value, DomainListener x)
            throws InconsistencyException;

    public abstract  int removeAbove(int value, DomainListener x)
            throws InconsistencyException;

}
```

SparseSet Domain

```
public class SparseSetDomain extends IntDomain {

    private SparseSet domain;
    private int offset;

    public SparseSetDomain(int min, int max) {
        offset = min;
        domain = new SparseSet(max-min+1);
    }
    public int getMin() {
        return domain.getMin() + offset;
    }
    public void remove(int v, DomainListener x)
           throws InconsistencyException {
        if (domain.contains(v - offset)) {
            boolean maxChanged = getMax() == v;
            boolean minChanged = getMin() == v;
            domain.remove(v - offset);
            if (domain.getSize() == 0) throw INCONSISTENCY;
            x.change(domain.getSize());
            if (maxChanged) x.removeAbove(domain.getSize());
            if (minChanged) x.removeBelow(domain.getSize());
            if (domain.getSize() == 1) x.bind();
        }
    }
    . . .
}
```

Must be careful to notify correctly the listener

IntVarImpl 1/2

```
public class IntVarImpl implements IntVar {

    private Solver cp;
    private IntDomain domain;
    private Stack<Constraint> onDomain;
    private Stack<Constraint> onBind;


    public IntVarImpl(Solver cp, int min, int max) {
        this.cp = cp;
        domain = new SparseSetDomain(min,max);
        onDomain = new Stack<>();
        onBind  = new Stack<>();
    }

    public void propagateOnDomainChange(Constraint c) {
       onDomain.push(c);
    }

    public void propagateOnBind(Constraint c) {
       onBind.push(c);
    }


}
```

constraints interested to be called whenever the domain changes or if it bind

used by the constraint to register them-selves to the changes of the domains

IntVarImpl 2/2

```java
public class IntVarImpl implements IntVar {

    private DomainListener domListener = new DomainListener() {
        public void bind() { scheduleAll(onBind); }
        public void change(int domainSize){
         scheduleAll(onDomain);
        }
    };

    private void scheduleAll(Stack<Constraint> constraints) {
        for (int i = 0; i < constraints.size(); i++)
            cp.schedule(constraints.get(i));
    }

    public void remove(int v) throws InconsistencyException {
        domain.remove(v, domListener);
    }

    public void assign(int v) throws InconsistencyException {
        domain.removeAllBut(v, domListener);
    }

}
```

Schedule the constraints for the fix-point computation

Constraint API

```
public abstract class Constraint {

    protected final Solver cp;
    protected boolean scheduled = false;


    public Constraint(Solver cp) {
        this.cp = cp;
    }



    public abstract void post() throws InconsistencyException;

    public void propagate() throws InconsistencyException {}
}
```

state flag to avoid scheduling twice the constraint in the fix-point

setup the constraint:

- first check of consistency
- register propagation events
- often terminate by a call to propagate

the filtering

Constraint Example x != y + c

$$|D(y)| = 1 \Rightarrow \min(D(y)) + c \notin D(x)$$
$$|D(x)| = 1 \Rightarrow \min(D(x)) - c \notin D(y)$$

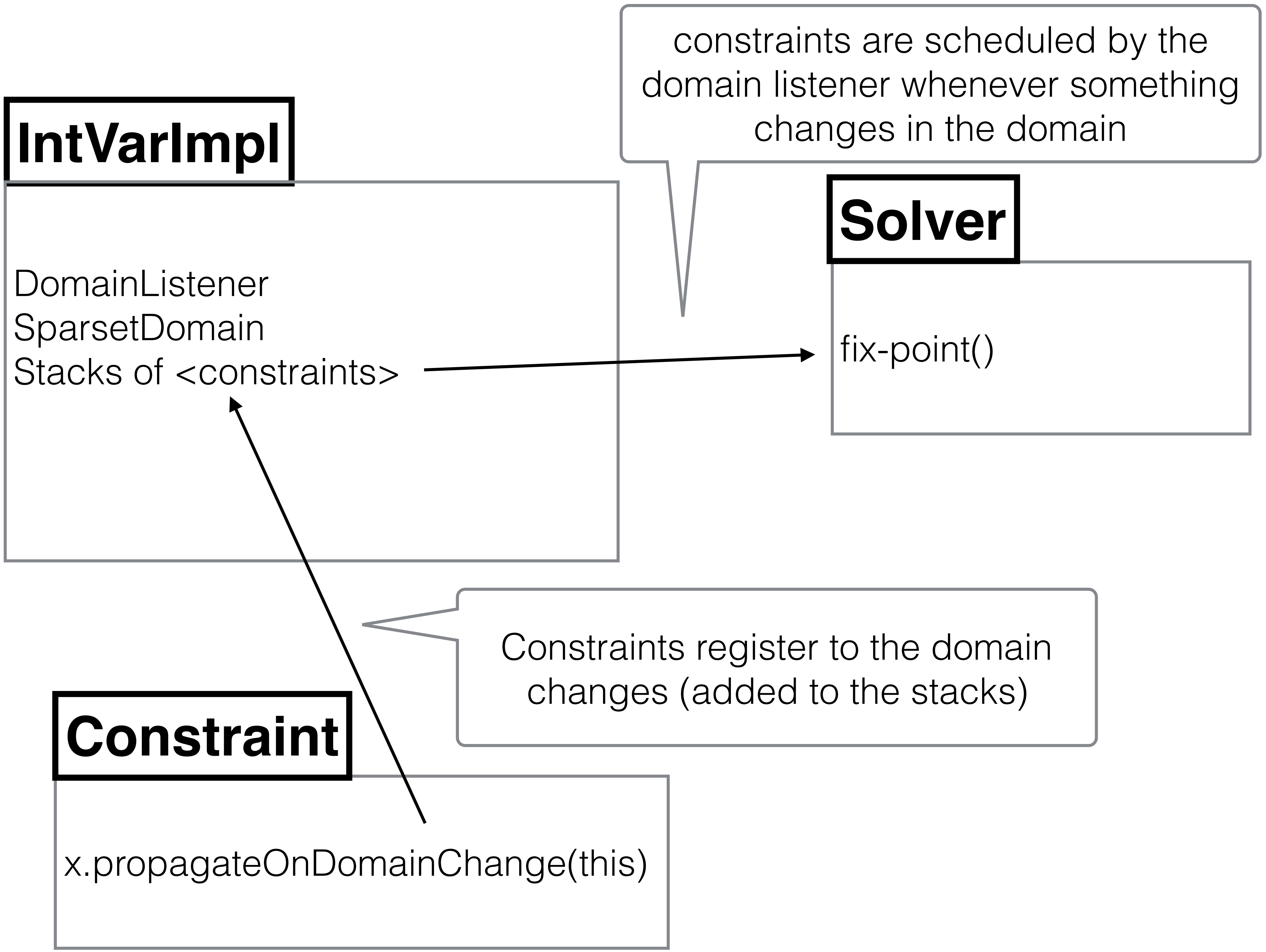
```
public class NotEqual extends Constraint {

    private IntVar x, y;
    private int c;

    public NotEqual(IntVar x, IntVar y, int c) {. . .}

    @Override
    public void post() throws InconsistencyException {
        if (y.isBound())
            x.remove(y.getMin() + c);
        else if (x.isBound())
            y.remove(x.getMin() - c);
        else {
            x.propagateOnBind(this);
            y.propagateOnBind(this);
        }
    }
    @Override
    public void propagate() throws InconsistencyException {
        if (y.isBound()) x.remove(y.getMin() + c);
        else y.remove(x.getMin() - c);
        this.deactivate();
    }
}
```

The solver and the fix-point

```
public class Solver {

    private Stack<Constraint> propagationQueue = new Stack<>();

    public void schedule(Constraint c) {
        if (!c.scheduled && c.isActive()) {
            c.scheduled = true;
            propagationQueue.add(c);
        }
    }
    public void fixPoint() throws InconsistencyException {
        boolean failed = false;
        while (!propagationQueue.isEmpty()) {
            Constraint c = propagationQueue.pop();
            c.scheduled = false;
            if (!failed) {
                try { c.propagate(); }
                catch (InconsistencyException e) {
                    failed = true;
                }
            }
        }
        if (failed) throw new InconsistencyException();
    }
    public void post(Constraint c, boolean enforceFixPoint)
                                          throws InconsistencyException {
        c.post();
        if (enforceFixPoint) fixPoint();
    }
}
```

Contains all the constraints scheduled for the fix-point

So far so good

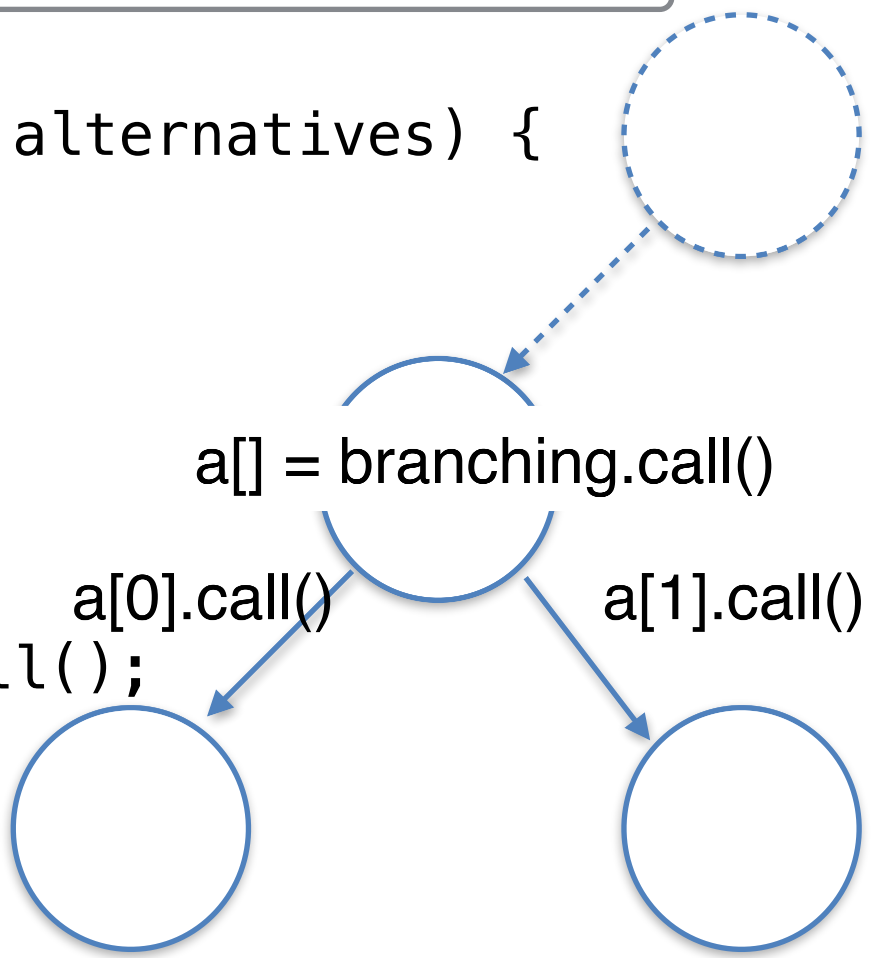


DFS Skeleton Implementation

