C++ Data Structures

Nexus team

Mahdi Akbari Zarkesh

CONTENTS

01

Order

The time complexity of an algorithm is the total amount of time required by an algorithm to complete its execution.

Practice

```
int n;
cin>>n;
for (int i=0;i<n;i++)
    cout<<n;
```

O(N)

```
int n;
cin>>n;
for (int i=0;i<n;i++)
    for (int j=0;j<n;j++)
        cout<<n;
```

O(N^2)

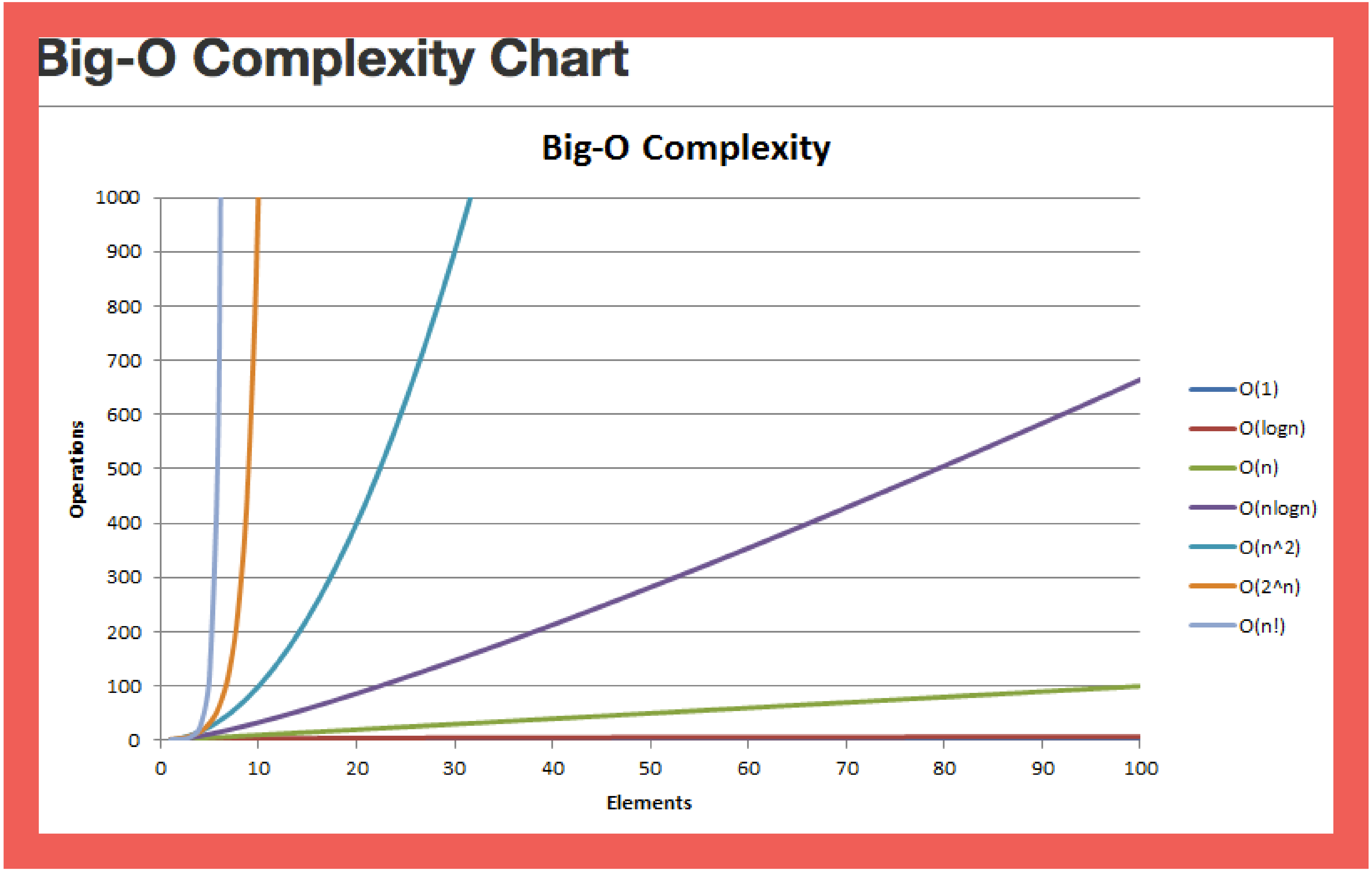
```
int N;
cin>>N;
int a = 0, i = N;
while (i > 0) {
    a += i;
    i /= 2;
}
```

O($log(n)$)

```
int N;
cin>>N;
int a = 0, b = 0;
for (i = 0; i < N; i++) {
    a = a + rand();
}
for (j = 0; j < N; j++) {
    b = b + rand();
}
```

O(N)

Big-O



Big-O Complexity Chart
Big-O Complexity
1000
900
800
700
600
500
400
300
200
100
0
Operations
0
10
20
30
40
50
60
70
80
90
100
Elements
O(1)
O(logn)
O(n)
O(nlogn)
O(n^2)
O(2^n)
O(n!)

