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 $c = M\,[x]$. 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

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

Algorithm 1 Insert

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

```
// ============================================================================
// 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
```

Algorithm 3 Random Select

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

```
20 //     introducing a cell to contain the element of interest, we can then
21 //     swap the contents to be removed with the back of the vector prior to
22 //     removal (for a constant-time back-removal operation).  This leaves
23 //     only the need to identify the candidate cell of the vector for
24 //     deletion, which may be done directly by maintaining a reference in
25 //     the map.
26 //
27 //
28 // Implementation:
29 //
30 //   On insertion of x:T, check for existence of x in map M.  If not present,
31 //     insert a cell c containing x into the back of vector V, and add an
32 //     entry to the map mapping the element to the newly inserted cell,
33 //     say x->c.
34 //
35 //   On deletion of x:T, look up the containing cell c from the map,
36 //     say c = M[x].  Swap the contents of c with the cell at the back of
37 //     the vector.  Update the map from the newly swapped contents to the
38 //     cell c.  Remove x from the set.  Finally, remove the last element of
39 //     the vector.
40 //
41 // Performance:
42 //
43 //   insert  constant time complexity:                          O(1)
44 //   removal constant time complexity:                          O(1)
45 //   random selection amortized constant time complexity:       O(1)
46 //
47 //   linear space complexity:                                   O(N)
48 //
49 //
50 namespace demo::algo1 {
51
52   template<class T>
53   class RandomSet
54   {
55
56    // cell_t - a LISP-style container cell with a single content register
57    typedef struct cell_s
58    {
59      T cr;
60      cell_s(T &x) : cr(x) {};
61      inline void swap(struct cell_s &c) { std::swap(cr, c.cr); };
62    } cell_t;
63
64    public:
```

```
65
66      RandomSet()
67       : pdf_(0, std::numeric_limits<T>::max())
68       {};
69
70      // inserts an element into the set with constant time complexity
71      inline void insert(T x)
72      {
73        // If x is not already in the map, insert a cell containing x at the
74        //   back of the random vector.  Map x to the last cell in the vector.
75        if(map_.find(x) == map_.end())
76        {
77          pick_.push_back(cell_t(x));
78          map_.insert_or_assign(x, std::ref(pick_.back()));
79        }
80      } // insert
81
82      // removes an element from the set with constant time complexity
83      inline void remove(T x)
84      {
85        auto &top       = pick_.back(); // last element inserted
86        auto &candidate = map_.at((x)); // the x to be removed
87        candidate.get().swap(top);      // swap x to the back of the vector
88        // update the cell reference for the map for the cell reference
89        //   previously at the back of the vector
90        map_.emplace(candidate.get().cr, std::ref(candidate));
91        // remove x from map
92        map_.erase(x);
93        // remove x from vector (now at back of vector)
94        pick_.pop_back();
95      } // remove
96
97      // returns the number of elements in the set
98      inline std::size_t size() const
99      {
100       return map_.size();
101     } // size
102
103     // returns an element from the set with uniform random probability
104     //   in constant-time
105     inline T& get_random()
106     {
107       return pick_[pdf_(gen_) % pick_.size()].cr;
108     } // get_random
109
```

```
110     // prints the contents of the set
111     friend inline std::ostream& operator<<(std::ostream &os, const RandomSet<T> &o)
112     {
113       os << "{";
114       for(auto it=o.pick_.begin(); it!=o.pick_.end()-1; ++it) os << it->cr << ",";
115       os << o.pick_.back().cr << "}" << std::endl;
116       return os;
117     } // operator<<
118 
119    private:
120 
121     // a map from an element to a cell containing the element T
122     std::unordered_map<T, std::reference_wrapper<cell_t> > map_;
123     // a vector of cells (containing element T)
124     std::vector<cell_t> pick_;
125     // randomization source
126     std::mt19937 gen_{std::random_device{}()};
127     // uniform distribution
128     std::uniform_int_distribution<T> pdf_;
129 
130   }; // RandomSet
131 
132 } // demo::algo1
```

