```
/*
 * NDFA.cs
 * Copyright: Andrew Matthews (c) 2007
 * All rights reserved.
 */
using System;
using System.Collections.Generic;
using C5;

namespace Automata
{
    public class HashDictionary<K, T> : C5.HashDictionary<K, TreeSet<T>>
    {
        public void Add(K key, T t)
        {
            TreeSet<T> x;
            if (Contains(key))
            {
                x = this[key];
            }
            else
            {
                x = new TreeSet<T>();
                Add(key, x);
            }
            x.Add(t);
        }
    }

    public class NDFA<Q, AB, R> : INdfa<Q, AB, R>
    {
        protected IPersistentSorted<Q> acceptStates = new TreeSet<Q>();
        protected IPersistentSorted<Q> allStates = new TreeSet<Q>();
        protected IComparer<Rec<Q, Q>> comparer;
        protected TreeSet<Q> currentStates;
        protected Q errorState;
        protected TreeSet<Q> previousStates;
        protected Q startState;
        protected C5.IDictionary<Rec<Q, Q>, TransitionFunction<Q, AB, R>> transitionFunc
            new C5.HashDictionary<Rec<Q, Q>, TransitionFunction<Q, AB, R>>();
        protected HashDictionary<Rec<Q, AB>, Q> transitionTable = new HashDictionary<Rec
        protected event ErrorHandler<Q, AB, R> Error;

        protected TreeSet<Rec<Q, Q>> GetNextStates(AB input)
        {
            TreeSet<Rec<Q, Q>> nextStates = new TreeSet<Rec<Q, Q>>();
            foreach (Q s in CurrentStates)
            {
                TreeSet<Rec<Q, Q>> newStatesToAdd = GetNextStatesFromState(s, input);
                foreach (Rec<Q, Q> newState in newStatesToAdd)
                {
                    if (!nextStates.Contains(newState))
                        nextStates.Add(newState);
                }
            }
            return nextStates;
        }

        protected TreeSet<Rec<Q, Q>> GetNextStatesFromState(Q q, AB input)
        {
            Rec<Q, AB> key = new Rec<Q, AB>(q, input);
            TreeSet<Q> nextStates = transitionTable[key];
            TreeSet<Rec<Q, Q>> result = new TreeSet<Rec<Q, Q>>(comparer);
            foreach (Q qn in nextStates)
            {
                result.Add(new Rec<Q, Q>(q, qn));
```

```
    }
    return result;
}

protected SortedList<Rec<Q, Q>, TransitionFunction<Q, AB, R>> GetTransitionFunctions(AB input)
{
    SortedList<Rec<Q, Q>, TransitionFunction<Q, AB, R>> result =
        new SortedList<Rec<Q, Q>, TransitionFunction<Q, AB, R>>();
    foreach (Q q in currentStates)
    {
        foreach (Rec<Q, Q> trn in GetNextStatesFromState(q, input))
        {
            result.Add(trn, transitionFunctions[trn]);
        }
    }
    return result;
}

public TreeSet<Q> PreviousStates
{
    get { return previousStates; }
    set { previousStates = value; }
}

#region Interface INonDeterministicAutomaton

public IPersistentSorted<Q> AcceptStates
{
    get { return acceptStates; }
    set { acceptStates = value; }
}

public IPersistentSorted<Q> AllStates
{
    get { return allStates; }
    set { allStates = value; }
}

public IComparer<Rec<Q, Q>> Comparer
{
    get { return comparer; }
    set { comparer = value; }
}

public IPersistentSorted<Q> CurrentStates
{
    get { return currentStates; }
    set
    {
        TreeSet<Q> newStates = new TreeSet<Q>();
        newStates.AddAll(value);
        currentStates = newStates;
    }
}

public Q ErrorState
{
    get { return errorState; }
    set { errorState = value; }
}

public bool IsErrorState
{
    get { return CurrentStates.Equals(ErrorState); }
}
```

```
public bool IsInAcceptState
{
    get
    {
        foreach (Q q in currentStates)
        {
            if (AcceptStates.Contains(q))
                return true;
        }
        return false;
    }
}

public IPersistentSorted<Q> PreviousState
{
    get { return previousStates; }
    set
    {
        TreeSet<Q> stateset = new TreeSet<Q>();
        stateset.AddAll(value);
        previousStates = stateset;
    }
}

public Q StartState
{
    get { return startState; }
    set
    {
        startState = value;
        CurrentStates = new TreeSet<Q>();
        CurrentStates.Add(startState);
    }
}

public C5.IDictionary<Rec<Q, Q>, TransitionFunction<Q, AB, R>> TransitionFunctions
{
    get { return transitionFunctions; }
    set { transitionFunctions = value; }
}

public HashDictionary<Rec<Q, AB>, Q> TransitionTable
{
    get { return transitionTable; }
    set { transitionTable = value; }
}

public IEnumerable<R> ProcessInput(IEnumerable<AB> inputStream)
{
    if (IsErrorState)
    {
        throw new ApplicationException("Cannot process input when in an Error State");
    }
    foreach (AB inputToken in inputStream)
    {
        TreeSet<Q> qcur = new TreeSet<Q>();
        TreeSet<Q> qnext = new TreeSet<Q>();
        qcur.AddAll(CurrentStates);
        foreach (Q q in qcur)
        {
            TreeSet<Rec<Q, Q>> transitions = GetNextStatesFromState(q, inputToken);
            foreach (Rec<Q, Q> transition in transitions)
            {
                TransitionFunction<Q, AB, R> func = transitionFunctions[transition];
                qnext.Add(transition.X2);
                yield return func(this, transition.X1, transition.X2, inputToken);
```

```
                    }
                }
                CurrentStates = qnext;
                PreviousState = qcur;
                if (qnext.IsEmpty)
                {
                    CurrentStates.Add(errorState);
                    if (Error != null)
                        Error(this, inputToken);
                }
            }
        }

        public R ProcessInputToken(AB inputToken)
        {
            throw new NotImplementedException();
        }

        public void SetErrorHandler(ErrorHandler<Q, AB, R> errorHandler)
        {
            Error += errorHandler;
        }

        public void SetStateComparer(IComparer<Rec<Q, Q>> comp)
        {
            comparer = comp;
        }

        #endregion
    }
}
```