# Counting Sort, HashTable
# CISC4080
# CIS, Fordham Univ.

Instructor: X. Zhang

# Recap

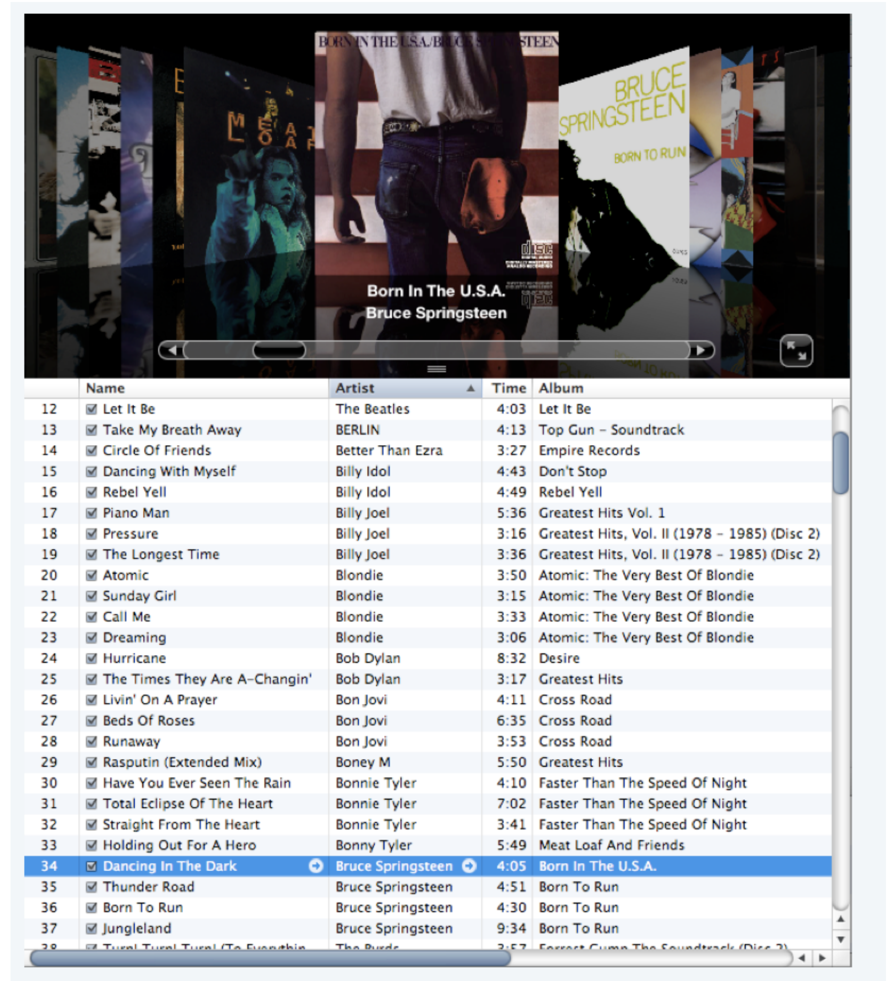
- What's algorithms, problem, problem instance
- Selection, bubble, insertion sort
- Recursive algorithm:
  - three questions for thinking recursively!
- 2 (3) questions for each algorithm:
  - Is it correct?
  - Is it efficient?
  - Can we do better (use less resource: faster, use less memory)?
- Measure running time: how long does it take?
  - Intuitively: it depends on size of problem instance, and problem instance itself!

# This class

- Counting Sort, and Radix Sort
- Pass function as parameter
- Dictionary
  - Direct Access Table
  - Hashtable
  - C++ STL unordered_map

# Sorting is everywhere!

- When **exploring your directory** (using finder in Mac), you can sort entries by name, type, last modified time…, in ascending order or descending order
  - In Excel, iTune, …
- More generally, we can provide a comparison function to decide “order” of two elements



# Passing Comparison Function

- http://storm.cis.fordham.edu/~zhang/cs4080/Demo/BubbleSortRecord.cpp

```
#include <functional>
…

//Sort vector of record, using InOrder function to compare and see if two
records are in order
void bubblesort (vector<Record> &  a,
 std::function<bool (Record,Record)> InOrder)
 //^^ InOrder is a function that takes two Record type parameters and return a
bool.
{

     int end_index = a.size()-1;

     for (; end_index>0; end_index--)
     {
           for (int i=0; i<=end_index-1;i++)
                 //if a[i],  a[i+1]) are in wrong order, originally:
                // if (a[i]>a[i+1]), if (!(a[i]<=a[i+1]))
                if (!InOrder(a[i],a[i+1]))
                     swap (a[i], a[i+1]);
     }
}
```

