Basic Data Structures

Ivan Yahir Gómez Mancilla

Contents

- Data structure definition
- Vector
- Deque
- List
- Map
- Unordered map
- Set
- Queue
- Priority Queue

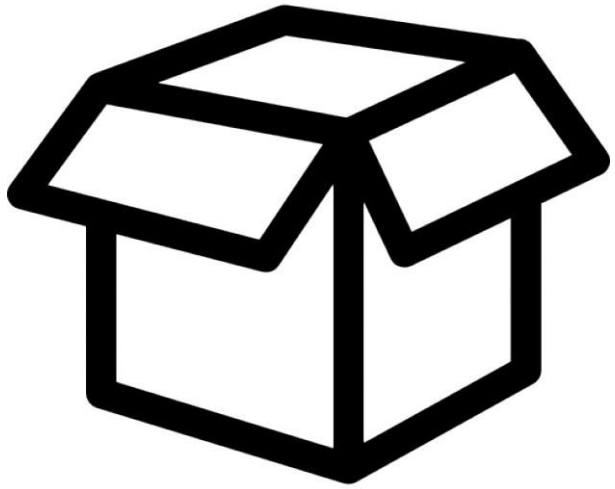
Data structure definition

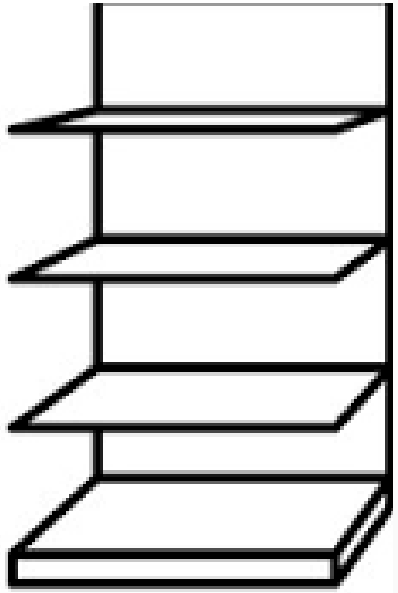
- Introduction to Algorithms (MIT Press):
  - A data structure is a way of organizing and storing data in a computer so that it can be accessed and modified efficiently. More precisely, a data structure is a collection of data values, the relationships among them, and the functions or operations that can be applied to the data.