Data Structure

Data Structure	Time Complexity								Space Complexity
	Average				Worst				Worst
	Indexing	Search	Insertion	Deletion	Indexing	Search	Insertion	Deletion	
Basic Array	O(1)	O(n)	-	-	O(1)	O(n)	-	-	O(n)
Dynamic Array	O(1)	O(n)	O(n)	O(n)	O(1)	O(n)	O(n)	O(n)	O(n)
Singly-Linked List	O(n)	O(n)	O(1)	O(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Doubly-Linked List	O(n)	O(n)	O(1)	O(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Skip List	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)	O(n)	O(n)	O(n)	O(n log(n))
Hash Table	-	O(1)	O(1)	O(1)	-	O(n)	O(n)	O(n)	O(n)
Binary Search Tree	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)	O(n)	O(n)	O(n)	O(n)
Cartresian Tree	-	O(log(n))	O(log(n))	O(log(n))	-	O(n)	O(n)	O(n)	O(n)
B-Tree	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)
Red-Black Tree	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)
Splay Tree	-	O(log(n))	O(log(n))	O(log(n))	-	O(log(n))	O(log(n))	O(log(n))	O(n)
AVL Tree	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)

02

C++ Data Structures

Vector std::vector

Use for

- Simple storage
- Adding but not deleting
- Serialization
- Quick lookups by index
- Easy conversion to C-style arrays
- Efficient traversal (contiguous CPU caching)

Do not use for

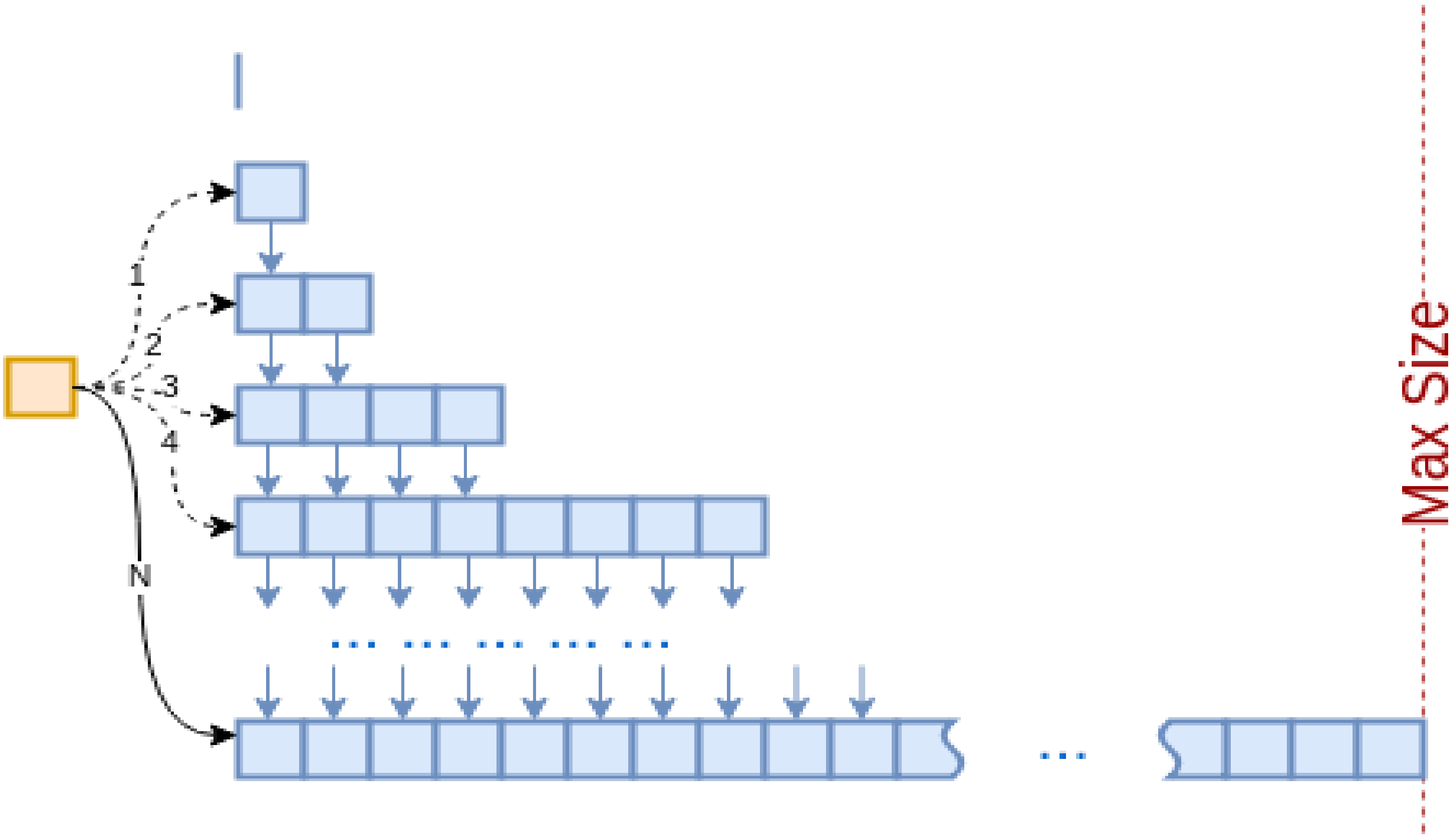
- Insertion/deletion in the middle of the list
- Dynamically changing storage
- Non-integer indexing

