Constraint Programming

Introduction to MiniCP Architecture

Purpose of this lecture

- Domains
- Variables
- Constraints
- Propagation: Fixpoint Algorithm

Domain

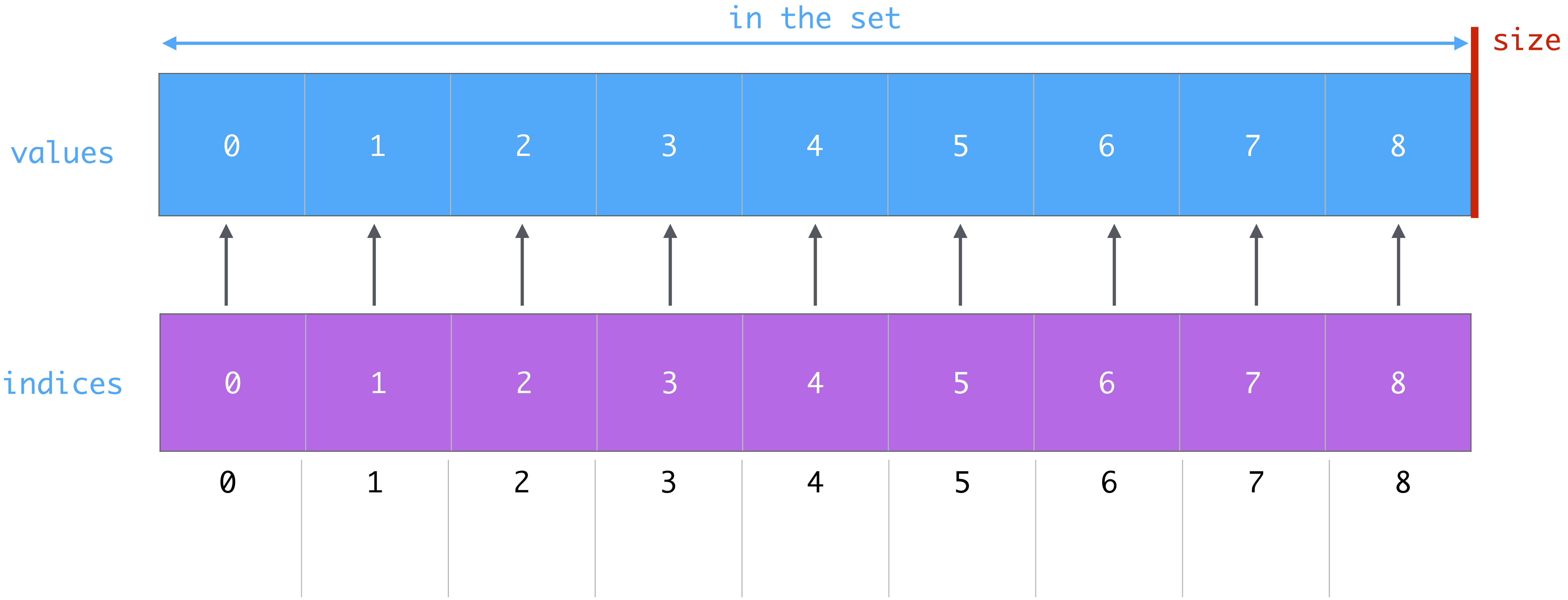
- What is a domain?
  - A finite set of integers that are the still possible values for a variable.
  - A simple API in order to:
    - Query the set.
    - Remove values from the set.
    - We never add a value.
- Questions:
  - Representation (Data Structure)?
  - API?

Representation

- Some possible choices:
  - A sparse set embedded in an array. ✅
  - A bit vector.
  - A tree set (like red-black trees).
  - A range list.
- Simplification [for the lecture]:
  - We assume that the set has a lower bound of 0.

Sparse Set: Visually

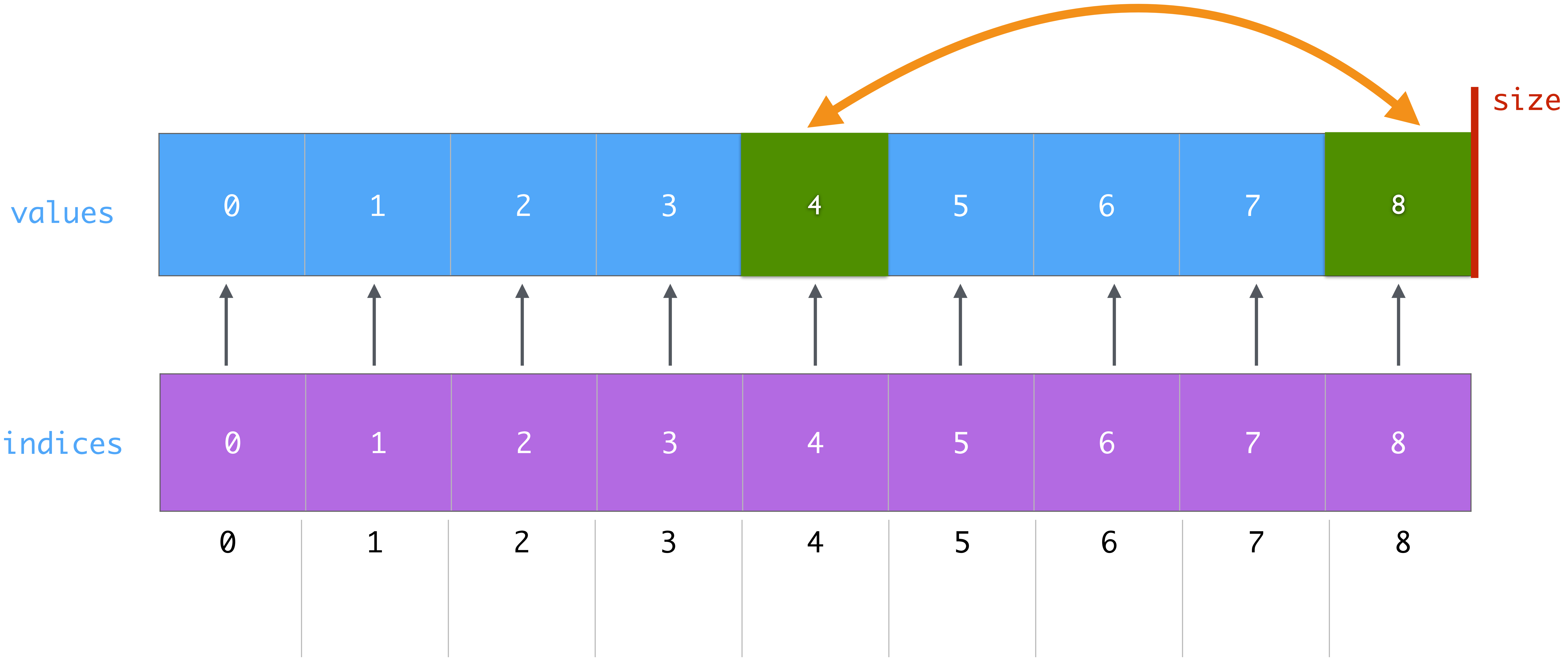
- Initialization of the set {0,1,2,3,4,5,6,7,8} for $n = 9$ (size)



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

Sparse Set: Visually

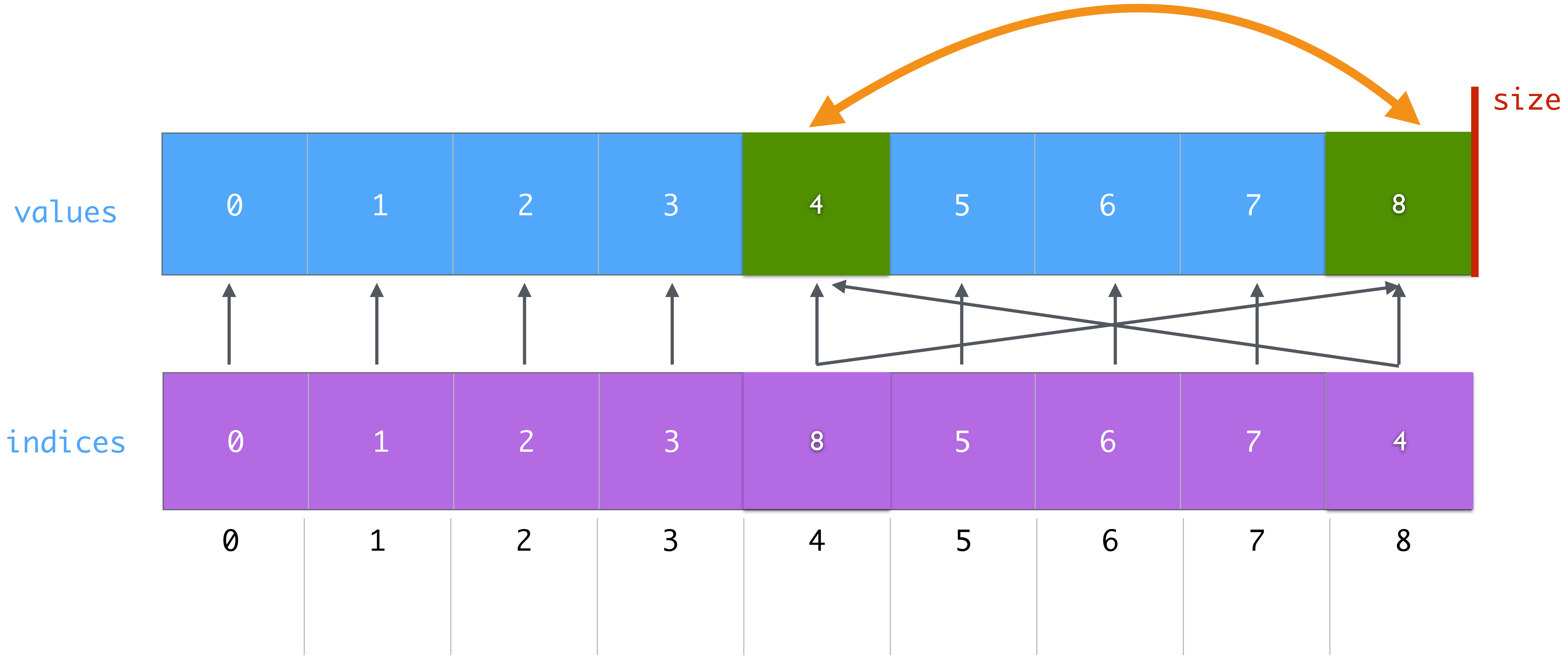
- Remove 4 from {0,1,2,3,4,5,6,7,8}



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

Sparse Set: Visually

- Remove 4 from {0,1,2,3,4,5,6,7,8}

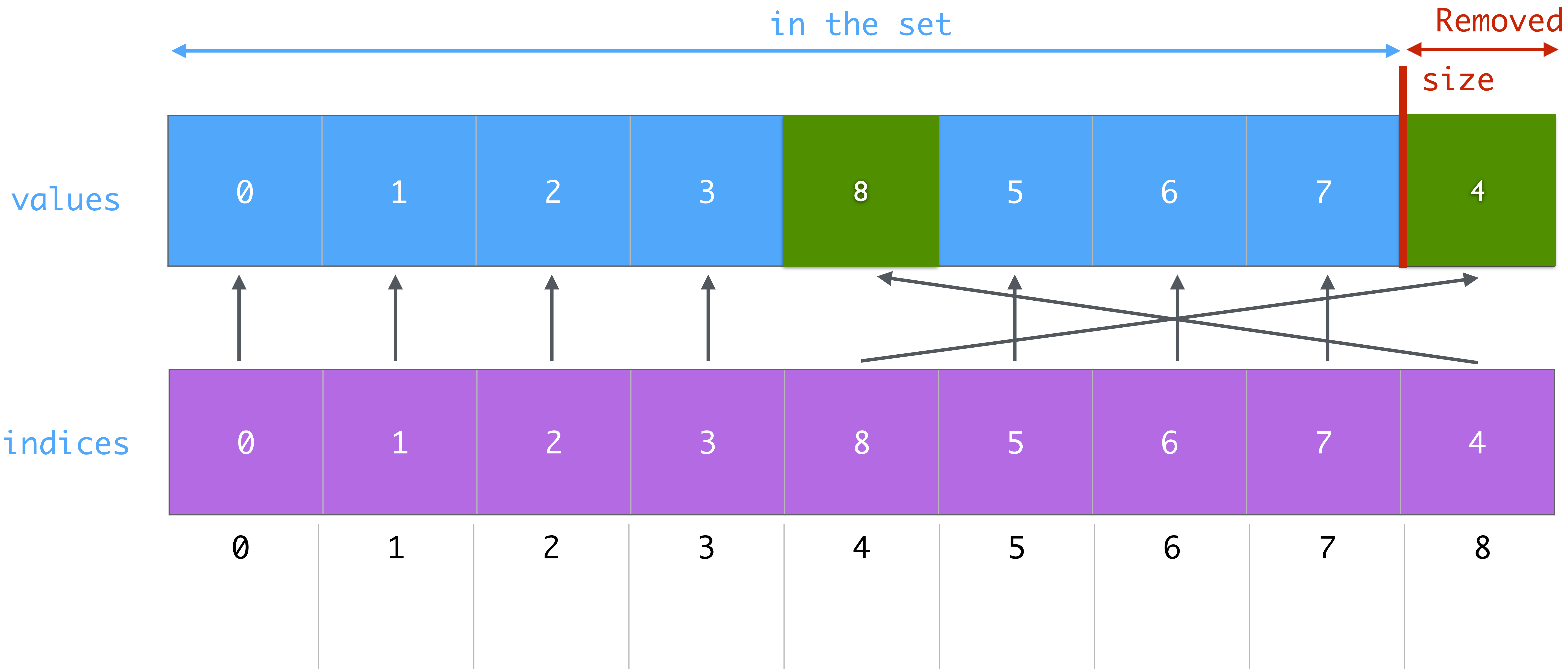


$$\forall v \in \{0..n-1\} \ : \ values[indices[v]] = v$$

Sparse Set: Visually

- Removal of 4 results in {0,1,2,3,8,5,6,7}

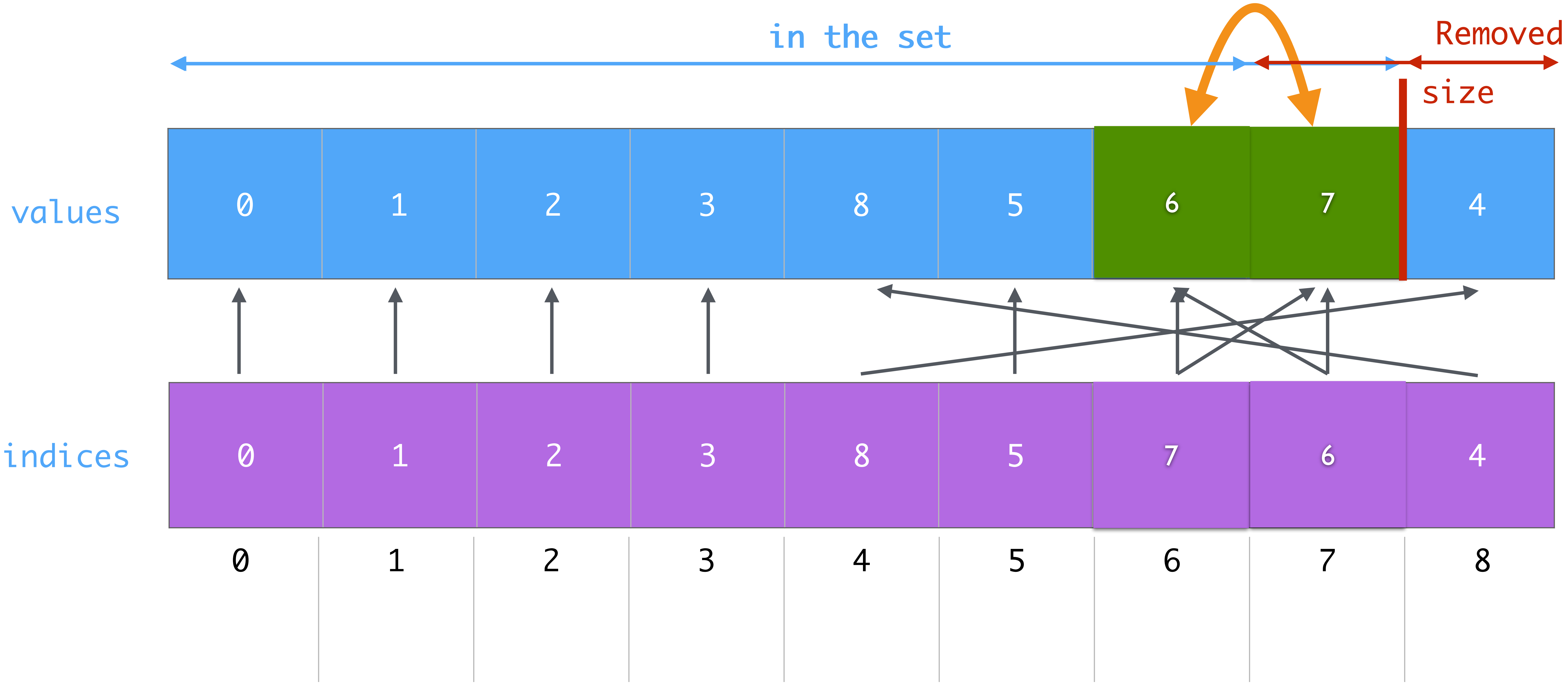
Runtime: $\Theta(1)$



$$\forall v \in \{0..n-1\} \; : \; values[indices[v]] = v$$

Sparse Set: Visually

- Remove 6 from {0,1,2,3,8,5,6,7}

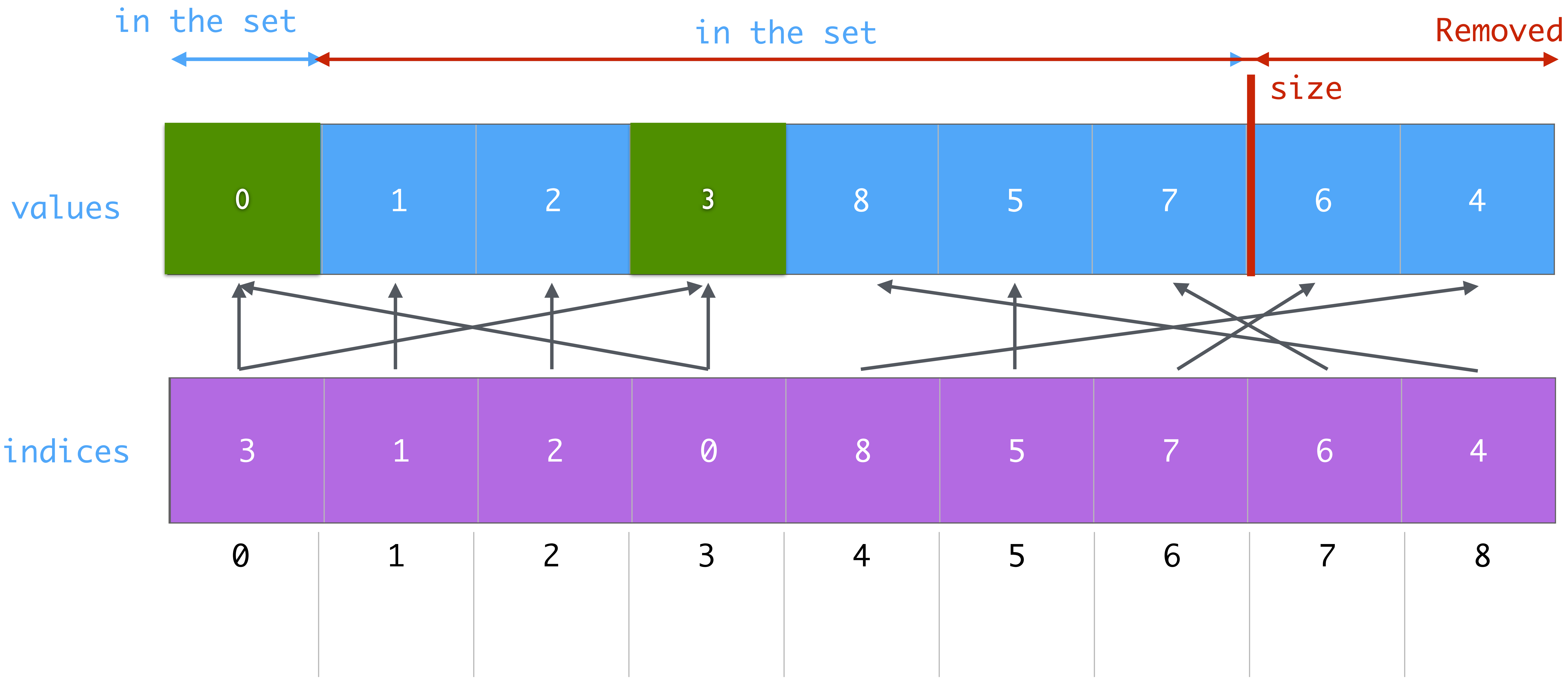


$$\forall v \in \{0..n-1\} \;:\; values[indices[v]] = v$$

Sparse Set: Visually

- Remove everything but value 3 from {0,1,2,3,8,5,7}

Runtime: $\Theta(1)$



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

Complexity of Other Operations

- min(D)?
- max(D)?
- size(D)?
- contains(D,v)?