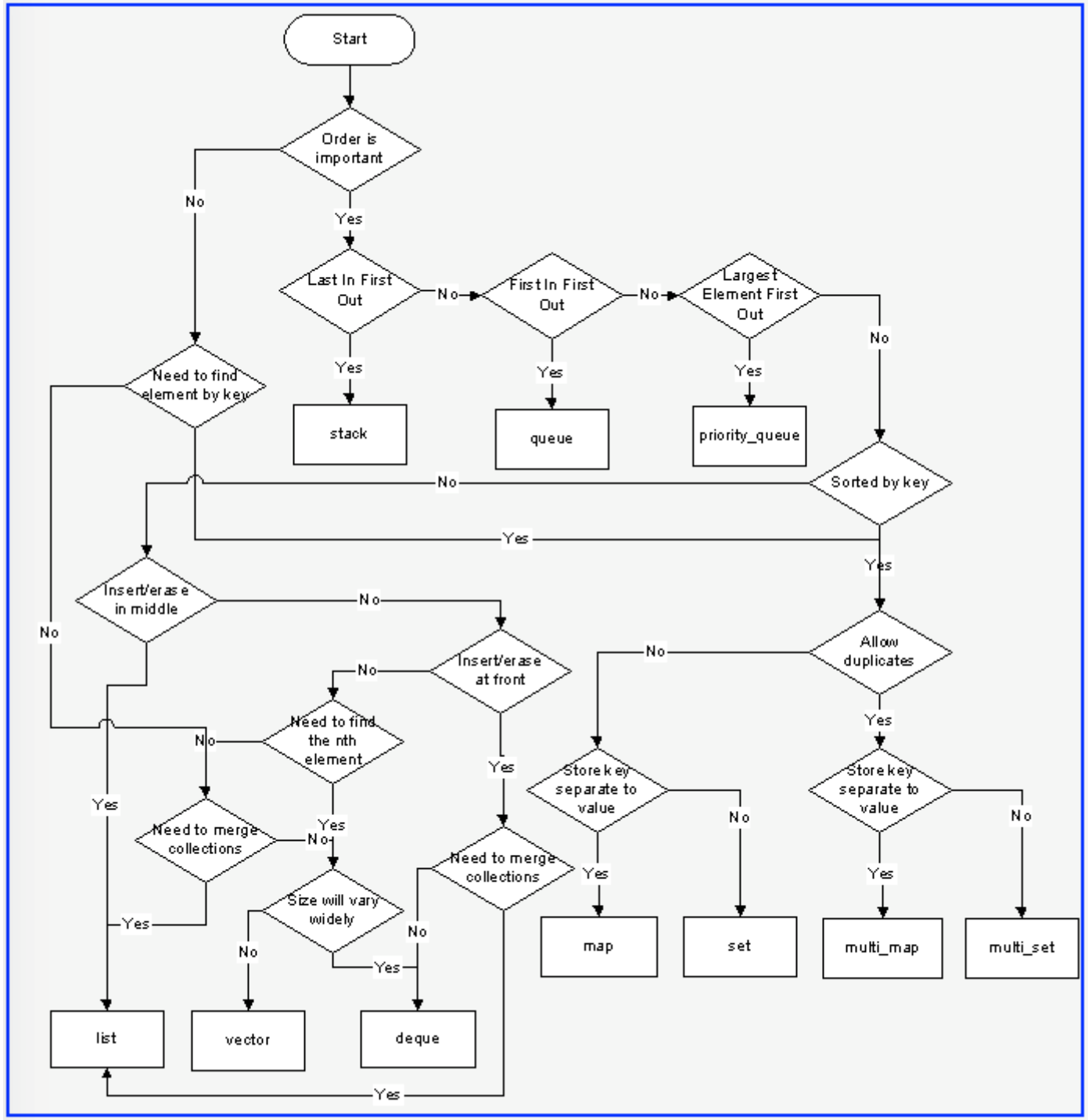
- Data Structures in C++ (Adam Drozdek):
  - A data structure is a way of organizing data in a computer so that it can be used efficiently. It is a collection of data values, the relationships among them, and the functions or operations that can be applied to the data.

- Competitive Programming 4 (Steven and Felix Halim):
  - A data structure is a way of organizing data in a computer so that it can be accessed and manipulated efficiently. It is a collection of data values, the relationships among them, and the functions or operations that can be applied to the data. In competitive programming, data structures are often used to optimize algorithms for speed and memory usage.

Data structures in real life






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
No
Sorted by key
Yes
Yes
Insert/erase in middle
No
No
Insert/erase at front
No
Allow duplicates
No
Yes
Need to find the nth element
No
Yes
Store key separate to value
No
Store key separate to value
No
Yes
Yes
Yes
Need to merge collections
No
Yes
Need to merge collections
Yes
No
Yes
Size will vary widely
No
Yes
map
set
multi_map
multi_set
No
Yes
list
vector
deque
Yes

Iterator (C++)

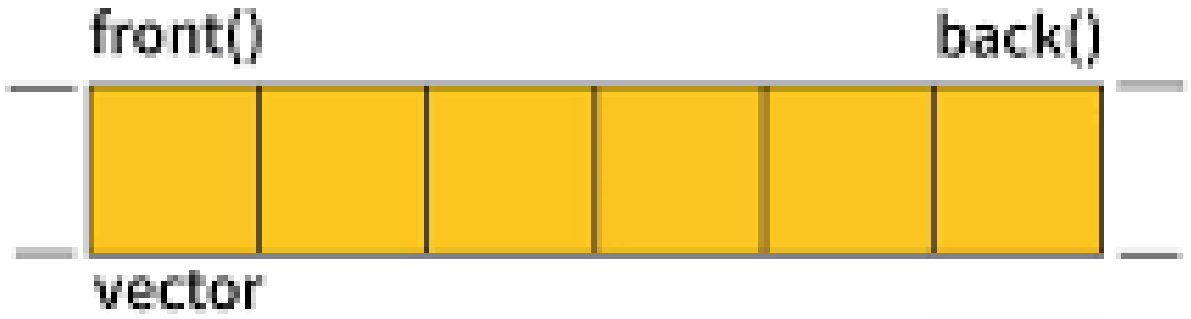
- An iterator in C++ is an object that is used to traverse a container, such as an array or a vector, and access the elements within it. The C++ Standard Template Library (STL) provides a set of iterator types that can be used to traverse different types of containers.
- STL iterators are objects that provide a way to access the elements of a container without exposing the underlying data structure. They act as a generalized pointer to the elements of a container and allow for generic algorithms to operate on different types of containers.

