Implementing QuantLib

Luigi Ballabio

7 The tree framework

Rollin’, rollin’, rollin

TOGETHER WITH Monte Carlo simulations, trees are among the most commonly used tools in quantitative finance. As usual, the dual challenge for a framework is to implement a number of reusable (and possibly composable) pieces and to provide customization hooks for injecting new behavior. The QuantLib tree framework has gone through a few revisions, and the current version is a combination of object-oriented and generic programming that does the job without losing too much performance in the process.

7.1. The `Lattice` and `DiscretizedAsset` classes

The two main classes of the framework are the `Lattice` and `DiscretizedAsset` classes, shown in listing 7.1 and 7.2 respectively. In our intentions, the `Lattice` class was to model the generic concept of a discrete lattice, which might have been a tree as well as a finite-difference grid. This never happened; the finite-difference framework went its separate way and is unlikely to come back any time soon. However, the initial design helped keeping the `Lattice` class clean: to this day, it contains almost no implementation details and is not tied to trees.

Its constructor takes a `TimeGrid` instance and stores it (its only concession to implementation inheritance, together with an inspector that returns the time grid). All other methods are pure virtual. The `initialize` method must set up a discretized asset so that it can be put on the lattice at a given time;[1] the `rollback` and `partialRollback` methods roll the asset backwards in time on the lattice down to the desired time (with a difference I’ll explain later); and the `presentValue` method returns what its name says.

Finally, the `grid` method returns the values of the discretized quantity underlying the lattice. This is a bit of a smell. The information was required in other parts of the library, and we didn’t have any better solution. However, this method has obvious shortcomings. On the one hand, it constrains the return type, which either

[1] I do realize this is mostly hand-waving right now. It will become clear as soon as we get to a concrete lattice.

Listing 7.1: Interface of the `Lattice` class.

```
class Lattice {
  public:
    Lattice(const TimeGrid& timeGrid) : t_(timeGrid) {}
    virtual ~Lattice() {}

    const TimeGrid& timeGrid() const { return t_; }

    virtual void initialize(DiscretizedAsset&,
                            Time time) const = 0;
    virtual void rollback(DiscretizedAsset&,
                          Time to) const = 0;
    virtual void partialRollback(DiscretizedAsset&,
                                 Time to) const = 0;
    virtual Real presentValue(DiscretizedAsset&) const = 0;

    virtual Disposable<Array> grid(Time) const = 0;
  protected:
    TimeGrid t_;
};
```

leaks implementation or forces a type conversion; and on the other hand, it simply makes no sense when the lattice has more than one factor, since the grid should be a matrix or a cube in that case. In fact, two-factor lattices implement it by having it throw an exception. All in all, this method is up for some serious improvement in future versions of the library.

The `DiscretizedAsset` class is the base class for the other side of the tree framework—the Costello to `Lattice`'s Abbott, as it were. It models an asset that can be priced on a lattice: it works hand in hand with the `Lattice` class to provide generic behavior, and has hooks that derived classes can use to add behavior specific to the instrument they implement.

As can be seen from the listing, it's not nearly as abstract as the `Lattice` class. Most of its methods are concrete, with a few virtual ones that use the Template Method pattern to inject behavior.

Its constructor takes no argument, but initializes a couple of internal variables. The main inspectors return the data comprising its state, namely, the time t of the lattice nodes currently occupied by the asset and its values on the same nodes; both inspectors give both read and write access to the data to allow the lattice implementation to modify them. A read-only inspector returns the lattice on which the asset is being priced.

Listing 7.2: Implementation of the DiscretizedAsset class.

```
class DiscretizedAsset {
  public:
    DiscretizedAsset()
    : latestPreAdjustment_(QL_MAX_REAL),
      latestPostAdjustment_(QL_MAX_REAL) {}
    virtual ~DiscretizedAsset() {}

    Time time() const { return time_; }
    Time& time() { return time_; }
    const Array& values() const { return values_; }
    Array& values() { return values_; }
    const shared_ptr<Lattice>& method() const {
        return method_;
    }

    void initialize(const shared_ptr<Lattice>&,
                    Time t);
    void rollback(Time to);
    void partialRollback(Time to);
    Real presentValue();

    virtual void reset(Size size) = 0;
    void preAdjustValues();
    void postAdjustValues();
    void adjustValues();

    virtual std::vector<Time> mandatoryTimes() const = 0;

  protected:
    bool isOnTime(Time t) const;
    virtual void preAdjustValuesImpl() {}
    virtual void postAdjustValuesImpl() {}

    Time time_;
    Time latestPreAdjustment_, latestPostAdjustment_;
    Array values_;

  private:
    shared_ptr<Lattice> method_;
};
```

Listing 7.2 (continued).

```
void DiscretizedAsset::initialize(
                         const shared_ptr<Lattice>& method,
                         Time t) {
    method_ = method;
    method_->initialize(*this, t);
}

void DiscretizedAsset::rollback(Time to) {
    method_->rollback(*this, to);
}

void DiscretizedAsset::partialRollback(Time to) {
    method_->partialRollback(*this, to);
}

Real DiscretizedAsset::presentValue() {
    return method_->presentValue(*this);
}

void DiscretizedAsset::preAdjustValues() {
    if (!close_enough(time(),latestPreAdjustment_)) {
        preAdjustValuesImpl();
        latestPreAdjustment_ = time();
    }
}

void DiscretizedAsset::postAdjustValues() {
    if (!close_enough(time(),latestPostAdjustment_)) {
        postAdjustValuesImpl();
        latestPostAdjustment_ = time();
    }
}

void DiscretizedAsset::adjustValues() {
    preAdjustValues();
    postAdjustValues();
}

bool DiscretizedAsset::isOnTime(Time t) const {
    const TimeGrid& grid = method()->timeGrid();
    return close_enough(grid[grid.index(t)],time());
}
```

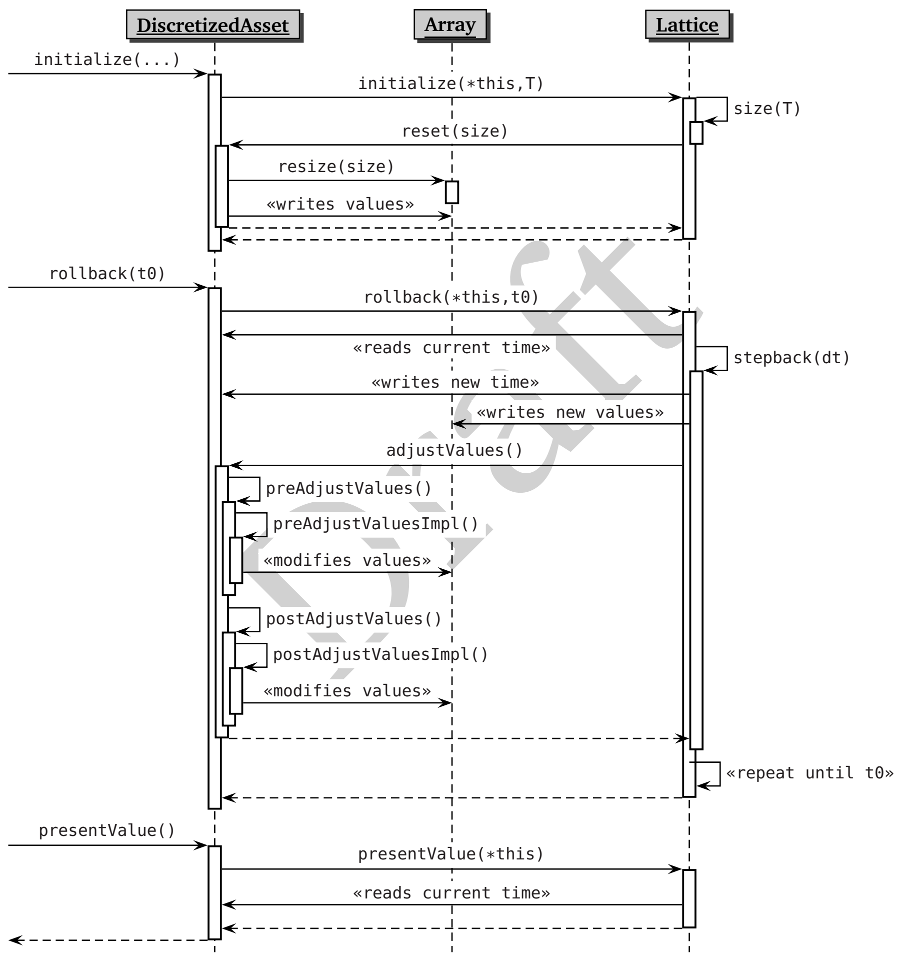

Figure 7.1: Sequence diagram of the interplay between the `DiscretizedAsset` and `Lattice` classes.

The next bunch of methods implements the common behavior that is inherited by derived classes and provide the interface to be called by client code. The body of a tree-based engine will usually contain something like the following after instantiating the tree and the discretized asset:

```
asset.initialize(lattice, T);
asset.rollback(t0);
results_.value = asset.presentValue();
```

The `initialize` method stores the lattice that will be used for pricing and sets the initial values of the asset (or rather its final values, since the time `T` passed at initialization is the maturity time); the `rollback` method rolls the asset backwards on the lattice until the time `t0`; and the `presentValue` method extracts the value of the asset as a single number.

The three method calls above seem simple, but their implementation triggers a good deal of behavior in both the asset and the lattice and involves most of the other methods of `DiscretizedAsset` as well as those of `Lattice`. The interplay between the two classes is not nearly as funny as *Who's on First*, but it's almost as complex to follow; thus, you might want to refer to the sequence diagram shown in figure 7.1.

The `initialize` method sets the calculation up by placing the asset on the lattice at its maturity time (the asset's, I mean) and preparing it for rollback. This means the one the one hand, the vector holding the asset values on each lattice node must be dimensioned correctly; and on the other hand, that it must be filled with the correct values. Like most of the `DiscretizedAsset` methods, `initialize` does this by delegating part of the actual work to the passed lattice; after storing it in the corresponding data member, it simply calls the lattice's `initialize` method passing the maturity time and the asset itself.

