Sets, Maps and Dictionary

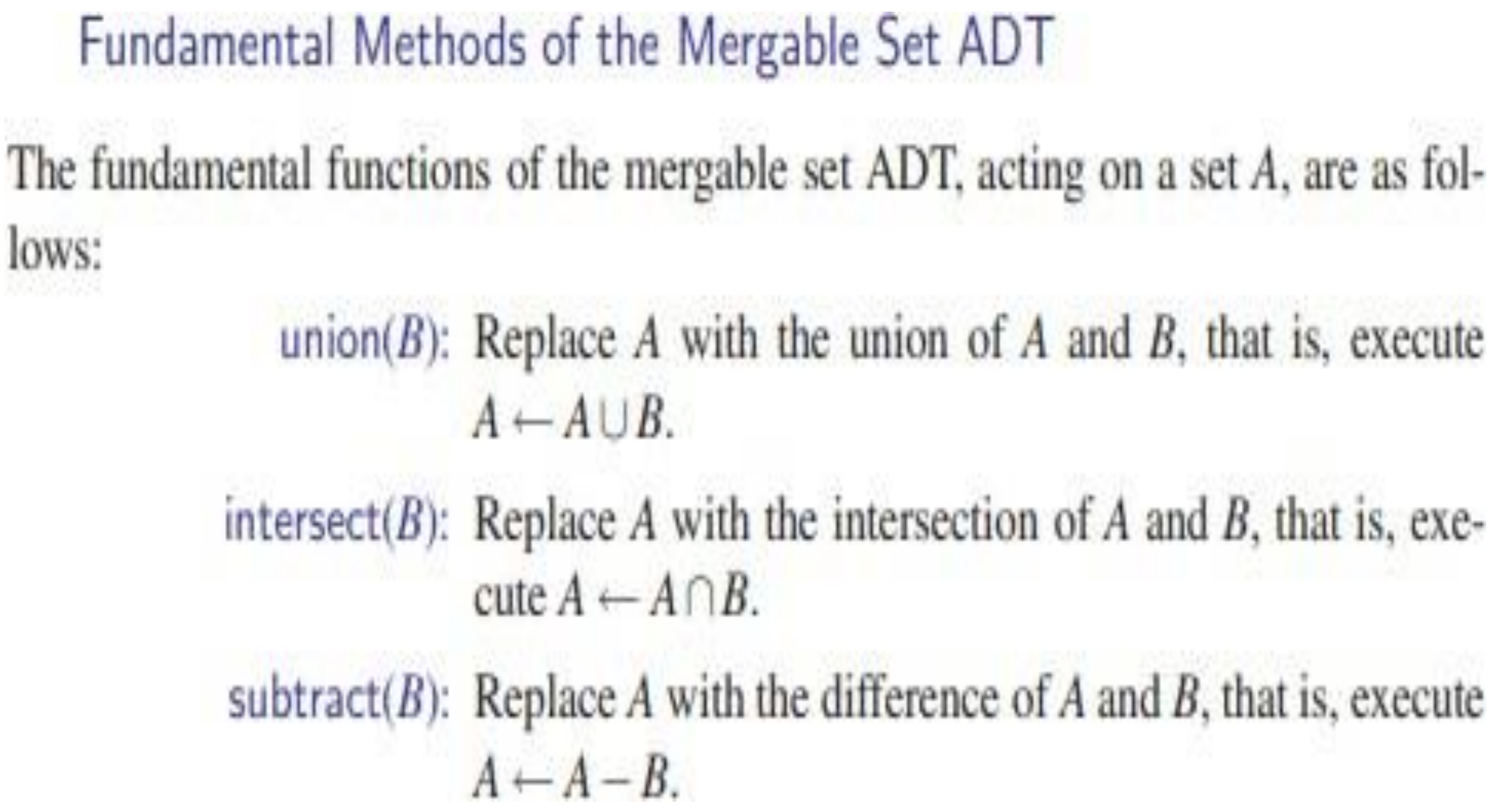
Ms. Swati Mali
swatimali@somaiya.edu

Ref: Data Structures and Algorithms in C++ 2e By Michael Goodrich, Roberto Tamassia and David Mount

Sets

- A set is defined as a collection that contains no duplicates
- Basic Operations we perform with sets are
  - Set Union(S1 U S2)
  - Set Intersection(S1 ∩ S2)
  - Set Difference(S1 - S2)

Sets



Fundamental Methods of the Mergable Set ADT

The fundamental functions of the mergable set ADT, acting on a set A, are as follows:

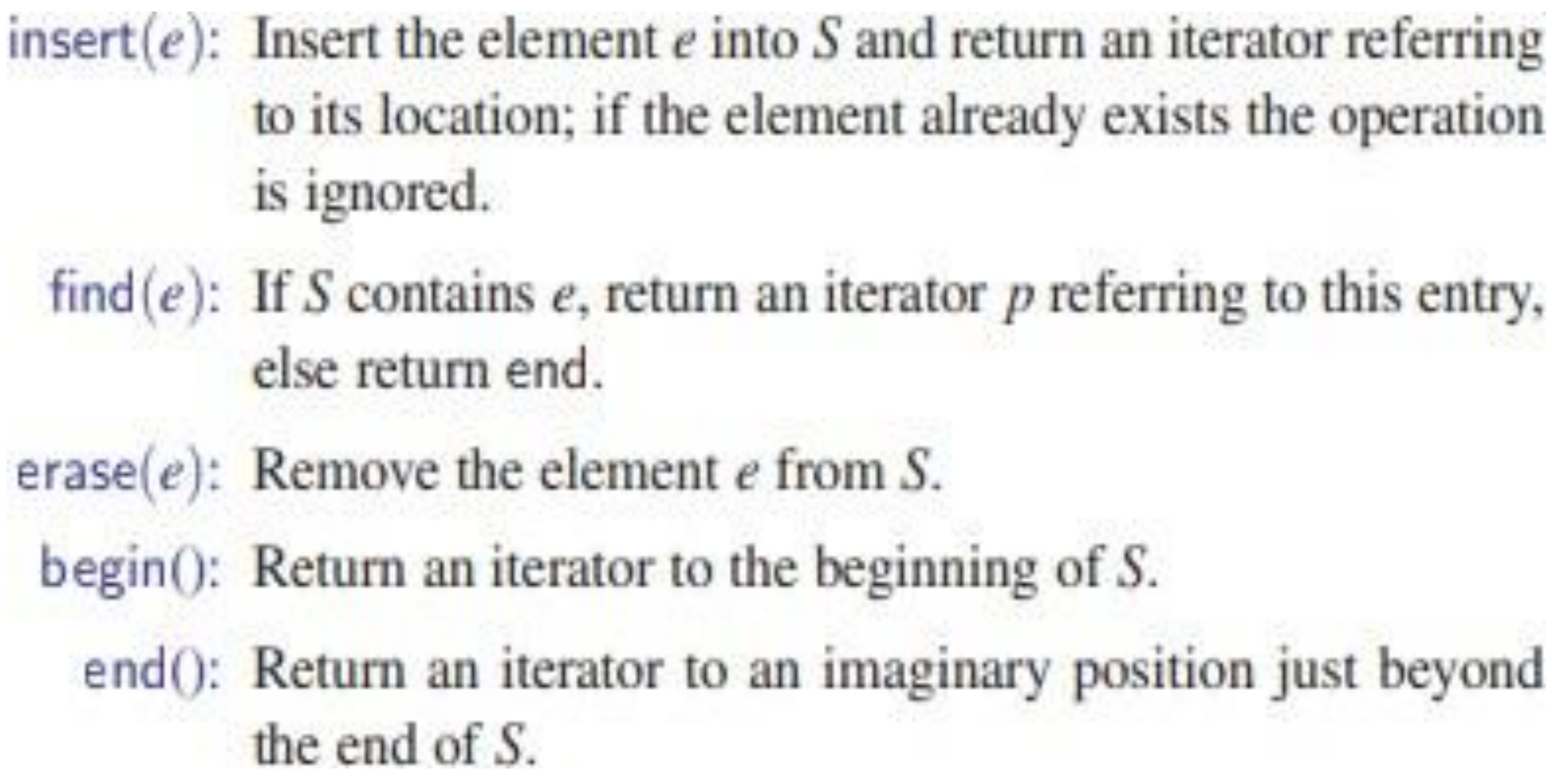
union(B): Replace A with the union of A and B, that is, execute $A \leftarrow A \cup B$.

intersect(B): Replace A with the intersection of A and B, that is, execute $A \leftarrow A \cap B$.

subtract(B): Replace A with the difference of A and B, that is, execute $A \leftarrow A - B$.

Sets ADT

Set ADT provides number of methods.



insert(e):	Insert the element e into S and return an iterator referring to its location; if the element already exists the operation is ignored.
find(e):	If S contains e, return an iterator p referring to this entry, else return end.
erase(e):	Remove the element e from S.
begin():	Return an iterator to the beginning of S.
end():	Return an iterator to an imaginary position just beyond the end of S.

basic functions associated with Set:

Set in C++ Standard Template Library (STL)

- begin() – Returns an iterator to the first element in the set.
- end() – Returns an iterator to the theoretical element that follows last element in the set.
- size() – Returns the number of elements in the set.
- max_size() – Returns the maximum number of elements that the set can hold.
- empty() – Returns whether the set is empty.

Reading assignment: Set in C++ Standard Template Library (STL)

- rbegin()– Returns a reverse iterator pointing to the last element in the container.
- rend()– Returns a reverse iterator pointing to the theoretical element right before the first element in the set container.
- crbegin()– Returns a constant iterator pointing to the last element in the container.
- crend() – Returns a constant iterator pointing to the position just before the first element in the container.