Data Structure in Java

```
package minicp.state;
import java.util.NoSuchElementException;

public class StateSparseSet {
    private int[] values,indices;
    private StateInt size, min, max;
    private int ofs, n;
    public StateSparseSet(StateManager sm, int n, int ofs) {
        this.n = n;
        this.ofs = ofs;
        size = sm.makeStateInt(n);
        min = sm.makeStateInt(0);
        max = sm.makeStateInt(n - 1);
        values = new int[n];
        indices = new int[n];
        for (int i = 0; i < n; i++) {
            values[i] = i;
            indices[i] = i;
        }
    }
  …
}
```

StateInt

```
int
int setValue(int)
int value()
```

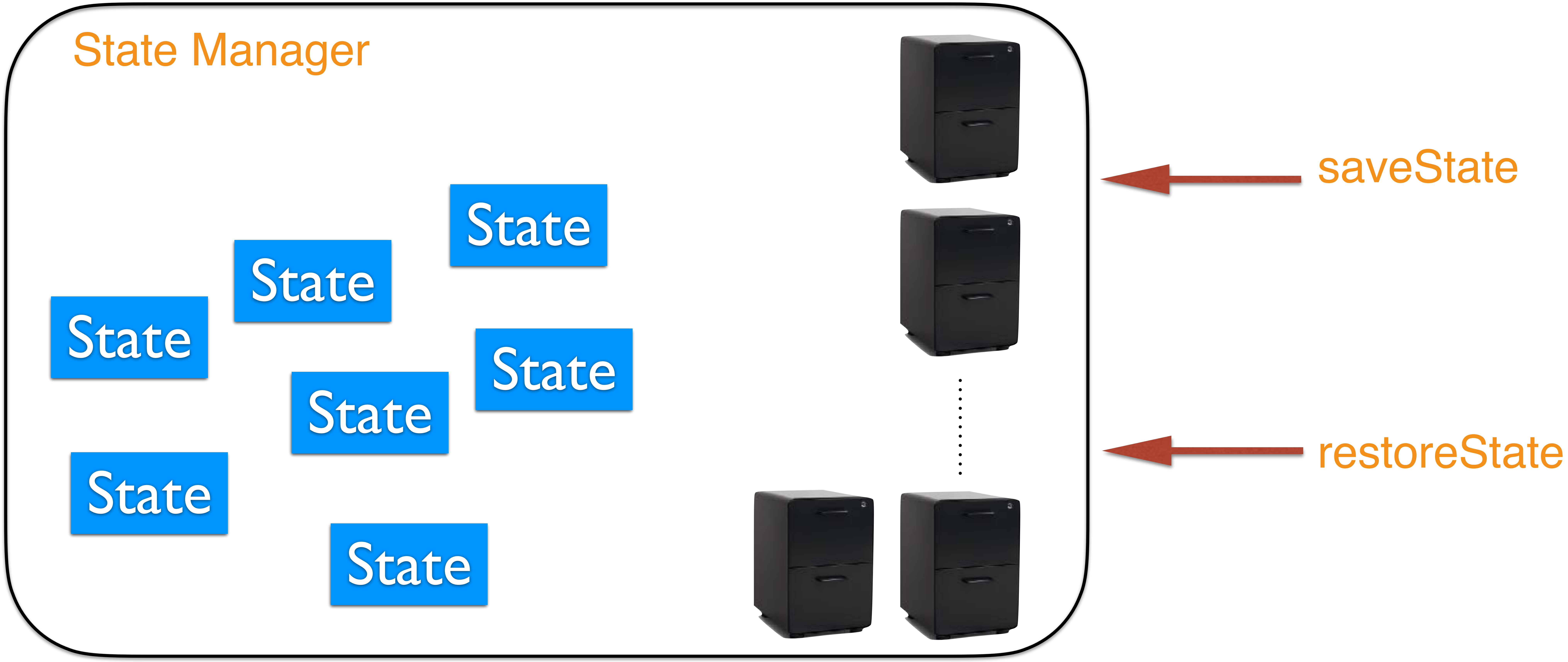
Factory

```
StateInt makeStateInt(int)
void saveState()
void restoreState()
…
```

State Factory

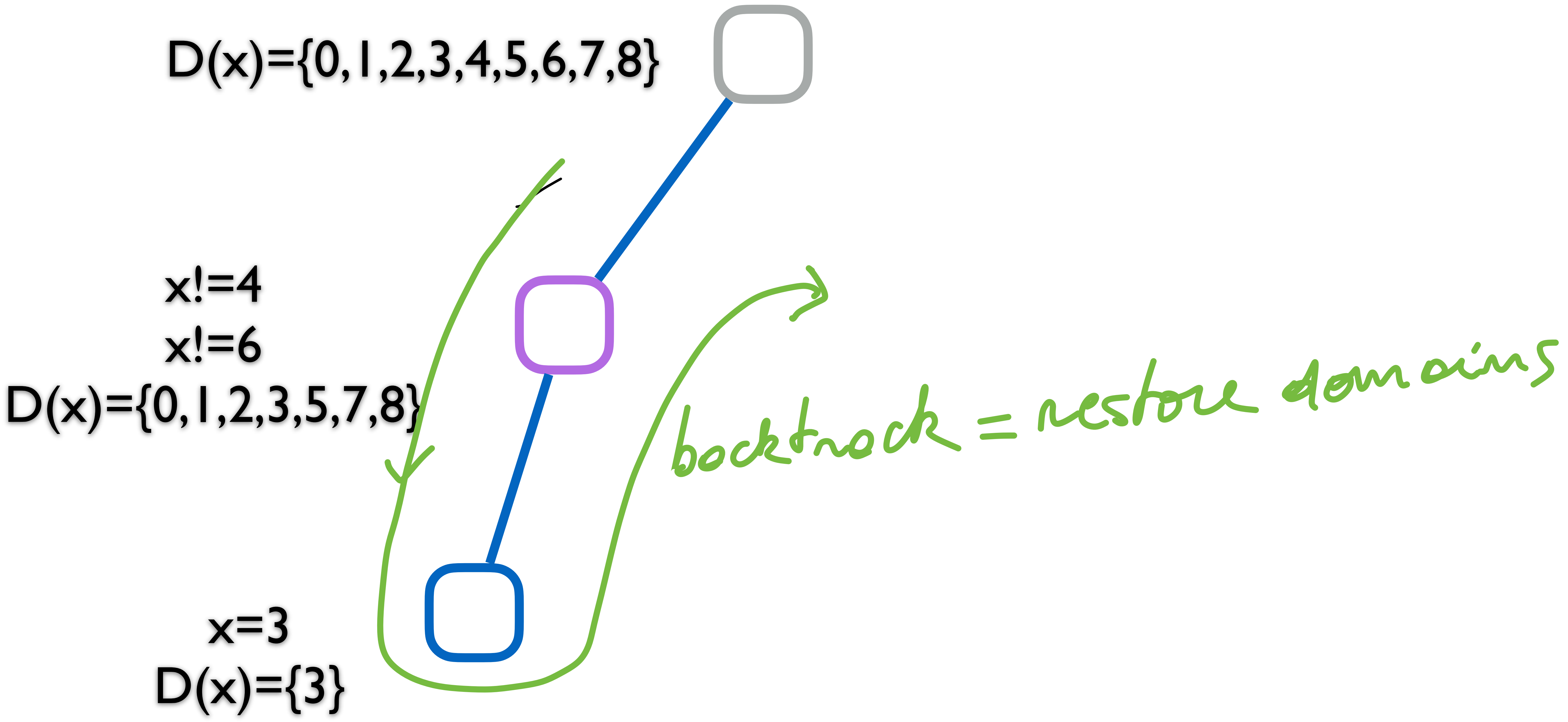
– It is a container of stateful abstraction.

– It can be backed up (saveState) and restored (restoreState).

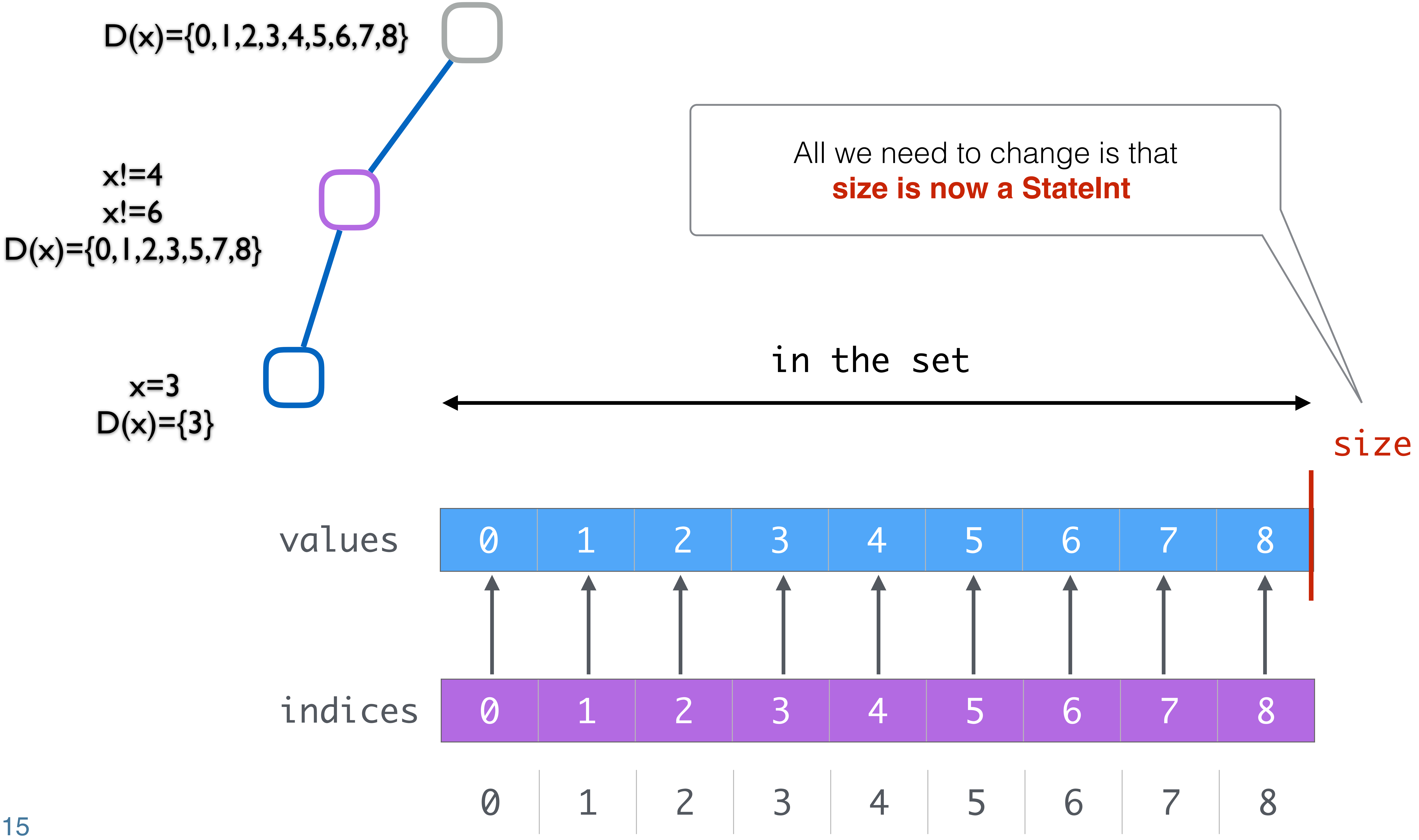


Why Do We Need Stateful Abstraction?

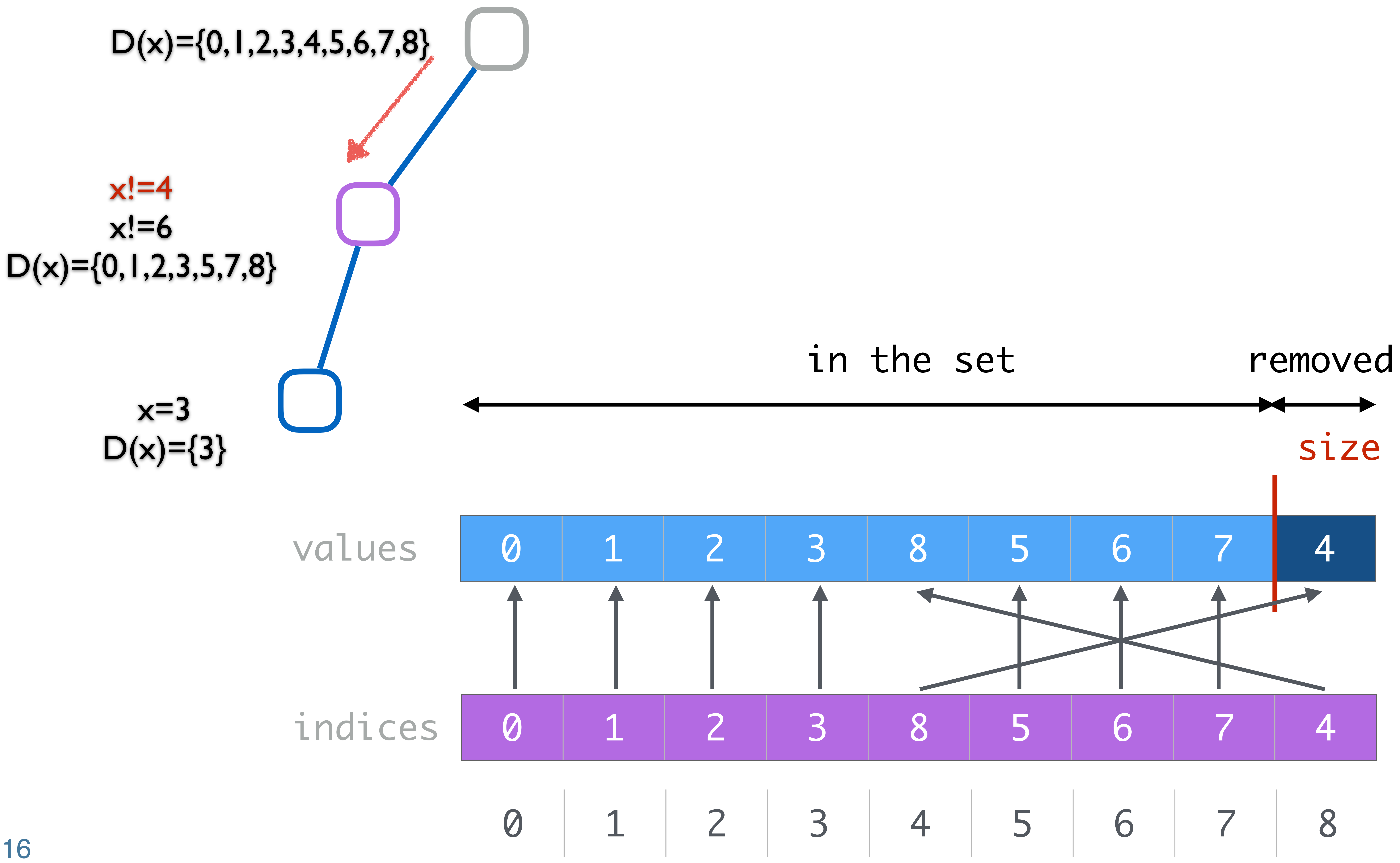
- To restore the state on backtrack.
- Consider this scenario:



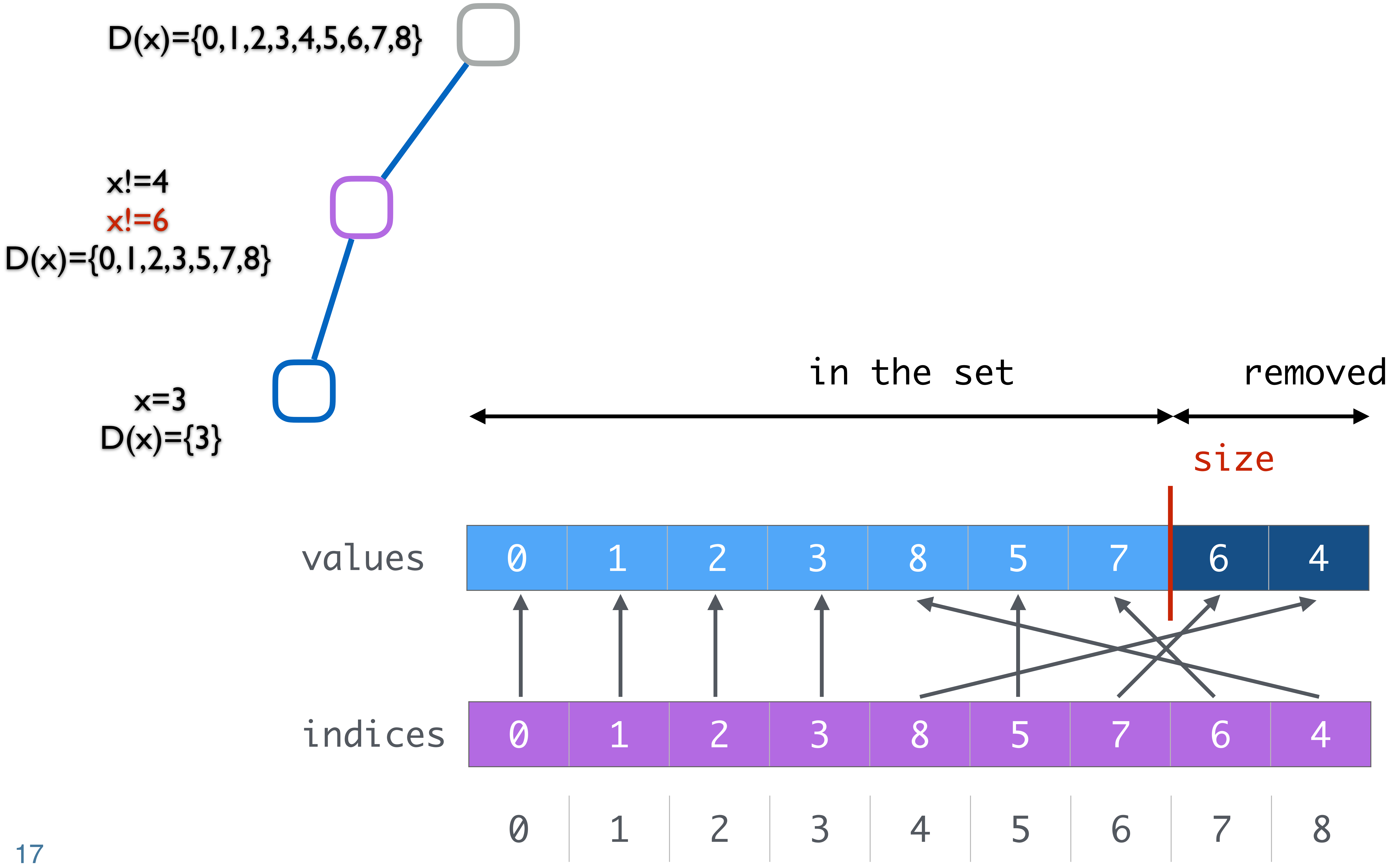
Stateful Sparse Set



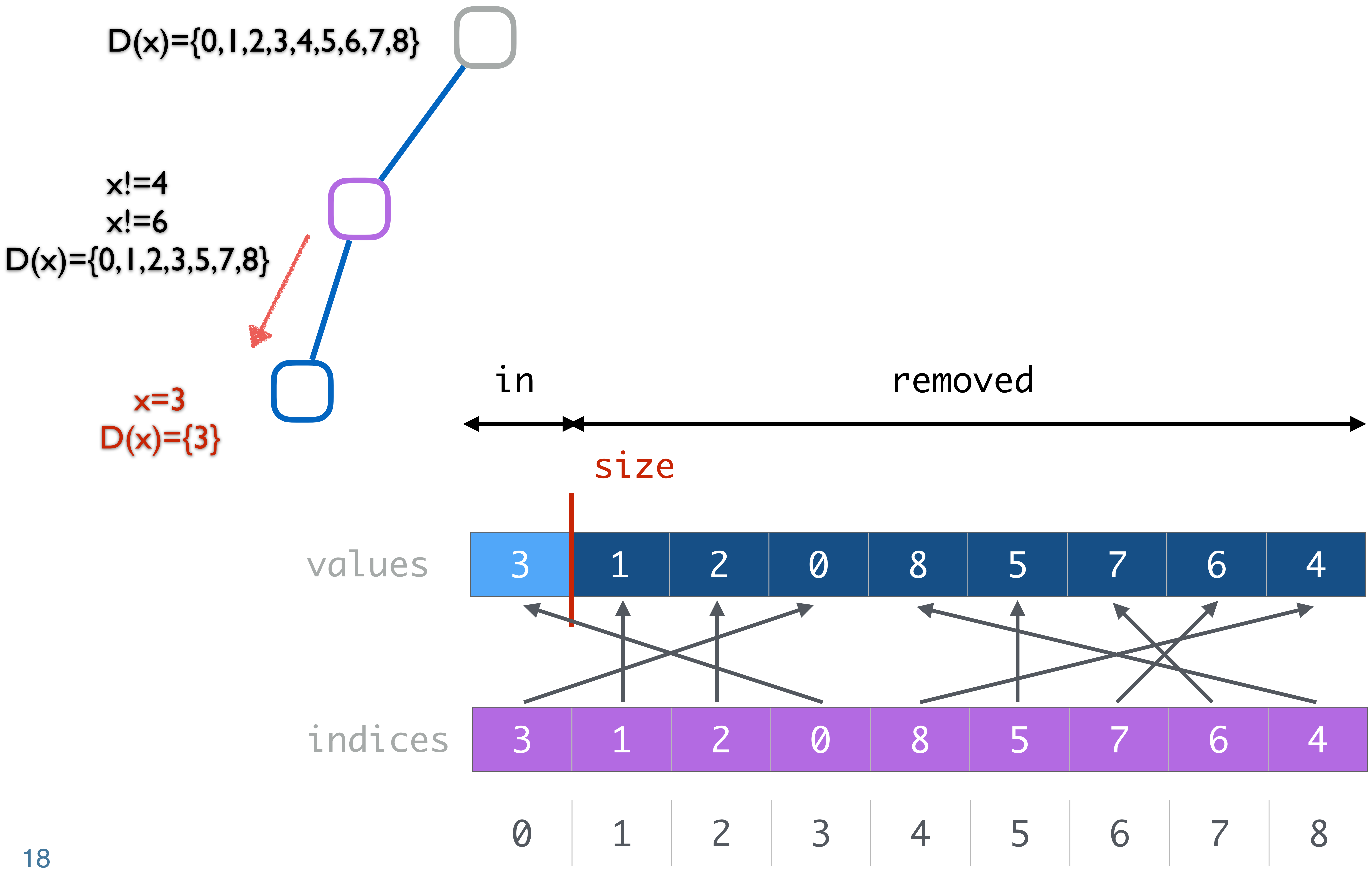
Removal Operations



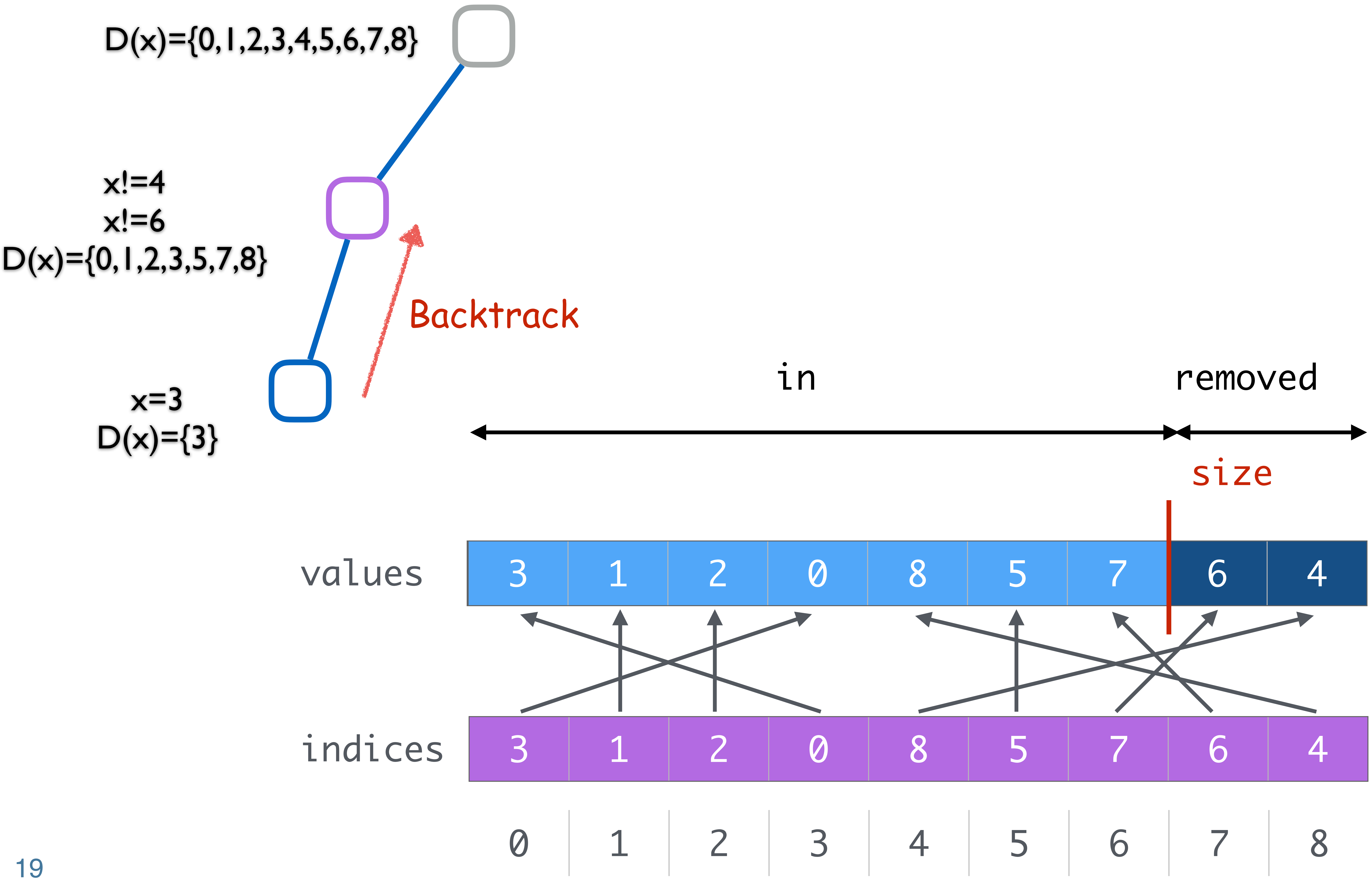
Removal Operations



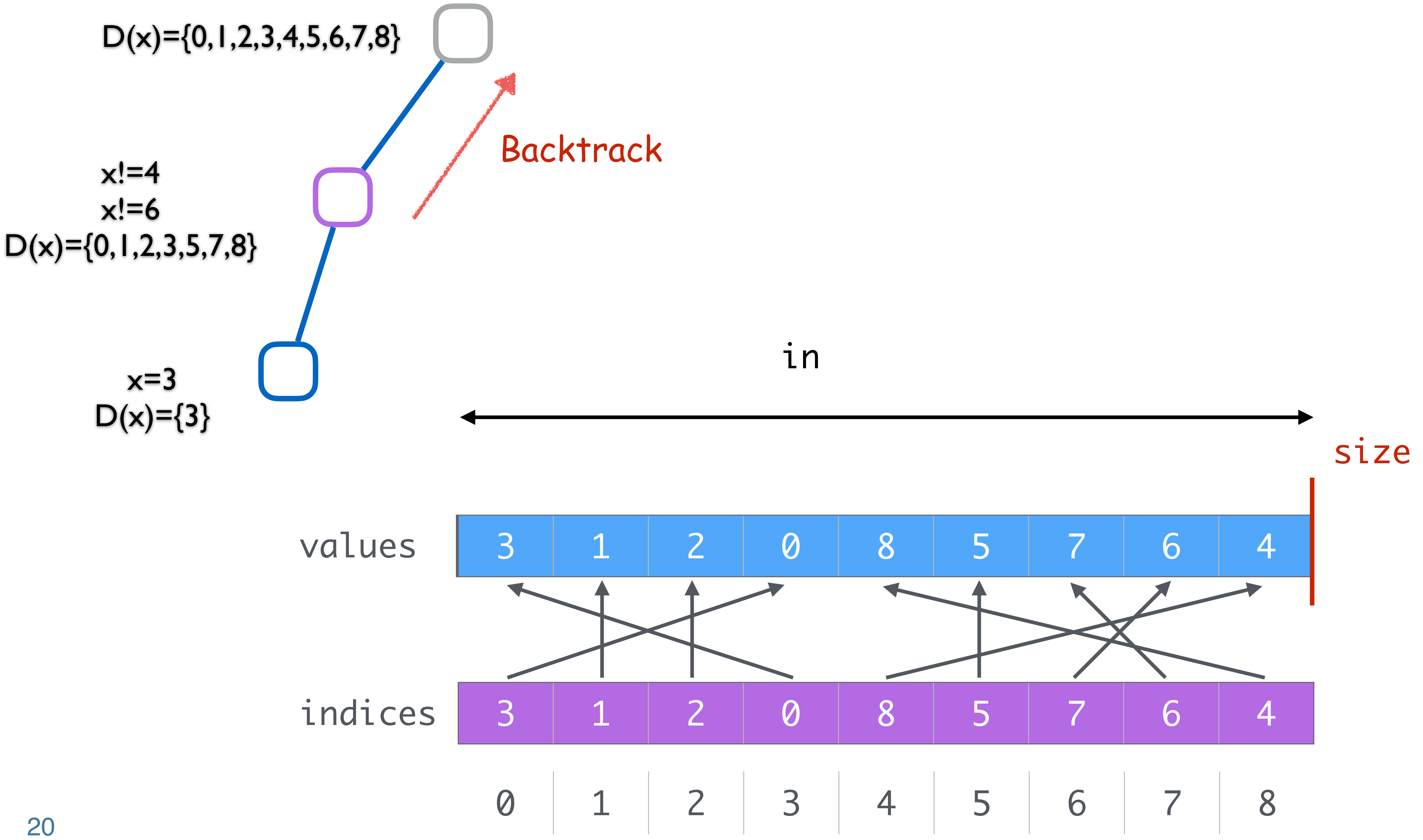
Removal Operations



Removal Operations



Removal Operations



Generalizing (the domains don't need to start at 0)

```
package minicp.state;
import java.util.NoSuchElementException;

public class StateSparseSet {
    private int[] values, indices;
    private StateInt size, min, max;
    private int ofs;
    private int n;
    public StateSparseSet(StateManager sm, int n, int ofs)
        this.n = n;
        this.ofs = ofs;
        size = sm.makeStateInt(n);
        min = sm.makeStateInt(0);
        max = sm.makeStateInt(n - 1);
        values = new int[n];
        indices = new int[n];
        for (int i = 0; i < n; i++) {
            values[i] = i;
            indices[i] = i;
        }
    }
  …
}
```

Add or subtract an offset to every operation!

```
public int max() {
    max.value() + ofs;
}
```

StateSparseSet in Java (abridged)

```java
public class StateSparseSet {
    // internal state
    void exchPositions(
                int val1,int val2) {
        int v1 = val1;
        int v2 = val2;
        int i1 = indices[v1];
        int i2 = indices[v2];
        values[i1] = v2;
        values[i2] = v1;
        indices[v1] = i2;
        indices[v2] = i1;
    }
    boolean isEmpty(){
        return size.value()==0; }
    int size(){ return size.value(); }
    boolean contains(int val) {
        val -= ofs;
        if (val < 0 || val >= n)
            return false;
        else
            return indices[val] < size();
    }

    boolean remove(int val) {
        if (!contains(val)) return false;
        val -= ofs;
        int s = size();
        exchPositions(val, values[s - 1]);
        size.decrement();
        updateBoundsValRemoved(val);
        return true;
    }

    void removeAll(){size.setValue(0);}

    void removeBelow(int value) {
        if (max() < value) {
            removeAll();
        } else
          for (int v = min(); v < value; v++)
              remove(v);
    }
}
```

From StateSparseSet to IntDomain ADT

- Goal:
  - Build a domain ADT (abstract datatype) with a StateSparseSet.

```
package minicp.engine.core;

public interface IntDomain {
     int min();
     int max();
     int size();
     boolean contains(int v);
     boolean isSingleton();
     void remove(int v, DomainListener l);
     void removeAllBut(int v, DomainListener l);
     void removeBelow(int v, DomainListener l);
     void removeAbove(int v, DomainListener l);
     String toString();
}
```

IntDomain ADT Implementation

```
package minicp.engine.core;
public interface IntDomain {
    int min();
    int max();
    int size();
    boolean contains(int v);
    boolean isSingleton();
    void remove(int v, DomainListener l);
    void removeAllBut(int v, DomainListener l);
    void removeBelow(int v, DomainListener l);
    void removeAbove(int v, DomainListener l);
    String toString();
}
```

D(X)

```
package minicp.engine.core;
public interface DomainListener {
    void empty();
    void fix();
    void change();
    void changeMin();
    void changeMax();
}
```

IntDomain ADT Implementation

```
package minicp.engine.core;
public class SparseSetDomain implements IntDomain {
    private StateSparseSet domain;
    public SparseSetDomain(StateManager sm, int min, int max){
        domain = new StateSparseSet(sm, max - min + 1, min);
    }
    public int min() { return domain.min();}
    public int size(){ return domain.size();}
    public boolean contains(int v) {
                return domain.contains(v);}
    public boolean isSingleton(){ return domain.size() == 1;}
    public void remove(int v, DomainListener l) {
        if (domain.contains(v)) {
            boolean maxChanged = max() == v;
            boolean minChanged = min() == v;
            domain.remove(v);
            if (domain.size() == 0)l.empty();
            l.change();
            if (maxChanged) l.changeMax();
            if (minChanged) l.changeMin();
            if (domain.size() == 1) l.fix();
        }
    }
    …
}
```

DomainListener l

Purpose

- Basics of implementation:
  - Domains
  - Variables
  - Constraints
  - Propagation: Fixpoint Algorithm

Variables… in a nutshell!

- Variable:
  - State:
    - Recall the solver that created the variable.
    - Encapsulate the domain of the variable.
    - Track the constraints that mention the variable.
  - API:
    - Domain queries (getters = accessors).
    - Domain updates (contraction, fixing).
    - Hookups.

First and foremost:
what is the ADT?

Variable ADT

```
package minicp.engine.core;
import minicp.util.Procedure;

public interface IntVar {
    Solver getSolver();
    int min();
    int max();
    int size();
    boolean isFixed();
    boolean contains(int v);

    void remove(int v);
    void fix(int v);
    void removeBelow(int v);
    void removeAbove(int v);

    void whenFix(Procedure f);
    void whenBoundChange(Procedure f);
    void whenDomainChange(Procedure f);
    void propagateOnDomainChange(Constraint c);
    void propagateOnFix(Constraint c);
    void propagateOnBoundChange(Constraint c);
}
```

Queries

Updates

Hookups

Variable Representation

- Instance variables:
  - The solver.
  - The domain (as abstract reference).
  - Three stacks of constraints:
    - Holding references to constraints that mention this variable.
    - Devoted to specific ***events***
      - Domain values were lost.
      - Domain became a singleton.
      - Min or Max was changed

```
public class IntVarImpl implements IntVar {
    private Solver cp;
    private IntDomain domain;
    private StateStack<Constraint> onDomain;
    private StateStack<Constraint> onFix;
    private StateStack<Constraint> onBound;
…
```

Constructor

```
public IntVarImpl(Solver cp, int min, int max) {
     this.cp = cp;
     domain = new SparseSetDomain(cp.getStateManager(), min, max);
     onDomain = new StateStack<>(cp.getStateManager());
     onFix    = new StateStack<>(cp.getStateManager());
     onBound  = new StateStack<>(cp.getStateManager());
 }
```

$D(x) = \{min..max\}$

Part of the state too!

Queries

- Everything is delegated to the domain (or straightforward):

```
@Override public Solver getSolver() { return cp;}
@Override public int min()          { return domain.min();}
@Override public int max()          { return domain.max();}
@Override public int size()         { return domain.size();}
@Override public boolean contains(int v) { return domain.contains(v);}
@Override public boolean isFixed()       { return domain.isSingleton();}
```

Hookups

- Purposes:
  - Associate constraints to variable x of a CSP $< X,D,C >$:

$$cstr(x) = \{c \in C \mid x \in Vars(c)\}$$

  - Associate constraints to events on the domain of x:

$$onDomain(x) = \{c \in C \mid x \in Vars(c) \wedge \mathscr{F}_c \text{ calls } x.\text{propagateOnDomainChange}(c)\}$$

$$onFix(x) = \{c \in C \mid x \in Vars(c) \wedge \mathscr{F}_c \text{ calls } x.\text{propagateOnFix}(c)\}$$

$$onBound(x) = \{c \in C \mid x \in Vars(c) \wedge \mathscr{F}_c \text{ calls } x.\text{propagateOnBoundChange}(c)\}$$

  - When an event occurs, the variable triggers the “waking up” of those constraints, aka their *scheduling* or their *enqueuing* in the fixPoint algorithm.

Hookups

```
@Override
public void propagateOnDomainChange(Constraint c) {
    onDomain.push(c);
}
@Override
public void propagateOnFix(Constraint c) {
    onFix.push(c);
}
@Override
public void propagateOnBoundChange(Constraint c) {
    onBound.push(c);
}
```

Add to the stack!

