

C++ Data Structures

Prerequisites

CHAPTER 5: C++ STL LIBRARY

DR. ERIC CHOU

IEEE SENIOR MEMBER

Objectives

- C++ Data Collections: pair/tuple, list, set, map (Like Python)
- Associative Containers

LECTURE 1

Programming Paradigms

Abstraction of C++ Languages

- C++ is not a functional programming language. C++ has its roots in procedural and object-oriented programming.
- So, it's quite surprising that programming in a functional style becomes more and more important in C++. That is not only true for C++. That holds also for python, which has a lot of functional features and even for Java. Now Java has lambda functions.

C++11

- Functional

C++

- Generic
- Object-Oriented

C

- Procedural
- Structured

Object-oriented programming

- Object-oriented programming is based on the three concepts encapsulation, inheritance, and polymorphism.

Encapsulation

- An object encapsulates its attributes and methods and provides them via an interface to the outside world. This property that an object hides its implementation is often called *data hiding*.

Inheritance

- A derived class get all characteristics from its base class. You can use an instance of a derived class as an instance of its base class. We often speak about *code reuse* because the derived class automatically gets all characteristics of the base class.

Polymorphism

- Polymorphism is the ability to present the same interface for differing underlying data types. The term is from Greek and stands for many forms.

Object-Oriented Programming

```
1  class HumanBeing{
2  public:
3      HumanBeing(std::string n):name(n){}
4      virtual std::string getName() const{
5          return name;
6      }
7  private:
8      std::string name;
9  };
10
11  class Man: public HumanBeing{};
12
13  class Woman: public HumanBeing{};
```

- In the example, you only get the name of HumanBeing by using the method getName in line 4 (encapsulation).
- In addition, getName is declared as virtual. Therefore, derived classes can change the behaviour of their methods and therefore change the behaviour of their objects (polymorphism).
- Man and Woman are derived from HumanBeing.

Generic Programming

```
1  template <typename T> void xchg(T& x, T& y){
2      T t= x;
3      x= y;
4      y= t;
5  };
6  int i= 10;
7  int j= 20;
8  Man huber;
9  Man maier;
10
11 xchg(i,j);
12 xchg(huber,maier);
```

Template Function (Generic Function)

- The key idea of **generic programming** or programming with templates is to define families of functions or classes. By providing the concrete type you get automatically a function or a class for this type. Generic programming provides a similar abstraction to object-oriented programming. A big difference is that polymorphism of object-oriented programming will happen at runtime; that polymorphism of generic programming will happen in C++ at compile time. That the reason why polymorphism at runtime is often called dynamic polymorphism but polymorphism at compile is often called static polymorphism.
- By using the function template, I can exchange arbitrary objects.

Generic Programming

```
1  template <typename T, int N>
2  class Array{
3  public:
4      int getSize() const{
5          return N;
6      }
7  private:
8      T elem[N];
9  };
10
11  Array<double,10> doubleArray;
12  std::cout << doubleArray.getSize() << std::endl;
13
14  Array<Man,5> manArray;
15  std::cout << manArray.getSize() << std::endl;
```

Template Class (Generic Container)

- It doesn't matter for the function template if I exchange numbers or men (line 11 and 12). In addition, I have not to specify the type parameter (line) because the compiler can derive it from the function arguments (line 11 and 12).
- The automatic type deduction of function templates will not hold for class templates. In the concrete case, I have to specify the type parameter T and the non-type parameter N (line 1).
- Accordingly, the application of the class template Array is independent of the fact, whether I use doubles or men.

Functional programming

```
1  std::vector<int> vec{1,2,3,4,5,6,7,8,9};
2  std::vector<std::string> str{"Programming","in","a","functional","style."};
3
4  std::transform(vec.begin(),vec.end(),vec.begin(),
5                [](int i){ return i*i; }); // {1,4,9,16,25,36,49,64,81}
6
7  auto it= std::remove_if(vec.begin(),vec.end(),
8                          [](int i){ return ((i < 3) or (i > 8)) }); // {3,4,5,6,7,8}
9  auto it2= std::remove_if(str.begin(),str.end(),
10                        [](string s){ return (std::lower(s[0])); }); // "Programming"
11
12
13  std::accumulate(vec.begin(),vec.end(),[](int a,int b){return a*b;}); // 362880
14  std::accumulate(str.begin(),str.end(),
15                  [](std::string a,std::string b){return a + ":" + b;});
16  // "Programming:in:a:functional:style."
```

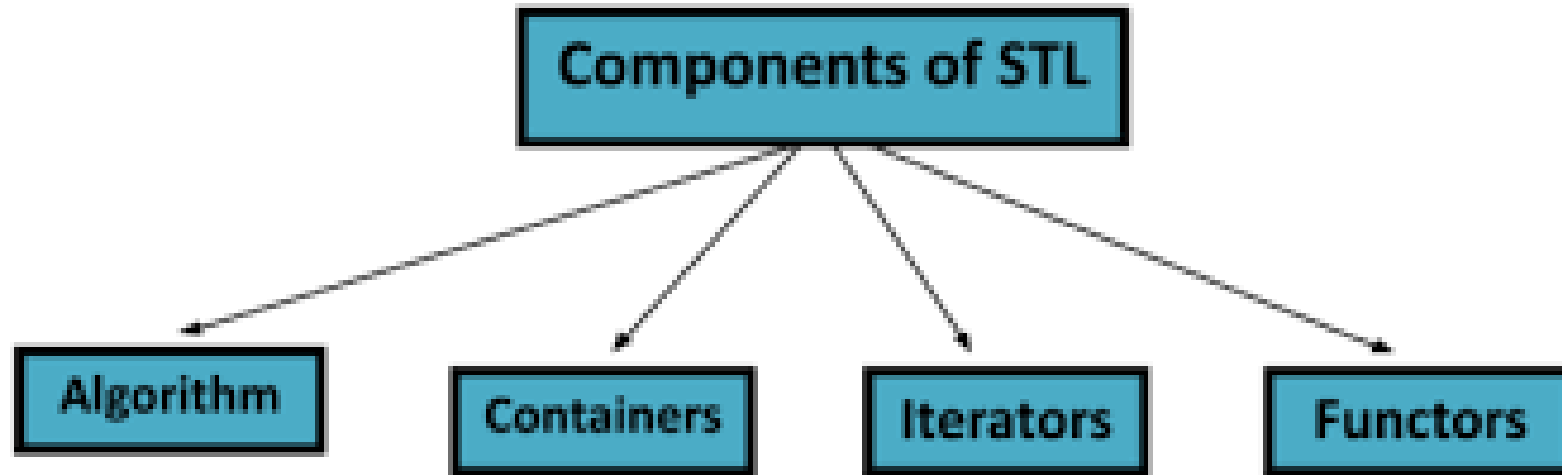
Functional Closure / Lambda Expression

Functional programming

- I will only say a few words about functional programming because I will and can not explain the concept of functional programming in a short remark. Only that much. I use the code snippet of the pendants in C++ to the typical functions in functional programming: map, filter, and reduce. These are the functions **std::transform**, **std::remove_if**, and **std::accumulate**.
- I apply in the code snippet two powerful features of functional programming. Both are now mainstream in modern **C++**: automatic type deduction with **auto** and **lambda** functions.

LECTURE 2

Generic Programming



Standard Template Library in C++

Container

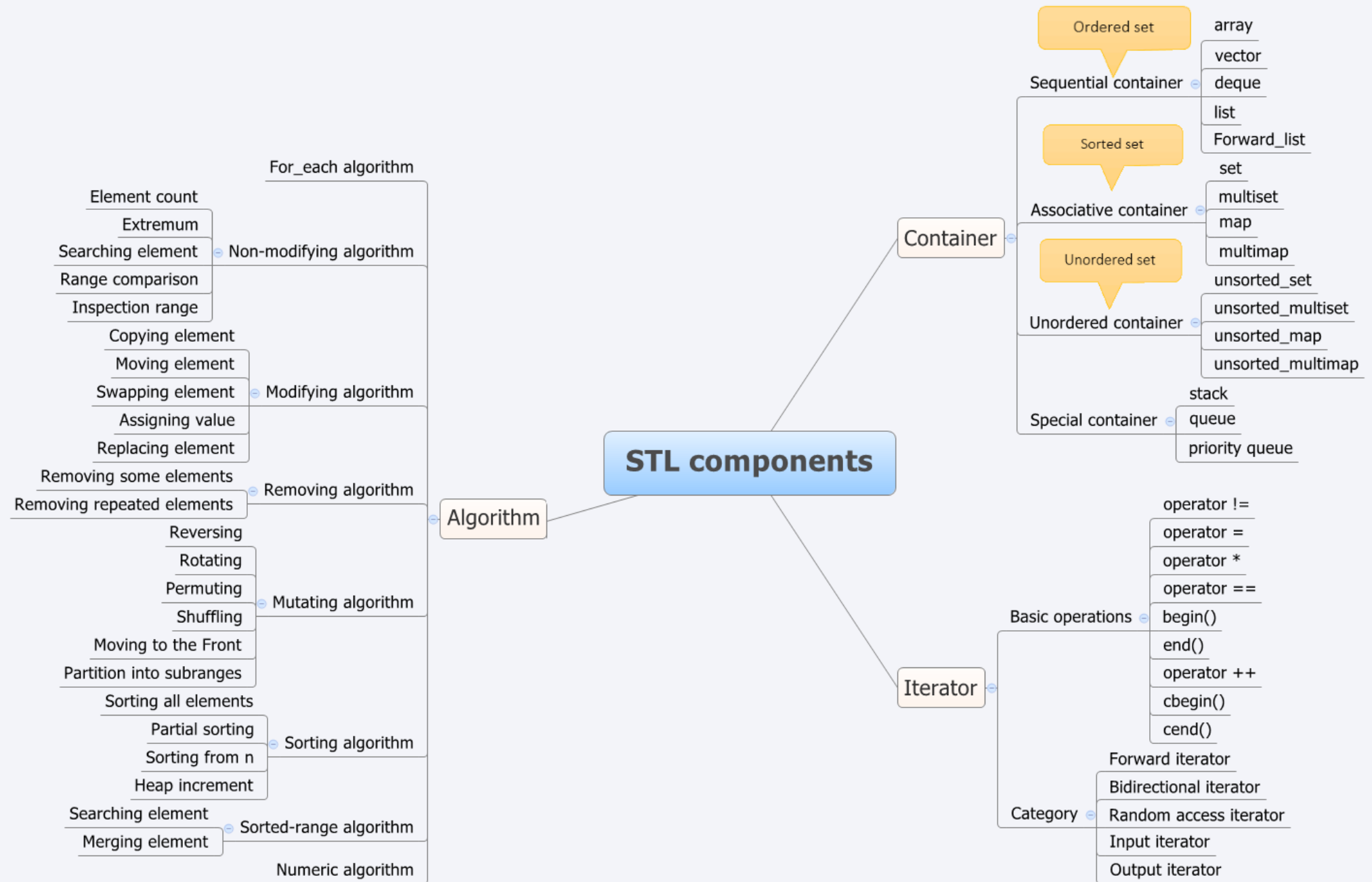
- Sequence Containers
- Associative Containers
- Container Adapters
- Unordered Associative Containers

