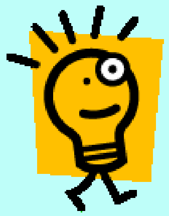
Hash Table

- **In the previous studies, all the searches had an efficiency of at least O(*logn*)**



- **Can it be faster?**
  - For example, if a primary key contains values from 0 to 99, then a table (array) of size 100 would be enough for each record to be directly located by the key value which is the subscript of the table
  - If we can match all key values to different slots of a table, we can make searching for a record very efficient
  - ➔ Hash Table: ideally to support search time O(1)

Hash function and hash key

- **Key values may not be numeric or may be very large, but we may transform the key into a value within a range**
- **E.g., suppose that there are at most m (10000) records in the file. Even if the key is in 8 digits, we may use a function, e.g., *key / 10000* to transform keys with 8 digits to a value from 0-9999**
- **Such a function which transforms a key into a value which may further transform to a subscript of an array, in a fixed length, is called *hash function***
- **The key being transformed is called the *hash of key***

Perfect hash function

- An ideal (*perfect*) hash function transforms all different hash of keys into different subscripts of a table

- When a file has a million records, it is difficult to have such a function

Hash Value and Hash Table index

- **A hash function transforms a key to a value which is called *hash value***
- **This value may need to further be transformed to a subscript of an array: *hashValue%m* where m is the table size**
- **The value which can map to a subscript of an array is called *hash table index***

Hash collision (clash)

- **When two hash of keys have the same hashed values, it is called a *hash collision* or a *hash clash***
- **E.g., given a hash function h(key) = key and the hash table size 1000, ==> hash table size: hi(h(1322)) = 1322 % 1000 = hi(h(2322)) = 2322 % 1000 = 322**
- **That means both key 1322 and 2322 may attempt to insert the record into the same position**

Resolving hash clashes

- There are two basic techniques:

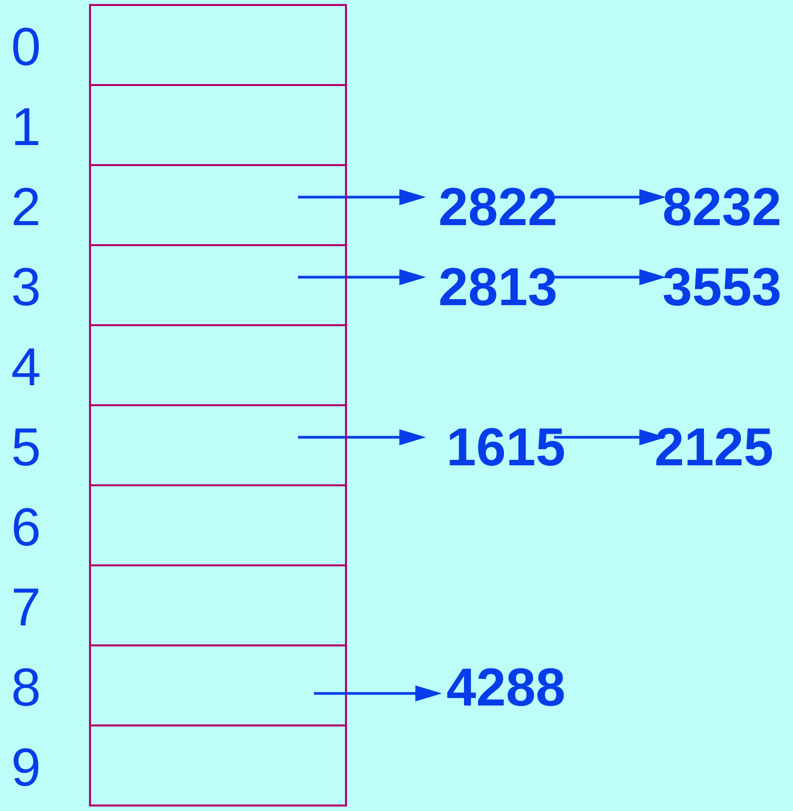
1. *Chaining* (*Open hashing*): Keys with the same hash values will be linked together and a search process should sequentially traverse all the items in the linked list

2. *Open Addressing* (*Closed Hashing*) : Whenever there is a clash, it will rehash – to find another slot in the table
   - many techniques: e.g., linear probing, quadratic probing

Chaining

- Example: h(key) = key % 10

  Input: 2822, 1615, 2813, 3553, 4288, 2125, 8232



Open Addressing: Linear probing

Place the record in the next available position in the array, i.e., *rh(i) = i+1*. E.g., (input: 2822, 1615, 2813, 3553, 4288, 2125, 8232)

0	
1	
2	**2822**
3	**2813**
4	**3553**
5	**1615**
6	**2125**
7	**8232**
8	**4288**
9	