# Passing Comparison Function

```
class Record{
private:
   string name;
   int b_year;
   int b_month;
   int b_day;

public:
    …
//Several examples of InOrder functions for different keys and orders
   friend bool InAscendingOrderByName (Record a, Record b);

   friend bool InAscendingOrderByBDate (Record a, Record b);

};

//return true if a's birthday is before or the same as b's
bool InAscendingOrderByBDate (Record a, Record b)
{
     if (a.b_year < b.b_year ||
        a.b_year == b.b_year && a.b_month < b.b_month ||
        a.b_year == b.b_year && a.b_month == b.b_month &&  a.b_day <= b.b_day)
             return true;
     else
             return false;
}
```

# Passing Comparison Function

```
int main()
{
        vector<Record> d;
        d.push_back (Record("Alice",2000, 3, 4));
        d.push_back (Record("Jack",2001, 4, 8));
        d.push_back (Record("Abe",1999,8,21));
        d.push_back (Record("Alice", 2001, 9, 19));
        d.push_back (Record("Bob", 2001,9,3));
        d.push_back (Record("Abe",2001,4,8));
        d.push_back (Record("Jack", 2001,9,19));


        cout <<"Before sorting\n";
        for (int i=0;i<d.size();i++)
                d[i].Print();


        /* sort d by ascending order of birthdate, i.e. from oldest to
youngest  */
        bubblesort (d, InAscendingOrderByBDate);

        cout <<"After sorting in ascending order by birthdate\n";
        for (int i=0;i<d.size();i++)
                d[i].Print();
}
```

# When there are duplicate keys

| Name | Age |
|---:|---:|
| Alice | 5 |
| Jack | 2 |
| Bob | 4 |
| Chris | 5 |
| Lisa | 4 |
| Jasmine | 3 |
| Bob | 2 |

sort by
ascending order
of field Age

| Name | Age |
|---:|---:|
| Jack | 2 |
| Bob | 2 |
| Jasmine | 3 |
| Bob | 4 |
| Lisa | 4 |
| Alice | 5 |
| Chris | 5 |

sort by
ascending order
of field Age

| Name | Age |
|---:|---:|
| Jack | 2 |
| Bob | 2 |
| Jasmine | 3 |
| Lisa | 4 |
| Bob | 4 |
| Alice | 5 |
| Chris | 5 |

# Stableness

| Name | Age |
|---|---|
| Alice | 5 |
| Jack | 2 |
| Bob | 4 |
| Chris | 5 |
| Lisa | 4 |
| Bob | 2 |
| Jasmine | 3 |
| Ed | 2 |

sort by
ascending order
of field Age

| Name | Age |
|---|---|
| Jack | 2 |
| Bob | 2 |
| Ed | 2 |
| Jasmine | 3 |
| Bob | 4 |
| Lisa | 4 |
| Alice | 5 |
| Chris | 5 |

Def:

A sorting algorithm is stable: if it **always** maintains relative order of elements with same key value

(Jack, 2) , (Bob, 2), (Ed, 2)
(Alice, 5) and (Chris, 5)
(Bob, 4), (Lisa, 4)

# Unstable Sorting Algorithm

| Name | Age |
|---|---|
| Alice | 5 |
| Jack | 2 |
| Bob | 4 |
| Chris | 5 |
| Lisa | 4 |
| Jasmine | 3 |
| Bob | 2 |

sort by
ascending ord
of field Age

| Name | Age |
|---|---|
| Jack | 2 |
| Bob | 2 |
| Jasmine | 3 |
| Lisa | 4 |
| Bob | 4 |
| Alice | 5 |
| Chris | 5 |

Relative positions of
(Bob,4) and (Lisa 4) changed!

So the sorting algorithm (used to generate the output) is not stable!

Def:

A sorting algorithm is not stable: if it **does not** maintains relative order of some elements with same key value

# This class

- Counting Sort, and Radix Sort
- Pass function as parameter
- Dictionary
  - Direct Access Table
  - Hashtable
  - C++ STL unordered_map

# Counting Sort

- **Input**: a list of integers or more generally, elements with an integer-valued key, with known value range
- **Output**: rearranged list by ascending (or descending order) of an integer valued key
- e.g., L[0…7]={5, 2, 4, 6, 4, 3, 2}, Min=1, Max=10
- Or a list of children at a Kindergarten:
  - (Alice, 5), (Jack 2), (Bob, 4), (Chris,5), (Lisa, 4), (Jasmine, 3), (Bob, 2)
  - sort by ascending order of age, all between 1 and 6 =>

# Counting Sort

- **Input**: a list of integers or elements with an integer-valued key, with known value range
- **Output**: rearranged list by ascending (or descending order) of an integer valued key
- e.g., L[0…7]={5, 2, 4, 6, 4, 3, 2}, Min=1, Max=10
- Or a list of children at a Kindergarten:
  - (Alice, 5), (Jack 2), (Bob, 4), (Chris,5), (Lisa, 4), (Jasmine, 3), (Bob, 2)
  - sort by ascending order of age, all between 1 and 6 =>
  - (Jack, 2), (Bob, 2), (Jasmine, 3), (Bob, 4), (Lisa, 4), (Alice, 5), (Chris, 5)