```java
@FunctionalInterface
public interface Choice {
    Alternative[] call();
}
@FunctionalInterface
public interface Alternative {
    void call();
}
public static Alternative[] branch(Alternative... alternatives) {
    return alternatives;
}

public class DFS {
    private Choice branching;
    public DFS(Choice b)  { branching = b;}
    public void dfs() {
        Alternative[] alternatives = branching.call();
        if (alternatives.length == 0)
            notifySolution();
        else
            for (a : alternatives) {
                a.call();
                dfs();
            }
    }
}
```

API for creation of child nodes.

Generate the child nodes



call the closure before recursion

Not enough: we need state restoration

- because what we typically want to do is with branch is:

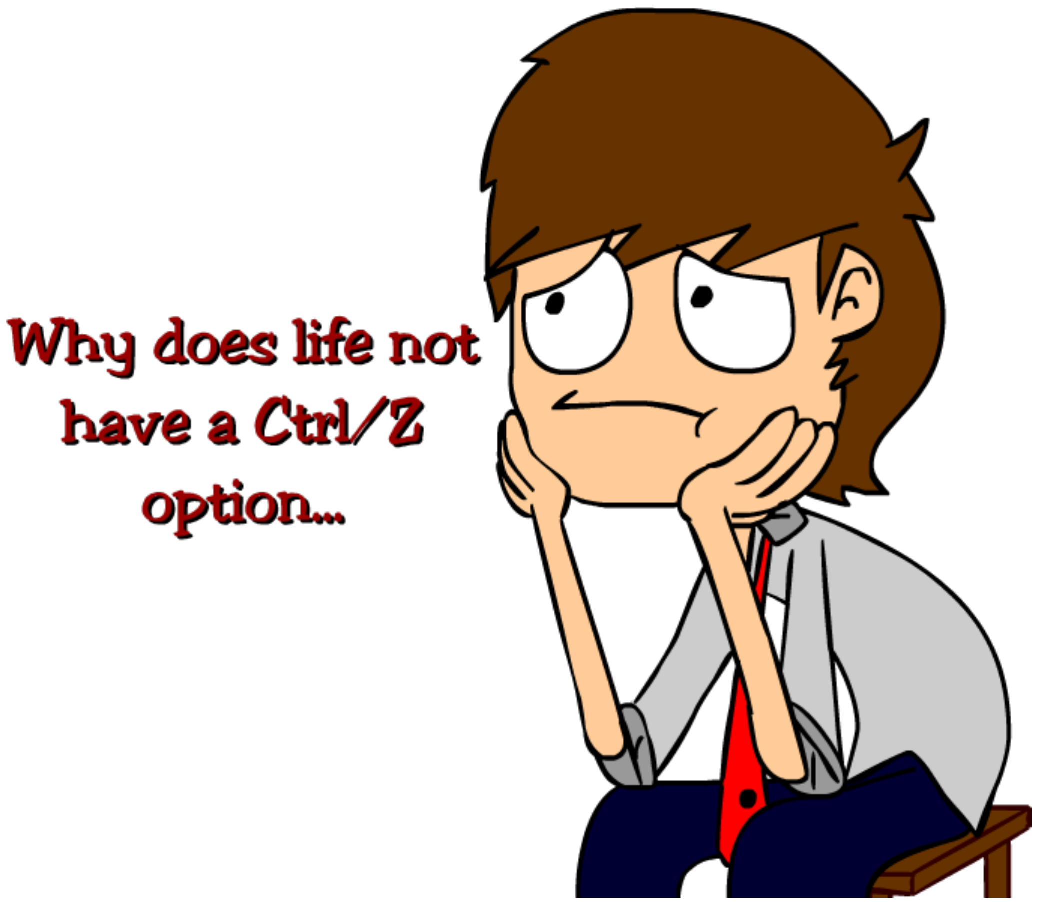
```
int n = 8;
Solver cp = makeSolver();
IntVar[] q = makeIntVarArray(cp, n, n);
. . .
SearchStatistics stats = makeDfs(cp,
        selectMin(q,
                qi -> qi.getSize() >
                qi -> qi.getSize(),
                qi -> {
                    int v = qi.getMin();
                    return branch(() -> equal(qi,v),
                                  () -> notEqual(qi,v));
                }
        )
).onSolution(() ->
        System.out.println("solu
).start();
```

will trigger the fix-point, shrink the domains, etc

and everything needs to be restored before backtracking and trying the alternative branch

The answer = The Trail

- The trail = a mechanism for doing and undoing



- before doing => **trail.push()**

- undoing => **trail.pop()**

State restoration with Trail

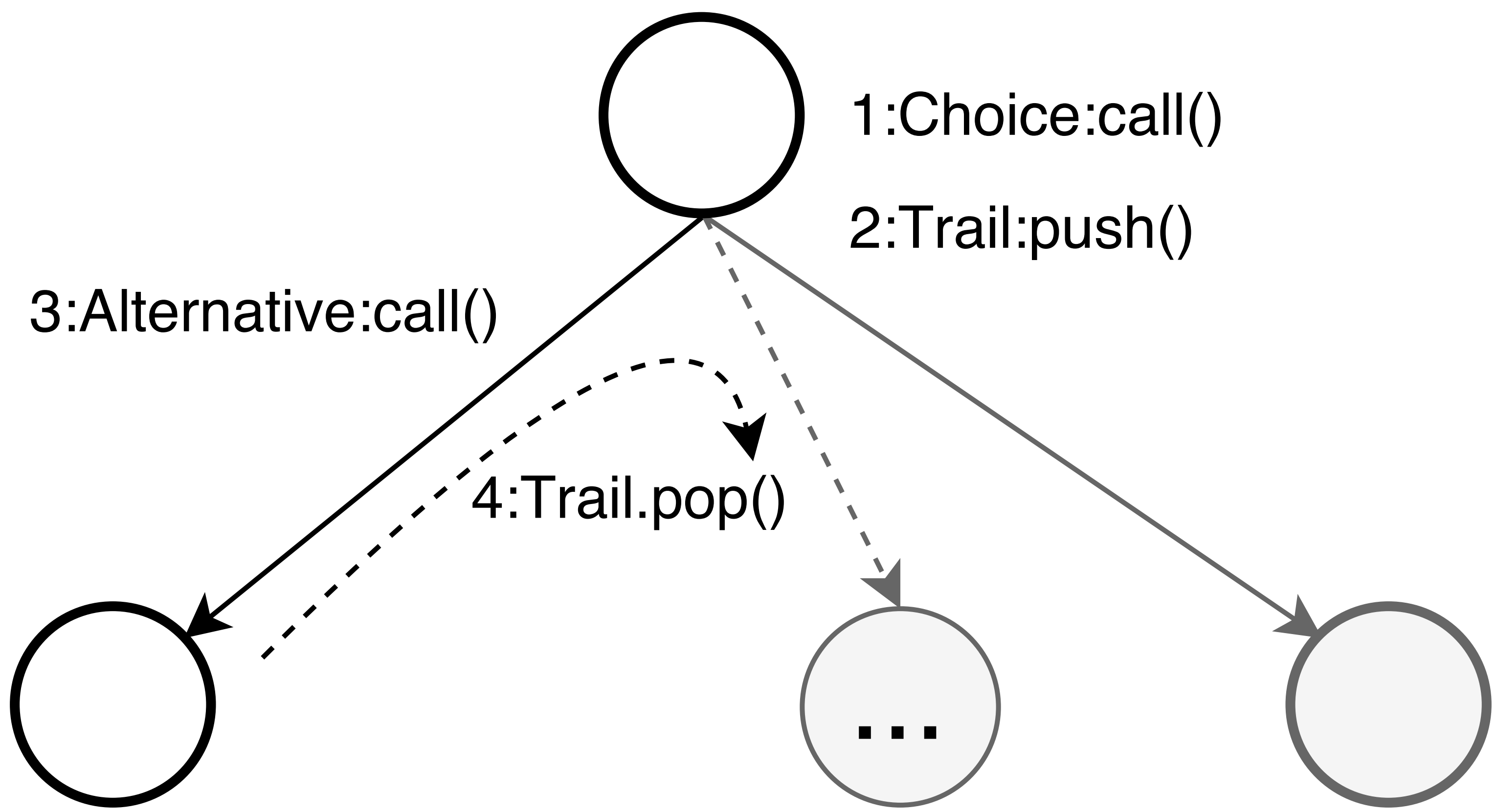
```
Trail trail = new Trail();

ReversibleInt a = new ReversibleInt(trail,7);
ReversibleInt b = new ReversibleInt(trail,13);

trail.push();     // record a=7, b=13
 a.setValue(11);
 b.setValue(14);
 trail.push();     // record a=11, b=14
  a.setValue(4);
  b.setValue(9);
 trail.pop();      // restore a=11, b=14
trail.pop();      // restore a=7, b=13
```

like an integer except that we can undo the changes with the push()/pop() of the trail

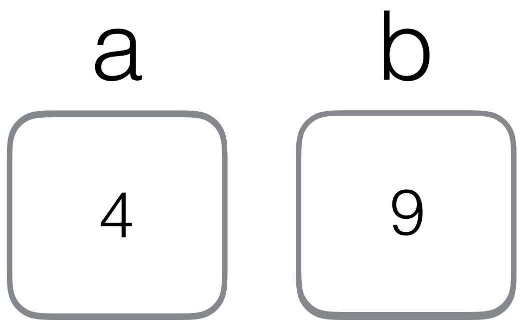
This is exactly what we need for the search

- Assume all our objects are reversibles (domains, variables, etc)

- The search would do

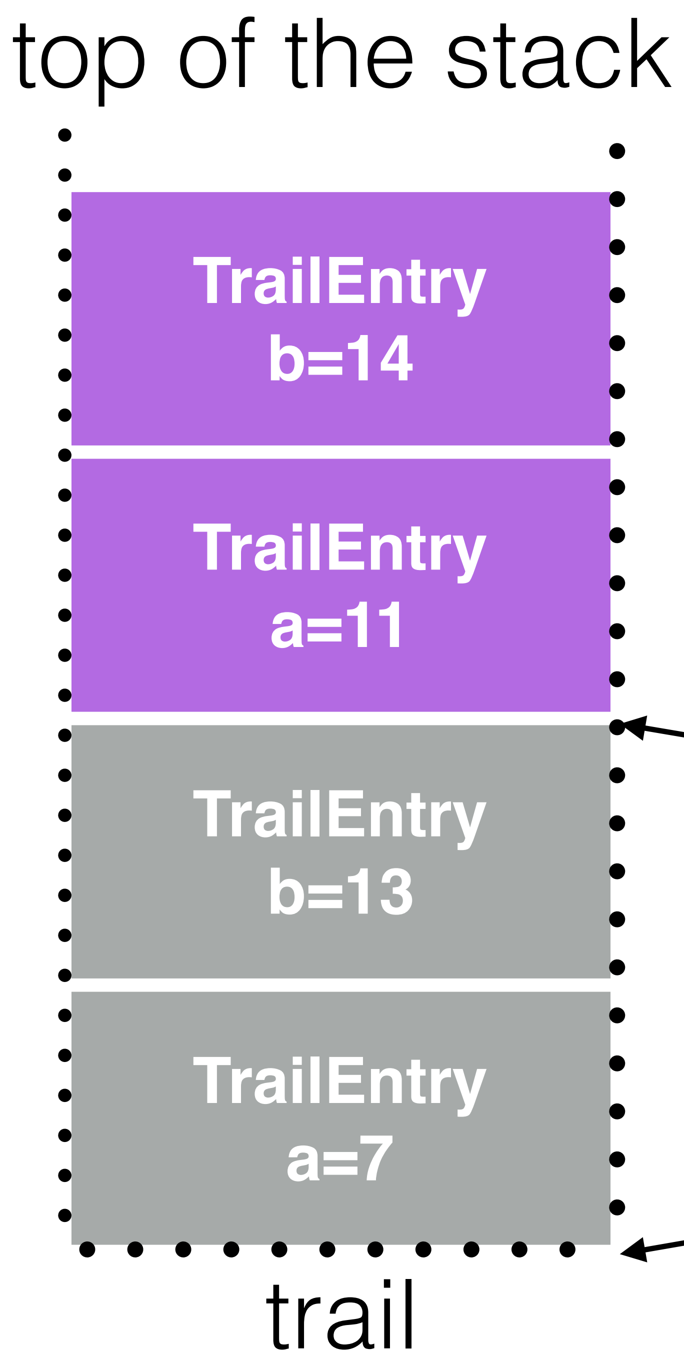


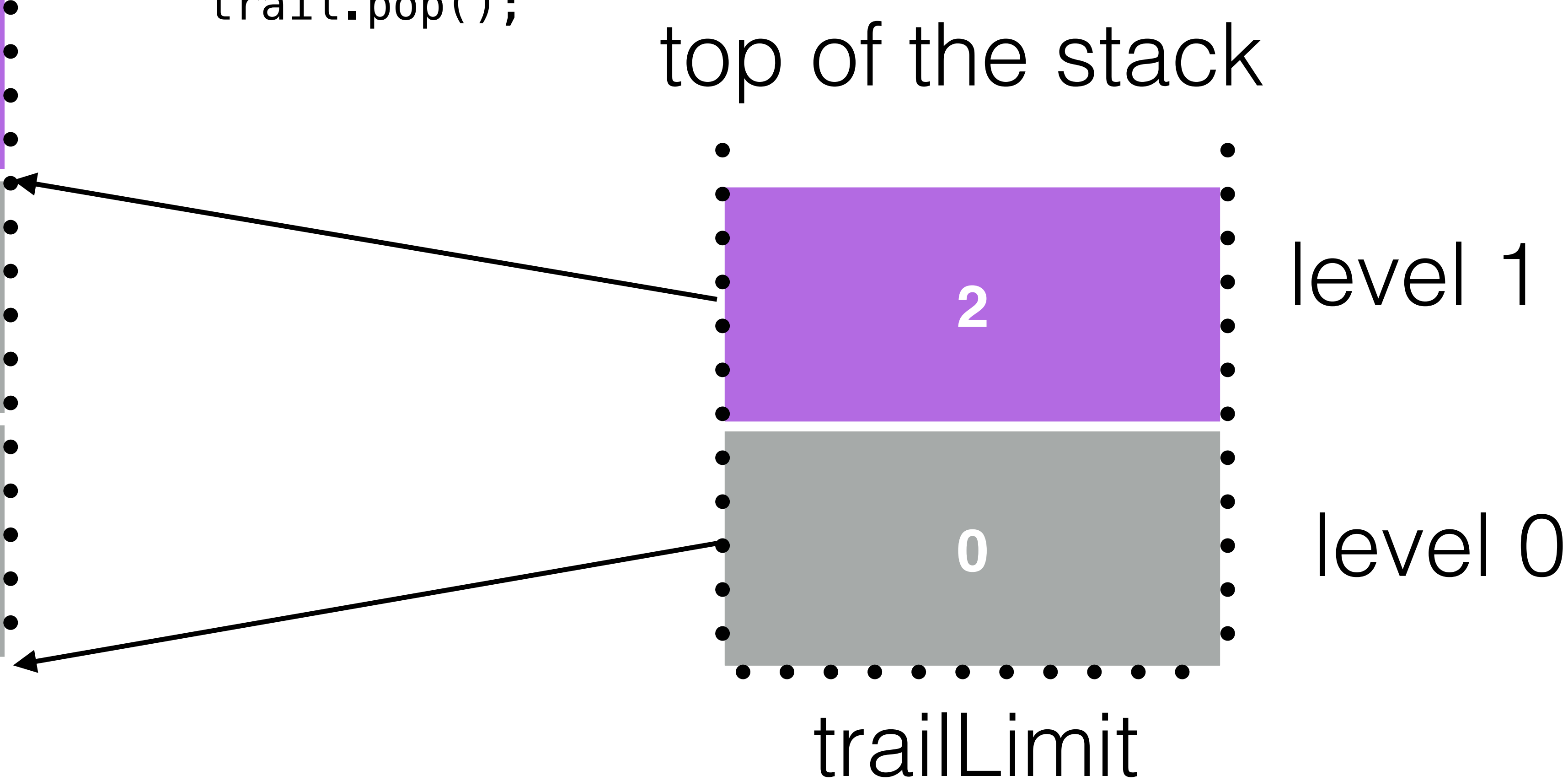
How is this trail implemented??




