Dictionaries

• Collection of pairs.
  ▪ (key, element)
  ▪ Pairs have different keys.

• Operations.
  ▪ find(theKey)
  ▪ erase(theKey)
  ▪ insert(theKey, theElement)
Application

- Collection of student records in this class.
  - \((\text{key, element}) = (\text{student name, linear list of assignment and exam scores})\)
  - All keys are distinct.
- Get the element whose key is John Adams.
- Update the element whose key is Diana Ross.
  - \text{insert()} implemented as update when there is already a pair with the given key.
  - \text{erase()} followed by \text{insert()}. 
Dictionary With Duplicates

- Keys are not required to be distinct.
- Word dictionary.
  - Pairs are of the form \((\text{word}, \text{meaning})\).
  - May have two or more entries for the same word.
    - (bolt, a threaded pin)
    - (bolt, a crash of thunder)
    - (bolt, to shoot forth suddenly)
    - (bolt, a gulp)
    - (bolt, a standard roll of cloth)
    - etc.
Represent As A Linear List

- $L = (e_0, e_1, e_2, e_3, ..., e_{n-1})$
- Each $e_i$ is a pair (key, element).
- 5-pair dictionary $D = (a, b, c, d, e)$.
  - $a = (a\text{Key}, a\text{Element}), b = (b\text{Key}, b\text{Element}),$ etc.
- Array or linked representation.
Array Representation

- **find**(theKey)
  - $O$(size) time

- **insert**(theKey, theElement)
  - $O$(size) time to verify duplicate, $O$(1) to add at right end.

- **erase**(theKey)
  - $O$(size) time.
Sorted Array

- elements are in ascending order of key.
- find(theKey)
  - O(log size) time
- insert(theKey, theElement)
  - O(log size) time to verify duplicate, O(size) to add.
- erase(theKey)
  - O(size) time.
Unsorted Chain

firstNode

- **findt(theKey)**
  - $O(\text{size})$ time

- **insert(theKey, theElement)**
  - $O(\text{size})$ time to verify duplicate, $O(1)$ to add at left end.

- **erase(theKey)**
  - $O(\text{size})$ time.
Sorted Chain

- Elements are in ascending order of Key.

- find(theKey)
  - $O(\text{size})$ time

- insert(theKey, theElement)
  - $O(\text{size})$ time to verify duplicate, $O(1)$ to put at proper place.
Sorted Chain

- Elements are in ascending order of Key.

- `erase(theKey)`
  - $\mathcal{O}(\text{size})$ time.
Skip Lists

• Worst-case time for find, insert, and erase is $O(\text{size})$.
• Expected time is $O(\log \text{size})$.
• We’ll skip skip lists.
Hash Tables

- Worst-case time for find, insert, and erase is $O(\text{size})$.
- 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.
  - 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:

<p>| | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>3,d</td>
<td>22,a</td>
<td>33,c</td>
<td></td>
<td>73,e</td>
<td>85,f</td>
</tr>
<tr>
<td>[0]</td>
<td>[1]</td>
<td>[2]</td>
<td>[3]</td>
<td>[4]</td>
<td>[5]</td>
</tr>
</tbody>
</table>

• get, put, and remove take $O(1)$ time.
What Can Go Wrong?

- 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.

- Where does (100, h) go? Not an appropriate hash function for every application.
What Can Go Wrong?