Now, the `Lattice` class doesn't implement the latter, which is left as purely virtual; but any sensible implementation in derived classes will do the dance shown in the sequence diagram. It might perform some housekeeping step of its own, not shown here; but first, it will determine the size of the lattice at the maturity time (that is, the number of nodes) probably by calling a corresponding method; then, it will store the time into the asset[2] and pass the size to the asset's `reset` method. The latter, implemented in derived classes, will resize the vector of values accordingly and fill it with instrument-specific values (for instance, a bond might store its redemption and the final coupon, if any).

Next is the `rollback` method. Again, it calls the corresponding method in the `Lattice` class, which performs the heavy-machinery work of stepping through the tree (or whatever kind of lattice it models). It reads the current time from the asset, so it knows on what nodes it is currently sitting; then, a close interplay begins.

[2]As you remember, the asset provides read/write access to its state.

The point is, the lattice can't simply roll the asset back to the desired time, since there might be all kind of events occurring: coupon payments, exercises, you name it. Therefore, it just rolls it back one short step, modifies the asset values accordingly—which includes both combining nodes and discounting—and then pauses to ask the asset if there's anything that needs to be done; that is, to call the asset's `adjustValues` method.

The adjustment is done in two steps, calling first the `preAdjustValues` and then the `postAdjustValue` method. This is done so that other assets have a chance to perform their own adjustment between the two; I'll show an example of this later on. Each of the two methods performs a bit of housekeeping[3] and then calls a virtual method (`preAdjustValuesImpl` or `postAdjustValueImpl`, respectively) which makes the instrument-specific adjustments.

When this is done, the ball is back in the lattice's field. The lattice rolls back another short step, and the whole thing repeats again and again until the assets reaches the required time.

Finally, the `presentValue` method returns, well, the present value of the asset. It is meant to be called by client code after rolling the asset back to the earliest interesting time—that is, to the time for which further rolling back to today's date only involves discounting and not adjustments of any kind: e.g., it might be the earliest exercise time of an option or the first coupon time of a bond. As usual, the asset delegates the calculation by passing itself to the lattice's `presentValue` method; a given lattice implementation might simply roll the asset back to the present time (if it isn't already there, of course) and read its value from the root node, whereas another lattice might be able to take a shortcut and calculate the present value as a function of the values on the nodes at the current time.

As usual, there's room for improvement. For instance, some of the steps are left as an implicit requirement; for instance, the fact that the lattice's `initialize` method should call the asset's `reset` method, or that `reset` must resize the array stored in the asset.[4] However, this is what we have at this time. Let's move on and build some assets.

7.1.1. Example: discretized bonds

Simple bonds (zero-coupon, fixed-rate and floating-rate) are simple enough to work as first examples but at the same time provide a range of features large enough for me to make a few points.

Well, maybe not the zero-coupon bond. It's probably the simplest possible asset (bar one with no payoff) so it's not that interesting on its own; however, it's going

[3]Namely, they store the time of the latest adjustment so that the asset is not adjusted twice at the same time; this might happen when composing assets, and would obviously wreak havoc on the results.

[4]In hindsight, the array should be resized in the lattice's `initialize` method before calling `reset`. This would minimize repetition, since we'll write many more assets than lattices.

Listing 7.3: Implementation of the `DiscretizedDiscountBond` class.

```
class DiscretizedDiscountBond : public DiscretizedAsset {
  public:
    DiscretizedDiscountBond() {}
    void reset(Size size) {
        values_ = Array(size, 1.0);
    }
    std::vector<Time> mandatoryTimes() const {
        return std::vector<Time>();
    }
};
```

to be useful as a helper class in the examples that follow, since it provides a way to estimate discount factors on the lattice.

Its implementation is shown in listing 7.3. It defines no mandatory times (because it will be happy to be initialized at any time you choose), it performs no adjustments (because nothing ever happens during its life), and its `reset` method simply fills each value in the array with one unit of currency. Thus, if you initialize an instance of his class at a time T and roll it back on a lattice until an earlier time t, the array values will then equal the discount factors from T to t as seen from the corresponding lattice nodes.

Things start to get more interesting when we turn to fixed-rate bonds. QuantLib doesn't provide discretized fixed-rate bonds at this time; listing 7.4 shows a simple implementation which works as an example, but should still be improved to enter the library.

The constructor takes and stores a vector of times holding the payment schedule, the vector of the coupon amounts, and the amount of the redemption. Note that the type of the arguments is different from what the corresponding instrument class is likely to store (say, a vector of `CashFlow` instances); this implies that the pricing engine will have to perform some conversion work before instantiating the discretized asset.

The presence of a payment schedule implies that, unlike the zero-coupon bond above, this bond cannot be instantiated at just any time; and in fact, this class implements the `mandatoryTimes` method by returning the vector of the payment times. Rollback on the lattice will have to stop at each such time, and initialization will have to be performed at the maturity time of the bond, i.e., the latest of the returned times. When one does so, the `reset` method will fill each value in the array with the redemption amount and then call the `adjustValues` method, which will take care of the final coupon.

Listing 7.4: Implementation of the `DiscretizedFixedRateBond` class.

```
class DiscretizedFixedRateBond : public DiscretizedAsset {
  public:
    DiscretizedFixedRateBond(const vector<Time>& paymentTimes,
                             const vector<Real>& coupons,
                             Real redemption)
    : paymentTimes_(paymentTimes), coupons_(coupons),
      redemption_(redemption) {}

    vector<Time> mandatoryTimes() const {
        return paymentTimes_;
    }
    void reset(Size size) {
        values_ = Array(size, redemption_);
        adjustValues();
    }
  private:
    void postAdjustValuesImpl() {
        for (Size i=0; i<paymentTimes_.size(); i++) {
            Time t = paymentTimes_[i];
            if (t >= 0.0 && isOnTime(t)) {
                addCoupon(i);
            }
        }
    }
    void addCoupon(Size i) {
        values_ += coupons_[i];
    }

    vector<Time> paymentTimes_;
    vector<Real> coupons_;
    Real redemption_;
};
```

In order to enable `adjustValues` to do so, this class overrides the virtual `postAdjustValuesImpl` method. (Why the *post-* version of the method and not the *pre-*, you say? Bear with me a few more pages: I need a bit more context to explain. All will become clear.) The method loops over the payment times and checks, by means of the probably poorly named `isOnTime` method, whether any of them equals the current asset time. If this is the case, we add the corresponding coupon amount to the asset values. For readability, the actual work is factored out in the `addCoupon` method. The coupon amounts will be automatically discounted as the asset is rolled back on the lattice.

Finally, let's turn to the floating-rate bond, shown in listing 7.5; this, too, is a simplified implementation.

The constructor takes and stores the vector of payment times, the vector of fixing times, and the notional of the bond; there are no coupon amounts, since they will be estimated during the calculation. For simplicity of implementation, we'll assume that the accrual time for the i-th coupon equals the time between its fixing time and its payment time.

The `mandatoryTimes` method returns the union of payment times and fixing times, since we'll need to work on both during the calculations. The `reset` method is similar to the one for fixed-rate bonds, and fills the array with the redemption value (which equals the notional of the bond) before calling `adjustValues`.

Adjustment is performed in the overridden `preAdjustValuesImpl` method. (Yes, the *pre-* version. Patience.) It loops over the fixing times, checks whether any of them equals the current time, and if so calls the `addCoupon` method.

Now, like the late Etta James in one of her hits, you'd be justified in shouting "Stop the wedding". Of course the coupon should be added at the payment date, right? Well, yes; but the problem is that we're going backwards in time. In general, at the payment date we don't have enough information to add the coupon; it can only be estimated based on the value of the rate at an earlier time that we haven't yet reached. Therefore, we have to keep rolling back on the lattice until we get to the fixing date, at which point we can calculate the coupon amount and add it to the bond. In this case, we'll have to take care ourselves of discounting from the payment date, since we passed that point already.

That's exactly what the `addCoupon` method does. First of all, it instantiates a discount bond at the payment time T and rolls it back to the current time, i.e., the fixing date t, so that its value equal at the j-th node the discount factors D_j between t and T. From those, we could estimate the floating rates r_j (since it must hold that $1 + r_j(T - t) = 1/D_j$) and then the coupon amounts; but with a bit of algebra, we can find a simpler calculation. The coupon amount C_j is given by $Nr_j(T - t)$, with N being the notional; and since the relation above tells us that $r_j(T - t) = 1/D_j - 1$, we can substitute that to find that $C = N(1/D_j - 1)$. Now, remember that we already rolled back to the fixing date, so if we add the amount

Listing 7.5: Implementation of the `DiscretizedFloatingRateBond` class.

```
class DiscretizedFloatingRateBond : public DiscretizedAsset {
  public:
    DiscretizedFloatingRateBond(const vector<Time>& paymentTimes,
                                const vector<Time>& fixingTimes,
                                Real notional)
    : paymentTimes_(paymentTimes), fixingTimes_(fixingTimes),
      notional_(notional) {}

    vector<Time> mandatoryTimes() const {
        vector<Time> times = paymentTimes_;
        std::copy(fixingTimes_.begin(), fixingTimes_.end(),
                  back_inserter(times));
        return times;
    }
    void reset(Size size) {
        values_ = Array(size, notional_);
        adjustValues();
    }
  private:
    void preAdjustValuesImpl() {
        for (Size i=0; i<fixingTimes_.size(); i++) {
            Time t = fixingTimes_[i];
            if (t >= 0.0 && isOnTime(t)) {
                addCoupon(i);
            }
        }
    }
    void addCoupon(Size i) {
        DiscretizedDiscountBond bond;
        bond.initialize(method(), paymentTimes_[i]);
        bond.rollback(time_);

        for (Size j=0; j<values_.size(); j++) {
             values_[j] += notional_ * (1.0 - bond.values()[j]);
        }
    }

    vector<Time> paymentTimes_;
    vector<Time> fixingTimes_;
    Real notional_;
};
```

here we also have to discount it because it won't be rolled back from the payment date. This means that we'll have to multiply it by D_j, and thus the amount to be added to the j-th value in the array is simply $C_j = N(1/D_j - 1)D_j = N(1 - D_j)$. The final expression is the one that can be seen in the implementation of addCoupon.