# Counting Sort

/* **Input**: a list of integers or elements with an integer-valued key, with known value range

**Output**: rearranged list by ascending (or descending order) of an integer valued key */

- e.g., L[0…7]={5, 2, 4, 6, 4, 3, 2}, Min=1, Max=10
- Ideas:
  - Count frequency of each integer values in L, how many 1's? How many 2's? … How many 10's?
    - how?
  - Use this to decide where each list element should be stored in sorted list

# Counting Sort

/* **Input**: a list of integers or elements with an integer-valued key, with known value range

**Output**: rearranged list by ascending (or descending order) of an integer valued key */

- e.g., L[0…7]={5, 2, 4, 6, 4, 3, 2}, Min=1, Max=10
- Ideas:
  - Count frequency of each integer values in L, how many 1's? How many 2's? … How many 10's?

| Value | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|---|
| Count | 0 | 2 | 1 | 2 | 1 | 1 | 0 | 0 | 0 | 0 |

  -

# Counting Sort

/* **Input**: a list of integers or elements with an integer-valued key, with known value range

**Output**: rearranged list by ascending (or descending order) of an integer valued key */

- e.g., L[0…8]={5, 2, 1, 4, 6, 4, 3, 2}, Min=1, Max=10
- Ideas:
  - 2. How many elements are <=1, 2, 3, 4, …?

| Value | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|---|
| Count | 1 | 2 | 1 | 2 | 1 | 1 | 0 | 0 | 0 | 0 |
| Cumulative | 1 | 3 | 4 | 6 | 7 | 8 | 8 | 8 | 8 | 8 |

  - There are 6 elements that are <=4

# Counting Sort

3. Where each list element should be stored?

e.g., the two 4's?

Input:

| index | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|---|
| L | 5 | 2 | 1 | 4 | 6 | 4 | 3 | 2 |

Look up count and cumulative using 4 as index:

| Value | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|---|
| Count | 1 | 2 | 1 | 2 | 1 | 1 | 0 | 0 | 0 | 0 |
| Cumulative | 1 | 3 | 4 | 6 | 7 | 8 | 8 | 8 | 8 | 8 |

- Value 4 appears twice in the list,
- There are 6 elements that are <=4 => L[0…5]

Output:

| index | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|---|---|
| L | | | | | | | | | |

# Counting Sort

3. Where each list element should be stored? e.g., the two 4's?

Input:

| index | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|---|
| L | 5 | 2 | 1 | 4 | 6 | 4 | 3 | 2 |

Look up count and cumulative using 4 as index:

| Value | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|---|
| Count | 1 | 2 | 1 | 2 | 1 | 1 | 0 | 0 | 0 | 0 |
| Cumulative | 1 | 3 | 4 | 6 | 7 |  | 8 | 8 | 8 | 8 |

Output:

| index | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|---|---|
| L |  |  |  |  | 4 | 4 |  |  |  |

# CountingSort (L[0…n-1], k)

```
{
    Let count[0…k] be a new array
    for i=0 to k
        count [i] = 0
   // count[v] stores # of occurrences of value v


   // make count[v] stores # of values <=v


   // place list elements to R[0…n-1], in sorted order (making use of count array)


   //Copy R to L


}
```

# CountingSort (L[0…n-1], k)

```
{
    Let count[0…k] be a new array
    for i=0 to k
        count [i] = 0
   // count[v] stores # of occurrences of value v
   for j=0 to n-1
        count[ L[j] ] ++     //we see one more value of L[j], so increment the counter value

   // make count[v] stores # of values <=v
   for i=1 to k
        count[i] = count[i-1]+1

   // place list elements to R[0…n-1], in sorted order (making use of count array)
   for j=n-1 downto 0
        p = count[ L[j] ]
        count[ L[j] ] = count[L[j]]-1 //next occurrence of value L[j] should go here… (to the left)
        R[p] = L[j]

   //Copy R to L
    for I= 0 to n-1
        L[j] = R[j]

}
```

# Counting Sort

Input:

| index | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|---|
| A | 5 | 2 | 1 | 4 | 6 | 4 | 3 | 2 |

Count occurrences

•

| Value | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|---|
| C | 1 | 2 | 1 | 2 | 1 | 1 | 0 | 0 | 0 | 0 |

Count <=

| Value | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|---|
| C | 1 | 3 | 4 | 6 | 7 | 8 | 8 | 8 | 8 | 8 |

Output:

| index | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|---|
| A | | | | | | | | |