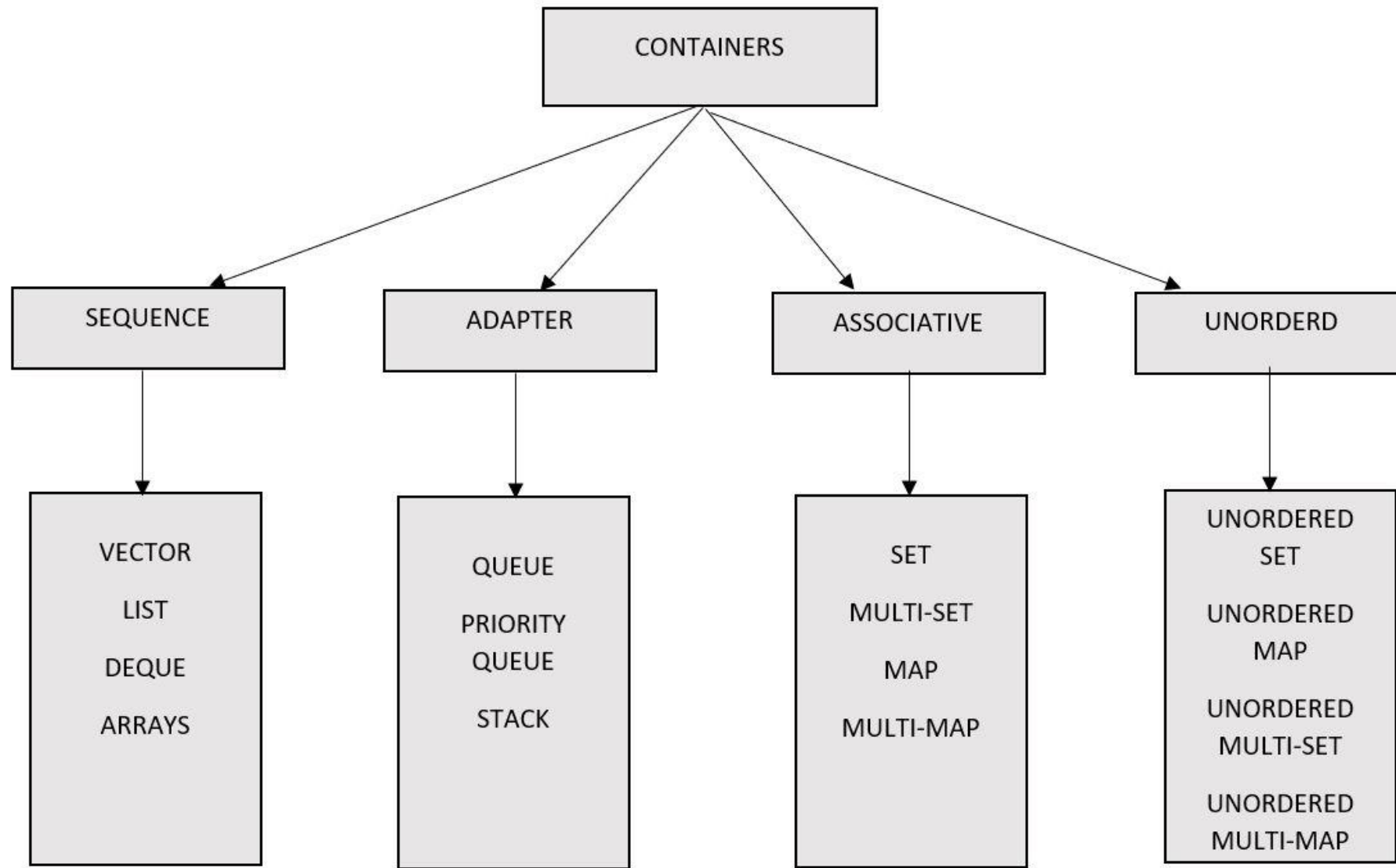
Iterator

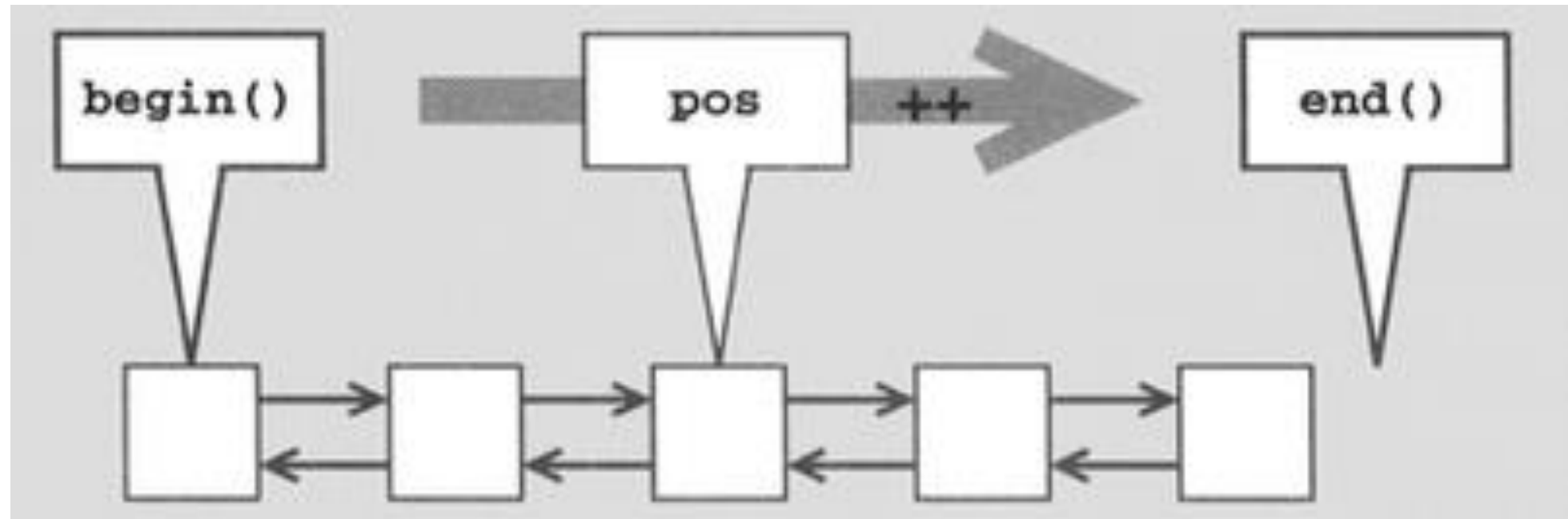
- begin()
- next()
- prev()
- advance()
- end()

Algorithm

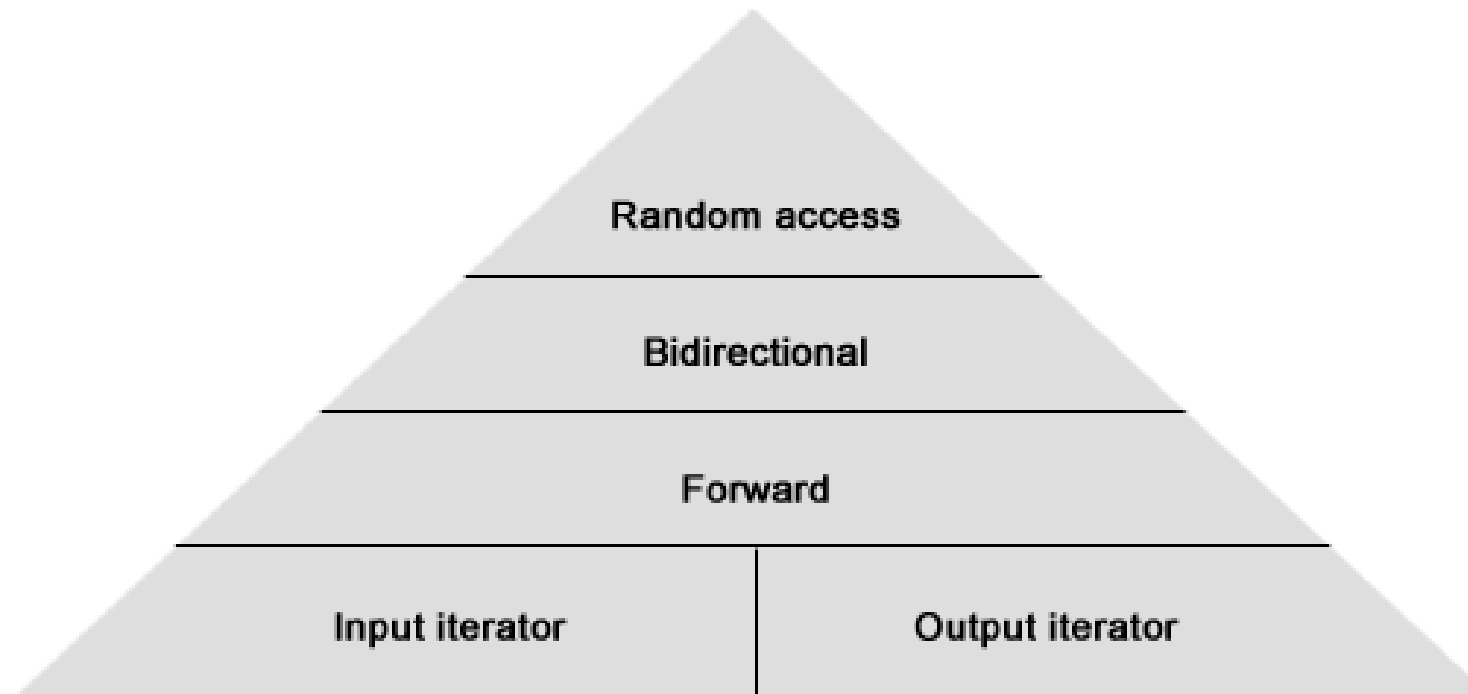
- Sorting Algorithms
- Search algorithms
- Non modifying algorithms
- Modifying algorithms
- Numeric algorithms
- Minimum and Maximum operations







