Standard Template Library

The modern C++

Your help

Your new friend

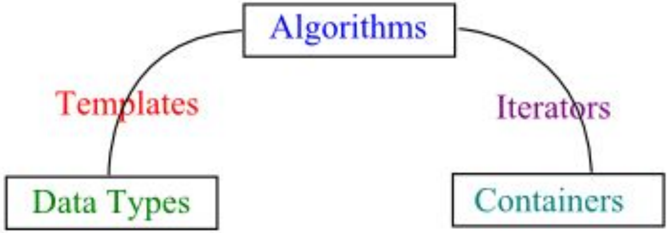
http://cplusplus.com/

http://en.cppreference.com/w/

Scott Meyers, Herb Sutter, Andrei Alexandrescu

Librărie → Un set de clase → Generice

- Algoritmi implementați → super optimi
- **Structuri de date** (containere) implementate → super mega optime
- **Iteratori** (wait, whaaaaaat?)

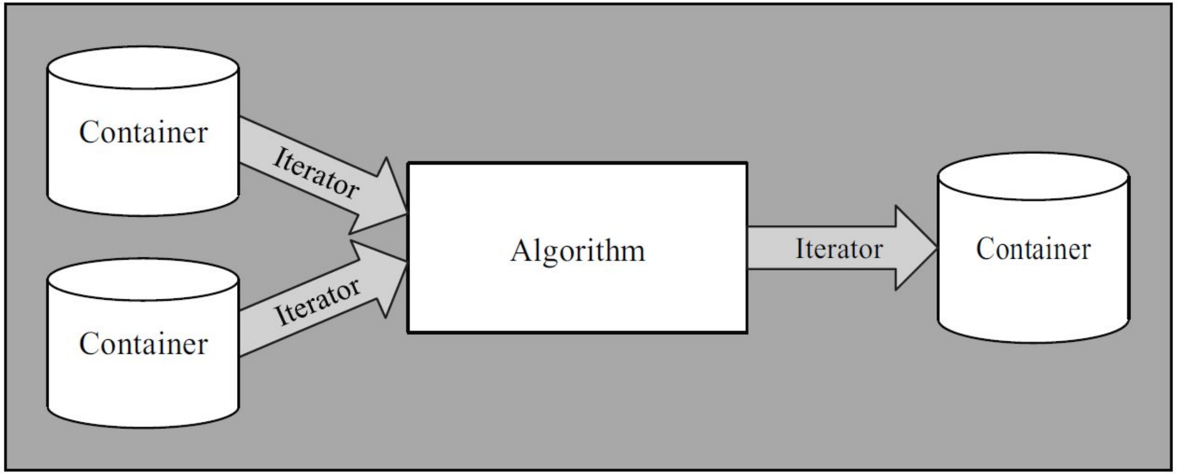


1. Templates

make algorithms independent of the data types

2. Iterators

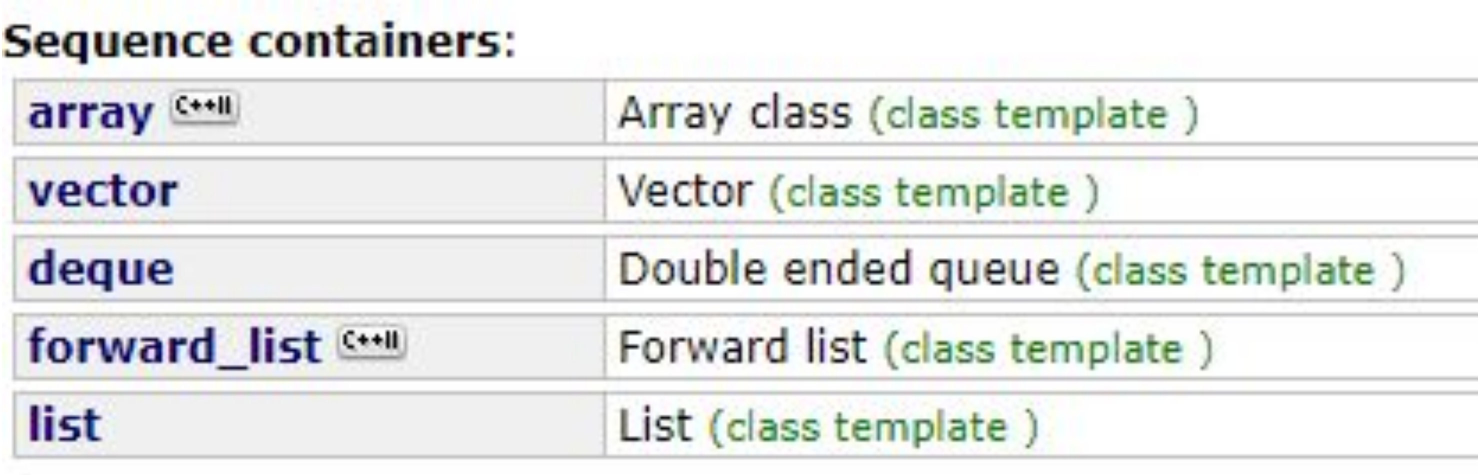
make algorithms independent of the containters



Structuri de date

Tipuri:

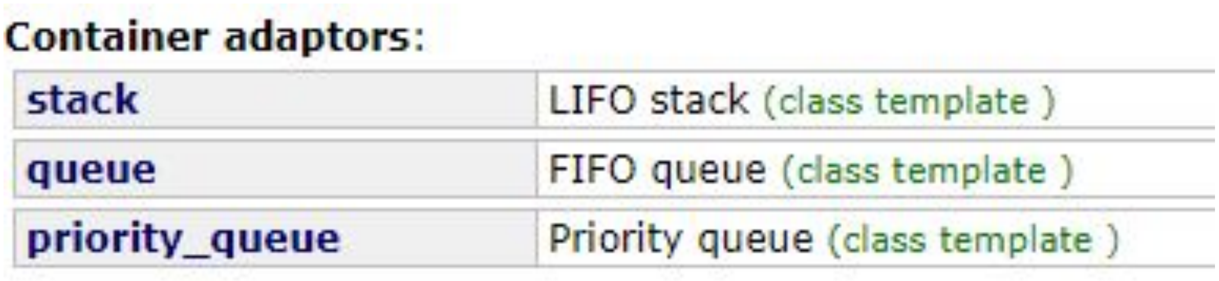
1. **Secvențiale / elementare:**



Sequence containers:

array C++11	Array class (class template)
vector	Vector (class template)
deque	Double ended queue (class template)
forward_list C++11	Forward list (class template)
list	List (class template)

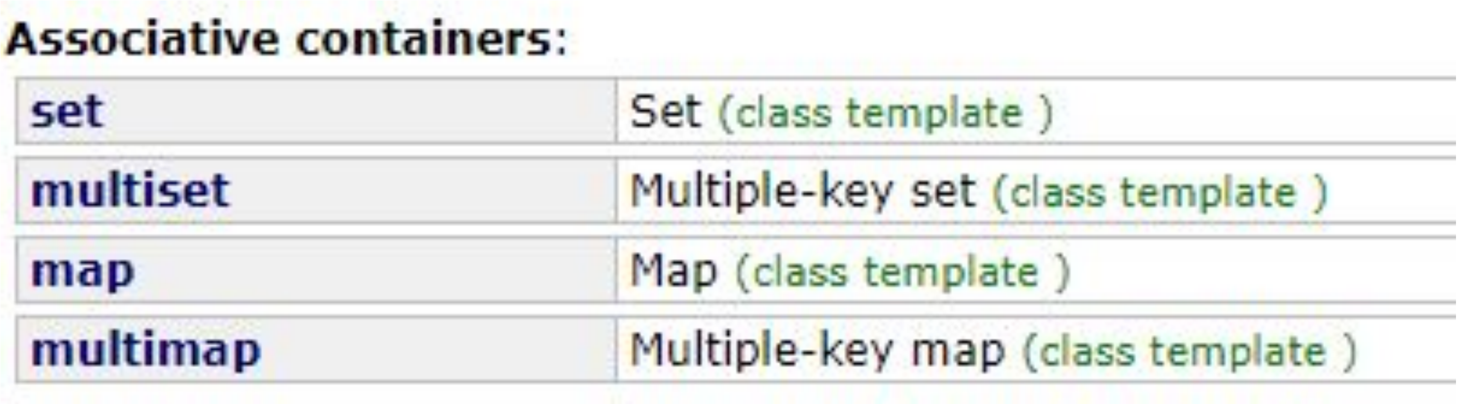
2. Adaptori / Structuri care au la bază o structură elementară



