Hashing

Prof. Ki-Hoon Lee
Dept. of Computer Engineering
Kwangwoon University

Dictionaries

- Collection of pairs.
  - (key, element)
  - Pairs have different keys.
- Operations.
  - Get(theKey)
  - Delete(theKey)
  - Insert(theKey, theElement)

Hash Tables

- Worst-case time for Get, Insert, and Delete is O(n).
  - n: the number of pairs in a dictionary
- Expected time is O(1).

Ideal Hashing

- Uses a 1D array (or table) table[0:b-1].
  - Each position of this array is a bucket.
  - b is the number of buckets in the table.
  - A bucket can normally hold only one dictionary pair.
- Uses a hash function f that converts each key k into an index in the range [0, b-1].
  - f(k) is the home bucket for key k.
- Every dictionary pair (key, element) is stored in its home bucket table[f(key)].

Ideal Hashing Example

- Pairs are: (22,a), (33,c), (3,d), (73,e), (85,f).
- Hash table is table[0:7], b = 8.
- Hash function is key/11.
- Pairs are stored in table as below:

(3,d)		(22,a)	(33,c)			(73,e)	(85,f)
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]

- Get, Insert, and Delete take O(1) time.

What Can Go Wrong?

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]
(3,d)		(22,a)	(33,c)			(73,e)	(85,f)

- Where does (26,g) go?
- Keys that have the same home bucket are synonyms.
  - 22 and 26 are synonyms with respect to the hash function that is in use.
- The home bucket for (26,g) is already occupied.

What Can Go Wrong?

(3,d)		(22,a)	(33,c)			(73,e)	(85,f)

- A collision occurs when the home bucket for a new pair is occupied by a pair with a different key.
- An overflow occurs when there is no space in the home bucket for the new pair.
- When a bucket can hold only one pair, collisions and overflows occur together.
- Need a method to handle overflows.

Hash Table Issues

- Choice of hash function.
- Overflow handling method.
- Size (number of buckets) of hash table.

Hash Functions

- Two parts:
  - Convert key into a nonnegative integer in case the key is not an integer.
    - Done by the function hash().
  - Map an integer into a home bucket.
    - f(k) is an integer in the range [0, b-1], where b is the number of buckets in the table.

String to Integer

- Each character is 1 byte long.
- An int is 4 bytes.
- A 2 character string s may be converted into a unique 4 byte non-negative int using the code:

```
int answer = s.at(0);
answer = (answer << 8) + s.at(1);
```

- Strings that are longer than 3 characters do not have a unique non-negative int representation.

String to Nonnegative Integer

```
template<>
class hash<string>
{
  public:
  size_t operator()(const string theKey) const
  {// Convert theKey to a nonnegative integer.
      unsigned long hashValue = 0;
      int length = (int) theKey.length();
      for (int i = 0; i < length; i++)
          hashValue = 5 * hashValue +
                      theKey.at(i);

      return size_t(hashValue);
   }
};
```

Map into a Home Bucket

(3,d)		(22,a)	(33,c)			(73,e)	(85,f)
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]

- Most common method is by division.

homeBucket = hash(theKey) % divisor;

- divisor equals the number of buckets b.
- 0 <= homeBucket < divisor = b

Uniform Hash Function

(3,d)		(22,a)	(33,c)			(73,e)	(85,f)
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]

- Let keySpace be the set of all possible keys.
- A uniform hash function maps the keys in keySpace into buckets such that approximately the same number of keys get mapped into each bucket.

Uniform Hash Function

(3,d)		(22,a)	(33,c)			(73,e)	(85,f)
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]

- Equivalently, the probability that a randomly selected key has bucket i as its home bucket is 1/b, 0 <= i < b.
- A uniform hash function minimizes the likelihood of an overflow when keys are selected at random.

Hashing by Division

- keySpace = all ints.
- For every b, the number of ints that get mapped (hashed) into bucket i is approximately $2^{32}/b$.
- Therefore, the division method results in a uniform hash function when keySpace = all ints.
- In practice, keys tend to be correlated.
- So, the choice of the divisor b affects the distribution of home buckets.