Tying it together…

```
public class IntVarImpl implements IntVar {
    private Solver cp;
    private IntDomain domain;
    private StateStack<Constraint> onDomain;
    private StateStack<Constraint> onFix;
    private StateStack<Constraint> onBound;

    private DomainListener domListener = new DomainListener() {
        public void empty() { throw InconsistencyException.INCONSISTENCY; }
        public void fix()       { scheduleAll(onFix); }
        public void change()    { scheduleAll(onDomain); }
        public void changeMin() { scheduleAll(onBound); }
        public void changeMax() { scheduleAll(onBound); }
    };
    protected void scheduleAll(StateStack<Constraint> constraints) {
        for (int i = 0; i < constraints.size(); i++)
            cp.schedule(constraints.get(i));
    }
```

A “listener” object embedded in the variable!

Purpose

- – Domains
- – Variables
- – Constraints
- – Propagation: Fixpoint Algorithm

Constraints… in a nutshell!

- What we really have:
  - A *propagator* implements a filtering algorithm for a constraint:

$$c(x_0, \ldots, x_{n-1}) \Rightarrow \mathcal{F}_c(\langle D_0, \ldots, D_{n-1} \rangle)$$

- API:
  - **post** the constraint:
    - state the constraint and hook up its variables.
  - **propagate** the constraint:
    - remove values from the domains of *Vars*(*c*) that are not in any solutions.

Constraint ADT

```
package minicp.engine.core;

public interface Constraint {
    void post();
    void propagate();                                  Core API


    void setScheduled(boolean scheduled);
    boolean isScheduled();
    void setActive(boolean active);
    boolean isActive();                Performance API (later!)
}
```

Implementation

MiniCP

```
public abstract class AbstractConstraint implements Constraint {
    private final Solver cp;
    private boolean scheduled = false;
    private final State<Boolean> active;
    public AbstractConstraint(Solver cp) {
        this.cp = cp;
        active = cp.getStateManager().makeStateRef(true);
    }
    public Solver getSolver() { return cp;}

    public void post()          {}
    public void propagate()     {}

    public void setScheduled(boolean scheduled)  { this.scheduled = scheduled;}
    public boolean isScheduled()                 { return scheduled;}
    public void setActive(boolean active)        { this.active.setValue(active);}
    public boolean isActive()                    { return active.value();}
}
```

Perf. API

The $x \neq y + c$ Constraint

```
public class NotEqual extends AbstractConstraint {
    private final IntVar x, y;
    private final int c;
    public NotEqual(IntVar x, IntVar y, int c) {
        super(x.getSolver());
        this.x = x; this.y = y; this.c = c;
    }
    @Override public void post() {
        if (y.isFixed()) x.remove(y.min() + c);
        else if (x.isFixed()) y.remove(x.min() - c);
        else {
            x.propagateOnFix(this);
            y.propagateOnFix(this);
        }
    }
    @Override public void propagate() {
        if (y.isFixed()) x.remove(y.min() + c);
        else y.remove(x.min() - c);
        setActive(false);
    }
}
```

The x ≠ y + c Constraint

```
@Override public void post() {
    if (y.isFixed())
       x.remove(y.min() + c);
    else if (x.isFixed())
       y.remove(x.min() - c);
    else {
       x.propagateOnFix(this);
       y.propagateOnFix(this);
    }
}
@Override public void propagate() {
    if (y.isFixed())
       x.remove(y.min() + c);
    else y.remove(x.min() - c);
    setActive(false);
}
```

Initial inference

Hook up the constraint

Respond to fixing event

$$|D(y)| = 1 \Rightarrow \mathcal{F}_c(D)(x) = D(x) \setminus \{\min(D(y)) + c\}$$

$$|D(x)| = 1 \Rightarrow \mathcal{F}_c(D)(y) = D(y) \setminus \{\min(D(x)) - c\}$$

Creating Variables & Constraints

- Factory Design Pattern (by “Gang of Four”):
  - Collect all the “creational” behavior in one place:

```
public final class Factory {
    private Factory() { throw new UnsupportedOperationException();}
    public static Solver makeSolver() …
    public static IntVar makeIntVar(Solver cp, int sz)          {
         return new IntVarImpl(cp, sz);}
    public static IntVar makeIntVar(Solver cp, int min, int max) {
         return new IntVarImpl(cp, min, max);}
    // -------------- constraints -----------------------
    public static Constraint equal(IntVar x, int v) {
        return new AbstractConstraint(x.getSolver()) {
            @Override
            public void post() { x.fix(v);}
        };
    }
    public static Constraint notEqual(IntVar x, IntVar y, int c) {
         return new NotEqual(x, y, c);}

    …
}
```

Purpose

- Domains
- Variables
- Constraints
- Propagation: Fixpoint Algorithm

Goal

- Orchestrate the whole process:
  - from adding constraints
  - to computing the fixpoint.

Solver ADT

```
package minicp.engine.core;


public interface Solver {
    StateManager getStateManager();

    void post(Constraint c);
    void post(Constraint c, boolean enforceFixPoint);
    Objective minimize(IntVar x);
    Objective maximize(IntVar x);
```

Model construction

```
    void schedule(Constraint c);
    void fixPoint();
```

Propagation API

```
    void onFixPoint(Procedure listener);
}
```

MiniCP Implementation

```
package minicp.engine.core;

import …;

public class MiniCP implements Solver {
    private Queue<Constraint> propagationQueue = new ArrayDeque<>();
    private final StateManager sm;

    public MiniCP(StateManager sm)                  { this.sm = sm;}
    @Override public StateManager getStateManager() { return sm;}

    @Override public void post(Constraint c)      { post(c, true);}
    @Override public void post(Constraint c, boolean enforceFixPoint) {
        c.post();
        if (enforceFixPoint) fixPoint();
    }
    @Override public Objective minimize(IntVar x) { return new Minimize(x);}
    @Override public Objective maximize(IntVar x) { return minimize(Factory.minus(x));}
    …
}
```

MiniCP Implementation

- Schedule:
  - Add a constraint to the queue.
  - But only do so if the constraint is *active* [to be defined later].
  - And avoid adding it more than once if it is already in the queue!

- Code:

```
public void schedule(Constraint c) {
    if (c.isActive() && !c.isScheduled()) {
        c.setScheduled(true);
        propagationQueue.add(c);
    }
}
```

MiniCP Implementation

- Propagate:
  - Handle a constraint pulled from the queue.
  - Record that it is no longer in the queue.
  - Apply the filtering if it is still active.

- Code:

```
private void propagate(Constraint c) {
     c.setScheduled(false);
     if (c.isActive())
         c.propagate();
}
```

MiniCP Implementation

- Fixpoint:
  - Pull constraints from the queue and propagate them.
  - If some filtering finds a contradiction, then catch the exception!
  - If an exception (contradiction) was raised, then clear the queue.

- Code:

```
public void fixPoint() {
    try {
        while (!propagationQueue.isEmpty()) {
            propagate(propagationQueue.remove());
        }
    } catch (InconsistencyException e) {
        while (!propagationQueue.isEmpty())
            propagationQueue.remove().setScheduled(false);
        throw e;
    }
}
```

Constraint Programming

Variable Views

An 8-Queens Model

```
import static minicp.cp.Factory.*
public class NQueens {
   public static void main(String[] args) {
      int n = 8; // number of queens and size of board
      Solver cp = makeSolver();
      IntVar[] q = makeIntVarArray(cp,n,0,n-1);
      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], i-j));
            cp.post(notEqual(q[i], q[j], j-i));
         }

   }
}}
```

```
import static minicp.cp.Factory.*
public class NQueens {
   public static void main(String[] args) {
      int n = 8; // number of queens and size of board
      Solver cp = makeSolver();
      IntVar[] q = makeIntVarArray(cp,n,0,n-1);
      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(plus(q[i],j-i), q[j]));
            cp.post(notEqual(minus(q[i],j-i), q[j]));
         }

   }
}}
```

Constructor Blow-up

```
public NotEqualPlus(IntVar x, IntVar y, int v)  // x != y + v
public NotEqualMul(IntVar x, IntVar y, int v)   // x != y * v
```

…

And you'd need to do this for all the constraints.

Code pasting, source of bugs (more complex algorithms), etc.
We need to find a software engineering solution.

An 8-Queens Model

```
import static minicp.cp.Factory.*
public class NQueens {
   public static void main(String[] args) {
      int n = 8; // number of queens and size of board
      Solver cp = makeSolver();
      IntVar[] q = makeIntVarArray(cp,n,0,n-1);
      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], i-j));
              cp.post(notEqual(q[i], q[j], j-i));
          }

   }
}}
```

```
import static minicp.cp.Factory.*
public class NQueens {
   public static void main(String[] args) {
      int n = 8; // number of queens and size of board
      Solver cp = makeSolver();
      IntVar[] q = makeIntVarArray(cp,n,0,n-1);
      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(plus(q[i],j-i), q[j]));
              cp.post(notEqual(minus(q[i],j-i), q[j]));
          }

   }
}}
```

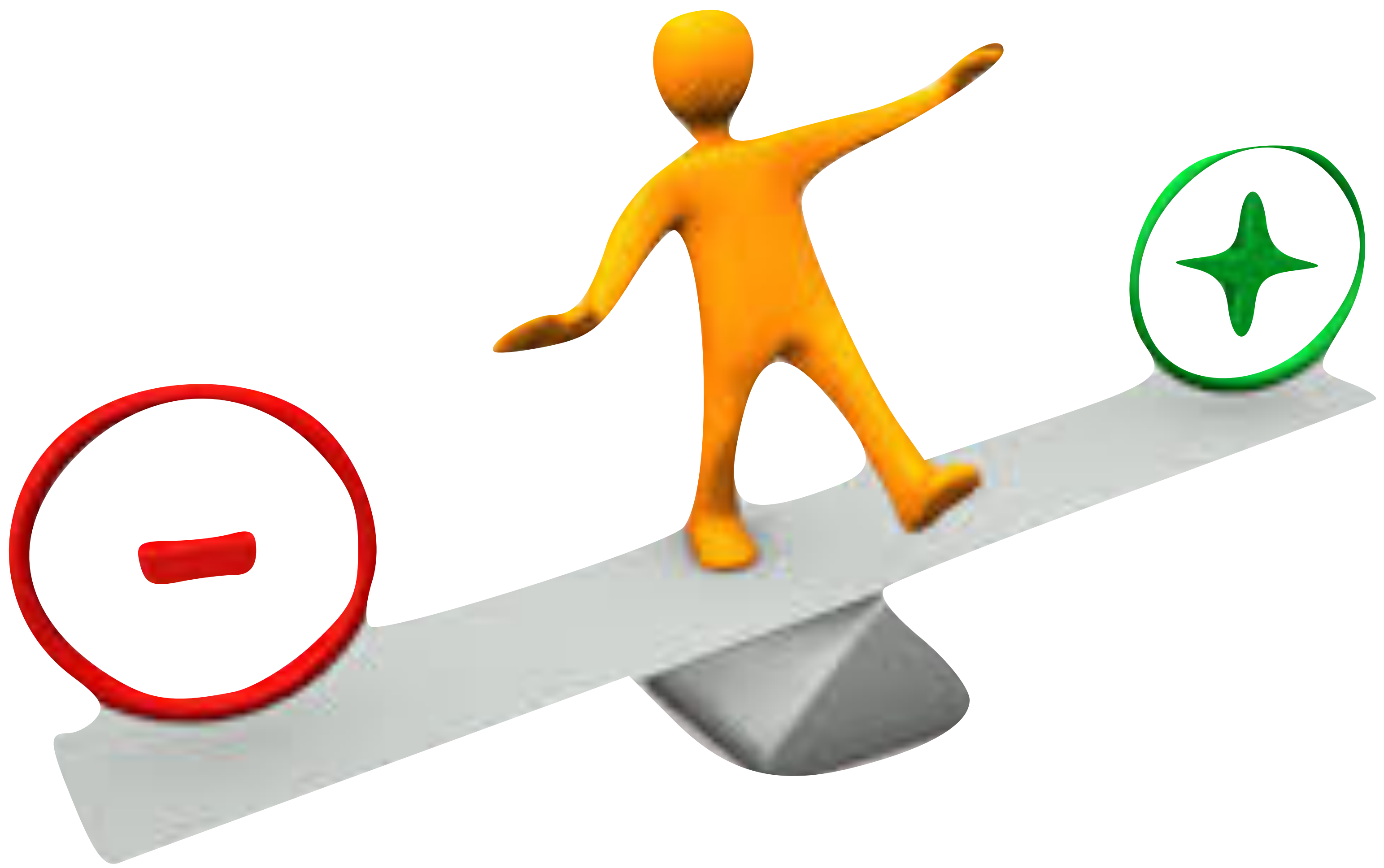
How to implement this?

Option 1

- Do a decomposition:
  - Introduce a variable for each expression.
  - Add an equality constraint between the fresh variable and the expression.
  - Collect the new variables in an array.
  - Reuse the good old NotEqual constraint.

Practically

- Constraints to consider:
  - Offset: $Y = X + o$
  - Opposite: $Y = -X$
  - Scale: $Y = a * X$ (with $a > 0$)
- Pros:
  - Easy to do.
  - Reuse of the other constraints.
- Cons:
  - This ***adds*** to the fixpoint computation!
- Can we…
  - Have our cake and eat it too?



Idea

- Databases: this was solved a long time ago…
  - They have TABLES.
  - They have **VIEWS.**
- Meaning?
  - A view is a fake table.
  - It acts like a table.
  - It smells like a table.
  - But there is neither a table nor storage behind it.
- So…
  - What about having **FAKE** variables that pretend to be, but rely on, others?

Variable Views

- Capture simple relations:
  - Binary (i.e., on 2 variables).
  - Bijective.
- Examples:
  - Offset: $Y = X + o$
  - Opposite: $Y = -X$
  - Scale: $Y = a * X$ (for $a > 0$)
- How to do this?
  - Delegation!

Variable Views: Taxonomy

- Views delegate to the “real” variable and apply the mapping.

Expression	View
Y = X + o	Y = IntVarViewOffset(X,o)
Y = – X	Y = IntVarViewOpposite(X)
Y = a * X	Y = IntVarViewMul(X,a)

Examples

- Consider:
  - Y = X + 2
  - X is an integer variable with D(X) = {1..10}
  - Y is a view variable
- Operations:
  - contains(Y, 8) translates into contains(X, 8 **– 2**)
  - removeBelow(Y, 5) translates into removeBelow(X, 5 **– 2**)
  - min(Y) translates into min(X) **+ 2**
  - max(Y) translates into max(X) **+ 2**

Variable Views: Offset Implementation

```
public class IntVarViewOffset implements IntVar {
    private final IntVar x;
    private final int o;
    public IntVarViewOffset(IntVar x, int offset) { // y = x + o
        this.x = x;
        this.o = offset;
    }
    public Solver getSolver()                            { return x.getSolver();}
    public void propagateOnDomainChange(Constraint c) { x.propagateOnDomainChange(c);}
    public void propagateOnFix(Constraint c)             { x.propagateOnFix(c);}
    public void propagateOnBoundChange(Constraint c)  { x.propagateOnBoundChange(c);}

    public int min()  { return x.min() + o;}
    public int max()  { return x.max() + o;}
    public int size() { return x.size();}
    public boolean isFixed() { return x.isFixed();}
    public boolean contains(int v) { return x.contains(v - o);}
    public void remove(int v) {x.remove(v - o);}
    public void fix(int v) {x.fix(v - o);}
    public void removeBelow(int v) { x.removeBelow(v - o);}
    public void removeAbove(int v) { x.removeAbove(v - o);}
}
```

Then…

- Use the variable view exactly like a normal variable!
- There are even factory methods to instantiate a view, such as `plus` and `minus`:

```
import static minicp.cp.Factory.*
public class NQueens {
   public static void main(String[] args) {
      int n = 8; // number of queens and size of board
      Solver cp = makeSolver();
      IntVar[] q = makeIntVarArray(cp,n,0,n-1);
      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(plus(q[i],j-i), q[j]));
              cp.post(notEqual(minus(q[i],j-i), q[j]));
          }

      }
   }
}
```

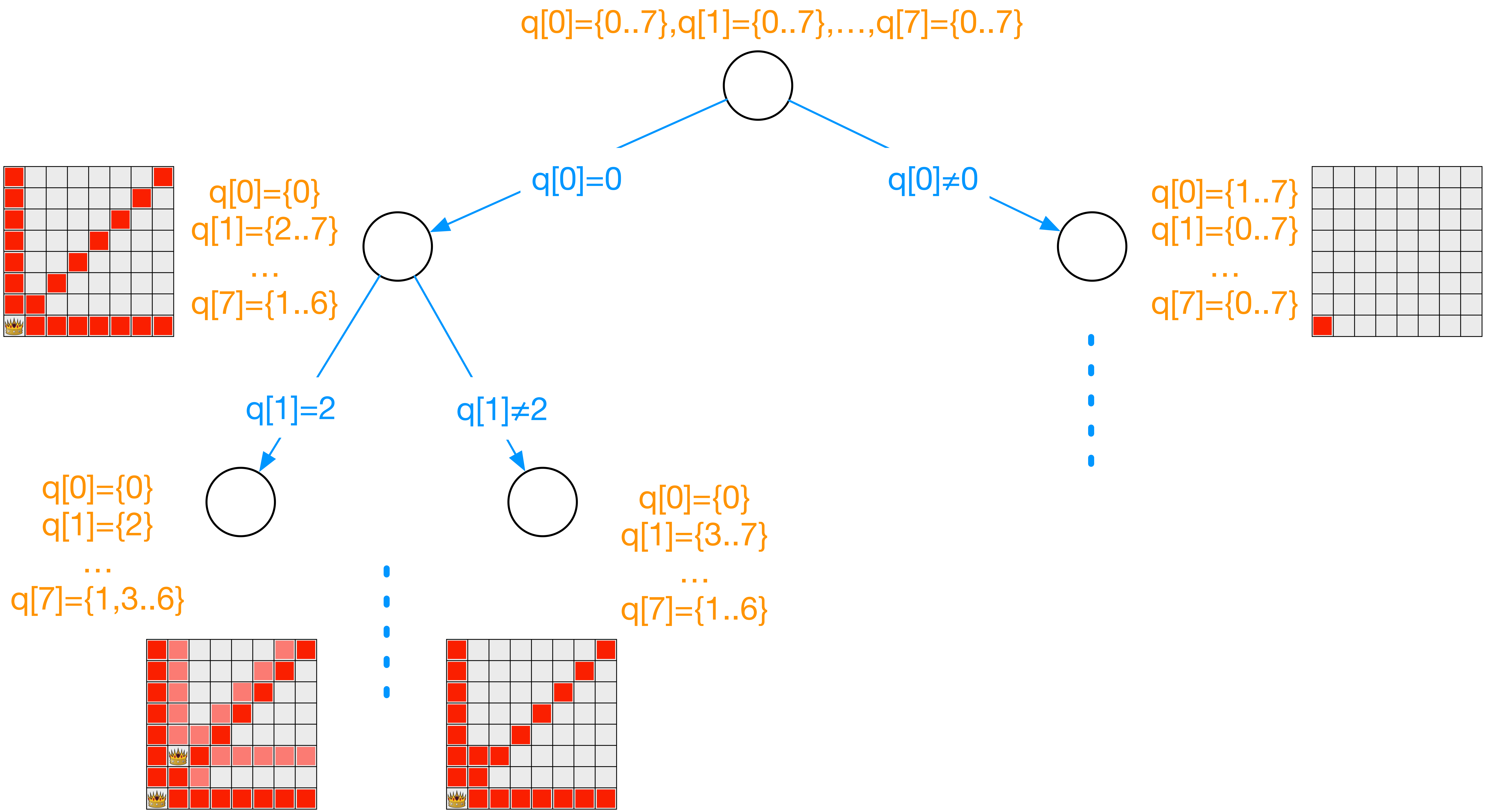
Constraint Programming

State Management and Search

Search

- Questions:
  - What does the search tree **look like**?
  - How does one **define** the search tree?
  - How does one **explore** the search tree?
  - How does one **terminate** the search?