Reading assignment: Set in C++ Standard Template Library (STL)

- cbegin()– Returns a constant iterator pointing to the first element in the container.
- cend() – Returns a constant iterator pointing to the position past the last element in the container.
- size() – Returns the number of elements in the set.
- max_size() – Returns the maximum number of elements that the set can hold.
- empty() – Returns whether the set is empty.

Reading assignment: Set in C++ Standard Template Library (STL)

- insert(const g) – Adds a new element ‘g’ to the set.
- iterator insert (iterator position, const g) – Adds a new element ‘g’ at the position pointed by iterator.
- erase(iterator position) – Removes the element at the position pointed by the iterator.
- erase(const g)– Removes the value ‘g’ from the set.
- clear() – Removes all the elements from the set.

Reading assignment: Set in C++ Standard Template Library (STL)

- key_comp() / value_comp() – Returns the object that determines how the elements in the set are ordered ('<' by default).
- find(const g) – Returns an iterator to the element 'g' in the set if found, else returns the iterator to end.
- count(const g) – Returns 1 or 0 based on the element 'g' is present in the set or not.
- lower_bound(const g) – Returns an iterator to the first element that is equivalent to 'g' or definitely will not go before the element 'g' in the set.

Reading assignment: Set in C++ Standard Template Library (STL)

- upper_bound(const g) – Returns an iterator to the first element that will go after the element ‘g’ in the set.
- equal_range()– The function returns an iterator of pairs. (key_comp). The pair refers to the range that includes all the elements in the container which have a key equivalent to k.
- emplace()– This function is used to insert a new element into the set container, only if the element to be inserted is unique and does not already exists in the set.

Reading assignment: Set in C++ Standard Template Library (STL)

- emplace_hint()– Returns an iterator pointing to the position where the insertion is done. If the element passed in the parameter already exists, then it returns an iterator pointing to the position where the existing element is.
- swap()– This function is used to exchange the contents of two sets but the sets must be of same type, although sizes may differ.
- operator= – The '=' is an operator in C++ STL which copies (or moves) a set to another set and set::operator= is the corresponding operator function.
- get_allocator()– Returns the copy of the allocator object associated with the set.

Reading assignment

Refer: https://www.geeksforgeeks.org/set-in-cpp-stl/

Disjoint sets and partitions

A1 and A2 are called disjoint partitions of A iff

- A1 U A2 = A
- A1 ∩ A2 = Φ

- E.g. A1={1,2,3,4,5} and A2= {2,4,6}, A3= {6,7} and A= {1,2,3,4,5,6,7}
- A1 and A2 are not disjoint partitions of A
- A1 and A3 are disjoint partitions of A

Set partition using union- find operation

- Union: creates disjoint subsets
- Find: checks connectivity

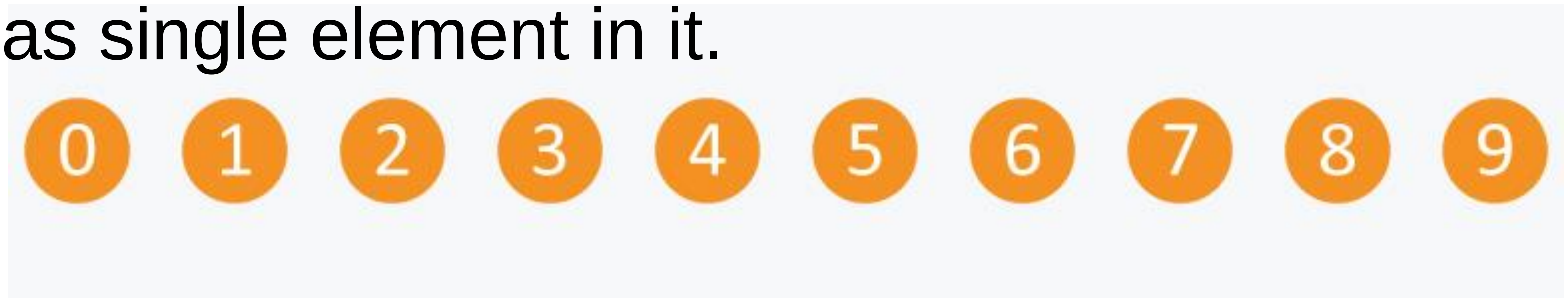
Example

Example:

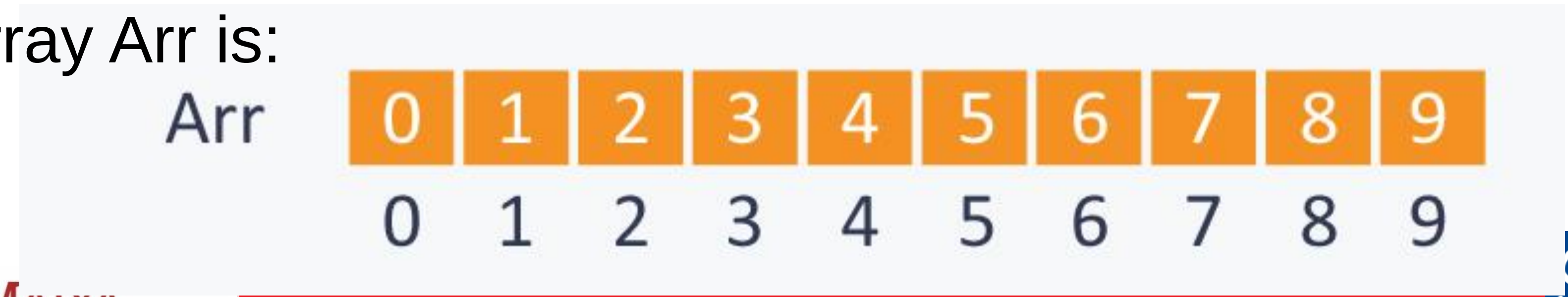
S = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9}.

N = 10

Initially there are 10 subsets and each subset has single element in it.



When each subset contains only single element, the array Arr is:



Ref: https://www.hackerearth.com/practice/notes/disjoint-set-union-union-find/

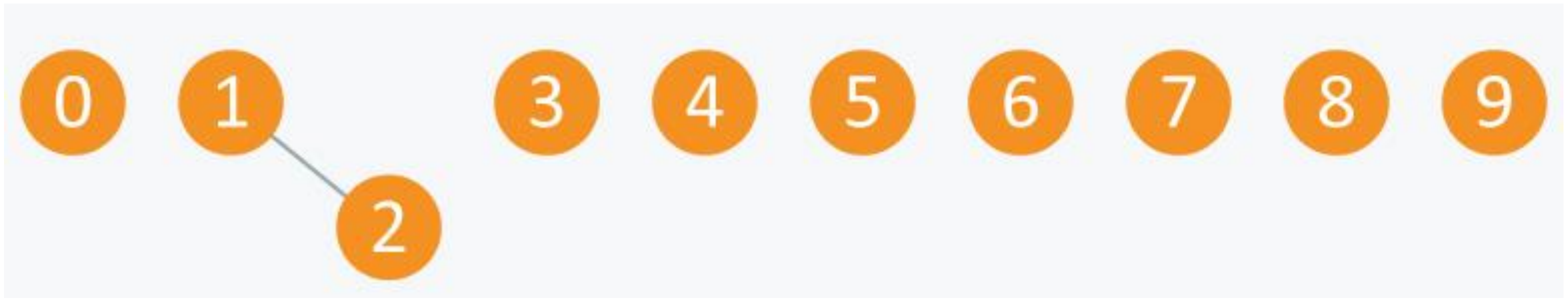
Example

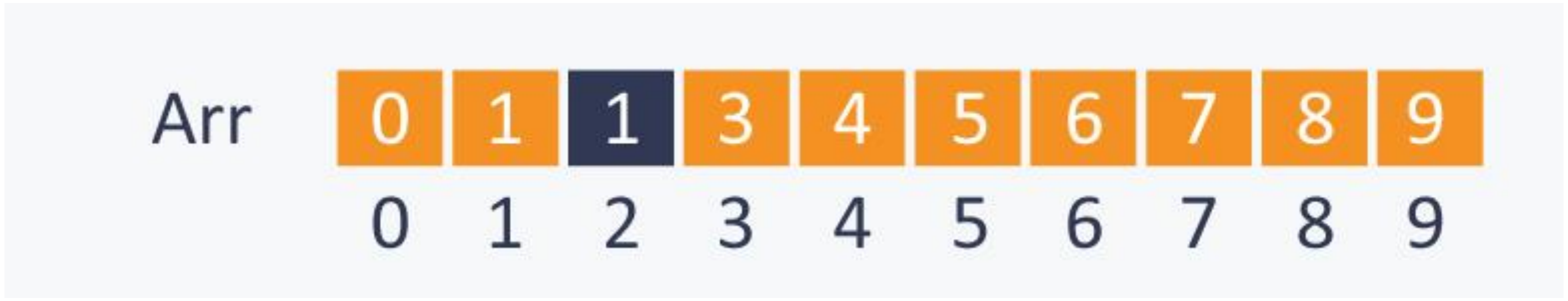
Perform the following operations on the set:

1) Union(2, 1))=**(1st element=>child,2nd element=parent)**
2) Union(4, 3)
3) Union(8, 4)
4) Union(9, 3)
5) Union(6, 5)
6) Union(5, 2)

Find(6,1), find(8,9) find(7,1)

1) Union(2, 1)