Iterator functions

containerType<_valueType_>::iterator _iteratorName_	Creates an iterator
front_inserter(_containerName_) = _valueToInsert_	Inserts a value at first position of the container.
back_inseter(_containerName_) = _valueToInsert_	Insert a value at the end of the container
containerName.begin()	Returns an iterator to the first position of a container
containerName.end()	Returns an iterator to the next position of the final position of container
containerName.rbegin()	Returns a begin iterator in reverse order
containerName.rend()	Returns an end iterator in reverse order
containerName.front()	Returns the first element from a container
containerName.back()	Returns the last element from a container
containerType<_containerValueTyper_>::difference_type _variableName_ = distance(_firstContainerIterator_, _lastContainerIterator_)	Returns the number of increases first iterator needs before being equal to last iterator.
empty(_containerName_)	Returns true if container is empty, else false.
inserter(_containerName_, _containerIteratorPosition_) = _valueToInsert_	Inserts a value into a specified position
containerName.size()	Returns the size of the container
containerName.clear()	Clears totally a data structure

Vector

- A vector data structure is a linear data structure that is used to store a collection of elements. It is also known as a dynamic array. Unlike traditional arrays, a vector can dynamically resize itself when new elements are added to it. This makes it more flexible and convenient to use in many programming applications.



Usage

- Use for:
  - Simple storage
  - Adding but not deleting
  - Quick lookups by index
- Do not use for:
  - Insertion/deletion in the middle of the list
  - Dynamically changing storage

Time Complexity

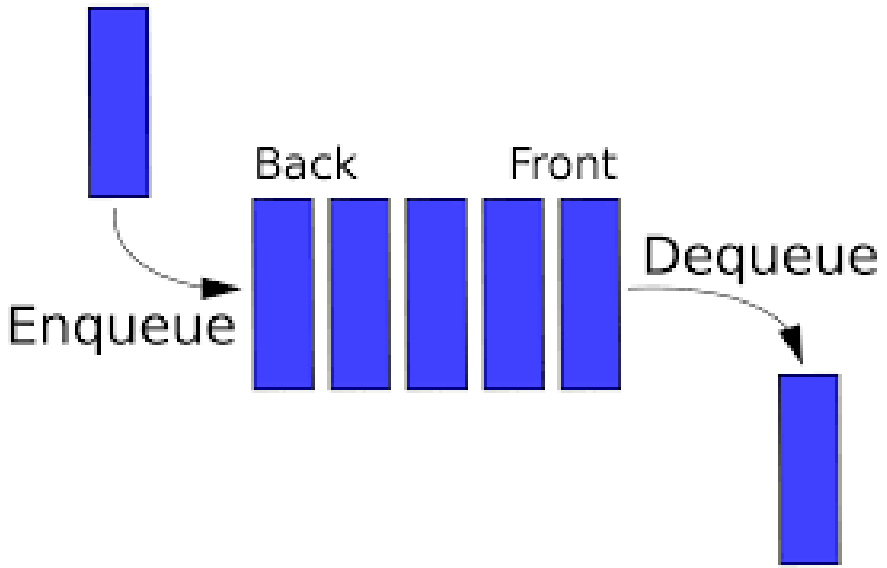
Operation	Time Complexity
Insert Head	`O(n)`
Insert Index	`O(n)`
Insert Tail	`O(1)`
Remove Head	`O(n)`
Remove Index	`O(n)`
Remove Tail	`O(1)`
Find Index	`O(1)`
Find Object	`O(n)`

Vector functions

vector <_valueType_> _vectorName_(_*size*_, _*initialValue*_)	Creates an empty vector
vectorName.resize(_size_, _initialValue_)	Resizes a vector with an specified size and an initial value
vectorName.shrink_to_fit()	deletes oversized capacity on a vector
vectorName.assign(_firstIterator_, _lastIterator_)	Assign vector values to another one
vectorName.assign(_numberValues_, _value_)	Assign an specified number of values with a specified value to a vector
vectorName.at(_index_)	Returns the value situated at a specified index

Queue

- In computer science, a queue is a linear data structure that represents a sequence of elements, where the elements are inserted at one end (the back) and removed from the other end (the front).