Container adaptors:

stack	LIFO stack (class template)
queue	FIFO queue (class template)
priority_queue	Priority queue (class template)

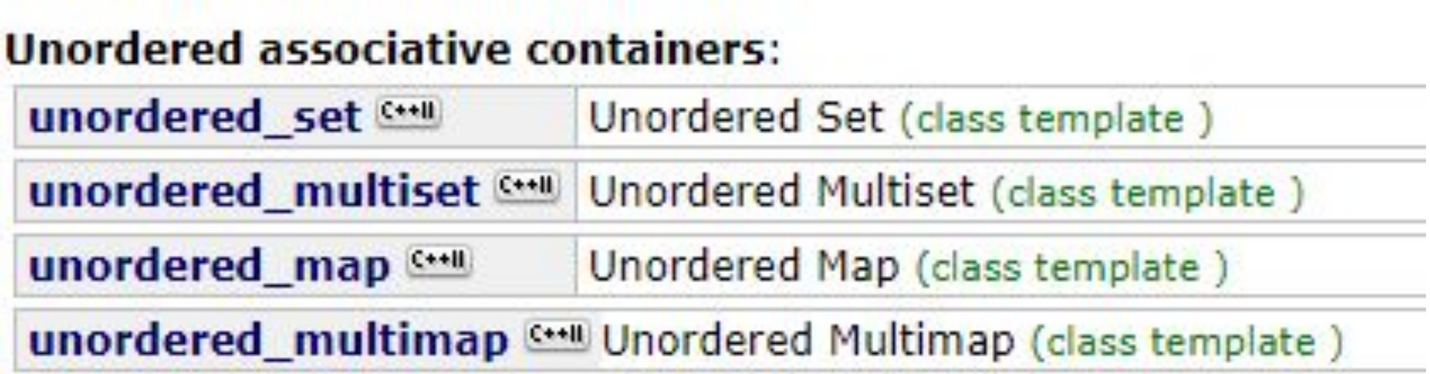
3. Asociative / Structuri în care elementele sunt sortate și căutarea este rapidă



Associative containers:

set	Set (class template)
multiset	Multiple-key set (class template)
map	Map (class template)
multimap	Multiple-key map (class template)

4. Asociative si nesortate



Unordered associative containers:

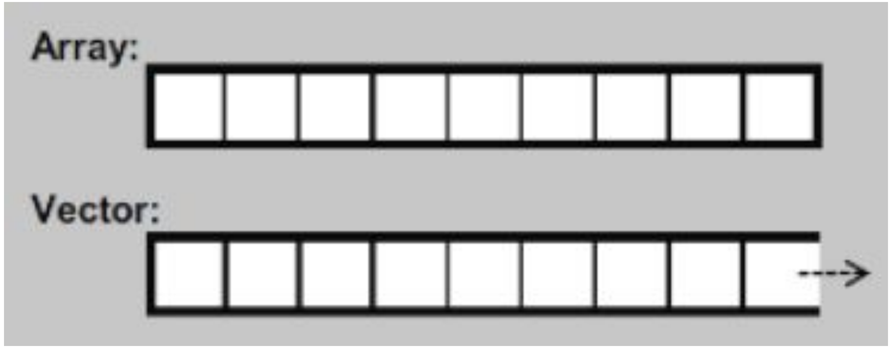
unordered_set C++11	Unordered Set (class template)
unordered_multiset C++11	Unordered Multiset (class template)
unordered_map C++11	Unordered Map (class template)
unordered_multimap C++11	Unordered Multimap (class template)

Bonus… Wait for it….

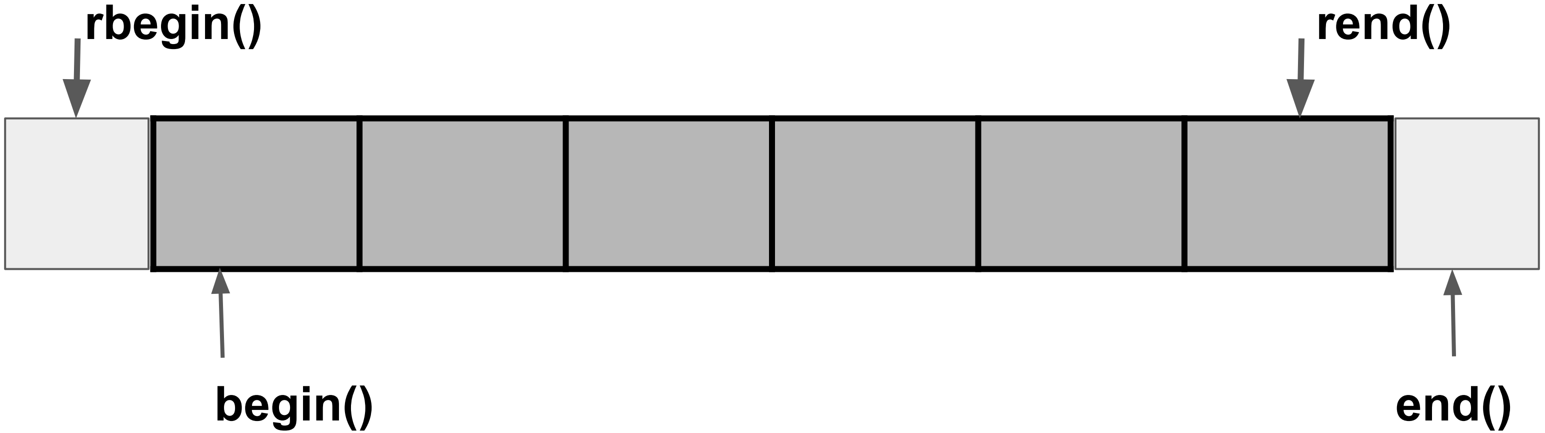
std::string

std::vector vs. std::array

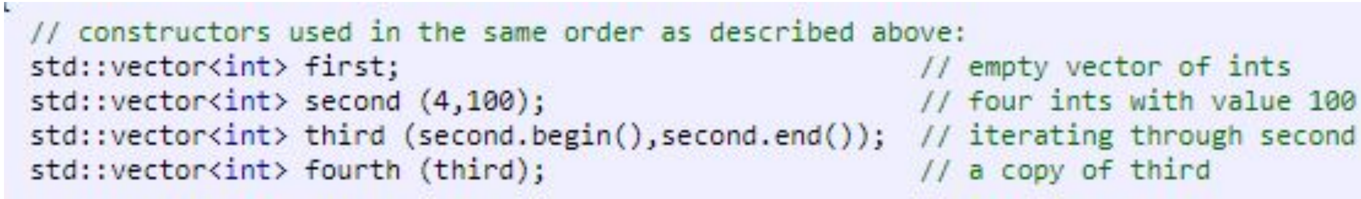
- Elementele sunt accesate prin poziție
- PRO: Accesare cu [] → random access (! iterator)
- PRO: Lungimea vectorului se poate schimba, array-ul are lungime fixă
- PRO: Eficient pentru adaugare (la sfârșitul structurii), ștergere, parcurgere
- CON: Ineficient pentru inserarea elementelor la o anumită poziție (vector)
- Baza vectorului este un array dinamic