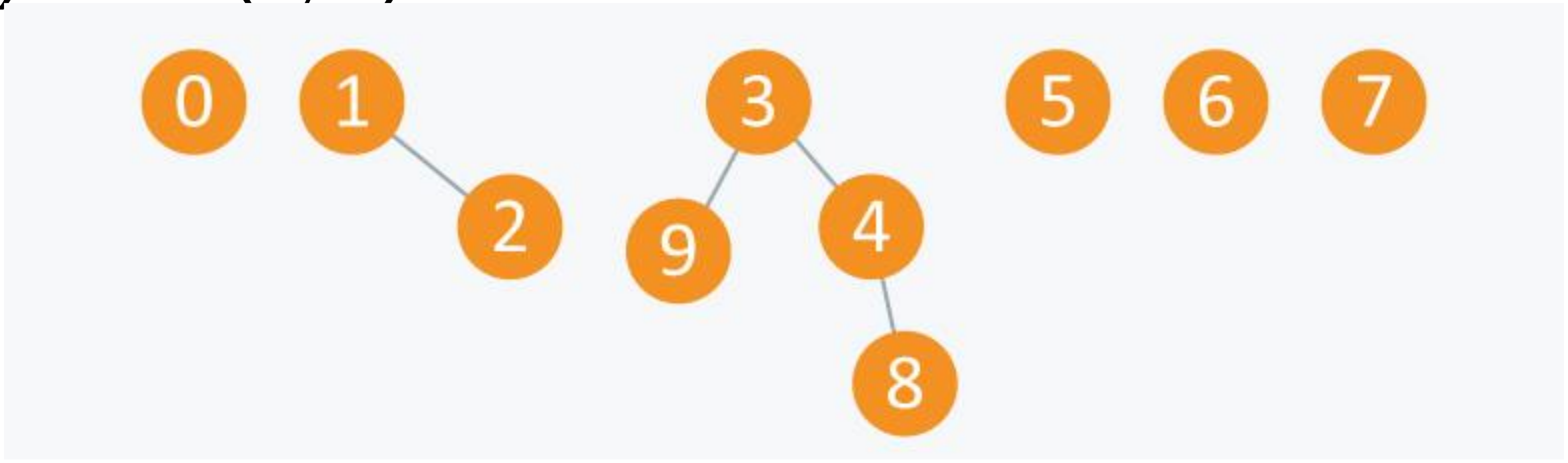


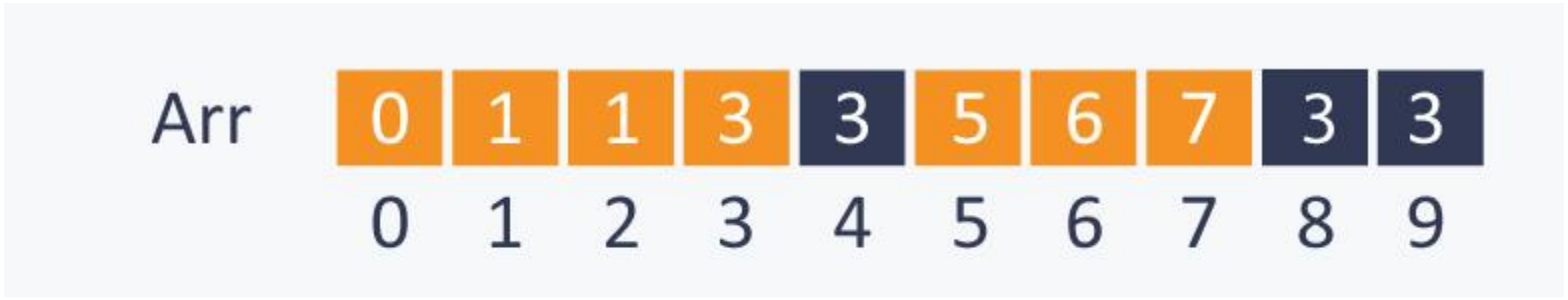


Arr	0	1	1	3	4	5	6	7	8	9
	0	1	2	3	4	5	6	7	8	9

Ref: https://www.hackerearth.com/practice/notes/disjoint-set-union-union-find/

2) Union(4, 3)
3) Union(8, 4)
4) Union(9, 3)

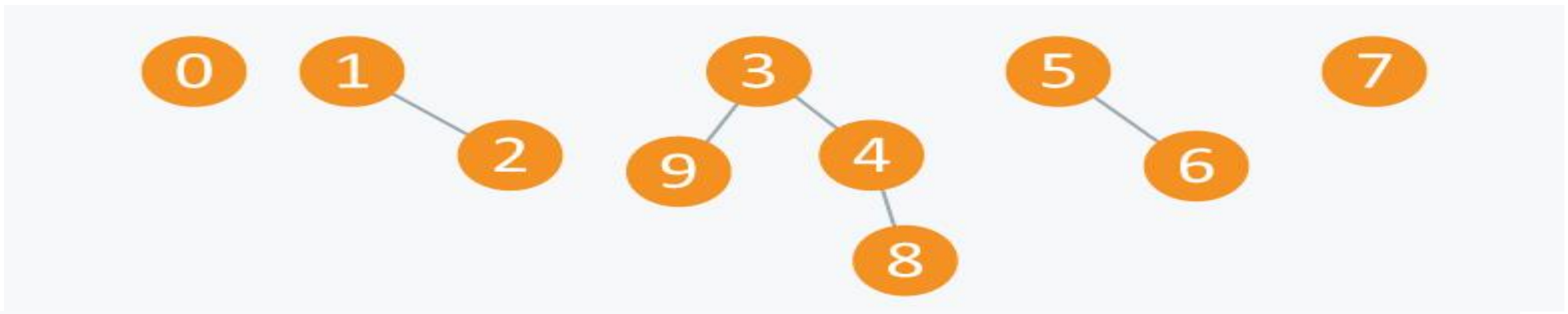


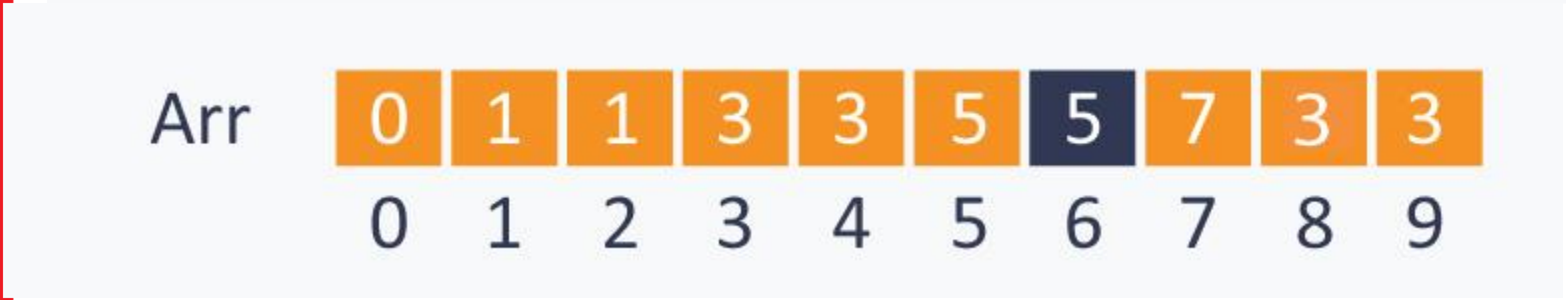


	0	1	2	3	4	5	6	7	8	9
Arr	0	1	1	3	3	5	6	7	3	3

Ref: https://www.hackerearth.com/practice/notes/disjoint-set-union-union-find/

5) Union(6, 5)





Arr	0	1	1	3	3	5	5	7	3	3
	0	1	2	3	4	5	6	7	8	9

➔5 subsets.

A1= {3, 4, 8, 9},

A2= {1, 2},

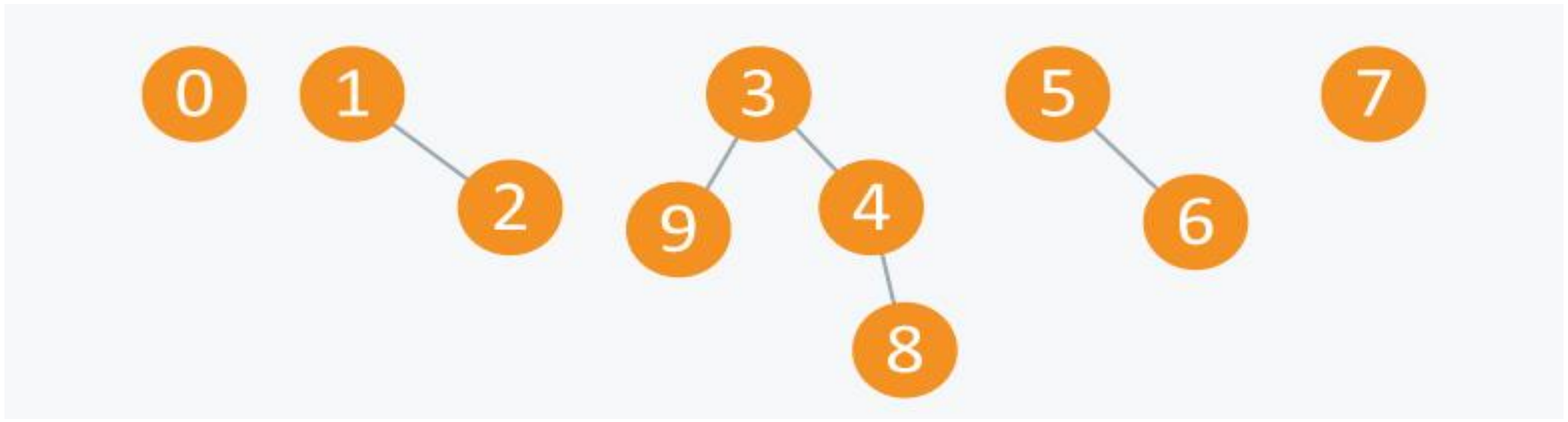
A3= {5, 6}

A4= {0}

A5 = {7}.