- 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:
  ```cpp
  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

- Most common method is by division.
  \[ \text{homeBucket} = \text{hash(theKey) } \% \text{ divisor}; \]
- \text{divisor} equals number of buckets \( b \).
- \( 0 \leq \text{homeBucket} < \text{divisor} = b \)
Uniform Hash Function

<table>
<thead>
<tr>
<th>(3,d)</th>
<th>(22,a)</th>
<th>(33,c)</th>
<th>(73,e)</th>
<th>(85,f)</th>
</tr>
</thead>
<tbody>
<tr>
<td>[0]</td>
<td>[1]</td>
<td>[2]</td>
<td>[3]</td>
<td>[4]</td>
</tr>
</tbody>
</table>

• 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

<table>
<thead>
<tr>
<th>(3,d)</th>
<th>(22,a)</th>
<th>(33,c)</th>
<th>(73,e)</th>
<th>(85,f)</th>
</tr>
</thead>
<tbody>
<tr>
<td>[0]</td>
<td>[1]</td>
<td>[2]</td>
<td>[3]</td>
<td>[4]</td>
</tr>
</tbody>
</table>

- Equivalently, the probability that a randomly selected key has bucket $i$ as its home bucket is $1/b$, $0 \leq i < b$.
- 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

• 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

- 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.
STL hash_map

- Simply uses a divisor that is an odd number.
- This simplifies implementation because we must be able to resize the hash table as more pairs are put into the dictionary.
  - Array doubling, for example, requires you to go from a 1D array table whose length is \( b \) (which is odd) to an array whose length is \( 2b+1 \) (which is also odd).
unordered_map

1. template < class Key,
2.   class T,
3.   class Hash = hash<Key>,
4.   class Pred = equal_to<Key>,
5.   class Alloc = allocator< pair<const Key,T> >
6.   > class unordered_map;

1. std::unordered_map<std::string,double> myrecipe =
   {{"milk",2.0},{"flour",1.5}};
2. myrecipe.insert
   (std::make_pair<std::string,double>("eggs",6.0))

1. std::unordered_map<std::string,double>::const_iterator
   got = myrecipe.find(“eggs”);
2.
3. if ( got == mymap.end() )
4.    std::cout << "not found";
5. else
6.    std::cout << got->first << " is " << got->second;
Implementation using sorted chains

1. void insert(const pair<const K, E>& thePair) {
2.     int homeBucket = (int) hash(thePair.first) % divisor;
3.     int homeSize = table[homeBucket].size();
4.     table[homeBucket].insert(thePair);
5.     if (table[homeBucket].size() > homeSize) dSize++;
6. }

1. pair<const K, E>* find(const K& theKey) const {
2.     return table[hash(theKey) % divisor].find(theKey);
3. }
Application: Histograms

• For counting number of occurrences of an item (or characteristic) in a list.

• Key of mapping is characteristic.

• Value of mapping is count.
1. class tree {
2.     public:
3.         tree(int height) { this->height = height; }
4.         int getHeight() { return height; }
5.     private:
6.         int height;
7. }; 
8. 
9. int main() {
10.    int numTrees = 1000, maxHeight = 50;
11.    tree *forest[numTrees];
12.    for (int i = 0; i < numTrees; i++) {
13.        forest[i] = new tree(rand() % maxHeight);
14.    }
15.    unordered_map<int, int> treeCount;
for (int i = 0; i < numTrees; i++) {
    int count = treeCount[forest[i]->getHeights()];

    if (count) {
        treeCount[forest[i]->getHeights()] = count + 1;
    } else {
        treeCount[forest[i]->getHeights()] = 1;
    }
}

for (int i = 0; i < maxHeight; i++) {
    cout << i << "m " << treeCount[i] << endl;
}
return 0;
Application: i18n

- If your application is going to be used internationally, you may want to provide translations for all of your text strings.

- Instead of hardcoding strings in your user interface, use lookups to property files.
Properties Files

*Messages_en_US.properties*

- yesMessage=Yes
- noMessage=No

*Messages_en_FR.properties*

- yesMessage=Oui
- noMessage=Non

1. `Locale locale = new Locale(language, country);`
2. `ResourceBundle captions= ResourceBundle.getBundle("Messages", locale);`
3. `yesCaption =captions.getString("yesMessage");`
Application: Hosts File

• Sometimes you want to override the IP address furnished by the DNS servers for a given domain name.

• Hosts file provides dictionary of IP/domain name mappings.

• In Windows 7: C:\windows\System32\drivers\etc\hosts