As you probably noted, the above hinges on the assumption that the accrual time equals the time between payment and fixing time. If this were not the case, the calculation would no longer simplify and we'd have to change the implementation; for instance, we might instantiate a first discount bond at the accrual end date to estimate the floating rate and the coupon amount, and a second one at the payment date to calculate the discount factors to be used when adding the coupon amount to the bond value. Of course, the increased accuracy would cause the performance to degrade since addCoupon would roll back two bonds, instead of one. You can choose either implementation based on your requirements.

7.1.2. Example: discretized option

Sorry to have kept you waiting, folks. Here is where I finally explain the pre- vs post-adjustment choice, after the previous example helped me put my ducks in a row. I'll do so by showing an example of an asset class (the DiscretizedOption class, shown in listing 7.6) that can be used to wrap an underlying asset and obtain an option to enter the same: for instance, it could take a swap and yield a swaption. The implementation shown here is a slightly simplified version of the one provided by QuantLib, since it assumes a Bermudan exercise (or European, if one passes a single exercise time). Like the implementation in the library, it also assumes that there's no premium to pay in order to enter the underlying deal.

Onwards. The constructor takes and, as usual, stores the underlying asset and the exercise times; nothing to write much about.

The mandatoryTimes method takes the vector of times required by the underlying and adds the option's exercise times. Of course, this is done so that both the underlying and the option can be priced on the same lattice; the sequence of operations to get the option price will be something like:

```
underlying = shared_ptr<DiscretizedAsset>(...);
option = shared_ptr<DiscretizedAsset>(
             new DiscretizedOption(underlying, exerciseTimes));
grid = TimeGrid(..., option->mandatoryTimes());
lattice = shared_ptr<Lattice>(new SomeLattice(..., grid, ...));
underlying->initialize(lattice, T1);
option->initialize(lattice, T2);
option->rollback(t0);
NPV = option->presentValue();
```

in which, first, we instantiate both underlying and option and retrieve the mandatory times from the latter; then, we create the lattice and initialize both assets (usually at different times, e.g., the maturity date for a swap and the latest exercise

Listing 7.6: Implementation of the `DiscretizedOption` class.

```
class DiscretizedOption : public DiscretizedAsset {
  public:
    DiscretizedOption(
              const shared_ptr<DiscretizedAsset>& underlying,
              const vector<Time>& exerciseTimes)
    : underlying_(underlying), exerciseTimes_(exerciseTimes) {}

    vector<Time> mandatoryTimes() const {
        vector<Time> times = underlying_->mandatoryTimes();
        for (Size i=0; i<exerciseTimes_.size(); ++i)
            if (exerciseTimes_[i] >= 0.0)
                times.push_back(exerciseTimes_[i]);
        return times;
    }
    void reset(Size size) {
        QL_REQUIRE(method() == underlying_->method(),
                   "option and underlying were initialized on "
                   "different lattices");
        values_ = Array(size, 0.0);
        adjustValues();
    }
  private:
    void postAdjustValuesImpl() {
        underlying_->partialRollback(time());
        underlying_->preAdjustValues();
        for (Size i=0; i<exerciseTimes_.size(); i++) {
            Time t = exerciseTimes_[i];
            if (t >= 0.0 && isOnTime(t))
                applyExerciseCondition();
        }
        underlying_->postAdjustValues();
    }
    void DiscretizedOption::applyExerciseCondition() {
        for (Size i=0; i<values_.size(); i++)
            values_[i] = std::max(underlying_->values()[i],
                                  values_[i]);
    }

    shared_ptr<DiscretizedAsset> underlying_;
    vector<Time> exerciseTimes_;
};
```

date for the corresponding swaption); and finally, we roll back the option and get its price. As we'll see in a minute, the option also takes care of rolling the underlying back as needed.

Back to the class implementation. The `reset` method performs the sanity check that underlying and option were initialized on the same lattice, fills the values with zeroes (what you end up with if you don't exercise), and then calls `adjustValues` to take care of a possible exercise.

Which brings us to the crux of the matter, i.e., the `postAdjustValuesImpl` method. The idea is simple enough: if we're on an exercise time, we check whether keeping the option is worth more than entering the underlying asset. To do so, we roll the underlying asset back to the current time, compare values at each node, and set the option value to the maximum of the two; this latest part is abstracted out in the `applyExerciseCondition` method.

The tricky part of the problem is that the underlying might need to perform an adjustment of its own when rolled back to the current time. Should this be done before or after the option looks at the underlying values?

It depends on the particular adjustment. Let's look at the bonds in the previous example. If the underlying is a discretized fixed-rate bond, and if the current time is one of its payment times, it needs to adjust its values by adding a coupon. This coupon, though, is being paid now and thus is no longer part of the asset if we exercise and enter it. Therefore, the decision to exercise must be based on the bond value without the coupon; i.e., we must call the `applyExerciseCondition` method before adjusting the underlying.

The discretized floating-rate bond is another story. It adjusts the values if the current time is one of its fixing times; but in this case the corresponding coupon is just starting and will be paid at the end of the period, and so must be added to the bond value before we decide about exercise. Thus, the conclusion is the opposite: we must call `applyExerciseCondition` after adjusting the underlying.

What should the option do? It can't distinguish between the two cases, since it doesn't know the specific behavior of the asset it was passed; therefore, it lets the underlying itself sort it out. First, it rolls the underlying back to the current time, but without performing the final adjustment (that's what the `partialRollback` method does); instead, it calls the underlying's `preAdjustValues` method. Then, if we're on an exercise time, it performs its own adjustment; and finally, it calls the underlying's `postAdjustValues` method.

This is the reason the `DiscretizedAsset` class has both a `preAdjustValues` and a `postAdjustValues` method; they're there so that, in case of asset composition, the underlying can choose on which side of the fence to be when some other adjustment (such as an exercise) is performed at the same time. In the case of our previous example, the fixed-rate bond will add its coupon in `postAdjustValues` and have it excluded from the future bond value, while the floating-rate bond will add its coupon in `preAdjustValues` and have it included.

Unfortunately, this solution is not very robust. For instance, if the exercise dates were a week or two before the coupon dates (as is often the case) the option would break for fixed-rate coupons, since it would have no way to stop them from being added before the adjustment. The problem can be solved: in the library, this is done for discretized interest-rate swaps by adding fixed-rate coupons on their start dates, much in the same way as floating-rate coupons. Another way to fix the issue would be to roll the underlying back only to the date when it's actually entered, then to make a copy of it and roll the copy back to the exercise date without performing any adjustment. Both solutions are somewhat clumsy at this time; it would be better if QuantLib provided some means to do it more naturally.

7.2. Trees and tree-based lattices

In order to use the assets I just described, we need a lattice. The first question we have to face is at the very beginning of its design: should the tree itself be the lattice, or should we have a separate lattice class using a tree?

Even if the first alternative can be argued (a tree can very well be seen as a kind of lattice, so an *is-a* relationship would be justified), the library currently implements the second. I'm afraid the reasons for the choice are lost in time; one might be that having a class for the lattice and one for the tree allows us to separate different concerns. The tree has the responsibility for maintaining the structure and the probability transitions between nodes; the lattice uses the tree structure to roll the asset back and adds some additional features such as discounting.

7.2.1. The `Tree` class template

As I wrote above, a tree provides information about its structure. It can be described by the number of nodes at each level (each level corresponding to a different time), the nodes that can be reached from each other node at the previous level, and the probabilities for each such transition.

The interface chosen to represent this information (shown in listing 7.7) was an index-based one. Nodes were not represented as instances of a `Node` class or structure; instead, the tree was described—not implemented, mind you—as a series of vectors of increasing size, and the nodes were identified by their indexes into the vectors. This gives more flexibility in implementing different trees; one tree class might store the actual vectors of nodes, while another might just calculate the required indexes as needed. In the library, trinomial trees are implemented in the first way and binomial trees in the second. I'll describe both later on.

The methods shown in the interface fully describe the tree structure. The `columns` method returns the number of levels in the tree; evidently, the original designer visualized the time axis as going from left to right, with the tree growing in the same direction. If we were to change the interface, I'd probably go for a more neutral name, such as `size`.

Listing 7.7: Original interface of the Tree class.

```
class Tree {
  public:
    Tree(Size columns);
    virtual ~Tree();
    Size columns() const;
    virtual Size size(Size i) const = 0;
    virtual Real underlying(Size i, Size j) const = 0;
    virtual Size branches() const = 0;
    virtual Size descendant(Size i, Size j,
                            Size branch) const = 0;
    virtual Real probability(Size i, Size index,
                             Size branch) const = 0;
};
```

Listing 7.8: Implementation of the Tree class template.

```
template <class T>
class Tree : public CuriouslyRecurringTemplate<T> {
  public:
    Tree(Size columns) : columns_(columns) {}
    Size columns() const { return columns_; }
  private:
    Size columns_;
};
```

The size name was actually taken for another method, which returns the number of nodes at the i-th level of the tree; the underlying method takes two indexes i and j and returns the value of the variable underlying the tree at the j-th node of the i-th level.

The remaining methods deal with branching. The branches method returns the number of branches for each node (2 for binomial trees, 3 for trinomial ones and so on). It takes no arguments, which means that we're assuming such a number to be constant; we'll have no trees with a variable number of branches. The descendant identifies the nodes towards which a given node is branching: it takes the index i of the level, the index j of the node and the index of a branch, and returns the index of the corresponding node on level i+1 as exemplified in figure 7.2. Finally, the probability method takes the same arguments and returns the probability to follow a given branch.