Selecting the Divisor

- Because of this correlation, applications tend to have a bias towards keys that map into odd integers (or into even ones).
- When the divisor is an even number, odd integers hash into odd home buckets and even integers into even home buckets.
  - 20%14 = 6, 30%14 = 2, 8%14 = 8
  - 15%14 = 1, 3%14 = 3, 23%14 = 9
- The bias in the keys results in a bias toward either the odd or even home buckets.

Selecting the Divisor (cont.)

- When the divisor is an odd number, odd (even) integers may hash into any home.
  - 20%15 = 5, 30%15 = 0, 8%15 = 8
  - 15%15 = 0, 3%15 = 3, 23%15 = 8
- The bias in the keys does not result in a bias toward either the odd or even home buckets.
- Better chance of uniformly distributed home buckets.
- So do not use an even divisor.

Selecting the Divisor (cont.)

- Similar biased distribution of home buckets is seen, in practice, when the divisor is a multiple of prime numbers such as 3, 5, 7, …
- The effect of each prime divisor p of b decreases as p gets larger.
- Ideally, choose b so that it is a prime number.
- Alternatively, choose b so that it has no prime factor smaller than 20.

Overflow Handling

- An overflow occurs when the home bucket for a new pair (key, element) is full.
- We may handle overflows by:
  - Search the hash table in some systematic fashion for a bucket that is not full.
    - Linear probing (linear open addressing).
    - Quadratic probing.
    - Random probing.
  - Eliminate overflows by permitting each bucket to keep a list of all pairs for which it is the home bucket.
    - Array linear list.
    - Chain.

Linear Probing

- When inserting a new pair whose key is k, we search the hash table buckets in the order, $ht[(h(k) + i)\%b],\ 0 \le i \le b-1$
  - ht: hash table
  - h: hash function
  - b: the number of buckets
- This search terminates when we reach the first unfilled bucket and the new pair is inserted into this bucket
- In case no such bucket is found, the hash table is full and it is necessary to increase the table size

Linear Probing

Put(N,7)

Value	Key
RID	R
0	B
1	O
2	E
3	P
4	V
5	L
6	X
7	N
8	K
9	M

Hash(key)
0
2
3
3
9
10
0
1
9
0

First
Empty
Slot ?

	KEY	VAL
0	B	0
1	X	6
2	O	1
3	E	2
4	P	3
5	N	7
6		
7		
8		
9	V	4
10	L	5

%M

Linear Probing Example

	insert(76) 76%7 = 6	insert(93) 93%7 = 2	insert(40) 40%7 = 5	insert(47) 47%7 = 5	insert(10) 10%7 = 3	insert(55) 55%7 = 6
0				47	47	47
1						55
2		93	93	93	93	93
3					10	10
4						
5			40	40	40	40
6	76	76	76	76	76	76
probes:	1	1	1	3	1	3

Search with Linear Probing

- Consider a hash table A that uses linear probing
- get(k)
  - We start at cell $h(k)$
  - We probe consecutive locations until one of the following occurs
    - An item with key k is found, or
    - An empty cell is found, or
    - N cells have been unsuccessfully probed

Algorithm *get*(k)
$i \leftarrow h(k)$
$p \leftarrow 0$
repeat
$c \leftarrow A[i]$
if $c = \varnothing$
return *null*
else if $c.key() = k$
return $c.element()$
else
$i \leftarrow (i+1) \bmod N$
$p \leftarrow p + 1$
until $p = N$
return *null*

Program 8.4:Linear probing

```
========================================
template <class K, class E>
pair<K,E>* LinearProbing<K,E>::Get(const K& k)
{
// Search the linear probing hash table ht
// (each bucket has exactly one slot) for k.
// If a pair with this key is found,
// return a pointer to this pair;
// otherwise, return NULL.
  int i = h(k);    // home bucket
  int j;
  for (j = i; ht[j] && ht[j]->first != k;) {
    j = (j + 1) % b;          // treat the table as circular
    if (j == i) return NULL;// back to start point
  }
  if (ht[j]->first == k) return ht[j];
  return NULL;
}
========================================
```

Performance of Linear Probing

0				4				8				12				16
34	0	45				6	23	7			28	12	29	11	30	33

- Worst-case find/insert/erase time is O(n), where n is the number of pairs in the table.
- This happens when all pairs are in the same cluster.

