CH9. HASH TABLES, MAPS, AND SKIP LISTS

CSED233 Data Structure

Prof. Hwanjo Yu
POSTECH

The Map ADT

- A collection of key-value pairs, also called entries

```
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
};
```

```
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
};
```

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	$\{(7,B)\}$

The STL map Class

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.

```
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";
}
```

- Implement Map using List? => O(n)

Hash Tables

- Bucket Array and Hash Function:
  - *Store (k, v) into the bucket A[h(k)]*
  - *Hash function h(k) determines the location (array index) of the entry (k,v) in the bucket array*

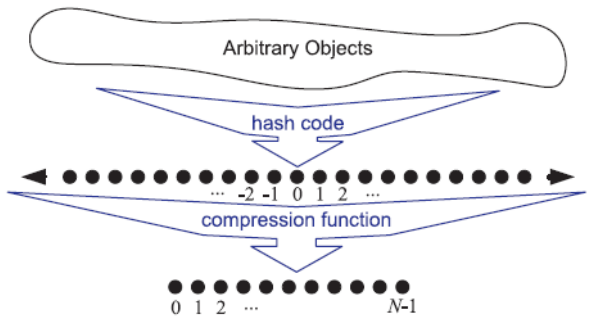


Figure 9.3: The two parts of a hash function: hash code and compression function.

- Hash code
  - *Integer assigned to k (could be even negative, but avoid collision)*
  - *Convert (char, short, long, ..) into integer*
  - *Polynomial hash code for strings*
    - $x_0 a^{k-1} + x_1 a^{k-2} + \cdots + x_{k-2} a + x_{k-1}$
    - $(\ldots((a x_0 + x_1)a + x_2)a + \cdots + x_{k-2})a + x_{k-1}$
    - 33, 37, 39, and 41 are good choices for English strings producing < 7 collisions for 50k words
  - *Cyclic shift hash codes, e.g., to shift 5 bits,*
    - Produce 4, 6 collisions with 5, 6 bit shifts
    - But 23,739 collisions with 0 bit shift (simple sum)
  - *Use reinterpret cast for float*

```
int hashCode(const char* p, int len) {
  unsigned int h = 0;
  for (int i = 0; i < len; i++) {
    h = (h << 5) | (h >> 27);
    h += (unsigned int) p[i];
  }
  return hashCode(int(h));
}
```

```
int hashCode(const float& x) {              // hash a float
  int len = sizeof(x);
  const char* p = reinterpret_cast<const char*>(&x);
  return hashCode(p, len);
}
```

Compression function, Collision handling

- Compression function
  - *Division method to make it into [0, N-1]*
    - $h(k) = |k| \ mod \ N$, N is a prime number.
  - *MAD method (Multiply Add and Divide)*
    - $h(k) = |ak + b| \ mod \ N, a \ mod \ N \neq 0$
- Collision handling
  - ***Separate chaining***: use list
  - ***Open addressing***: use only bucket
    - Linear probing

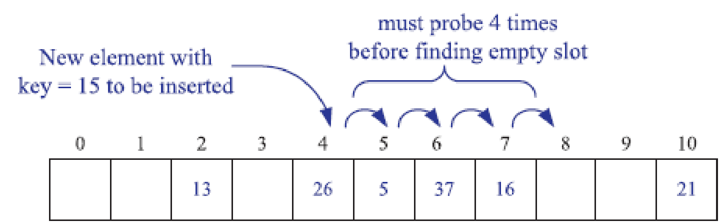


Figure 9.5: An insertion into a hash table using linear probing to resolve collisions. Here we use the compression function $h(k) = k \bmod 11$.