```java
public interface TrailEntry {
    public void restore();
}
```

```java
Trail trail = new Trail();

ReversibleInt a = new ReversibleInt(trail,7);
ReversibleInt b = new ReversibleInt(trail,13);

trail.push();
  a.setValue(11);
  b.setValue(14);
  trail.push();
    a.setValue(4);
    b.setValue(9);
  trail.pop();
trail.pop();
```





Trail Implementation

```
public class Trail {
    public long magic = 0;
    private Stack<TrailEntry> trail = new Stack<TrailEntry>();
    private Stack<Integer>    trailLimit = new Stack<Integer>();

    public void pushOnTrail(TrailEntry entry) {
        trail.push(entry);
    }

    public void push(){
        magic++;
        trailLimit.push(trail.size());
    }

    public void pop() {
        int n = trail.size() - trailLimit.pop();
        for (int i = 0; i < n; i++) trail.pop().restore();
        magic++;
    }
}
```

```
public interface TrailEntry {
    public void restore();
}
```

ReversibleInteger (not optimized)

```
public class ReversibleInt implements RevInt {

   class TrailEntryInt implements TrailEntry {
        private final int v;
        public TrailEntryInt(int v) {
            this.v = v;
        }
        public void restore() { ReversibleInt.this.v = v;}
    }

    private Trail trail;
    private int v;

    public ReversibleInt(Trail trail, int initial) {. . . }


    public int setValue(int v) {
        if (v != this.v) {
            trail.pushOnTrail(new TrailEntryInt(v));
            this.v = v;
        }
        return this.v;
    }
}
```

restore the value

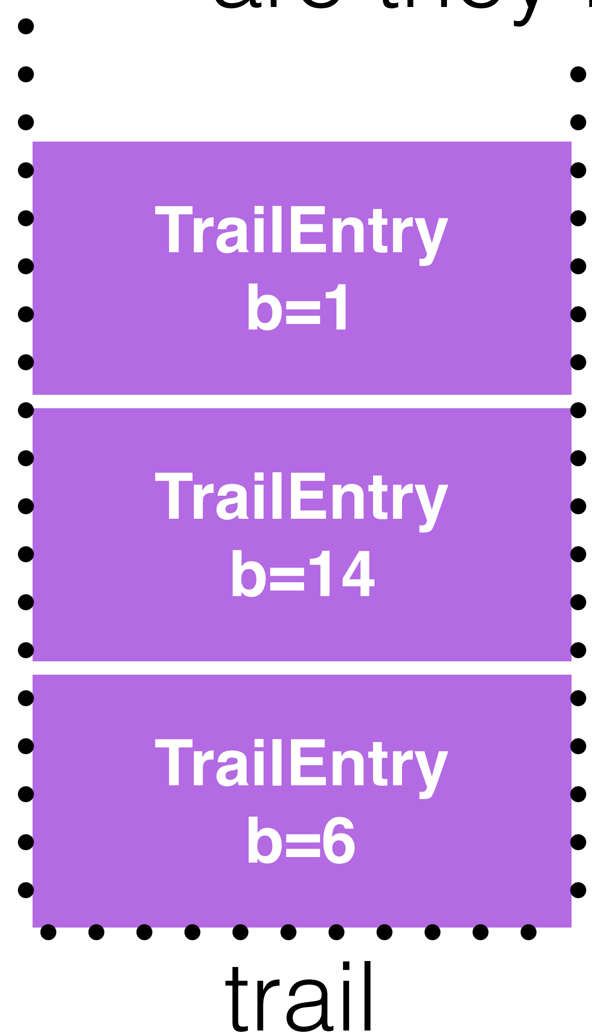
create and stack the trail entry containing v before replacing it (only if the new value is different)

Implementation trick

```
b.setValue(6);

trail.push();
 a.setValue(11);
 b.setValue(14);
 b.setValue(1);
 b.setValue(4);
 trail.push();
  a.setValue(4);
  b.setValue(9);
 trail.pop();
trail.pop();
```

only the one created at that time is useful. The value b=14 and b=1 will never be restored

3 trail entries are stacked on the trail, are they really necessary?



ReversibleInteger

```
public class ReversibleInt implements RevInt {

    private Trail trail;
    private int v;
    private Long lastMagic = -1L;

    private void trail() {
        long trailMagic = trail.magic;
        if (lastMagic != trailMagic) {
            lastMagic = trailMagic;
            trail.pushOnTrail(new TrailEntryInt(v));
        }
    }

    public int setValue(int v) {
        if (v != this.v) {
            trail();
            this.v = v;
        }
        return this.v;
    }
}
```

time-stamping coming from the trail

if the time-stamp is the same the relevant TrailEntry already exists

otherwise create it and record the trail time-stamp

What do we need to make solver state reversible?

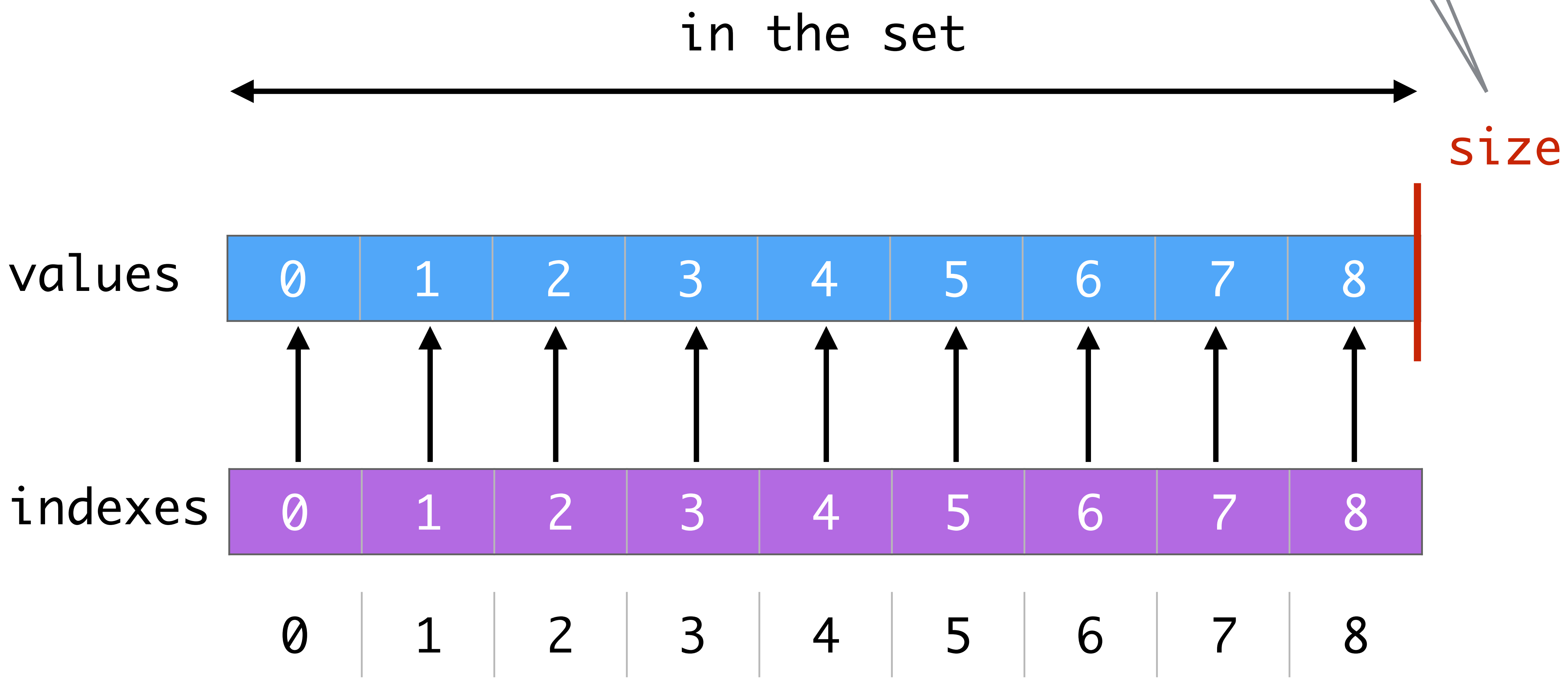
Wee need:

- Reversible Domains => Reversible Sparse-Sets
- Reversible addition of constraints (they must withdraw upon backtrack
- Reverse all the state that you can possibly put inside the constraints
  - Constraint implementors should only focus on incremental aspects down in the search tree

Reversible Sparset-Set

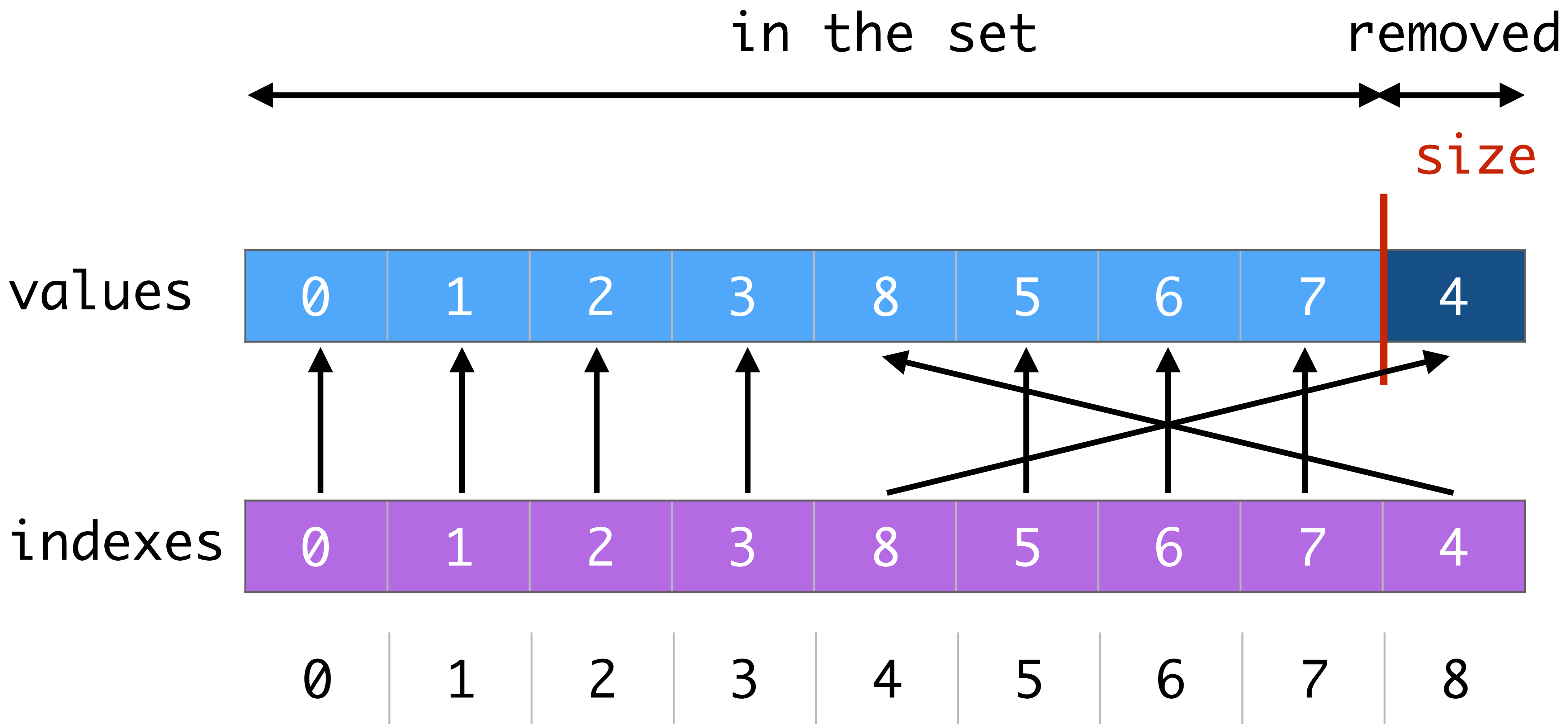
```
Trail trail = new Trail();
ReversibleSparseSet set = new ReversibleSparseSet(trail,9);
trail.push();
set.remove(4);
```

All we need to change is
size is now a ReversibleInt



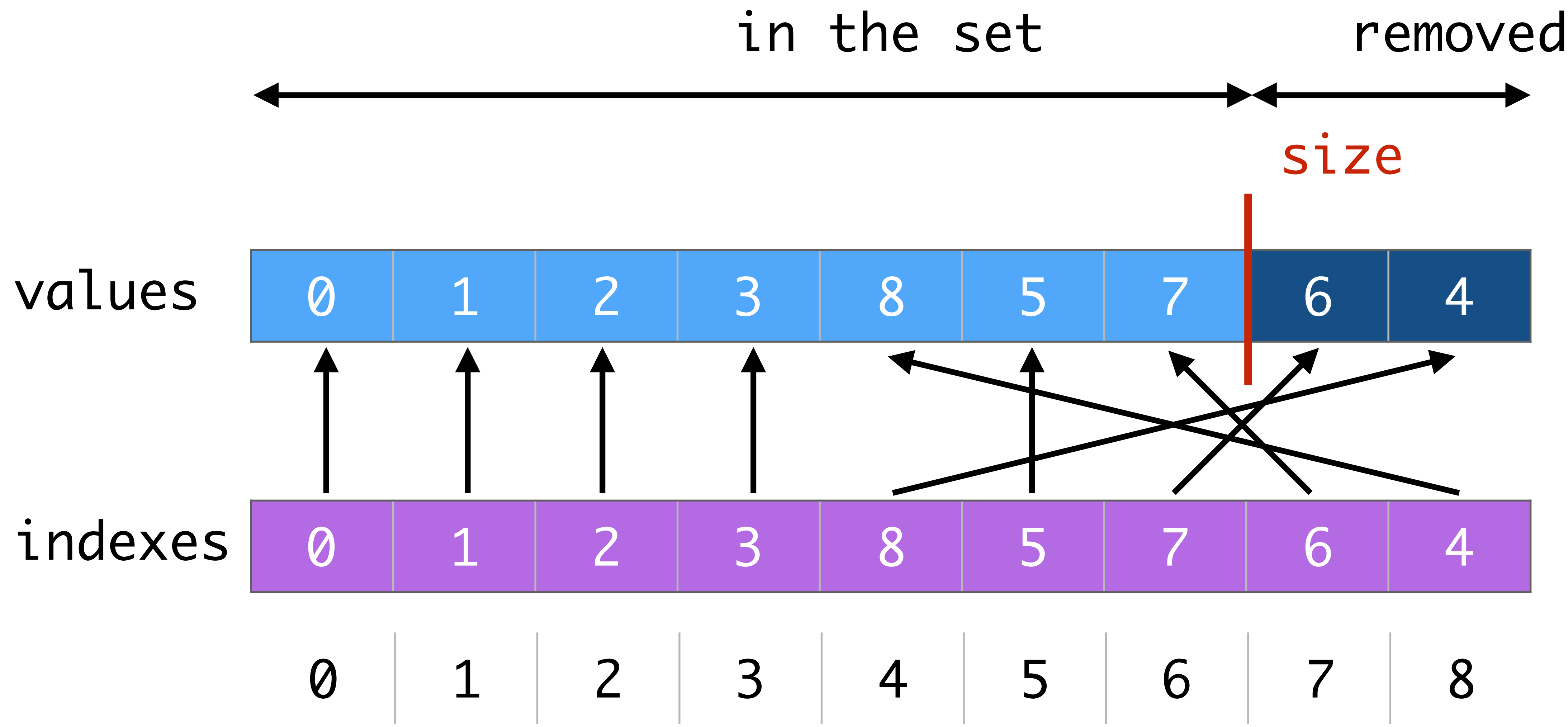
Removal operation