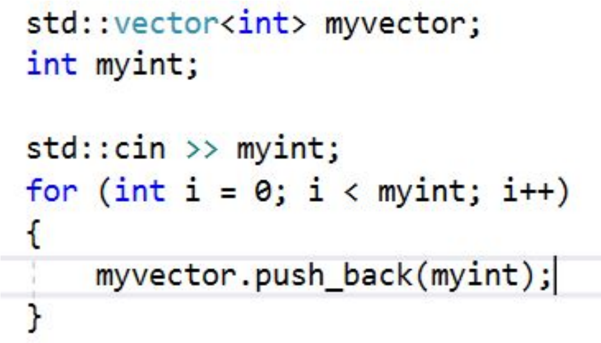
Iteratori



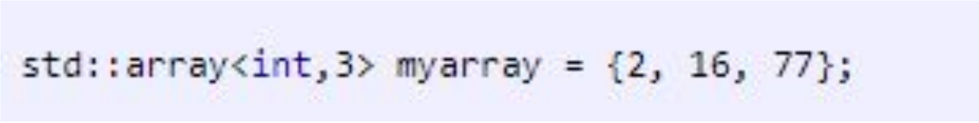
std::vector vs. std::array



```
// constructors used in the same order as described above:
std::vector<int> first;                                // empty vector of ints
std::vector<int> second (4,100);                       // four ints with value 100
std::vector<int> third (second.begin(),second.end());  // iterating through second
std::vector<int> fourth (third);                       // a copy of third
```



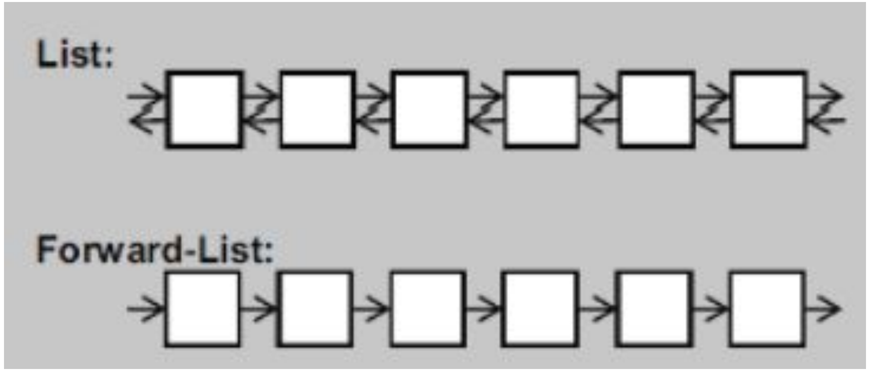
```
std::vector<int> myvector;
int myint;

std::cin >> myint;
for (int i = 0; i < myint; i++)
{
    myvector.push_back(myint);
}
```



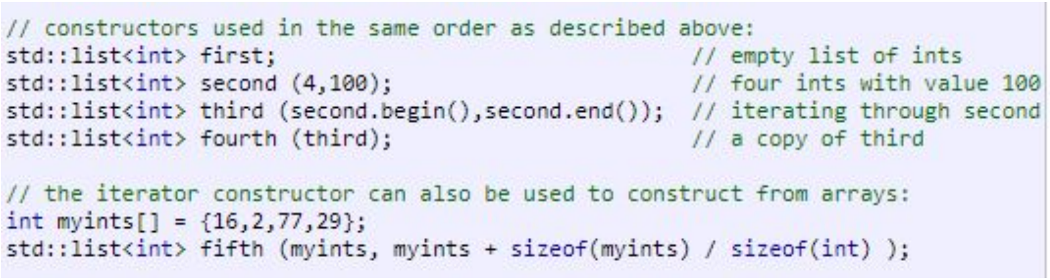
```
std::array<int,3> myarray = {2, 16, 77};
```

std::list vs. std::forward_list

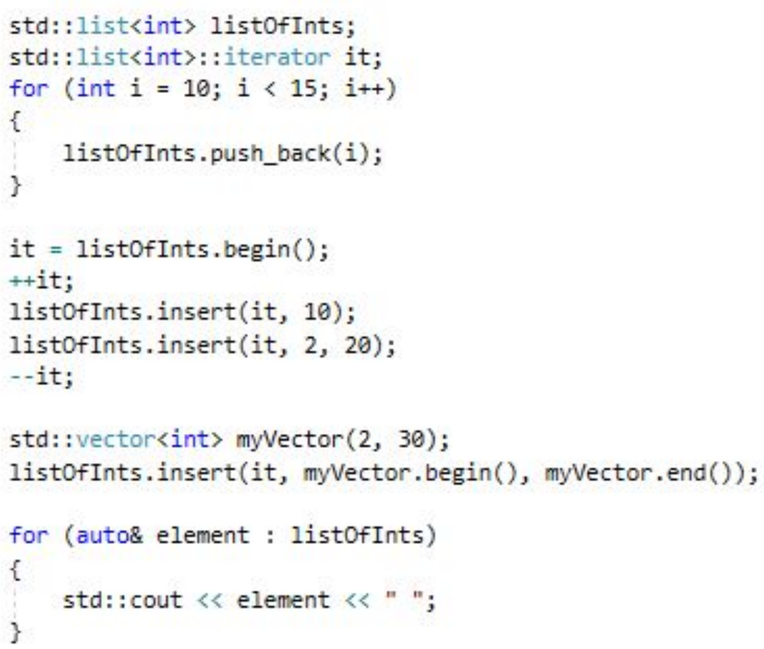
- Listă dublu înlănțuită și listă lănțuită
- PRO: Mai eficient pentru: inserarea, extragerea și mutarea elementelor de pe orice poziție
- CON: Nu avem acces direct la orice element din listă, se va itera de la o poziție cunoscută (begin() sau end()) până la poziția dorită



std::list   vs.   std::forward_list



```cpp
// constructors used in the same order as described above:
std::list<int> first;                                // empty list of ints
std::list<int> second (4,100);                       // four ints with value 100
std::list<int> third (second.begin(),second.end());  // iterating through second
std::list<int> fourth (third);                       // a copy of third

// the iterator constructor can also be used to construct from arrays:
int myints[] = {16,2,77,29};
std::list<int> fifth (myints, myints + sizeof(myints) / sizeof(int) );
```

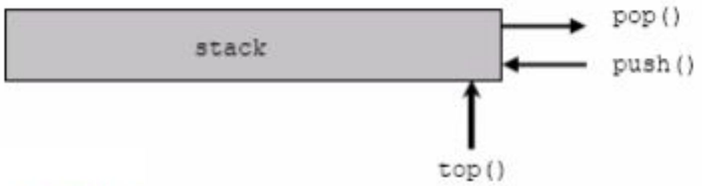


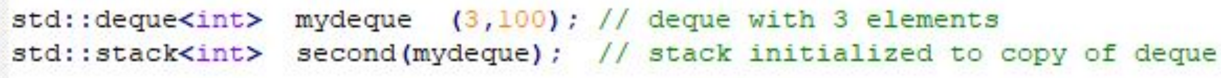
```cpp
std::list<int> listOfInts;
std::list<int>::iterator it;
for (int i = 10; i < 15; i++)
{
    listOfInts.push_back(i);
}

it = listOfInts.begin();
++it;
listOfInts.insert(it, 10);
listOfInts.insert(it, 2, 20);
--it;

std::vector<int> myVector(2, 30);
listOfInts.insert(it, myVector.begin(), myVector.end());

for (auto& element : listOfInts)
{
    std::cout << element << " ";
}
```

std::queue - std::stack

Containerul de bază (*underlying container*) este **std::deque**

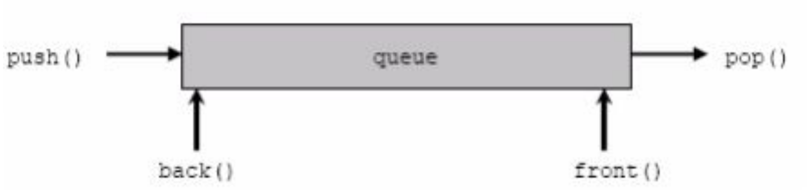
- std::stack → LIFO

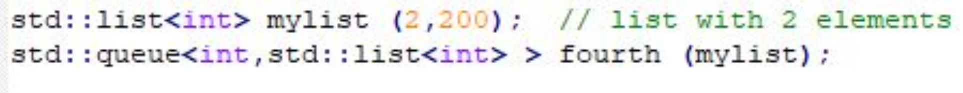




```
std::deque<int>  mydeque  (3,100); // deque with 3 elements
std::stack<int>  second(mydeque);  // stack initialized to copy of deque
```