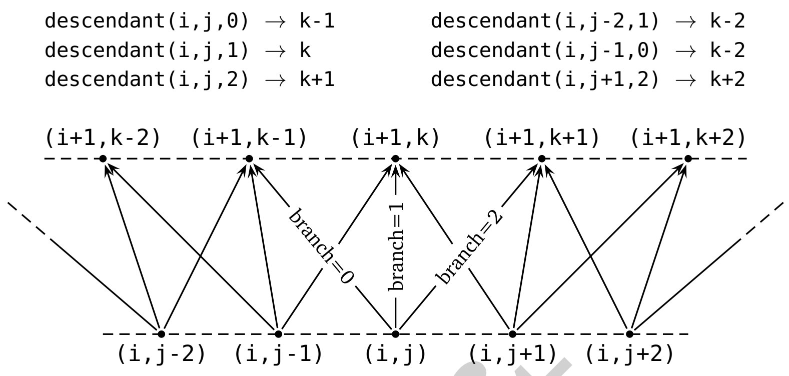

Figure 7.2: Schematics of a sample tree.

The original implementation actually sported a base `Tree` class as shown in listing 7.7. It worked, but it committed what amounts to a mortal sin in numerical code: it declared as virtual some methods (such as `descendant` or `probability`) that were meant to be called thousands of times inside tight nested loops.[5] After a while, we started to notice that small glaciers were disappearing in the time it took us to calibrate a short-rate model.

Eventually, this led to a reimplementation. We used the Curiously Recurring Template Pattern (CRTP) to replace run-time with compile-time polymorphism (see the aside on the following page if you need a refresher) which led to the `Tree` class template shown in listing 7.8. The virtual methods are gone, and only the non-virtual `columns` method remains; the machinery for the pattern is provided by the `CuriouslyRecurringTemplate` class template, which avoids repetition of boilerplate code by implementing the `const` and non-`const` versions of the `impl` method used in the pattern.

This was not a redesign; the existing class hierarchy was left unchanged. Switching to CRTP yielded slightly more complex code, as you'll see when I describe binomial and trinomial trees. However, it paid off in speed: with the new implementation, we saw the performance increase up to 10x. Seeing the results, I didn't stop to ask myself many question (yes, I'm the guilty party here) and I missed an opportunity to simplify the code. With a cooler head, I might have noticed that the base tree classes don't call methods defined in derived classes, so I could have simplified the code further by dropping CRTP for simple templates.[6] Yet another thing to remember when we decide to redesign parts of the library.

[5]If you're wondering why this is bad, see for instance [28] for a short explanation—along with a lot more information on using C++ for numerical work.

[6]If a new branch of the hierarchy were to need CRTP, it could be added locally to that branch.

Aside: curiouser and curiouser.

The Curiously Recurring Template Pattern (CRTP) was named and popularized by James Coplien in 1995 [9] but it predates his description. It is a way to achieve a static version of the Template Method pattern (see [14], of course), in which a base class implements common behavior and provides hooks for customization by calling methods that will be implemented in derived classes.
The idea is the following:

```
template <T>
class Base {
  public:
    T& impl() {
        return static_cast<T&>(*this);
    }
    void foo() {
        ...
        this->impl().bar();
        ...
    }
};

class Derived : public Base<Derived> {
  public:
    void bar() { ... }
};
```

Now if we instantiate a `Derived` and call its `foo` method, it will in turn call the correct `bar` implementation: that is, the one defined in `Derived` (note that `Base` doesn't declare `bar` at all).
We can see how this work by working out the expansion made by the compiler. The `Derived` class doesn't define a `foo` method, so it inherits it from the its base class; that is, `Base<Derived>`. The `foo` method calls `impl`, which casts `*this` to a reference to the template argument `T`, i.e., `Derived` itself. We called the method on a `Derived` instance to begin with, so the cast succeeds and we're left with a reference to our object that has the correct type. At this point, the compiler resolves the call to `bar` on the reference to `Derived::bar` and thus the correct method is called. All this happens at compile time, and enables the compiler to inline the call to `bar` and perform any optimization it sees fit.

Why go through all this, and not just call `bar` from `foo` instead? Well, it just wouldn't work. If we didn't declare `bar` into `Base`, the compiler wouldn't know what method we're talking about; and if we did declare it, the compiler would always resolve the call to the `Base` version of the method. CRTP gives us a way out: we can define the method in the derived class and use the template argument to pass its type to the base class so that the static cast can close the circle.

Listing 7.9: Implementation of the `BinomialTree` class template.

```
template <class T>
class BinomialTree : public Tree<T> {
  public:
    enum Branches { branches = 2 };
    BinomialTree(
         const boost::shared_ptr<StochasticProcess1D>& process,
         Time end, Size steps)
    : Tree<T>(steps+1) {
        x0_ = process->x0();
        dt_ = end/steps;
        driftPerStep_ = process->drift(0.0, x0_) * dt_;
    }
    Size size(Size i) const {
        return i+1;
    }
    Size descendant(Size, Size index, Size branch) const {
        return index + branch;
    }
  protected:
    Real x0_, driftPerStep_;
    Time dt_;
};
```

7.2.2. Binomial and trinomial trees

As I mentioned earlier, a tree might implement its required interface by storing actual nodes or by just calculating their indexes on demand. The `BinomialTree` class template, shown in listing 7.9, takes the second approach. In a way, the nodes *are* the indexes; they don't exist as separate entities.

The class template models a rather strict subset of the possible binomial trees, that is, recombining trees with constant drift and diffusion for the underlying variable[7] and with constant time steps.

Even with these constraints, there's quite a few different possibilities for the construction of the tree, so most of the implementation is left to derived classes. This class acts as a base and provides the common facilities; first of all, a `branches` enumeration whose value is defined to be 2 and which replaces the corresponding method in listing 7.7. Nowadays, we'd use a `static const Size` data member for

[7]The library contains, in its "experimental" folder, an extension of this class to non-constant parameters. You're welcome to have a look at it and send us feedback. For illustration purposes, though, I'll stick to the constant version here.

the purpose; the enumeration is a historical artifact from a time when compilers were less standard-compliant.

The class constructor takes a one-dimensional stochastic process providing the dynamics of the underlying, the end time of the tree (with the start time assumed to be 0, another implicit constraint), and the number of time steps. It calculates and stores a few simple quantities: the initial value of the underlying, obtained by calling the `x0` method of the process; the size of a time step, obtained by dividing the total time by the number of steps; and the amount of underlying drift per step, once again provided by the process (since the parameters are constant, the drift is calculated at $t = 0$). The class declares corresponding data members for each of these quantities.

Unfortunately, there's no way to check that the given process is actually one with constant parameters. On the one hand, the library doesn't declare a specific type for that kind of process; it would be orthogonal to the declaration of different process classes, and it would lead to multiple inheritance—the bad kind—if one wanted to define, say, a constant-parameter Black-Scholes process (it would have to be both a constant-parameter process and a Black-Scholes process). Therefore, we can't enforce the property by restricting the type to be passed to the constructor. On the other hand, we can't write run-time tests for that: we might check a few sample points, but there's no guarantee that the parameters don't change elsewhere.

One option that remains is to document the behavior and trust the user not to abuse the class. Another (which we won't pursue now for backward compatibility, but might be for a future version of the library) would be for the constructor to forgo the process and take the constant parameters instead. However, this would give the user the duty to extract the parameters from the process at each constructor invocation. As usual, we'll have to find some kind of balance.

The two last methods define the structure of the tree. The `size` method returns the number of nodes at the i-th level; the code assumes that there's a single node (the root of the tree) at level 0, which gives $i + 1$ nodes at level i because of recombination. This seems reasonable enough, until we realize that we also assumed that the tree starts at $t = 0$. Put together, these two assumptions prevent us from implementing techniques such as, say, having three nodes at $t = 0$ in order to calculate Delta and Gamma by finite differences. Luckily enough, this problem might be overcome without losing backward compatibility: if one were to try implementing it, the technique could be enabled by adding additional constructors to the tree, thus allowing to relax the assumption about the start time.

Finally, the `descendant` method describes the links between the nodes. Due to the simple recombining structure of the tree, the node at index 0 on one level connects to the two nodes at index 0 and 1 on the next level, the node at index 1 to those at index 1 and 2, and in general the node at index i to those at index i and $i + 1$. Since the chosen branch is passed as an index (0 or 1 for a binomial tree) the method just has to add the index of the node to that of the branch to return

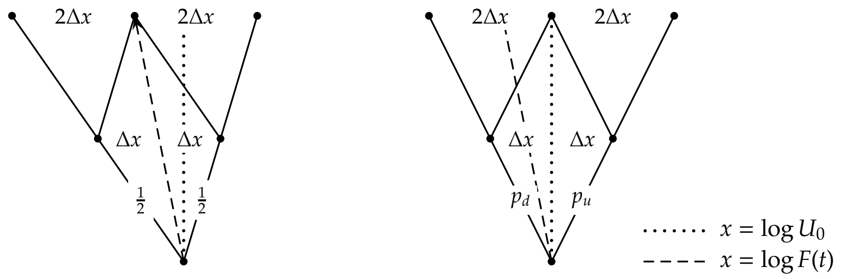

Figure 7.3: Equal-probabilities vs equal-jumps binomial trees.

the correct result. Neither index is range-checked, since the method will be called almost always inside a loop; checks would not only be costly (which is possible, but I haven't measured the effect, so I'll leave it at that) but actually redundant, since they will already be performed while setting up the loop.

The implementation of the `BinomialTree` class template misses the methods that map the dynamics of the underlying to the tree nodes: that is, the `underlying` and `probability` methods. They are left to derived classes, since there are several ways in which they can be written. In fact, there are families of ways: the first two class templates in listing 7.10 are meant to be base classes for two such families.