```
Trail trail = new Trail();
ReversibleSparseSet set = new ReversibleSparseSet(trail,9);
trail.push();
set.remove(4);
set.remove(6);
```



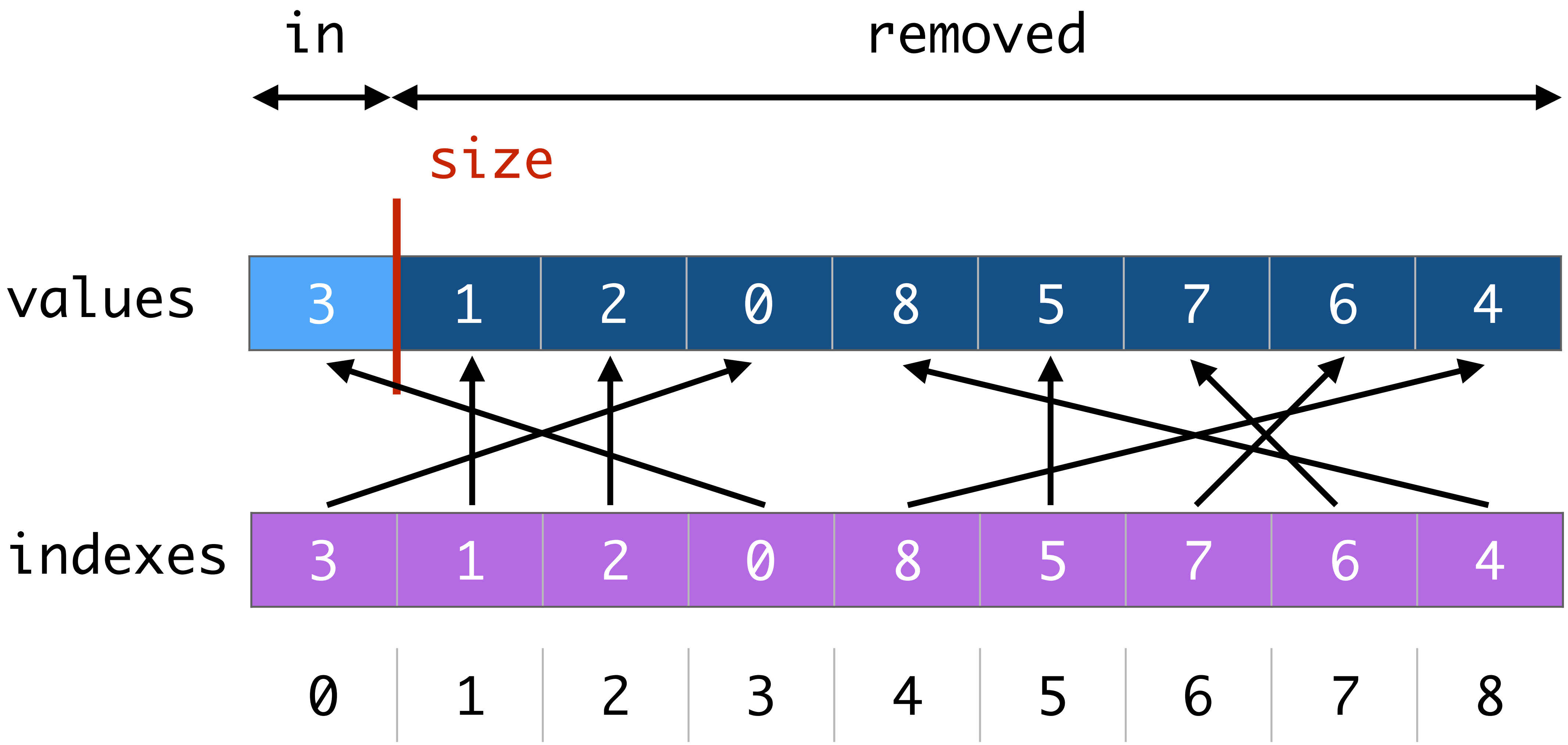
Removal operation

```
Trail trail = new Trail();
ReversibleSparseSet set = new ReversibleSparseSet(trail,9);
trail.push();
set.remove(4);
set.remove(6);
train.push()
set.assign(3);
```



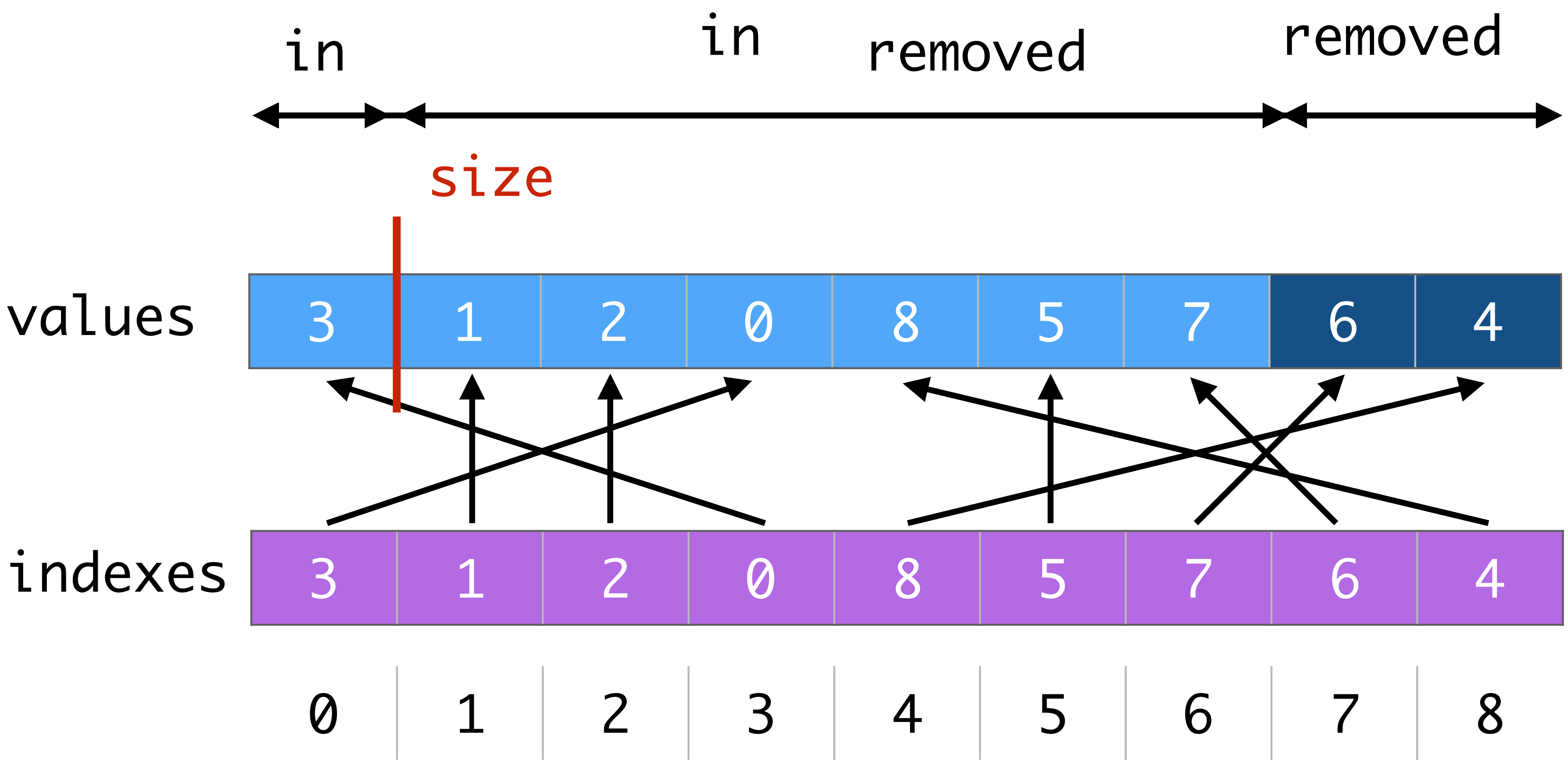
Removal operation