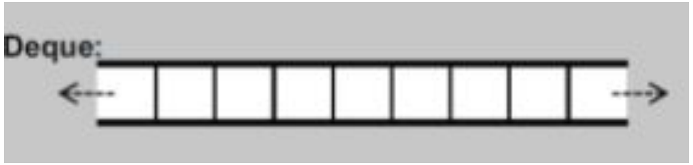
- std::queue → FIFO



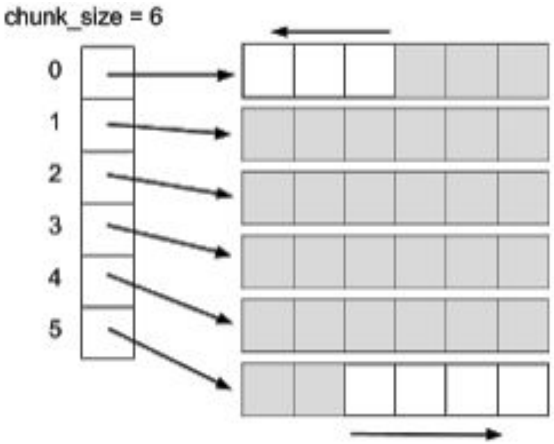


```
std::list<int> mylist (2,200);  // list with 2 elements
std::queue<int,std::list<int> > fourth (mylist);
```

std::deque



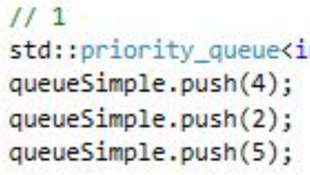
- Double ended queue
- Este o combinație între vector și listă:
  - Din vector: accesare aleatoare a elementelor
  - Din listă: inserare eficientă (chiar și la începutul structurii)
- În memorie este alocată ca niște blocuri (*chunks) continue*



std::priority_queue

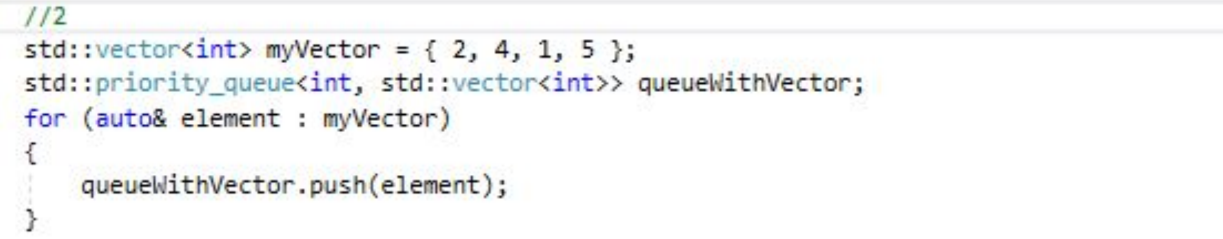
- fiecare element are o prioritate stabilită (heap-max)
- **template <class T, class Container = vector<T>,   class Compare = less<typename Container::value_type> > class priority_queue;**

1. std::priority_queue<int>



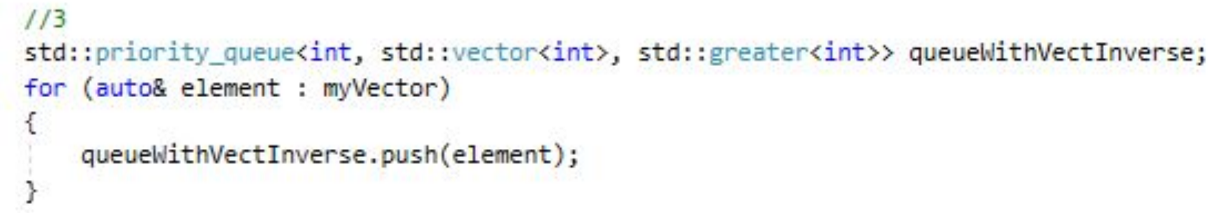
```
// 1
std::priority_queue<int> queueSimple;
queueSimple.push(4);
queueSimple.push(2);
queueSimple.push(5);
```

2.std::priority_queue<int, std::vector<int>>



```
//2
std::vector<int> myVector = { 2, 4, 1, 5 };
std::priority_queue<int, std::vector<int>> queueWithVector;
for (auto& element : myVector)
{
    queueWithVector.push(element);
}
```

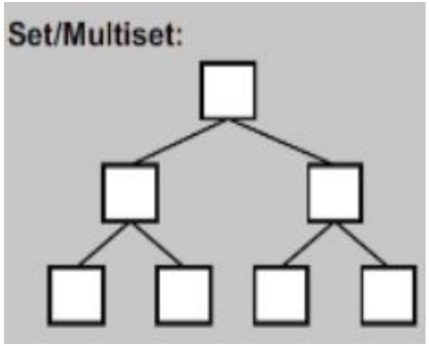
3. std::priority_queue<int, std::vector<int>,
std::greater<int>>



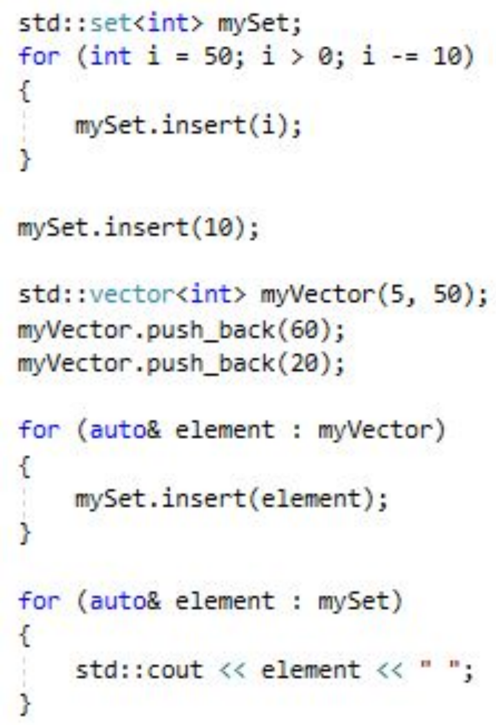
```
//3
std::priority_queue<int, std::vector<int>, std::greater<int>> queueWithVectInverse;
for (auto& element : myVector)
{
    queueWithVectInverse.push(element);
}
```

std::set

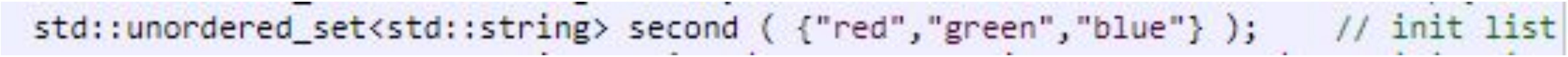
- PRO: Un set conține elemente **unice** (const), **sortate**
- Se accesează după **cheie** (= valoarea după care este sortat, cheia să fie comparabilă)
- PRO: Implementare: arbore binar de căutare (Mai exact?)
- în **unordered_set**: organizarea elementelor se întâmplă cu ajutorul unui hash table, accesare mai rapidă
- **Multi_set:** conține elemente duplicate (sortarea se va face cu o funcție hash)



std::set



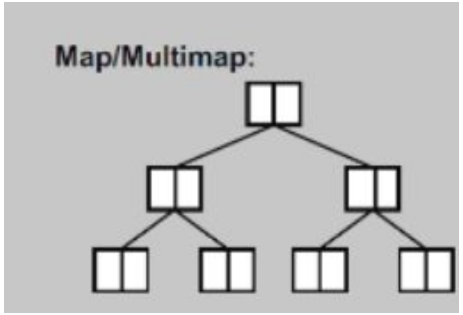
```
std::set<int> mySet;
for (int i = 50; i > 0; i -= 10)
{
    mySet.insert(i);
}

mySet.insert(10);

std::vector<int> myVector(5, 50);
myVector.push_back(60);
myVector.push_back(20);

for (auto& element : myVector)
{
    mySet.insert(element);
}

for (auto& element : mySet)
{
    std::cout << element << " ";
}
```