Usage

- FIFO Operations
  - online ordering system
  - CPU scheduling

Time complexity

Linked-List Based Queue

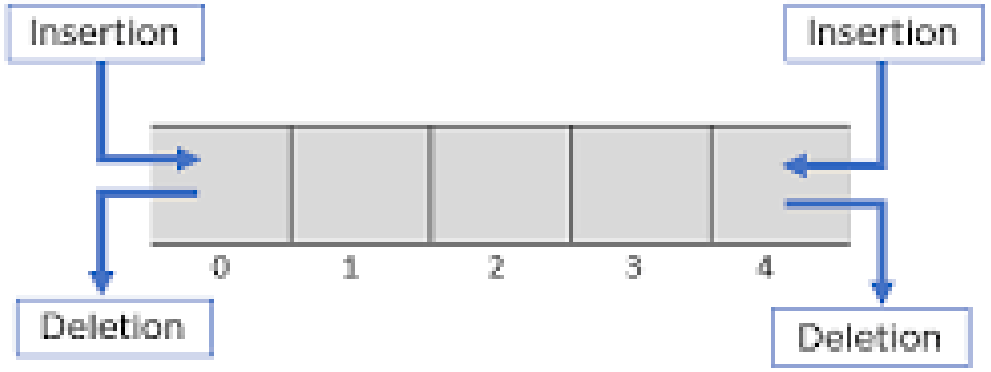
Operations	Average case	Worst case
enqueue	O(1)	O(1)
dequeue	O(1)	O(1)
Space Complexity	O(n)	O(n)

Queue Operations

queue <_valueType_> _queueName_	Creates an empty queue
queueName.pop()	Removes first element from a queue
queueName.push(_value_)	Pushes a value at the back of the queue

Double-ended queue (Deque)

- linear data structure in computer science that represents a collection of elements arranged in a specific order. It is similar to a queue in that it supports adding and removing elements, but it also supports adding and removing elements from both ends of the deque.



Usage

- Use for
  - Similar purpose of std::vector
  - Basically std::vector with efficient push_front and pop_front
  - Do not use for
- Do not use for
  - C-style contiguous storage (not guaranteed)

Time Complexity

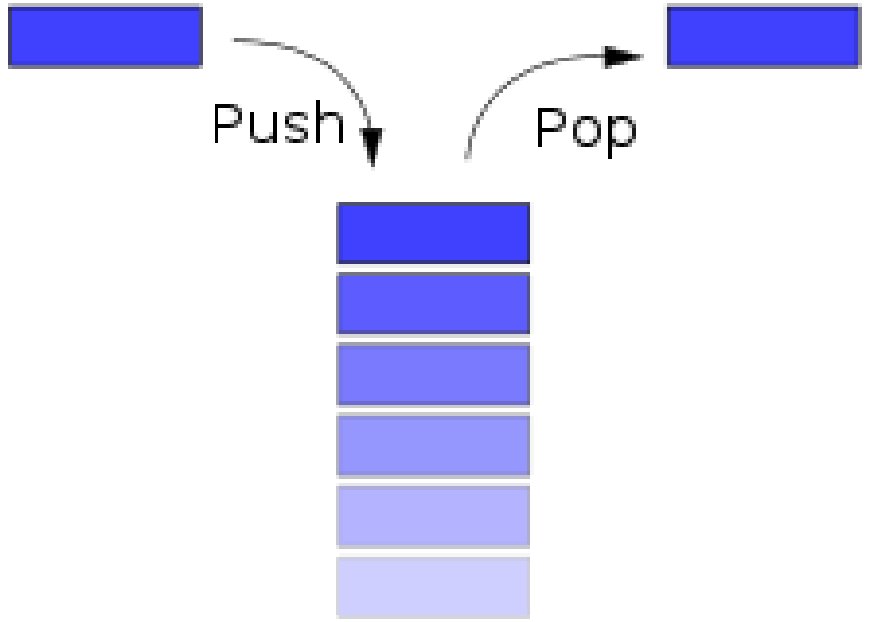
Operation	Time Complexity
Insert Head	`O(1)`
Insert Index	`O(n) or O(1)`
Insert Tail	`O(1)`
Remove Head	`O(1)`
Remove Index	`O(n)`
Remove Tail	`O(1)`
Find Index	`O(1)`
Find Object	`O(n)`

