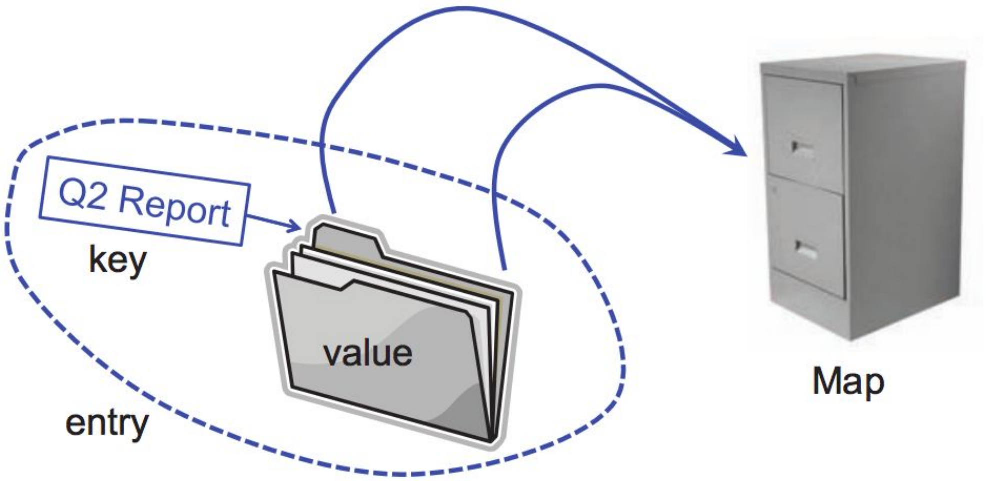
Data Structures & Programming

Maps

Golnar Sheikhshab

Maps



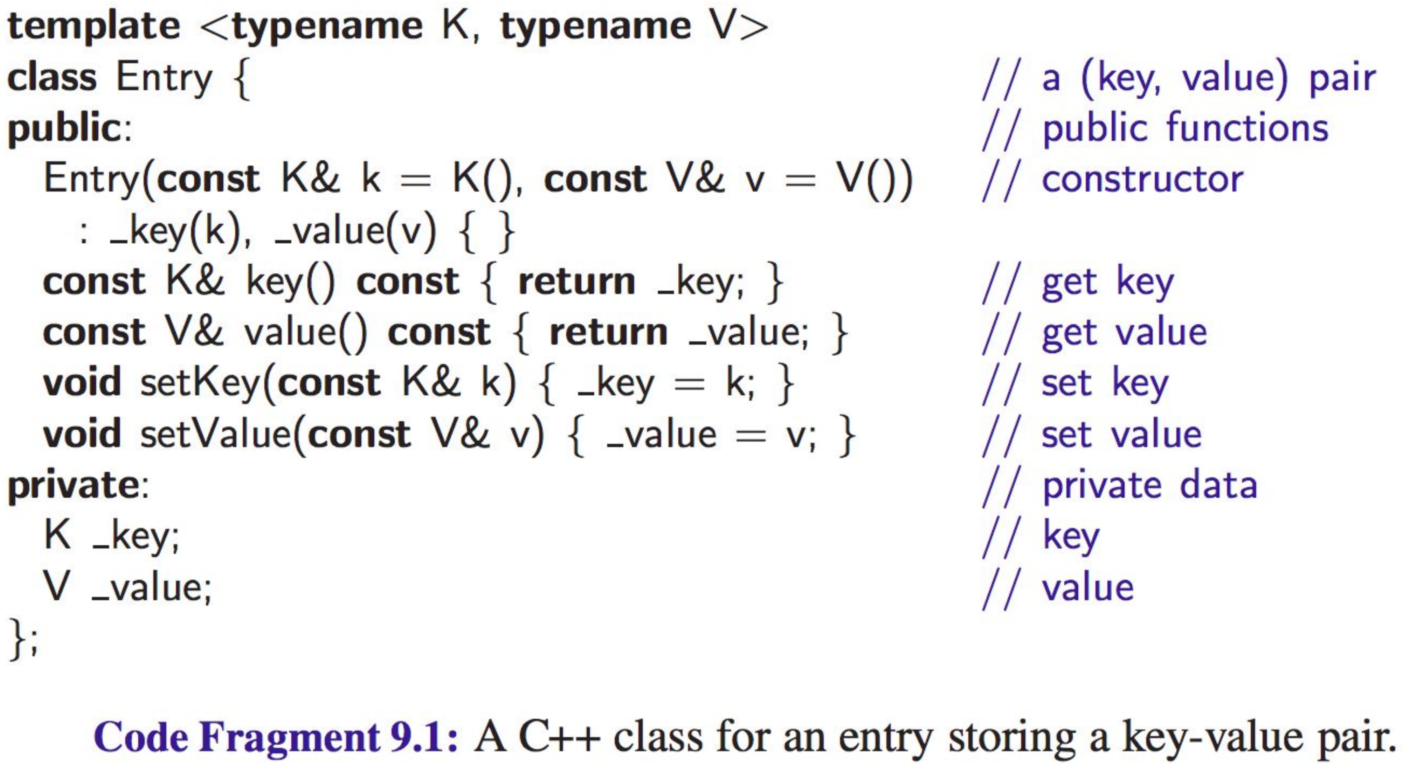
Store **key-values** (**entries**)