# CountingSort (L[0…n-1], k)

```
{
 Let count[0…k] be a new array
 for i=0 to k
      count [i] = 0
// count[v] stores # of occurrences of value v
for j=0 to n-1
       count[ L[j] ] ++     //we see one more value of L[j], so increment the counter value

// make count[v] stores # of values <=v
for i=1 to k
       count[i] = count[i-1]+1

Create a separate vector R of same size as L
// place list elements to R[0…n-1], in sorted order (making use of count array)
for j=n-1 downto 0
       p = count[ L[j] ] //L[J] is p-th smallest number in L
       R[p-1] = L[j]  //L[j] should go to index p-1
       count[ L[j] ] = count[L[j]]-1 //next occurrence of value L[j] should go here… (to the left)
       //The above three lines can be combined into one:
       // R[-- count[L[j]] ] = L[j]

//Copy R to L, one element at a time
// You can also use vector assignment: L = R
 for j= 0 to n-1
      L[j] = R[j]
```

Which sorting algorithm to use,
Selection sort or counting sort?

* to sort a list of 100 employees by their SSN

# Radix Sorting

- Radix Sort: sort digit by digit
  - from least significant digit to most significant digit, uses (stable) counting sort to sort by each digit

| | | | | | | |
|---|---|---|---|---|---|---|
| 329 | | 720 | | 720 | | 329 |
| 457 | | 355 | | 329 | | 355 |
| 657 | | 436 | | 436 | | 436 |
| 839 | → | 457 | → | 839 | → | 457 |
| 436 | | 657 | | 355 | | 657 |
| 720 | | 329 | | 457 | | 720 |
| 355 | | 839 | | 657 | | 839 |

# Radix Sorting

| | | | |
|---|---|---|---|
| 329 | 720 | 720 | 329 |
| 457 | 355 | 329 | 355 |
| 657 | 436 | 436 | 436 |
| 839 | 457 | 839 | 457 |
| 436 | 657 | 355 | 657 |
| 720 | 329 | 457 | 720 |
| 355 | 839 | 657 | 839 |

- Why does it work?
  - How to reason about it…

# Dictionary

- Dictionary: a data structure that stores a dynamic set of elements, supporting INSERT, DELETE, SEARCH operations
  - elements have distinct key fields, or each element is a (key, value) pair
- Usage example:
  - graph traversal algorithms (BFS and DFS) store node's **color, distance, predecessor, …**
    - if color[v]==WHITE //look up color of node v

  color[ ] is a dictionary, node is key, value field is Color
  - Word count program
  - Clouting sort: array count[ ]

# Simplest implementation

- Direct address table: array + <u>use key as index</u> into the array
  - only applicable when key is integer type
  - If T is the array, then T[i] stores the element whose key is i
  - Ex: in counting sort, we use array C to keep track occurrence of different int values
    - c[i] stores the number of time value i appear in array a[]
- Limitation:
  - key has to be integer type
  - table/array needs to be big enough to have one slot for every possible key

# BST Implementation

- If key type is ordered (i.e., one can compare two given keys, k1, k2), one can use binary search tree
  - each node stores a key, value pair
  - pairs with smaller keys => stored in left subtree
  - pairs with larger keys => stored in right subtree
  - insert O(log n), delete O(log n), search O(log n)
- In C++ STL, ordered_map implements dictionary using BST
  - #include <ordered_map>
  - // wordsCnt is a dictionary/map, key is string, value is int type
  - ordered_map<string, int> wordsCnt;
    - //stores occurrence for each word
  - string word;
  - inputFile>>word;
  - wordsCnt[word]++; //increment occurrence by 1

# Hash table Implementation

- If key type is not ordered (i.e., one cannot compare two given keys, k1, k2,), one can use hash table
  - insert, delete, search: almost constant time operation

- unordered_map in C++ STL
  - #include <unordered_map> wordsCnt;
  - unordered_map<string, int> wordsCnt; //stores occurrence for each word
  - string word;
  - inputFile>>word;
  - wordsCnt[word]++; //increment occurrence by 1