Source Code

```
1  // random-set.cc
2  // Mac Radigan
3  
4  
5    #include <assert.h>
6    #include <cstdlib>
7    #include <functional>
8    #include <iomanip>
9    #include <iostream>
10   #include <iterator>
11   #include <memory>
12   #include <random>
13   #include <stdexcept>
14   #include <sys/types.h>
15   #include <unordered_map>
16   #include <vector>
17 
```

```
18  // ============================================================================
19  // RandomSet
20  // ============================================================================
21  //
22  //   a set-like container supporting amortized constant time insertion,
23  //     removal, and uniform random element selection
24  //
25  // ----------------------------------------------------------------------------
26  //
27  // Background:
28  //
29  //   This algorithm makes use of the standard unordered map's (hash map
30  //     implementation) direct access for insert operations, and the standard
31  //     vector's efficient amortized time complexity random access for
32  //     randomly choosing an element.
33  //
34  //   This leaves only the need for removal from both the map and vector in
35  //     constant time.  This is already supported by the unordered map, but
36  //     for the vector is only true for back insertion/removal.  By
37  //     introducing a cell to contain the element of interest, we can then
38  //     swap the contents to be removed with the back of the vector prior to
39  //     removal (for a constant-time back-removal operation).  This leaves
40  //     only the need to identify the candidate cell of the vector for
41  //     deletion, which may be done directly by maintaining a reference in
42  //     the map.
43  //
44  //
45  // Implementation:
46  //
47  //   On insertion of x:T, check for existence of x in map M.  If not present,
48  //     insert a cell c containing x into the back of vector V, and add an
49  //     entry to the map mapping the element to the newly inserted cell,
50  //     say x->c.
51  //
52  //   On deletion of x:T, look up the containing cell c from the map,
53  //     say c = M[x].  Swap the contents of c with the cell at the back of
54  //     the vector.  Update the map from the newly swapped contents to the
55  //     cell c.  Remove x from the set.  Finally, remove the last element of
56  //     the vector.
57  //
58  // Performance:
59  //
60  //   insert  constant time complexity:                          O(1)
61  //   removal constant time complexity:                          O(1)
62  //   random selection amortized constant time complexity:       O(1)
```

```
63  //
64  //   linear space complexity:                                    O(N)
65  //
66  //
67  namespace demo::algo1 {
68
69    template<class T>
70    class RandomSet
71    {
72
73     // cell_t - a LISP-style container cell with a single content register
74     typedef struct cell_s
75     {
76       T cr;
77       cell_s(T &x) : cr(x) {};
78       inline void swap(struct cell_s &c) { std::swap(cr, c.cr); };
79     } cell_t;
80
81     public:
82
83      RandomSet()
84       : pdf_(0, std::numeric_limits<T>::max())
85       {};
86
87      // inserts an element into the set with constant time complexity
88      inline void insert(T x)
89      {
90        // If x is not already in the map, insert a cell containing x at the
91        //   back of the random vector.  Map x to the last cell in the vector.
92        if(map_.find(x) == map_.end())
93        {
94          pick_.push_back(cell_t(x));
95          map_.insert_or_assign(x, std::ref(pick_.back()));
96        }
97      } // insert
98
99      // removes an element from the set with constant time complexity
100     inline void remove(T x)
101     {
102       auto &top       = pick_.back(); // last element inserted
103       auto &candidate = map_.at((x)); // the x to be removed
104       candidate.get().swap(top);      // swap x to the back of the vector
105       // update the cell reference for the map for the cell reference
106       //   previously at the back of the vector
107       map_.emplace(candidate.get().cr, std::ref(candidate));
```

```
108         // remove x from map
109         map_.erase(x);
110         // remove x from vector (now at back of vector)
111         pick_.pop_back();
112       } // remove
113
114       // returns the number of elements in the set
115       inline std::size_t size() const
116       {
117         return map_.size();
118       } // size
119
120       // returns an element from the set with uniform random probability
121       //   in constant-time
122       inline T& get_random()
123       {
124         return pick_[pdf_(gen_) % pick_.size()].cr;
125       } // get_random
126
127       // prints the contents of the set
128       friend inline std::ostream& operator<<(std::ostream &os, const RandomSet<T> &o)
129       {
130         os << "{";
131         for(auto it=o.pick_.begin(); it!=o.pick_.end()-1; ++it) os << it->cr << ",";
132         os << o.pick_.back().cr << "}" << std::endl;
133         return os;
134       } // operator<<
135
136     private:
137
138       // a map from an element to a cell containing the element T
139       std::unordered_map<T, std::reference_wrapper<cell_t> > map_;
140       // a vector of cells (containing element T)
141       std::vector<cell_t> pick_;
142       // randomization source
143       std::mt19937 gen_{std::random_device{}()};
144       // uniform distribution
145       std::uniform_int_distribution<T> pdf_;
146
147     }; // RandomSet
148
149   } // demo::algo1
150
151   //
152   // main test driver
```

```
153   //
154   int main(int argc, char *argv[])
155   {
156
157     // default element type (domain)
158     typedef int64_t element_t;
159
160     demo::algo1::RandomSet<element_t> rset;
161
162     // insert
163     rset.insert(1);
164     rset.insert(3);
165     rset.insert(6);
166     rset.insert(8);
167
168     // remove
169     rset.remove(6);
170
171     // print
172     std::cout << rset;
173
174     // random selection
175     if( rset.size() > 0 )
176     {
177       for(int64_t k=0; k<10; ++k)
178       {
179         std::cout << "random x = "
180                   << rset.get_random()
181                   << std::endl;
182       }
183     }
184
185     return EXIT_SUCCESS;
186   } // main
187
188 // *EOF
```

Unit Test Results

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

```
random x = 8
random x = 3
random x = 1
random x = 8
random x = 8
random x = 8
random x = 3
```