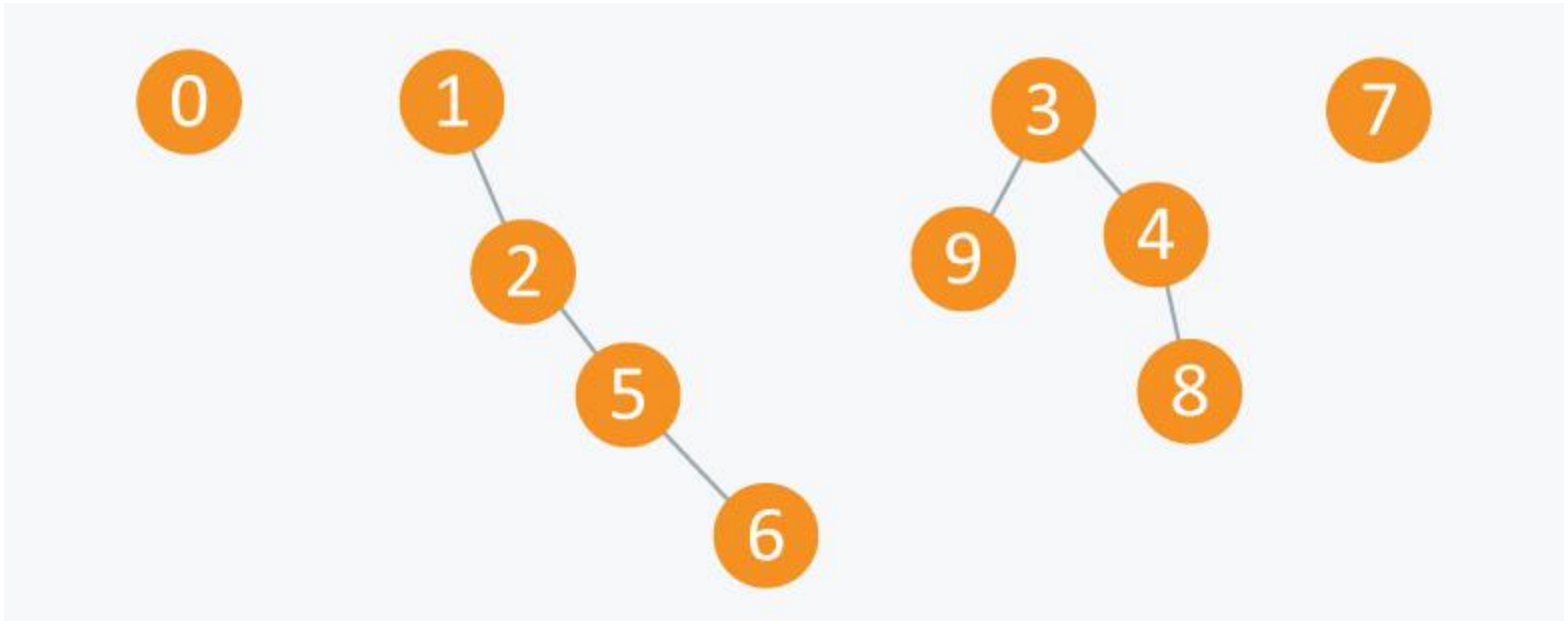
All these subsets are said to be Connected Components.

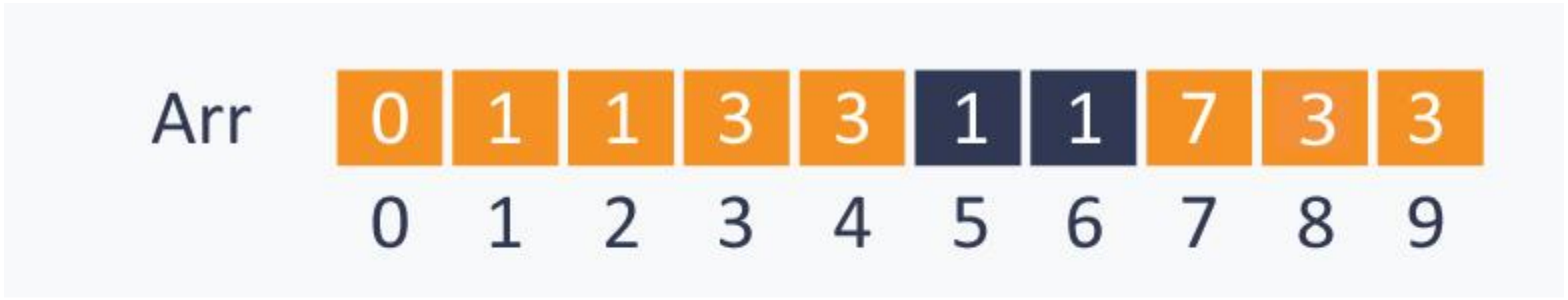
- Find (0, 7) = False as 0 and 7 are disconnected
- Find (8, 9) = True as 8 and 9 are connected directly or indirectly



Ref: https://www.hackerearth.com/practice/notes/disjoint-set-union-union-find/

6) Union(5, 2)





Arr										
	0	1	1	3	3	1	1	7	3	3
	0	1	2	3	4	5	6	7	8	9

Ref: https://www.hackerearth.com/practice/notes/disjoint-set-union-union-find/

Applications of set partitioning

- Elections
- Divide and conquer
- Classification
- Pattern matching
- Mutually exclusive processes in OS
- Combinatorial explosion problem where repetition is not allowed

Applications of set partitioning

- commonly used in a variety of computer science applications, including algorithms, data analysis, and databases.
- The main advantage of using a set data structure is that it allows you to perform operations on a collection of elements in an efficient and organized way.

Maps

Also known as:

- table, search table, associative array, or associative container

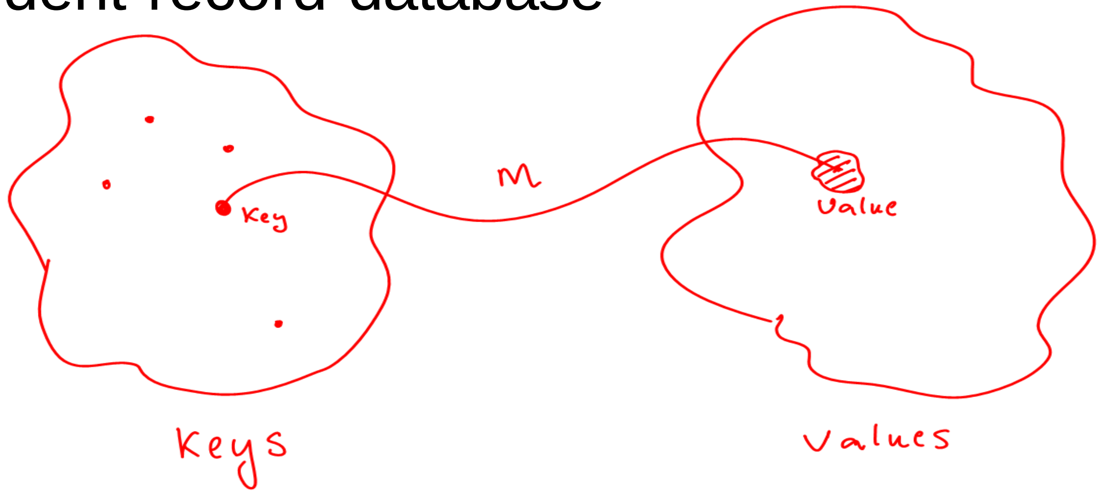
A data structure optimized for a very specific kind of search / access

- with a *bag* we access by asking "is X present"
- with a *list* we access by asking "give me item number X"
- with a *queue* we access by asking "give me the item that has been in the collection the longest."