Visualization for 8-Queens



Search Tree

- Nodes are computational states:
  - They represent subproblems that are to be solved.

- Branching decisions are choices made:
  - They must *partition* the search space.
  - They act on variables.

Branching Decisions

- Branching decisions must partition the search space.
- In practice:
  - One often partitions the domain of only *one* variable.
  - Example:

$$\text{Given } D(x) = \{0..(n-1)\}$$

- Option 1: binary labeling $x = v \lor x \neq v$
- Option 2: binary split $x \leq \text{mid}(D(x)) \lor x > \text{mid}(D(x))$
- Option 3: *n*-ary labeling $x = 0 \lor x = 1 \lor x = 2 \lor \cdots \lor x = n-1$
- …

In General

- Given a CSP $CSP = \langle X, D, C \rangle$

- branch with $branching = \{c_0, \ldots, c_{k-1}\}$

- such that all solutions are preserved and found only once:

$$\bigcup_{i \in \{0..k-1\}} \mathcal{S}(\langle X, D, C \cup \{c_i\} \rangle) = \mathcal{S}(\langle X, D, C \rangle)$$

$$\forall i \neq j \in \{0..k-1\} : \mathcal{S}(\langle X, D, C \cup \{c_i\} \rangle) \cap \mathcal{S}(\langle X, D, C \cup \{c_j\} \rangle) = \emptyset$$

Requirements

- We may not lose solutions when we branch.
- We may not repeat solutions when solving the child nodes: efficiency.

All 3 options in the example above satisfy both requirements:

$$x = v \vee x \neq v$$

$$x = 0 \vee x = 1 \vee x = 2 \vee \cdots \vee x = n - 1$$

$$x \leq \mathsf{mid}(D(x)) \vee x > \mathsf{mid}(D(x))$$

Observations

- This is a recursive process:
  - Do the same reasoning at every node of the tree.

- The variable selection at any node has an impact on the tree shape.

- The partition selection impacts the shape and size of the subtrees.

How does one define the search tree?

- The definition of a search tree is made by a *branching scheme*:
  - Algorithm, to be applied recursively, that generates the branching decisions.
  - Each time:
    - Select a variable.
    - Select a partition.
    - Produce a set of constraints: $branching = \{c_0, \ldots, c_{k-1}\}$

- Example with a static variable ordering:
  - Let *i* be the index of the first still unfixed variable of the model.
  - First try the minimum value in the domain of the selected variable:

$$branching_i = \{x_i = \min(D(x_i)),\ x_i \neq \min(D(x_i))\}$$

How does one explore the search tree?

- Tree exploration means the order in which the nodes are examined.

- What is easy to implement?

DFS (depth-first search)

- Why?

  – Low memory requirement.
  – Chronological backtracking.

How does one terminate the search?

- When all the variables are fixed:
  - We have a solution and we can stop.
- When the domain of some variable becomes empty (it has a *wipe-out*):
  - There is a contradiction and we should backtrack.
- When at least one variable x has $|D(x)| \geq 2$:
  - We need to recur.

- Therefore:
  - If we want *one* solution, then we stop as soon as all variables are fixed.
  - If we want *all* solutions, then we display each solution *and* backtrack.

Depth-First Search

- It is a template, really!
- The abstract algorithm:

Data: The CSP $\langle X, \mathcal{D}, C \rangle$
Result: $\texttt{CPSearch}(\langle X, \mathcal{D}, C \rangle) = \mathcal{S}(\langle X, \mathcal{D}, C \rangle)$
$\mathcal{D}^* \leftarrow \mathcal{F}(\langle X, \mathcal{D}, C \rangle)$
if $|\mathcal{D}^*| = 0$ **then**
  return $\emptyset$
end

Empty-domain detection

else if $|\mathcal{D}^*| = 1$ **then**
  return $\{\mathcal{D}^*\}$
end

Solution detection

else
  $(c_0, \cdots, c_{k-1}) \leftarrow \texttt{branch}(\langle X, \mathcal{D}^*, C \rangle)$

Branching

  return $\bigcup_{I=1}^{k} \texttt{CPSearch}(\langle X, \mathcal{D}^*, C \cup \{c_i\} \rangle)$

Recur

end

Constraint Programming

DFS Implementation

What Will It Take?

- The algorithm is generic:
  - Branching scheme: **make it a first-order function**.
  - Branching scheme encapsulates the selection of a variable and a partitioning.
- The algorithm is recursive:
  - Makes DFS straightforward.
  - Explores the leftmost path first!
- Key operation to recur on:
  - Given $\langle X,D,C \rangle$, find a branching $\{ c_0,\ldots,c_{k-1} \}$.
  - From a branching $\{ c_0,\ldots,c_{k-1} \}$ and $\langle X,D,C \rangle$:
    - **Produce** $\langle X, D, C \cup \{c_i\} \rangle$ to form each of the k recursive calls.
    - **Discard** $\langle X, D, C \cup \{c_i\} \rangle$ when returning from a recursive call.

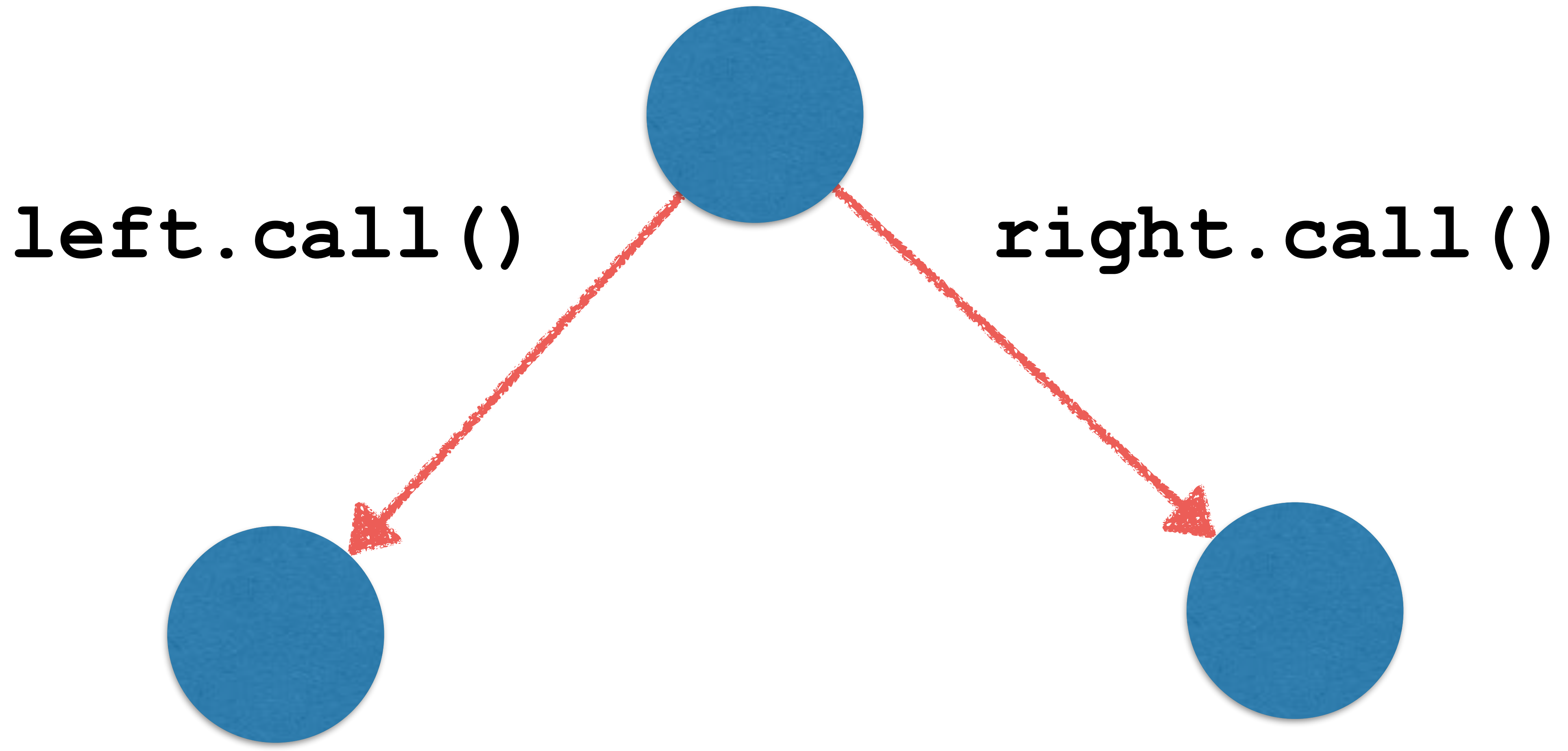
Branching Scheme: Finding a Branching

- Task:
  - Select an unfixed variable to branch on, if there is one; else selection fails.
  - Select a partitioning of the domain of the selected variable (e.g., via = and ≠).
  - Return a set of constraints to branch on.
- Outcomes:
  - If variable selection fails, then return an empty set of branches (= constraints).
  - If variable selection succeeds, then return a set of at least 2 branches.

Procedures

```java
@FunctionalInterface
public interface Procedure {
    /**
     * Calls the procedure
     */
    void call();
}
```

```java
Procedure left = () -> cp.post(equal(qi, v));
Procedure right = () -> cp.post(notEqual(qi, v))
return new Procedure[]{left,right};
```



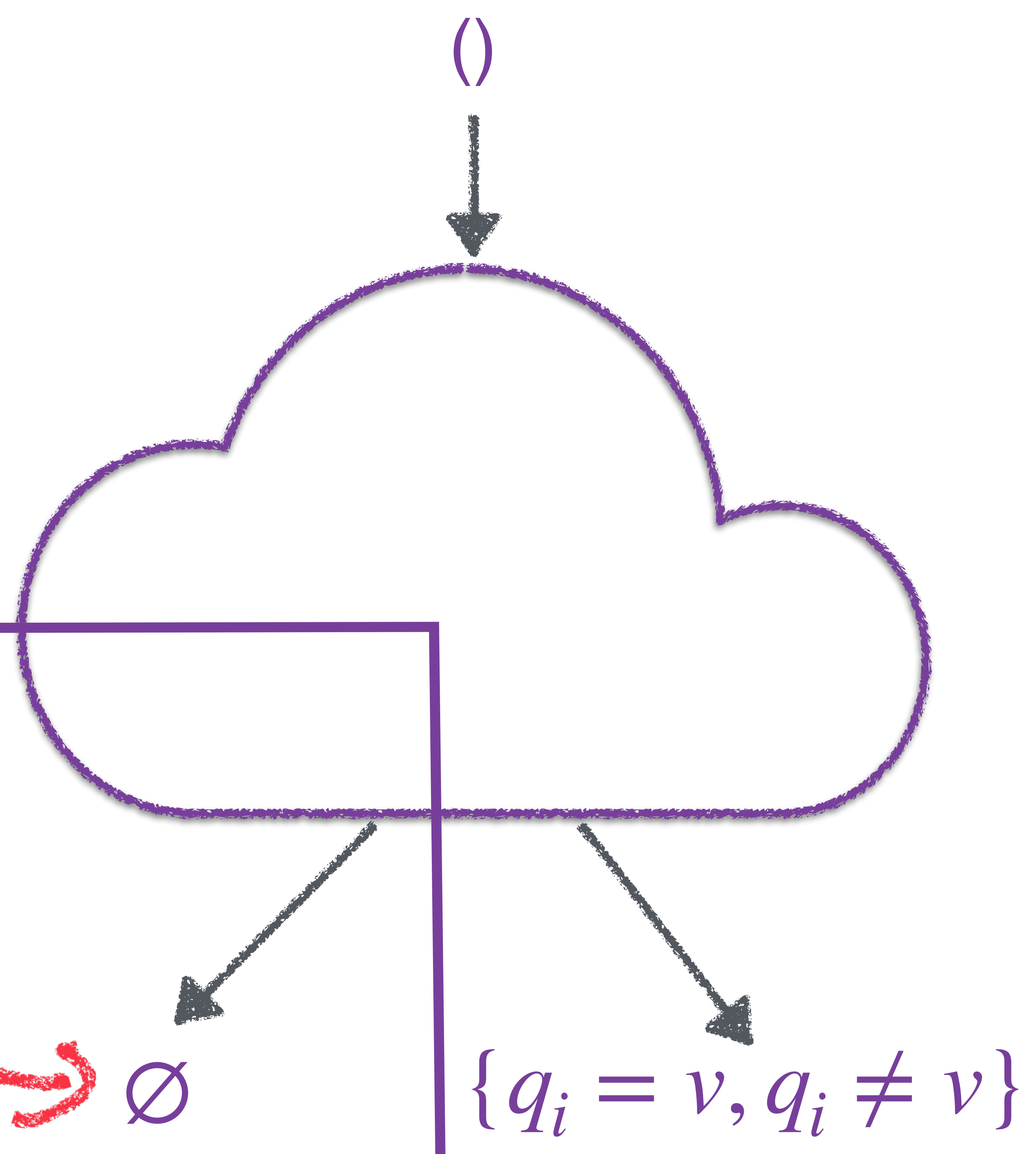

```java
Procedure myProc = () -> System.out.println("hello");
myProc.call();
```

Branching Scheme: A First-Order Example

```
public class NQueens {
    public static void main(String[] args) {
        int n = 8; // number of queens and size of board
        Solver cp = makeSolver();
        IntVar[] q = makeIntVarArray(cp,n,0,n-1);
        // …constraints… …
        DFSearch search = Factory.makeDfs(cp, () -> {
                int idx = -1;
                for (int k = 0; k < q.length; k++)
                   if (q[k].size() > 1) { idx = k; break; }
                if (idx == -1) return new Procedure[0];
                else {
                    IntVar qi = q[idx];
                    int v = qi.min();
                    Procedure left = () -> cp.post(equal(qi, v));
                    Procedure right = () -> cp.post(notEqual(qi, v));
                    return new Procedure[]{left,right};
                }
         });

    }
}}
```

branching scheme

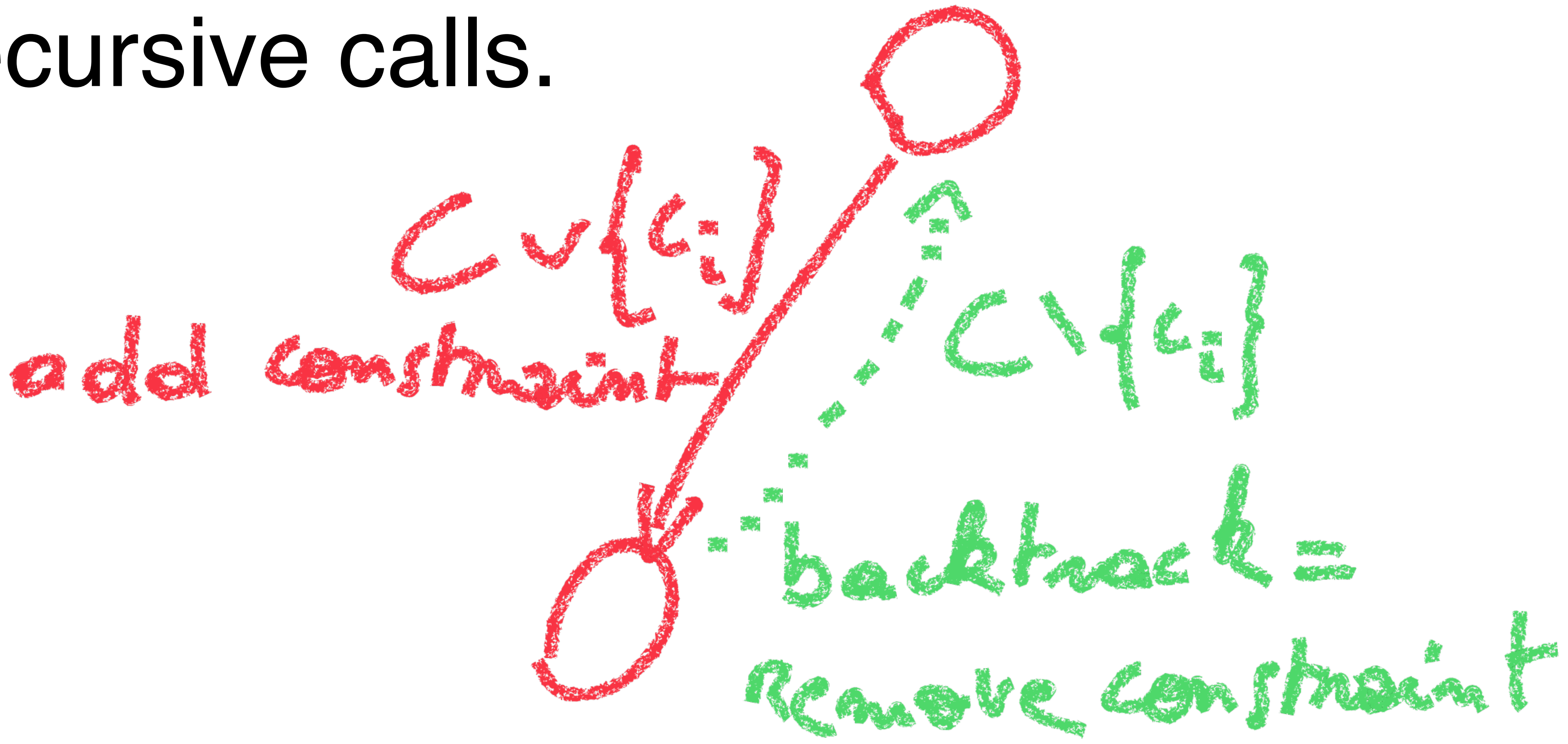


Constraint Programming

DFS State Restoration

Produce $\langle X, D, C \cup \{c_i\}\rangle$

- Iterate:
  - **Produce** $\langle X, D, C \cup \{c_i\}\rangle$ to form each of the k recursive calls.



- That is an *entire* CSP each time!

- On top of this:
  - **Discard** $\langle X, D, C \cup \{c_i\}\rangle$ when returning from a recursive call.

- How can we do this?
- How can we do this efficiently?

Basic Idea [naïve]

- Do *not* copy the whole CSP.
- Instead:
  - Do an **in-place** modification.
  - Add c_i to the *current* CSP.
- But:
  - **Back up** anything that *might* change as a result!
  - That means… backup the whole state.
  - So funny… we have a StateManager!

Abstraction

- Let us assume that the StateManager does the right thing as long as:

    1. We “push” a backup before we recur in each iteration.
    2. We “restore” the top-most backup before we iterate.

- We will later show how to make a backup.