Time Complexity

Operation	Time Complexity
Insert Head	O(n)
Insert Index	O(n)
Insert Tail	O(1)
Remove Head	O(n)
Remove Index	O(n)
Remove Tail	O(1)
Find Index	O(1)
Find Object	O(n)

```
std::vector<int> v;
// Insert head, index, tail
v.insert(v.begin(), value);             // head
v.insert(v.begin() + index, value);     // index
v.push_back(value);                     // tail
// Access head, index, tail
int head = v.front();        // head
int value = v.at(index);     // index
int tail = v.back();         // tail
// Size
unsigned int size = v.size();
// Iterate
for(std::vector<int>::iterator it = v.begin(); it != v.end(); it++) {
    std::cout << *it << std::endl;
}
// Remove head, index, tail
v.erase(v.begin());              // head
v.erase(v.begin() + index);      // index
v.pop_back();                    // tail
// Clear
v.clear();
```

Vector std::vector



It's a single contiguous storage (a 1d array). Each time it runs out of capacity it gets reallocated and stored objects are moved to the new larger place — this is why you observe addresses of the stored objects changing.

It has always been this way, not since C++17.

Use for

- Insertion into the middle/beginning of the list
- Efficient sorting (pointer swap vs. copying)

Do not use for

- Direct access

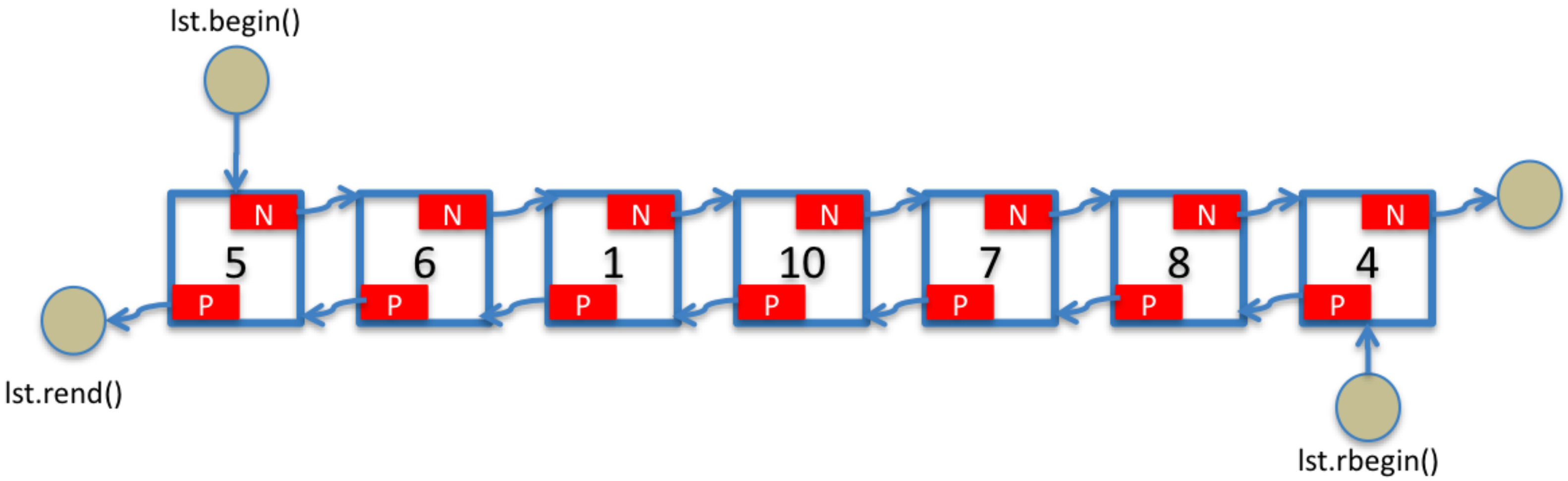
Time Complexity

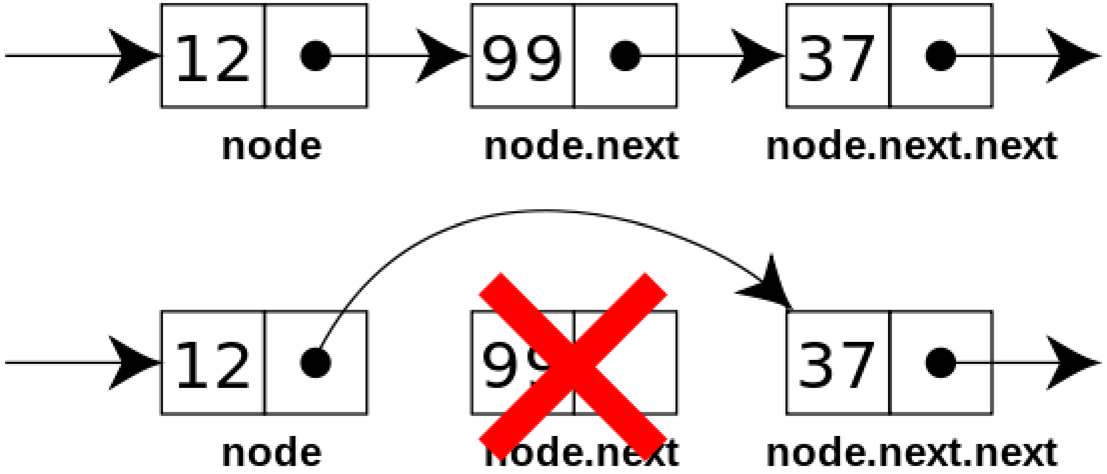
Operation	Time Complexity
Insert Head	O(1)
Insert Index	O(n)
Insert Tail	O(1)
Remove Head	O(1)
Remove Index	O(n)
