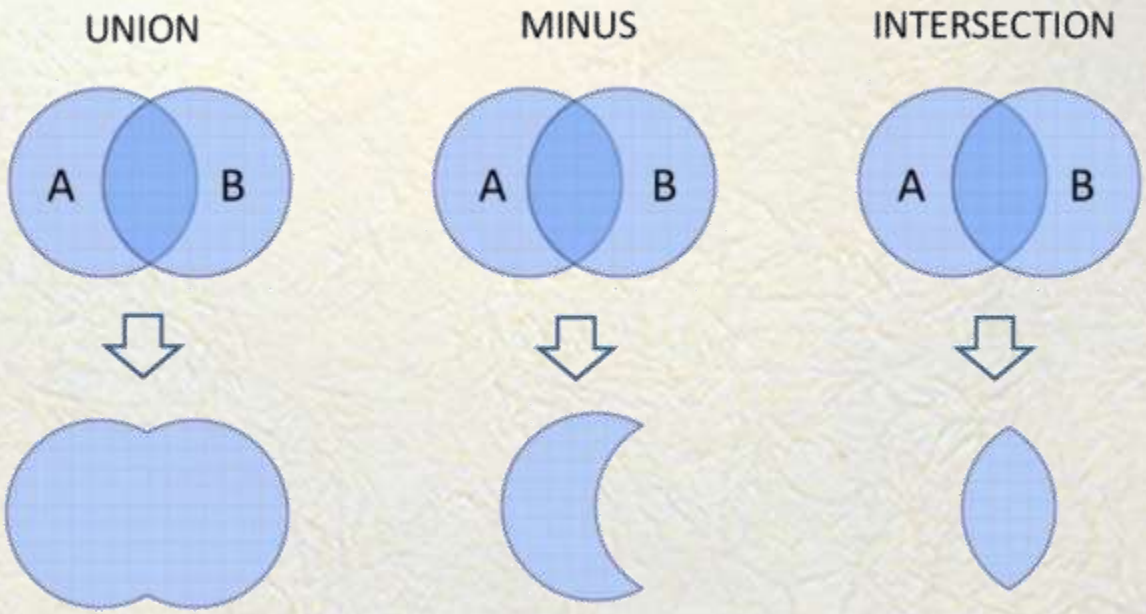
Set, Map and Hash table

University of Technology and Engineering
Vietnam National University Hanoi

Set

- A set is a collection of elements which are not in any particular order
- All elements of a set are different

Set operations



Set operations Union

- **Definition**: Let A and B be sets, the union of two sets A and B is the set that contains all elements in A, B, or both.

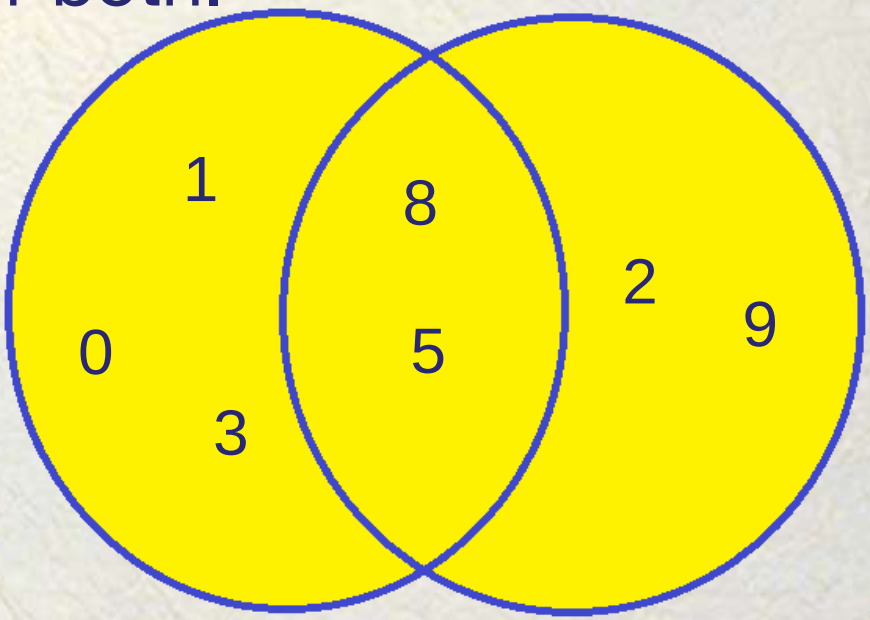
- Example:

  A={0,1,3, 5,8}

  B= {2,5,8,9}

  A∪B = {0,1,2,3,5,8,9}

  Note: A∪B = B∪A



Set operations Intersection

- **Definition**: Let A and B be sets, the intersection of two sets A and B is the set of elements that are in both A and B.

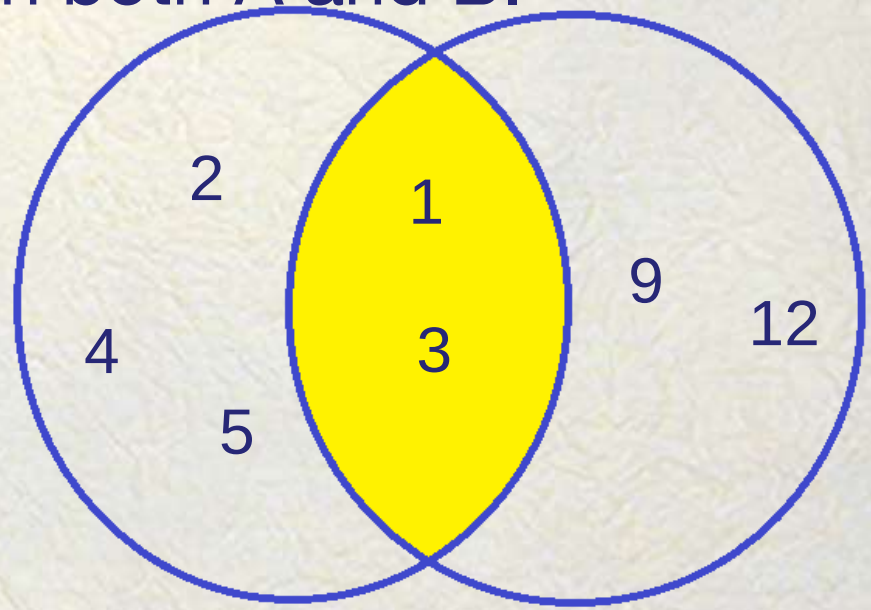
- Example:

  A={1,2, 3,4,5}

  B= {1,3,9,12}

  A ∩B = {1,3}

  Note: A∩B = B∩A



Set operations Minus

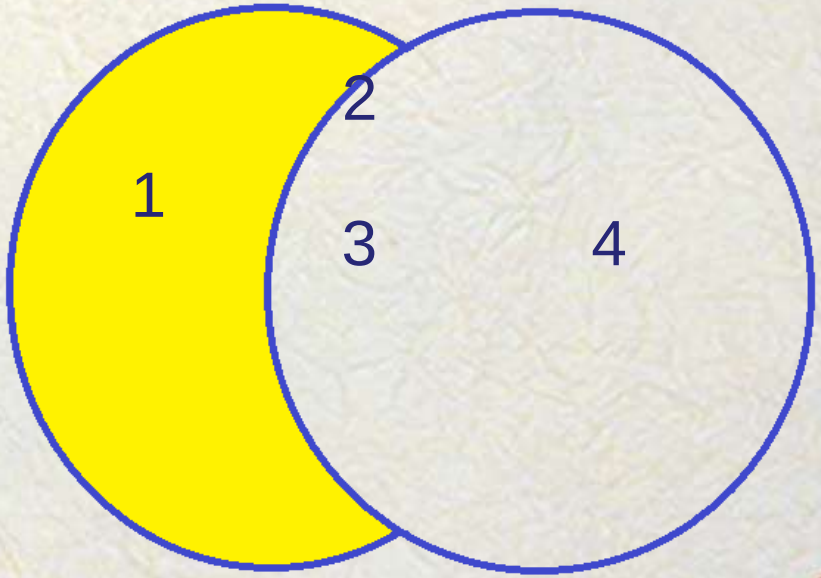
- **Definition**: Let A and B be sets, the difference of A minus B (A – B) is the set of elements that are in A, but not in B.