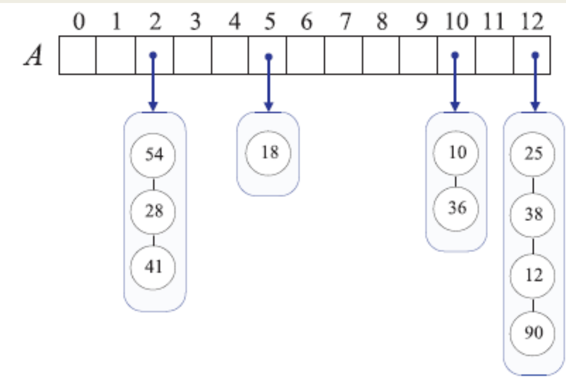


Figure 9.4: A hash table of size 13, storing 10 entries with integer keys, with collisions resolved by separate chaining. The compression function is $h(k) = k \bmod 13$. For simplicity, we do not show the values associated with the keys.

    - Quadratic probing
    - Double hashing
  - *If memory space is not a major issue, separate chaining*
- Load factor: $\lambda = \frac{n}{N} < 0.5 \ (open \ addressing), or < 0.9 \ (separate \ chaining)$
  - *If well distributed, O(n/N) thus O(1)*

C++ Hash Table implementation

```
template <typename K, typename V, typename H>
class HashMap {
public:                                              //
  typedef Entry<const K,V> Entry;                    //
  class Iterator;                                    //
public:                                              //
  HashMap(int capacity = 100);                       //
  int size() const;                                  //
  bool empty() const;                                //
  Iterator find(const K& k);                         //
  Iterator put(const K& k, const V& v);              //
  void erase(const K& k);                            //
  void erase(const Iterator& p);                     //
  Iterator begin();                                  //
  Iterator end();                                    //
protected:                                           //
  typedef std::list<Entry> Bucket;                   //
  typedef std::vector<Bucket> BktArray;              //
  // ...insert HashMap utilities here
private:
  int n;                                             //
  H hash;                                            //
  BktArray B;                                        //
public:                                              //
  // ...insert Iterator class declaration here
};
```

```
Iterator finder(const K& k);
Iterator inserter(const Iterator& p, const Entry& e);
void eraser(const Iterator& p);
typedef typename BktArray::iterator BItor;
typedef typename Bucket::iterator EItor;
static void nextEntry(Iterator& p)
  { ++p.ent; }
static bool endOfBkt(const Iterator& p)
  { return p.ent == p.bkt->end(); }
```

```
class Iterator {                                     // an iterator (& position)
private:
  EItor ent;                                         // which entry
  BItor bkt;                                         // which bucket
  const BktArray* ba;                                // which bucket array
public:
  Iterator(const BktArray& a, const BItor& b, const EItor& q = EItor())
    : ent(q), bkt(b), ba(&a) { }
  Entry& operator*() const;                          // get entry
  bool operator==(const Iterator& p) const;          // are iterators equal?
  Iterator& operator++();                            // advance to next entry
  friend class HashMap;                              // give HashMap access
};
```

```
/* HashMap⟨K,V,H⟩ :: */                              // advance to next entry
  Iterator& Iterator::operator++() {
    ++ent;                                           // next entry in bucket
    if (endOfBkt(*this)) {                           // at end of bucket?
      ++bkt;                                         // go to next bucket
      while (bkt != ba->end() && bkt->empty())       // find nonempty bucket
        ++bkt;
      if (bkt == ba->end()) return *this;            // end of bucket array?
      ent = bkt->begin();                            // first nonempty entry
    }
    return *this;                                    // return self
  }
```

Ordered Maps (Ch. 9.3)

- Ordered search tables
  - $O(n)$ *for insert and delete*
  - $O(\log n)$ *for search using binary search*
- Binary search

