To be able to pass seed values to a particular variable selection in tie-breaking, the `tiebreak` function is also overloaded for the class `VarBranchOptions`. For example,

```
VarBranchOptions o;
branch(home, x, tiebreak(INT_VAR_SIZE_MIN, INT_VAR_RND),
                INT_VAL_MIN,
                tiebreak(o, VarBranchOptions::time()));
```

passes a seed value based on the current time to the random variable selection used for tie-breaking. The same can also be achieved by

```
branch(home, x, tiebreak(INT_VAR_SIZE_MIN, INT_VAR_RND),
                INT_VAL_MIN,
                tiebreak(VarBranchOptions:def,
                         VarBranchOptions::time()));
```

9.1.4 Using branch filter functions

By default, a variable-value branching continues to branch until all variables passed to the branching are assigned. This behavior can be changed by using a *branch filter function*.

A branch filter function is called during branching for each variable to be branched on. If the filter function returns **`true`**, the variable is considered for branching. Otherwise, the variable is simply ignored.

A branch filter function can be specified when a branching is posted. The filter function to be used by a branching can be passed by creating an object of class `VarBranchOptions` (see Generic branching support) and passing that object to the `branch()` function.

A branch filter function is of type `BranchFilter` (see Generic branching support) defined as

```
typedef bool (*BranchFilter)(const Space& home, int i, const Var& y);
```

That is, a branch filter function takes the `home` space and the position `i` of the variable `y` as argument. The position `i` refers to the position of the variable `y` in the array of variables used for posting the branching.

Assume, for example, that we want to branch only on variables from a variable array `x` for branching with a domain size of at least `4`. Consider the sketch of a model shown in Figure 9.3.

The branch filter function can be defined as a static member function of the class `Model` as follows:

DEFINE FILTER FUNCTION ≡
```
static bool filter(const Space& home, int i, const Var& y) {
  return static_cast<const IntVar&>(y).size >= 4;
}
```

BRANCH FILTER FUNCTION SKETCH ≡
```
class Model : public Space {
protected:
  IntVarArray x;
public:
  Model(void) : ··· {
    ···
    ▶ POST BRANCHING
  }
  ···
  ▶ DEFINE FILTER FUNCTION
};
```

Figure 9.3: Model sketch for branch filter function

Note that the variable y must first be cast to the variable type used for branching (`IntVar` in our case).

Specifying that the branching should use the filter function is done as follows:

POST BRANCHING ≡
```
VarBranchOptions o(&filter);
branch(home, x, ···, ···, o);
```

One could also use the member `bf` of a variable branch option to set a branch filter function:

```
VarBranchOptions o;
o.bf = &filter;
branch(home, x, ···, ···, o);
```

In many cases it can be more convenient to perform the actual filtering in a **`const`** member function, say `filter()`, that has access to all members of the script `Model`. Then, the `filter()` member function can be called by some other static member function:

```
bool filter(int i, const IntVar& y) const {
  ···
}
static bool trampoline(const Space& home, int i, const Var& y) {
  return static_cast<const Model&>(home)
           .filter(i,static_cast<const IntVar&>(y));
}
```

where the function `trampoline()` now serves as branch filter function.

`INT_ASSIGN_MIN`	smallest value
`INT_ASSIGN_MED`	median value (rounding downwards)
`INT_ASSIGN_MAX`	maximum value
`INT_ASSIGN_RND`	random value

`SET_ASSIGN_MIN_INC`	include smallest element
`SET_ASSIGN_MIN_EXC`	exclude smallest element
`SET_ASSIGN_MED_INC`	include median element (rounding downwards)
`SET_ASSIGN_MED_EXC`	exclude median element (rounding downwards)
`SET_ASSIGN_MAX_INC`	include largest element
`SET_ASSIGN_MAX_EXC`	exclude largest element
`SET_ASSIGN_RND_INC`	include random element
`SET_ASSIGN_RND_EXC`	exclude random element

Figure 9.4: Value selection for assigning variables

Branch filter functions and tie-breaking. Tie-breaking requires that several objects of class `VarBranchOptions` are combined via the `tiebreak()` function (see Section 9.1.3). However, the branch filter function (if defined) is always taken from the first object. That is, when defining option objects by

```
VarBranchOptions o1(&f1), o2(&f2);
```

and passing them by

```
branch(home, x, tiebreak(···, ···), ···,
                tiebreak(o1,o2));
```

the branch filter function `f1()` is used.

9.1.5 Assigning integer and set variables

A special variant of branching is *assigning* variables: for a not yet assigned variable the branching creates a single alternative which assigns the variable a value. The effect of assigning is that assignment is interleaved with constraint propagation. That is, after an assignment has been done, the next assignment will be done only after the effect of the previous assignment has been propagated.

For example, the next code fragment assigns all integer variables in `x` their smallest possible value:

```
assign(home, x, INT_ASSIGN_MIN);
```

The strategy to select the value to be assigned is defined by a value of type `IntAssign` (see Assigning) for integer and Boolean variables and by a value of type `SetAssign` (see