Data structures n' containers in C++

ARRAY: #include **<vector>**

push_back( ) pop_back( ) $O(1)$ operator [ ]	begin( ) end( ) insert( ) $O(n)$ erase( )	size() empty()

LIST: #include **<list>**

push_front( ) push_back( ) $O(1)$ pop_back( ) pop_front( )	begin( ) end( ) insert( ) $O(n)$ erase( )	size() empty()

STACK: #include **<stack>**

push( ) pop( ) $O(1)$ top( )		size() empty()

QUEUE: #include **<queue>**

push( ) pop( ) $O(1)$ front( ) back( )		size() empty()

MAX BINARY HEAP: #include **<priority_queue>**

push( ) $O(\log n)$ pop( )	top() $O(1)$	size() empty()

BINARY SEARCH TREE: #include **<set>** or **<map>**

insert( ) erase( ) $O(\log n)$ operator [ ] (map only)	begin( ) end( ) $O(\log n)$ find( ) *return iterator or set::end* count( ) *return 0 or 1*	size() empty()

Nota: if you want to insert several times the same value prefer to use **<multiset>** or **<multimap>** but operator [] will be not available anymore.

HASH TABLE: #include **<unordered_set>** or **<unordered_map>**

insert( ) $O(1)$ erase( ) operator [ ] (unordered_map only)	begin( ) end( ) $O(1)$ find( ) *return iterator or set::end* count( ) *return 0 or 1* reserve( ) *set the number of buckets*	size() empty()

Nota: hash_table is a very efficient data structure but elements can not be ordered.

For the following data structures there is no base container implemented in C++ so you have to do by hand.

MIN BINARY HEAP: *the package <algorithm> contains a function make_heap() to make a binary heap and by default this is a max binary heap.*

```
std::vector<int> minHeap;
std::make_heap(minHeap.begin(), minHeap.end(), std::greater<int>());
// to add a value
minHeap.push_back(x);
push_heap(minHeap.begin(), minHeap.end(), std::greater<int>());
// to remove
pop_heap(minHeap.begin(), minHeap.end(), std::greater<int>());
minHeap.pop_back();
```

BINARY TREE: *a simple binary tree is not very useful because you often need some properties (ordered, max value at root) which force you to use a <set> or <map> or <priority_queue>. If however you still want to implement one the easiest way is maybe to use a simple <vector> (like binary heaps) and to move through the tree using i *2 to get the left child and i *2 + 1 to get the right child. To insert an element you can perform a push_back() so your tree will be balanced. If you don't want this behavior we can use a struct node with pointers and implement a function to insert and delete an element:*

```
class Node
{
    int key_value;
    node *left;
    node *right;
};
```

GRAPH/N-ARIES TREE: *are nodes identified using index from 0 to N?*

- *YES => the node index (vertex) will match the vector index*

```
vector<vector<int> > graph;
            or
vector<vector<pair<int, int> > > graph; //edges have weight
```

- *NO => you have to create dedicated classes*

```
class Edge;
class Vertex
{
    int id;
    vector<Edge*> ptr_edges;
}
class Edge
{
     Vertex *from;
     Vertex *to;
     int cost;
}
class Graph
{
     vector<Vertex> vertices;
}
```