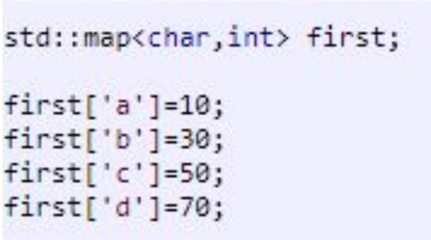
```
std::unordered_set<std::string> second ( {"red","green","blue"} );    // init list
```

std::map

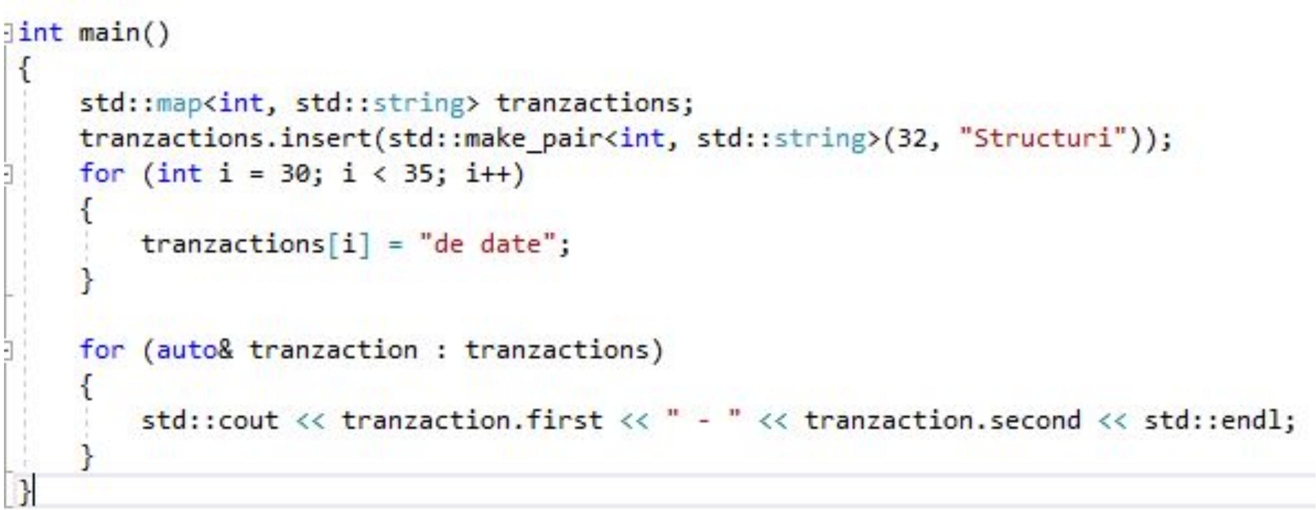
- Elementele dintr-o mapă sunt formate din perechi (**std::pair**)
  - **O cheie (key)**: unica, elementele se sorteaza dupa aceasta
  - **O valoare asociată cheii (mapped value)**
- PRO: Accesare direct cu operatorul [] → Ex: myMap[key] = newValue;
- Implementare: Arbore binar de căutare (posibil: arbore roșu-negru)
- **unordered_map**: elementele nu sunt sortate, cheia se folosește pentru a identifica un element. Se folosește o funcție hash.
- **Multi_map**: chei duplicate



std::map



```
std::map<char,int> first;

first['a']=10;
first['b']=30;
first['c']=50;
first['d']=70;
```



```
int main()
{
    std::map<int, std::string> tranzactions;
    tranzactions.insert(std::make_pair<int, std::string>(32, "Structuri"));
    for (int i = 30; i < 35; i++)
    {
        tranzactions[i] = "de date";
    }

    for (auto& tranzaction : tranzactions)
    {
        std::cout << tranzaction.first << " - " << tranzaction.second << std::endl;
    }
}
```

Metode

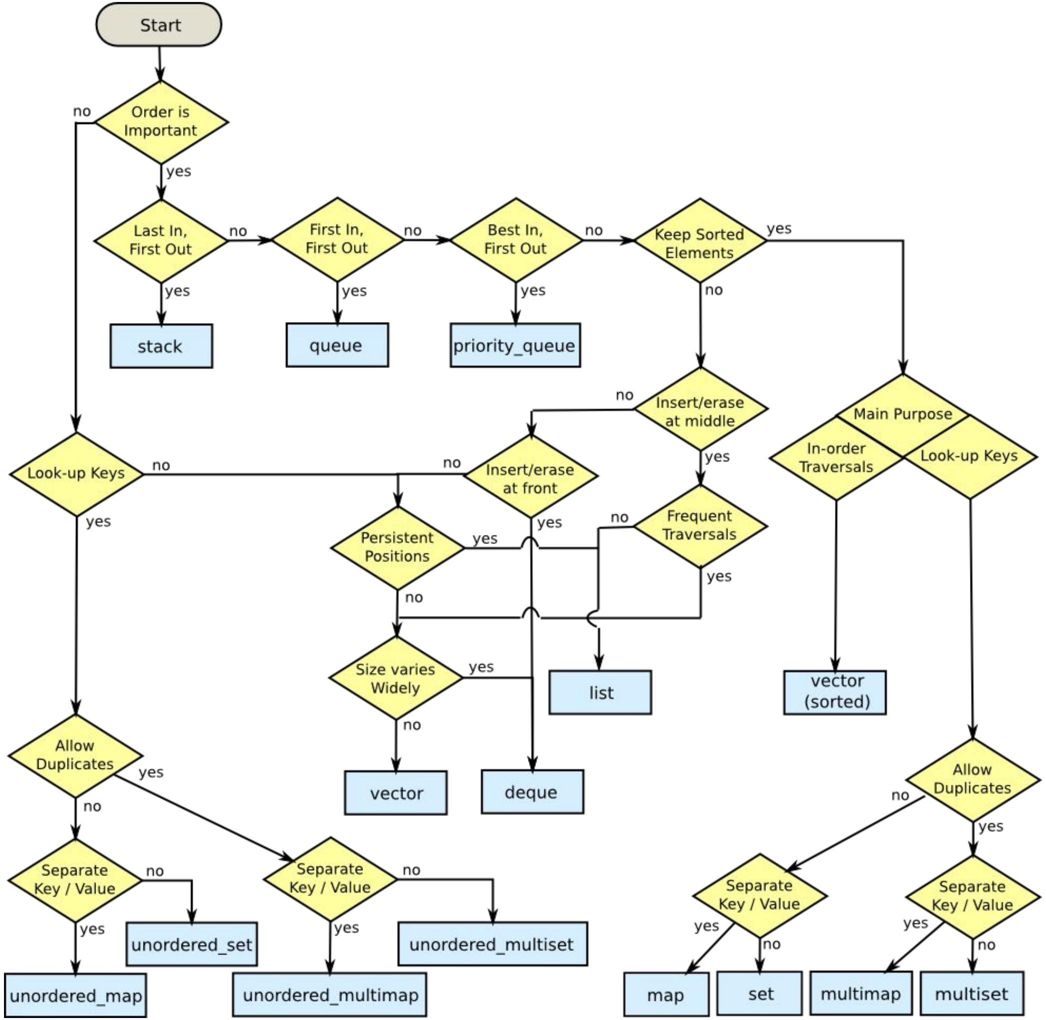
	Vector	Stack	Queue	Deque	Priority_ Queue	List

Element access	**operator [], at()**
	front()
	back()
		top	**back**()
			front()
				operator []
				at()
				front()
				back()
					top()	front()
						back()

Insert Element	push_back()
	insert()
		push()	**push**()	push_back()
				push_front()
				insert
					push()	push()
						push_back()

Remove Element	pop_back()
	erase()
		pop()	**pop**()	pop_back()
				pop_front()
				erase()
					pop()	pop()
						pop_back()
						remove()
						remove_if()

WHOA !!
NOW THAT'S TOO MUCH INFORMATION.
imgflip.com


Start
Order is Important
no
yes
Last In, First Out
no
yes
stack
First In, First Out
no
yes
queue
Best In, First Out
no
yes
priority_queue
Keep Sorted Elements
yes
no
Insert/erase at middle
no
yes
Insert/erase at front
no
yes
Frequent Traversals
no
yes
Persistent Positions
yes
no
Size varies Widely
yes
no
list
vector
deque
Main Purpose
In-order Traversals
Look-up Keys
vector (sorted)
Look-up Keys
no
yes
Allow Duplicates
yes
no
Separate Key / Value
no
yes
unordered_set
unordered_map
Separate Key / Value
no
yes
unordered_multiset
unordered_multimap
Allow Duplicates
no
yes
Separate Key / Value
yes
no
map
set
Separate Key / Value
yes
no
multimap
multiset