Locate them quickly

Keys must be **unique** (they are like indices or addresses)

Maps are sometimes called **associative store** or **associative container**s.

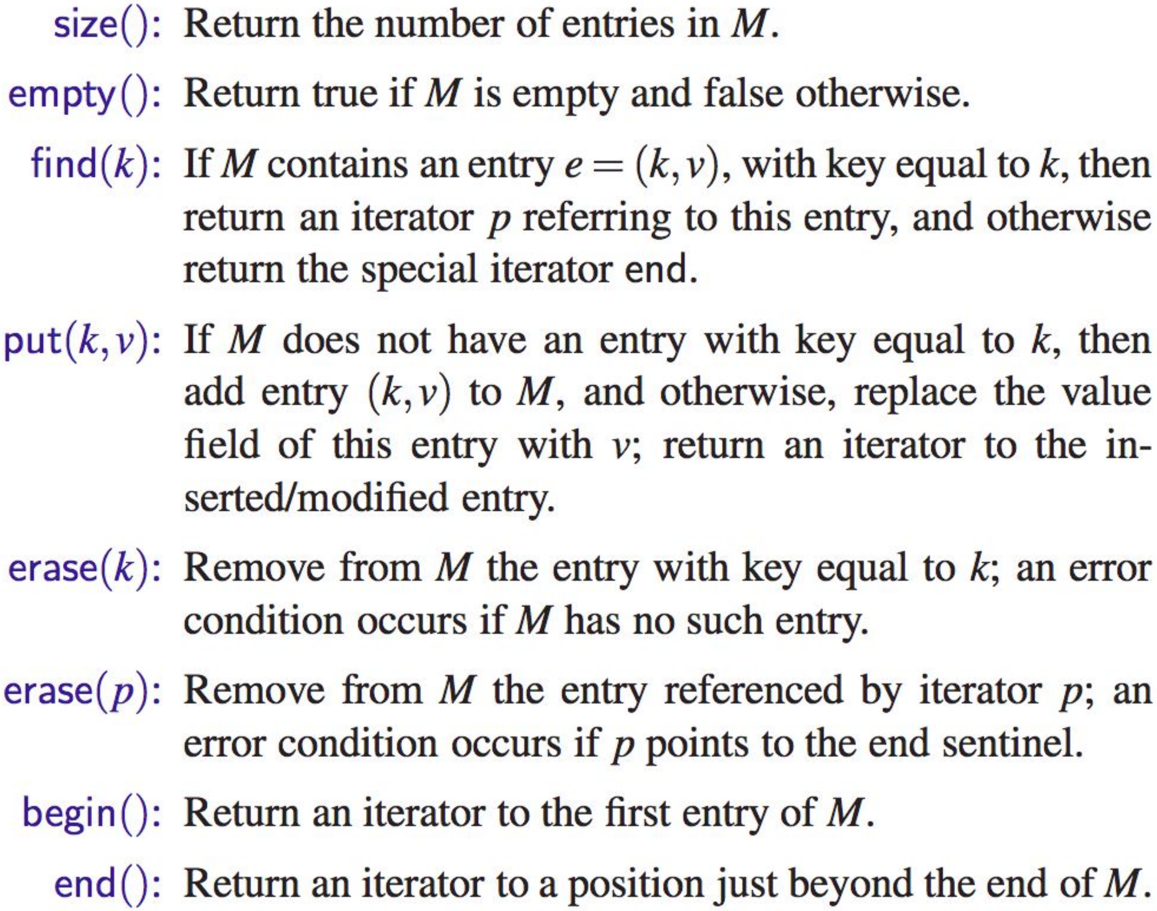
Map entry (key-value pair)



```
template <typename K, typename V>
class Entry {                                         // a (key, value) pair
public:                                               // public functions
  Entry(const K& k = K(), const V& v = V())           // constructor
    : _key(k), _value(v) { }
  const K& key() const { return _key; }               // get key
  const V& value() const { return _value; }           // get value
  void setKey(const K& k) { _key = k; }               // set key
  void setValue(const V& v) { _value = v; }           // set value
private:                                              // private data
  K _key;                                             // key
  V _value;                                           // value
};
```

Code Fragment 9.1: A C++ class for an entry storing a key-value pair.