3553: h(3553)=3, rh(1)=4

2125: h(2125)=5, rh(1)=6

8232: h(8232)=2,
rh(1)=3,r(2)=4,
rh(3)=5, rh(4)=6, rh(5)=7

Open addressing -- quadratic rehash

the jth rehash is $h_j(key) = (h(key)+j^2)$ % array_size

⇨ **E.g., (input: 2822, 1615, 2813, 3553, 4288, 2125, 8232)**

0	
1	**8232**
2	**2822**
3	**2813**
4	**3553**
5	**1615**
6	**2125**
7	
8	**4288**
9	

8232: h(8232)=2, h_1=2+1=3,
h_2=2+4=6,
h_3=(2+9)%10=1

3553: h(3553)=3, h_1=3+1=4

2125: h(2125)=5, h_1=5+1=6

Hash table re-sizing

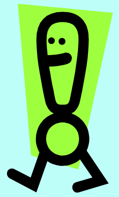
- ⇨ When a hash table is full or nearly full, it requires **re-sizing** ➔to increase the size of the hash table
- ⇨ One of the methods is to take its first prime which is twice as large as the old table size
- ⇨ For the previous table size 10 ➔ new table size is 23 and new hash function is *h(key)=key%23*

					1615		2813		2125	4288	3553					2822					8232	
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22

Load Factor

- To determine if a hash table is full or nearly full, *load factor* is used
- The value of the load factor is the ratio of number of elements (m) to the slots (n) of the table: m/n

Acceptable ranges of load factor



- **For different addressing methods, the load factor has different acceptable ranges**
  - Closed addressing (chaining): about 2 to 4 – if key values are well distributed in the table, it is expected that every linked list has one or more nodes than the load factor, i.e., every hit may require at most 4 to 6 visits
  - Open addressing: less than about 0.7 – it is the percentage of slots being occupied – a larger percentage may make a key to be rehashed many times – no more O(1)

Exercises

- **Ford's 12:15.a-b++**

  hf(x) = x, m=11, data: 1, 13, 12, 53, 77, 29, 31,22

  - a) Construct the hash table by using linear probe addressing
  - Construct the table again by using rehash function:

    index = (index + 5) % 11
  - b) Construct the hash table by using chaining with separate lists; and also
  - Determine the load factors of the tables.
  - Depict the hash table after resize, the one resulting from linear probing.

Hash Functions for integer data

- A hash function usually produces a non-negative value
- A common hash function of numeric data is simply *hash(x) = abs(x)*
- Ford's: hash(x) = x^2 / 256 % 65536

Hash Functions for real numbers

- **Ford's:**
  - hash(x) = 0 if x = 0; otherwise
  - hashval = abs(2 * fabs(frexp(x,&exp)) -1);
    where frexp() is a C library function which is used to decompose *num* into two parts: a mantissa between 0.5 and 1 (returned by the function) and an exponent returned as *exp*; and scientific notation works like this:
    `x = mantissa * (2 ^ exp)`
    (Reference: www.cppreference.com)
- **ICarnegie: *hash(x) = floor(m * (frac(x * r))*, where typically, r can be the Golden Ratio *(sqrt(5) – 1)/2* and *m* is the table size**

Hash functions for strings

- **It is quite easy to think about converting each character to its ASCII value (65-90 and 97-122) and then accumulate its sum as the hash values – all permutations of a word hash to the same slot!**
- **The value of a character at different positions multiplies a factor then sums up the result – making a string similar to a number**
  - when the factor is too small, it may not be significant
  - when the factor is too large, the resulting value would overflow – only the last few characters become accountable!

Two implementations of hf for strings

```
size_t  hashFord (const string& str) {
  size_t prime = 2049982463;
  int n = 0, i;
  for (i = 0; i < str.length(); i++)
     n = n*8 + str[i];
  return n > 0 ? (n % prime) : (-n % prime);
}
```

```
size_t  hashCraniegie (char const* str) {
  size_t  smallPrime = 13; // can be other values
  size_t  h = 0;
  while (*str)
     h = smallPrime * h + *str++;
  return h;
}
```

Hashing for any objects

- When hashing for objects other than numeric or strings, a similar method as hashing for strings can be used

```
size_t  hashFord(Object const& x)   {
char const *p = reinterpret_cast<const string>(x);
  return hashFord(x);
}
```

```
size_t  hashCranegie(Object const& x)   {
char const *p = reinterpret_cast<char const*>(&x);
  return hashCranegie(x);
}
```