In a *Map* we access by asking "give me the *value* associated with this *key.*"

Maps

- A Map models a searchable collection of key-value entries
- The main operations of a map are for searching, inserting, and deleting items
- Multiple entries with the same key are **not** allowed
- Applications:
  - address book
  - student-record database



Maps

- A ***map*** allows to store elements so they can be located quickly using keys.
- ***key*** as a unique identifier
- A map stores key-value pairs (*k*,*v*), called ***entries***,
- each key is unique, so the association of keys to values defines a mapping.
- E.g. In a map storing student records (such as the student's name, address, and course grades), the key might be the student's ID number.
- Sometimes referred to as ***associative stores*** or ***associative containers***, as the key associated with an object determines its "location" in the data structure

Maps

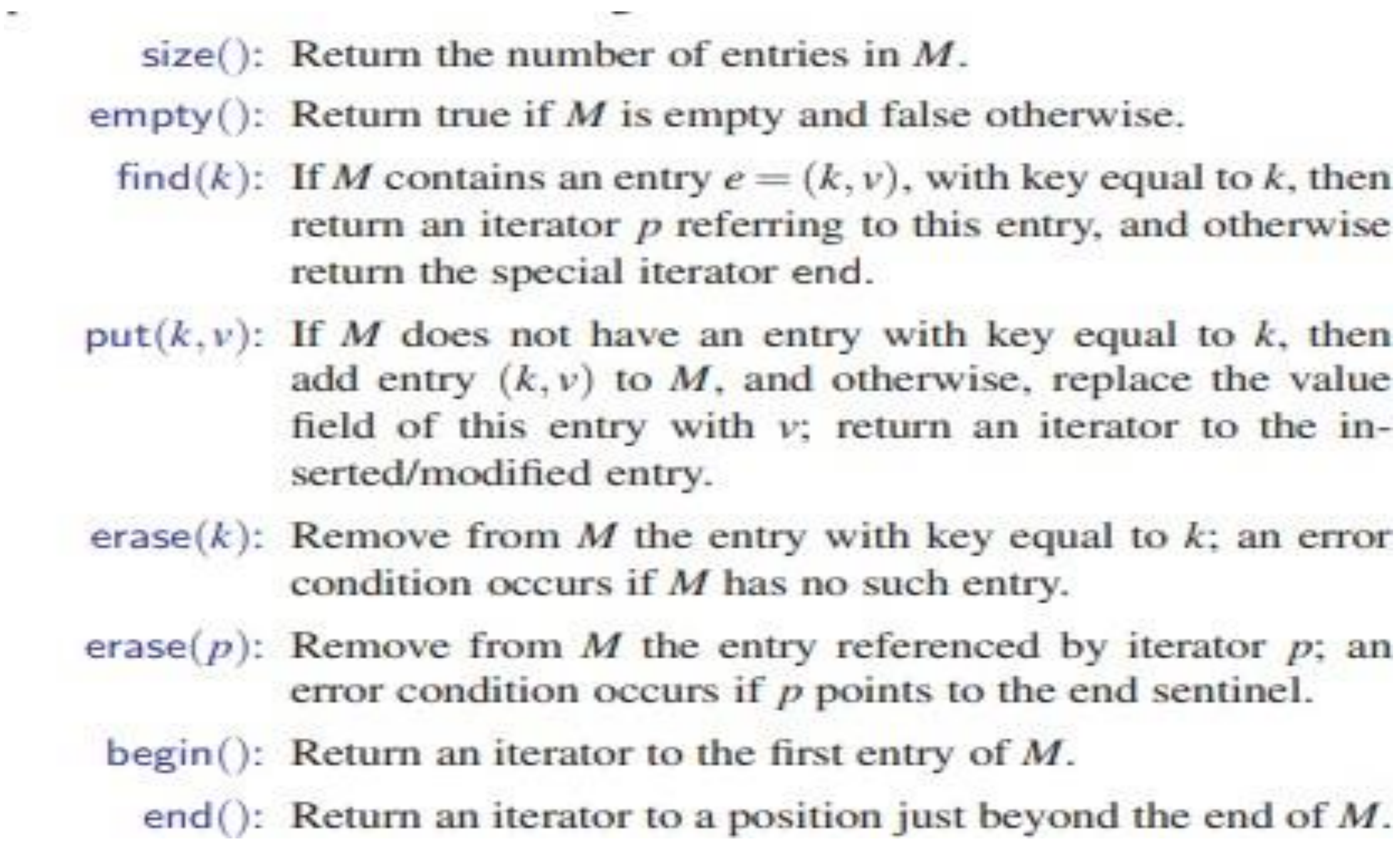
- Used where each key is to be viewed as a kind of unique ***index*** address for its value, that is,
- E.g. if we wish to store student records, we would probably want to use student ID objects as keys (and disallow two students having the same student ID).
- In other words, the key associated with an object can be viewed as an “address” for that object.

- ideal to use in look-up type situations where there is an identifying value and an actual value that is represented by the identifying value.
- examples :
  - Student ID numbers and last names.
  - House numbers on a street and the number of pets in each house

Map ADT

- Value definition: Map is a collection of key value entries, with each value associated with a distinct key.
- Assumption: map provides a special pointer object, which permits us to reference entries of the map, called ***position***.
- ***Iterator*** references entries and navigate around the map.
- Given a map iterator *p*, the associated entry may be accessed by dereferencing the iterator, **p*.
- The individual key and value can be accessed using *p*->key() and *p*->value(), respectively.
- Can be implemented using associative arrays

Map ADT



size():	Return the number of entries in M.
empty():	Return true if M is empty and false otherwise.
find(k):	If M contains an entry $e = (k, v)$, with key equal to k, then return an iterator p referring to this entry, and otherwise return the special iterator end.
put(k, v):	If M does not have an entry with key equal to k, then add entry (k, v) to M, and otherwise, replace the value field of this entry with v; return an iterator to the inserted/modified entry.
erase(k):	Remove from M the entry with key equal to k; an error condition occurs if M has no such entry.
erase(p):	Remove from M the entry referenced by iterator p; an error condition occurs if p points to the end sentinel.
begin():	Return an iterator to the first entry of M.
end():	Return an iterator to a position just beyond the end of M.

MAP

A map is any data structure that groups a dynamic number of *key-value pairs* together,

Map allows us to

retrieve values by key,

to insert new key-value pairs, and

to update the values associated with keys.

Courtesy: https://noncombatant.org/2014/03/02/maps-and-their-applications/

MAP

A *Map* is a type of fast key lookup data structure that offers a flexible means of indexing into its individual elements.
Unlike most **array data structures** that

> **only allow access to the elements by means of integer** indices,
> **the indices for a Map can be nearly any scalar numeric value or a character vector.**

MAP

Indices into the elements of a Map are called *keys*. These keys, along with the data *values* associated with them, are stored within the Map.

Each entry of a Map contains exactly one unique key and its corresponding value.

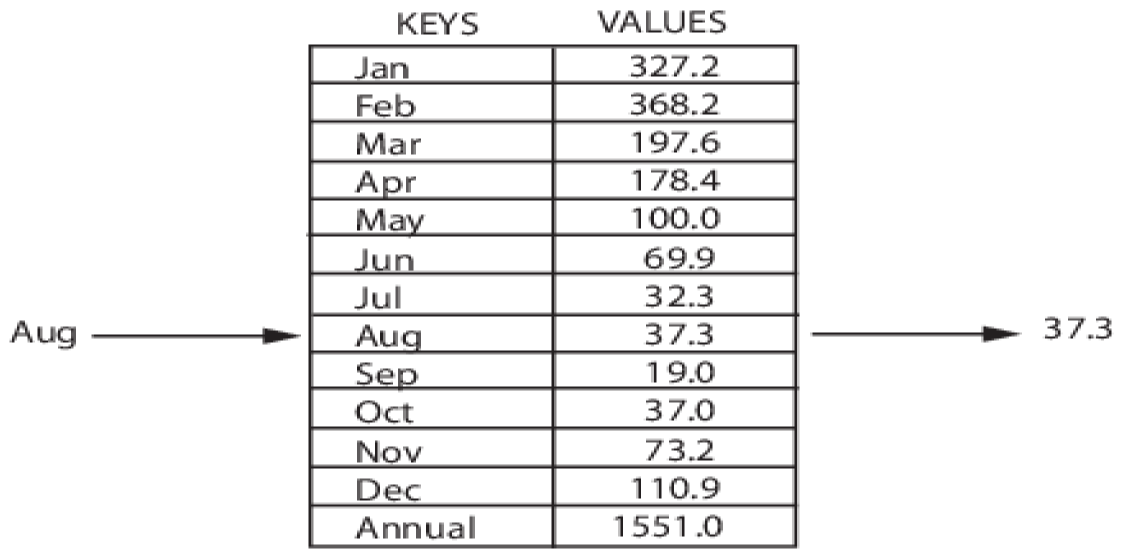
No two mapped values can have same key values.

A map cannot contain duplicate keys

MAP-Example

Indexing into the Map of rainfall statistics shown below with a character vector representing the month of August yields the value internally associated with that month, 37.3.

Mean monthly rainfall statistics (mm)



KEYS	VALUES
Jan	327.2
Feb	368.2
Mar	197.6
Apr	178.4
May	100.0
Jun	69.9
Jul	32.3
Aug	37.3
Sep	19.0
Oct	37.0
Nov	73.2
Dec	110.9
Annual	1551.0

MAP

Keys are not restricted to integers as they are with other arrays.

Specifically, a key may be any of the following types:

- 1-by-N character array
- **Scalar real double** or single
- **Signed or unsigned scalar integer**

MAP

The values stored in a Map can be of any type.
This includes

- **arrays of numeric values,**
- **structures,**
- **cells,**
- **character arrays,**
- objects, or
- **other Maps.**

MAP ADT Functions

Some basic functions associated with Map:
begin() – Returns an iterator to the first element in the map
end() – Returns an iterator to the theoretical element that follows last element in the map
size() – Returns the number of elements in the map
max_size() – Returns the maximum number of elements that the map can hold
empty() – Returns whether the map is empty
pair insert(keyvalue, mapvalue) – Adds a new element to the map
erase(iterator position) – Removes the element at the position pointed by the iterator
clear() – Removes all the elements from the map

begin() function

Used to return an iterator pointing to the first element of the map container.
begin() function returns a bidirectional iterator to the first element of the container.

Syntax :
***mapname*.begin()**
Parameters : No parameters are passed.
Returns : This function returns a bidirectional iterator pointing to the first element.

Demonstrates begin() and end()

```
#include <iostream>
#include <map>
using namespace std;

int main()
{
    // declaration of map container
    map<char, int> mymap;
    mymap['a'] = 1;
    mymap['b'] = 2;
    mymap['c'] = 3;

    // using begin() to print map
    for (auto it = mymap.begin(); it != mymap.end(); ++it)
        cout << it->first << " = "
             << it->second << '\n';
    return 0;
}
```

Output:

```
a = 1
b = 2
c = 3
```

What is a map in C++?
A C++ map is a way to store a key-value pair.