Map ADT



size(): Return the number of entries in M.

empty(): Return true if M is empty and false otherwise.

find(k): If M contains an entry $e = (k, v)$, with key equal to k, then return an iterator p referring to this entry, and otherwise return the special iterator end.

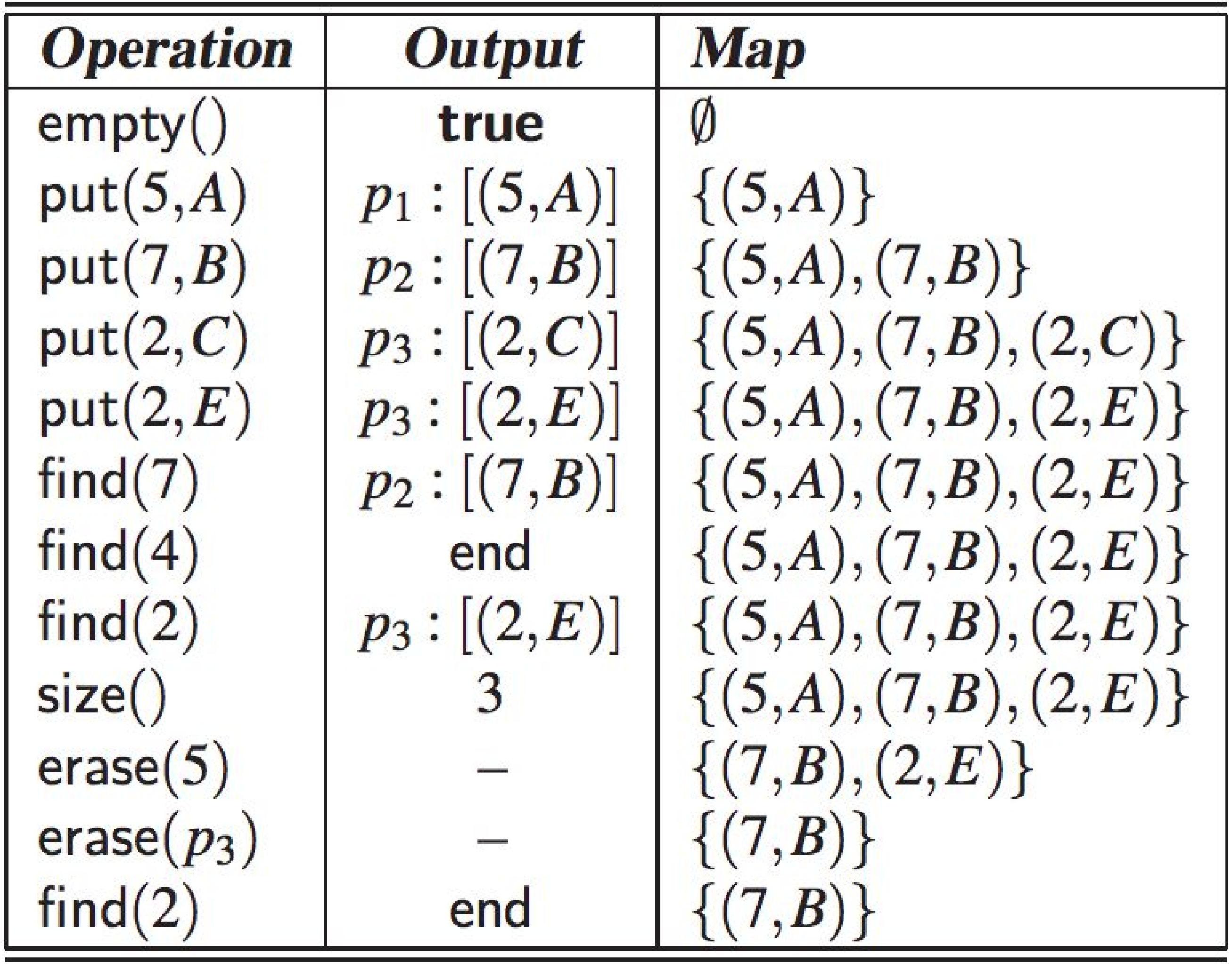
put(k, v): If M does not have an entry with key equal to k, then add entry (k, v) to M, and otherwise, replace the value field of this entry with v; return an iterator to the inserted/modified entry.

erase(k): Remove from M the entry with key equal to k; an error condition occurs if M has no such entry.