The StateManager API

```
package minicp.state;
import minicp.util.Procedure;

public interface StateManager {
    int getLevel();
    void saveState();
    void restoreState();
    void restoreStateUntil(int level);
    void onRestore(Procedure listener);

    void withNewState(Procedure body);

    <T> State<T> makeStateRef(T initValue);
    StateInt makeStateInt(int initValue);
}
```

Low-level "Backup" API

Convenience API

Factory API

Convenience Functions

```
public class SomeStateImplementation {
   …
   public void withNewState(Procedure body) {
      final int level = getLevel();
      saveState();
      body.call();
      restoreStateUntil(level);
   }
   public void restoreStateUntil(int level) {
      while (getLevel() > level)
          restoreState();
   }
}
```

Example of StateManager API

```
StateManager sm = new Trailer() // new Copier();

// Two stateful int inside sm
StateInt a = sm.makeStateInt(7);
StateInt b = sm.makeStateInt(13);
// Record current state a=7, b=13 and increase the level to 0:
sm.saveState();
  a.setValue(6);
  a.setValue(11);
  // Record current state a=11, b=13 and increase the level to 1:
  sm.saveState();
    a.setValue(4);
    b.setValue(9);
  // Restore a=11, b=13:
  sm.restoreState();
// Restore a=7, b=13:
sm.restoreState();
```

DFS Template

```
package minicp.search;
public class DFSearch {
    private Supplier<Procedure[]> branching;
    private StateManager sm;
    private List<Procedure> solutionListeners = new LinkedList<Procedure>();
    public DFSearch(StateManager sm, Supplier<Procedure[]> branching) {
        this.sm = sm;
        this.branching = branching;
    }
    public void onSolution(Procedure listener) { solutionListeners.add(listener);}
    private void notifySolution() { solutionListeners.forEach(s -> s.call());}
    private SearchStatistics solve(SearchStatistics statistics) {
        sm.withNewState(() -> {
            try {
                dfs(statistics);
                statistics.setCompleted();
            } catch (StopSearchException ignored) {}
        });
        return statistics;
    }
    public SearchStatistics solve() { return solve(new SearchStatistics());}
}
```

DFS Template

```
private void dfs(SearchStatistics statistics) {
    Procedure[] branches = branching.get();
    if (branches.length == 0) {
        statistics.incrSolutions();
        notifySolution();
    } else {
        for (Procedure b : branches) {
            sm.withNewState(() -> {
                try {
                    statistics.incrNodes();
                    b.call();
                    dfs(statistics);
                } catch (InconsistencyException e) {
                    statistics.incrFailures();
                }
            });
        }
    }
}
```

Test branching validity

Report a solution

Branch & Recur

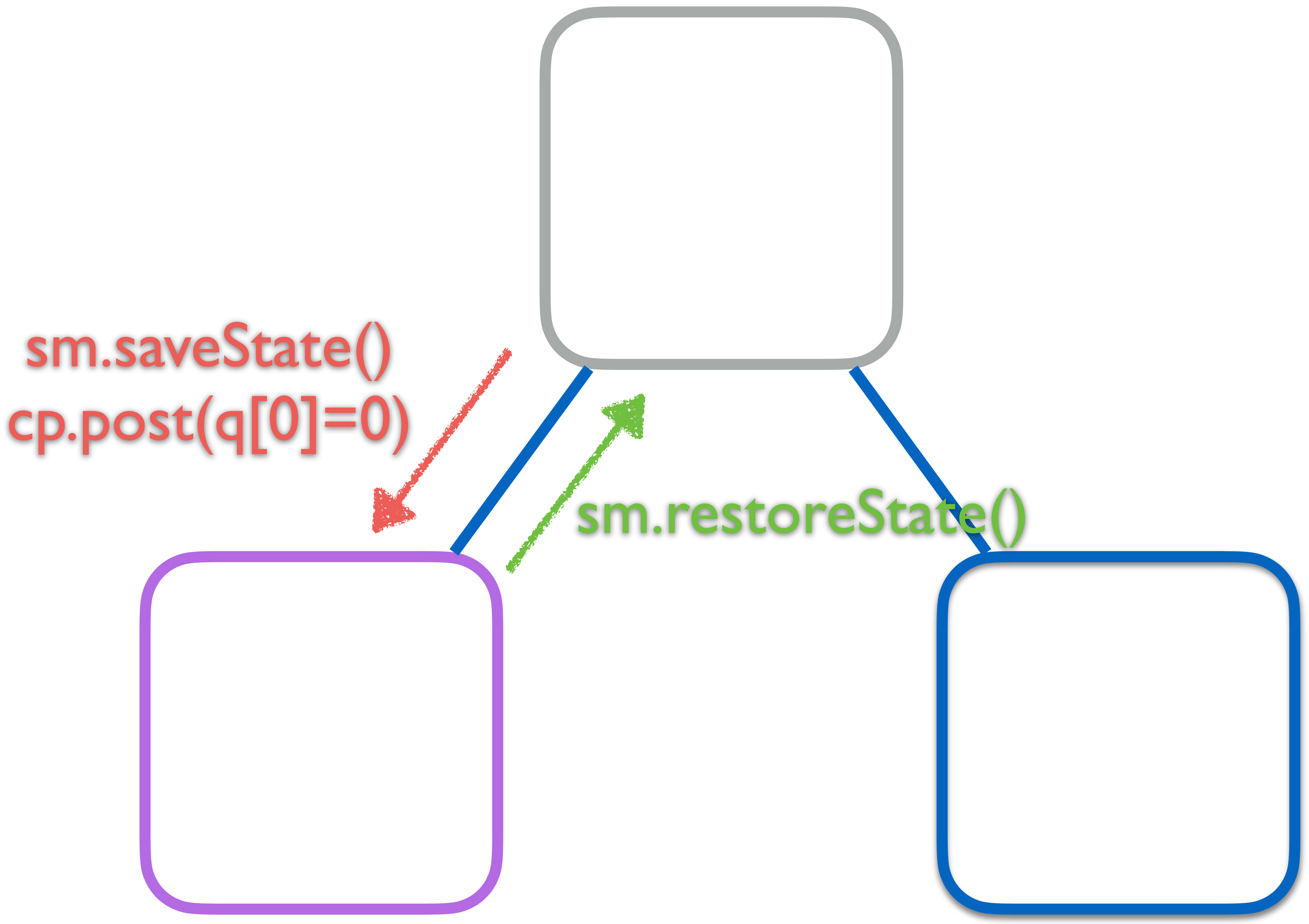
For each branch…

Equivalent to

```
private void dfs(SearchStatistics statistics) {
   Procedure[] branches = branching.get();
    if (branches.length == 0) {
        statistics.incrSolutions();
        notifySolution();
    } else {
        for (Procedure b : branches) {
            sm.saveState();
            try {
                    statistics.incrNodes();
                    b.call();
                    dfs(statistics);
            } catch     (InconsistencyException e) {
                    statistics.incrFailures();
            }
            sm.restoreState();
        }
    }
}
```

Visually

- At every node:
  - Save the state (domains, constraints, etc).
  - Restore it on backtrack.



Constraint Programming

State Management: Copying vs Trailing

Two different strategies

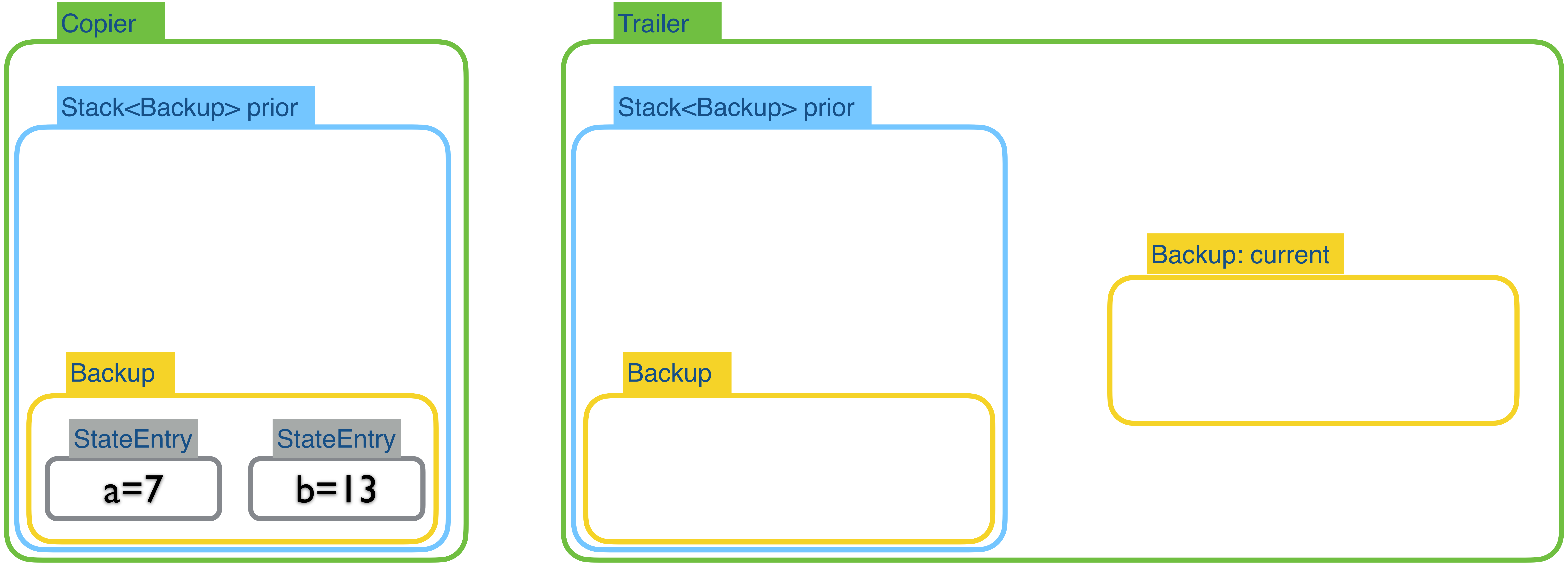
- Copier (eager / brute-force)
  - When **saveState** is called:
    - Copy all stateful objects into a backup.
    - Push that backup onto a stack.
  - When a stateful object is modified:
    - Do nothing!
  - When **restoreState** is called:
    - Pop the topmost backup from the stack.
    - Restore the content of the popped backup.
- This is a *eager* or brute-force backup, copy everything without working about small changes
- Easier to parallelize the search.

- Trailer (incremental and lazy)
  - When **saveState** is called:
    - Push the current backup onto a stack.
    - Create a new current backup that is empty.
  - When a stateful object is modified:
    - Log the change in the current backup.
  - When **restoreState** is called:
    - Restore the content of the current backup.
    - Pop the topmost backup from the stack.
    - Restore the content of the popped backup into the current backup.
- This is a *lazy* backup: it can be seen as “backup on write”.
- Not so easy to parallelize the search

Trailer vs Copier

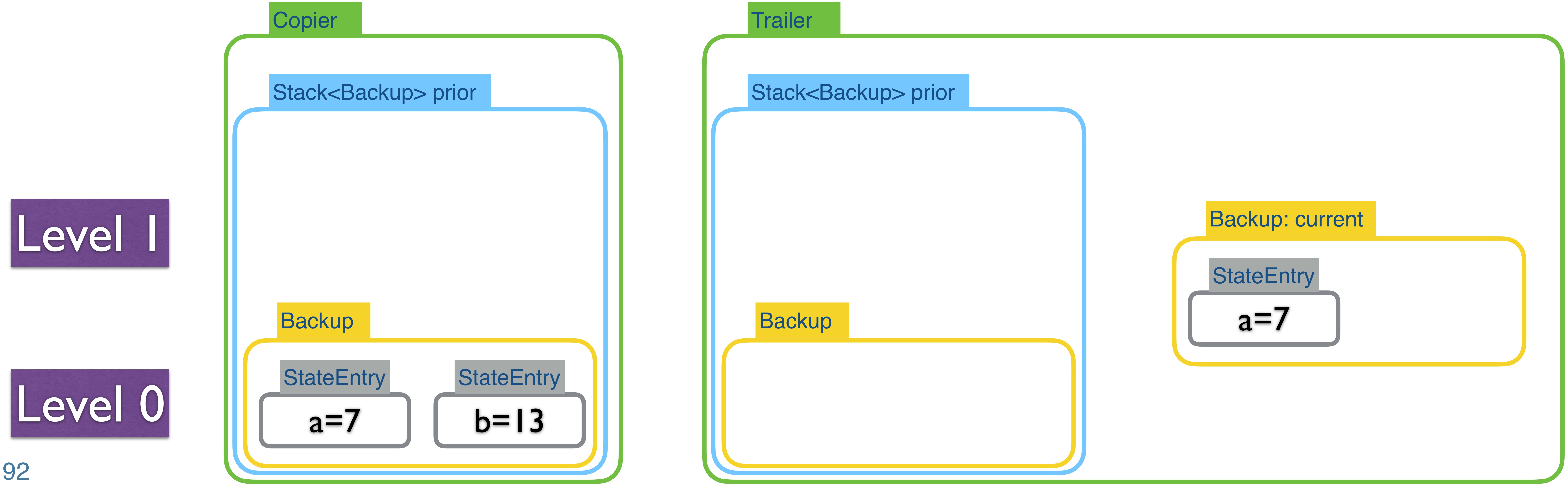
```
StateManager sm = new Trailer() // new Copier();
StateInt a = sm.makeStateInt(7);
StateInt b = sm.makeStateInt(13);
sm.saveState();
```

Level 0



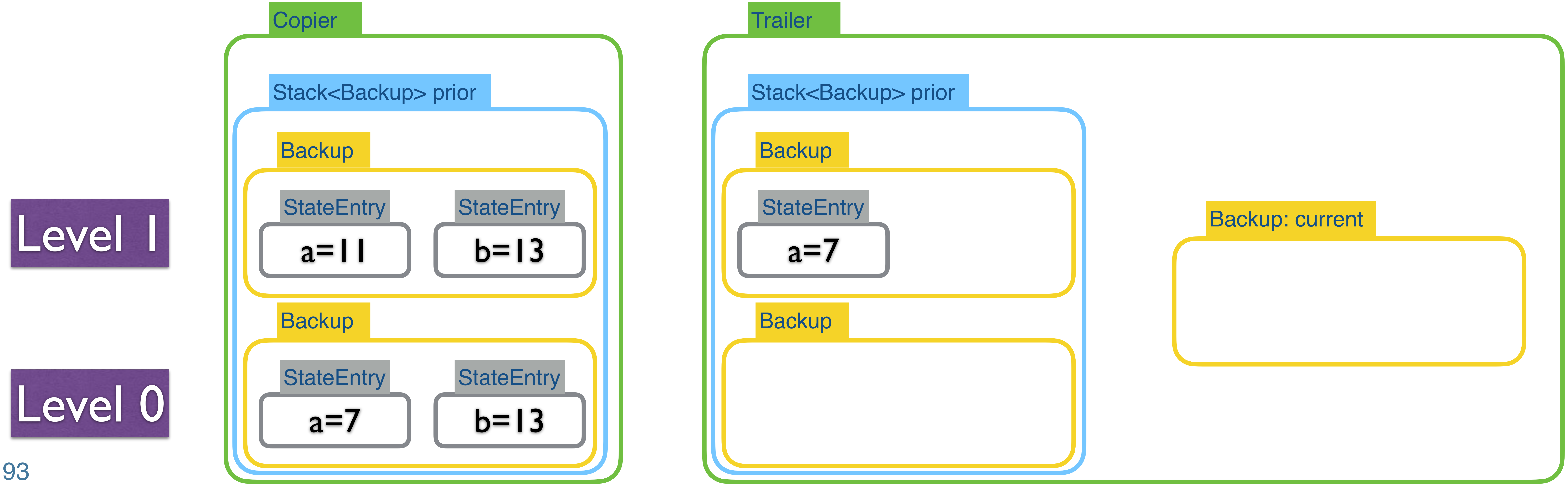
Trailer vs Copier

```
StateManager sm = new Trailer() // new Copier();
StateInt a = sm.makeStateInt(7);
StateInt b = sm.makeStateInt(13);
sm.saveState();
  a.setValue(6);
  a.setValue(11);
```



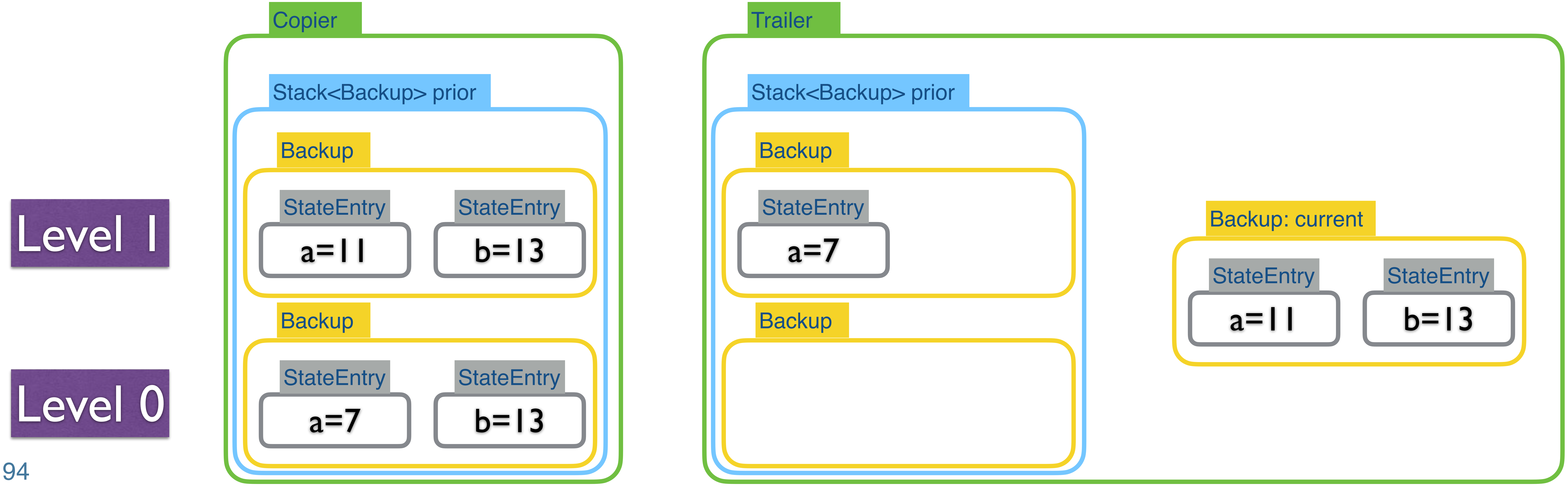
Trailer vs Copier

```
StateManager sm = new Trailer() // new Copier();
StateInt a = sm.makeStateInt(7);
StateInt b = sm.makeStateInt(13);
sm.saveState();
  a.setValue(6);
  a.setValue(11);
  sm.saveState();
```



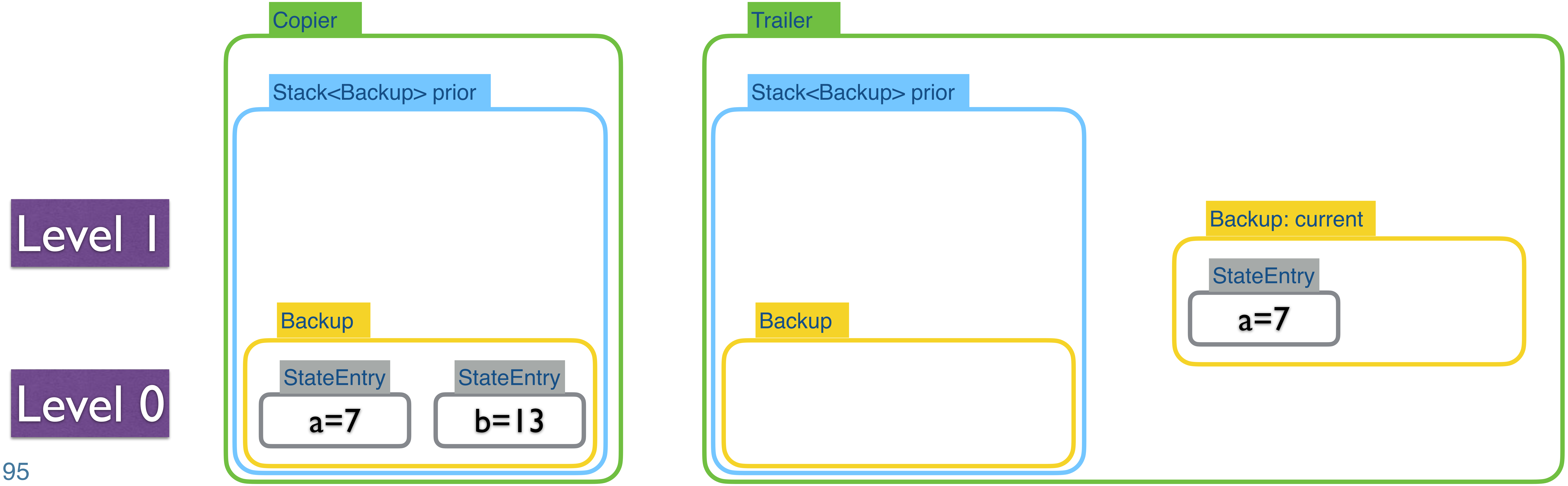
Trailer vs Copier

```
StateManager sm = new Trailer() // new Copier();
StateInt a = sm.makeStateInt(7);
StateInt b = sm.makeStateInt(13);
sm.saveState();
  a.setValue(6);
  a.setValue(11);
  sm.saveState();
    a.setValue(4);
    b.setValue(9);
```