Expected Performance

0				4				8				12				16
34	0	45				6	23	7			28	12	29	11	30	33

- α = loading density = (the number of pairs)/b.
  - $\alpha = 12/17$.
- S_n = expected number of buckets examined in a successful search when n is large
- U_n = expected number of buckets examined in an unsuccessful search when n is large
- Time to insert governed by U_n.

Expected Performance

- $S_n \approx ½(1 + 1/(1 - \alpha))$
- $U_n \approx ½(1 + 1/(1 - \alpha)^2)$
- Note that $0 <= \alpha <= 1$.

alpha	S_n	U_n
0.50	1.5	2.5
0.75	2.5	8.5
0.90	5.5	50.5

$\alpha <= 0.75$ is recommended.

Hash Table Design

- Performance requirements are given, determine maximum permissible loading density.
- We want a successful search to make no more than 10 compares (expected).
  - $S_n \approx \frac{1}{2}(1 + 1/(1 - \alpha))$
  - $\alpha <= 18/19$
- We want an unsuccessful search to make no more than 13 compares (expected).
  - $U_n \approx \frac{1}{2}(1 + 1/(1 - \alpha)^2)$
  - $\alpha <= 4/5$
- So $\alpha <= \min\{18/19, 4/5\} = 4/5$.

Hash Table Design

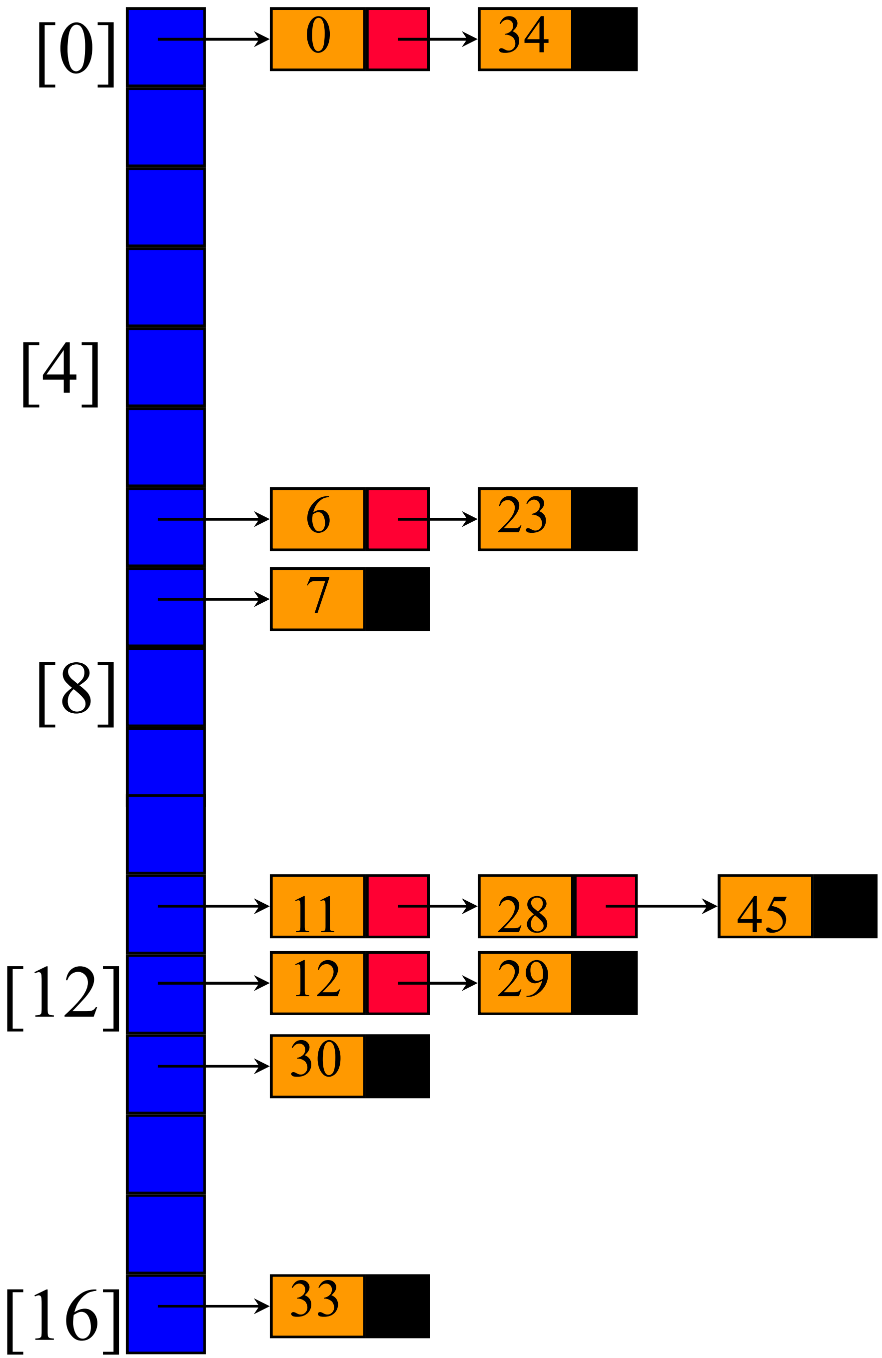
- Dynamic resizing of table.
  - Whenever loading density exceeds threshold (4/5 in our example), rehash into a table of approximately twice the current size.
- Fixed table size. • α = (the number of pairs)/b
  - Know maximum number of pairs.
  - No more than 1000 pairs.
  - Loading density <= 4/5 => b >= 5/4*1000 = 1250.
  - Pick b (equal to divisor) to be a prime number or an odd number with no prime divisors smaller than 20.