Remove Tail	O(1)
Find Index	O(n)
Find Object	O(n)

```
std::list<int> l;
// Insert head, index, tail
l.push_front(value);                    // head
l.insert(l.begin() + index, value);     // index
l.push_back(value);                     // tail
// Access head, index, tail
int head = l.front();                                          // head
int value = std::next(l.begin(), index);                   // index
int tail = l.back();                                           // tail
unsigned int size = l.size();// Size
// Iterate
for(std::list<int>::iterator it = l.begin(); it != l.end(); it++) {
    std::cout << *it << std::endl;
}// Remove head, index, tail
l.pop_front();                    // head
l.erase(l.begin() + index);       // index
l.pop_back();                     // tail
l.clear();// Clear
// Splice: Transfer elements from list to list
l.splice(l.begin() + index, list2);
l.remove(value);// Remove: Remove an element by value
l.unique();// Unique: Remove duplicates
l.merge(list2);// Merge: Merge two sorted lists
l.sort();// Sort: Sort the list
l.reverse();// Reverse: Reverse the list order
```

List std::list and std::forward_list

List std::list





List stores elements at non contiguous memory location i.e. it internally uses a doubly linked list i.e.

Map std::map and std::unordered_map

Use for

- Key-value pairs
- Constant lookups by key
- Searching if key/value exists
- Removing duplicates
- std::map
- Ordered map
- std::unordered_map
- Hash table