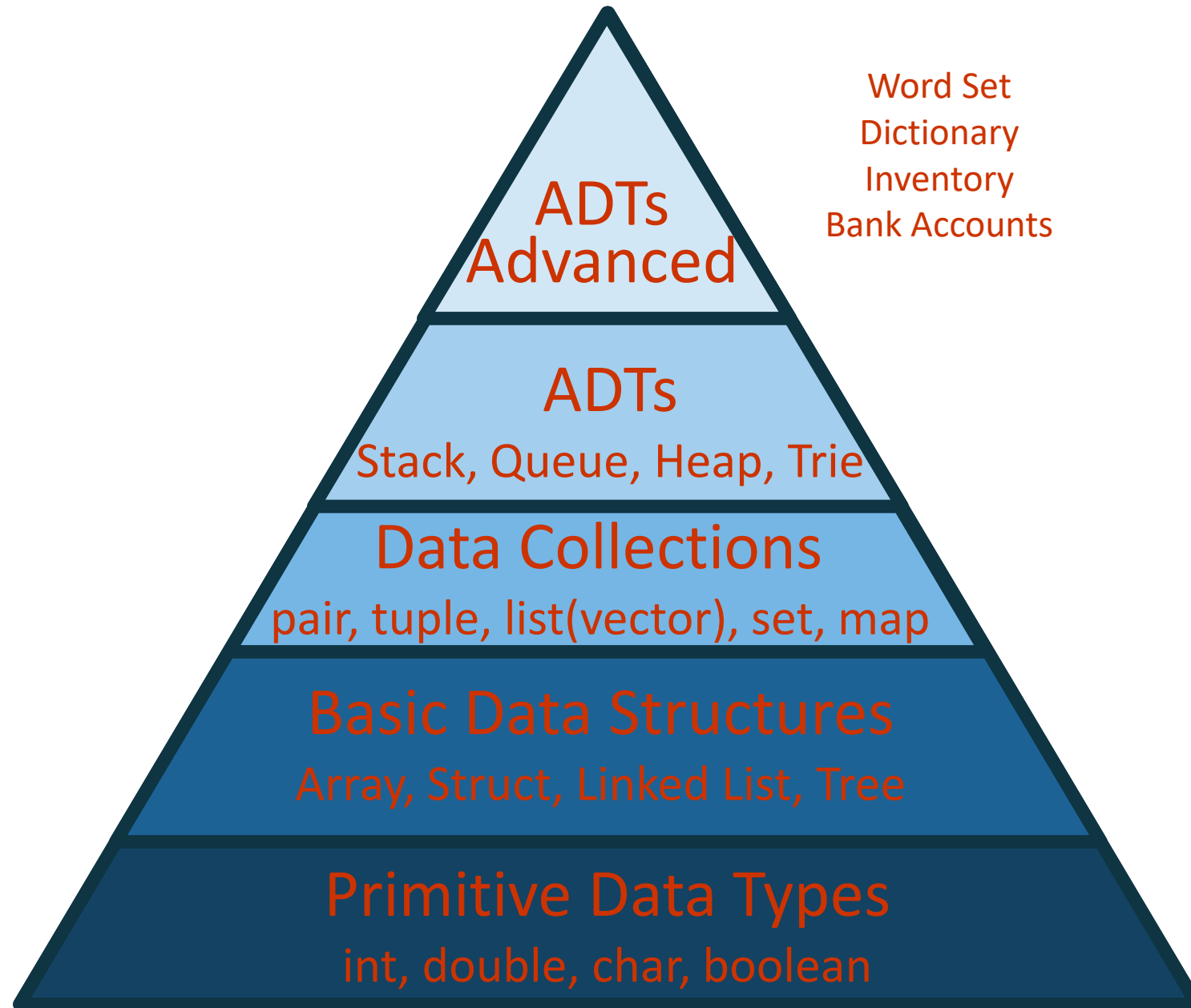
Iterator categories



Iterator categories	Provider
Input iterator	istream
Output iterator	ostream
Forward iterator	
Bidirectional iterator	list, set, multiset, map, multimap
Random access iterator	vector, deque, array

LECTURE 3

Data Structures, Data Collections, and Abstract Data Types



LECTURE 4

Move Constructor

Move Constructor

- Move a data container from one instance to another.
- A move constructor allows the resources owned by an **rvalue** object to be moved into an **lvalue** without creating its copy.
- An **rvalue** is an expression that does not have any memory address, and an **lvalue** is an expression with a memory address.

What is an Rvalue reference?

RHS value, LHS variable

Before jumping on to the rvalue reference, let's see one of the limitations in C++:

```
int &j = 20;
```

The above code snippet will give an error because, in C++, a variable cannot be referenced to a temporary object. The correct way to do it is to create an rvalue reference using && :

```
int &&j = 20;
```

Temporary objects are often created during the execution of a C++ program.

Copy Constructor

- Make duplicated copy.
- Now, when the code is executed, the default constructor is called at the time that the temporary object A is created. The copy constructor is called as the temporary object of A is pushed back in the vector.
- In the above code, there is a serious performance overhead as the temporary object A is first created in the default constructor and then in the copy constructor.

copy2.cpp

```
#include <iostream>
#include <vector>
using namespace std;
class A{
    int *ptr;
public:
    A(){
        // Default constructor
        cout << "Calling Default constructor\n";
        ptr = new int ;
    }
    A( const A & obj){
        // Copy Constructor: copy of object is created
        this->ptr = new int;
        // Deep copying
        cout << "Calling Copy constructor\n";
    }
    ~A(){
        // Destructor
        cout << "Calling Destructor\n";
        delete ptr;
    }
};

int main() {
    vector <A> vec;
    vec.push_back(A()); // make copy here. Deep Copy
    return 0;
}
```

Calling Default constructor

Calling Copy constructor

Calling Destructor

Calling Destructor

Move Constructor

- Do not copy. Only copy reference. Shallow Copy
- When the code is executed, the move constructor is called instead of the copy constructor. With the move constructor, the copy of the temporary object of A is avoided. For a large number of push_back statements, using the move constructor is an efficient choice.
- Note: a move constructor can be explicitly called, with the move() function, for already created objects.

```

class A{
    int *ptr;
public:
    A(){
        // Default constructor
        cout << "Calling Default constructor\n";
        ptr = new int ;
    }
    A( const A & obj){
        // Copy Constructor: copy of object is created
        this->ptr = new int;
        // Deep copying
        cout << "Calling Copy constructor\n";
    }
    A ( A && obj){
        // Move constructor: It will simply shift the resources, without creating a copy.
        cout << "Calling Move constructor\n";
        this->ptr = obj.ptr;
        obj.ptr = NULL;
    }

    ~A(){
        // Destructor
        cout << "Calling Destructor\n";
        delete ptr;
    }
};

```

move2.cpp

```

int main() {
    vector <A> vec;
    vec.push_back(A());
    return 0;
}

```

Calling Default constructor
 Calling Move constructor
 Calling Destructor
 Calling Destructor

LECTURE 5

pair

Pair in C++ Standard Template Library (STL)

The pair container is a simple container defined in **<utility>** header consisting of two data elements or objects.

- The first element is referenced as 'first' and the second element as 'second' and the order is fixed (first, second).
- Pair is used to combine together two values which may be different in type. Pair provides a way to store two heterogeneous objects as a single unit.
- Pair can be assigned, copied and compared. The array of objects allocated in a map or hash_map are of type 'pair' by default in which all the 'first' elements are unique keys associated with their 'second' value objects.
- To access the elements, we use variable name followed by dot operator followed by the keyword first or second.

Pair Syntax

The general syntax is:

```
pair<T1, T2> pair1, pair2;
```

- The above line of code creates two pairs i.e. pair1 and pair2. Both these pairs have the first object of type T1 and the second object of type T2.
- T1 is the first member and T2 is the second member of pair1 and pair2.

Methods

- **Operator (=)**: Assign values to a pair.
- **swap**: Swaps the contents of the pair.
- **make_pair()**: Create and returns a pair having objects defined by the parameter list.
- **Operators(== , != , > , < , <= , >=)** : Compares two pairs lexicographically.

pair.cpp

```
#include <iostream>
using namespace std;
int main (){
    pair<int,int> pair1, pair3;
    pair<int,string> pair2;

    pair1 = make_pair(1, 2);
    pair2 = make_pair(1, "SoftwareTestingHelp");
    pair3 = make_pair(2, 4);
    cout<< "\nPair1 First member: "<<pair1.first << endl;
    cout<< "\nPair2 Second member:"<<pair2.second << endl;
    if(pair1 == pair3)
        cout<< "Pairs are equal" << endl;
    else
        cout<< "Pairs are not equal" << endl;

    return 0;
}
```

Pair1 First member: 1

Pair2 Second member:SoftwareTestingHelp
Pairs are not equal

LECTURE 6

tuple

Tuples in C++

What is a tuple?

- A tuple is an object that can hold a **number** of elements. The elements can be of different data types. The elements of tuples are initialized as arguments in order in which they will be accessed.
- Similar to Python Tuple

Operations on tuple

- 1. get() :-** get() is used to access the tuple values and modify them, it accepts the index and tuple name as arguments to access a particular tuple element. **get()** function will get LHS value.
- 2. make_tuple() :-** make_tuple() is used to assign tuple with values. The values passed should be in order with the values declared in tuple.

tuple1.cpp

```
// C++ code to demonstrate tuple, get() and make_pair()
#include<iostream>
#include<tuple> // for tuple
using namespace std;
int main() {
    // Declaring tuple
    tuple <char, int, float> geek;
    // Assigning values to tuple using make_tuple()
    geek = make_tuple('a', 10, 15.5);
    // Printing initial tuple values using get()
    cout << "The initial values of tuple are : ";
    cout << get<0>(geek) << " " << get<1>(geek);
    cout << " " << get<2>(geek) << endl;
    // Use of get() to change values of tuple
    get<0>(geek) = 'b';
    get<2>(geek) = 20.5;
    // Printing modified tuple values
    cout << "The modified values of tuple are : ";
    cout << get<0>(geek) << " " << get<1>(geek);
    cout << " " << get<2>(geek) << endl;
    return 0;
}
```

The initial values of tuple are : a 10 15.5

The modified values of tuple are : b 10 20.5

Operations on tuple

- 3. **tuple_size** :- It returns the number of elements present in the tuple.
- 4. **swap()** :- The swap(), swaps the elements of the two different tuples.
- 5. **tie()** :- The work of tie() is to unpack the tuple values into separate variables. There are two variants of tie(), with and without “ignore” , the “ignore” ignores a particular tuple element and stops it from getting unpacked.
- 6. **tuple_cat()** :- This function concatenates two tuples and returns a new tuple.

tuple3.cpp

The size of tuple is : 3

```
//C++ code to demonstrate tuple_size
#include<iostream>
#include<tuple> // for tuple_size and tuple
using namespace std;
int main(){

    // Initializing tuple
    tuple <char,int,float> geek(20,'g',17.5);

    // Use of size to find tuple_size of tuple
    cout << "The size of tuple is : ";
    cout << tuple_size<decltype(geek)>::value << endl;

    return 0;

}
```

tuple4.cpp

```
//C++ code to demonstrate swap()
#include<iostream>
#include<tuple> // for swap() and tuple
using namespace std;
int main(){
    // Initializing 1st tuple
    tuple <int,char,float> tup1(20,'g',17.5);
    // Initializing 2nd tuple
    tuple <int,char,float> tup2(10,'f',15.5);
    // Printing 1st and 2nd tuple before swapping
    cout << "The first tuple elements before swapping are : ";
    cout << get<0>(tup1) << " " << get<1>(tup1) << " "
        << get<2>(tup1) << endl;
    cout << "The second tuple elements before swapping are : ";
    cout << get<0>(tup2) << " " << get<1>(tup2) << " " << get<2>(tup2) << endl;
    // Swapping tup1 values with tup2
    tup1.swap(tup2);
    // Printing 1st and 2nd tuple after swapping
    cout << "The first tuple elements after swapping are : ";
    cout << get<0>(tup1) << " " << get<1>(tup1) << " " << get<2>(tup1) << endl;
    cout << "The second tuple elements after swapping are : ";
    cout << get<0>(tup2) << " " << get<1>(tup2) << " " << get<2>(tup2) << endl;
    return 0;
}
```