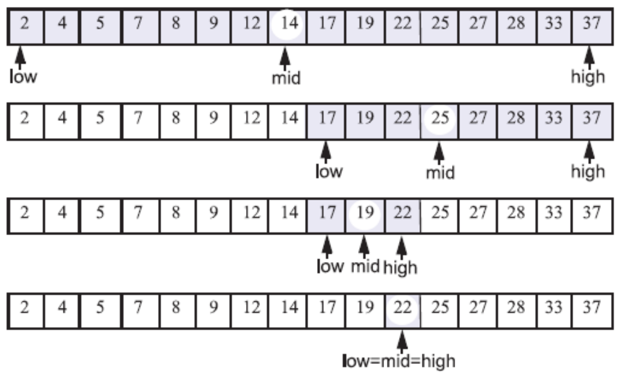


Figure 9.7: Example of a binary search to perform operation find(22), in a map with integer keys, implemented with an ordered vector. For simplicity, we show the keys, not the whole entries.

Algorithm BinarySearch($L, k, low, high$):
  Input: An ordered vector L storing n entries and integers low and $high$
  Output: An entry of L with key equal to k and index between low and $high$, if such an entry exists, and otherwise the special sentinel end

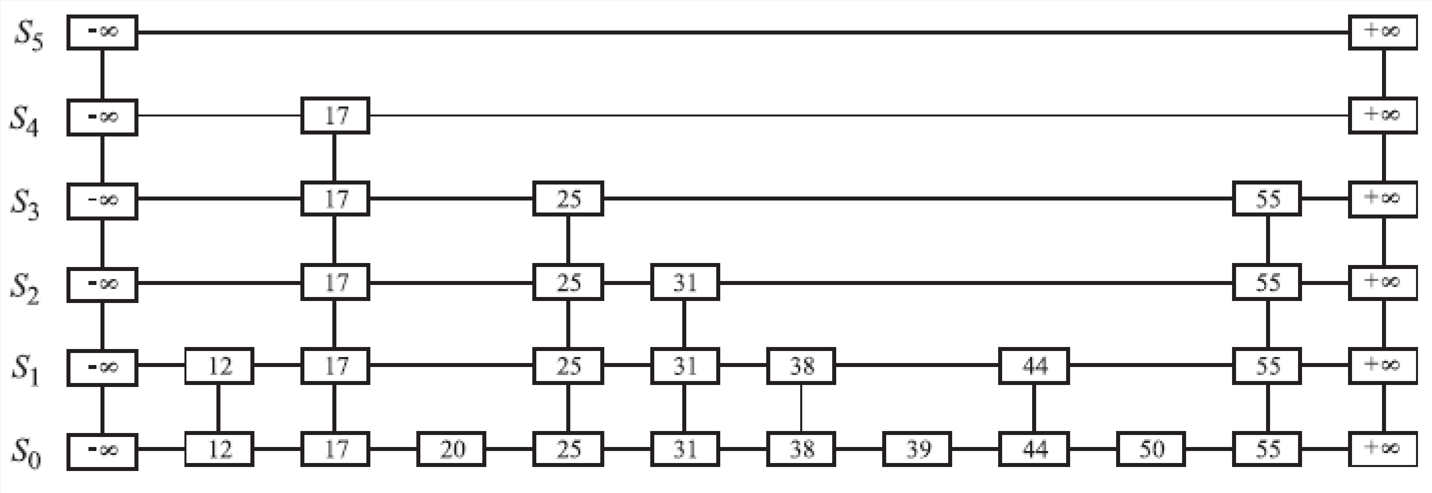
if $low > high$ **then**
  return end
else
  $mid \leftarrow \lfloor (low + high)/2 \rfloor$
  $e \leftarrow L.\text{at}(mid)$
  if $k = e.\text{key}()$ **then**
    return e
  else if $k < e.\text{key}()$ **then**
    return BinarySearch($L, k, low, mid - 1$)
  else
    return BinarySearch($L, k, mid + 1, high$)

Code Fragment 9.18: Binary search in an ordered vector.