Deque funcitons

deque<_valueType_> _dequeName_	Creates an empty deque
dequeName.push_front(_valueToPush_)	Pushes a value at the front of the deque
dequeName.pop_front()	Pops the value from the front of the deque
dequeName.push_back(_valueToPush_)	Pushes a value at the back of the deque
dequeName.pop_back()	Pops the value from the back of the deque
dequeName.erase(_iterator_)	Erases deque's iterator position

Stack

- linear data structure in which elements are added and removed from the top, known as the "top of the stack." It follows the Last-In-First-Out (LIFO) principle, which means that the last element added to the stack is the first element to be removed.



Usage

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

Time Complexity

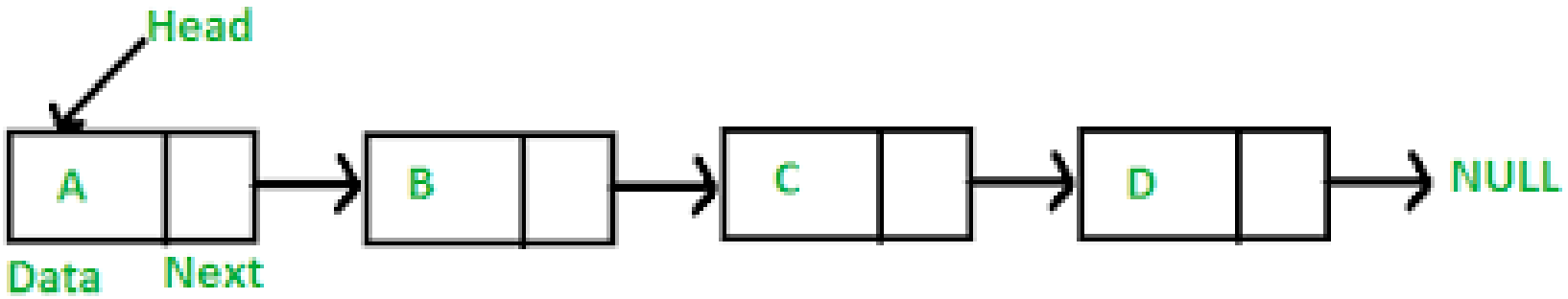
Operation	Time Complexity
Push	`O(1)`
Pop	`O(1)`
Top	`O(1)`

Stack functions

stack <_valueType_> _stackName_	Creates an empty stack
stackName.pop()	Pops the top element on stack
stackName.push(_value_)	Pushes a top element on a stack
stackName.top()	Returns the top element from a stack

List

- collection of elements or items that are stored in a specific order. Each item in the list is identified by its position or index in the list.



Usage

- Use for
  - Insertion into the middle/beginning of the list
  - Efficient sorting (pointer swap vs. copying)
- Do not use for
  - Direct access