The first tuple elements before swapping are : 20 g 17.5
The second tuple elements before swapping are : 10 f 15.5
The first tuple elements after swapping are : 10 f 15.5
The second tuple elements after swapping are : 20 g 17.5

tuple5.cpp

The unpacked tuple values (without ignore) are : 20 g 17.5

The unpacked tuple values (with ignore) are : 20 17.5

```
// C++ code to demonstrate working of tie()
#include<iostream>
#include<tuple> // for tie() and tuple
using namespace std;
int main(){
    // Initializing variables for unpacking
    int i_val;
    char ch_val;
    float f_val;

    // Initializing tuple
    tuple<int,char,float> tup1(20,'g',17.5);

    // Use of tie() without ignore
    tie(i_val,ch_val,f_val) = tup1;

    // Displaying unpacked tuple elements without ignore
    cout << "The unpacked tuple values (without ignore) are : ";
    cout << i_val << " " << ch_val << " " << f_val;
    cout << endl;
    // Use of tie() with ignore ignores char value
    tie(i_val,ignore,f_val) = tup1;

    // Displaying unpacked tuple elements with ignore
    cout << "The unpacked tuple values (with ignore) are : ";
    cout << i_val << " " << f_val;
    cout << endl;
    return 0;
}
```



```
// C++ code to demonstrate working of tuple_cat()
#include<iostream>
#include<tuple> // for tuple_cat() and tuple
using namespace std;
int main(){
    // Initializing 1st tuple
    tuple <int,char,float> tup1(20,'g',17.5);

    // Initializing 2nd tuple
    tuple <int,char,float> tup2(30,'f',10.5);

    // Concatenating 2 tuples to return a new tuple
    auto tup3 = tuple_cat(tup1,tup2);

    // Displaying new tuple elements
    cout << "The new tuple elements in order are : ";
    cout << get<0>(tup3) << " " << get<1>(tup3) << " ";
    cout << get<2>(tup3) << " " << get<3>(tup3) << " ";
    cout << get<4>(tup3) << " " << get<5>(tup3) << endl;
    return 0;
}
```

tuple6.cpp

The new tuple elements in order are : 20 g 17.5 30 f 10.5

Sorting on Vector of Tuples

- Load parallel data vectors on to a vector of tuples.
- Design a comparator function (callable)
- Run Sort algorithm in STL `<algorithm>`

```
#include <iostream>
#include <string>
#include <vector>
#include <tuple>
#include <algorithm>
using namespace std;
```

```
typedef tuple<string,double,int> mytuple;
```

```
bool mycompare (const mytuple &lhs, const mytuple &rhs){
    return get<1>(lhs) < get<1>(rhs);
}
```

```
int main(void){
    vector<mytuple> data;
    data.push_back(make_tuple("abc",4.5,1));
    data.push_back(make_tuple("def",5.5,-1));
    data.push_back(make_tuple("wolf",-3.47,1));
    sort(data.begin(),data.end(),mycompare);
    for(vector<mytuple>::iterator iter = data.begin(); iter != data.end(); iter++){
        cout << get<0>(*iter) << "\t" << get<1>(*iter) << "\t" << get<2>(*iter) << endl;
    }
}
```

tuple_sort.cpp

```
wolf -3.47    1
abc  4.5    1
def  5.5   -1
```

LECTURE 7

list

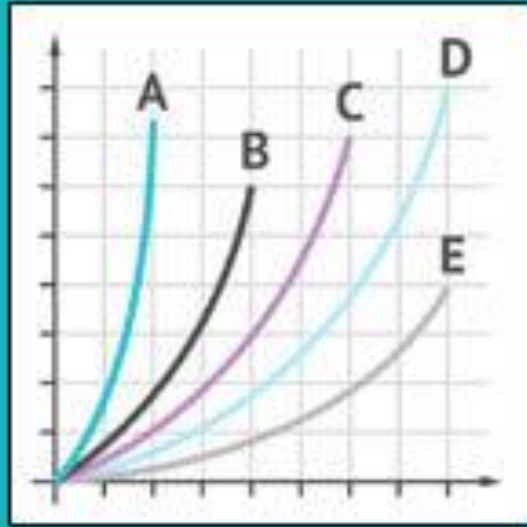
What is an `std::list`?

- In C++, the `std::list` refers to a storage container. The **`std::list`** allows you to insert and remove items from anywhere. The `std::list` is implemented as a doubly-linked list. This means list data can be accessed bi-directionally and sequentially.
- The Standard Template Library **list doesn't support fast random access**, but it supports sequential access from all directions.
- You can scatter list elements in different memory chunks. The information needed for sequential access to data is stored in a container. The `std::list` can expand and shrink from both ends as needed during runtime. An internal allocator automatically fulfills the storage requirements.

Why use std::list?

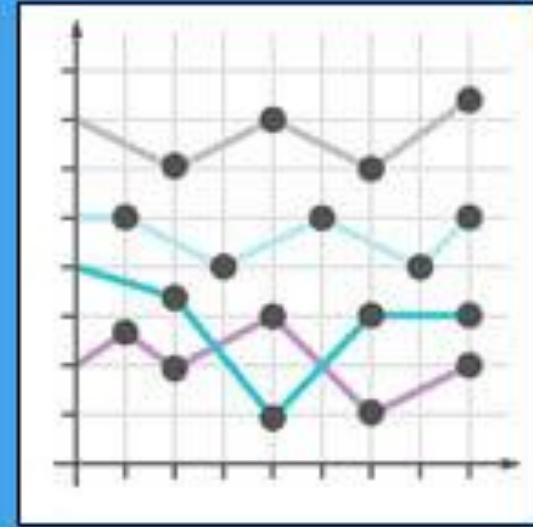
Here, are reason of using std::List :

- The **std::list** does better compare to other sequence containers like array and vector.
- They have a **better performance** in inserting, moving, and extracting elements from any position.
- The **std::list** also does better with algorithms that perform such operations intensively.




C++ vector

VS



C++ list

www.educba.com

	Vector	List
Memory	Continuous Pre-allocation for future usage	Non-continuous Allocate when use 
Extra space	No need, only element itself	Need extra space to point previous element and next element (linked list)
Insertion/ Erase	$O(1)$ if <code>push_back()</code> / <code>pop_back()</code> $O(n)$ if <code>insert()</code> / <code>erase()</code> somewhere else	$O(1)$

List Syntax

- To define **the std::list**, we have to import the <list> header file. Here is the **std::list** definition syntax:

```
template <class Type, class Alloc=allocator<T>> class list;
```

Here is a description of the above parameters:

- **T** – Defines the type of element contained.
 - You can substitute T by any data type, even user-defined types.
- **Alloc** – Defines the type of the allocator object.
 - This uses the allocator class template by default. It's value-dependent and uses a simple memory allocation model.

C++ List Functions

Function	Description
insert()	This function inserts a new item before the position the iterator points.
push_back()	This functions add a new item at the list's end.
push_front()	It adds a new item at the list's front.
pop_front()	It deletes the list's first item.
size()	This function determines the number of list elements.
front()	To determines the list's first items.
back()	To determines the list's last item.
reverse()	It reverses the list items.
merge()	It merges two sorted lists.

list.cpp

```
#include <algorithm>
#include <iostream>
#include <list>
int main() {
    std::list<int> my_list = { 12, 5, 10, 9 };

    for (int x : my_list) {
        std::cout << x << '\n';
    }
}
```

12

5

10

9

<list> Constructors

Here is the list of functions provided by the <list> header file:

- Default constructor `std::list::list()`- It creates an empty list, that, with zero elements.
- Fill constructor `std::list::list()`- It creates a list with n elements and assigns a value of zero (0) to each element.
- Range constructor `std::list::list()`- creates a list with many elements in the range of first to last.
- Copy constructor `std::list::list()`- It creates a list with a copy of each element contained in the existing list.
- Move constructor `std::list::list()`- creates a list with the elements of another list using move semantics.
- Initializer list constructor `std::list::list()`-It creates a list with the elements of another list using move semantics.

```
#include <iostream>
#include <list>
using namespace std;
int main(void) {
    list<int> l1;
    list<int> l1 = { 10, 20, 30 };
    list<int> l2(l1.begin(), l1.end());
    list<int> l3(move(l1));
    cout << "Size of list l: " << l.size() << endl;
    cout << "List l2 contents: " << endl;
    for (auto it = l2.begin(); it != l2.end(); ++it)
        cout << *it << endl;
    cout << "List l3 contents: " << endl;
    for (auto it = l3.begin(); it != l3.end(); ++it)
        cout << *it << endl;
    return 0;
}
```

list2.cpp

Size of list l: 0

List l2 contents:

10

20

30

List l3 contents:

10

20

30

Container properties

Property	Description
Sequence	Sequence containers order their elements in a strict linear sequence. Elements are accessed by their position in the sequence.
Doubly-linked list	Every element has information on how to locate previous and next elements. This allows for constant time for insertion and deletion operations.
Allocator-aware	An allocator object is used for modifying the storage size dynamically.

Inserting into a List

- There are different functions that we can use to insert values into a list.

list3.cpp

```
#include <algorithm>
#include <iostream>
#include <list>
int main() {
    std::list<int> my_list = { 12, 5, 10, 9 };
    my_list.push_front(11);
    my_list.push_back(18);
    auto it = std::find(my_list.begin(), my_list.end(), 10);
    if (it != my_list.end()) {
        my_list.insert(it, 21);
    }
    for (int x : my_list) {
        std::cout << x << '\n';
    }
}
```

11
12
5
21
10
9
18

Deleting from a List

- It's possible to delete items from a list. The `erase()` function allows you to delete an item or a range of items from a list.
- To delete a single item, you simply pass one integer position. The item will be deleted.
- To delete a range, you pass the starting and the ending iterators. Let's demonstrate this.

```
#include <algorithm>
#include <iostream>
#include <list>
using namespace std;
int main() {
    std::list<int> my_list = { 12, 5, 10, 9 };
    cout << "List elements before deletion: ";
    for (int x : my_list) {
        std::cout << x << '\n';
    }
    list<int>::iterator i = my_list.begin();
    my_list.erase(i);
    cout << "\nList elements after deletion: ";
    for (int x : my_list) {
        std::cout << x << '\n';
    }
    return 0;
}
```

list4.cpp

List elements before deletion: 12

5

10

9

List elements after deletion: 5

10

9

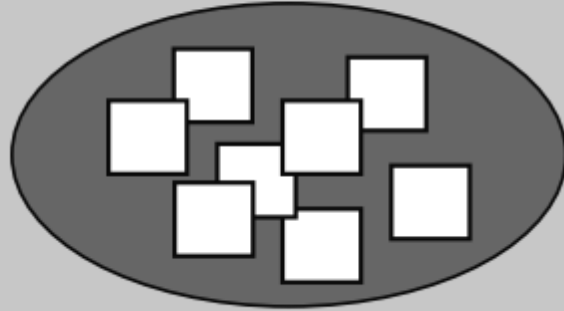
Summary

1. The `std::list` is a storage container.
2. It allows the insertion and deletion of items from anywhere at constant time.
3. It is implemented as a doubly-link
4. The `std::list` data can be accessed bi-directionally and sequentially.
5. `std::list` doesn't support fast random access. However, it supports sequential access from all directions.
6. You can scatter list elements of `std::list` in different memory chunks.
7. You can shrink or expand `std::list` from both ends as needed during runtime.
8. To insert items into `std::list`, we use the `insert()` function.
9. To delete items from the `std::list`, we use the `erase()` function.