- Comparing Map implementations

Method	*List*	*Hash Table*	*Search Table*
size, empty	$O(1)$	$O(1)$	$O(1)$
find	$O(n)$	$O(1)$ exp., $O(n)$ worst-case	$O(\log n)$
insert	$O(1)$	$O(1)$	$O(n)$
erase	$O(n)$	$O(1)$ exp., $O(n)$ worst-case	$O(n)$

Skip Lists (Ch. 9.4)



- $O(\log n)$ for search and update on average
- Traversing operations: after(p), before(p), below(p), above(p)
- Search starts from the top-left position
- Insertion first searches the position and inserts it. Upper level decide whether to insert randomly.

Dictionaries (Ch. 9.5)

- Similar to Map but with allowing multiple entries having the same key.

Operation	Output	Dictionary
empty()	**true**	$\emptyset$
insert$(5,A)$	$p_1 : [(5,A)]$	$\{(5,A)\}$
insert$(7,B)$	$p_2 : [(7,B)$	$\{(5,A),(7,B)\}$
insert$(2,C)$	$p_3 : [(2,C)$	$\{(5,A),(7,B),(2,C)\}$
insert$(8,D)$	$p_4 : [(8,D)$	$\{(5,A),(7,B),(2,C),(8,D)\}$
insert$(2,E)$	$p_5 : [(2,E)$	$\{(5,A),(7,B),(2,C),(2,E),(8,D)\}$
find(7)	$p_2 : [(7,B)$	$\{(5,A),(7,B),(2,C),(2,E),(8,D)\}$
find(4)	end	$\{(5,A),(7,B),(2,C),(2,E),(8,D)\}$
find(2)	$p_3 : [(2,C)$	$\{(5,A),(7,B),(2,C),(2,E),(8,D)\}$
findAll(2)	(p_3,p_4)	$\{(5,A),(7,B),(2,C),(2,E),(8,D)\}$
size()	5	$\{(5,A),(7,B),(2,C),(2,E),(8,D)\}$
erase(5)	–	$\{(7,B),(2,C),(2,E),(8,D)\}$
erase(p_3)	–	$\{(7,B),(2,E),(8,D)\}$
find(2)	$p_5 : [(2,E)]$	$\{(7,B),(2,E),(8,D)\}$

The operation findAll(2) *returns the iterator pair* (p_3,p_4), *referring to the entries* $(2,C)$ *and* $(8,D)$. *Assuming that the entries are stored in the order listed above, iterating from* p_3 *up to, but not including,* p_4, *would enumerate the entries* $\{(2,C),(2,E)\}$.

Dictionary implementation

- Similar to Map but with allowing multiple entries having the same key.

```
template <typename K, typename V, typename H>
class HashDict : public HashMap<K,V,H> {
public:                                              // public functions
  typedef typename HashMap<K,V,H>::Iterator Iterator;
  typedef typename HashMap<K,V,H>::Entry Entry;
  // ...insert Range class declaration here
public:                                              // public functions
  HashDict(int capacity = 100);                      // constructor
  Range findAll(const K& k);                         // find all entries with k
  Iterator insert(const K& k, const V& v);           // insert pair (k,v)
};
```

Code Fragment 9.24: The class HashDict, which implements the dictionary ADT.

```
/* HashDict⟨K,V,H⟩ :: */
  Iterator insert(const K& k, const V& v) {
    Iterator p = finder(k);
    Iterator q = inserter(p, Entry(k, v));
    return q;
  }
```

```
class Range {                                        // an iterator range
private:
  Iterator _begin;                                   // front of range
  Iterator _end;                                     // end of range
public:
  Range(const Iterator& b, const Iterator& e)        // constructor
    : _begin(b), _end(e) { }
  Iterator& begin() { return _begin; }               // get beginning
  Iterator& end() { return _end; }                   // get end
};
```

```
/* HashDict⟨K,V,H⟩ :: */                             // fi
  Range findAll(const K& k) {
    Iterator b = finder(k);                          // l
    Iterator p = b;
    while (!endOfBkt(p) && (*p).key() == (*b).key()) {
      ++p;
    }
    return Range(b, p);                              // r
  }
```