## An example (given in Graph lab)
## Accessing unordered_map

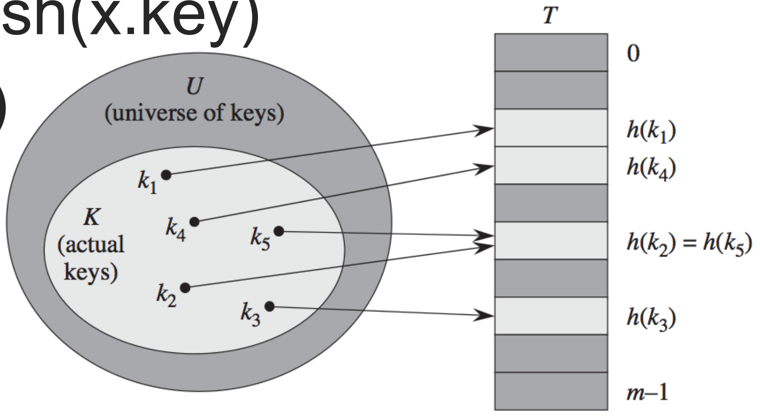
```
#include <unordered_map>

#include <fstream>


int main()
{
     unordered_map<string,int> wordsCount;
     char filename[256];
     ifstream input; //declare an ifstream object, which represents a disk file from which
       //we will read info.
     string word;

     cout <<"Enter the file you want to analyze:";
     cin >> filename;

     //Open the disk file
     input.open (filename);

     if (input.is_open())
     {
          //reading from the file is similar to reading from standard input (cin)
          while (input >> word){ //as long as we successfully read a word
               wordsCount[word]++;  //Increment the count for the word

               //when a word is encounted for the first time, wordsCount[word] is
               //accessed for the first time, the value will be initialized to 0 automatically
          } //continue until we reached the end of file

          //Close the file
          input.close();

     } else
     {
          cout <<"Failed to open file " << filename<<endl;
          exit(1);
     }
```

```
//Search a unordered_map
char cont;
do{
    cout <<"Enter a word:";
    cin >> word;
    map<string,int>::iterator it;

    it = wordsCount.find(word);
    if (it==wordsCount.end())
    {
        cout <<" does not appear\n";
        //if accessed (as below), it will be initialized to
        // default value, for int, it's 0
        cout <<"if accessed?"<<wordsCount[word]<<endl;
    }
    else
        cout <<" appears "<<wordsCount[word]<<" times\n";

    cout <<"Continue (y/n)?";
    cin >> cont;
} while (cont=='y');
```

```
    //iterate through a map
    cout <<"Display the words and count\n";
    map<string,int>::iterator it;
    cout <<"word   count\n";
    for (it=wordsCount.begin();it!=wordsCount.end();it++)
    {
        cout <<it->first<<" "<<it->second<<endl;
    }
}
```

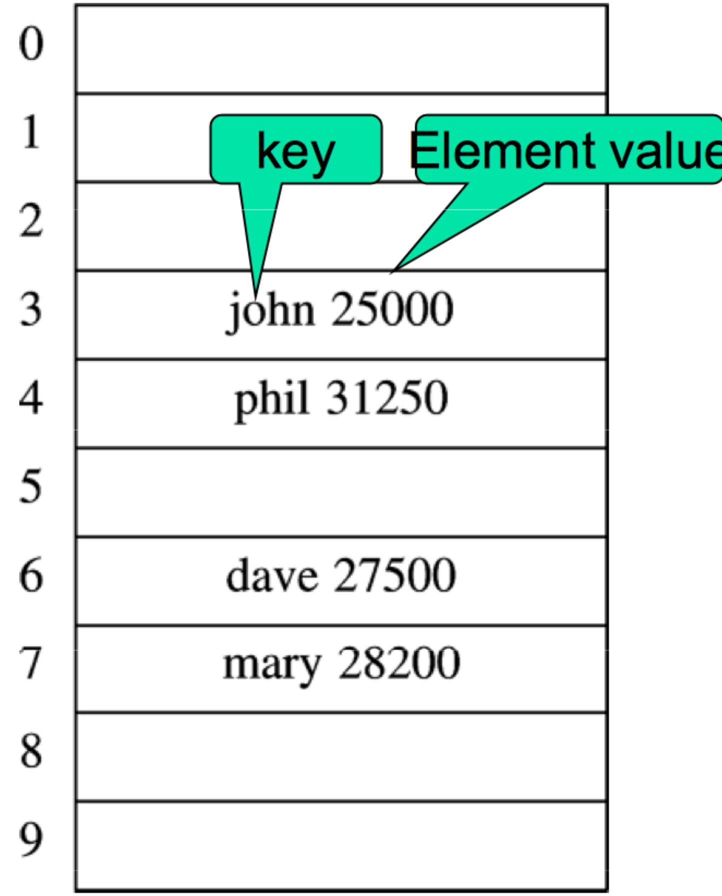
# Hash Table

- Ideas:
  - use a table, T, of certain size, say m, to store a collection of (key, value) pairs
  - use a hash function to map key to index of the table (array)
    - int hash (Key k) // return value 0…m-1
- Given an element x (with key k, value v)
  - store element at index hash(x.key)
  - i.e., T[ hash (k) ] = x (k, v)