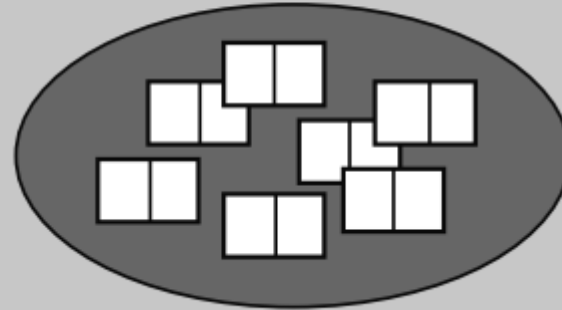
LECTURE 8

Associative Containers

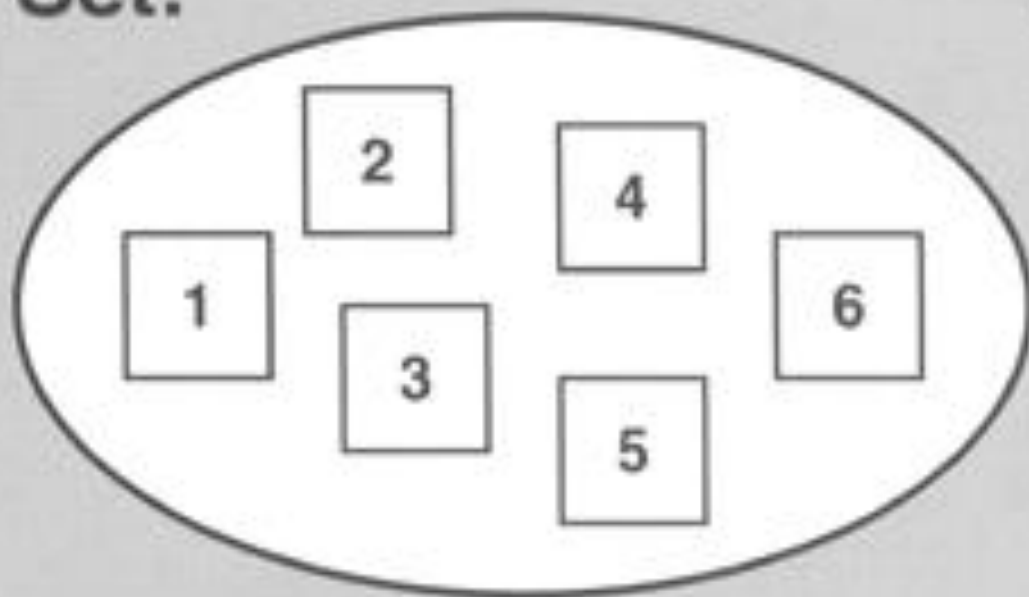
Unordered Set/Multiset:



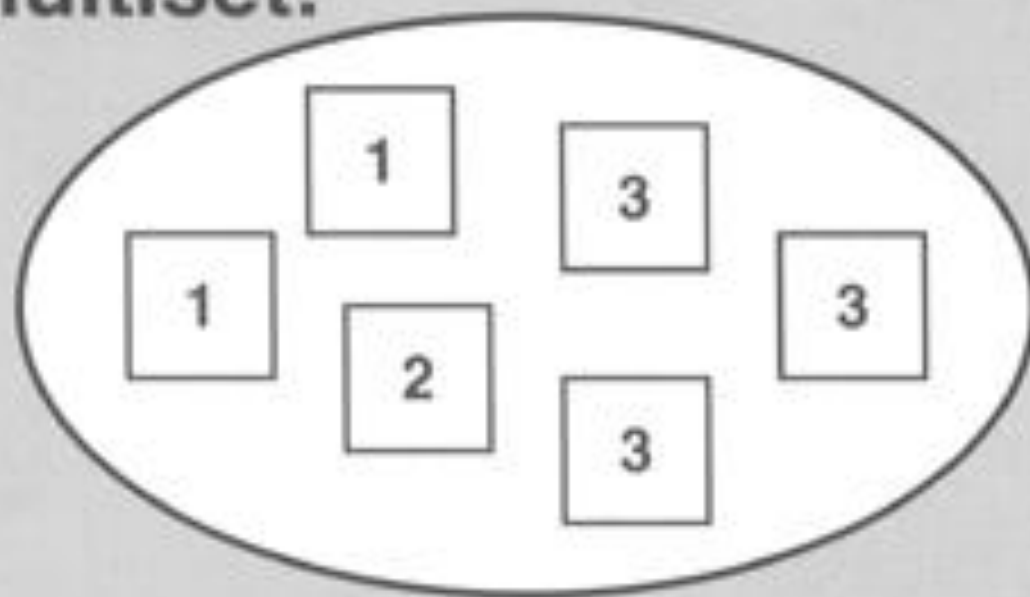
Unordered Map/Multimap:



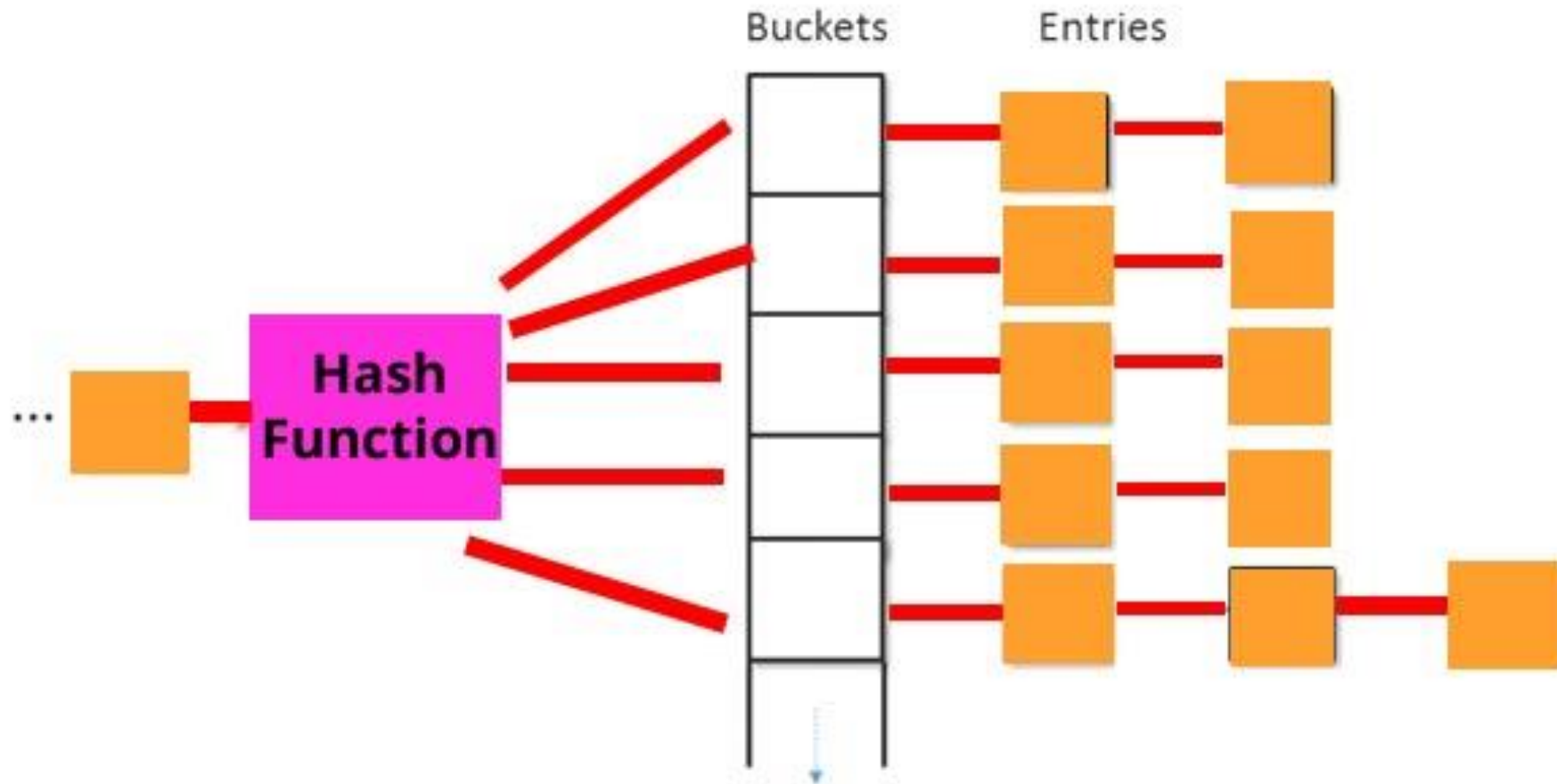
Set:



Multiset:

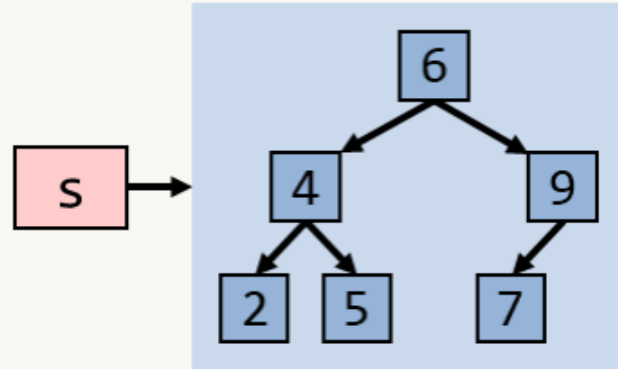


Implementation of Unordered Associative Containers



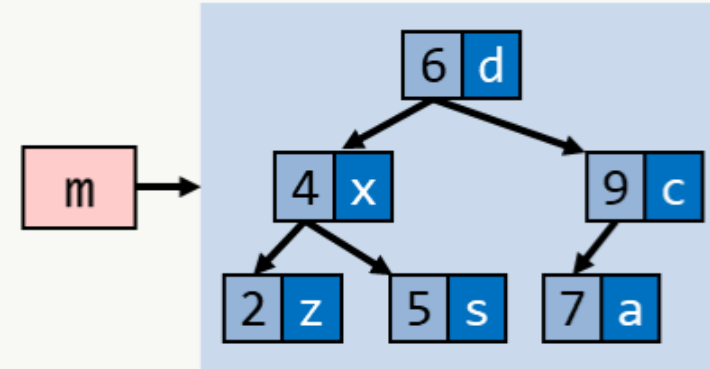
set<Key>

multiset<K>



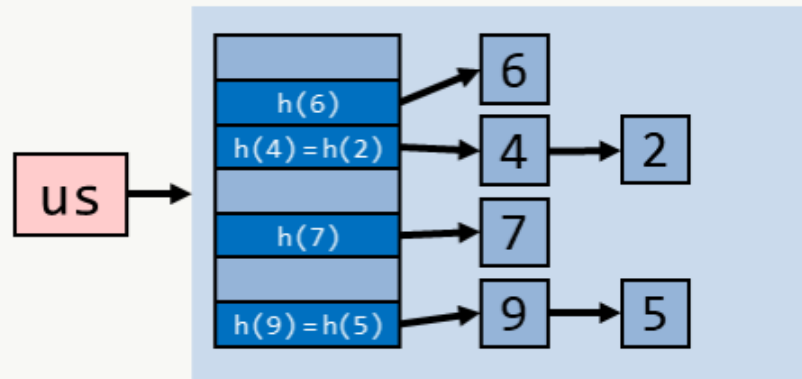
map<Key, Value>

multimap<K, V>



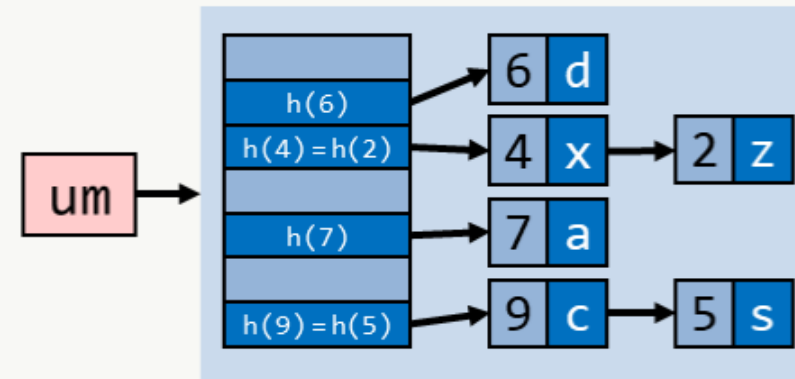
unordered_set<Key>

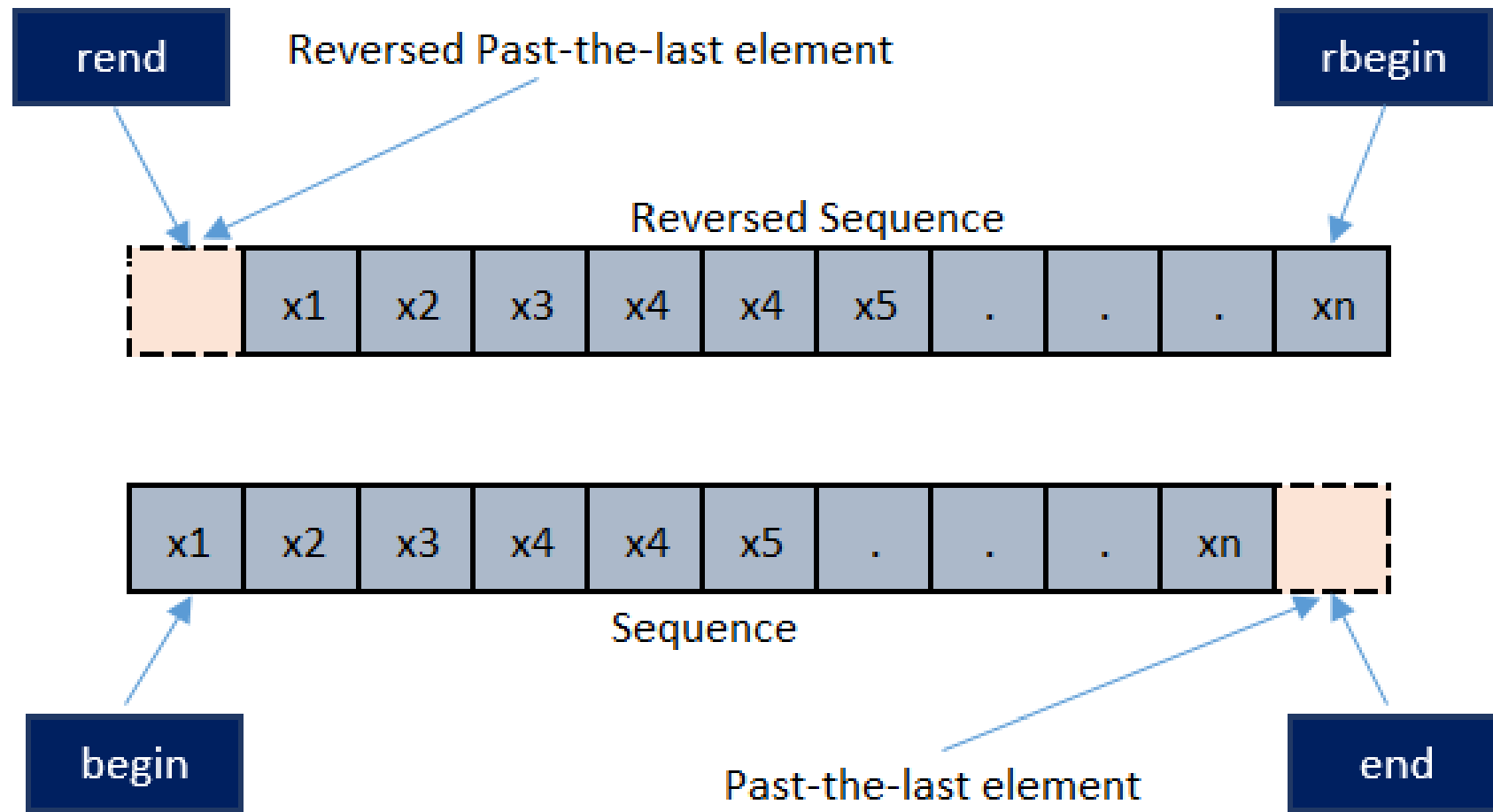
unordered_multiset<Key>



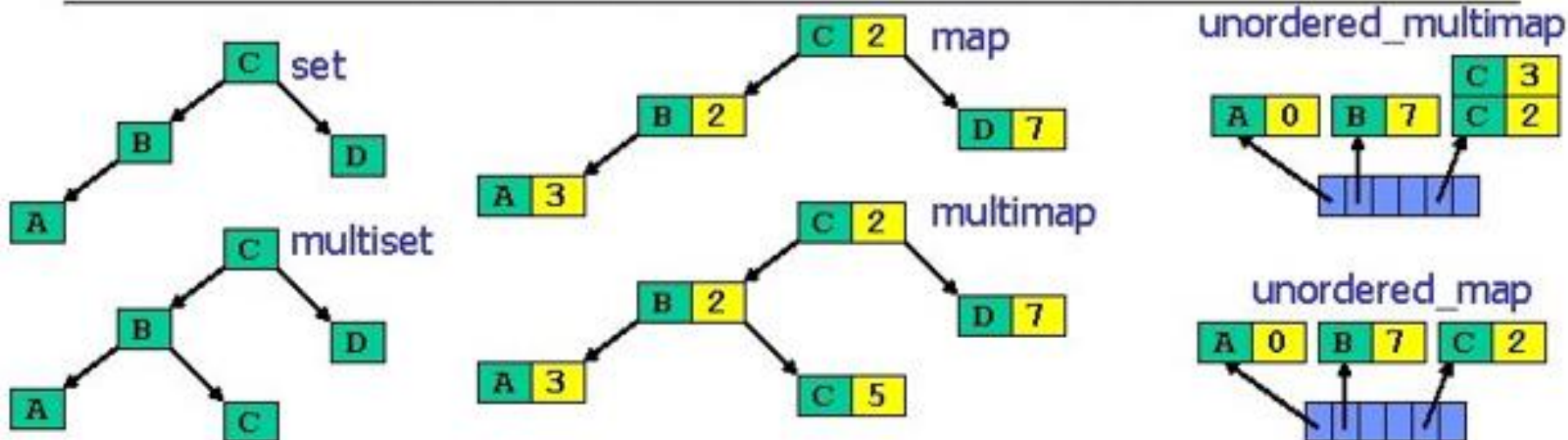
unordered_map<Key, Value>

unordered_multimap<Key, Value>





Associative Containers



- Associative containers support efficient key lookup
 - vs. sequence containers, which lookup by position
- Associative containers differ in 3 design dimensions
 - Ordered vs. unordered (tree vs. hash structured)
 - We'll look at ordered containers today, unordered next time
 - Set vs. map (just the key or the key and a mapped type)
 - Unique vs. multiple instances of a key

Associative Container	Sorted	Value	Several identical keys possible	Access time
<code>std::set</code>	yes	no	no	logarithmic
<code>std::unordered_set</code>	no	no	no	constant
<code>std::map</code>	yes	yes	no	logarithmic
<code>std::unordered_map</code>	no	yes	no	constant
<code>std::multiset</code>	yes	no	yes	logarithmic
<code>std::unordered_multiset</code>	no	no	yes	constant
<code>std::multimap</code>	yes	yes	yes	logarithmic
<code>std::unordered_multimap</code>	no	yes	yes	constant

CONTAINER	TYPES OF ITERATOR SUPPORTED
Vector	Random-Access
List	Bidirectional
Deque	Random-Access
Map	Bidirectional
Multimap	Bidirectional
Set	Bidirectional
Multiset	Bidirectional
Stack	No iterator Supported
Queue	No iterator Supported
Priority-Queue	No iterator Supported

LECTURE 9

set

Set in C++

- Set is a C++ STL container used to store the unique elements, and all the elements are stored in a sorted manner.
- Once the value is stored in the set, it cannot be modified within the set; instead, we can remove this value and can add the modified value of the element.
- Sets are implemented using Binary search trees.

Using sets

- Sets are traversed using the iterators.
- We have to include the two main header files to work with the sets.

```
#include<set>
```

```
#include<iterator>
```

- Instead of using the above two header files, we can also use only this one header file.
- This one header file contains all the header file , so we can only use this one header file , instead of all the other header files .

```
#include<bits/stdc++.h>
```

Basic Functions

- **begin()** :- Returns an iterator to the first element of the set .
- **end()** :- Returns an iterator to the theoretical element that follows last element in the set .
- **empty()** :- check wheather the set is empty or not .
- **max_size()** :-Returns the maximum number of element that set can hold .
- **size()** :- Returns the number of elements in the set .


```
#include<bits/stdc++.h>
using namespace std;
int main(){
    set <int> s;
    // inserting elements in random order .
    s.insert( 60 ) ;
    s.insert( 10 ) ;
    s.insert( 20 ) ;
    s.insert( 20 ) ;
    s.insert( 40 ) ;
    s.insert( 50 ) ;

    // printing set s initialising the iterator, iterating to the beginning of the set.
    set<int >::iterator it ;
    cout << "The element of set s are : \n";
    for (it = s.begin() ; it != s.end() ; it++){
        cout << *it<<" ";
    }
    cout << endl;
    cout<< "The size of set : \n " << s.size() <<endl ;
    return 0 ;
}
```

set1.cpp

The element of set s are :

1020405060

The size of set :

5

Store in Different Order

- To store the elements in decreasing order in the set.
- By default, elements are stored in ascending sorted order in the set. To store the elements in descending order, we have to use the given declaration syntax.

```
// declaration syntax of the set to store the elements  
// in decreasing sorted order.
```

```
set<int, greater<int>> s;
```

Functions

find() :-

Returns an iterator pointing to the element, if the element is found else return an iterator pointing to the end of the set.

