Random Set

Mac Radigan

A set-like data structure that supports Insert, Remove, and GetRandomElement efficiently

Background

This algorithm makes use of the standard unordered map's (hash map implementation) direct access for insert operations, and the standard vector's efficient amortized time complexity random access for randomly choosing an element.

This leaves only the need for removal from both the map and vector in constant time. This is already supported by the unordered map, but for the vector is only true for back insertion/removal. By introducing a cell to contain the element of interest, we can then swap the contents to be removed with the back of the vector prior to removal (for a constant-time back-removal operation). This leaves only the need to identify the candidate cell of the vector for deletion, which may be done directly by maintaining a reference in the map.

Implementation

On insertion of $x\colon T$, check for existence of x in map M. If not present, insert a cell c containing x into the back of vector V, and add an entry to the map mapping the element to the newly inserted cell, say $x \to c$.

On deletion of $x\colon T$, look up the containing cell c from the map, say $c = M\left[x\right]$. Swap the contents of c with the cell at the back of the vector. Update the map from the newly swapped contents to the cell c. Remove x from the set. Finally, remove the last element of the vector.

Algorithm 1 Insert

```
given element to insert $x$, having members vector $V$ and map $M$
if $x \notin M$ then
    let $c = Cell\left(x\right)$
    $V_{end} \leftarrow c$
    $M_x \leftarrow ref\{c\}$
end if
```

Algorithm 2 Remove

```
given element to remove $x,$ having members vector $V$ and map $M$
let $top = V_{end}$
let $candidate = M_x$
$swap\,(top_{cr}, candidate_{cr})$
$M_{candiate_{cr}} \leftarrow ref\,(candidate)$
remove $M_x$
remove $V_{end}$
```

Algorithm 3 Random Select

```
 having members vector $V$ and map $M$
$k = U[0, |V| - 1]$
return  $V_k$
```

Performance

Measure	Time Complexity
insert	constant time complexity O(1)
removal	constant time complexity O(1)
random selection	amortized constant time complexity O(1)

```
// ==========================================================================
// RandomSet
// ==========================================================================
//
//   a set-like container supporting amortized constant time insertion,
//     removal, and uniform random element selection
//
// --------------------------------------------------------------------------
//
// Background:
//
//   This algorithm makes use of the standard unordered map's (hash map
//     implementation) direct access for insert operations, and the standard
//     vector's efficient amortized time complexity random access for
//     randomly choosing an element.
//
//   This leaves only the need for removal from both the map and vector in
//     constant time.  This is already supported by the unordered map, but
//     for the vector is only true for back insertion/removal.  By
//     introducing a cell to contain the element of interest, we can then
//     swap the contents to be removed with the back of the vector prior to
//     removal (for a constant-time back-removal operation).  This leaves
//     only the need to identify the candidate cell of the vector for
//     deletion, which may be done directly by maintaining a reference in
//     the map.
//
//
// Implementation:
//
//   On insertion of x:T, check for existence of x in map M.  If not present,
//     insert a cell c containing x into the back of vector V, and add an
//     entry to the map mapping the element to the newly inserted cell.
//     the vector.  Update the map from the newly swapped contents to the
//     cell c.  Remove x from the set.  Finally, remove the last element of
//     the vector.
//
// Performance:
//
//   insert  constant time complexity:                           O(1)
//   removal constant time complexity:                           O(1)
//   random selection amortized constant time complexity:        O(1)
//
//   linear space complexity:                                    O(N)
//
//
namespace demo::algo1 {

  template<class T>
  class RandomSet
  {

   // cell_t - a LISP-style container cell with a single content register
   typedef struct cell_s
   {
     T cr;
     cell_s(T &x) : cr(x) {};
     inline void swap(struct cell_s &c) { std::swap(cr, c.cr); };
   } cell_t;

   public:

    RandomSet()
     : pdf_(0, std::numeric_limits<T>::max())
     {};

    // inserts an element into the set with constant time complexity
    inline void insert(T x)
    {
      // If x is not already in the map, insert a cell containing x at the
      //   back of the random vector.  Map x to the last cell in the vector.
      if(map_.find(x) == map_.end())
      {
        pick_.push_back(cell_t(x));
        map_.insert_or_assign(x, std::ref(pick_.back()));
      }
    } // insert

    // removes an element from the set with constant time complexity
    inline void remove(T x)
    {
      auto &top       = pick_.back(); // last element inserted
      auto &candidate = map_.at((x)); // the x to be removed
      candidate.get().swap(top);      // swap x to the back of the vector
      // update the cell reference for the map for the cell reference
      //   previously at the back of the vector
      map_.emplace(candidate.get().cr, std::ref(candidate));
      // remove x from map
      map_.erase(x);
      // remove x from vector (now at back of vector)
      pick_.pop_back();
    } // remove

    // returns the number of elements in the set
    inline std::size_t size() const
    {
      return map_.size();
    } // size

    // returns an element from the set with uniform random probability
    //   in constant-time
    inline T& get_random()
    {
      return pick_[pdf_(gen_) % pick_.size()].cr;
    } // get_random

    // prints the contents of the set
    friend inline std::ostream& operator<<(std::ostream &os, const RandomSet<T> &o)
    {
      os << "{";
      for(auto it=o.pick_.begin(); it!=o.pick_.end()-1; ++it) os << it->cr << ",";
      os << o.pick_.back().cr << "}" << std::endl;
      return os;
    } // operator<<

   private:

    // a map from an element to a cell containing the element T
    std::unordered_map<T, std::reference_wrapper<cell_t> > map_;
    // a vector of cells (containing element T)
    std::vector<cell_t> pick_;
    // randomization source
    std::mt19937 gen_{std::random_device{}()};
```

```
    // uniform distribution
    std::uniform_int_distribution<T> pdf_;

  }; // RandomSet

} // demo::algo1
```