```
Trail trail = new Trail();
ReversibleSparseSet set = new ReversibleSparseSet(trail,9);
trail.push();
set.remove(4);
set.remove(6);
train.push()
set.assign(3);
trail.pop();
```



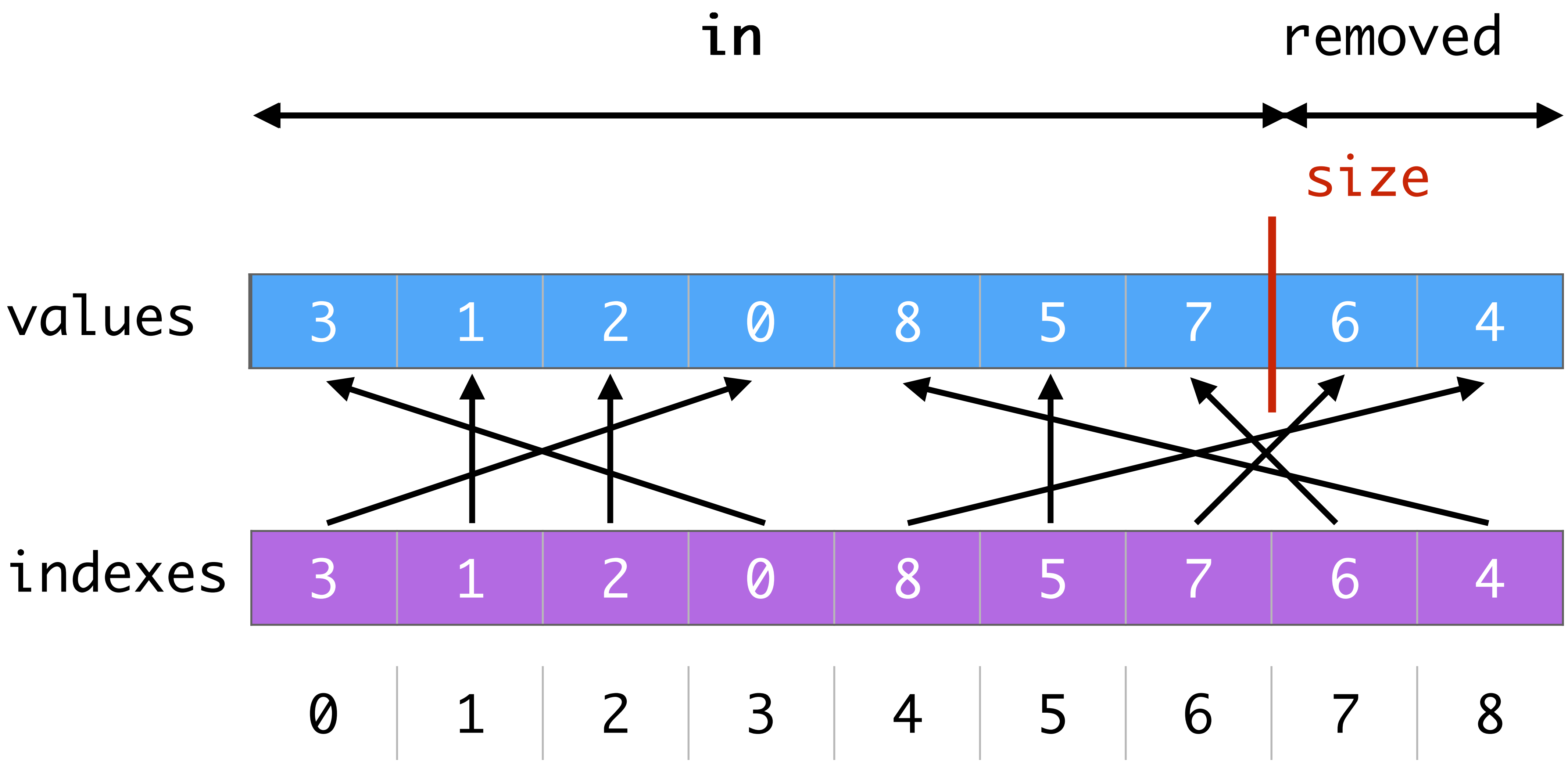
Removal operation

```
Trail trail = new Trail();
ReversibleSparseSet set = new ReversibleSparseSet(trail,9);
trail.push();
set.remove(4);
set.remove(6);
train.push()
set.assign(3);
trail.pop(); // {0,1,2,3,5,7,8}
```

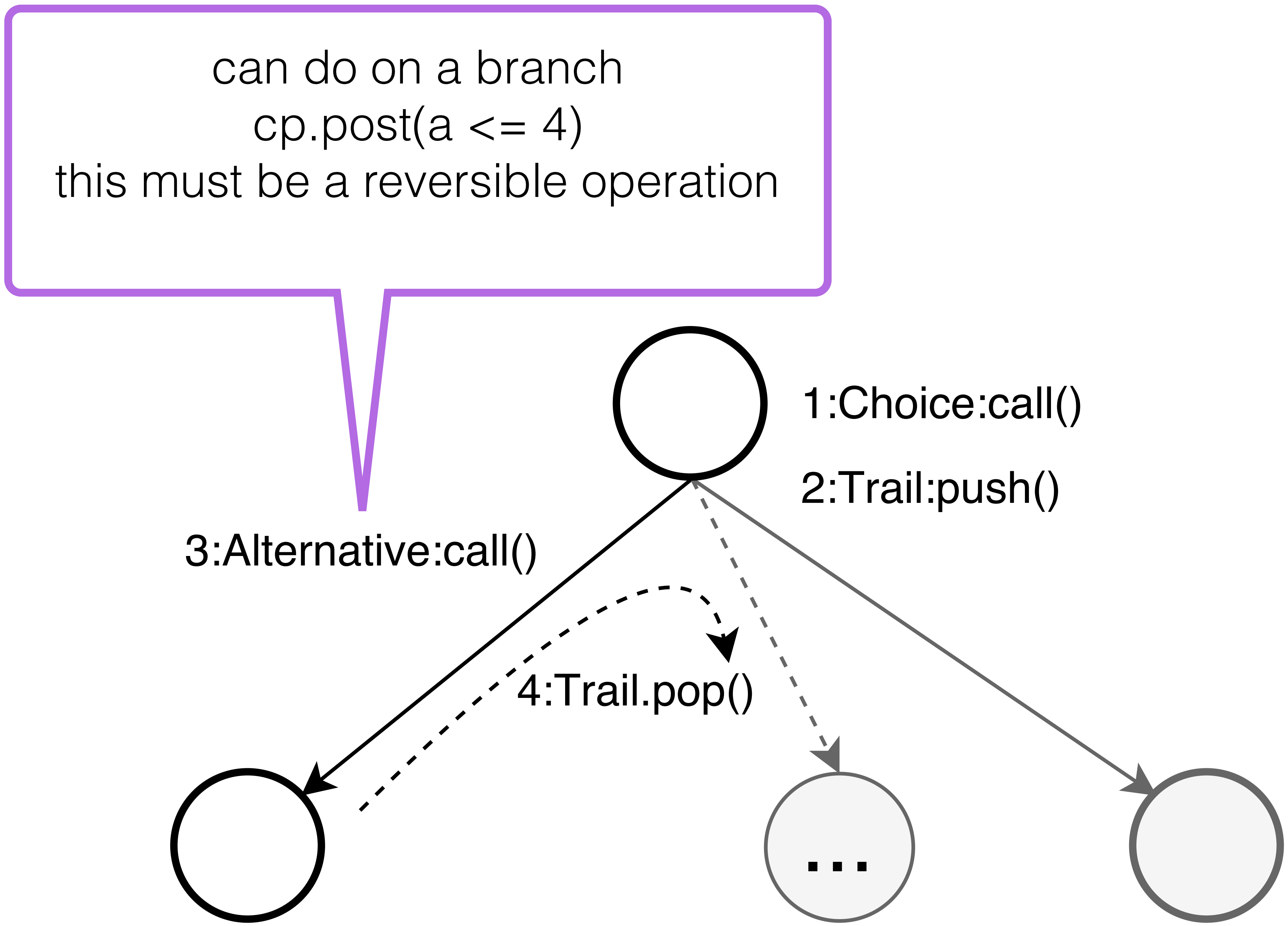


Removal operation

```
Trail trail = new Trail();
ReversibleSparseSet set = new ReversibleSparseSet(trail,9);
trail.push();
set.remove(4);
set.remove(6);
train.push()
set.assign(3);
trail.pop(); // {0,1,2,3,5,7,8}
trail.pop(); // {0..9}
```



Adding a constraint = reversible operation



IntVarImpl: Making it reversible

encapsulates a **ReversibleSparseSet**

```
public class IntVarImpl implements IntVar {

    private Solver cp;
    private IntDomain domain;
    private ReversibleStack<Constraint> onDomain;
    private ReversibleStack<Constraint> onBind;

    private DomainListener domListener = new DomainListener() {
        public void bind() { scheduleAll(onBind); }
        public void change(int domainSize){
         scheduleAll(onDomain);
        }
    };
    public IntVarImpl(Solver cp, int min, int max) {
        this.cp = cp;
        cp.registerVar(this);
        domain = new SparseSetDomain(cp.getTrail(),min,max);
        onDomain = new ReversibleStack<>(cp.getTrail());
        onBind  = new ReversibleStack<>(cp.getTrail());
    }



}
```

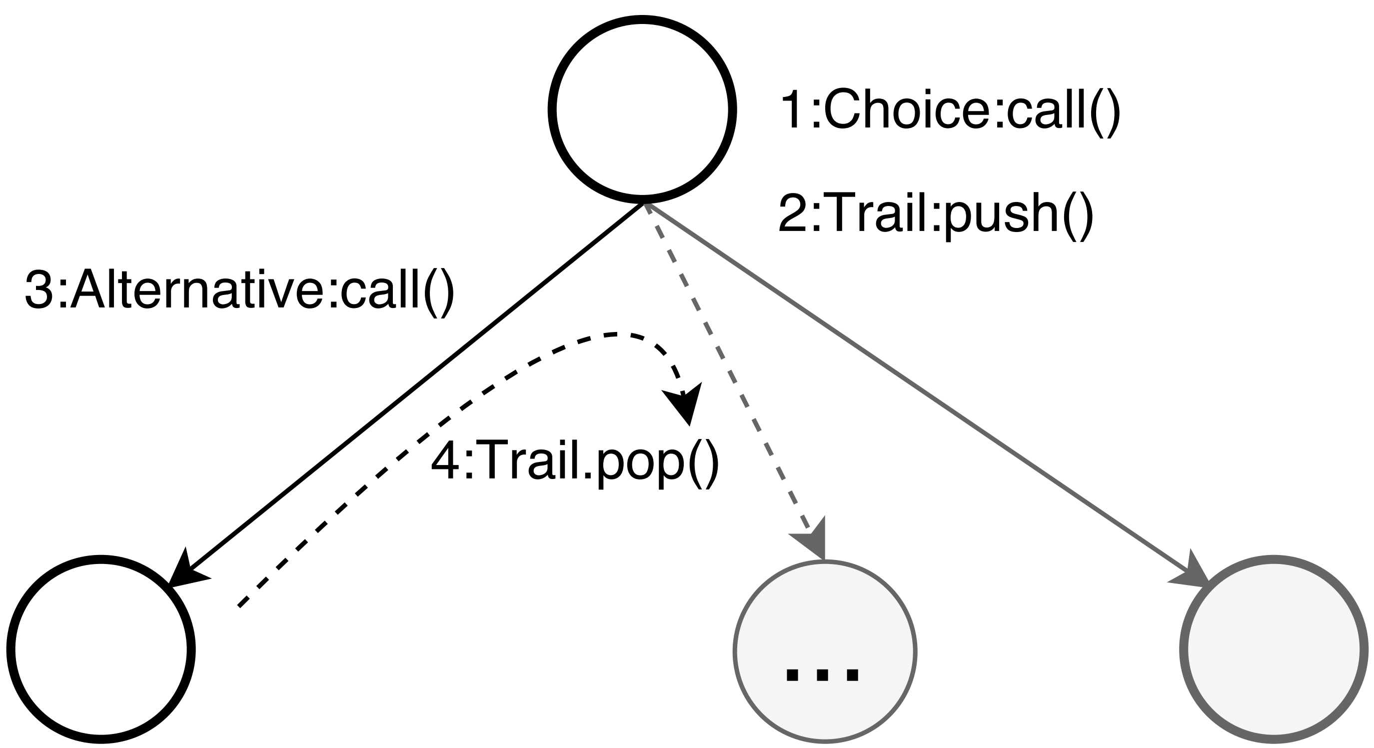
ReversibleStack

```
public class ReversibleStack<E> {

    ReversibleInt size;
    ArrayList<E> stack;