Syntax:

```
set_name.find(key) ;
```

erase() :-

this method is used to delete the elements from the set

Syntax:

```
set_name.erase(starting_iterator , ending_iterator) ;
```

```

#include<bits/stdc++.h>
using namespace std;
int main(){
    set<int>s ;
    s.insert(34); s.insert(14); s.insert(13); s.insert(56);
    s.insert(5); s.insert(23); s.insert(10); s.insert(4);

    set<int>::iterator it;
    cout<<"the elements of set are : \n";
    for(it = s.begin() ; it != s.end() ; it++){
        cout<<*it<< " ";
    }
    cout<< endl;

    // Removing all elements upto 10
    cout<< "elements of set s after deleting elements upto 10 \n";
    s.erase(s.begin(), s.find(10));
    for (it = s.begin() ; it!=s.end() ; it++) cout<< *it << " ";
    // Performing Lower bound and upper bound in set
    cout<< "\nlower bound of 13 \n";
    cout<< *s.lower_bound(13)<<endl;
    cout<< "lower bound of 34\n";
    cout<< *s.lower_bound(34) << endl;
    cout<< "upper bound of 13 \n";
    cout<< *s.upper_bound(13) <<endl;
    cout<< "upper bound of 34 \n";
    cout<< *s.upper_bound(34) <<endl ;
    return 0;
}

```

set2.cpp

the elements of set are :

4 5 10 13 14 23 34 56

elements of set s after deleting
elements upto 10

10 13 14 23 34 56

lower bound of 13
13

lower bound of 34
34

upper bound of 13
14

upper bound of 34
56

Some methods on set

- 1.**begin()**:- Returns an iterator to the first element in the set.
- 2.**end()**:- Returns an iterator to the theoretical element that follows the last element in the set.
- 3.**empty()**:- check whether the set is empty or not.
- 4.**size()** :- Returns the number of elements in the set .
- 5.**max_size()** :- Returns the maximum number of elements that set can hold.
- 6.**rbegin()**:- Returns a reverse pointer, pointing to the last element of the set.
- 7.**rend()**:- Returns a reverse iterator pointing to the theoretical element right before the first element in the set container.
- 8.**erase (iterator position)**:- Removes the element at the position pointed by the pointer.
- 9.**erase(const x)** :- Removes the value x from the set .

Some methods on set

- 10. **insert(const x)** :- Inserts a new element to the set .
- 11. **find(x)**:- Returns an iterator based on the position of the element if it is found, else return iterator at the end.
- 12. **count(const x)** :-Return 1 or 0 based on the element x is found in the set or not .
- 13. **lower_bound(const x)**:- Returns an iterator to the element x if it is found else, return the iterator to the next element of it.
- 14. **upper_bound(const x)** :- Returns an iterator to the first element after the element x .
- 15. **emplace()**:- this function is used to insert the new element into the set container, only if the element to be inserted is unique and does not already exist in the set.
- 16. **swap()**:- This function is used to swap the content of the two sets, but the sets must be of the same type.
- 17. **clear()** :- Removes all elements from the set .

LECTURE 10

Difference between set, multiset, and unordered_set

Set

- A Set stores the elements in sorted order.
- Set stores **unique** elements.
- Elements can only be inserted and deleted but cannot be modified within the set.
- Sets are implemented using a **binary search tree**.
- Sets are traversed using iterators.

set3.cpp

Element is present in the set

```
#include <iostream>
#include <unordered_set>
using namespace std;
int main(){
    std::unordered_set<int> s = { 1, 2, 3, 4, 5 };
    int key = 3;

    if (s.find(key) != s.end()) {
        std::cout << "Element is present in the set" << endl;
    }
    else {
        std::cout << "Element not found" << endl;
    }
    return 0;
}
```

```

#include<bits/stdc++.h>
using namespace std;
int main(){
    set<int> s;
    s.insert(13);    s.insert(11);    s.insert(44);    s.insert(1);
    s.insert(10);    s.insert(25);    s.insert(11);

    set<int>::iterator ptr1, ptr2 ,ptr3;
    cout << "set elements are initially  : \n";
    for (ptr1 = s.begin(); ptr1 != s.end(); ptr1++){
        cout<< *ptr1 << " ";
    }
    cout <<endl ;

    set <int , greater<int>> s2(s.begin(), s.end());
    set <int , greater<int>>::iterator itr ;
    ptr2 = s.find(11);
    ptr3 = s.find(25);
    // elements from 11 to 25 are erased from the set using the erase().
    s.erase(ptr2 , ptr3);
    cout << "set element after erase  : \n";
    for (ptr1 = s.begin() ; ptr1 != s.end() ; ptr1++){
        cout << *ptr1 << " ";
    }
    cout << endl ;
    cout << "elements of set s2 are : \n";
    for (itr = s2.begin() ; itr != s2.end(); itr++){
        cout << *itr << " ";
    }
    return 0;
}

```

set4.cpp

set elements are initially :

1 10 11 13 25 44

set element after erase :

1 10 25 44

elements of set s2 are :

44 25 13 11 10 1

Multiset

- It also stores the elements in a **sorted** manner, either ascending or decreasing.
- Multiset allows storage of the **same values many times**; or, in other words, duplicate values can be stored in the multiset.
- We can delete the elements using iterators.

Note: – All other properties of a multiset are the same as that of the set, the difference being it allows storage of the same multiple values.

```

#include<bits/stdc++.h>
using namespace std;
int main(){
    multiset<int> ms;
    ms.insert(14); ms.insert(10); ms.insert(20); ms.insert(5);
    ms.insert(14);    // duplicate element added .
    ms.insert(25);
    ms.insert(25);    // duplicate element added .
    ms.insert(29);

    // declaring iterators to traverse the mutiset.
    multiset<int>::iterator it1, it2, it3;
    cout << "elements of multiset are : \n";
    for (it1 = ms.begin() ; it1 != ms.end() ; it1++){
        cout << *it1 << " ";
    }

    it1 = ms.find(10);
    it2 = ms.find(25);
    // removing the elements of multiset from 10 to 25 (exclusive) .
    ms.erase(it1, it2);
    cout << "elements of multiset after removing the elements : \n";
    for (it1 = ms.begin(); it1 != ms.end(); it1++){
        cout<< *it1 << " ";
    }
    return 0 ;
}

```

multiset1.cpp

elements of multiset are :
 5 10 14 14 20 25 25 29
 elements of multiset after
 removing the elements :
 5 25 25 29

Unordered_Set

- `Unordered_set` can store elements in any order, with no specified order for storing elements. It generally depends upon the machine that you are using.
- Stores unique elements, no duplicates allowed.
- `unordered_set` uses the hash table for storing the element.

```

#include<bits/stdc++.h>
using namespace std;
int main(){
    unordered_set<int> us;
    us.insert(14); us.insert(10);
    us.insert(20); us.insert(14);    // duplicates added .
    us.insert(12); us.insert(8);
    unordered_set<int>::iterator it1 , it2;
    cout << "elements of unordered_set are \n";
    for (it1 = us.begin(); it1 != us.end(); it1++){
        cout << *it1 << " ";
    }
    // erasing element 14
    it2 = us.find(14);
    us.erase(it2);
    cout<< "elements of unordered_set after erasing the element : \n";
    for(it1 = us.begin(); it1 != us.end(); it1++){
        cout << *it1 << " ";
    }
    return 0;
}

```

unorderedset1.cpp

elements of unordered_set are
 8 12 20 10 14 elements of
 unordered_set after erasing the
 element :
 8 12 20 10

LECTURE 11

map

What is a C++ Map?

- C++ map is an associative container that is used to store and sort elements in an orderly fashion. It's a part of the C++ Standard Template Library. Using the map, you can effortlessly search for elements based on their keys.
- Keys and values are essential to a C++ map. This is because elements are key-value pairs that follow a specific sequence. You can add or delete keys, but you can't alter them. Values, on the contrary, can be changed.

Note: since the keys are unique, no two values can have the same keys.

C++ Map Syntax

- The map syntax parameters are:

```
template < class Key,                                // map::key_type
          class T,                                    // map::mapped_type
          class Compare = less<Key>,                 // map::key_compare
          class Alloc = allocator<pair<const Key,T>> // map::allocator_type

> class map;
```

- **Key** specifies the keys that are going to be saved in the map.
- **T** represents the content that associates with the key.
- **Compare** is a binary predicate. It takes two keys as arguments and returns a bool.
- **Alloc** defines the storage allocation model. Usually it goes by the default value.

Map

- A C++ map is a way to store a key-value pair. A map can be declared as follows:

```
#include <iostream>

#include <map>

map<int, int> sample_map;
```

- Each map entry consists of a pair: a key and a value. In this case, both the key and the value are defined as integers, but you can use other types as well: strings, vectors, types you define yourself, and more.

C++ map use cases

- There are two main reasons why the map type can be valuable to C++ developers. First, a map allows **fast access** to the value using the key. This property is useful when building any kind of index or reference. Second, the map ensures that a key is **unique** across the entire data structure, which is an excellent technique for avoiding duplication of data.
- By virtue of these two advantages, a map is a common choice if you're developing, for instance, a trading application in which you need to store stock prices by ticker symbol. If you're creating a weather application, a map would be an effective way to save and look up the current temperature in a set of cities around the world. In an e-commerce store, you'll likely need a map to find products by identifiers or categories.

Constructing a map

There are five ways to construct a map in C++, but two of them are much more commonly used than the others. The first way is to create an empty map, then add elements to it:

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

int main() {
    map<int, string> sample_map;
    sample_map.insert(pair<int, string>(1, "one"));
    sample_map.insert(pair<int, string>(2, "two"));
    cout << sample_map[1] << " " << sample_map[2] << endl;
}
```

one two

Constructing a map

- In this example, we create a map that uses integers as keys and strings as values. We use the pair construct to create a key-value pair on the fly and insert that into our map.

Constructing a map

- The second often-used option is to initialize the map to a list of values at declaration. This option has been available since the C++11 standard and therefore isn't supported by older compilers, but it allows for clearer declaration:

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

int main() {
    map<int, string> sample_map{{1, "one"}, {2, "two"}};
    cout << sample_map[1] << " " << sample_map[2] << endl;
}
```

one two

Constructing a map

- Other ways to create a map include copying an existing map, copying parts of an existing map (by indicating a start position and an end position for the copy) and moving elements from another map without creating an intermediate copy.