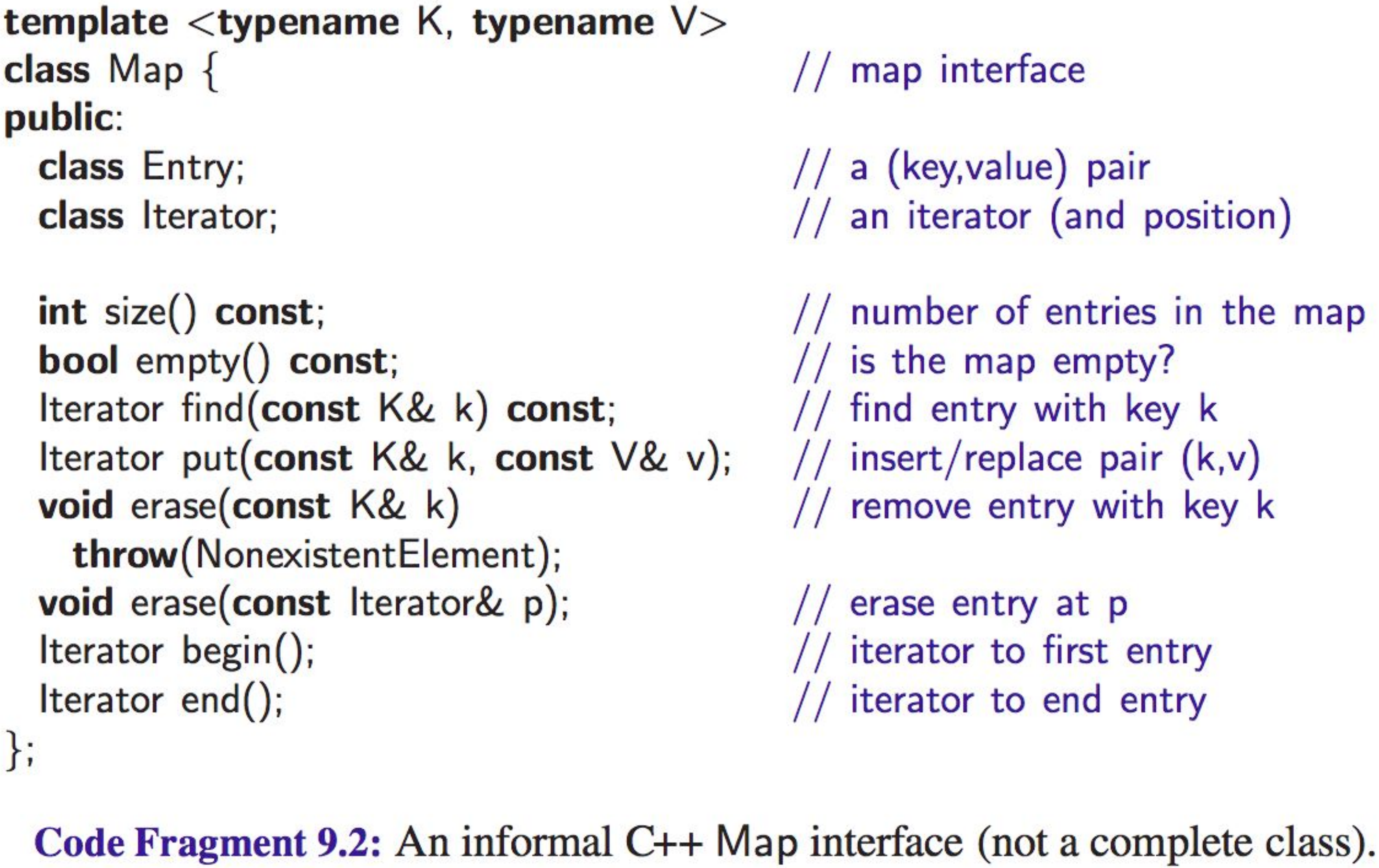
erase(p): Remove from M the entry referenced by iterator p; an error condition occurs if p points to the end sentinel.

begin(): Return an iterator to the first entry of M.

end(): Return an iterator to a position just beyond the end of M.