Size of a hash table q2 q1

- **If the table size is a power of 2 ➔ easy calculation – however, only the k low order bits of the key values are meaningful**
  - Table size: $b_{sn} \ldots b_{sn-k} 0 \ldots . 0$ (k 0 bits)
  - Any hash key: $b_{hn} \ldots b_{hn-k}\ b_{hk} \ldots b_{h1}$ ➔ hash key % table size = $b_{hk} \ldots b_{h1}$
  - ➔ Any hash key values with the same lower k bits will be hashed into the same table slot!
- **The table size is suggested to be *prime* in order to allow the key value to be evenly distributed**
- **It also depends on the property of the key values**
  - e.g., a key value may always end with a zero
  - e.g., a key value may always be an even number

Hash tables in the STL

- **Not standard yet!**
- **Many come with C++ and support:**
  - hash_set
  - hash_multiset
  - hash_map
  - hash_multimap
  - *multi* : allows duplicate key values
  - *set*: only key values
  - *map*: key and its values ➔ pair<KeyType, ValueType>

Hash Sets and Hash Maps in the STL

- **Hash Sets Vs Hash Maps**
  - Hash sets only store if a certain key exists, no other information is associated; i.e., no data is stored
  - Hash maps allow each entry (slot) of the table to include a pair of information: <key, value>

- ***pair<type_1, type_2>* is a template class in the STL. It allows easy definition of some objects which have two members. E.g., for the hash map:**

```
typedef pair<KeyType, ValueType>  Entry;
Entry   e1;
```

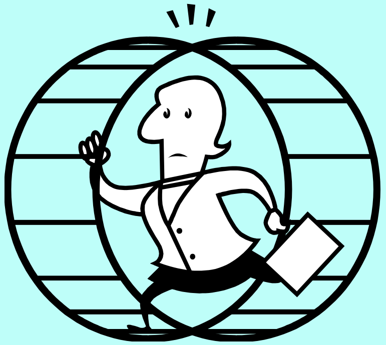
To locate the members, e.g., e1.first is the key, e1.second is the value

STL's *hash_map* in g++ (from RedHat)

- **Header file at <hash_map> or <ext/hash_map>**
- **Template format:**
  - __gnu_cxx::hash_map < KeyType, ValueType, HashFun, equalFun>
  - hash_map is in the namespace __gnu_cxx
  - *KeyType*, V*alueType* are types for the pair<> stored in the map
  - *HashFun* is ***a function object*** which allows a user to specify his/her own *hash function*
  - *equalFun* is ***a function object*** which allows a user to specify how two keys are said to be equal
  - Default hash functions: *hash<T>* where T is a basic type supported by C++,
    - hash<const int>
    - hash<const char *>

Function Object

- A function object is defined as a class in which there is a function to overload the operator *call;* symbol: ()
- E.g., Ford's d_hashf.h
- One of the advantages is a function can preserve the states (different values) of associated variables among different function calls
  - Usually, the variables defined inside a function will be lost after the function has finished



An example of using hash_map

- ⇨ **A sample: demoSTLHash.cpp**
- ⇨ **Another simple:**

```
#include <iostream>
#include <ext/hash_map>
......  // function objects
int main() {
   __gnu_cxx::hash_map<const char*, int,
              HashFun, Eqstr> seasons;
   seasons[“spring"] = 90;
   seasons[“summer"] = 91;
   seasons[“autum"]  = 92;
   seasons[“winter"] = 92;
}
```

Examples of map's required function objects

```
class HashFun {
 public:
   size_t operator()(char const* str)  const {
     size_t  h = 0;
     for (; *str; ++str)  h = 13 * h + *str;
     return  h;
   }
};
class Eqstr {
 public:
   bool operator()(const char* s1,
                   const char* s2) const {
      return strcmp(s1, s2) == 0;
   }
};
```

Hash Table vs BST

- **Timing for searching**
  - Ideally, hash table has the complexity of O(1) while BST has a complexity of O(log n)
  - However, it may require more than O(log n) if many keys are clashed to the same slot. Even with the load factor, a hash table may maintain an optimal time in searching but it takes very much time when the hash table is required to re-size in order to maintain an acceptable load factor
- **Sequential scan and range scan**
  - The in-order traversal on a BST is a sequential scan, and range scan is just a partial scan of the in-order traversal
  - Hash table does not easily support sequential scan on key values unless the hash function maintains the order of the key values – such a hash function may not distribute very well different key values into different slots

Summary

- Hashing is an efficient way to locate a record
- However, an ideal hashing function is difficult to achieve
- In this class, we have studied some techniques to resolve hash clashes: Open hashing (chaining) and closed hashing (linear probing, and quadratic probing)
- When a hash table exceeds its load factor (depending on different hashing methods), the table is required to re-size
- Compared to a binary tree, it lacks of the flexibility in scanning a range of key values

Reference

- **Ford: 12.1-3, 12.5**
- **STL online references**
  - http://www.sgi.com/tech/stl
  - http://www.cppreference.com/
- **Sample programs: demoSTLHash.cpp, Ford: d_hash.h**

-- END --