Basic Functions

Function	Definition
<code>begin()</code>	Returns a bidirectional iterator (the pointer) to the first element of the map container. It should run in constant time.
<code>end()</code>	Returns the bidirectional iterator just past the end of the map. You'll have to decrement the iterator before you can access the last element of the container.
<code>empty()</code>	Checks if the map container is empty or not. Returns TRUE if the container has no elements and FALSE if otherwise.
<code>insert({key, element})</code>	Adds some data or an element with a particular key into the map.
<code>find(key)</code>	Runs in logarithmic time and returns an iterator to where the key is present in the map. If the key is not found, the iterator is returned to the end of the map.
<code>size()</code>	Returns the total amount of elements present in the map container.
<code>erase()</code>	Eliminates keys and elements at any given position or range in the map.
<code>clear()</code>	Runs in linear time, this function deletes all elements from the map and leaves 0 as its size.

Accessing map elements

- In order to access the elements of the map, you can use array-style square brackets syntax:

```
cout << sample_map[1] << endl;
```

- Another option, available as of C++11, is the at method:

```
cout << sample_map.at(1) << endl;
```

Iterating through elements

- In some cases, you might need to walk through a map and retrieve all the values in it. You can do this by using an iterator—a pointer that facilitates sequential access to a map's elements.
- An iterator is bound to the shape of the map, so when creating an iterator you'll need to specify which kind of map the iterator is for. Once you have an iterator, you can use it to access both keys and values in a map. Here's what the code would look like:

m3.cpp

one two three four

```
#include <bits/stdc++.h>
using namespace std;
int main() {
    map<int, string> sample_map { { 1, "one"}, { 2, "two" } };
    sample_map[3] = "three";
    sample_map.insert({ 4, "four" });

    map<int, string>::iterator it;
    for (it = sample_map.begin(); it != sample_map.end(); it++) {
        cout << it->second << " ";
    }
    cout << endl;
}
```

Iterating through elements

- In this example, the map contains pairs of `<int, string>`, so we create an iterator that matches that format. We use the `sample_map.begin()` method to point the iterator to where it should start, and indicate that the for loop should stop when we reach the `sample_map.end()` location—the end of the map.
- The iterator provides an `it->first` function to access the first element in a key-value pair (the key), and then `it->second` can be used to access the value. So, using the example above, we would print out only the values from our sample map.

When not to use a C++ map

- The map in C++ is a great fit for quickly looking up values by key. However, searching the contents of a map by value requires iterating through an entire map. If you want to be able to find values in a map, iterating through it can be slow as a map gets large.
- The [Boost library](#) offers a [bi-directional map](#) which performs better when searching for values often. This data structure isn't included in the standard C++ library, so you'll need to install the Boost library on each machine where you compile or run your program (dynamic linking), or alternatively include the Boost library inside your executable (static linking).
- If you find yourself needing to search a map by value in a relatively simple program, it may be that a map is the wrong object to use. Consider using a C++ vector, queue, stack or other data structure which might end up making the program more straightforward and more efficient.

unordered_map

- `unordered_map` is an associated container that stores elements formed by combination of key value and a mapped value. The key value is used to uniquely identify the element and mapped value is the content associated with the key. Both key and value can be of any type predefined or user-defined.
- Internally `unordered_map` is implemented using Hash Table, the key provided to map are hashed into indices of hash table that is why performance of data structure depends on hash function a lot but on an average the cost of **search, insert and delete** from hash table is $O(1)$.

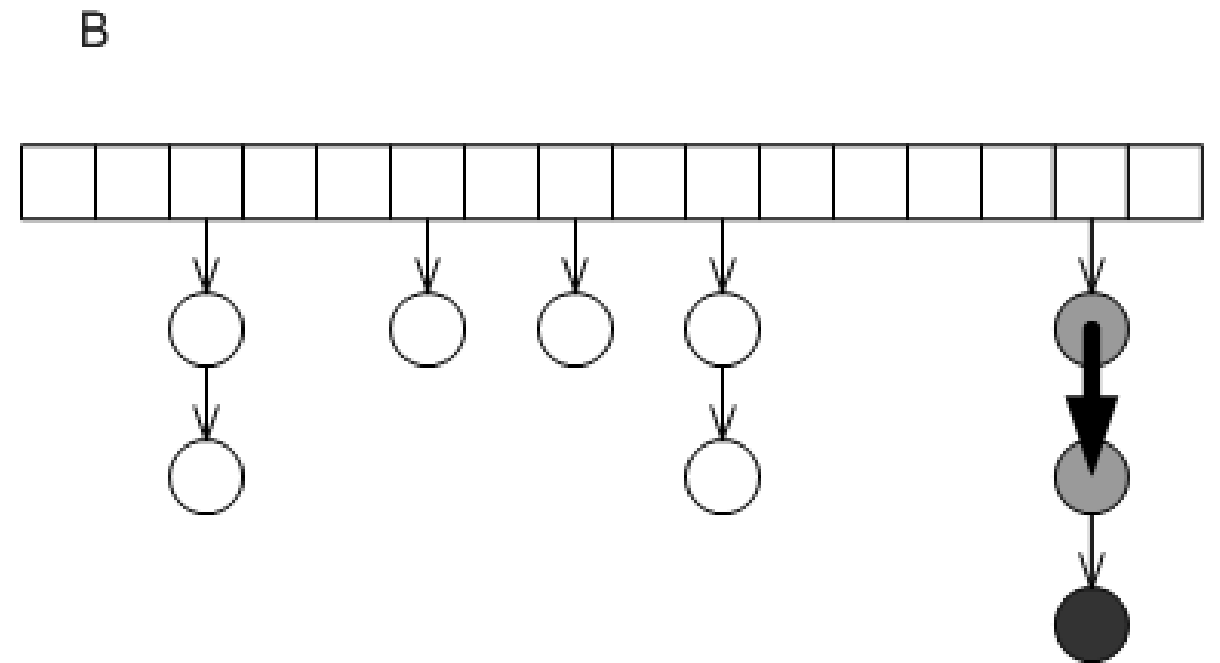
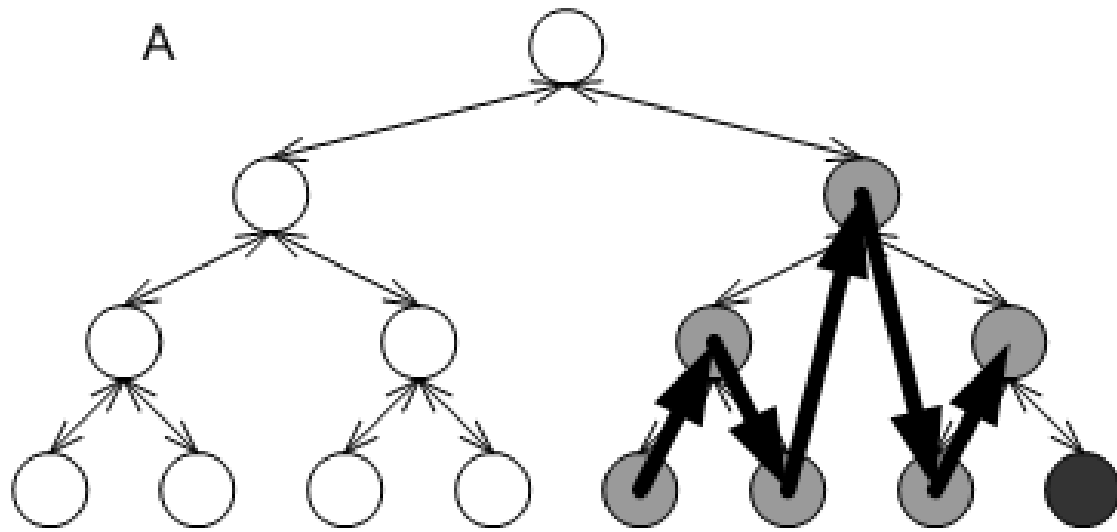


Figure: map and unordered_map

umap1.cpp

Contribute 30

Practice 20

GeeksforGeeks 10

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

int main(){
    unordered_map<string, int> umap;

    // inserting values by using [] operator
    umap["GeeksforGeeks"] = 10;
    umap["Practice"] = 20;
    umap["Contribute"] = 30;

    // Traversing an unordered map
    for (auto x : umap)
        cout << x.first << " " << x.second << endl;
    return 0;
}
```


multimap

- Multi-map in C++ is an associative container like map. It internally store elements in key value pair. But unlike map which store only unique keys, multimap can have duplicate keys.
- Also, it internally keep elements in sorted order of keys. By default it uses < operator to compare the keys.

MultiMap Usage Scenario

- Let's discuss a practical scenario, where we should use multi-map. Suppose, given a string and we want to store the position of each character in that string. For example, String is ***"this is it"*** and position of each character in string is,

t occurs at 0 , 9

h occurs at 1

i occurs at 2 , 5 , 8

s occurs at 3 , 6

- Here key is 'char' and value is 'int' i.e. position of character in string. So, there are multiple key value pairs with duplicate keys. Therefore, we will use multi-map to store this elements because in multi-map we can have duplicate keys.

Declaring a Multi-Map in C++

- Let's create a multi-map of char & int i.e.

```
std::multimap<char, int> mmapOfPos;
```

- We need to include following header file for it,

```
#include <map>
```

Initializing a MultiMap with C++11 Initializer list

```
std::multimap<char, int> mmapOfPos={  
    {'t', 1},  
    {'h', 1},  
    {'i', 2},  
    {'s', 3},  
    {'i', 5},  
    {'s', 6},  
    {'i', 8},  
};
```

Inserting a key value pair in MultiMap

- We can also insert a key value pair in multimap using insert() member function i.e.

```
mmapOfPos.insert(std::pair<char, int>('t', 9));
```

Iterating over the Multi-Map using Iterator

```
// Iterate over the multimap using Iterator
for(std::multimap<char, int>::iterator
    it = mmapOfPos.begin();
    it != mmapOfPos.end(); it++)
    cout << it->first << " :: " <<
        it->second << std::endl;
```

Iterating over the MultiMap using C++11 Range Based for loop

```
for(std::pair<char, int> elem : mmapOfPos)
    cout << elem.first << " :: " <<
        elem.second << std::endl;
```

```

#include <iostream>
#include <map>
#include <iterator>
#include <algorithm>
using namespace std;
int main(){
    multimap<char, int> mmapOfPos ={
        {'t', 1}, {'h', 1}, {'i', 2}, {'s', 3},
        {'i', 5}, {'s', 6}, {'i', 8},
    };
    // Inserting an element in map
    mmapOfPos.insert(std::pair<char, int>('t', 9));
    // Iterate over the multimap using Iterator
    for (multimap<char, int>::iterator it = mmapOfPos.begin();
        it != mmapOfPos.end(); it++)
        cout << it->first << " :: " << it->second << endl;
    cout << "*****" << endl;
    // Iterate over the multimap using range based for loop
    for (pair<char, int> elem : mmapOfPos)
        cout << elem.first << " :: " << elem.second << endl;
    return 0;
}

```

mmap1.cpp

```

h::1
i::2
i::5
i::8
s::3
s::6
t::1
t::9
*****
h::1
i::2
i::5
i::8
s::3
s::6
t::1
t::9

```