Maps are part of the C++ STL (Standard Template Library).

The Standard Template Library (STL) is a set of C++ template classes to provide common programming data structures and functions such as lists, stacks, arrays, etc.
It is a library of container classes, algorithms, and iterators.

https://www.geeksforgeeks.org/the-c-standard-template-library-stl/

Standard Containers

A container is a holder object that stores a collection of other objects (its elements).

The container manages the storage space for its elements and provides member functions to access them, either directly or through iterators (reference objects with similar properties to pointers).

https://cplusplus.com/reference/iolibrary/

end() function

end() function is used to return an iterator pointing to past the last element of the map container.
Since it does not refer to a valid element, it cannot de-referenced end() function returns a bidirectional iterator.
Syntax :
***mapname*.end()**
Parameters : No parameters are passed.
Returns : This function returns a bidirectional iterator pointing to the next of last element.

insert()

A built-in function in C++ STL which is used to insert elements with a particular key in the map container.

Syntax:

iterator map_name.insert({key, element})

Parameters:

The function **accepts a pair that consists of a key and element** which is to be inserted into the map container. The function does not insert the key and element in the map if the key already exists in the map.

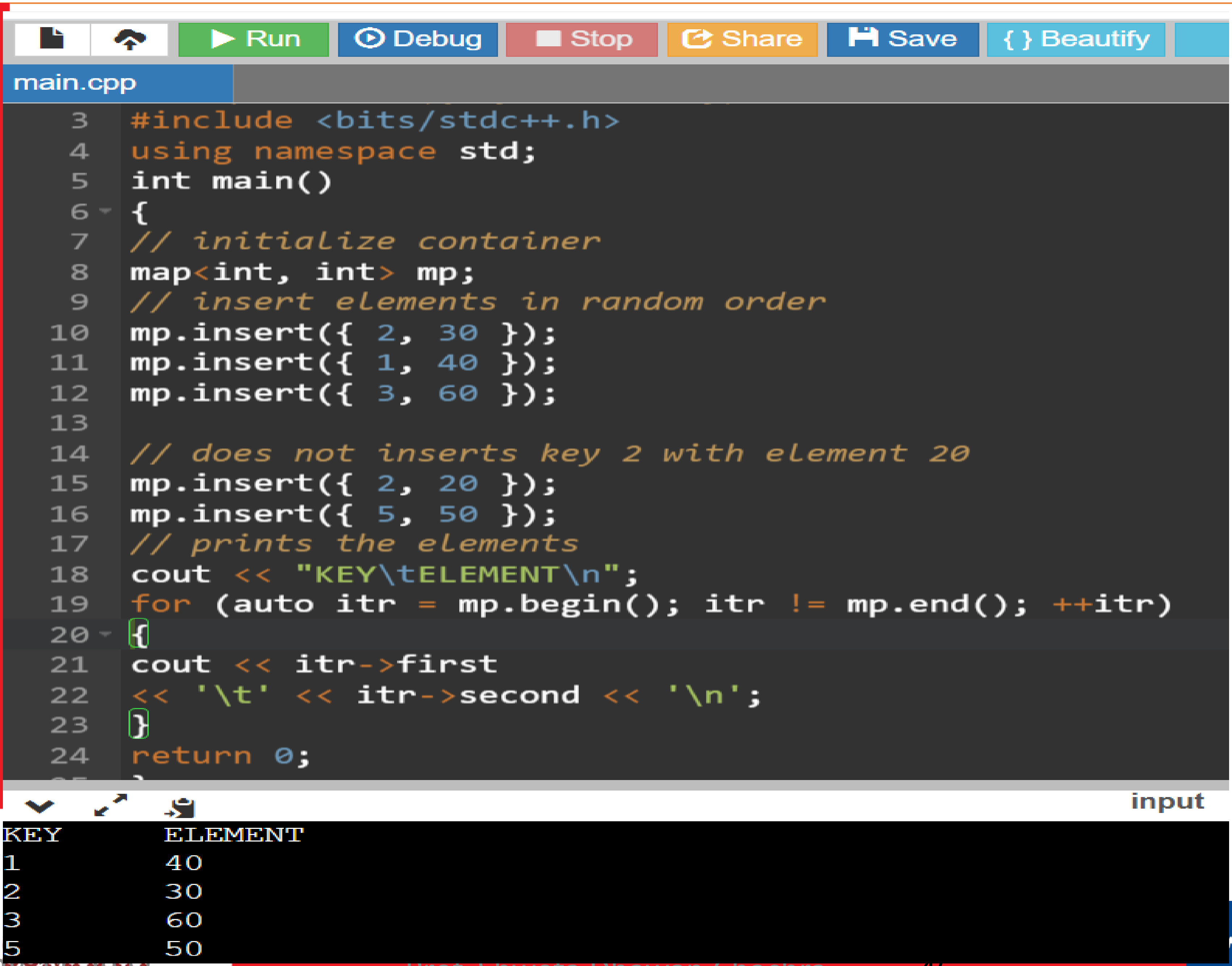
Return Value:

The function returns an iterator pointing to the new element in the container.

insert()

```
// C++ program to illustrate
// map::insert({key, element})
#include <bits/stdc++.h>
using namespace std;

int main()
{
    // initialize container
    map<int, int> mp;

    // insert elements in random order
    mp.insert({ 2, 30 });
    mp.insert({ 1, 40 });
    mp.insert({ 3, 60 });

// does not inserts key 2 with element
20
    mp.insert({ 2, 20 });
    mp.insert({ 5, 50 });

    // prints the elements
    cout << "KEY\tELEMENT\n";
    for (auto itr = mp.begin(); itr !=
mp.end(); ++itr) {
        cout << itr->first
             << '\t' << itr->second << '\n';
    }
    return 0;
}
```

```
OUTPUT-
KEY ELEMENT
1 40
2 30
3 60
5 50
```



```cpp
#include <bits/stdc++.h>
using namespace std;
int main()
{
// initialize container
map<int, int> mp;
// insert elements in random order
mp.insert({ 2, 30 });
mp.insert({ 1, 40 });
mp.insert({ 3, 60 });

// does not inserts key 2 with element 20
mp.insert({ 2, 20 });
mp.insert({ 5, 50 });
// prints the elements
cout << "KEY\tELEMENT\n";
for (auto itr = mp.begin(); itr != mp.end(); ++itr)
{
cout << itr->first
<< '\t' << itr->second << '\n';
}
return 0;
```

```
KEY       ELEMENT
1         40
2         30
3         60
5         50
```

bits/stdc++.h

It is basically a header file that includes every standard library. In programming contests, Using this file is a good idea, when you want to reduce the time wasted in doing chores; especially when your rank is time sensitive.

In programming contests, people do focus more on finding the algorithm to solve a problem than on software engineering.

- From, software engineering perspective, it is a good idea to minimize the include.
- If you use it actually includes a lot of files, which your program may not need, thus increases both compile time and program size unnecessarily.

size() function

In C++, **size()** function is used to return the total number of elements present in the map.

Syntax:
map_name.size()

Return Value: It returns the number of elements present in the map.

size() function

```
Input : map1 = {
                 {1, "India"},
                 {2, "Nepal"},
                 {3, "Sri Lanka"},
                 {4, "Myanmar"}
                }
        map1.size();
Output: 4

Input : map2 = {};
        map2.size();
Output: 0
```

clear()

clear() function is used to remove all the elements from the map container and thus leaving it’s size 0.

Syntax:
map1.clear() where map1 is the name of the map.

Parameters:
No parameters are passed.

Return Value:
None

clear()

```
Input : map1 = {
                  {1, "India"},
                  {2, "Nepal"},
                  {3, "Sri Lanka"},
                  {4, "Myanmar"}
                }
         map1.clear();

Output: map1 = {}
```

clear()

```
#include <bits/stdc++.h>
using namespace std;

int main()
{
    // Take any two maps
    map<int, string> map1, map2;

    // Inserting values
    map1[1] = "India";
    map1[2] = "Nepal";
    map1[3] = "Sri Lanka";
    map1[4] = "Myanmar";

   // Print the size of map
    cout<< "Map size before running function:
\n";
    cout << "map1 size = " << map1.size() <<
endl;
    cout << "map2 size = " << map2.size() <<
endl;;
        // Deleting the map elements
    map1.clear();
    map2.clear();

    // Print the size of map
    cout<< "Map size after running function:
\n";
    cout << "map1 size = " << map1.size() <<
endl;
    cout << "map2 size = " << map2.size();
    return 0;
}
```

Output:

```
Map size before running function:
map1 size = 4
map2 size = 0

Map size after running function:
map1 size = 0
map2 size = 0
```

erase()

- A built-in function in C++ STL which is used to erase element from the container.
- It can be used to **erase keys, elements** at any specified position or a given range.
- **Syntax :**

map_name.erase(key)

Parameters:

The function accepts **one mandatory parameter *key*** which specifies the key to be erased in the map container.