Do not use for

- Sorting

Time Complexity

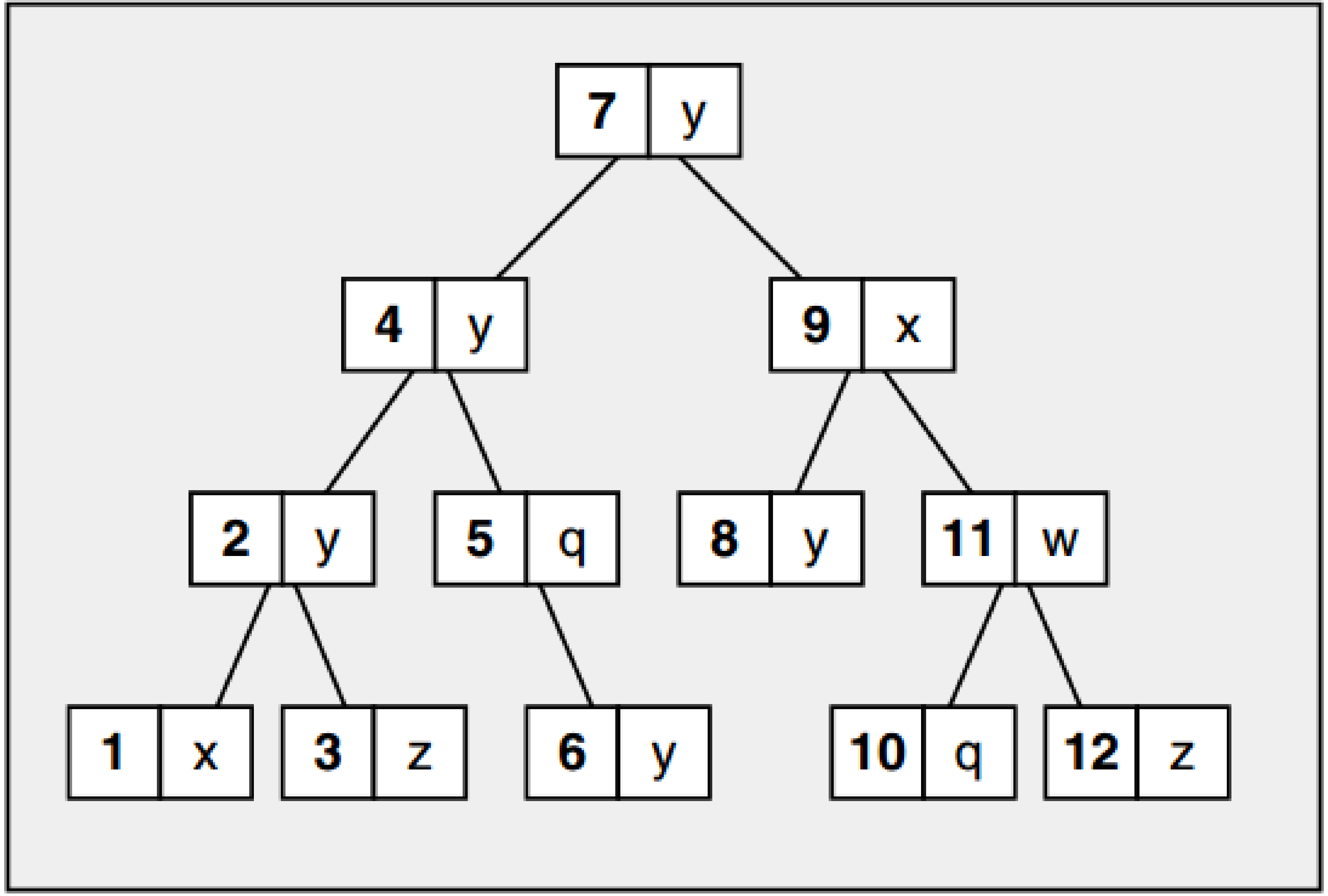
std::map

Operation	Time Complexity
Insert	O(log(n))
Access by Key	O(log(n))
Remove by Key	O(log(n))
Find/Remove Value	O(log(n))

std::unordered_map

Operation	Time Complexity
Insert	O(1)
Access by Key	O(1)
Remove by Key	O(1)
Find/Remove Value	--

```
std::map<std::string, std::string> m;
m.insert(std::pair<std::string, std::string>("key", "value"));// Insert
std::string value = m.at("key");// Access by key
unsigned int size = m.size();// Size
// Iterate
for(std::map<std::string, std::string>::iterator it = m.begin(); it != m.end(); it++) {
    std::cout << *it << std::endl;
}
// Remove by key
m.erase("key");
// Clear
m.clear();
// Find if an element exists by key
bool exists = (m.find("key") != m.end());
// Count the number of elements with a certain key
unsigned int count = m.count("key");
```



Map std::map

Maps and multimaps sort their elements automatically according to the element's keys. Thus they have good performance when searching for elements that have a certain key. Searching for elements that have a certain value promotes bad performance.