Operation	***Output***	***Map***
empty()	**true**	$\emptyset$
put$(5,A)$	$p_1 : [(5,A)]$	$\{(5,A)\}$
put$(7,B)$	$p_2 : [(7,B)]$	$\{(5,A),(7,B)\}$
put$(2,C)$	$p_3 : [(2,C)]$	$\{(5,A),(7,B),(2,C)\}$
put$(2,E)$	$p_3 : [(2,E)]$	$\{(5,A),(7,B),(2,E)\}$
find(7)	$p_2 : [(7,B)]$	$\{(5,A),(7,B),(2,E)\}$
find(4)	end	$\{(5,A),(7,B),(2,E)\}$
find(2)	$p_3 : [(2,E)]$	$\{(5,A),(7,B),(2,E)\}$
size()	3	$\{(5,A),(7,B),(2,E)\}$
erase(5)	–	$\{(7,B),(2,E)\}$
erase(p_3)	–	$\{(7,B)\}$
find(2)	end	$\{(7,B)\}$

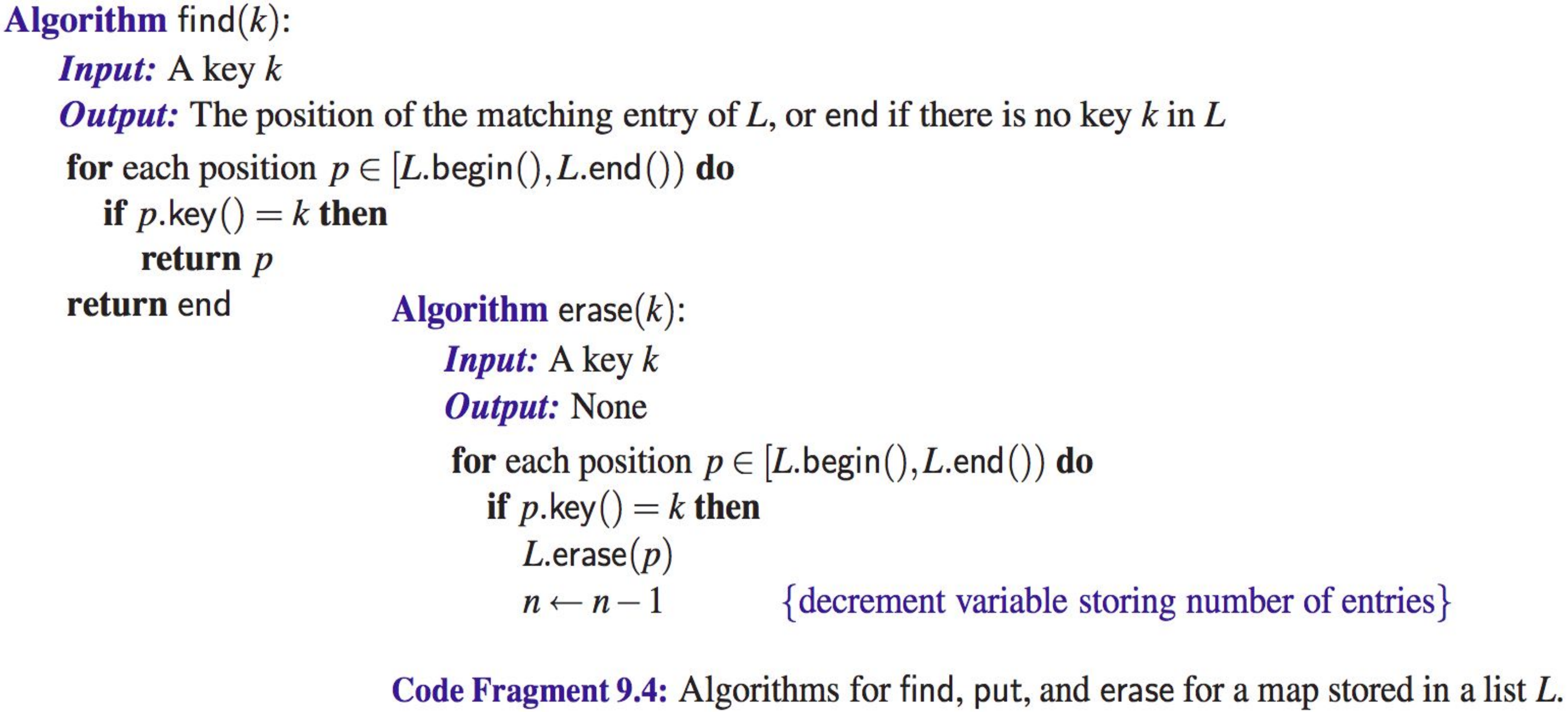


```
template <typename K, typename V>
class Map {                                   // map interface
public:
  class Entry;                                // a (key,value) pair
  class Iterator;                             // an iterator (and position)

  int size() const;                           // number of entries in the map
  bool empty() const;                         // is the map empty?
  Iterator find(const K& k) const;            // find entry with key k
  Iterator put(const K& k, const V& v);       // insert/replace pair (k,v)
  void erase(const K& k)                      // remove entry with key k
    throw(NonexistentElement);
  void erase(const Iterator& p);              // erase entry at p
  Iterator begin();                           // iterator to first entry
  Iterator end();                             // iterator to end entry
};
```

Code Fragment 9.2: An informal C++ Map interface (not a complete class).