- Example:

  A={1,2,3}

  B= {2,3,4}

  A - B = {1}



Using set library

Unordered sets are containers that store unique elements in no particular order

```
// unordered_set::insert
#include <iostream>
#include <string>
#include <array>
#include <unordered_set>
int main () {
      std::unordered_set<std::string> myset = {"yellow","green","blue"};
      std::array<std::string,2> myarray = {"black","white"};
      std::string mystring = "red";

      myset.insert (mystring); // copy insertion
      myset.insert (myarray.begin(), myarray.end()); // range insertion
      myset.insert ( {"purple","orange"} ); // initializer list insertion

      std::cout << "myset contains:";
      for (const std::string& x: myset) std::cout << " " << x;
      std::cout << std::endl;

      return 0;
}
```

Example: Insert elements to a set

Maps

- A map models a searchable collection of key-value entries
- Multiple entries with the same key are **not** allowed

Key	Value
0000001	Le Sy Vinh
0000002	Nguyen Van An
0000003	Tran Quoc Hung

The map operations

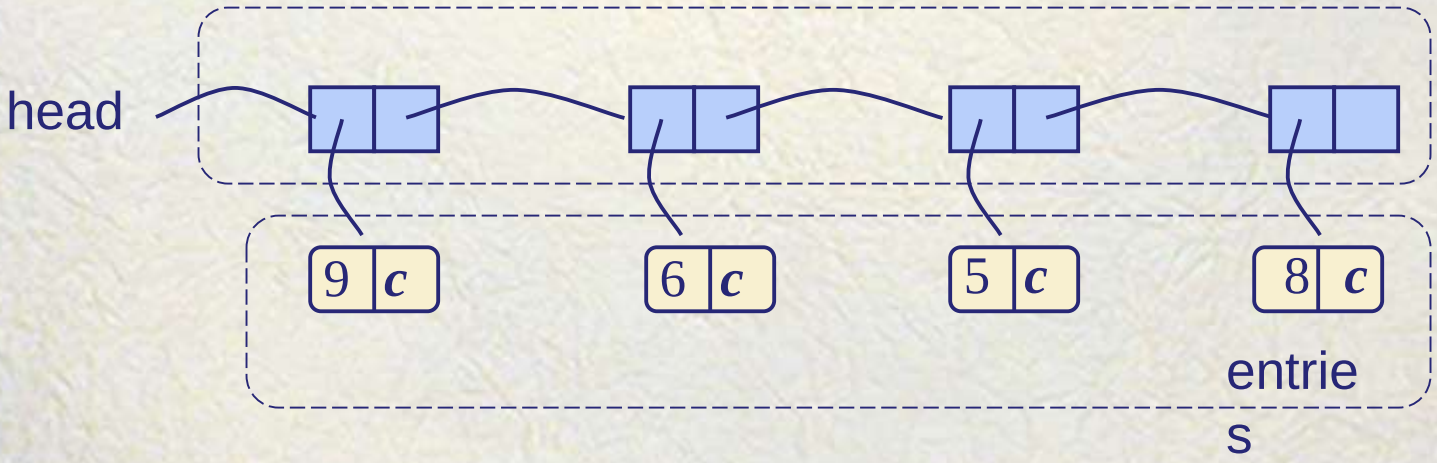
- get(k): if the map M has an entry with key k, return its associated value; else, return null
- put(k, v): insert entry (k, v) into the map M; if key k is not already in M, then return null; else, return old value associated with k
- remove(k): if the map M has an entry with key k, remove it from M and return its associated value; else, return null