Note:multimaps allow duplicates, whereas maps do not

Use for

- Removing duplicates
- Ordered dynamic storage

Do not use for

- Simple storage
- Direct access by index

Notes

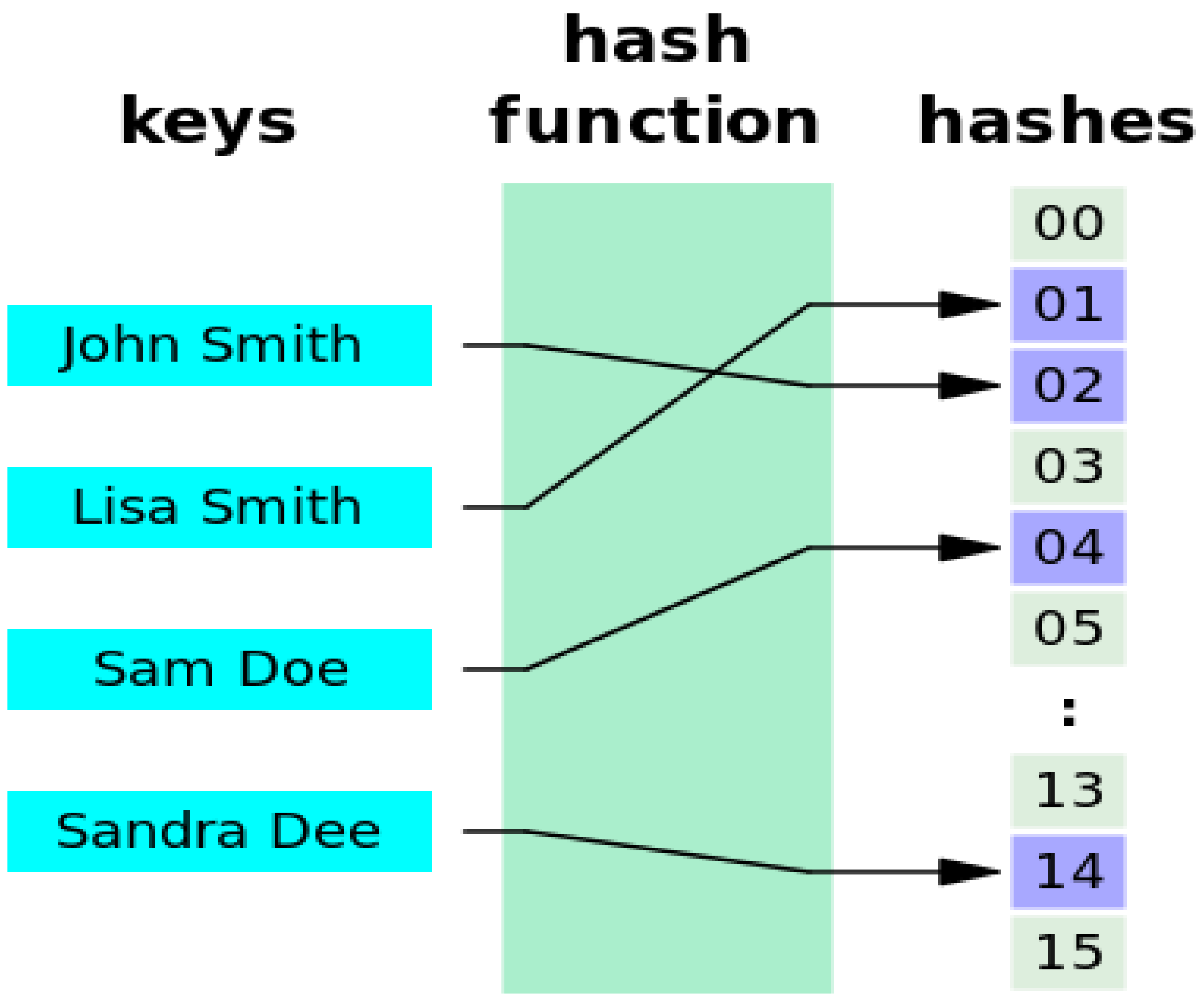
- Sets are often implemented with binary search trees

Time Complexity

Operation	Time Complexity
Insert	O(log(n))
Remove	O(log(n))
Find	O(log(n))

```
std::set<int> s;
s.insert(20);// Insert
unsigned int size = s.size();// Size
// Iterate
for(std::set<int>::iterator it = s.begin(); it != s.end(); it++) {
    std::cout << *it << std::endl;
}
s.erase(20);// Remove
s.clear();// Clear
// Find if an element exists
bool exists = (s.find(20) != s.end());
// Count the number of elements with a certain value
unsigned int count = s.count(20);
```

Set std::set



Set std::unordered_set

hash_set is an extension that is not part of the C++ standard. Lookups should be O(1) rather than O(log n) for set, so it will be faster in most circumstances.

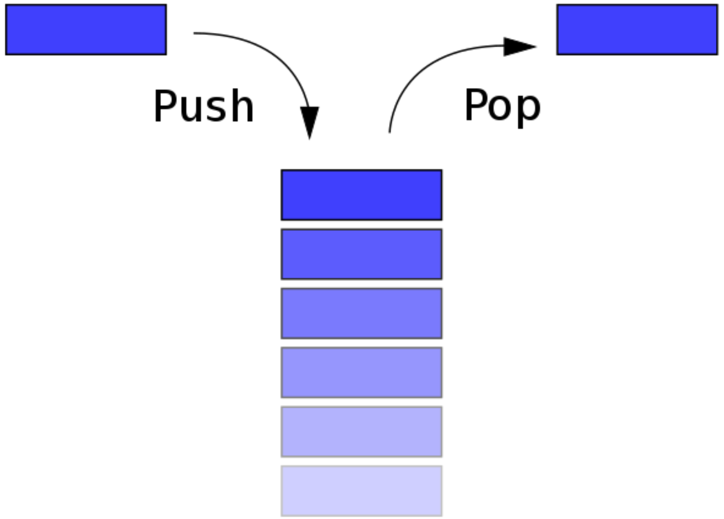
Before C++11 :stl::set is implemented as a binary search tree. hashset is implemented as a hash table.