The first, `EqualProbabilitiesBinomialTree`, can be used for those trees in which from each node there's the same probability to go to either branch. This obviously fixes the implementation of the `probability` method, which identically returns ½, but also sets constraints on the `underlying` method. Equal probability to go along each branch means that the spine of the tree (that is, the center nodes of each level) is the most probable place to be; and in turn, this means that such nodes must be placed at the forward value of the underlying for the corresponding time (see figure 7.3 for an illustration of the idea). The class defines an `underlying` method that implements the resulting formula for the logarithm of the underlying (you can tell that the whole thing was written with the Black-Scholes process in mind). A data member `up_` is defined and used to hold the half-displacement from the forward value per each move away from the center, represented by Δx in the figure; however, the constructor of the class does not give it a value, leaving this small freedom to derived classes.

The second template, `EqualJumpsBinomialTree`, is used for those trees in which going to either branch implies an equal amount of movement in opposite directions for the logarithm of the underlying (again, see figure 7.3); since one of the directions goes with the drift and the other against it, the probabilities are different. The template defines the `underlying` method accordingly, leaving to

Listing 7.10: A few classes derived from the `BinomialTree` class template.

```
template <class T>
class EqualProbabilitiesBinomialTree : public BinomialTree<T> {
  public:
    EqualProbabilitiesBinomialTree(
         const boost::shared_ptr<StochasticProcess1D>& process,
         Time end,
         Size steps)
    : BinomialTree<T>(process, end, steps) {}
    Real underlying(Size i, Size index) const {
        int j = 2*int(index) - int(i);
        return this->x0_*std::exp(i*this->driftPerStep_
                                  + j*this->up_);
    }
    Real probability(Size, Size, Size) const {
        return 0.5;
    }
  protected:
    Real up_;
};


template <class T>
class EqualJumpsBinomialTree : public BinomialTree<T> {
  public:
    EqualJumpsBinomialTree(
         const boost::shared_ptr<StochasticProcess1D>& process,
         Time end,
         Size steps)
    : BinomialTree<T>(process, end, steps) {}
    Real underlying(Size i, Size index) const {
        int j = 2*int(index) - int(i);
        return this->x0_*std::exp(j*this->dx_);
    }
    Real probability(Size, Size, Size branch) const {
        return (branch == 1 ? pu_ : pd_);
    }
  protected:
    Real dx_, pu_, pd_;
};
```

Listing 7.10 (continued).

```
class JarrowRudd
    : public EqualProbabilitiesBinomialTree<JarrowRudd> {
  public:
    JarrowRudd(const boost::shared_ptr<StochasticProcess1D>&,
               Time end, Size steps, Real strike)
    : EqualProbabilitiesBinomialTree<JarrowRudd>(process,
                                                 end, steps) {
        up_ = process->stdDeviation(0.0, x0_, dt_);
    }
};

class CoxRossRubinstein
    : public EqualJumpsBinomialTree<CoxRossRubinstein> {
  public:
    CoxRossRubinstein(
               const boost::shared_ptr<StochasticProcess1D>&,
               Time end, Size steps, Real strike)
    : EqualJumpsBinomialTree<CoxRossRubinstein>(process,
                                                end, steps) {
        dx_ = process->stdDeviation(0.0, x0_, dt_);
        pu_ = 0.5 + 0.5*driftPerStep_/dx_;
        pd_ = 1.0 - pu_;
    }
};

class Tian : public BinomialTree<Tian> {
  public:
    Tian(const boost::shared_ptr<StochasticProcess1D>&,
         Time end, Size steps, Real strike)
    : BinomialTree<Tian>(process, end, steps) {
        // sets up_, down_, pu_, and pd_
    }
    Real underlying(Size i, Size index) const {
        return x0_ * std::pow(down_, Real(int(i)-int(index)))
                   * std::pow(up_, Real(index));
    };
    Real probability(Size, Size, Size branch) const {
        return (branch == 1 ? pu_ : pd_);
    }
  protected:
    Real up_, down_, pu_, pd_;
};
```

derived classes the task of filling the value of the `dx_` data member; and it also provides an implementation for the `probability` method that returns one of two values (`pu_` or `pd_`) which must also be specified by derived classes.

Finally, the three classes in the second page of listing 7.10 are examples of actual binomial trees, and as such are no longer class templates. The first implements the Jarrow-Rudd tree; it inherits from `EqualProbabilitiesBinomialTree` (note the application of CRTP) and its constructor sets the `up_` data member to the required value, namely, the standard deviation of the underlying distribution after one time step. The second implements the Cox-Ross-Rubinstein tree, inherits from `EqualJumpsBinomialTree`, and its constructor calculates the required probabilities as well as the displacement. The third one implements the Tian tree, and provides an example of a class that doesn't belong to either family; as such, it inherits from `BinomialTree` directly and provides all required methods.

Trinomial trees are implemented as different beasts entirely, and have a lot more leeway in connecting nodes. The way they're built (which is explained in greater detail in Brigo and Mercurio [4]) is sketched in figure 7.4: on each level in the tree, nodes are placed at an equal distance between them based on a center node with the same underlying value as the root of the tree; in the figure, the center nodes are A_3 and B_3, placed on the same vertical line. As you can see, the distance can be different on each level.

Once the nodes are in place, we build the links between them. Each node on a level corresponds, of course, to an underlying value x at the given time t. From each of them, the process gives us the expectation value for the next time conditional to starting from (x, t); this is represented by the dotted lines. For each forward value, we determine the node which is closest and use that node for the middle branch. For instance, let's look at the node A_4 in the figure. The dynamics of the underlying gives us a forward value corresponding to the point F_4 on the

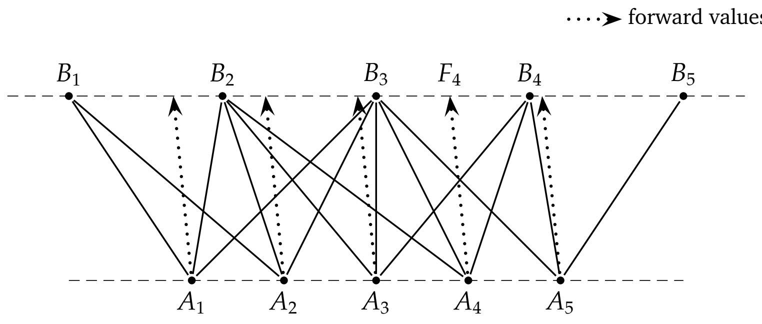

Figure 7.4: Schematics of trinomial tree branching.

Listing 7.11: Sketch of the `TrinomialTree` class.

```
class TrinomialTree : public Tree<TrinomialTree> {
    class Branching;
  public:
    enum Branches { branches = 3 };
    TrinomialTree(
           const boost::shared_ptr<StochasticProcess1D>& process,
           const TimeGrid& timeGrid,
           bool isPositive = false);
    Real dx(Size i) const;
    const TimeGrid& timeGrid() const;
    Size size(Size i) const  {
        return i==0 ? 1 : branchings_[i-1].size();
    }
    Size descendant(Size i, Size index, Size branch) const;
    Real probability(Size i, Size index, Size branch) const;
    Real underlying(Size i, Size index) const {
        return i==0 ? x0_;
            x0_ + (branchings_[i-1].jMin() + index)*dx(i);
    }
  protected:
    std::vector<Branching> branchings_;
    Real x0_;
    std::vector<Real> dx_;
    TimeGrid timeGrid_;
};
```

next level, which is closest to the node B_3. Therefore, A_4 branches out to B_3 in the middle and to its nearest siblings B_2 on the left and B_4 on the right.

As you see from the figure, it can well happen that two nodes on one level have forward values which are closest to the same node on the next level; see, for instance, nodes A_3 and A_4 going both to B_3 in the middle. This means that, while it's guaranteed by construction that three branches start from each node, there's no telling beforehand how many branches *go* to a given node; here we range from B_5 being the end node of just one branch to B_3 being the end node of five. There is also no telling how many nodes we'll need at each level: in this case, five B nodes are enough to receive all the branches starting from five A nodes, but depending on the dynamics of the process we might have needed more nodes or fewer (you can try it: modify the distances in the figure and see what happens).

The logic I just described is implemented by QuantLib in the `TrinomialTree` class, shown in listing 7.11. Its constructor takes a one-dimensional process, a

Listing 7.12: Implementation of the `TrinomialTree::Branching` inner class.

```
class TrinomialTree::Branching {
  public:
    Branching()
    : probs_(3), jMin_(QL_MAX_INTEGER), jMax_(QL_MIN_INTEGER) {}
    Size size() const {
        return jMax_ - jMin_ + 1;
    }
    Size descendant(Size i, Size branch) const {
        return k_[i] - jMin_ + branch - 1;
    }
    Real probability(Size i, Size branch) const {
        return probs_[branch][i];
    }
    Integer jMin() const;
    Integer jMax() const;
    void add(Integer k, Real p1, Real p2, Real p3) {
        k_.push_back(k);
        probs_[0].push_back(p1);
        probs_[1].push_back(p2);
        probs_[2].push_back(p3);

        jMin_ = std::min(jMin_, k-1);
        jMax_ = std::max(jMax_, k+1);
    }
  private:
    std::vector<Integer> k_;
    std::vector<std::vector<Real> > probs_;
    Integer jMin_, jMax_;
};
```

time grid specifying the times corresponding to each level of the tree (which don't need to be at regular intervals) and a boolean flag which, if set, constrains the underlying variable to be positive.

I'll get to the construction in a minute, but first I need to show how the information is stored; that is, I need to describe the inner `Branching` class, shown in listing 7.12. Each instance of the class stores the information for one level of the tree; e.g., one such instance could encode figure 7.4.