Example

Operation	*Output*	*Map*
put(5,*A*)	null	(5,*A*)
put(7,*B*)	null	(5,*A*),(7,*B*)
put(2,*C*)	null	(5,*A*),(7,*B*),(2,*C*)
put(8,*D*)	null	(5,*A*),(7,*B*),(2,*C*),(8,*D*)
put(2,*E*)	*C*	(5,*A*),(7,*B*),(2,*E*),(8,*D*)
get(7)	*B*	(5,*A*),(7,*B*),(2,*E*),(8,*D*)
get(4)	null	(5,*A*),(7,*B*),(2,*E*),(8,*D*)
get(2)	*E*	(5,*A*),(7,*B*),(2,*E*),(8,*D*)
remove(5)	*A*	(7,*B*),(2,*E*),(8,*D*)
remove(2)	*E*	(7,*B*),(8,*D*)
get(2)	null	(7,*B*),(8,*D*)

A Simple List-Based Map

We can easily implement a map using a singly linked list



The get(k) Algorithm

Algorithm get(*k*):
    p = head;
    while *p is not null* **do**
        if *p*→element.key = *k* **then**
            return *p*→element.value;
        p = p→next;
    return null {there is no entry with key equal to *k*};

Complexity?

The put(k,v) Algorithm

```
Algorithm put(k,v):
    p = head;
    while p is not null do
        if p.element.key = k then
            t = p→element.value;
            p→element.value = v;
            return t {return the old value};
        p = p→next;
    insertLast((k,v));
    return null    {there was no previous entry with key equal to k};
```

Complexity?

The remove(k) Algorithm

```
Algorithm remove(k):
    p = head;
    while p is not null do
        if p.element.key = k  then
            t = p→element.value;
            remove (p);
            return t  {return the old value};
        p = p→next;
    return null {there is no entry with key equal to k};
```

Complexity?

Performance of a List-Based Map

- get(k): O(n)
- put(k, v): O(n)
- remove(k): O(n)

Need a data structure to implement map efficiently

Using map library

```
#include <iostream>
#include <string>
#include <map>

int main (){
   std::map<char,int> mymap;
   mymap['a']=10;
   mymap['b']=20;
   mymap['c']=30;

   mymap.at('a') = 15;
   mymap.at('b') = 50;

   std::map<char,int>::iterator it;
   for (it=mymap.begin(); it!=mymap.end(); ++it)
      std::cout << it->first << ": " << it->second << '\n';

   return 0;
}
```

Simple map

How to implement a simple map whose keys are integer numbers in range 0…1000?

Key	**0**	**1**	**2**	**500**	**1000**
Value	A	B	B	G	F

0	1	2			500				1000
A	B	B			G				F

Using an array of 1001 elements to store a map

Complexity of simple map operations

- get(k):

  return M[k];

  Complexity: O(1)
- put(k, v):

  old_value = M[k];

  M[k] = v;

  return old_value;

  Complexity: O(1)
- remove(k):

  M[k] = Null;

  Complexity: O(1)

Hash Tables

Hash Functions

A hash function h maps keys of a given type to integers in a fixed interval $[0, N-1]$. The integer $h(x)$ is called the hash value of key x

Example: $h(x) = x \bmod N$

Key	0	3	1001	1002	1005
h(key)	0	3	1	2	5
Value	A	B	B	G	F

h(key) = key mod 1000

Hash Tables

A hash table is used to store a map after hashing the keys

Example: Hashing keys by a hash function

Key	**0**	**3**	**1001**	**1002**	**1005**
h(key)	0	3	1	2	5
Value	A	B	B	G	F

Hash table

0	1	2	3		5				1000
A	B	G	B		F				

Collision

The hash values of two keys might have the same value (collision) causing different elements are mapped to the same cell on the hash table.

Key	**0**	**3**	**1001**	**1003**	**1005**
h(key)	0	3	1	3	5
Value	A	B	B	G	F

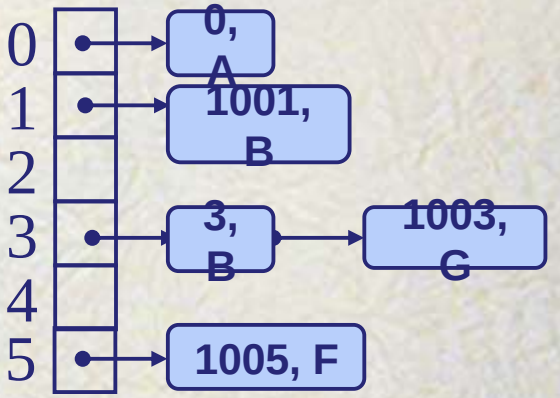
h(key) = key mod 1000

Collision Handling

Separate Chaining: let each cell in the table point to a linked list of entries that map there