    public ReversibleStack(Trail rc) {
        size = new ReversibleInt(rc,0);
        stack = new ArrayList<E>();
    }

    public void push(E elem) {
        stack.add(size.getValue(),elem);
        size.increment();
    }

    public int size() { return size.getValue(); }

    public E get(int index) {
       return stack.get(index);
    }
}
```

All we need to change is
size = ReversibleInt

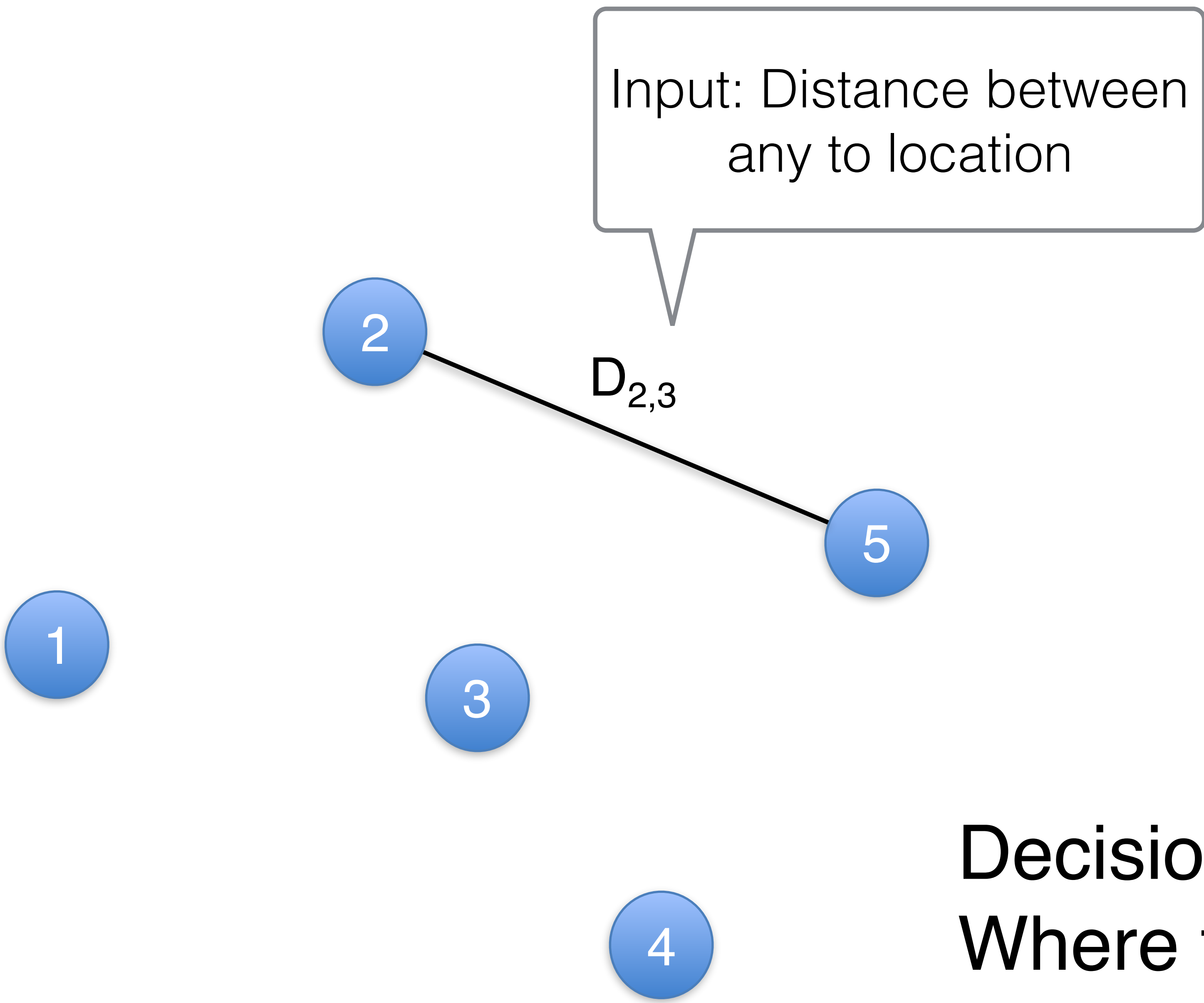
DFS with explicit state restoration

```java
public class DFS {
    private Trail  trail;
    private Choice branching;

    public DFS(Trail t,Choice b)  { . . . }

    public void dfs() {
        Alternative[] alternatives = branching.call();
        if (alternatives.length == 0)
            notifySolution();
        else
            for (a : alternatives) {
                trail.push();
                a.call();
                dfs();
                trail.pop();
            }
    }
}
```

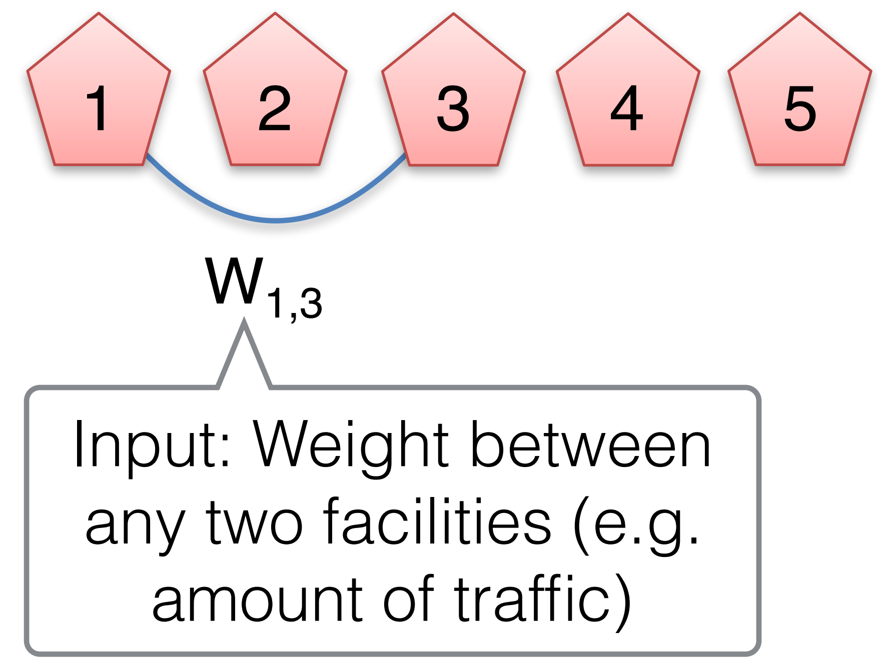


Quadratic Assignment Problem (QAP)

Locations:



Facilities:

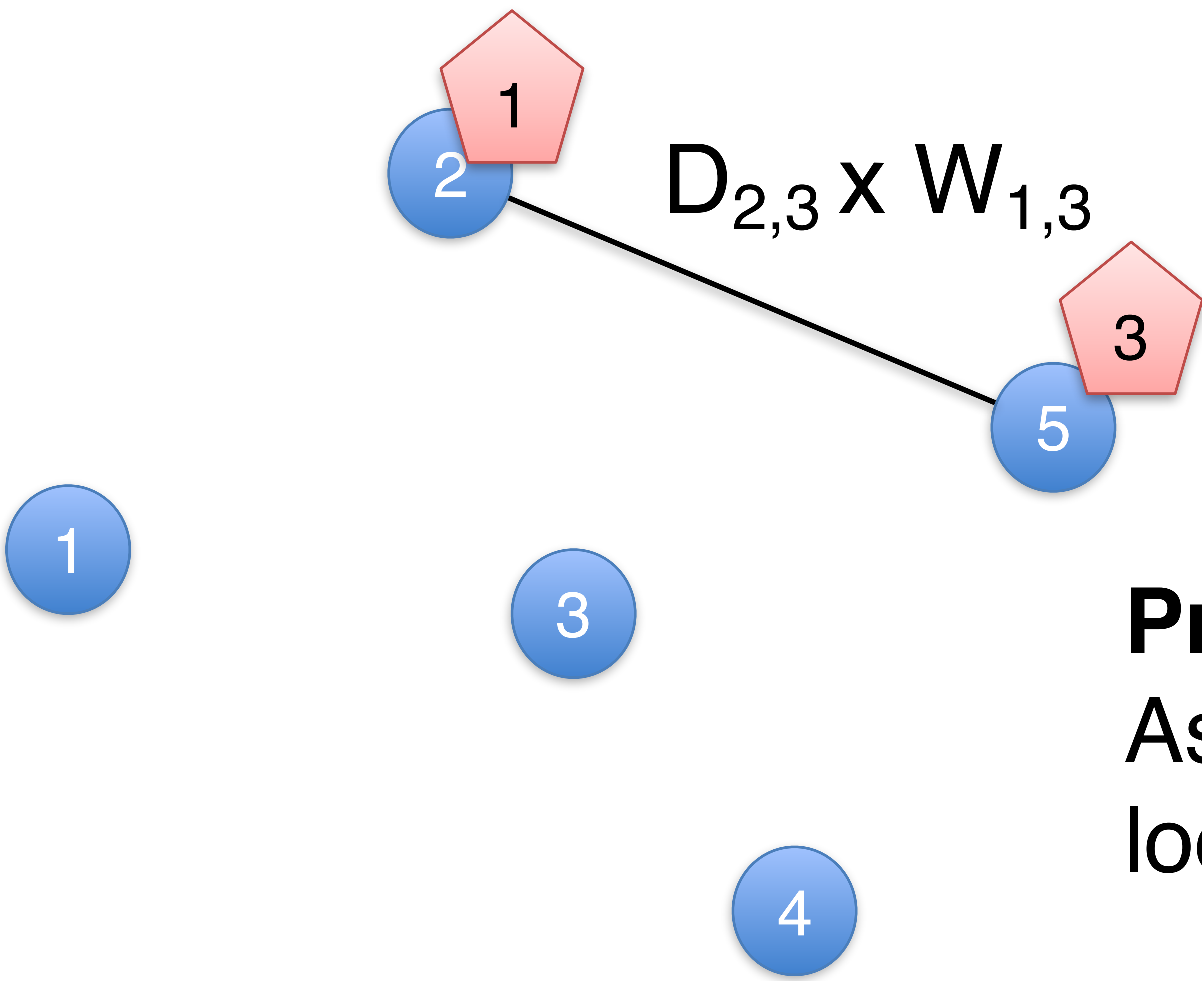


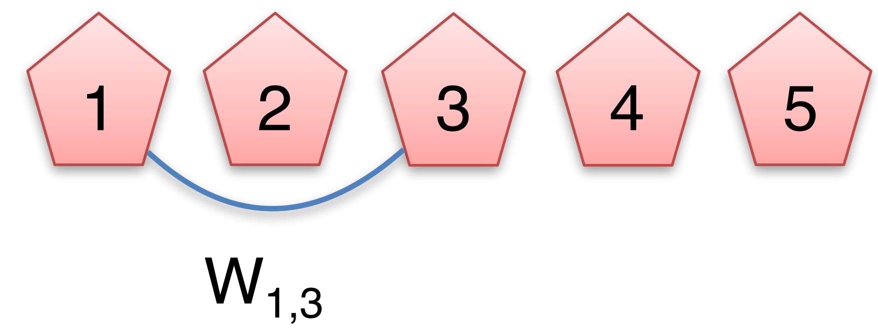
Decision:
Where to place each warehouse?

QAP

Locations:

Facilities:





Problem:
Assign one facility to each location **minimizing** $\sum_{i,j} D_{x_i,x_j} \cdot W_{i,j}$

2D element constraint =
2D array indexed by two variables

Quadratic Assignment Model

```
Solver cp = makeSolver();
IntVar[] x = makeIntVarArray(cp, n, n);

cp.post(allDifferent(x));