As I mentioned, nodes are placed based on a center node corresponding to the initial value of the underlying. That's the only available reference point, since we don't know how many nodes we'll use on either side. Therefore, the `Branching` class uses an index system that assigns the index $j = 0$ to the center node and works outwards from there. For instance, on the lower level in the figure we'd have $j = 0$ for A_3, $j = 1$ for A_4, $j = -1$ for A_2 and so on; on the upper level, we'd start from $j = 0$ for B_3. These indexes will have to be translated to those used by the tree interface, that start at 0 for the leftmost node (A_1 in the figure).

To hold the tree information, the class declares as data members a vector of integers, storing for each lower-level node the index (in the branching system) of the corresponding mid-branch node on the upper level;[8] three vectors, declared for convenience of access as a vector of vectors, storing the probabilities for each of the three branches; and two integers storing the minimum and maximum node index used on the upper level (again, in the branching system).

Implementing the tree interface requires some care with the several indexes we need to juggle. For instance, let's go back to `TrinomialTree` and look at the `size` method. It must return the number of nodes at level *i*; and since each branching holds information on the nodes of its upper level, the correct figure must be retrieved from `branchings_[i-1]` (except for the case $i = 0$, for which the result is 0 by construction). To allow this, the `Branching` class provides a `size` method that returns the number of points on the upper level; since `jMin_` and `jMax_` store the indexes of the leftmost and rightmost node, respectively, the number to return is `jMax_ - jMin_ + 1`. In the figure, indexes go from −2 to 2 (corresponding to B_1 and B_5) yielding 5 as the number of nodes.

The `descendant` and `probability` methods of the tree both call corresponding methods in the `Branching` class. The first returns the descendant of the *i*-th lower-level node on the given branch (specified as 0, 1 or 2 for the left, mid and right branch, respectively). To do so, it first retrieves the index *k* of the mid-branch node in the internal index system; then it subtracts `jMin`, which transforms it in the corresponding external index; and finally takes the branch into account by adding `branch-1` (that is, −1, 0 or 1 for left, mid and right branch). The `probability` method is easy enough: it selects the correct vector based on the branch and returns the probability for the *i*-th node. Since the vector indexes are zero-based, no conversion is needed.

Finally, the `underlying` method is implemented by `TrinomialTree` directly, since the branching doesn't store the relevant process information. The `Branching` class only needs to provide an inspector `jMin`, which the tree uses to determine the offset of the leftmost node from the center; it also provides a `jMax` inspector, as well as an `add` method which is used to build the branching. Such method should probably be called `push_back` instead; it takes the data for a single node (that is, mid-branch index and probabilities), adds them to the back of the corresponding vectors, and updates the information on the minimum and maximum indexes.

[8]There's no need to store the indexes of the left-branch and right-branch nodes, as they are always the neighbors of the mid-branch one.

Listing 7.13: Construction of a `TrinomialTree` instance.

```
TrinomialTree::TrinomialTree(
           const boost::shared_ptr<StochasticProcess1D>& process,
           const TimeGrid& timeGrid,
           bool isPositive)
: Tree<TrinomialTree>(timeGrid.size()), dx_(1, 0.0),
  timeGrid_(timeGrid) {
    x0_ = process->x0();
    Size nTimeSteps = timeGrid.size() - 1;
    Integer jMin = 0, jMax = 0;

    for (Size i=0; i<nTimeSteps; i++) {
        Time t = timeGrid[i];
        Time dt = timeGrid.dt(i);

        Real v2 = process->variance(t, 0.0, dt);
        Volatility v = std::sqrt(v2);
        dx_.push_back(v*std::sqrt(3.0));

        Branching branching;
        for (Integer j=jMin; j<=jMax; j++) {
            Real x = x0_ + j*dx_[i];
            Real f = process->expectation(t, x, dt);
            Integer k = std::floor((f-x0_)/dx_[i+1] + 0.5);

            if (isPositive)
                while (x0_+(k-1)*dx_[i+1]<=0)
                    k++;

            Real e = f - (x0_ + k*dx_[i+1]);
            Real e2 = e*e, e3 = e*std::sqrt(3.0);

            Real p1 = (1.0 + e2/v2 - e3/v)/6.0;
            Real p2 = (2.0 - e2/v2)/3.0;
            Real p3 = (1.0 + e2/v2 + e3/v)/6.0;

            branching.add(k, p1, p2, p3);
        }
        branchings_.push_back(branching);
        jMin = branching.jMin();
        jMax = branching.jMax();
    }
}
```

What remains now is for me to show (with the help of listing 7.13 and, again, of figure 7.4) how a `TrinomialTree` instance is built.

As you saw a couple of pages back, the constructor takes the stochastic process for the underlying variable, a time grid, and a boolean flag. The number of times in the grid corresponds to the number of levels in the tree, so it is passed to the base `Tree` constructor; also, the time grid is stored and a vector `dx_`, which will store the distances between nodes at the each level, is initialized. The first level has just one node, so there's no corresponding distance to speak of; thus, the first element of the vector is just set to 0. Other preparations include storing the initial value `x0_` of the underlying and the number of steps; and finally, declaring two variables `jMin` and `jMax` to hold the minimum and maximum node index. At the initial level, they both equal 0.

After this introduction, the tree is built recursively from one level to the next. For each step, we take the initial time `t` and the time step `dt` from the time grid. They're used as input to retrieve the variance of the process over the step, which is assumed to be independent of the value of the underlying (as for binomial trees, we have no way to enforce this). Based on the variance, we calculate the distance to be used at the next level and store it in the `dx_` vector;[9] and after this, we finally build a `Branching` instance.

To visualize the process, let's refer again to the figure. The code cycles over the nodes on the lower level, whose indexes range between the current values of `jMin` and `jMax`: in our case, that's -2 for A_1 and 2 for A_5. For each node, we can calculate the underlying value `x` from the initial value `x0_`, the index `j`, and the distance `dx_[i]` between the nodes. From `x`, the process can give us the forward value `f` of the variable after `dt`; and having just calculated the distance `dx_[i+1]` between nodes on the upper level, we can find the index `k` of the node closest to `f`. As usual, `k` is an internal index; for the node A_4 in the figure, whose forward value is F_4, the index `k` would be 0 corresponding to the center node B_3.

If the boolean flag `isPositive` is true, we have to make sure that no node corresponds to a negative or null value of the underlying; therefore, we check the value at the left target node (that would be the one with index `k-1`, since `k` is the index of the middle one) and if it's not positive, we increase `k` and repeat until the desired condition holds. In the figure, if the underlying were negative at node B_1 then node A_1 would branch to B_2, B_3 and B_4 instead.

Finally, we calculate the three transition probabilities `p1`, `p2` and `p3` (the formulas are derived in [4]) and store them in the current `Branching` instance together with `k`. When all the nodes are processed, we store the branching and update the values of `jMin` and `jMax` so that they range over the nodes on the upper level; the new values will be used for the next step of the main `for` loop (the one over time steps) in which the current upper level will become the lower level.

[9]The value of $\sqrt{3}$ times the variance is suggested by Hull and White as resulting in best stability.

7.2.3. The `TreeLattice` class template

After this rather long detour on trees, we're now back to lattices. The `TreeLattice` class, shown in listing 7.14, inherits from `Lattice` and is used as a base class for lattices that are implemented in terms of one or more trees. Of course, it should be called `TreeBasedLattice` instead; the shorter name was probably chosen before we discovered the marvels of automatic completion—or English grammar.

This class template acts as an adapter between the `Lattice` class, from which it inherits the interface, and the `Tree` class template which will be used for the implementation. Once again, we used CRTP (which wasn't actually needed for trees, but it is in this case); the behavior of the lattice is written in terms of a number of methods that must be defined in derived classes. For greater generality, there is no mention of trees in the `TreeLattice` class. It's up to derived classes to choose what kind of trees they should contain and how to use them.

The `TreeLattice` constructor is simple enough: it takes and stores the time grid and an integer `n` specifying the order of the tree (2 for binomial, 3 for trinomial and so on). It also performs a check or two, and initializes a couple of data members used for caching data; but I'll gloss over that here.

The interesting part is the implementation of the `Lattice` interface, which follows the outline I gave back in section 7.1. The `initialize` method calculates the index of the passed time on the stored grid, sets the asset time, and finally passes the number of nodes on the corresponding tree level to the asset's `reset` method. The number of nodes is obtained by calling the `size` method through CRTP; this is one of the methods that derived classes will have to implement, and (like all other such methods) has the same signature as the corresponding method in the tree classes.

The `rollback` and `partialRollback` methods perform the same work, with the only difference that `rollback` performs the adjustment at the final time and `partialRollback` doesn't. Therefore, it's only to be expected that the one is implemented in terms of the other; `rollback` performs a call to `partialRollback`, followed by another to the asset's `adjustValues` method.

The rollback procedure is spelled out in `partialRollback`: it finds the indexes of the current time and of the target time on the grid, and it loops from one to the other. At each step, it calls the `stepback` method, which implements the actual numerical work of calculating the asset values on the `i`-th level of the tree from those on level `i+1`; then it updates the asset and, at all steps except the last, calls its `adjustValues` method.

The implementation of the `stepback` method defines, by using it,[10] the interface that derived classes must implement. It determines the value of the asset at each node by combining the values at each descendant node, weighed by the corresponding transition probability; the result is further adjusted by discounting it. All

[10]That's currently the only sane way, since concepts were left out of C++11.