Time Complexity

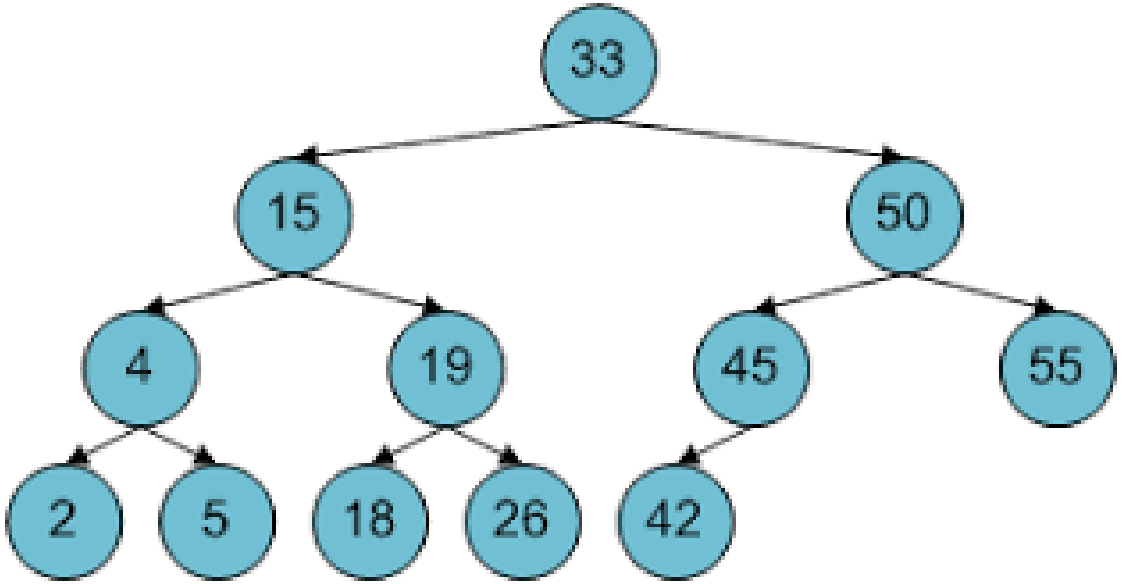
Operation	Time Complexity
Insert Head	`O(1)`
Insert Index	`O(n)`
Insert Tail	`O(1)`
Remove Head	`O(1)`
Remove Index	`O(n)`
Remove Tail	`O(1)`
Find Index	`O(n)`
Find Object	`O(n)`

List functions

list <_valueType_> _listName_;	Creates an empty list
listName.assign(_list2IteratorFirst_, _list2IteratorLast_)	Assigns a section of a list to another one
listName.assign(_numberOfvalues_, _value_)	Assigns n specified equal values to a list
listName.assign(_value_)	Assigns a value into a list
list1Name.merge(_list2Name_, _*compareFunction*_)	Merge a second list into first one and sort elements in increasing order
listName.insert(_positionIterator_, _value_)	Inserts a value into an specified position
listName.push_front(_valueToPush_)	Pushes a value at the front of the list
listName.pop_front()	Pops the value from the front of the list
listName.push_back(_valueToPush_)	Pushes a value at the back of the list
listName.pop_back()	Pops the value from the back of the list
listName.remove(_value_)	Removes all values in a list that matches the specified one
listName.revome_if(_compareFunction_)	Removes every element that makes unary function returns true
listName.resize(_newSize_, _*value*_)	Resizes the list and new spaces added are filled with an specified value.
listName.reverse()	Reverses list order
listName.unique(_*compareFunction*_)	removes adyacent equal (or any condition specified by a binary predicate) values from a list.
list1Name.splice(_positionIterator_, _list2Name_, _*valueList2Iterator*_)	inserts a value from a list or the whole list into another list at an specified position
list1Name.splice(_positionIterator_, _list2Name_, _list2FirstIterator_, _list2LastIterator_)	Inserts a range of values from a list into another at an specified position.

Set

- collection of unique and unordered elements or items. In other words, a set is a data structure that contains only distinct elements, with no repeated values. The elements in a set can be of any data type, such as integers, strings, or even objects.



Usage

- Use for
  - Removing duplicates
  - Ordered dynamic storage
- Do not use for
  - Simple storage
  - Direct access by index

Time Complexity

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

Set functions

set <_valueType_> _setName_	Creates an empty set
setName.contains(_value_)	Returns true if an specified value is contained at set
setName.find(_value_)	Returns position of a found specified value, else .end()
setName.insert(_value_)	Inserts a value into an specified postion of a set
setName.lower_bound(_value_)	Returns an iterator with the first value of a set that is greater that or equal to an specified value
setName.upper_bound(_value_)	Returns an iterator with the frist value of a set that is less than or equal to a specified value

Pair

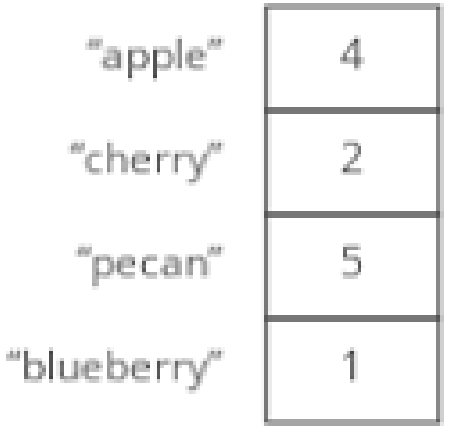
- simple data structure in C++ that consists of two elements, known as the first and second values. Each value in a pair can be of any data type, and the two values do not have to be of the same data type.