Linear List of Synonyms

- Each bucket keeps a linear list of all pairs for which it is the home bucket.
- The linear list may or may not be sorted by key.
- The linear list may be an array linear list or a chain (i.e., linked list).

Sorted Chains

- Put in pairs whose keys are 6, 12, 34, 29, 28, 11, 23, 7, 0, 33, 30, 45
- Home bucket = key % 17.



```
Program 8.5:Chain search
======================================
template <class K, class E>
pair<K,E>* Chaining<K,E>::Get(const K& k)
{
// Search the chained hash table ht for k.
// If a pair with this key is found,
// return a pointer to this pair;
// otherwise, return NULL.
  int i = h(k);    //  home bucket
  // search the chain ht[i]
  for (ChainNode<pair<K,E> >* current = ht[i];
       current;
       current = current->link)
    if (current->data.first == k)
       return &current->data;
  return NULL;
}
======================================
```

STL unordered_map (hash_map)

```
#include <iostream>
#include <string>
#include <unordered_map>

int main()
{
    // Create an unordered_map of three strings (that map to strings)
    std::unordered_map<std::string, std::string> u = {
        {"RED","#FF0000"},
        {"GREEN","#00FF00"},
        {"BLUE","#0000FF"}
    };

    // Iterate and print keys and values of unordered_map
    for( const auto& n : u ) {
        std::cout << "Key:[" << n.first << "] Value:[" << n.second << "]\n";
    }

    // Add two new entries to the unordered_map
    u["BLACK"] = "#000000";
    u["WHITE"] = "#FFFFFF";

    // Output values by key
    std::cout << "The HEX of color RED is:[" << u["RED"] << "]\n";
    std::cout << "The HEX of color BLACK is:[" << u["BLACK"] << "]\n";

    return 0;
}
```

Output:

```
Key:[RED] Value:[#FF0000]
Key:[BLUE] Value:[#0000FF]
Key:[GREEN] Value:[#00FF00]
The HEX of color RED is:[#FF0000]
The HEX of color BLACK is:[#000000]
```

auto specifier (since C++11)

- For variables, specifies that the type of the variable that is being declared will be automatically deduced from its initializer.

```
auto d = 5.0; // 5.0 is a double literal, so d will be type double
auto i = 1 + 2; // 1 + 2 evaluates to an integer, so i will be type int
```

`auto` (자동범위변수)

- 일반적인 지역 변수 형태로 블럭 안에서만 유효하며 블럭의 실행이 끝나면 소멸
- 스택에 메모리 할당
- 일반적으로 C에서 `auto` 키워드는 생략되어있음. 즉 아무 표시하지 않은 변수는 `auto` 와 같은 의미.
- C++에서 auto 키워드를 사용할 경우 "자동 타입 추론"이라는 완전히 다른 의미를 가지게 되므로 주의할것.

```
int add(int x, int y)
{
    return x + y;
}

int main()
{
    auto sum = add(5, 6); // add() returns an int, so sum will be type int
    return 0;
}
```