Listing 7.14: Sketch of the TreeLattice class template.

```
template <class Impl>
class TreeLattice : public Lattice,
                    public CuriouslyRecurringTemplate<Impl> {
 public:
   TreeLattice(const TimeGrid& timeGrid, Size n);
   void initialize(DiscretizedAsset& asset, Time t) const {
       Size i = t_.index(t);
       asset.time() = t;
       asset.reset(this->impl().size(i));
   }
   void rollback(DiscretizedAsset& asset, Time to) const {
       partialRollback(asset,to);
       asset.adjustValues();
   }
   void partialRollback(DiscretizedAsset& asset, Time to) const {
       Integer iFrom = Integer(t_.index(asset.time()));
       Integer iTo = Integer(t_.index(to));
       for (Integer i=iFrom-1; i>=iTo; --i) {
           Array newValues(this->impl().size(i));
           this->impl().stepback(i, asset.values(), newValues);
           asset.time() = t_[i];
           asset.values() = newValues;
           if (i != iTo) // skip the very last adjustment
               asset.adjustValues();
       }
   }
   void stepback(Size i, const Array& values,
                         Array& newValues) const {
       for (Size j=0; j<this->impl().size(i); j++) {
           Real value = 0.0;
           for (Size l=0; l<n_; l++) {
               value += this->impl().probability(i,j,l) *
                        values[this->impl().descendant(i,j,l)];
           }
           value *= this->impl().discount(i,j);
           newValues[j] = value;
       }
   }
   Real presentValue(DiscretizedAsset& asset) const {
       Size i = t_.index(asset.time());
       return DotProduct(asset.values(), statePrices(i));
   }
   const Array& statePrices(Size i) const;
};
```

in all, the required interface includes the `size` method, which I’ve already shown; the `probability` and `descendant` methods, with the same signature as the tree methods of the same name; and the `discount` method, which takes the indexes of the desired level and node and returns the discount factor between that node and its descendants (assumed to be independent of the particular descendant).

Finally, the `presentValue` method is implemented by returning the dot-product of the asset values by the state prices at the current time on the grid. I’ll cheerfully ignore the way the state prices are calculated; suffice it to say that using them is somewhat more efficient than rolling the asset all the way back to $t = 0$.

Now, why does the `TreeLattice` implementation calls methods with the same signature as those of a tree (thus forcing derived classes to define them) instead of just storing a tree and calling its methods directly? Well, that’s the straightforward thing to do if you have just an underlying tree; and in fact, most one-dimensional lattices will just forward the calls to the tree they store. However, it wouldn’t work for other lattices (say, two-dimensional ones); and in that case, the wrapper methods used in the implementation of `stepback` allow us to adapt the underlying structure, whatever that is, to their common interface.

The library contains instances of both kinds of lattices. Most—if not all—of those of the straightforward kind inherit from the `TreeLattice1D` class template, shown in listing 7.15. It doesn’t define any of the methods required by `TreeLattice`; and the method it does implement (the `grid` method, defined as pure virtual in the `Lattice` base class) actually requires another one, namely, the `underlying` method. All in all, this class does little besides providing a useful categorization; the storage and management of the underlying tree is, again, left to derived classes.[11]

One such class is the inner `OneFactorModel::ShortRateTree` class, shown in listing 7.16. Its constructor takes a trinomial tree, built by any specific short-rate model according to its dynamics; an instance of the `ShortRateDynamics` class, which I’ll gloss over; and a time grid, which could have been extracted from the tree so I can’t figure out why we pass it instead. The grid is passed to the base-class constructor, together with the order of the tree (which is 3, of course); the tree and the dynamics are stored as data members.

As is to be expected, most of the required interface is implemented by forwarding the call to the corresponding tree method. The only exception is the `discount` method, which doesn’t have a corresponding tree method; it is implemented by asking the tree for the value of its underlying value at the relevant node, by retrieving the short rate from the dynamics, and by calculating the corresponding discount factor between the time of the node and the next time on the grid.

[11] One might argue for including a default implementation of the required methods in `TreeLattice1D`. This would probably make sense; it would make it easier to implement derived classes in the most common cases, and could be overridden if a specific lattice needed it.

Listing 7.15: Interface of the TreeLattice1D class template.

```
template <class Impl>
class TreeLattice1D : public TreeLattice<Impl> {
  public:
    TreeLattice1D(const TimeGrid& timeGrid, Size n);
    Disposable<Array> grid(Time t) const;
};
```

Listing 7.16: Implementation of the OneFactorModel::ShortRateTree class.

```
class OneFactorModel::ShortRateTree
    : public TreeLattice1D<OneFactorModel::ShortRateTree> {
  public:
    ShortRateTree(
          const boost::shared_ptr<TrinomialTree>& tree,
          const boost::shared_ptr<ShortRateDynamics>& dynamics,
          const TimeGrid& timeGrid)
    : TreeLattice1D<OneFactorModel::ShortRateTree>(timeGrid, 3),
      tree_(tree), dynamics_(dynamics) {}
    Size size(Size i) const {
        return tree_->size(i);
    }
    Real underlying(Size i, Size index) const {
        return tree_->underlying(i, index);
    }
    Size descendant(Size i, Size index, Size branch) const {
        return tree_->descendant(i, index, branch);
    }
    Real probability(Size i, Size index, Size branch) const {
        return tree_->probability(i, index, branch);
    }
    DiscountFactor discount(Size i, Size index) const {
        Real x = tree_->underlying(i, index);
        Rate r = dynamics_->shortRate(timeGrid()[i], x);
        return std::exp(-r*timeGrid().dt(i));
    }
  private:
    boost::shared_ptr<TrinomialTree> tree_;
    boost::shared_ptr<ShortRateDynamics> dynamics_;
};
```

Listing 7.17: Implementation of the `TreeLattice2D` class template.

```
template <class Impl, class T = TrinomialTree>
class TreeLattice2D : public TreeLattice<Impl> {
  public:
    TreeLattice2D(const boost::shared_ptr<T>& tree1,
                  const boost::shared_ptr<T>& tree2,
                  Real correlation)
    : TreeLattice<Impl>(tree1->timeGrid(),
                        T::branches*T::branches),
      tree1_(tree1), tree2_(tree2), m_(T::branches,T::branches),
      rho_(std::fabs(correlation)) { ... }
    Size size(Size i) const {
        return tree1_->size(i)*tree2_->size(i);
    }
    Size descendant(Size i, Size index, Size branch) const {
        Size modulo = tree1_->size(i);

        Size index1 = index % modulo;
        Size index2 = index / modulo;
        Size branch1 = branch % T::branches;
        Size branch2 = branch / T::branches;

        modulo = tree1_->size(i+1);
        return tree1_->descendant(i, index1, branch1) +
               tree2_->descendant(i, index2, branch2)*modulo;
    }
    Real probability(Size i, Size index, Size branch) const {
        Size modulo = tree1_->size(i);

        Size index1 = index % modulo;
        Size index2 = index / modulo;
        Size branch1 = branch % T::branches;
        Size branch2 = branch / T::branches;

        Real prob1 = tree1_->probability(i, index1, branch1);
        Real prob2 = tree2_->probability(i, index2, branch2);
        return prob1*prob2 + rho_*(m_[branch1][branch2])/36.0;
    }
  protected:
    boost::shared_ptr<T> tree1_, tree2_;
    Matrix m_;
    Real rho_;
};
```

Note that, by modifying the dynamics, it is possible to change the value of the short rate at each node while maintaining the structure of the tree unchanged. This is done in a few models in order to fit the tree to the current interest-rate term structure; the `ShortRateTree` class provides another constructor, not shown here, that takes additional parameters to perform the fitting procedure.

As an example of the second kind of lattice, have a look at the `TreeLattice2D` class template, shown in listing 7.17. It acts as base class for lattices with two underlying variables, and implements most of the methods required by `TreeLattice`.[12]

The two variables are modeled by correlating the respective trees. Now, I'm sure that any figure I might draw would only add to the confusion. However, the idea is that the state of the two variables is expressed by a pair of node from the respective trees; that the transitions to be considered are those from pair to pair; and that all the possibilities are enumerated so that they can be retrieved by means a single index and thus can match the required interface.

For instance, let's take the case of two trinomial trees. Let's say we're at level i (the two trees must have the same time grid, or all bets are off). The first variable has a value that corresponds to node j on its tree, while the second sits on node k. The structure of the first tree tells us that on next level, the first variable might go to nodes j'_0, j'_1 or j'_2 with different probabilities; the second tree gives us k'_0, k'_1 and k'_2 as target nodes for the second variable. Seen as transition between pairs, this means that we're at (j,k) on the current level and that on the next level we might go to any of (j'_0,k'_0), (j'_1,k'_0), (j'_2,k'_0), (j'_0,k'_1), (j'_1,k'_1), and so on until (j'_2,k'_2) for a grand total of $3 \times 3 = 9$ possibilities. By enumerating the pairs in lexicographic order like I just did, we can give (j'_0,k'_0) the index 0, (j'_1,k'_0) the index 1, and so on until we give the index 8 to (j'_2,k'_2). In the same way, if on a given level there are n nodes on the first tree and m on the second, we get $n \times m$ pairs that, again, can be enumerated in lexicographic order: the pair (j,k) is given the index $k \times n + j$.

At this point, the implementation starts making sense. The constructor of `TreeLattice2D` takes and stores the two underlying trees and the correlation between the two variables; the base-class `TreeLattice` constructor is passed the time grid, taken from the first tree, and the order of the lattice, which equals the product of the orders of the two trees; for two trinomial trees, this is $3 \times 3 = 9$ as above.[13] The constructor also initializes a matrix `m_` that will be used later on.

The size of the lattice at a given level is the product $n \times m$ of the sizes of the two trees, which translates in a straightforward way into the implementation of the `size` method.

[12]In this, it differs from `TreeLattice1D` which didn't implement any of them. We might have had a more symmetric hierarchy by leaving `TreeLattice2D` mostly empty and moving the implementation to a derived class. At this time, though, it would sound a bit like art for art's sake.

[13]The current implementation assumes that the two trees are of the same type, but it could easily be made to work with two trees of different orders.

Things get more interesting with the two following methods. The `descendant` method takes the level `i` of the tree; the index of the lattice node, which is actually the index of a pair (j, k) among all those available; and the index of the branch to take, which by the same token is a pair of branches. The first thing it does is to extract the actual pairs. As I mentioned, the passed index equals $k \times n + j$, which means that the two underlying indexes can be retrieved as `index % n` and `index / n`. The same holds for the two branches, with n being replaced by the order of the first tree. Having all the needed indexes and branches, the code calls the `descendant` method on the two trees, obtaining the indexes j' and k' of the descendant nodes; then it retrieves the size n' of the first tree at the next level; and finally returns the combined index $k' \times n' + j'$.