Algoritm

- const T& **min** (const T& a, const T& b);
- ForwardIterator **max_element** (ForwardIterator first, ForwardIterator last);
- bool **is_sorted** (ForwardIterator first, ForwardIterator last);
- void **sort** (RandomAccessIterator first, RandomAccessIterator last);
- void **reverse** (BidirectionalIterator first, BidirectionalIterator last);
- void **generate** (ForwardIterator first, ForwardIterator last, Generator gen);
- ForwardIterator **remove_if** (ForwardIterator first, ForwardIterator last, UnaryPredicate pred);
- OutputIterator **copy** (InputIterator first, InputIterator last, OutputIterator result);
- OutputIterator **copy_if** (InputIterator first, InputIterator last, OutputIterator result, UnaryPredicate pred);
- ForwardIterator1 **search** (ForwardIterator1 first1, ForwardIterator1 last1,ForwardIterator2 first2, ForwardIterator2 last2);
- Function **for_each** (InputIterator first, InputIterator last, Function fn);
- **count** (InputIterator first, InputIterator last, const T& val);
- OutputIterator **transform** (InputIterator first1, InputIterator last1, OutputIterator result, UnaryOperation op);

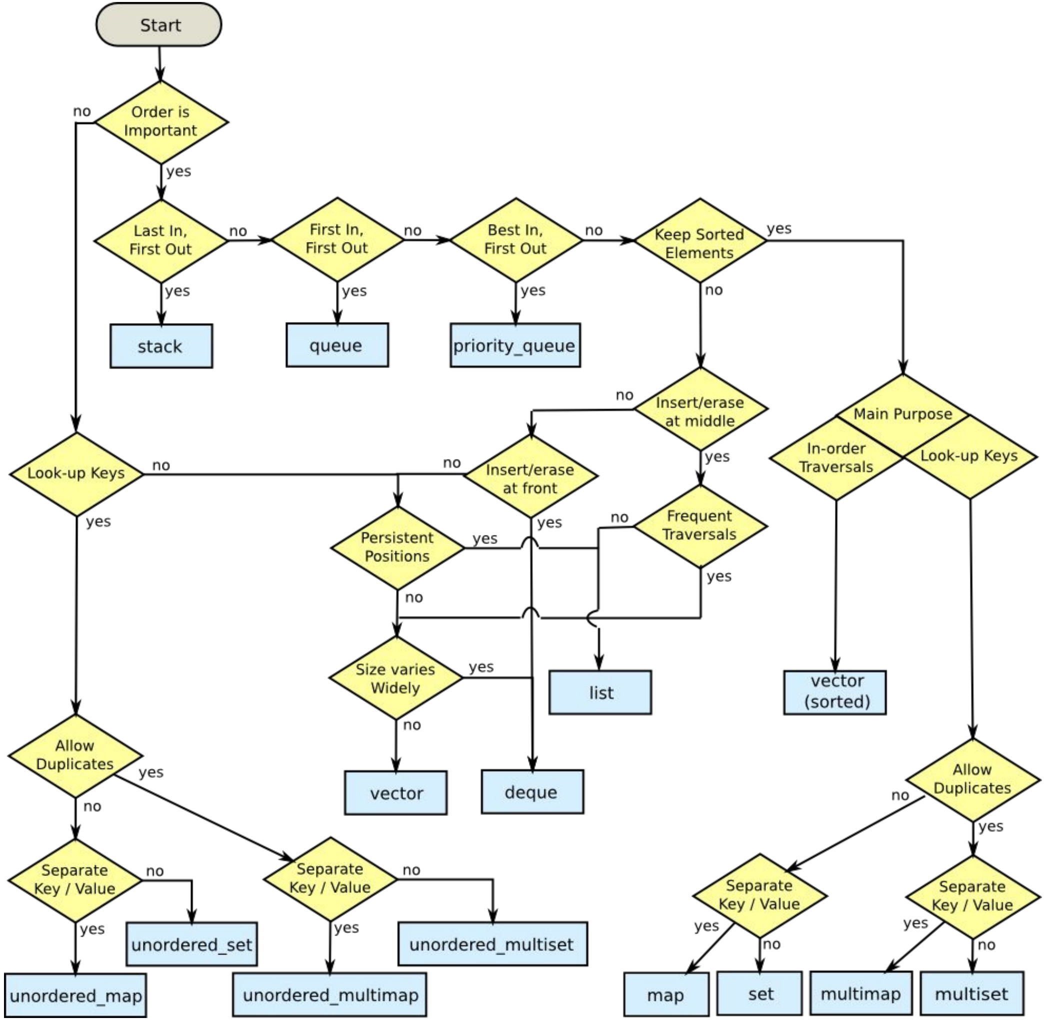
! Functors + STL !

Practice 1

A building security system has door locks that are opened by typing a four-digit access code into a keypad.

Each employee is given a different access code, which is activated using the activate() function. When an employee leaves the company, his or her access code is deactivated using the deactivate()function.

Store the access codes!


Start
Order is Important
no
yes
Last In, First Out
no
yes
stack
First In, First Out
no
yes
queue
Best In, First Out
no
yes
priority_queue
Keep Sorted Elements
yes
no
Insert/erase at middle
no
yes
Main Purpose
In-order Traversals
Look-up Keys
vector (sorted)
Look-up Keys
no
yes
Insert/erase at front
no
yes
Frequent Traversals
no
yes
Persistent Positions
yes
no
Size varies Widely
yes
no
list
vector
deque
Allow Duplicates
yes
no
Separate Key / Value
no
yes
unordered_set
unordered_map
Separate Key / Value
no
yes
unordered_multiset
unordered_multimap
Allow Duplicates
no
yes
Separate Key / Value
yes
no
map
set
Separate Key / Value
yes
no
multimap
multiset

Practice 1

A building security system has door locks that are opened by typing a four-digit access code into a keypad.

Each employee is given a different access code, which is activated using the activate() function. When an employee leaves the company, his or her access code is deactivated using the deactivate()function.

Store the access codes!