Use for

- First-In Last-Out operations
- Reversal of elements

Time Complexity

Operation	Time Complexity
Push	O(1)
Pop	O(1)
Top	O(1)




```cpp
std::stack<int> s;
// Push
s.push(20);
// Size
unsigned int size = s.size();
// Pop remove last element :)
s.pop();
// Top
int top = s.top();
```

Stack std::stack

Use for

- First-In First-Out operations
- Ex: Simple online ordering system (first come first served)
- Ex: Semaphore queue handling
- Ex: CPU scheduling (FCFS)

Notes

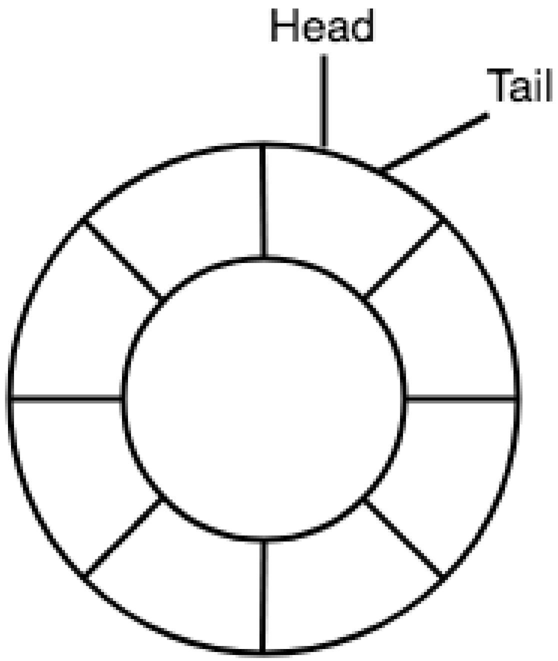
- Often implemented as a std::deque

```
std::queue<int> q;
// Insert
q.push(value);
//Access head, tail
int head = q.front();          // head
int tail = q.back();           // tail
// Size
unsigned int size = q.size();
// Remove
q.pop();
```

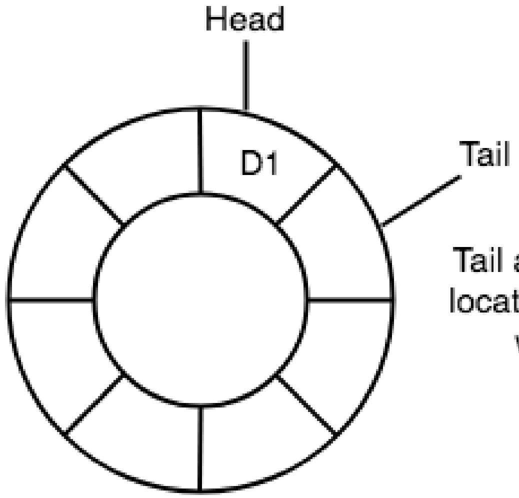
Queue std::queue

Queue std::queue

Initially the queue is empty, as Head and Tail are at same location



A simple circular queue with size 8



Tail always points to the location where new data will be inserted.

Circular Queue is also a linear data structure, which follows the principle of FIFO(First In First Out), but instead of ending the queue at the last position, it again starts from the first position after the last, hence making the queue behave like a circular data structure.

Use for

- Similar purpose of std::vector
- Basically std::vector with efficient push_front and pop_front

Do not use for

- C-style contiguous storage (not guaranteed)

Notes

- Pronounced 'deck'
- Stands for Double Ended Queue

```
std::deque<int> d;
// Insert head, index, tail
d.push_front(value);                          // head
d.insert(d.begin() + index, value);           // index
d.push_back(value);                           // tail
// Access head, index, tail
int head = d.front();         // head
int value = d.at(index);      // index
int tail = d.back();          // tail
// Size
unsigned int size = d.size();
// Iterate
for(std::deque<int>::iterator it = d.begin(); it != d.end(); it++) {
    std::cout << *it << std::endl;
}
// Remove head, index, tail
d.pop_front();                    // head
d.erase(d.begin() + index);       // index
d.pop_back();                     // tail
// Clear
d.clear();
```

Deque std::deque

Use for

- First-In First-Out operations where priority overrides arrival time
- Ex: CPU scheduling (smallest job first, system/user priority)
- Ex: Medical emergencies (gunshot wound vs. broken arm)