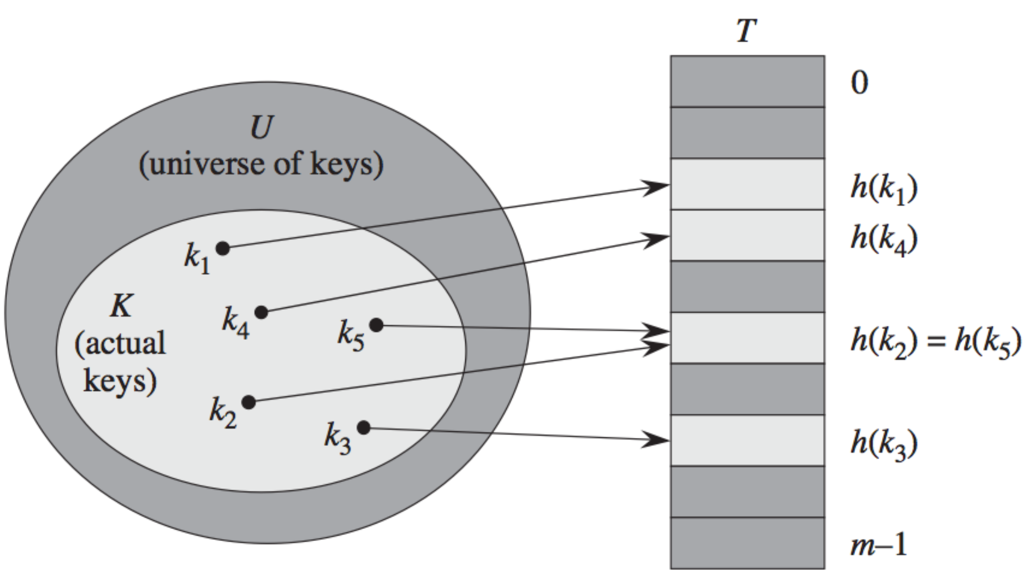
# HashTable Operations

- **Insert a new key value pair:**
  - Table[h(“john”)]=Element(“John”, 25000)
- **Delete element by key**
  - Table[h(“john”)]=NULL
- Search by key
  - return Table[h(“dave”)]
- Assuming running time of h() is constant, all above operations takes constant time



# Hashing: unavoidable collision

- Table T of size m, $m = \Theta(|K|)$
- A **hash function**: $h : U \rightarrow \{0, ..., m-1\}$
- Given |U| > |m|, hash function is many-to-one function, by pigeonhole theorem
  - Collisions cannot be avoided



# Hash Function

- Good hash function:
  - fast to compute
  - Ideally, map any key equally likely to any of the slots, independent of other keys
- Hash Function:
  - first stage: map non-integer key to integer
  - second stage: map integer to [0…m-1], where m is size of hash table

# First stage: key => integer

- Any basic type is represented in binary
- Composite type which is made up of basic type
  - a character string (each char is coded as an int by ASCII code), e.g.,“pt”
    - add all chars up, ‘p’+’t’=112+116=228
    - radix notation: ‘p’*128+’t’=14452
      – treat “pt” as base 128 number…
  - a point type: (x,y) an ordered pair of int
    - x+y
    - ax+by // pick some non-zero constants a, b, …
  - IP address:four integers in range of 0…255
    - add them up
    - radix notation: $150*256^3+108*256^2+68*256+26$

# Hash Function: integer =>[0, m-1]

- **Division method**: divide integer by m (size of hash table) and take remainder
  - h(key) = key mod m
- if key's value are randomly uniformly distributed all integer values, above hash function is uniform
- But often times data are not randomly distributed,
  - What if m=100, all keys have same last two digits?
  - Similarly, if $m=2^p$, then result is simply lowest-ordred p bits
- Rule of thumbs: choose m to be a prime not too close to exact powers of 2

# Hash Function: second stage

- **Multiplication method**: pick a constant A in the range of (0,1),

$$h(k) = \lfloor m(kA - \lfloor kA \rfloor) \rfloor$$

  - take fraction part of kA, and multiply with m
- e.g., m=10000, A=
  - h(123456)=41.
- Advantage: m could be exact power of 2…