std::unordered_set<unsigned int>

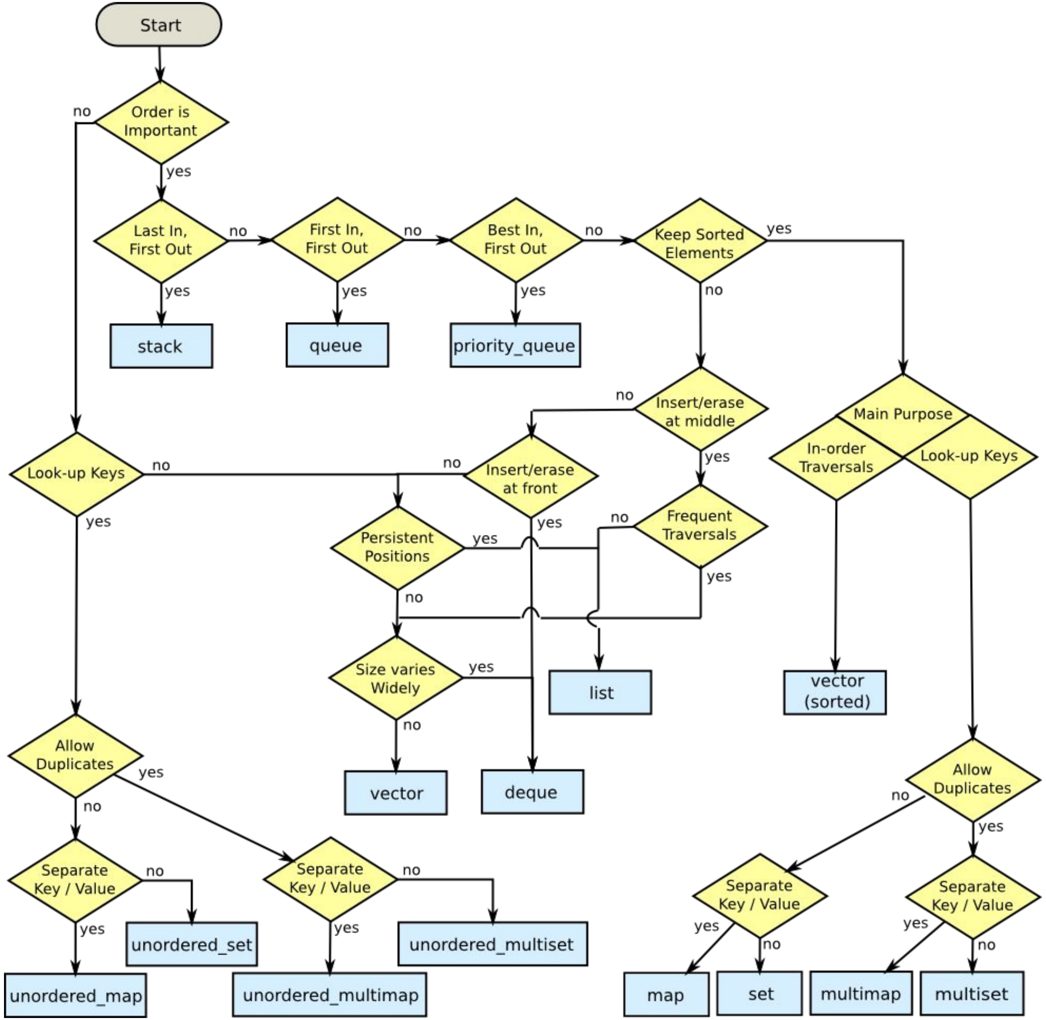
Practice 2

Write a program that asks the user to type 10 integers (stored in S) and an integer value (stored in V).

The program must search if the value V exists in the array and must remove the first occurrence of V.

Then shifting each following element left and adding a zero at the end of the array.

S = ? data structure


Start
Order is Important
no
yes
Last In, First Out
no
yes
stack
First In, First Out
no
yes
queue
Best In, First Out
no
yes
priority_queue
Keep Sorted Elements
yes
no
Insert/erase at middle
no
yes
Main Purpose
In-order Traversals
Look-up Keys
vector (sorted)
Look-up Keys
no
yes
Insert/erase at front
no
yes
Frequent Traversals
no
yes
Persistent Positions
yes
no
Size varies Widely
yes
no
list
vector
deque
Allow Duplicates
yes
no
Separate Key / Value
no
yes
unordered_set
unordered_map
Separate Key / Value
no
yes
unordered_multiset
unordered_multimap
Allow Duplicates
no
yes
Separate Key / Value
yes
no
map
set
Separate Key / Value
yes
no
multimap
multiset

Practice 2

Write a program that asks the user to type 10 integers (stored in S) and an integer value (stored in V).

The program must search if the value V exists in the array and must remove the first occurrence of V.

Then shifting each following element left and adding a zero at the end of the array.

S = ? data structure

std::list<int>

Practice 3

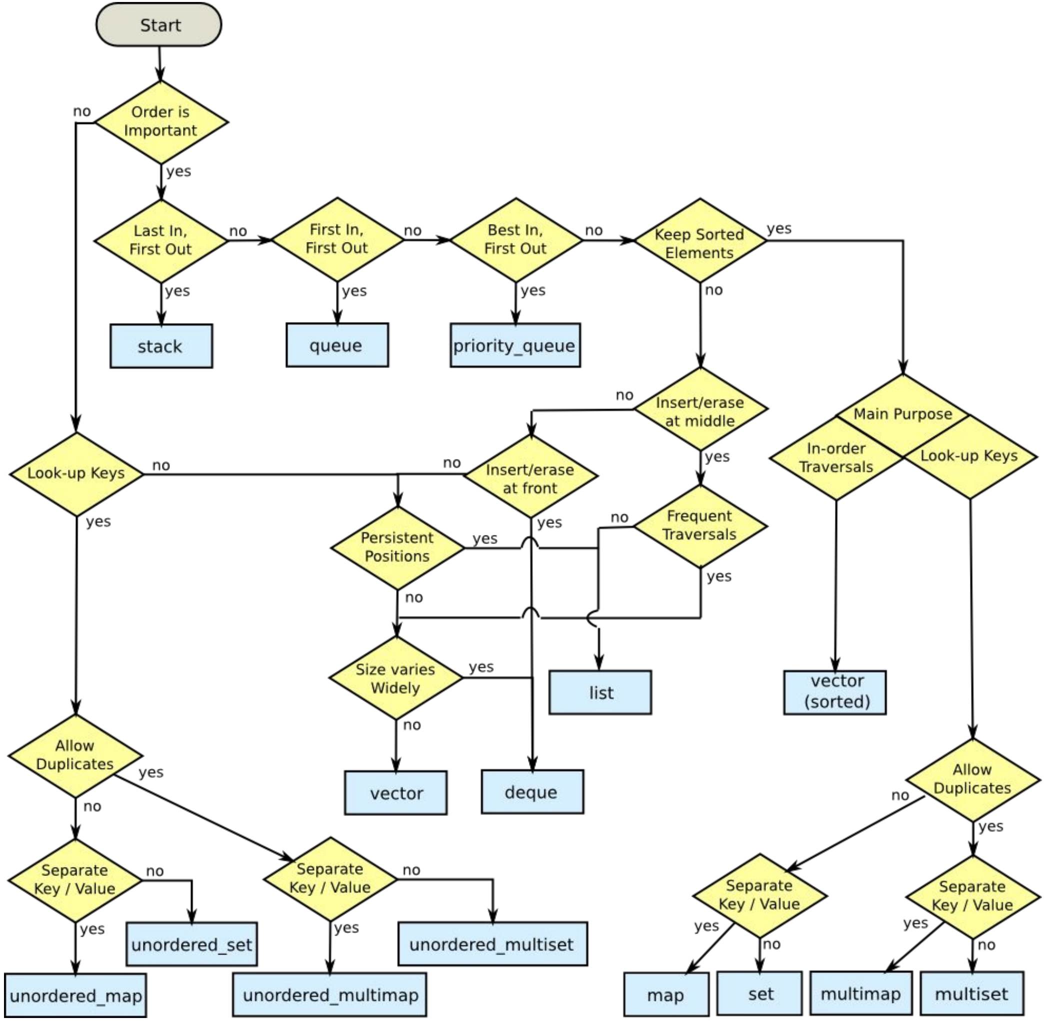
Write a program that asks the user **Type a name (or "exit")**

If that name was already typed in, display the associated phone number.

Otherwise ask for a number to be typed in and store the name and number. Then ask for another name until the user types "exit".

When “exit” is typed, print all names alphabetically and their corresponding phone number.

1) In what type of data structure will you store these informations?


Start
Order is Important
no
yes
Last In, First Out
no
yes
stack
First In, First Out
no
yes
queue
Best In, First Out
no
yes
priority_queue
Keep Sorted Elements
yes
no
Insert/erase at middle
no
yes
Main Purpose
In-order Traversals
Look-up Keys
Look-up Keys
no
yes
Insert/erase at front
no
yes
Frequent Traversals
no
yes
Persistent Positions
yes
no
Size varies Widely
yes
no
list
vector (sorted)
vector
deque
Allow Duplicates
yes
no
Separate Key / Value
no
yes
unordered_set
unordered_map
Separate Key / Value
no
yes
unordered_multiset
unordered_multimap
Allow Duplicates
no
yes
Separate Key / Value
yes
no
map
set
Separate Key / Value
yes
no
multimap
multiset

Practice 3

Write a program that asks the user **Type a name (or "exit")**

If that name was already typed in, display the associated phone number.

Otherwise ask for a number to be typed in and store the name and number. Then ask for another name until the user types "exit".