// build the objective function
IntVar[] weightedDist = new IntVar[n*n];
int ind = 0;
for (int i = 0; i < n; i++) {
    for (int j = 0; j < n; j++) {
        weightedDist[ind] = mul(element(d,x[i],x[j]),w[i][j]);
        ind++;
    }
}
IntVar objective = sum(weightedDist);

DFSearch dfs = makeDfs(cp,firstFail(x));

cp.post(minimize(objective,dfs));
```

$$D_{x_i,x_j} \cdot W_{i,j}$$

Element2D(**int[][]** T, IntVar x, IntVar y, IntVar z)

- T[x][y] = z

y

x	0	1	2	3
0	**1**	**8**	**9**	**6**
1	**1**	**9**	**2**	**4**
2	**9**	**8**	**9**	**8**
3	**1**	**9**	**2**	**5**

- How to create an efficient propagator for Element2D?

- Don't want to create holes in D(z) but well in D(x) and D(y)

T[x][y] = z

y

x

	0	1	2	3	rSup
0	1	8	9	6	4
1	1	9	2	4	4
2	9	8	9	8	4
3	1	9	2	5	4
cSup	4	4	4	4	

- D(x) = {0,1,2,3}
- D(y) = {0,1,2,3}
- D(z) = [1..9] (interval domain)

sorted

z	x	y
1	0	0
1	1	0
1	3	0
2	1	2
2	3	2
4	1	3
5	3	3
6	0	3
8	0	1
8	2	1
8	2	3
9	0	2
9	1	1
9	2	0
9	2	2
9	3	1

low

up

T[x][y] = z

y

x

	0	1	2	3	rSup
0	**1**	**8**	**9**	**6**	**4**
1	**1**	**9**	**2**	**4**	**4**
2	**9**	**8**	**9**	**8**	**4**
3	**1**	**9**	**2**	**5**	**4**
cSup	**4**	**4**	**4**	**4**	

- D(x) = {0,1,2,3}
- D(y) = {0,1,2,3}
- D(z) = [1..7] (interval domain)

sorted

low

z	x	y
1	0	0
1	1	0
1	3	0
2	1	2
2	3	2
4	1	3
5	3	3
6	0	3
8	0	1
8	2	1
8	2	3
9	0	2
9	1	1
9	2	0
9	2	2
9	3	1

up

T[x][y] = z

y

x	0	1	2	3	rSup
0	1	8	9	6	4
1	1	9	2	4	4
2	9	8	9	8	4
3	1	9	2	5	3
cSup	4	3	4	4	

- D(x) = {0,1,2,3}
- D(y) = {0,1,2,3}
- D(z) = [1..7] (interval domain)

sorted

	z	x	y
low	1	0	0
	1	1	0
	1	3	0
	2	1	2
	2	3	2
	4	1	3
	5	3	3
	6	0	3
	8	0	1
	8	2	1
	8	2	3
	9	0	2
	9	1	1
	9	2	0
up	9	2	2
	9	3	1

T[x][y] = z

y

x

	0	1	2	3	rSup
0	1	8	9	6	4
1	1	9	2	4	4
2	9	8	9	8	3
3	1	9	2	5	3
cSup	4	3	3	4	

- D(x) = {0,1,2,3}
- D(y) = {0,1,2,3}
- D(z) = [1..7] (interval domain)

sorted

	z	x	y
low	1	0	0
	1	1	0
	1	3	0
	2	1	2
	2	3	2
	4	1	3
	5	3	3
	6	0	3
	8	0	1
	8	2	1
	8	2	3
	9	0	2
	9	1	1
up	9	2	0
	9	2	2
	9	3	1

T[x][y] = z

y

x

	0	1	2	3	rSup
0	1	8	9	6	4
1	1	9	2	4	4
2	9	8	9	8	2
3	1	9	2	5	3
cSup	3	3	3	4	

- D(x) = {0,1,2,3}
- D(y) = {0,1,2,3}
- D(z) = [1..7] (interval domain)

sorted

low

up

z	x	y
1	0	0
1	1	0
1	3	0
2	1	2
2	3	2
4	1	3
5	3	3
6	0	3
8	0	1
8	2	1
8	2	3
9	0	2
9	1	1
9	2	0
9	2	2
9	3	1

T[x][y] = z

y

x

	0	1	2	3	rSup
0	1	8	9	6	4
1	1	9	2	4	3
2	9	8	9	8	2
3	1	9	2	5	3
cSup	3	2	3	4	

- D(x) = {0,1,2,3}
- D(y) = {0,1,2,3}
- D(z) = [1..7] (interval domain)

sorted

low

up

z	x	y
1	0	0
1	1	0
1	3	0
2	1	2
2	3	2
4	1	3
5	3	3
6	0	3
8	0	1
8	2	1
8	2	3
9	0	2
9	1	1
9	2	0
9	2	2
9	3	1

T[x][y] = z

y

x

	0	1	2	3	rSup
0	1	8	9	6	3
1	1	9	2	4	3
2	9	8	9	8	2
3	1	9	2	5	3
cSup	3	2	2	4	

- D(x) = {0,1,2,3}
- D(y) = {0,1,2,3}
- D(z) = [1..7] (interval domain)

sorted

	z	x	y
low	1	0	0
	1	1	0
	1	3	0
	2	1	2
	2	3	2
	4	1	3
	5	3	3
	6	0	3
	8	0	1
	8	2	1
up	8	2	3
	9	0	2
	9	1	1
	9	2	0
	9	2	2
	9	3	1

T[x][y] = z

x \ y	0	1	2	3	rSup
0	1	8	9	6	3
1	1	9	2	4	3
2	9	8	9	8	1
3	1	9	2	5	3
cSup	3	2	2	3	

sorted

	z	x	y
low	1	0	0
	1	1	0
	1	3	0
	2	1	2
	2	3	2
	4	1	3
	5	3	3
	6	0	3
	8	0	1
up	8	2	1
	8	2	3
	9	0	2
	9	1	1
	9	2	0
	9	2	2
	9	3	1

- D(x) = {0,1,2,3}
- D(y) = {0,1,2,3}
- D(z) = [1..7] (interval domain)

T[x][y] = z

y

x

	0	1	2	3	rSup
0	1	8	9	6	3
1	1	9	2	4	3
2	9	8	9	8	0
3	1	9	2	5	3
cSup	3	1	2	3	

- D(x) = {0,1,~~2~~,3}
- D(y) = {0,1,2,3}
- D(z) = [1..7] (interval domain)

sorted

low

up

z	x	y
1	0	0
1	1	0
1	3	0
2	1	2
2	3	2
4	1	3
5	3	3
6	0	3
8	0	1
8	2	1
8	2	3
9	0	2
9	1	1
9	2	0
9	2	2
9	3	1

T[x][y] = z

y

x

	0	1	2	3	rSup
0	1	8	9	6	2
1	1	9	2	4	3
2	9	8	9	8	0
3	1	9	2	5	3
cSup	3	0	2	3	

- D(x) = {0,1,~~2~~,3}
- D(y) = {0,~~1~~,2,3}
- D(z) = [1..6,~~7~~] (interval domain)

sorted

	z	x	y
low	1	0	0
	1	1	0
	1	3	0
	2	1	2
	2	3	2
	4	1	3
	5	3	3
up	6	0	3
	8	0	1
	8	2	1
	8	2	3
	9	0	2
	9	1	1
	9	2	0
	9	2	2
	9	3	1

T[x][y] = z

y

x

	0	1	2	3	rSup
0	**1**	**8**	**9**	**6**	**2**
1	**1**	**9**	**2**	**4**	**3**
2	**9**	**8**	**9**	**8**	**0**
3	**1**	**9**	**2**	**5**	**3**
cSup	**3**	**0**	**2**	**3**	

sorted

low

up

z	x	y
1	0	0
1	1	0
1	3	0
2	1	2
2	3	2
4	1	3
5	3	3
6	0	3
8	0	1
8	2	1
8	2	3
9	0	2
9	1	1
9	2	0
9	2	2
9	3	1

- D(x) = {0,1,3}
- D(y) = {0,2,3}
- D(z) = [2..6] (interval domain)

T[x][y] = z

y

x

	0	1	2	3	rSup
0	1	8	9	6	1
1	1	9	2	4	2
2	9	8	9	8	0
3	1	9	2	5	2
cSup	0	0	2	3	

- D(x) = {0,1,3}
- D(y) = {~~0~~,2,3}
- D(z) = [2..6] (interval domain)

sorted

z	x	y
1	0	0
1	1	0
1	3	0
2	1	2
2	3	2
4	1	3
5	3	3
6	0	3
8	0	1
8	2	1
8	2	3
9	0	2
9	1	1
9	2	0
9	2	2
9	3	1

low

up

T[x][y] = z

y

x

	0	1	2	3	rSup
0	1	8	9	6	1
1	1	9	2	4	2
2	9	8	9	8	0
3	1	9	2	5	2
cSup	0	0	2	3	

sorted

z	x	y
1	0	0
1	1	0
1	3	0
2	1	2
2	3	2
4	1	3
5	3	3
6	0	3
8	0	1
8	2	1
8	2	3
9	0	2
9	1	1
9	2	0
9	2	2
9	3	1

low

up

- D(x) = {0,1,3}
- D(y) = {~~2~~,3}
- D(z) = [2..6] (interval domain)

T[x][y] = z

y

x

	0	1	2	3	rSup
0	1	8	9	6	1
1	1	9	2	4	2
2	9	8	9	8	0
3	1	9	2	5	2
cSup	0	0	2	3	

- D(x) = {0,1,3}
- D(y) = {~~2~~,3}
- D(z) = [2..6] (interval domain)

sorted

low

up

z	x	y
1	0	0
1	1	0
1	3	0
2	1	2
2	3	2
4	1	3
5	3	3
6	0	3
8	0	1
8	2	1
8	2	3
9	0	2
9	1	1
9	2	0
9	2	2
9	3	1

T[x][y] = z

y

x

	0	1	2	3	rSup
0	1	8	9	6	1
1	1	9	2	4	1
2	9	8	9	8	0
3	1	9	2	5	1
cSup	0	0	0	3	

- D(x) = {0,1,3}
- D(y) = {~~2~~,3}
- D(z) = [~~2,3~~,4..6] (interval domain)

sorted

	z	x	y
	1	0	0
	1	1	0
	1	3	0
	2	1	2
	2	3	2
low	4	1	3
	5	3	3
up	6	0	3
	8	0	1
	8	2	1
	8	2	3
	9	0	2
	9	1	1
	9	2	0
	9	2	2
	9	3	1

T[x][y] = z

y

x

	0	1	2	3	rSup
0	1	8	9	6	1
1	1	9	2	4	1
2	9	8	9	8	0
3	1	9	2	5	1
cSup	0	0	0	3	

sorted

z	x	y
1	0	0
1	1	0
1	3	0
2	1	2
2	3	2
4	1	3
5	3	3
6	0	3
8	0	1
8	2	1
8	2	3
9	0	2
9	1	1
9	2	0
9	2	2
9	3	1

low

up

What do we need to restore these values on backtrack?

Implementation 1/2

```
public class Element2D extends Constraint {

  private final int[][] T;
  private final IntVar x, y, z;

  private final ReversibleInt[] nRowsSup,nColsSup ;
  private final ReversibleInt low, up;
  private final ArrayList<Tripple> xyz;

  public void post() throws InconsistencyException {
     . . . // initialize counters nRowsSup, nColsSup
     x.propagateOnDomainChange(this);
     y.propagateOnDomainChange(this);
     z.propagateOnBoundChange(this);
     propagate();
  }

}
```

Implementation 2/2

```
public class Element2D extends Constraint {

  private void updateSupports(int lostPos) throws InconsistencyException {
     if (nColsSup[xyz.get(lostPos).x].decrement() == 0) {
         x.remove(xyz.get(lostPos).x);
     }
     if (nRowsSup[xyz.get(lostPos).y].decrement() == 0) {
         y.remove(xyz.get(lostPos).y);
     }
  }
  public void propagate() throws InconsistencyException {
     int l = low.getValue();
     int u = up.getValue();
     int zMin = z.getMin();
     while (xyz.get(l).z < zMin ||
            !x.contains(xyz.get(l).x) ||
            !y.contains(xyz.get(l).y)) {
         updateSupports(l);
         l++;
         if (l > u) throw new InconsistencyException();
     }
     z.removeBelow(xyz.get(l).z);
     low.setValue(l);
     . . . // do something similar for updating u
  }
}
```

We decrement the support counters as we were only removing values, the trail will take care to restore everything

Set the low value to the first consistent entry in our table. Trail will restore it on backtrack

Quadratic Assignment

