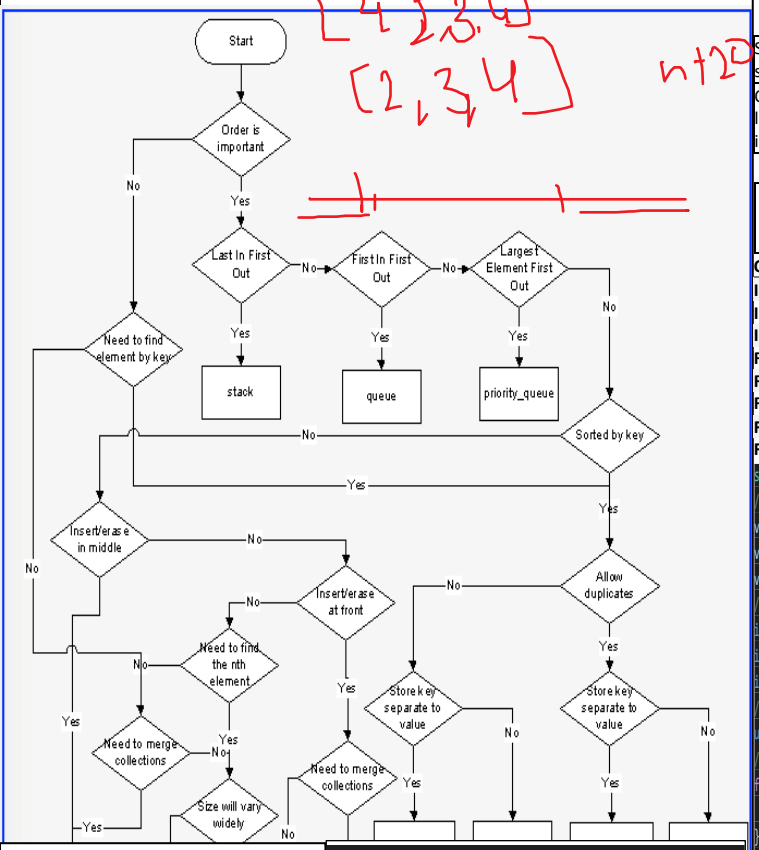
C++ STL Cheat Sheet

Complexities
Data Structures

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)

STL Choices



Deque
Use for

Similar to Vector	Basically vector with efficient push_front and pop_front

Do not use for

C-style contiguous storage (not guaranteed)

```
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;
}
```

Stack
Use for

FILO implementation	Reversal of elements

OPERATION	TIME COMPLEXITY
PUSH	O(1)
POP	O(1)
TOP	O(1)

```
stack<int> s;
// Container-Specific Operations
// Push
s.push(20);
// Size
unsigned int size = s.size();
// Pop
s.pop();
// Top
int top = s.top();
```

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.

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();
```

List/ Forward List
Use for

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

Do not use for

Efficient sorting (pointer swap vs. copying)

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
// Size
unsigned int size = l.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
// Clear
l.clear();
// Container-Specific Operations
//------------------------------
// Splice: Transfer elements from list to list
// splice(iterator pos, list &x) // splice(iterator pos, list &x, iterator i)
// splice(iterator pos, list &x, iterator first, iterator last)
.splice(l.begin() + index, list2);
// Remove: Remove an element by value
.remove(value);
// Unique: Remove duplicates
.unique();
// Merge: Merge two sorted lists
.merge(list2);
// Sort: Sort the list
.sort();
// Reverse: Reverse the list order
.reverse();
```

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

OPERATION	TIME COMPLEXITY
STD::MAP	
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	
INSERT	O(1)
ACCESS BY KEY	O(1)
REMOVE BY KEY	O(1)
FIND/REMOVE VALUE	--

```
map<string, string> m;
// Insert
m.insert(pair<string, string>("key", "value"));
// Access by key
string value = m.at("key");
// Size
unsigned int size = m.size();
// Iterate
for(map<string, string>::iterator it = m.begin(); it != m.end(); it++)
    cout << *it << endl;
// Remove by key
m.erase("key");
// Clear
m.clear();

// Container-Specific Operations
// Find if an element exists by key
bool exists = (m.find("key") != m.end());
// Count the number of elements with a certain key
```

Set
Use for

Removing Duplicates	Ordered dynamic storage

Do not use for

Simple storage and direct access by index

OPERATION	TIME COMPLEXITY
INSERT	O(log n)
REMOVE	O(log n)
FINE	O(log n)

```
set<int> s;
// General Operations
// Insert
s.insert(20);
// Size
unsigned int size = s.size();
// Iterate
for(set<int>::iterator it = s.begin(); it != s.end(); it++)
    cout << *it << endl;
// Remove
s.erase(20);
// Clear
s.clear();

// Container-Specific Operations
// 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);
```

Queue
Use for

FIFO implementation	deque also there

```
queue<int> q;
// General Operations // 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();
```

Priority Queue
Use for

FIFO operations where **priority** overrides arrival time

```
priority_queue<int> p;
// General Operations // Insert
p.push(value); //Larger value inserted at start
// Access
int top = p.top();  // 'Top' element
// Size
unsigned int size = p.size();
// Remove
p.pop();

// Min heap PQ
priority_queue <int, vector<int>, greater<int>> p;
p.push(x);
p.pop();
```