List based implementation

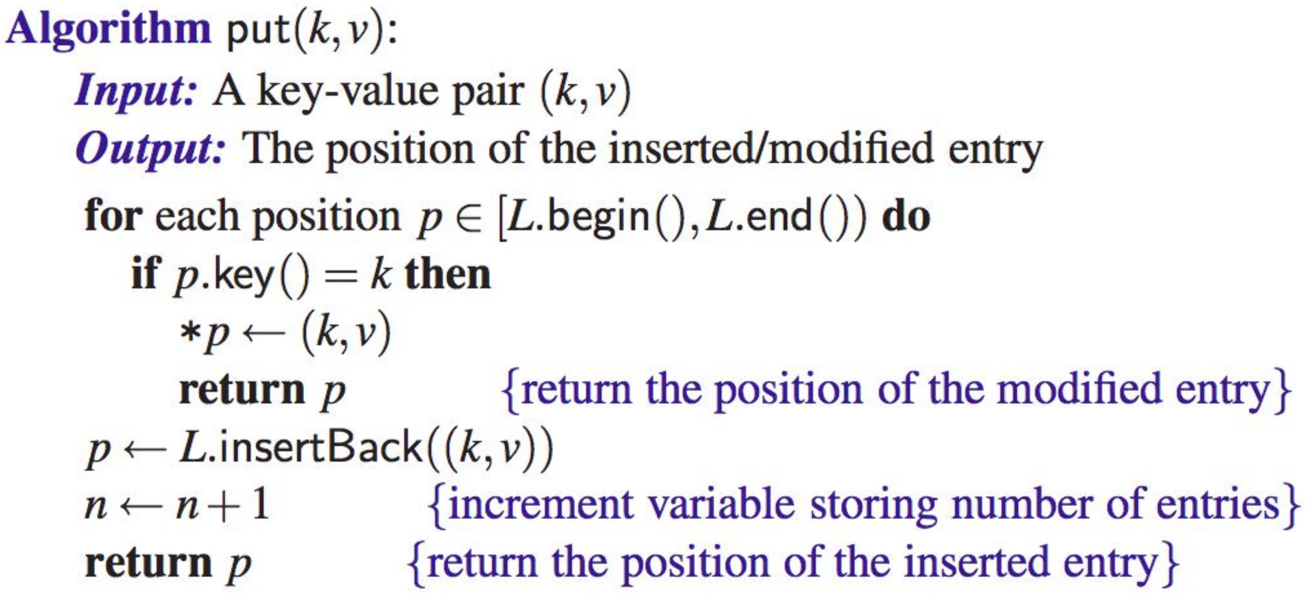


```
Algorithm find(k):
    Input: A key k
    Output: The position of the matching entry of L, or end if there is no key k in L
    for each position $p \in [L.\text{begin}(), L.\text{end}())$ do
        if $p.\text{key}() = k$ then
            return p
    return end
```

```
Algorithm erase(k):
    Input: A key k
    Output: None
    for each position $p \in [L.\text{begin}(), L.\text{end}())$ do
        if $p.\text{key}() = k$ then
            $L.\text{erase}(p)$
            $n \leftarrow n - 1$            {decrement variable storing number of entries}
```