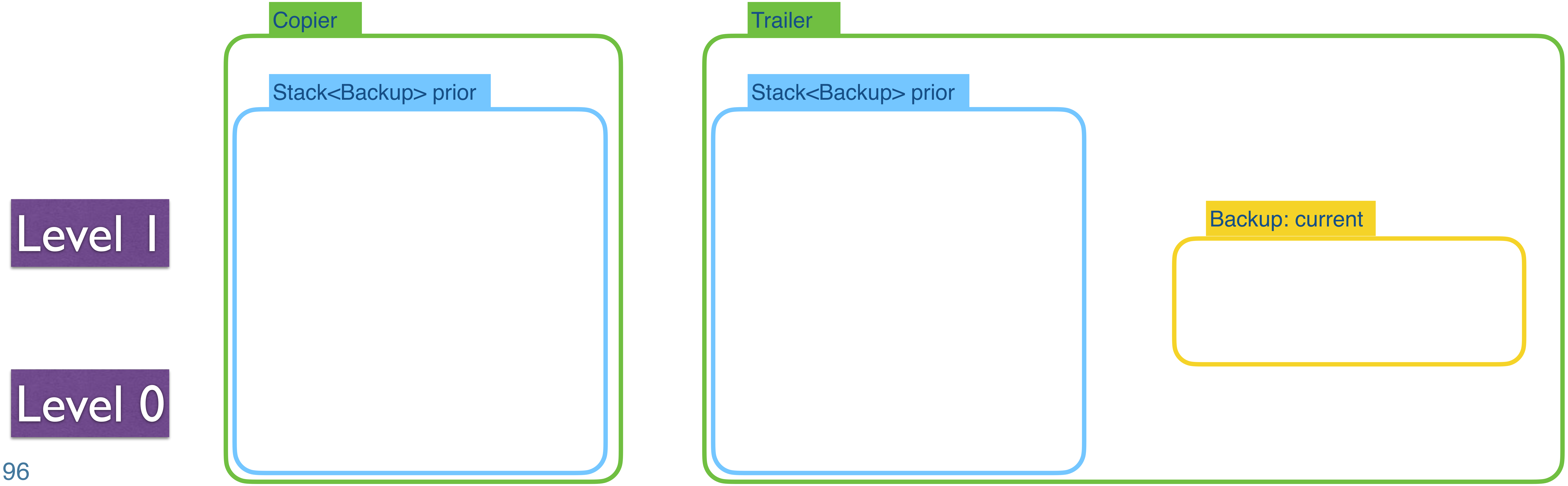
Trailer vs Copier

```
StateManager sm = new Trailer() // new Copier();
StateInt a = sm.makeStateInt(7);
StateInt b = sm.makeStateInt(13);
sm.saveState();
  a.setValue(6);
  a.setValue(11);
  sm.saveState();
    a.setValue(4);
    b.setValue(9);
  sm.restoreState();
```



Trailer vs Copier

```
StateManager sm = new Trailer() // new Copier();
StateInt a = sm.makeStateInt(7);
StateInt b = sm.makeStateInt(13);
sm.saveState();
  a.setValue(6);
  a.setValue(11);
  sm.saveState();
    a.setValue(4);
    b.setValue(9);
  sm.restoreState();
sm.restoreState();
```



Two different strategies

- Copier (eager / brute-force)
  - When **saveState** is called:
    - Copy all stateful objects into a backup.
    - Push that backup onto a stack.
  - When a stateful object is modified:
    - Do nothing!
  - When **restoreState** is called:
    - Pop the topmost backup from the stack.
    - Restore the content of the popped backup.
- This is a *eager* or brute-force backup, copy everything without working about small changes
- Easier to parallelize the search.

- Trailer (incremental and lazy)
  - When **saveState** is called:
    - Push the current backup onto a stack.
    - Create a new current backup that is empty.
  - When a stateful object is modified:
    - Log the change in the current backup.
  - When **restoreState** is called:
    - Restore the content of the current backup.
    - Pop the topmost backup from the stack.
    - Restore the content of the popped backup into the current backup.
- This is a *lazy* backup: it can be seen as "backup on write".
- Not so easy to parallelize the search

Copier Doing a StateEntry Eager Backup

```
public class Copy<T> implements Storage, State<T> {
    class CopyStateEntry implements StateEntry {
        private final T v;
        CopyStateEntry(T v)                  { this.v = v;}
        @Override public void restore() { Copy.this.v = v;}
    }
    private T v;
    protected Copy(T initial)  { v = initial;}
    public T setValue(T v)     { return this.v = v;}
    public T value()           { return v;}
    public String toString() { return String.valueOf(v);}
    public StateEntry save() { return new CopyStateEntry(v);}
}
```

Object used to record a snapshot and possibly restore its content later

```
public class CopyInt extends Copy<Integer> implements StateInt {
    protected CopyInt(int initial) {
        super(initial);
    }
}
```

Copier StateManager

MiniCP

```
public class Copier implements StateManager {
    class Backup extends Stack<StateEntry> {
        private int sz;
        Backup() {
            sz = store.size();
            for (Storage s : store)
                add(s.save());
        }
        void restore() {
            store.setSize(sz);
            for (StateEntry se : this)
                se.restore();
        }
    }
    private Stack<Storage> store;
    private Stack<Backup> prior;
    public Copier() {
        store = new Stack<Storage>();
        prior = new Stack<Backup>();
    }
  ……
}
```

A backup is a full snapshot computed upon “saveState” and pushed onto the stack

All the StateX objects to store upon “saveState”

Copier StateManager

```
public int getLevel()       { return prior.size() - 1;}
public int storeSize()      { return store.size();}

public void saveState()     { prior.add(new Backup());}
public void restoreState()  { prior.pop().restore();}

public void withNewState(Procedure body) {
      final int level = getLevel();
      saveState();
      body.call();
      restoreStateUntil(level);
 }
public void restoreStateUntil(int level) {
      while (getLevel() > level)
          restoreState();
 }
public StateInt makeStateInt(int initValue) {
     CopyInt s = new CopyInt(initValue);
     store.add(s);
     return s;
 }
}
```

```
public class Copier implements StateManager {
    class Backup extends Stack<StateEntry> {
        private int sz;
        Backup() {
            sz = store.size();
            for (Storage s : store)
                add(s.save());
        }
        void restore() {
            store.setSize(sz);
            for (StateEntry se : this)
                se.restore();
        }
    }
    private Stack<Storage> store;
    private Stack<Backup> prior;
    public Copier() {
        store = new Stack<Storage>();
        prior = new Stack<Backup>();
    }
  ......
}
```

Factory creation of “Copyable Int”

Copier: Complexity Analysis

- When creating a state:
  - Zero.
- When **saveState** is called:
  - Iterate over all stateful objects in order to back them up.
  - Cost is O(# stateful objects), in time and space.
- When **restoreState** is called:
  - Iterate over all entries in the popped backup.
  - Cost is again O(# stateful objects).

Two different strategies

- Copier (eager / brute-force)
  - When **saveState** is called:
    - Copy all stateful objects into a backup.
    - Push that backup onto a stack.
  - When a stateful object is modified:
    - Do nothing!
  - When **restoreState** is called:
    - Pop the topmost backup from the stack.
    - Restore the content of the popped backup.
- This is a *eager* or brute-force backup, copy everything without working about small changes
- Easier to parallelize the search.

- Trailer (incremental and lazy)
  - When **saveState** is called:
    - Push the current backup onto a stack.
    - Create a new current backup that is empty.
  - When a stateful object is modified:
    - Log the change in the current backup.
  - When **restoreState** is called:
    - Restore the content of the current backup.
    - Pop the topmost backup from the stack.
    - Restore the content of the popped backup into the current backup.
- This is a *lazy* backup: it can be seen as “backup on write”.
- Not so easy to parallelize the search

Custom State Object

- We need a “Stateful Int” object:
  - that is created by the Trailer Factory;
  - that lazily backs up to the state.
- Bottomline:
  - Programmers need never worry about how stateful objects are managed.

Implementation

```
public class Trailer implements StateManager {
    static class Backup extends Stack<StateEntry> {
        Backup() {}
        void restore() {
            for (StateEntry se : this)
                se.restore();
        }
    }
```

Lazy backup object

```
    private Stack<Backup> prior;
```

Stack of prior backups

```
    private Backup current;
```

Current lazy backup

```
    private long magic = 0L;
    public Trailer() {
        prior = new Stack<Backup>();
        current = new Backup();
    }
    public long getMagic() { return magic;}
}
```

Implementation

```
public void pushState(StateEntry entry) { current.push(entry);}
```

Enter a change

```
public int getLevel() { return prior.size() - 1;}
public void saveState() {
    prior.add(current);
    current = new Backup();
    magic++;
}
```

Save the current lazy backup and start a new one!

```
public void restoreState() {
    current.restore();
    current = prior.pop();
    notifyRestore();
}
```

Undo all the changes in current lazy backup and restore current to previous

```
public void restoreStateUntil(int level) {
    while (getLevel() > level)
        restoreState();
}
```

```
public StateInt makeStateInt(int initValue) {
    return new TrailInt(this,initValue);}
}
```

Factory creation of "Trailable Int"

Doing a StateEntry Lazy Backup…

```
public class Trail<T> implements State<T> {
    class TrailStateEntry implements StateEntry {
        private final T v;
        TrailStateEntry(T v)  { this.v = v;}
        public void restore() { Trail.this.v = v;}
    }
    private Trailer trail;
    private T v;
    private long lastMagic = -1L;
    protected Trail(Trailer trail, T initial) {
        this.trail = trail;v = initial;
        lastMagic = trail.getMagic() - 1;
    }
    private void trail() {
        long trailMagic = trail.getMagic();
        if (lastMagic != trailMagic) {
            lastMagic = trailMagic;
            trail.pushState(new TrailStateEntry(v));
        }
    }
    public T setValue(T v) {
        if (v != this.v) {
            trail();this.v = v;
        }
        return this.v;
    }
    public T value() { return this.v;}
}
```

Lazy backup logic

Constraint Programming

Data Structures with State: Sets, Stacks, etc.

A Stateful Set of Integers

- SparseSet again…
  - A set of integers:
    - With a StateInt each for the size, minimum, and maximum.
- Usage scenario, for the domain of a variable:
  - Only remove values from the set.
  - On backtrack, restore values into the set.

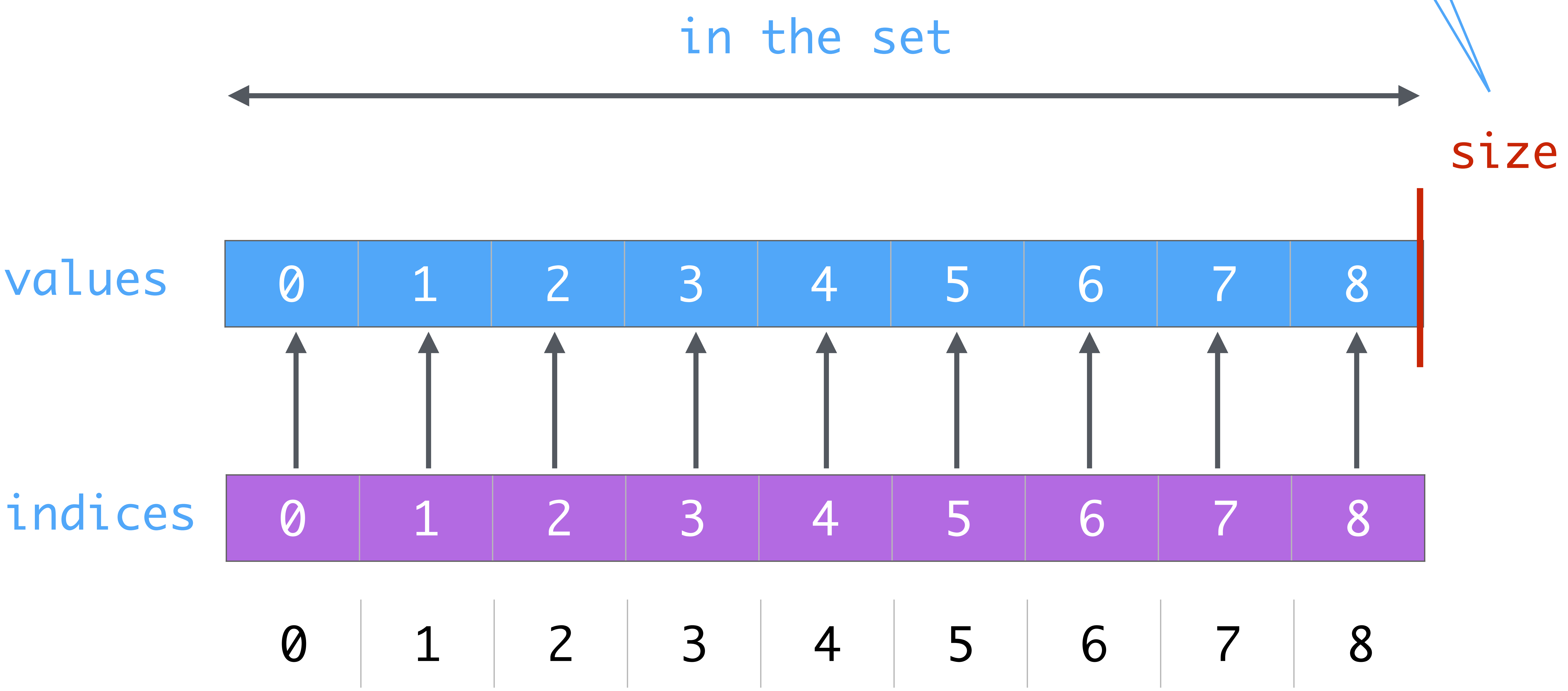
Revisiting SparseSet Operations…

- Let’s visualize this!

StateSparseSet

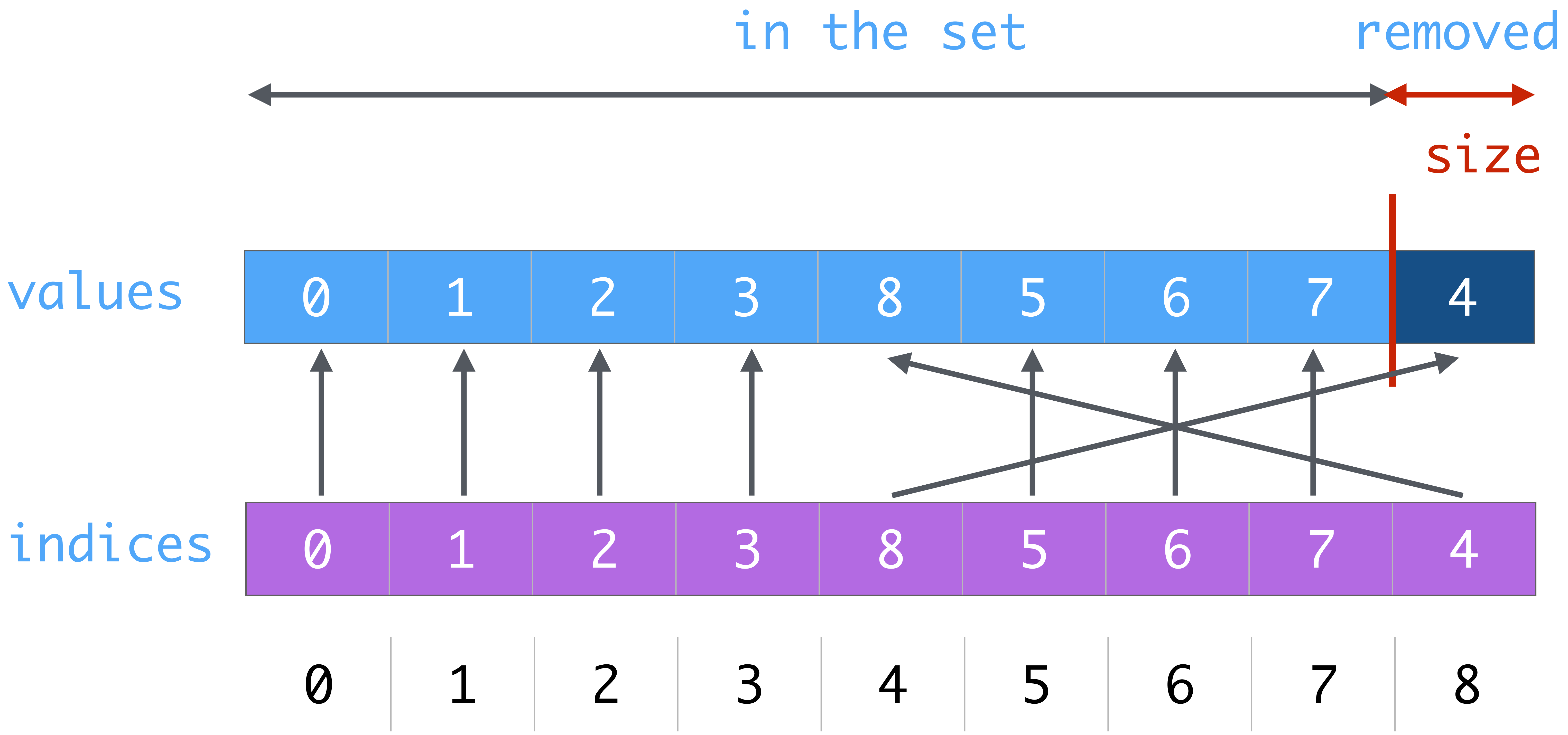
```
StateManager sm;
StateSparseSet set = sm.makeSparseSet(9);
sm.saveState();
set.remove(4);
```