Notes

- Often implemented as a std::vector

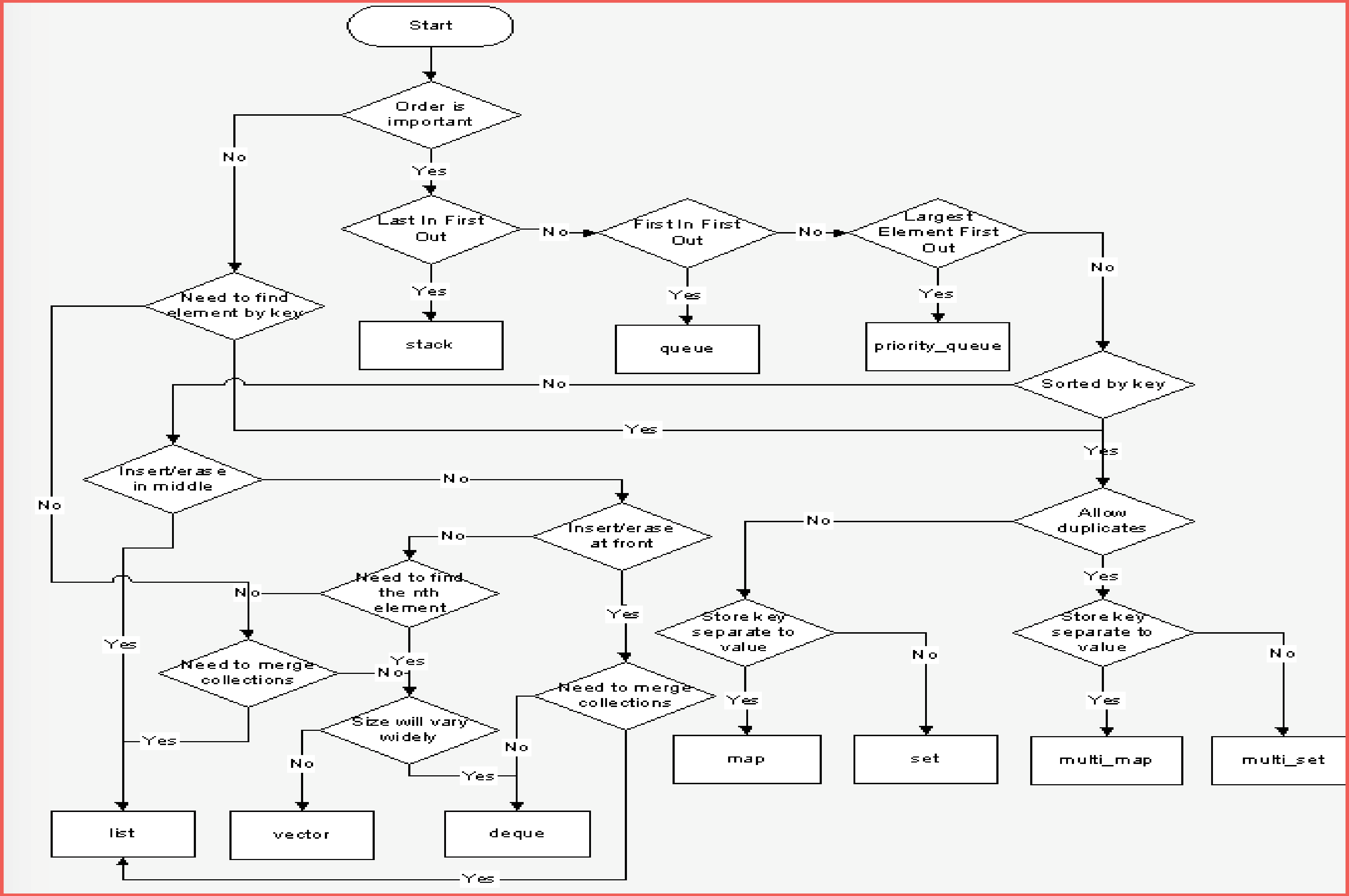
```cpp
std::priority_queue<int> p;
// Insert
p.push(value);
// Access
int top = p.top();  // 'Top' element
unsigned int size = p.size();// Size
p.pop();// Remove

auto cmp = [](int left, int right) { return (left ^ 1) < (right ^ 1); };
std::priority_queue<int, std::vector<int>, decltype(cmp)> q3(cmp);
for(int n : {1,8,5,6,3,4,0,9,7,2})
    q3.push(n);
```

Priority Queue std::priority_queue

03

Conclusion


Start
Order is important
No
Yes
Last In First Out
No
First In First Out
No
Largest Element First Out
No
Yes
Yes
Yes
stack
queue
priority_queue
Need to find element by key
Sorted by key
No
Yes
Yes
Insert/erase in middle
No
Allow duplicates
No
Insert/erase at front
No
No
Need to find the nth element
No
Yes
Yes
Yes
Store key separate to value
No
Store key separate to value
No
Need to merge collections
No
Yes
Need to merge collections
Size will vary widely
Yes
No
Yes
No
Yes
Yes
map
set
multi_map
multi_set
list
vector
deque
Yes

THANK YOU

Nexus team

Mahdi Akbari Zarkesh