

# CS 475/575

## C++ Interface for Event Priority Lists Code & Some Random Distributions

Agustín J. González  
(modified by C. M. Overstreet)  
v. 2/27/02

### 1.1 Event List Data Types to C++

This document describes a consistent and common Application Programming Interface (API) for the four data types: linked list, two list, henriksen, and splay trees. This interface is an abstract class in C++ that specifies the behavior of a priority list regardless of its implementation. It has the big advantage of letting the programmer to write code without worrying on the actual data structure chosen in a particular application. The Application Programming Interface is as follows:

```
struct Event {
    double prio; // this member has to be provided

    //Any other fields the application may need
    Event *next; // just to save memory management in the performance
    // comparison of this assignment.

    Event (double t)
        : prio(t) {}
};

class PrioQueue
{
public:
    // Methods
    virtual bool isEmpty() const=0;
    virtual void enqueue(Event * event)=0;
    virtual Event * dequeue()=0;

protected:
    // Constructor
    PrioQueue() {} // disabled
};
```

This code has two classes. One is the Event class that along with the three basic messages of the abstract class allow a common protocol for priority queues. The other structure is Node class that is defined within the implementation of each version of priority queue data structure. Thus the interfaces for each of the implementation is as follows:

```
//=====
// LinkedPrioQueue Class Interface
//=====
class LinkedPrioQueue : public PrioQueue
{
```

```

public:
    // Constructor
    LinkedPrioQueue();

    // Destructor
    ~LinkedPrioQueue();

    // Methods
    bool isEmpty() const { return q == NULL; }
    void enqueue(Event * n);
    Event * dequeue();

private:
    struct Node {
        Node * leftlink;
        Event * event;
        double prio; // it is replicated to prevent double indexing

        Node (Event * e)
            : event(e), prio(e->prio) {}
    };

    const LinkedPrioQueue& operator= (const LinkedPrioQueue &p) {}
    // Data member
    Node * q;
};

//=====
// TwoListPrioQueue Class Interface
//=====
class TwoListPrioQueue : public PrioQueue
{
public:
    // Constructor
    TwoListPrioQueue();

    // Destructor
    ~TwoListPrioQueue();

    // Methods
    bool isEmpty() const { return length==0; }
    void enqueue(Event * event);
    Event * dequeue();

private:
    struct Node {
        Node * leftlink;
        Event * event;
        double prio; // it is replicated to prevent double indexing

        Node (Event * e)
            :event(e), prio(e->prio) {}
    };

    const TwoListPrioQueue & operator= (const TwoListPrioQueue &p) {}
    Node * sorted;
    Node * unsorted;
    int length;
    float cutoff;
    float delta;
};

```

```

};

//=====
// HenriksenPrioQueue Class Interface
//=====
const int CAPACITY = 4096; // 212

class HenriksenPrioQueue : public PrioQueue
{
public:
    // Constructor
    HenriksenPrioQueue();

    // Destructor
    ~HenriksenPrioQueue();

    // Methods
    bool isEmpty() const { return zero->rightlink==ptrvec[CAPACITY-1]; }
    void enqueue(Event * n);
    Event * dequeue();

private:
    struct Node {
        Node *leftlink, *rightlink;
        Event * event;
        double prio;

        Node (Event * e)
            : event(e), prio(e->prio),
            leftlink(NULL), rightlink(NULL){}
        Node (double p)
            : event(NULL), prio(p),
            leftlink(NULL), rightlink(NULL){}
    };

    // Method
    void setvec(int newsize);

    // Data members
    Node * ptrvec[CAPACITY];
    Node * zero;
    double timevec[CAPACITY];
    int vecsize, leftlim, starti, startj;
};

//=====
// SpTreePrioQueue Class Interface
//=====
class SpTreePrioQueue : public PrioQueue
{
public:
    // Constructor
    SpTreePrioQueue();

    // Destructor
    ~SpTreePrioQueue();

    // Methods
    bool isEmpty() const { return (root==NULL); }
    void enqueue(Event * n);

```

```

    Event * dequeue();

private:
struct Node {
    Node * leftlink;
    Node * rightlink;
    Node * uplink;
    Event * event;
    double prio; /* formerly char * key and even before time/timetyt */
    Node (Event * event)
        : event(event), prio(event->prio) {}
};
    // Data members
    Node * root;      // root node
};

```

The implementation of each method for each class is in Appendix A.

## 1.2 Random Distributions

A common API is defined for this class. Then several subclasses implement random number with various distributions. The distributions are uniform with the interval as a parameter, triangular with the interval as a parameter, bimodal with mean 1, and exponential with parameter lambda, which is the mean value. The common protocol is as follows:

```

//=====
// Abstract Random Interface Class
//=====
class Random
{
public:
    // Methods
    virtual double rand() =0;
    void setSeed(unsigned int s) { seed = s; }

protected:
    // Constructor
    Random() : seed(1) {} // default seed =1
    double seed;
};

```

The interface for each distribution class is as follows:

```

//=====
// Uniform Class Interface
//=====
class Uniform : public Random
{
public:
    // Constructors
    // default seed =1
    Uniform(); // default U(0,1)
    Uniform(double b); // U(0,b)
    Uniform(double a, double b); // U(a,b)

    // Method

```

```

    double rand();

private:
    double min, max;
};

//=====
// Exponential Class Interface
//=====
class Exponential : public Random
{
public:
    // Constructors
    Exponential(); // default means=1
    Exponential(double m);

    // Method
    double rand();

private:
    double mean;
};

//=====
// Triangular Class Interface
//=====
class Triangular : public Random
{
public:
    // Constructors
    Triangular(); // default min=0, max=1
    Triangular(double max);
    Triangular(double min, double max);

    // Method
    double rand();

private:
    double min, max;
};

//=====
// Bimodal Class Interface
//=====
class Bimodal : public Random
{
public:
    // Constructors
    Bimodal(); // default means=1

    // Method
    double rand();
};

```

The implementation of these classes are all based on the floating point version of the algorithm proposed by Park and Miller in "Random Number Generator: Good Ones are Hard to Find", CACM, Vol. 31, No. 10, Oct. 1988, p.1196.