Up to a certain point, the `probability` method performs the same calculations; that is, until it retrieves the two probabilities from the two underlying trees. If the two variables were not correlated, the probability for the transition would then be the product of the two probabilities. Since this is not the case, a correction term is added which depends from the passed correlation (of course) and also from the chosen branches. I'll gloss on the value of the correction as I already did on several other formulas.

This completes the implementation. Even though it contains a lot more code than its one-dimensional counterpart, `TreeLattice2D` is still an incomplete class. Actual lattices will have to inherit from it, close the CRTP loop, and implement the missing `discount` method.

I'll close this section by mentioning one feature that is currently missing from lattices, but would be nice to have. If you turn back to page 171, you'll notice that the `stepback` method assumes that the values we're rolling back are cash values; that is, it always discounts them. It could also be useful to roll values back without discounting. For instance, the probability that an option be exercised could be calculated by rolling it back on the trees without discounting, while adjusting it to 1 on the nodes where the exercise condition holds.

7.3. Tree-based engines

As you might have guessed, a tree-based pricing engine will perform few actual computations; its main job will rather be to instantiate and drive the needed discretized asset and lattice.[14]

7.3.1. Example: callable fixed-rate bonds

As an example, I'll sketch the implementation of a tree-based pricing engine for callable fixed-rate bonds. For the sake of brevity, I'll skip the description of the

[14]If you're pattern-minded, you can have your pick here. This implementation has suggestions of the Adapter, Mediator, or Facade pattern, even though it doesn't match any of them exactly.

Listing 7.18: Interface of the `CallableBond` inner classes.

```
class CallableBond::arguments : public PricingEngine::arguments {
  public:
    std::vector<Date> couponDates;
    std::vector<Real> couponAmounts;
    Date redemptionDate;
    Real redemptionAmount;
    std::vector<Callability::Type> callabilityTypes;
    std::vector<Date> callabilityDates;
    std::vector<Real> callabilityPrices;
    void validate() const;
};

class CallableBond::results : public Instrument::results {
  public:
    Real settlementValue;
};

class CallableBond::engine
    : public GenericEngine<CallableBond::arguments,
                           CallableBond::results> {};
```

`CallableBond` class.[15] Instead, I’ll just show its inner `arguments` and `results` classes, which act as its interface with the pricing engine and which you can see in listing 7.18 together with the corresponding `engine` class. If you’re interested in a complete implementation, you can look for it in QuantLib’s experimental folder.

Now, let’s move into engine territory. In order to implement the behavior of the instrument, we’ll need a discretized asset; namely, the `DiscretizedCallableBond` class, shown in listing 7.19.

To prevent much aggravation, its constructor takes and stores an instance of the `arguments` class. This avoids having to spell out the list of needed data in at least three places: the declaration of the data members, the constructor, and the client code that instantiates the discretized asset. Besides the `arguments` instance, the constructor is also passed a reference date and a day counter that are used in its body to convert the several bond dates into corresponding times. (The conversion is somewhat verbose, which suggests that we might be missing an abstraction here. However, “time converter” sounds a bit too vague. If you find it, please do let me know. The thing has been bugging me for a while.)

[15]To be specific, the class name should be `CallableFixedRateBond`; but that would get old very quickly here, so please allow me to use the shorter name.

Listing 7.19: Implementation of the `DiscretizedCallableBond` class.

```
class DiscretizedCallableBond : public DiscretizedAsset {
  public:
    DiscretizedCallableBond(const CallableBond::arguments& args,
                            const Date& referenceDate,
                            const DayCounter& dayCounter)
    : arguments_(args) {
        redemptionTime_ =
            dayCounter.yearFraction(referenceDate,
                                    args.redemptionDate);

        couponTimes_.resize(args.couponDates.size());
        for (Size i=0; i<couponTimes_.size(); ++i)
            couponTimes_[i] =
                dayCounter.yearFraction(referenceDate,
                                        args.couponDates[i]);
        // same for callability times
    }
    std::vector<Time> mandatoryTimes() const {
        std::vector<Time> times;

        Time t = redemptionTime_;
        if (t >= 0.0)
            times.push_back(t);
        // also add non-negative coupon times and callability times

        return times;
    }
    void reset(Size size) {
        values_ = Array(size, arguments_.redemptionAmount);
        adjustValues();
    }
  protected:
    void preAdjustValuesImpl();
    void postAdjustValuesImpl();
  private:
    CallableBond::arguments arguments_;
    Time redemptionTime_;
    std::vector<Time> couponTimes_;
    std::vector<Time> callabilityTimes_;
    void applyCallability(Size i);
    void addCoupon(Size i);
};
```

Listing 7.19 (continued).

```
void DiscretizedCallableBond::preAdjustValuesImpl() {
    for (Size i=0; i<callabilityTimes_.size(); i++) {
        Time t = callabilityTimes_[i];
        if (t >= 0.0 && isOnTime(t)) {
            applyCallability(i);
        }
    }
}

void DiscretizedCallableBond::postAdjustValuesImpl() {
    for (Size i=0; i<couponTimes_.size(); i++) {
        Time t = couponTimes_[i];
        if (t >= 0.0 && isOnTime(t)) {
            addCoupon(i);
        }
    }
}

void DiscretizedCallableBond::applyCallability(Size i) {
    switch (arguments_.callabilityTypes[i]) {
      case Callability::Call:
        for (Size j=0; j<values_.size(); j++) {
            values_[j] =
                std::min(arguments_.callabilityPrices[i],
                         values_[j]);
        }
        break;
      case Callability::Put:
        for (Size j=0; j<values_.size(); j++) {
            values_[j] =
                std::max(arguments_.callabilityPrices[i],
                         values_[j]);
        }
        break;
      default:
        QL_FAIL("unknown callability type");
    }
}

void DiscretizedCallableBond::addCoupon(Size i) {
    values_ += arguments_.couponAmounts[i];
}
```

Next comes the required `DiscretizedAsset` interface. The `mandatoryTimes` method collects the redemption time, the coupon times, and the callability times filtering out the negative ones; and the `reset` method resizes the array of the values, sets each one to the redemption amount, and proceeds to perform the needed adjustments—that is, the more interesting part of the class.

Being rather specialized, it is pretty unlikely that this class will be composed with others; therefore, it doesn't really matter in this case whether the adjustments go into `preAdjustValuesImpl` or `postAdjustValuesImpl`. However, for sake of illustration, I'll separate the callability from the coupon payments and manage them as pre- and post-adjustment, respectively.

The `preAdjustValuesImpl` loops over the callability times, checks whether any of them equals the current time, and calls the `applyCallability` method if this is the case. The `postAdjustValuesImpl` does the same, but checking the coupon times and calling the `addCoupon` method instead.

The `applyCallability` method is passed the index of the callability being exercised; it checks its type (both callable and puttable bonds are supported) and sets the value at each node to the value after exercise. The logic is simple enough: at each node, given the estimated value of the rest of the bond (that is, the current asset value) and the exercise premium, the issuer will choose the lesser of the two values while the holder will choose the greater. The `addCoupon` method is much simpler, and just adds the coupon amount to each of the values.

As you might have noticed, this class assumes that the exercise dates coincide with the coupon dates; it won't work if an exercise date is a few days before a coupon payment (the coupon amount would be added to the asset values before the exercise condition is checked). Of course, this is often the case, and it should be accounted for. Currently, the library implementation sidesteps the problem by adjusting each exercise date so that it equals the nearest coupon date. A better choice would be to detect which coupons are affected; each of them would be put into a new asset, rolled back until the relevant exercise time, and added after the callability adjustment.

Finally, listing 7.20 shows the `TreeCallableBondEngine` class. Its constructor takes and stores a handle to a short-rate model that will provide the lattice, the total number of time steps we want the lattice to have, and an optional reference date and day counter; the body just registers to the handle.

The `calculate` method is where everything happens. By the time it is called, the engine arguments have been filled by the instrument, so that base is covered; the other data we need are a date and a day counter for time conversion. Not all short-rate models can provide them, so, in a boring few lines of code not shown here, the engine tries to downcast the model to some specific class that does; if it fails, it falls back to using the ones optionally passed to the constructor.

At that point, the actual calculations can begin. The engine instantiates the discretized bond, asks it for its mandatory times, and uses them to build a time

Listing 7.20: Sketch of the TreeCallableBondEngine class.

```
class TreeCallableBondEngine : public CallableBond::engine {
  public:
    TreeCallableBondEngine(
                   const Handle<ShortRateModel>& model,
                   const Size timeSteps,
                   const Date& referenceDate = Date(),
                   const DayCounter& dayCounter = DayCounter());
    : model_(model), timeSteps_(timeSteps),
      referenceDate_(referenceDate), dayCounter_(dayCounter) {
        registerWith(model_);
    }
    void calculate() const {
        Date referenceDate;
        DayCounter dayCounter;

        // try to extract the reference date and the day counter
        // from the model, use the stored ones otherwise.

        DiscretizedCallableBond bond(arguments_,
                                     referenceDate,
                                     dayCounter);

        std::vector<Time> times = bond.mandatoryTimes();
        TimeGrid grid(times.begin(), times.end(), timeSteps_);
        boost::shared_ptr<Lattice> lattice = model_->tree(grid);

        Time redemptionTime =
            dayCounter.yearFraction(referenceDate,
                                    arguments_.redemptionDate);
        bond.initialize(lattice, redemptionTime);
        bond.rollback(0.0);
        results_.value = bond.presentValue();
    }
};
```

grid; then, the grid is passed to the model which returns a corresponding lattice based on the short-rate dynamics. All that remains is to initialize the bond at its redemption time (which in the current code is recalculated explicitly, but could be retrieved as the largest of the mandatory times), roll it back to the present time, and read its value.