# Multiplication Method

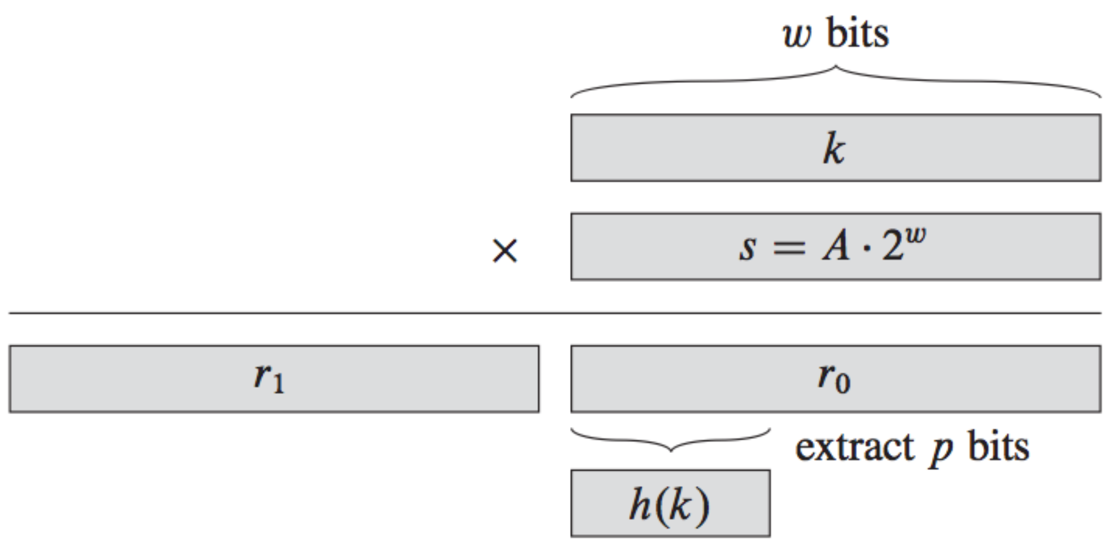



**Figure 11.4** The multiplication method of hashing. The $w$-bit representation of the key $k$ is multiplied by the $w$-bit value $s = A \cdot 2^w$. The $p$ highest-order bits of the lower $w$-bit half of the product form the desired hash value $h(k)$.

$$m = 2^p$$

$$h(k) = \lfloor m(kA - \lfloor kA \rfloor) \rfloor$$

# Exercise

- Write a hash function that maps string type to a hash table of size 250
  - First stage: using radix notation
    - "Hello!" => $\underbrace{\text{'H'*128^5+'e'*128^4+...+'!'}}_{x}$
  - Second stage:
    - x mod 250

# Exercise

- Write a hash function that maps a point type as below to a hash table of size 100

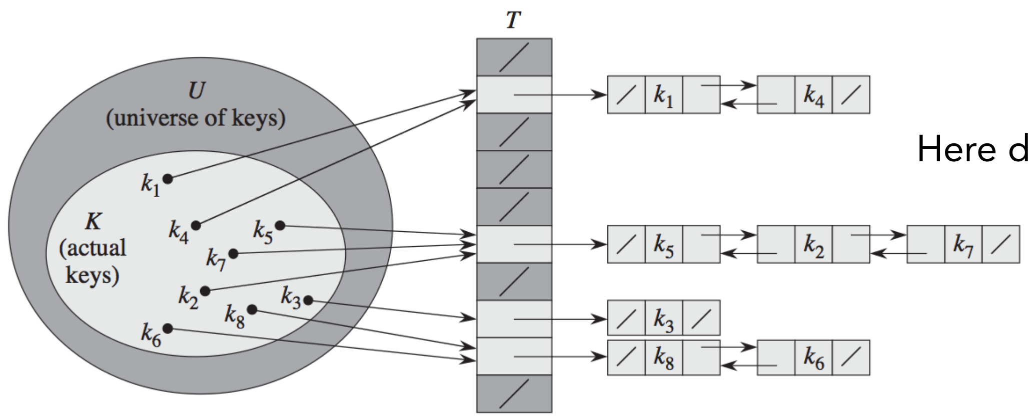
```
class point{
    int x, y;
}
```

# Collision Resolution

- Recall that h(.) is not one-to-one, so it maps multiple keys to same slot:
  - for distinct k1, k2, h(k1)=h(k2) => collision
- Two different ways to resolve collision
  - Chaining: store colliding keys in a linked list (bucket) at the hash table slot
    - dynamic memory allocation, storing pointers (overhead)
  - Open addressing: if slot is taken, try another, and another (a probing sequence)
    - clustering problem.

# Chaining

- Chaining: store colliding elements in a linked list at the same hash table slot
  - if all keys are hashed to same slot, hash table degenerates to a linked list.




Here doubly-linked list is used

**Figure 11.3** Collision resolution by chaining. Each hash-table slot $T[j]$ contains a linked list of all the keys whose hash value is $j$. For example, $h(k_1) = h(k_4)$ and $h(k_5) = h(k_7) = h(k_2)$. The linked list can be either singly or doubly linked; we show it as doubly linked because deletion is faster that way.