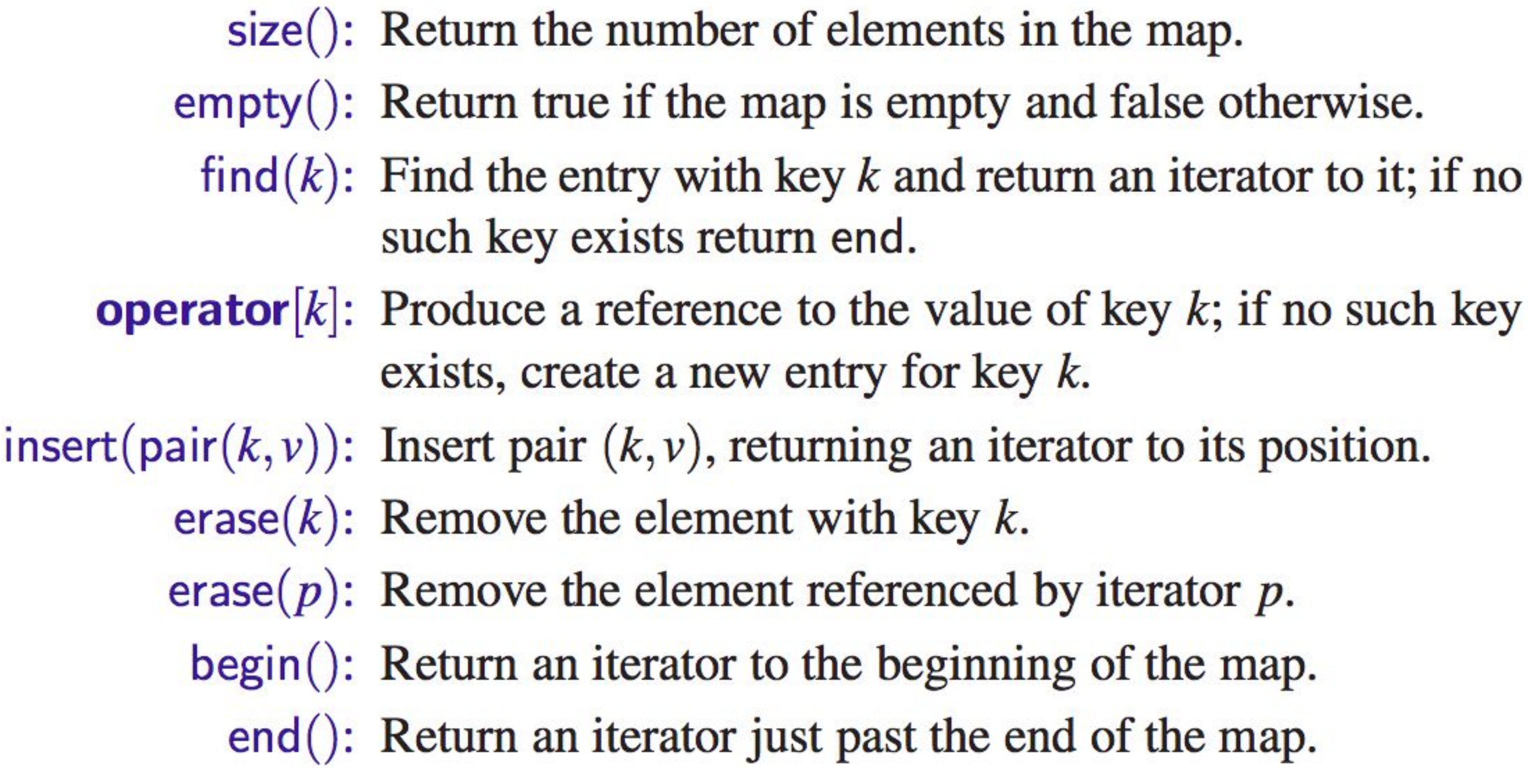
Code Fragment 9.4: Algorithms for find, put, and erase for a map stored in a list *L*.

List based implementation (continued)



```
Algorithm put($(k,v)$):
    Input: A key-value pair $(k,v)$
    Output: The position of the inserted/modified entry
    for each position $p \in [L.\text{begin}(), L.\text{end}())$ do
        if $p.\text{key}() = k$ then
            $*p \leftarrow (k,v)$
            return $p$          {return the position of the modified entry}
    $p \leftarrow L.\text{insertBack}((k,v))$
    $n \leftarrow n+1$          {increment variable storing number of entries}
    return $p$          {return the position of the inserted entry}
```

STL map functions (and operators)



size(): Return the number of elements in the map.

empty(): Return true if the map is empty and false otherwise.

find(k): Find the entry with key k and return an iterator to it; if no such key exists return end.

operator[k]: Produce a reference to the value of key k; if no such key exists, create a new entry for key k.

insert(pair(k, v)): Insert pair (k, v), returning an iterator to its position.