Pair functions

pair <_firstValueType_, _lastValueType_) _pairName_	Creates an empty pair
make_pair(_firstValue_, _secondValue_)	Creates a pair with two values
pairName.first	Returns first value of the pair
pairName.second	Returns second value of the pair

Map

- collection of key-value pairs, where each key is associated with a value. It is also known as a dictionary or associative array in some programming languages. The key-value pairs are stored in such a way that the keys are unique and the values can be accessed or updated using their associated keys.



Usage

- Use for
  - Key-value pairs
  - Constant lookups by key
  - Searching if key/value exists
  - Removing duplicates
- Do not use for
  - Sorting

Time Complexity

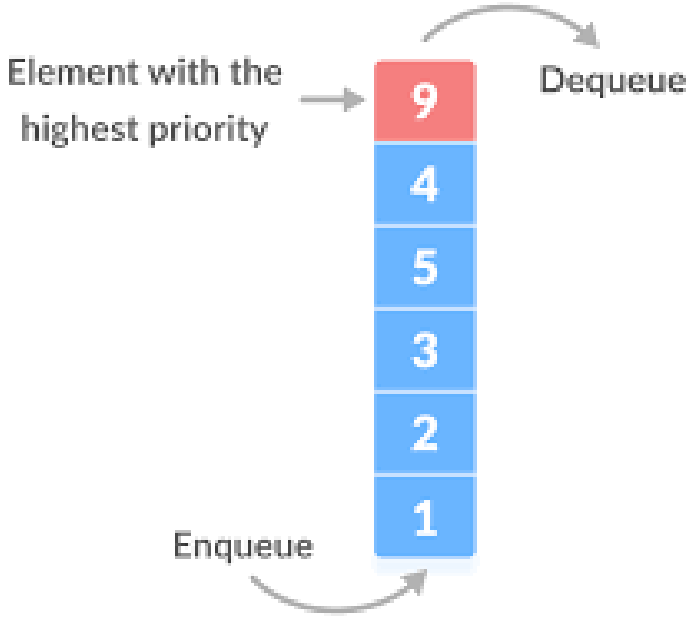
Operation	Time Complexity
Insert	`O(log(n))`
Access by Key	`O(log(n))`
Remove by Key	`O(log(n))`
Find/Remove Value	`O(log(n))`

Map functions

map <_keyType_, _valueType_> _mapName_	Creates an empty map
mapName.insert(pair <_keyValue_, _valueValue_>)	Inserts a value at a map
mapName.count(_keyValue_)	Counts number of values stores with a certain key (returns 1 or 0)
mapName.contains(_keyValue_)	Returns boolean if finds a key with the specified value
mapName.erase(_keyValue_)	Erases a key value and its assigned value
mapName.find(_keyValue_)	returns an iterator pointing to the finded value if it is finded, else returns .end()

Priority Queue (heap)

- abstract data type that stores a collection of elements, each of which has an associated priority. The elements in a priority queue are sorted based on their priority, so that the element with the highest priority is always at the front of the queue.



Time complexity

Procedure	Binary heap (worst-case)
MAKE-HEAP	$\Theta(1)$
INSERT	$\Theta(\lg n)$
MINIMUM	$\Theta(1)$
EXTRACT-MIN	$\Theta(\lg n)$
UNION	$\Theta(n)$
DECREASE-KEY	$\Theta(\lg n)$
DELETE	$\Theta(\lg n)$

Priority queue functions

priority_queue<_valueType_, _*specificContainer*_, _*compareFunction*_> _priorityQueueName_	creates an empty priority queue
priorityQueueName.push(_value_)	Pushes a value based on its priority
priorityQueueName.pop()	Pops the first priority value from a priority queue
priorityQueueName.top()	returns the first priority value from a priority queue