Source Code

```
// random-set.cc
// Mac Radigan


  #include <assert.h>
  #include <cstdlib>
  #include <functional>
  #include <iomanip>
  #include <iostream>
  #include <iterator>
  #include <memory>
  #include <random>
  #include <stdexcept>
  #include <sys/types.h>
  #include <unordered_map>
  #include <vector>

  // ===========================================================================
  // RandomSet
  // ===========================================================================
  //
  //   a set-like container supporting amortized constant time insertion,
  //     removal, and uniform random element selection
  //
  // ---------------------------------------------------------------------------
  //
  // Background:
  //
  //   This algorithm makes use of the standard unordered map's (hash map
  //     implementation) direct access for insert operations, and the standard
  //     vector's efficient amortized time complexity random access for
  //     randomly choosing an element.
  //
  //   This leaves only the need for removal from both the map and vector in
  //     constant time.  This is already supported by the unordered map, but
  //     for the vector is only true for back insertion/removal.  By
  //     introducing a cell to contain the element of interest, we can then
  //     swap the contents to be removed with the back of the vector prior to
  //     removal (for a constant-time back-removal operation).  This leaves
  //     only the need to identify the candidate cell of the vector for
  //     deletion, which may be done directly by maintaining a reference in
  //     the map.
  //
  //
  // Implementation:
  //
  //   On insertion of x:T, check for existence of x in map M.  If not present,
  //     insert a cell c containing x into the back of vector V, and add an
  //     entry to the map mapping the element to the newly inserted cell,
  //     say x->c.
  //
  //   On deletion of x:T, look up the containing cell c from the map,
  //     say c = M[x].  Swap the contents of c with the cell at the back of
  //     the vector.  Update the map from the newly swapped contents to the
  //     cell c.  Remove x from the set.  Finally, remove the last element of
  //     the vector.
  //
  // Performance:
  //
  //   insert  constant time complexity:                           O(1)
  //   removal constant time complexity:                           O(1)
  //   random selection amortized constant time complexity:        O(1)
  //
  //   linear space complexity:                                    O(N)
  //
  //
  namespace demo::algo1 {

    template<class T>
    class RandomSet
    {

     // cell_t - a LISP-style container cell with a single content register
     typedef struct cell_s
     {
       T cr;
       cell_s(T &x) : cr(x) {};
       inline void swap(struct cell_s &c) { std::swap(cr, c.cr); };
     } cell_t;

    public:

     RandomSet()
      : pdf_(0, std::numeric_limits<T>::max())
      {};

     // inserts an element into the set with constant time complexity
     inline void insert(T x)
     {
       // If x is not already in the map, insert a cell containing x at the
       //   back of the random vector.  Map x to the last cell in the vector.
       if(map_.find(x) == map_.end())
       {
         pick_.push_back(cell_t(x));
         map_.insert_or_assign(x, std::ref(pick_.back()));
       }
     } // insert

     // removes an element from the set with constant time complexity
     inline void remove(T x)
     {
       auto &top       = pick_.back(); // last element inserted
       auto &candidate = map_.at((x)); // the x to be removed
       candidate.get().swap(top);      // swap x to the back of the vector
       // update the cell reference for the map for the cell reference
       //   previously at the back of the vector
       map_.emplace(candidate.get().cr, std::ref(candidate));
       // remove x from map
       map_.erase(x);
       // remove x from vector (now at back of vector)
       pick_.pop_back();
     } // remove

     // returns the number of elements in the set
     inline std::size_t size() const
     {
       return map_.size();
     } // size

     // returns an element from the set with uniform random probability
     //   in constant-time
     inline T& get_random()
     {
       return pick_[pdf_(gen_) % pick_.size()].cr;
     } // get_random

     // prints the contents of the set
     friend inline std::ostream& operator<<(std::ostream &os, const RandomSet<T> &o)
     {
       os << "{";
       for(auto it=o.pick_.begin(); it!=o.pick_.end()-1; ++it) os << it->cr << ",";
       os << o.pick_.back().cr << "}" << std::endl;
       return os;
     } // operator<<

    private:

     // a map from an element to a cell containing the element T
     std::unordered_map<T, std::reference_wrapper<cell_t> > map_;
     // a vector of cells (containing element T)
     std::vector<cell_t> pick_;
     // randomization source
     std::mt19937 gen_{std::random_device{}()};
     // uniform distribution
     std::uniform_int_distribution<T> pdf_;

   }; // RandomSet

 } // demo::algo1

 //
 // main test driver
 //
 int main(int argc, char *argv[])
 {

   // default element type (domain)
   typedef int64_t element_t;

   demo::algo1::RandomSet<element_t> rset;

   // insert
   rset.insert(1);
   rset.insert(3);
   rset.insert(6);
   rset.insert(8);

   // remove
   rset.remove(6);
```

```

    // print
    std::cout << rset;

    // random selection
    if( rset.size() > 0 )
    {
      for(int64_t k=0; k<10; ++k)
      {
        std::cout << "random x = "
                  << rset.get_random()
                  << std::endl;
      }
    }

    return EXIT_SUCCESS;
  } // main

// *EOF
```

Unit Test Results

```
{1,3,8}
random x = 1
random x = 1
random x = 8
random x = 3
random x = 1
random x = 8
random x = 1
random x = 8
random x = 3
random x = 3
```