All we need to change is that **size is now a StateInt.**



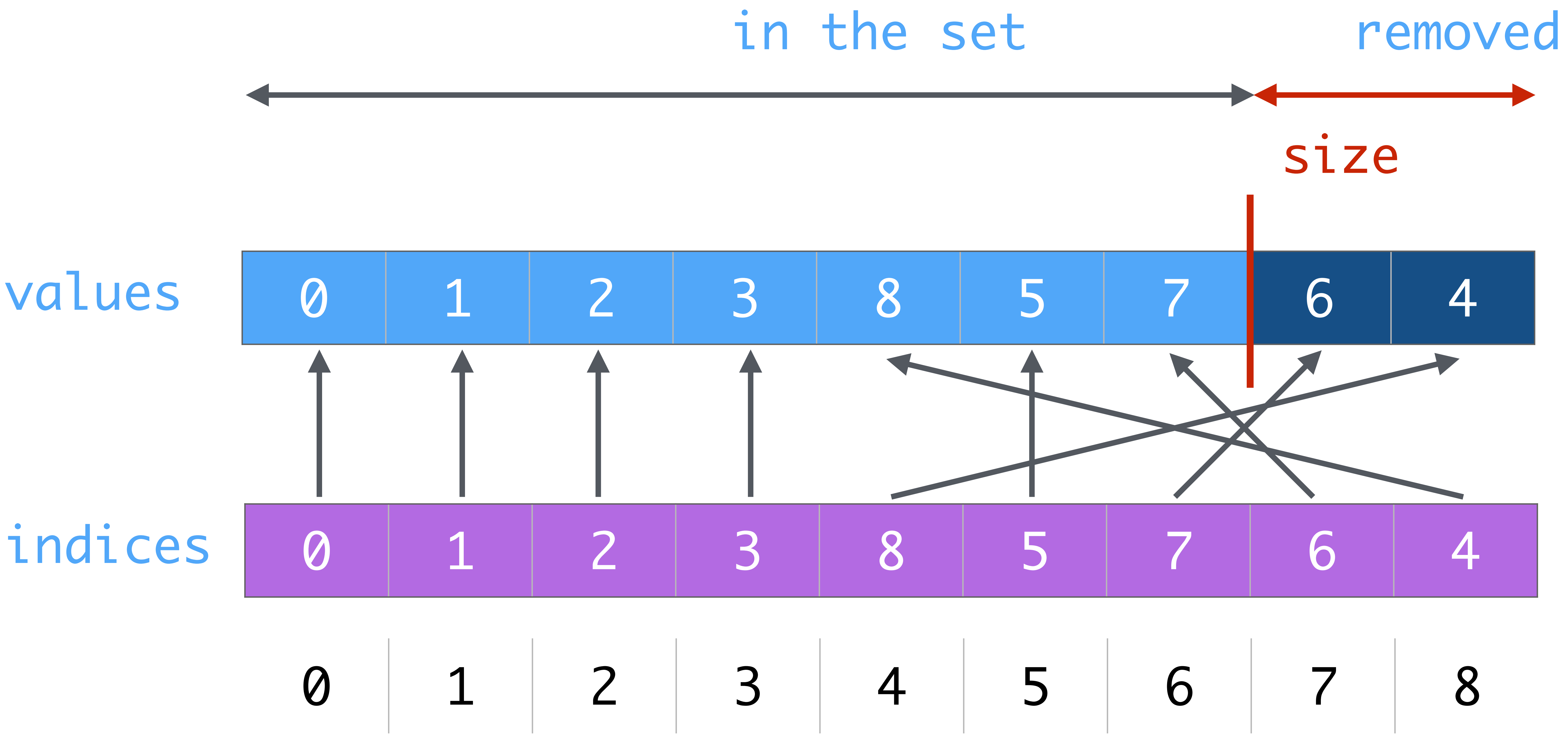
Removal Operation

```
StateManager sm;
StateSparseSet set = sm.makeSparseSet(9);
sm.saveState();
set.remove(4);
set.remove(6);
```



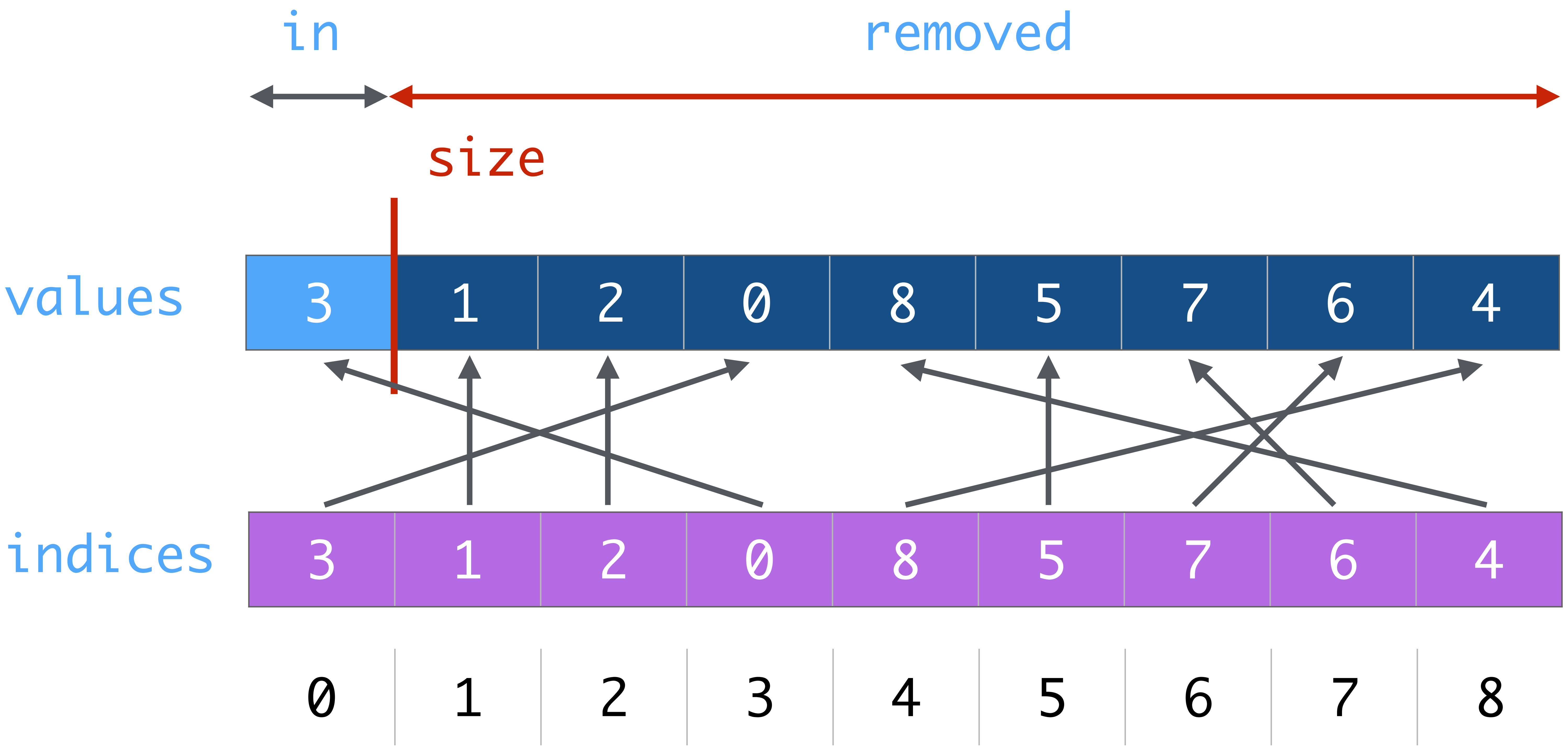
Removal Operation

```
StateManager sm;
StateSparseSet set = sm.makeSparseSet(9);
sm.saveState();
set.remove(4);
set.remove(6);
sm.saveState();
set.removeAllBut(3);
```



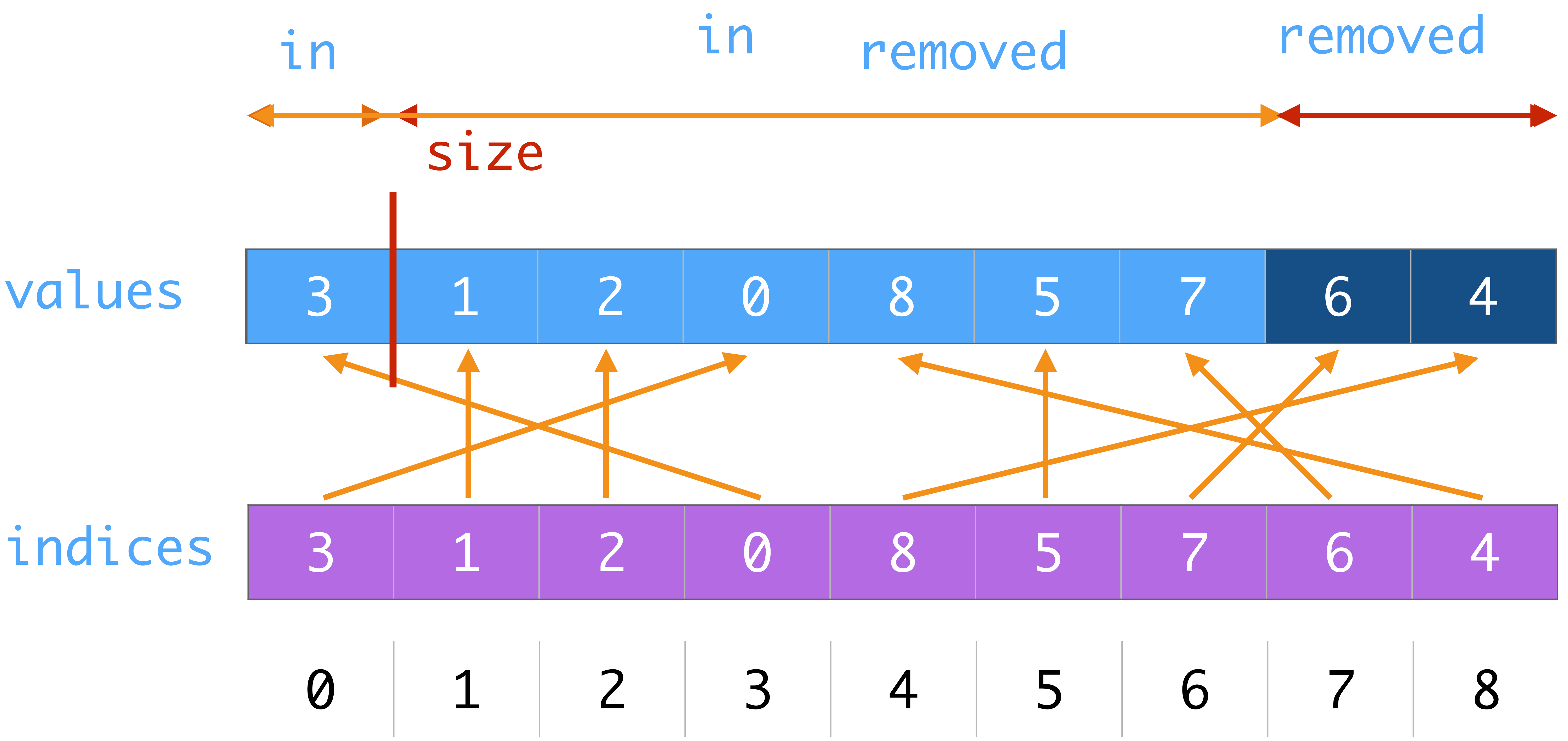
Removal Operation

```
StateManager sm;
StateSparseSet set = sm.makeSparseSet(9);
sm.saveState();
set.remove(4);
set.remove(6);
sm.saveState();
set.removeAllBut(3);
sm.restoreState(); // {0,1,2,3,5,7,8}
```



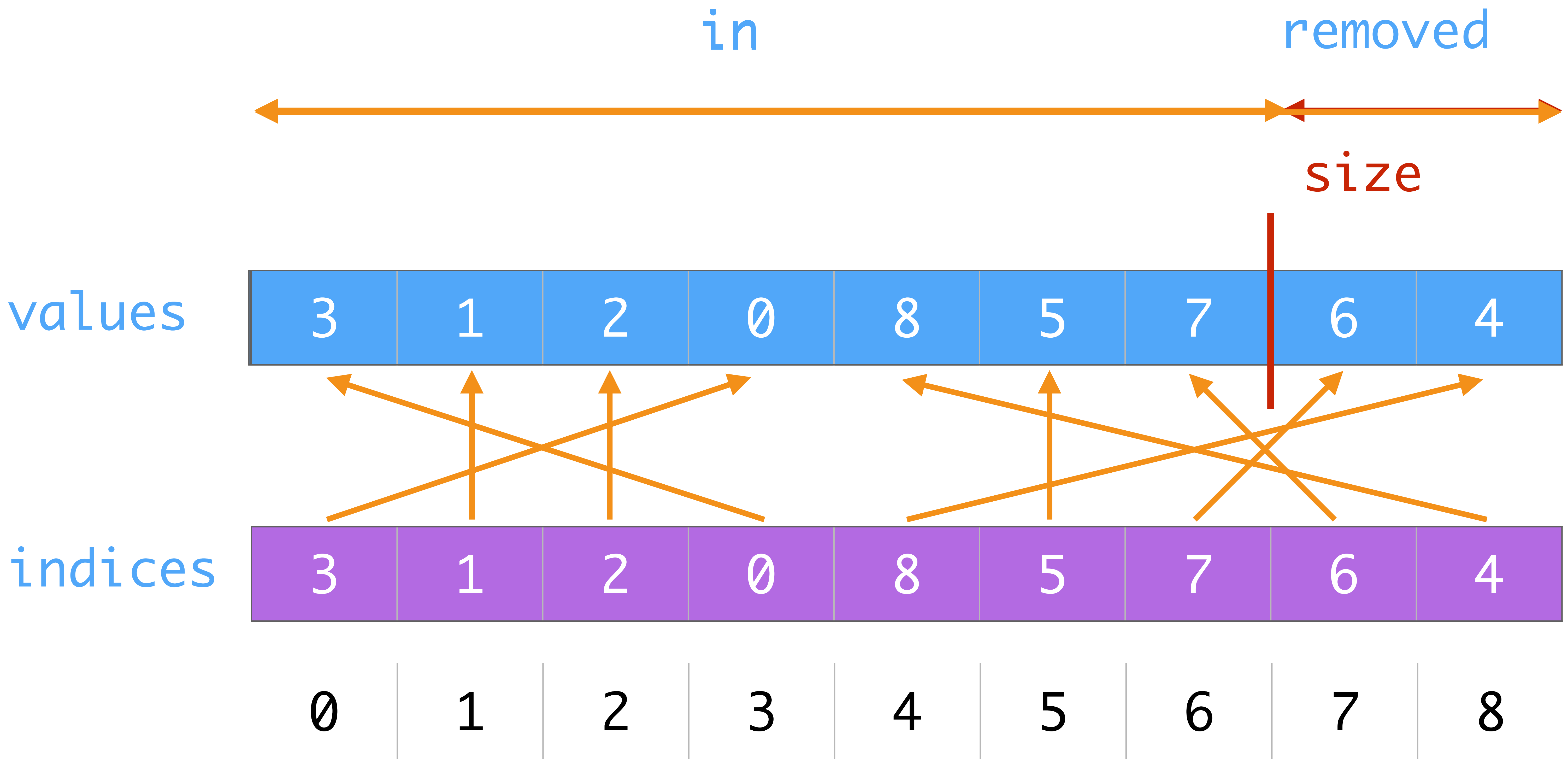
Removal Operation

```
StateManager sm;
StateSparseSet set = sm.makeSparseSet(9); : 9
sm.saveState();
set.remove(4);
set.remove(6);
sm.saveState();
set.removeAllBut(3);
sm.restoreState(); // {0,1,2,3,5,7,8}
```



Removal Operation

```
StateManager sm;
StateSparseSet set = sm.makeSparseSet(9); 9
sm.saveState();
set.remove(4);
set.remove(6);
sm.saveState();
set.removeAllBut(3);
sm.restoreState(); // {0,1,2,3,5,7,8}
sm.restoreState(); // {0..8}
```



Constraint Programming

Posting constraints during search
= reversible operations

A Stateful Container

- Another example of a “backtrack”-capable stateful object:
  - A stack of objects:
    - Only push objects onto the stack.
    - On backtrack (restore), the pushed objects should disappear.
  - Beware:
    - That does not work for popping objects!
- Usage scenario:
  - Stacks of constraints held for a variable (e.g., onDomain):
    - Only add constraints for the variable during the search.
    - On backtrack, the constraints are removed.

The StateStack

- Simple idea:
  - Maintain an ArrayList.
  - Maintain its size as a StateInt.
  - API:
    - Pushing adds at the end of the list and increases the size.
    - We never pop.
    - It pops automatically on backtrack!

Implementation

MiniCP

```
package minicp.state;
import java.util.ArrayList;
public class StateStack<E> {
    private StateInt size;
    private ArrayList<E> stack;

    public StateStack(StateManager sm) {
        size = sm.makeStateInt(0);
        stack = new ArrayList<E>();
    }

    public void push(E elem) {
        int s = size.value();
        if (stack.size() > s) stack.set(s, elem);
        else stack.add(elem);
        size.increment();
    }
    public int size()          { return size.value();}
    public E get(int index)  { return stack.get(index);}
}
```

Set up the state

Set up a StateInt @ 0

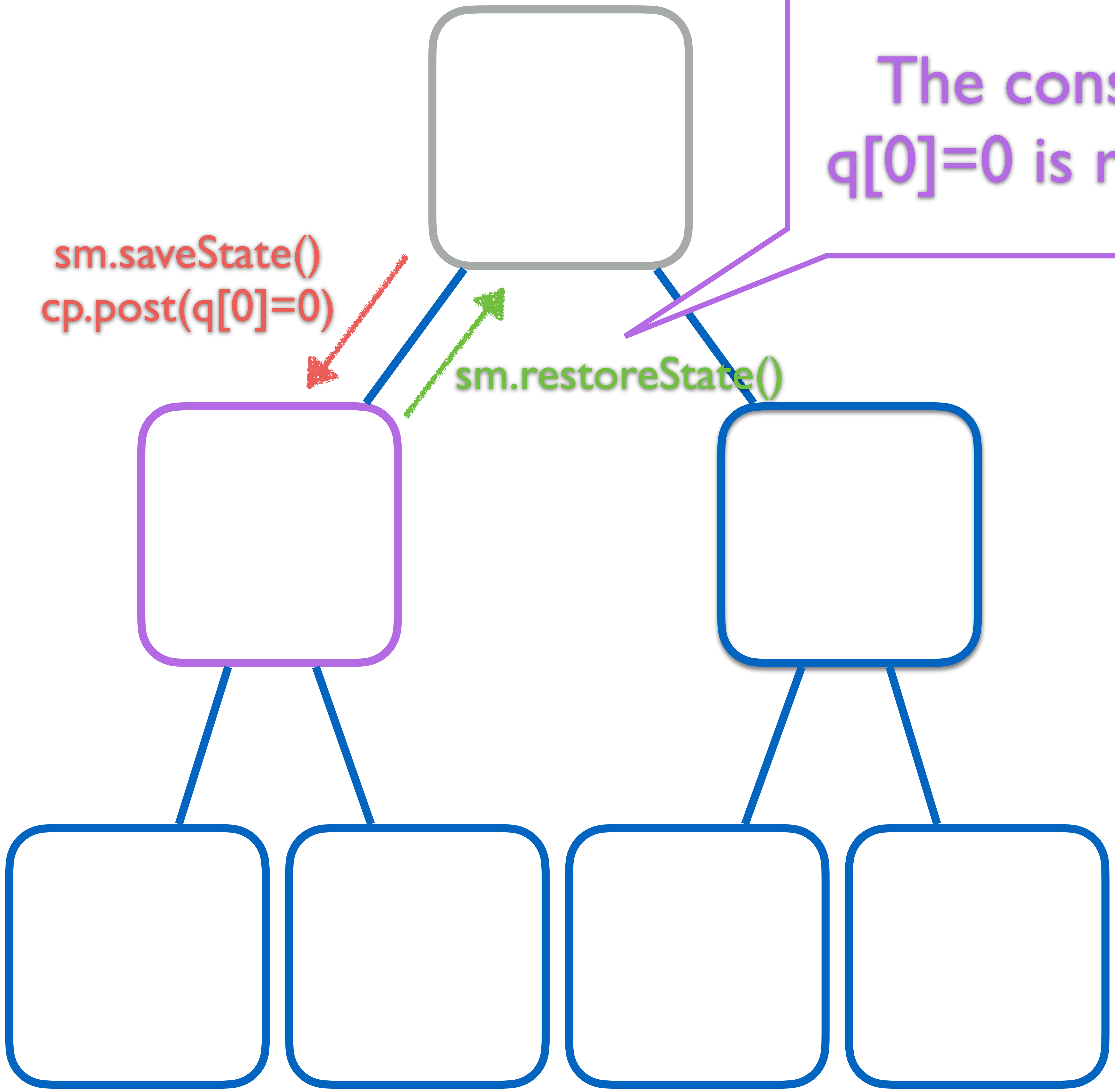
Update at the right offset

Update the size

Posting a Constraint is a Reversible Operation

```
public class IntVarImpl implements IntVar {
    private Solver cp;
    private IntDomain domain;
    private StateStack<Constraint> onDomain;
    private StateStack<Constraint> onFix;
    private StateStack<Constraint> onBound;
}
```

Because the hook-up mechanism is reversible.



The constraint q[0]=0 is removed.