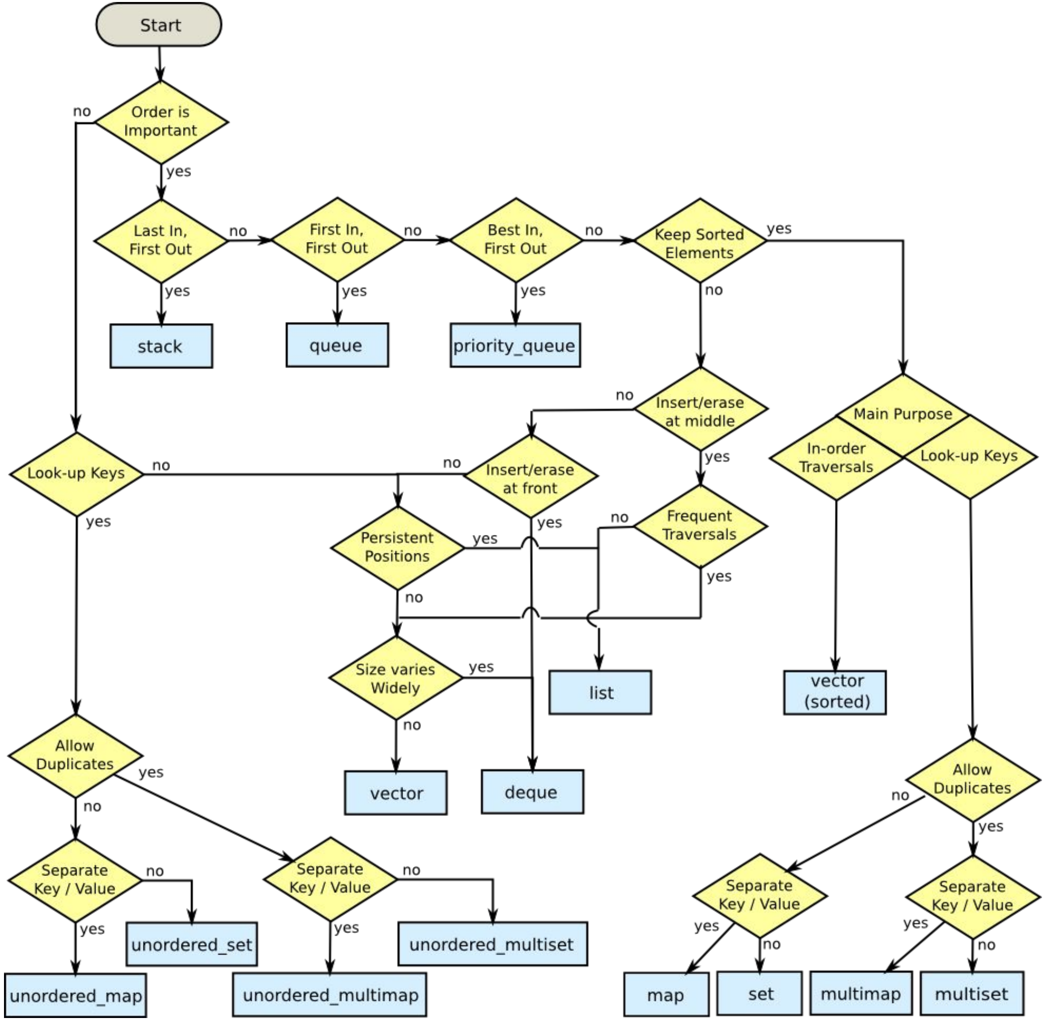
When “exit” is typed, print all names alphabetically and their corresponding phone number.

std::map<string,string> addressbook

Practice 4

Read a text from a file.The text contains several words. You will have to count the occurances of each word and figure out what are the 5 most frequently used words.

1.) In which container will you store the text?
2.) In which container will you store the counting of the words?


Start
Order is Important
no
yes
Last In, First Out
no
yes
stack
First In, First Out
no
yes
queue
Best In, First Out
no
yes
priority_queue
Keep Sorted Elements
yes
no
Main Purpose
In-order Traversals
Look-up Keys
vector (sorted)
Insert/erase at middle
no
yes
Look-up Keys
no
yes
Insert/erase at front
no
yes
Frequent Traversals
no
yes
Persistent Positions
yes
no
list
Size varies Widely
yes
no
vector
deque
Allow Duplicates
yes
no
Separate Key / Value
no
yes
unordered_set
unordered_map
Separate Key / Value
no
yes
unordered_multiset
unordered_multimap
Allow Duplicates
no
yes
Separate Key / Value
yes
no
map
set
Separate Key / Value
yes
no
multimap
multiset

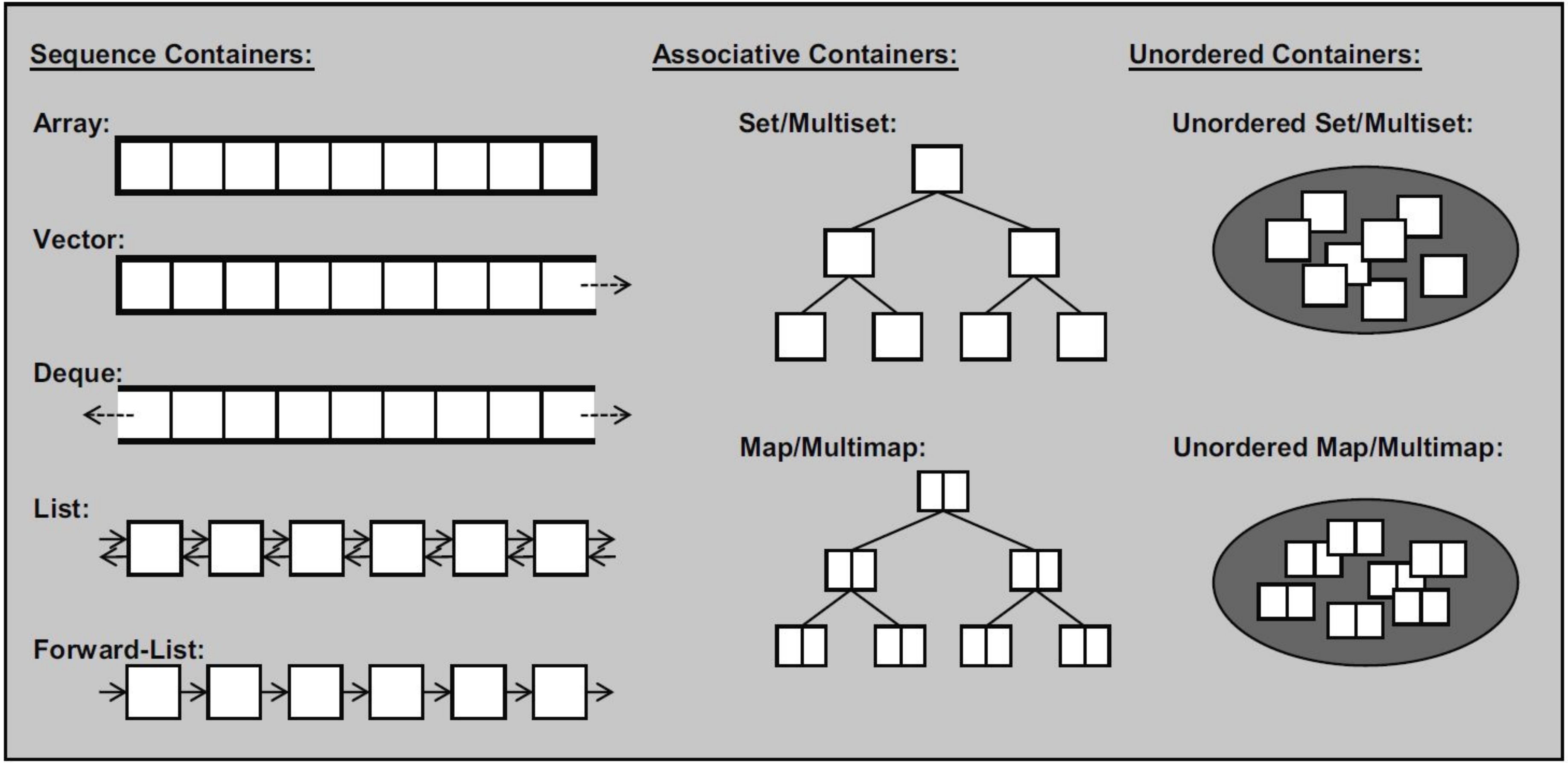
Practice 4

Read a text from a file.The text contains several words. You will have to count the occurances of each word and figure out what are the 5 most frequently used words.

1.) In which container will you store the text?
2.) In which container will you store the counting of the words?

1. std::vector<std::string>

2. std::unordered_map<std::string, int>



Sequence Containers:

Array:

Vector:

Deque:

List:

Forward-List:

Associative Containers:

Set/Multiset:

Map/Multimap:

Unordered Containers:

Unordered Set/Multiset:

Unordered Map/Multimap:

Contact

Andrea Tamas

Software development engineer @ Siemens Industry Software

andrea.tamas@outlook.com