Key	0	3	1001	1003	1005
h(key)	0	3	1	3	5
Value	A	B	B	G	F

h(key) = key mod 1000



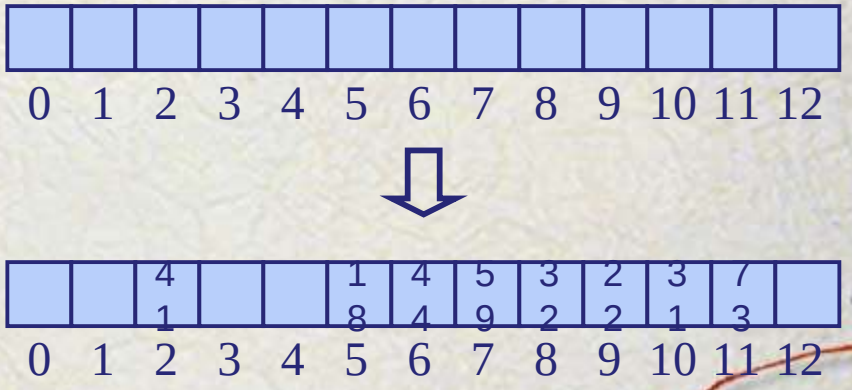
Hash table

Linear probing collions

- Open addressing: the colliding item is placed in a different cell of the table
- **Linear probing** handles collisions by placing the colliding item in the next (circularly) available table cell
- Each table cell inspected is referred to as a “probe”
- Colliding items lump together, causing future collisions to cause a longer sequence of probes

Example:

- $h(x) = x \bmod 13$
- Insert keys 18, 41, 22, 44, 59, 32, 31, 73



Hash Functions

A hash function: ***h(key)*** = key mod *N*

The number *N* should be a prime number to avoid as many collisions as possible.

Example:

{200, 205, 210, 215,.., 600}:

- 3 collisions with N=100
- No collision with N=101

Applications

- Digital signatures
- Message-authentication code (MAC)
- Password tables
- Key updating: key is hashed at specific intervals resulting in new key

Map operations with Separate Chaining used for Collisions

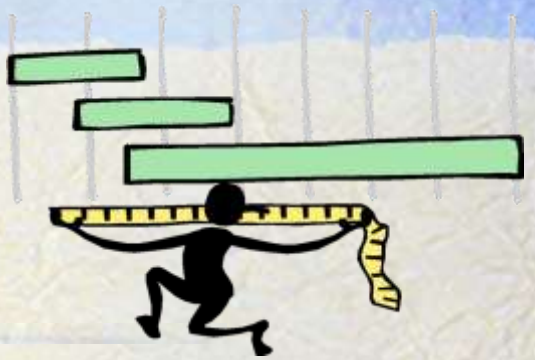
Let A[h(k)] be a linked list to store the map of all elements whose hash values are h(k).

Algorithm get(*k*):
    return *A*[*h*(*k*)].get(*k*);

Algorithm put(*k*,*v*):
    return A[*h*(*k*)].put(*k*,*v*);

Algorithm remove(*k*):
    return A[*h*(*k*)].remove(*k);*

Performance and applications



- In the worst case, searches, insertions and removals on a hash table take $O(n)$ time
- The load factor $\alpha = n/N$ affects the performance of a hash table
- In practice, hashing is very fast provided the load factor is not close to 100%
- The expected running time of map operations in a hash table is $O(1)$

Exercises

- Given a list of students (id, name):

  (1,A), (3,B), (10, C), (54, D) , (9, A)

Your task is to propose a hash function, and draw the hash table with the proposed hash function using both collision handling methods

- Given a list of students (id, name):

  (7,A), (3,B), (11, C), (4, D) , (8, G), (16, E), (21, B), (5, G)

Your task is to propose a hash function, and draw the hash table with the proposed hash function using both collision handling methods