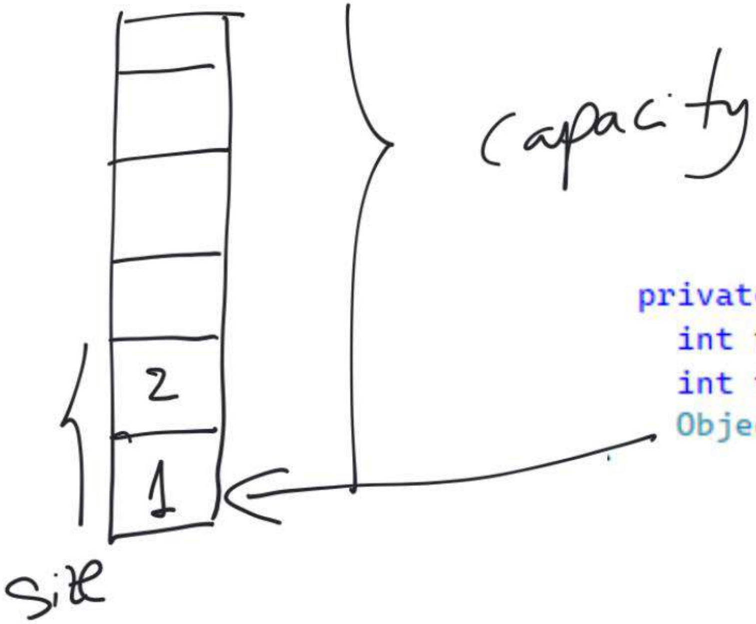
Qué sabemos:
- Algoritmos
- Órdenes de los algoritmos
- DRY
- Algoritmos + estructuras de datos = programas
- Pilas
- Colas
- Calculadora postfija

Hoy:
- Necesidad de entender los requisitos
- Vector
- Size vs Capacity
- Iteradores
- Programar

Common Data Structure Operations

Data Structure	Time Complexity								Space Complexity
	Average				Worst				Worst
	Access	Search	Insertion	Deletion	Access	Search	Insertion	Deletion	
Array	Θ(1)	Θ(n)	Θ(n)	Θ(n)	O(1)	O(n)	O(n)	O(n)	O(n)
Stack	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Queue	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Singly-Linked List	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Doubly-Linked List	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Skip List	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(n)	O(n)	O(n)	O(n)	O(n log(n))
Hash Table	N/A	Θ(1)	Θ(1)	Θ(1)	N/A	O(n)	O(n)	O(n)	O(n)
Binary Search Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(n)	O(n)	O(n)	O(n)	O(n)
Cartesian Tree	N/A	Θ(log(n))	Θ(log(n))	Θ(log(n))	N/A	O(n)	O(n)	O(n)	O(n)
B-Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)
Red-Black Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)
Splay Tree	N/A	Θ(log(n))	Θ(log(n))	Θ(log(n))	N/A	O(log(n))	O(log(n))	O(log(n))	O(n)
AVL Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)
KD Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(n)	O(n)	O(n)	O(n)	O(n)

Guardaropa




```cpp
private:
  int theSize;
  int theCapacity;
  Object * objects;
```

```
public:
  explicit vector( int initSize = 0 ) : theSize( initSize ), theCapacity( initSize + SPARE_CAPACITY )
    { objects = new Object[ theCapacity ]; }
```

```
~vector( )
 { delete [ ] objects; }
```

```cpp
bool empty( ) const
  { return size( ) == 0; }
int size( ) const
  { return theSize; }
int capacity( ) const
  { return theCapacity; }
```

```cpp
template <class Object>
const Object & vector<Object>::back( ) const
{
    if( empty( ) )
        throw UnderflowException( "Cannot call back on empty vector" );
    return objects[ theSize - 1 ];
}
```

```
auto v = vector<int>();
v[7]
```

```cpp
Object & operator[]( int index )
{
                                                    #ifndef NO_CHECK
    if( index < 0 || index >= size( ) )
        throw ArrayIndexOutOfBoundsException( index, size( ) );
                                                    #endif
    return objects[ index ];
}
```

```cpp
template <class Object>
void vector<Object>::pop_back( )
{
    if( empty( ) )
        throw UnderflowException( "Cannot call pop_back on empty vector" );
    theSize--;
}
```

```
template <class Object>
void vector<Object>::push_back( const Object & x )
{
    if( theSize == theCapacity )
        reserve( 2 * theCapacity + 1 );
    objects[ theSize++ ] = x;
}
```

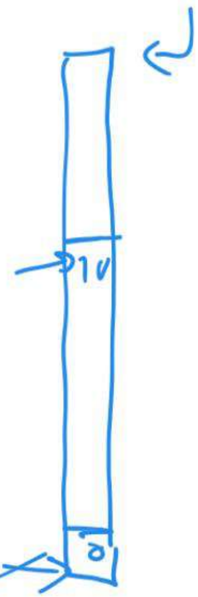
2+2


```
template <class Object>
void vector<Object>::reserve( int newCapacity )
{
    Object *oldArray = objects;

    int numToCopy = newCapacity < theSize ? newCapacity : theSize;
    newCapacity += SPARE_CAPACITY;

    objects = new Object[ newCapacity ];
    for( int k = 0; k < numToCopy; k++ )
        objects[ k ] = oldArray[ k ];

    theSize = numToCopy;
    theCapacity = newCapacity;

    delete [ ] oldArray;
}
```

→ otro puntero

T → numtocopy = newcap.

F → the size



```cpp
template <class Object>
void vector<Object>::resize( int newSize )
{
    if( newSize > theCapacity )
        reserve( newSize * 2 );
    theSize = newSize;
}
```

Algoritmos

Iteradores

Spotify

artista

album ⇐ vector<canciones>

Canciones

$n \times m \times o$

$n + m + o$

iterador ⟺ contenedor

auto i = v.begin()

begin(v)

```cpp
auto numeros = vector<int>(11,-1);
for (auto i = begin(numeros); i < numeros.end(); ++i) {
    cout << *i;
}

for (auto element : numeros) {
    cout << element;
}
auto result1 = std::find(begin(numeros), end(numeros), 1);
cout << ( result1 != std::end(numeros));
```

*i

int

1,7,4

begin

end

past the end

i

1	7	4	

4 (*i)

numeros[elemento]

```
  // Iterator stuff: not bounds checked
typedef Object * iterator;
typedef const Object * const_iterator;
```

```
iterator begin( )
  { return &objects[ 0 ]; }
const_iterator begin( ) const
  { return &objects[ 0 ]; }
iterator end( )
  { return &objects[ size( ) ]; }
const_iterator end( ) const
  { return &objects[ size( ) ]; }
```