Return Value:

The function **returns 1 if the key element is found** in the map else returns 0.

erase()

```
#include <bits/stdc++.h>
using namespace std;

int main()
{

    // initialize container
    map<int, int> mp;

    // insert elements in random order
    mp.insert({ 2, 30 });
    mp.insert({ 1, 40 });
    mp.insert({ 3, 60 });
    mp.insert({ 5, 50 });

// prints the elements
    cout << "The map before using erase() is : 
\n";
 cout << "KEY\tELEMENT\n";
                for (auto itr = mp.begin(); itr != 
mp.end(); ++itr) {
        cout << itr->first
             << '\t' << itr->second << '\n';
    }
   // function to erase given keys
    mp.erase(1);
    mp.erase(2);

    // prints the elements
    cout << "\nThe map after applying erase() is : 
\n";
    cout << "KEY\tELEMENT\n";
    for (auto itr = mp.begin(); itr != mp.end(); 
++itr) {
        cout << itr->first
             << '\t' << itr->second << '\n';
    }
    return 0;
}
```

erase()

```
The map before using erase() is :
KEY    ELEMENT
1    40
2    30
3    60
5    50

The map after applying erase() is :
KEY    ELEMENT
3    60
5    50
```

empty()

Used to check if the map container is empty or not.
Syntax :
***mapname*.empty()**
Parameters :
No parameters are passed.
Returns :
True, if map is empty
False, Otherwise

empty()

Examples:

```
Input  : map
         mymap['a']=10;
         mymap['b']=20;
         mymap.empty();
Output : False
```

Reading Assignment

https://www.geeksforgeeks.org/map-associative-containers-the-c-standard-template-library-stl/

Reading slides: Map library functions

Some basic functions associated with Map:
begin() – Returns an iterator to the first element in the map
end() – Returns an iterator to the theoretical element that follows last element in the map
size() – Returns the number of elements in the map
max_size() – Returns the maximum number of elements that the map can hold
empty() – Returns whether the map is empty

Reading slides: Map library functions

pair insert(keyvalue, mapvalue) – Adds a new element to the map

erase(iterator position) – Removes the element at the position pointed by the iterator

erase(const g)– Removes the key value ‘g’ from the map

clear() – Removes all the elements from the map

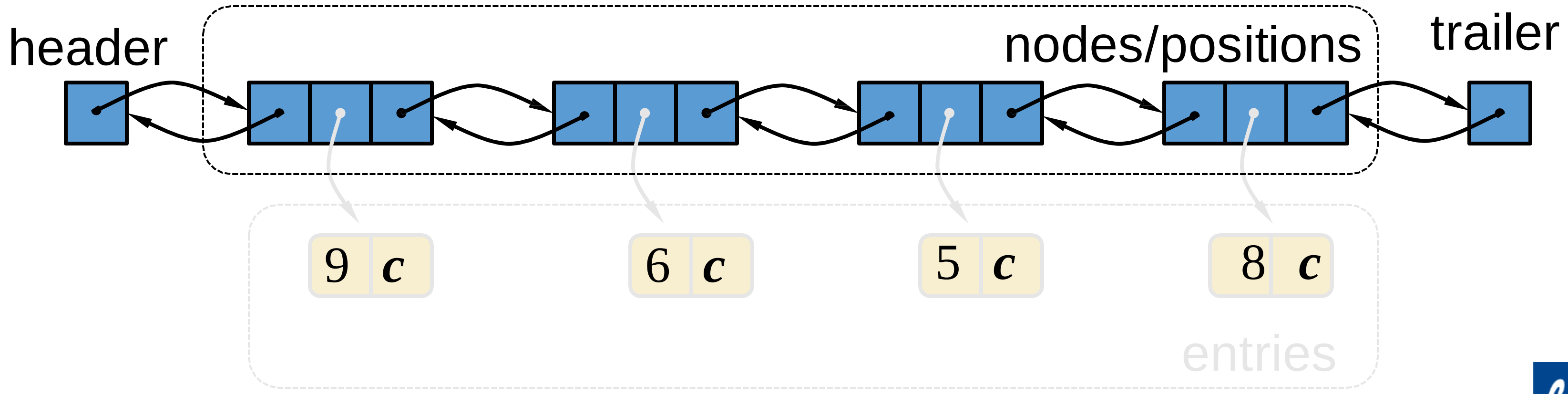
Map implementation

- Arrays
- A simple linked list of pairs
  - Slow (O(n)),
  - insufficient for general use.
- A hash table.
  - This is generally very fast (roughly O(1)),
  - Requires a good hash function for the key type.
- A binary search tree.
  - Fast (O(lg n)).
  - Unlike in a hash table, the keys will be ordered.
- A skip list.

Reading slide:A Simple List-Based Map

We can efficiently implement a map using an unsorted list

- We store the items of the map in a list S (based on a doubly-linked list), in arbitrary order



Reading slide: Hash-Based Map implementation

- Hash Map uses a hash table as its internal storage container.
- Keys stored based on hash codes and size of hash tables internal array

Reading slide: Tree-Based Map implementation

- Uses Height Balanced Binary Search Trees
- In java a Red - Black tree is used to implement a Map
- Somewhat slower than the HashMap

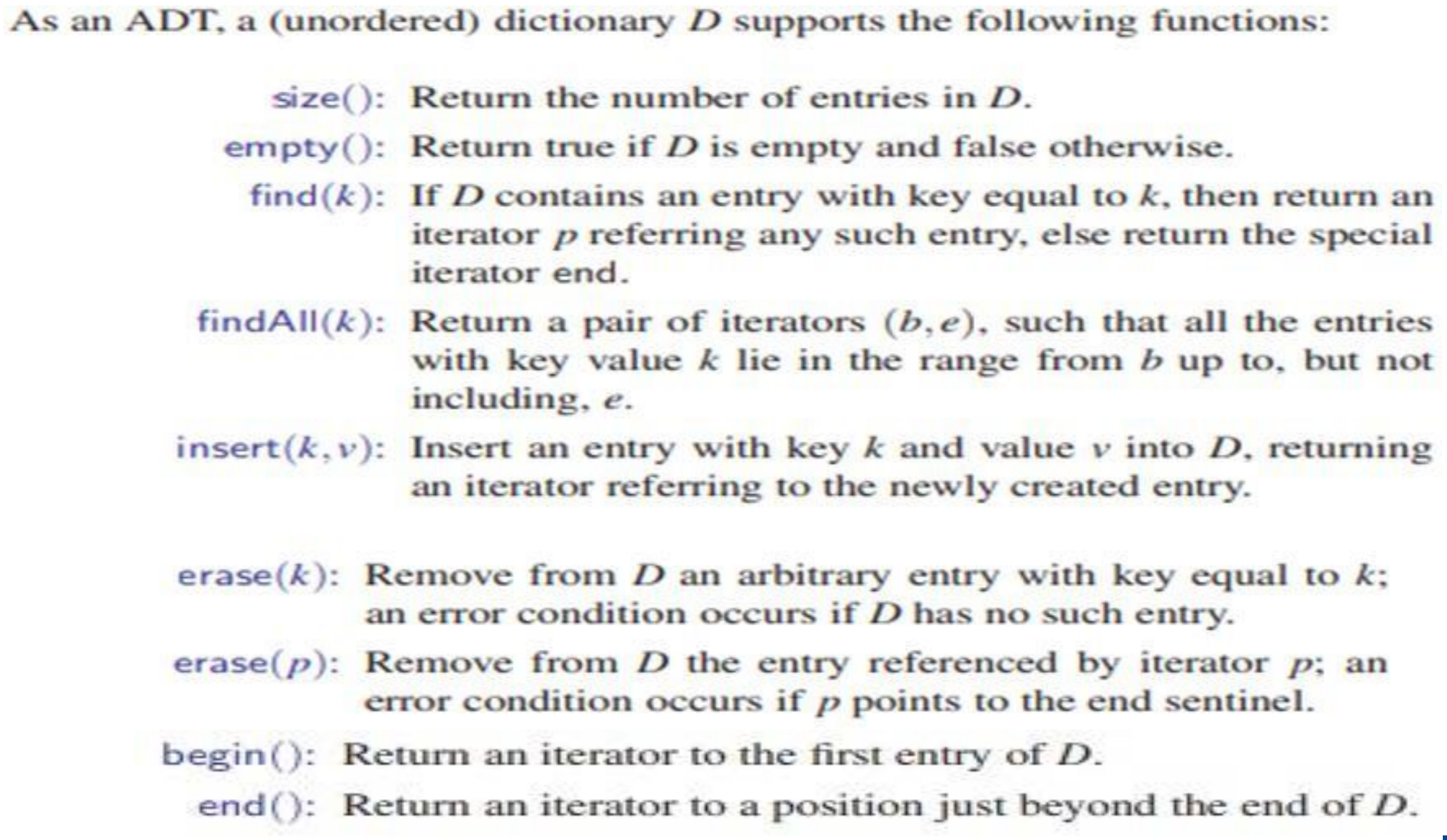
Dictionary

- A dictionary allows for keys and values to be of any object type.
- Unlike Map, a dictionary allows for multiple entries to have the same key
- For example
  - an English dictionary, which allows for multiple definitions for the same word.
  - we might want to store records for computer science authors indexed by their first and last names.
  - a multi-user computer game involving players visiting various rooms in a large castle might need a mapping from rooms to players. It is natural in this application to allow users to be in the same room simultaneously, however, to engage in battles.