```
Solver cp = makeSolver();
IntVar[] x = makeIntVarArray(cp, n, n);

cp.post(allDifferent(x));



// build the objective function
IntVar[] weightedDist = new IntVar[n*n];
int ind = 0;
for (int i = 0; i < n; i++) {
    for (int j = 0; j < n; j++) {
        weightedDist[ind] = mul(element(d,x[i],x[j]),w[i][j]);
        ind++;
    }
}
IntVar objective

DFSearch dfs = makeDfs(cp,firstFail(x));

cp.post(minimize(objective,dfs));
```

Any idea how CP is able to minimize?

Minimization with CP with a special constraint

```
public class Minimize extends Constraint {

    public int bound = Integer.MAX_VALUE;
    public final IntVar x;
    public final DFSearch dfs;

    public Minimize(IntVar x, DFSearch dfs) {. . . }

    protected void tighten() {
        if (!x.isBound())
          throw new RuntimeException("objective not bound");
        this.bound = x.getMax() - 1;
    }

    public void post() throws InconsistencyException {
        x.whenBoundsChange(() -> x.removeAbove(bound));
        // Ensure that the constraint is scheduled on backtrack
        dfs.onSolution(() -> {
            tighten();
            cp.schedule(this);
        });
        dfs.onFail(() -> cp.schedule(this));
    }
}
```

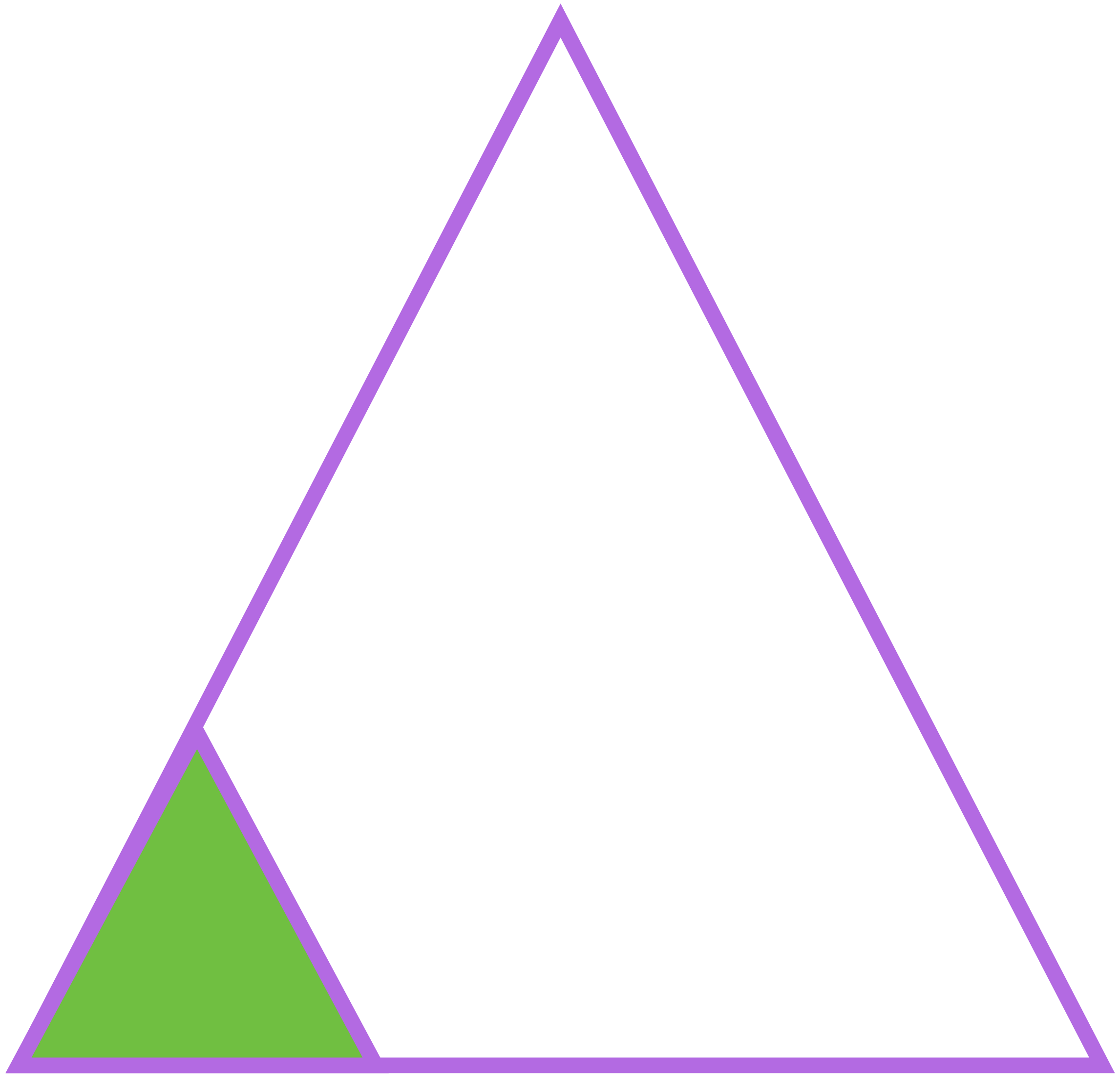
B&B Constraint

Tighten objective on each solution found
(this is why we need dfs)

Don't forget to schedule it in the
propagation queue on backtrack

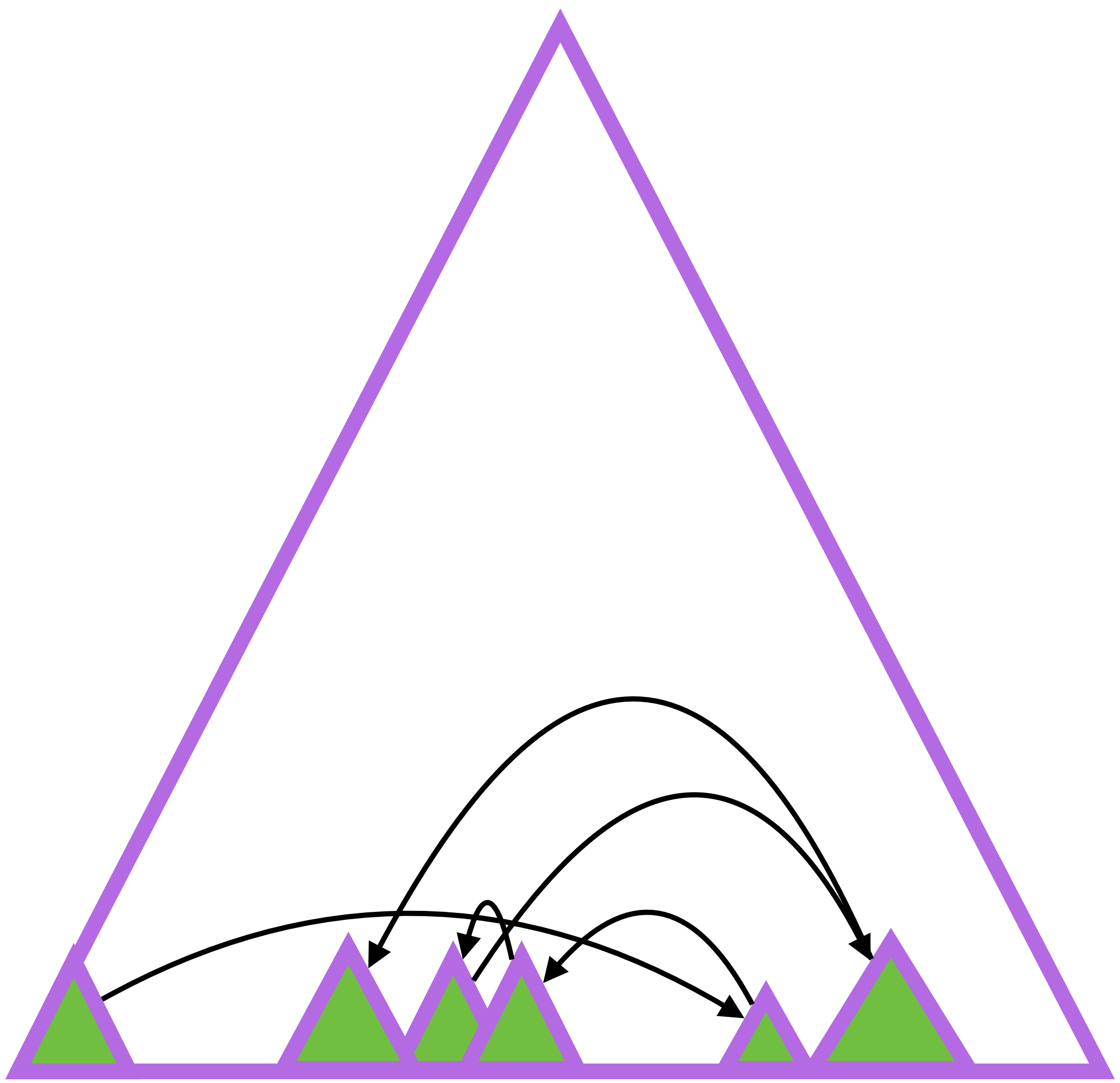
The weakness of CP

- Huge search tree
- Very poor exploration of the search space



How to fix this?

- When you get stuck for too-long not improving, restart at another place
- Intensify the search were it looks promising



Large Neighborhood Search (LNS)

- LNS = Fix + Relax + Restart

1. Find a first initial solution, S*
2. Randomly relax S* and re-optimize with search limit
   – Relax = fix some variables to their values in S*
3. Replace S* by the best solution found

It can be more general than that, for instance in scheduling relax = keep some of the precedences from best solution

Advantages over pure LS

- The neighborhood is large
  - no need for meta-heuristic to avoid local minima
- Modeling power of CP (declarative),
  - no need for designing complex neighborhood
  - ease of implementation
- Scalability of LS
  - very good « any-time » behavior

LNS on top of our QAP model

```
// Current best solution
int[] xBest = new int[n];
for (int i = 0; i < n; i++) {
    xBest[i] = i;
}

dfs.onSolution(() -> {
    // Update the current best solution
    for (int i = 0; i < n; i++) {
        xBest[i] = x[i].getMin();
    }
});
int nRestarts = 1000;
int failureLimit = 50;
Random rand = new java.util.Random(0);
for (int i = 0; i < nRestarts; i++) {
    // Record the state such that the fragment constraints can be cancelled
    cp.push();
    // Assign the fragment 50% of the variables randomly chosen
    for (int j = 0; j < n; j++) {
        if (rand.nextInt(100) < 50) {
            equal(x[j],xBest[j]);
        }
    }
    dfs.start(statistics -> statistics.nFailures >= failureLimit);
    // cancel all the fragment constraints
    cp.pop();
}
```

simple initial assignment (could be random)

update current best solution whenever one is found

fix randomly 50% of the variables to their value in the current best solution

start a DFS search but give it a maximum number of failure credit (not too long)

Do-It-Your-Self

- Monday:
  - Setup Mini-CP, Quick introduction
  - Implement: LessOrEqual constraint
- Tuesday
  - Introduction to Mini-CP Architecture: Trail, Reversible, Search
  - **Implement Element1D**
  - **Implement DFS with explicit stack**
- Wednesday:
  - Implement Circuit Constraint
  - Implement Custom Search for TSP
  - Implement and experiment LNS for TSP
- Friday
  - Decomposing Cumulative Constraint
  - Implement Time Table Filtering for Cumulative
- Optional (if you are supper fast)
  - Table Constraints, AllDifferent, Or (with watched literals)
  - Discrepancy search

Do it your -self

- Implement DFS with explicit stack of alternative (instead of recursion)

- Implement Element1D

Mini-Solver Competition

www.xcsp.org

Feel free to use MiniCP as a starting point

Mini-Solver Tracks

less than 8,000 lines discarding code for parsing XCSP3, comments and code of standard libraries).

Take away message

- Want to improve your CP knowledge
  - implement your own solver (MiniCP is a good starting point but don't hesitate to change or adopt a different design, domain implem, etc)
  - implement a few constraints (table, sum, element, etc)
    - the largest and most difficult code-base in a solver are the constraints!
    - try to design incremental filtering (you can already do a lot with reversible integers)
  - Implement a few black-box and LNS searches
  - Solve and model many problems

MiniCP

A Minimalistic Educational Solver

Laurent Michel, Pierre Schaus, Pascal Van Hentenryck

website: https://www.info.ucl.ac.be/~pschaus/minicp

code: https://bitbucket.org/pschaus/minicp

slides http://tinyurl.com/y8n4knhx