# Chaining: operations

- Insert (x):
  - insert x at the head of T[h(x.key)]
- Search (k)
  - search for an element with key x in list T[h(k)]
- Delete (x)
  - Delete x from the list T[h(x.key)]
- Running time of search and delete: proportional to length of list stored in h(x.key)

# Chaining: analysis

- Consider a hash table T with m slots stores n elements.
  - load factor
- Ideal case: any given element is equally likely to hash into any of the m slots, independently of where any other element is hashed to
  - average length of lists is
  - search and delete takes
- Worst case: If all keys are hashed to same slot, hash table degenerates to a linked list
  - search and delete takes

# Collision Resolution

- Open addressing: store colliding elements elsewhere in the table
  - Advantage: no need for dynamic allocation, no need to store pointers
- When inserting:
  - examine (probe) a sequence of positions in hash table until find empty slot
- When searching/deleting:
  - examine (probe) a sequence of positions in hash table until find element

# Open Addressing

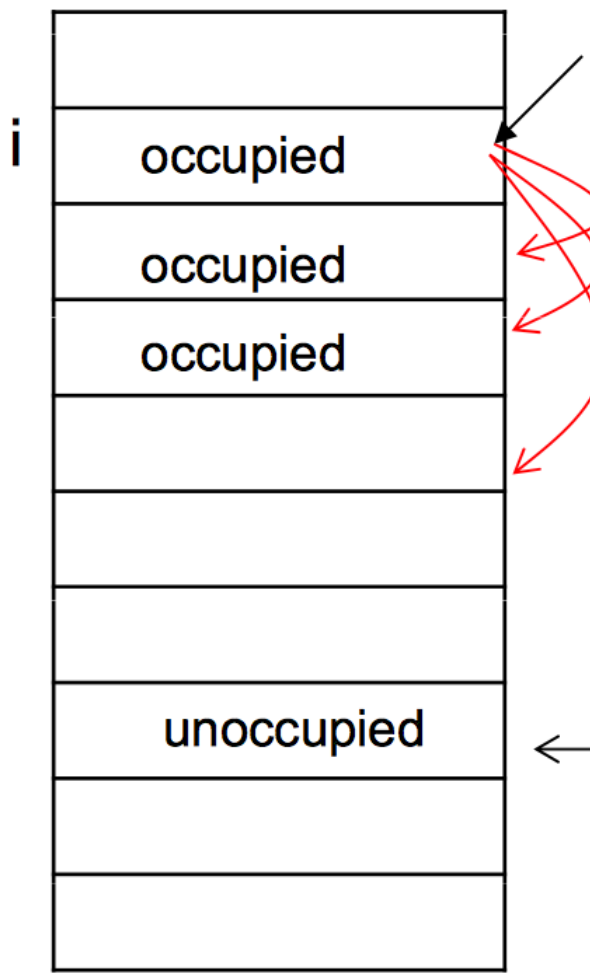
- Hash function: extended to probe sequence (m functions):

$$h_i(x), i = 0, 1, ..., m - 1$$

$$h_i(x) \neq h_j(x), \text{ for } i \neq j$$

- **insert**: if $h_0(k)$ is taken, try $h_1(k)$, and then $h_2(k)$, until find an empty slot
- **Search** for key k: if element at $h_0(k)$ is not a match, try $h_1(k)$, and then $h_2(k)$, ..until find matching element, or reach an empty slot
- **Delete** key k: first search for k, then mark its slot as DELETED

# Linear Probing




- Probing sequence

  $h_i(x)=(h(x)+i)$ mod m

  - try following indices in sequence
    - h(x) mod m,
    - (h(x)+1) mod m,
    - (h(x)+2) mod m, …
  - Continue until an empty slot is found
- Problem: primary clustering
  - if there are multiple keys mapped to a slot, the slots after it tends to be occupied

# Quadratic Probing

$$h_i(x) = (h(x) + c_1 i + c_2 i^2) \bmod m$$

- probe sequence:
  - $h_0(x)=h(x) \bmod m$
  - $h_1(x)=(h(x)+c_1+c_2) \bmod m$
  - $h_2(x)=(h(x)+2c_1+4c_2) \bmod m$
  - …
- Problem:
  - secondary clustering
  - choose $c_1$,$c_2$,m carefully so that all slots are probed

# Double Hashing

- Use two functions $f_1$,$f_2$:

$$h_i(x) = (f_1(x) + i \cdot f_2(x)) \bmod m$$

- Probe sequence:
  - $h_0(x)=f_1(x) \bmod m$,
  - $h_1(x)=(f_1(x)+f_2(x)) \bmod m$
  - $h_2(x)=(f_1(x)+2f_2(x)) \bmod m$,…
- $f_2(x)$ and m must be relatively prime for entire hash table to be searched/used
  - Two integers a, b are <u>relatively prime</u> with each other if their greatest common divisor is 1
  - e.g., $m=2^k$, $f_2(x)$ be odd
  - or, m be prime, $f_2(x)<m$

# Summary

- We so far have reviewed/studied:
  - Three basic sorting algorithms (comparison based) , counting sort, radix sorting
  - Recursive algorithms: three questions rule
  - Data structure review: dictionary, direct access table, hash table, using them in C++ STL
  - Passing function as parameter to sorting function
  - Stable sorting vs unstable sorting
- Lab2:
  - A group project with various small practices
- Next week: algorithm (running time) analysis