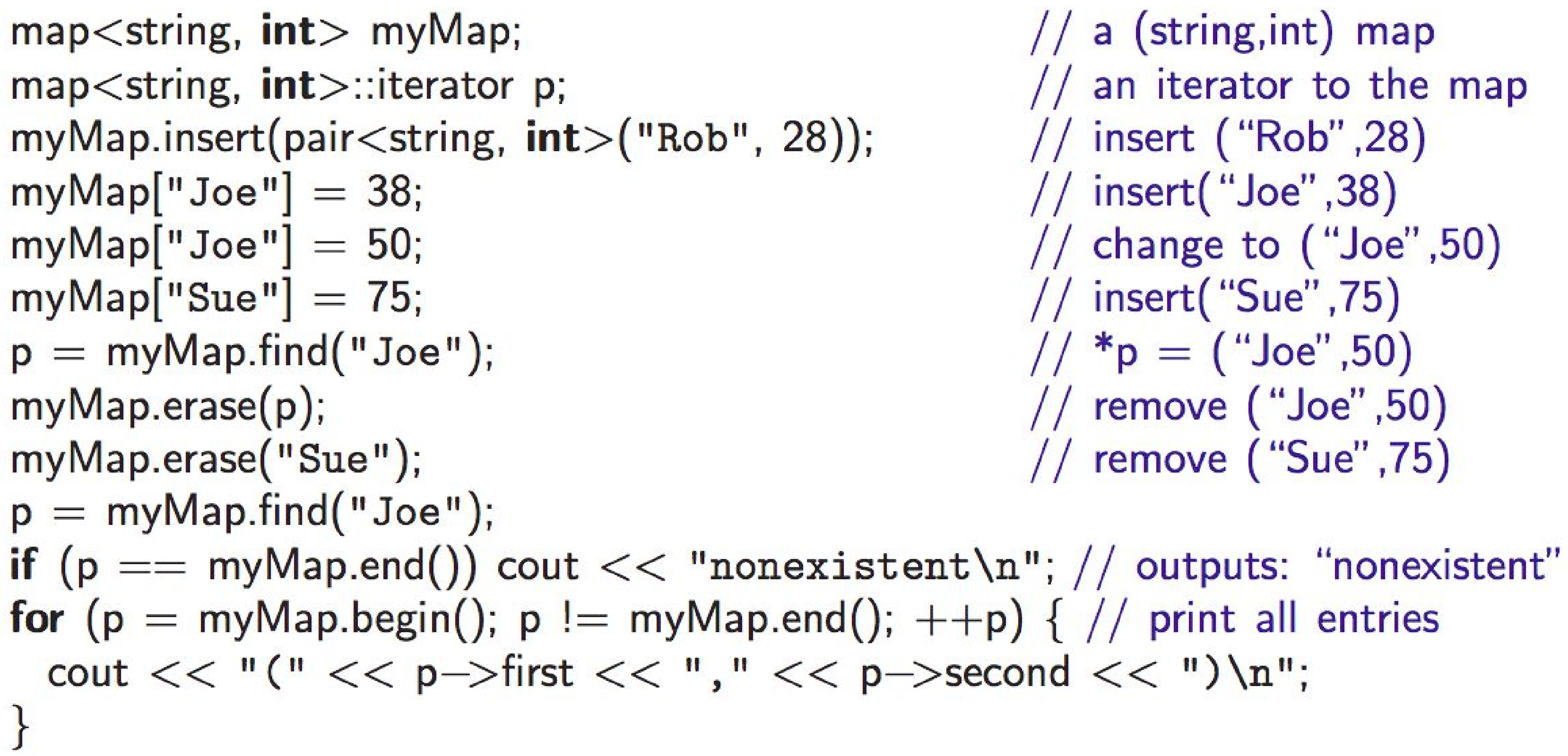
erase(k): Remove the element with key k.

erase(p): Remove the element referenced by iterator p.

begin(): Return an iterator to the beginning of the map.

end(): Return an iterator just past the end of the map.

An example of STL map in use



```
map<string, int> myMap;                               // a (string,int) map
map<string, int>::iterator p;                         // an iterator to the map
myMap.insert(pair<string, int>("Rob", 28));           // insert (“Rob” ,28)
myMap["Joe"] = 38;                                    // insert(“Joe” ,38)
myMap["Joe"] = 50;                                    // change to (“Joe” ,50)
myMap["Sue"] = 75;                                    // insert(“Sue” ,75)
p = myMap.find("Joe");                                // *p = (“Joe” ,50)
myMap.erase(p);                                       // remove (“Joe” ,50)
myMap.erase("Sue");                                   // remove (“Sue” ,75)
p = myMap.find("Joe");
if (p == myMap.end()) cout << "nonexistent\n"; // outputs: “nonexistent”
for (p = myMap.begin(); p != myMap.end(); ++p) { // print all entries
  cout << "(" << p->first << "," << p->second << ")\n";
}
```

Reading Material

Section 9.1 of the textbook