Reading assignment:

Similarities and differences in set, map and dictionary

The Dictionary ADT

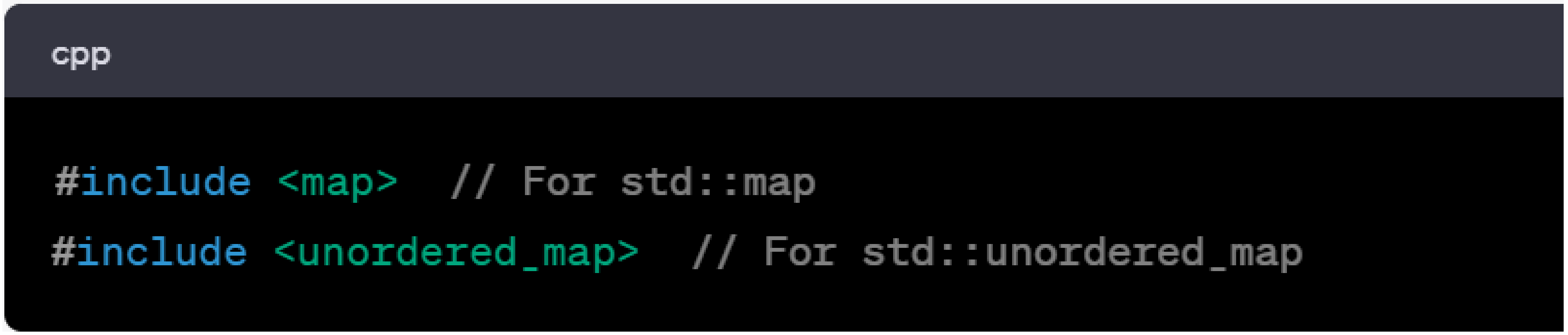


As an ADT, a (unordered) dictionary D supports the following functions:

size():	Return the number of entries in D.
empty():	Return true if D is empty and false otherwise.
find(k):	If D contains an entry with key equal to k, then return an iterator p referring any such entry, else return the special iterator end.
findAll(k):	Return a pair of iterators (b,e), such that all the entries with key value k lie in the range from b up to, but not including, e.
insert(k,v):	Insert an entry with key k and value v into D, returning an iterator referring to the newly created entry.
erase(k):	Remove from D an arbitrary entry with key equal to k; an error condition occurs if D has no such entry.
erase(p):	Remove from D the entry referenced by iterator p; an error condition occurs if p points to the end sentinel.
begin():	Return an iterator to the first entry of D.
end():	Return an iterator to a position just beyond the end of D.

Using dictionary data structure

- In C++, the std::map and std::unordered_map data structures are commonly used to implement a dictionary-like data structure.
- These containers allow you to store key-value pairs, making them useful for various tasks



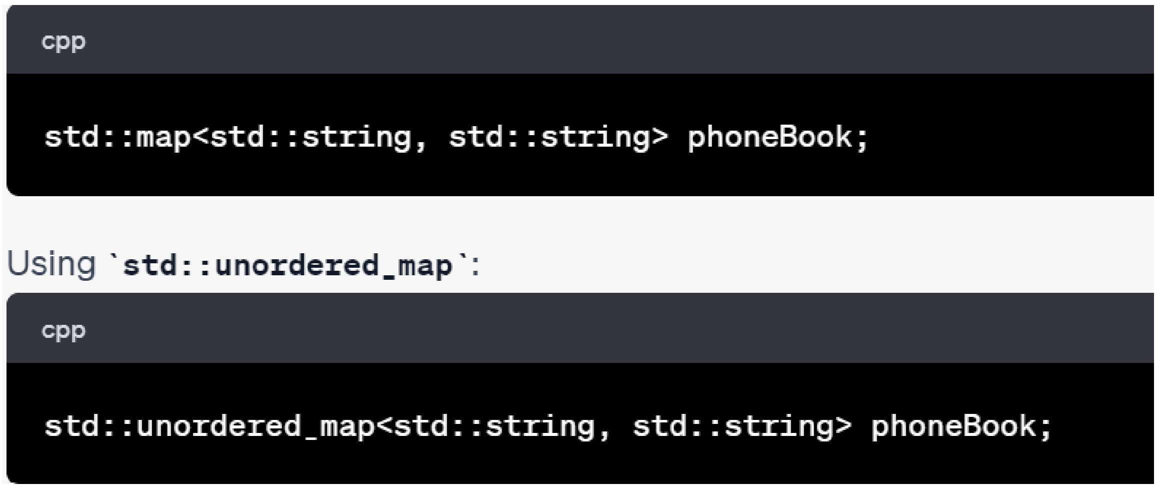
cpp

```cpp
#include <map>  // For std::map
#include <unordered_map>  // For std::unordered_map
```

The difference in map and unordered map :

- Primary difference is in the performance characteristics
- std::map is a sorted associative container that uses a red-black tree.
  - efficient for searching, insertion, and deletion operations with a time complexity of O(log n).
- std::unordered_map is a hash-based associative container.
  - offers faster average constant-time access for searching, insertion, and deletion (with a good hash function).

IMPLEMENTING DICTIONARY



cpp

```cpp
std::map<std::string, std::string> phoneBook;
```

Using `std::unordered_map`:

cpp

```cpp
std::unordered_map<std::string, std::string> phoneBook;
```

- Adding key value pairs:
  - phoneBook["Alice"] = "123-456-7890";
  - phoneBook["Bob"] = "987-654-3210";
    phoneBook.**insert**(std::make_pair("Charlie", "555-123-4567"));
- Access values using keys:
  - Std::string aliceNumber = phoneBook["Alice"];
- Find()

```
if (phoneBook.find("Bob") != phoneBook.end()) {
std::string bobNumber = phoneBook["Bob"];
}
```

- Remove items

```
phoneBook.erase("Alice");
```

- Count()

```
if (phoneBook.count("Bob") > 0) {
std::string bobNumber = phoneBook["Bob"];
 }
```

Using std::multimap (Ordered Dictionary):

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

int main() {
    std::multimap<std::string, int> orderedDictionary;

    // Inserting key-value pairs
    orderedDictionary.insert(std::make_pair("apple", 5));
    orderedDictionary.insert(std::make_pair("banana", 3));
    orderedDictionary.insert(std::make_pair("cherry", 7));
    orderedDictionary.insert(std::make_pair("banana", 8)); // Allowing duplicate key

    // Accessing and printing values for a specific key
    std::string keyToFind = "banana";
    auto range = orderedDictionary.equal_range(keyToFind);

    std::cout << "All values for key '" << keyToFind << "': ";
    for (auto it = range.first; it != range.second; ++it) {
        std::cout << it->second << " ";
    }
    std::cout << std::endl;

    // Iterating through the ordered dictionary
    std::cout << "Ordered Dictionary Contents: " << std::endl;
    for (const auto& pair : orderedDictionary) {
        std::cout << pair.first << " => " << pair.second << std::endl;
    }

    return 0;
}
```

Using std::unordered_multimap (Unordered Dictionary):

std::unordered_map is an unordered dictionary. It does not guarantee a specific order of the elements, but it provides faster access.

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

int main() {
    std::unordered_multimap<std::string, int> unorderedDictionary;

    // Inserting key-value pairs
    unorderedDictionary.insert(std::make_pair("apple", 5));
    unorderedDictionary.insert(std::make_pair("banana", 3));
    unorderedDictionary.insert(std::make_pair("cherry", 7));
    unorderedDictionary.insert(std::make_pair("banana", 8)); // Allowing duplicate key

    // Accessing and printing values for a specific key
    std::string keyToFind = "banana";
    auto range = unorderedDictionary.equal_range(keyToFind);

    std::cout << "All values for key '" << keyToFind << "': ";
    for (auto it = range.first; it != range.second; ++it) {
        std::cout << it->second << " ";
    }
    std::cout << std::endl;

    // Iterating through the unordered dictionary
    std::cout << "Unordered Dictionary Contents: " << std::endl;
    for (const auto& pair : unorderedDictionary) {
        std::cout << pair.first << " => " << pair.second << std::endl;
    }

    return 0;
}
```

Output:

```
All values for key 'banana': 3 8
Unordered Dictionary Contents:
banana => 3
banana => 8
cherry => 7
apple => 5
```

Dictionary Implementations

- **Unordered list:** In an unordered list, L, implementing a dictionary, we can maintain the location variable of each entry e to point to e's position in the underlying linked list for L.
- **Hash table with separate chaining:** Consider a hash table, with bucket array A and hash function h, that uses separate chaining for handling collisions. We use the location variable of each entry e to point to e's position in the list L implementing the list A[h(k)].
- **Ordered search table:** In an ordered table, T, implementing a dictionary, we should maintain the location variable of each entry e to be e's index in T.

Example

Problem statement:
Given names and phone numbers, assemble a phone book that maps friends' names to their respective phone numbers. You will then be given an unknown number of names to query your phone book for. For each query, print the associated entry from your phone book on a new line in the form name=phoneNumber; if an entry for is not found, print Not found instead.

Example.. contd

Input Format
The first line contains an integer, , denoting the number of entries in the phone book.
Each of the subsequent lines describes an entry in the form of space-separated values on a single line. The first value is a friend's name, and the second value is an -digit phone number.
After the lines of phone book entries, there are *an unknown number of lines of queries*. Each line (query) contains a name to look up, and you must continue reading lines until there is no more input.
Output Format
On a new line for each query, print Not found if the name has no corresponding entry in the phone book; otherwise, print the full and in the format name=phoneNumber.

Solution?

- Sample dictionary entries?
- Sample queries?
- Output?

Example 2

You have a weather forecast data having temperature details of few cities for few days for the year 2018

Build data structure to answer the following queries

- What is Temperature in Delhi on 9-11-2018
- What is max temperature recorded in Chennai in 2018
- Displaying the number of entries of 2018
- Deleting the entry of Mumbai on 9-11-2018
- Deleting all entries of Mumbai

City	Date	Temperature
Delhi	9-11-2018	45
Bangalore	9-11-2018	24
Ranchi	9-12-2018	28
Chennai	9-01-2018	38

Queries?

Thank you