Cheat Sheet

Gabriel

October 2024

El objetivo de este documento es entregar un resumen de diversas estructuras de datos y algoritmos implementadas en el lenguaje C++ y Python para los alumnos de la universidad técnica federico santa maría (aunque esto también debería servir para cualquier otra universidad).

Este documento es realizado con ningún intensivo monetario, es únicamente por el puro amor al arte. Se entiende que este documento podría tener errores (aunque no debería). Si alguien encuentra un error agradecería que abran un issue en el repositorio del cheat sheet (https://github.com/ProgramacionCompetitivaUTFSM/cheat_sheet_basic

Se insta fuertemente a los alumnos a revisar la documentación de los lenguajes para consultar más información con respecto a la implementación de estructuras de datos y algoritmos. Documentación C++: https://cplusplus.com/doc/; Documentación Python https://docs.python.org/3/.

Los contenidos de este documento vienen mayoritariamente de los cursos Estructuras de Datos y Lenguajes de Programación. Esto debido a que: (1) las estructuras mencionadas fueron estudiadas en el curso Estructuras de Datos; (2) poder aprender y entender un nuevo lenguaje gracias a su documentación se estudio es Lenguajes de Programación.

EDDs STL C++

vector

```cpp
// incluir vector
#include< vector >
// declaracion de vector que almacena tipo ‘T‘
vector< T > vec;
// iterador a un vector
// (es como un "puntero")
vector< T >::iterator itr;
// retorna iterador apuntando al primer elemento del vector
itr = vec.begin();
// retorna iterador apuntando al final del vector
// "justo despues del ultimo elemento"
itr = vec.end();
// retorna tamanio del vector
vec.size();
// agrega ‘ele‘ al final
// ‘ele‘ debe ser de tipo ‘T‘
vec.push_back(ele);
// elimina del final
vec.pop_back();
// inserta ‘ele‘ en la posicion p
// ‘ele‘ debe ser de tipo ‘T‘
vec.insert(vec.begin() + p, ele);
// elimina elemento de la posicion p
vec.erase(vec.begin() + p);
// retorna el valor almacenado en la posicion i
// 0 <= i < vec.size()
vec[i];
```

Complejidades temporales:

Operacion	Peor Caso
push_back	$\theta(1)$
pop_back	$\theta(1)$
insert	$\mathcal{O}(n)$
erase	$\mathcal{O}(n)$
[i]	$\theta(1)$

Complejidad espacial $\theta(n)$

stack

```cpp
// incluir stack
#include< stack >
// declaracion de stack que almacena tipo ‘T‘
stack< T > st;
// retorna tamanio del stack
st.size();
// inserta ‘ele‘ al tope del stack
// ‘ele‘ debe ser de tipo ‘T‘
st.push(ele);
// elimina del tope del stack
st.pop();
// retorna el elemento encima del stack
st.top();
```

Complejidad temporal: $\theta(1)$ todas las operaciones

Complejidad spacial: $\theta(n)$

queue

```cpp
// incluir queue
#include< queue >
// declaracion de queue que almacena tipo ‘T‘
queue< T > qu;
// retorna tamanio de la queue
qu.size();
// inserta ‘ele‘ al frente de la queue
// ‘ele‘ debe ser de tipo ‘T‘
qu.push(ele);
// elimina del final de la queue
qu.pop();
// retorna el elemento al frente de la queue
qu.front();
```

Complejidad temporal: $\theta(1)$ todas las operaciones

Complejidad spacial: $\theta(n)$

deque

```cpp
// incluir deque
#include< deque >
// declaracion de deque que almacena tipo ‘T‘
deque< T > dq;
// retorna tamanio de la deque
dq.size();
// inserta ‘ele‘ al frente de la deque
// ‘ele‘ debe ser de tipo ‘T‘
dq.push_front(ele);
// inserta ‘ele‘ al final de la deque
// ‘ele‘ debe ser de tipo ‘T‘
dq.push_back(ele);
// elimina del frente de la deque
dq.pop_front();
// elimina del final de la deque
dq.pop_back();
// retorna el elemento al frente de la deque
dq.front();
// retorna el elemento al frente de la deque
dq.back();
```

Complejidad temporal: $\theta(1)$ todas las operaciones

Complejidad spacial: $\theta(n)$

set

```
// incluir set
// set => ordered set
// o sea que esta ordenado
#include< set >
// declaracion de set que almacena tipo ‘T‘
set< T > st;
// iterador a un set
set< T >::iterator itr;
// retorna iterador aputnando al primer elemento del set
itr = st.begin();
// retorna iterador apuntando al final del set
// "justo despues del ultimo elemento"
itr = st.end();
// retorna tamanio del set
st.size();
// inserta ‘ele‘
// ‘ele‘ es de tipo ‘T‘
// si ya existe ‘ele‘ no hace nada
st.insert(ele);
// elimina ‘ele‘ del set
st.erase(ele);
// retorna un iterador elemento ‘ele‘
// si ‘ele‘ no esta retorna iterador al .end()
itr = st.find(k);
// retorna un iterador elemento con valor ‘ele2‘
// y ‘ele‘ <= ‘ele2‘, y no existe otra llave ‘ele3‘ tal que
// ‘ele‘ <= ‘ele3‘ <= ‘ele2‘
itr = st.lower_bound(ele);
// retorna un iterador al par que tenga como llave ‘ele2‘
// y ‘ele‘ < ‘ele2‘, y no existe otra llave ‘ele3‘ tal que
// ‘ele‘ < ‘ele3‘ < ‘ele2‘
itr = st.upper_bound(ele);
```

Complejidades temporales:

Operacion	Peor Caso
insert	$\mathcal{O}(\log n)$
erase	$\mathcal{O}(\log n)$
find	$\mathcal{O}(\log n)$
lower_bound	$\mathcal{O}(\log n)$
upper_bound	$\mathcal{O}(\log n)$

Complejidad espacial $\theta(n)$

unordered_set

```
// incluir unordered_set
// o sea que NO esta ordenado
#include< unordered_set >
// declaracion de set que almacena tipo ‘T‘
unordered_set< T > st;
// iterador a un set
unordered_set< T >::iterator itr;
// retorna tamanio del set
st.size();
// inserta ‘ele‘
// ‘ele‘ es de tipo ‘T‘
// si ya existe ‘ele‘ no hace nada
st.insert(ele);
// elimina ‘ele‘ del set
st.erase(ele);
// retorna un iterador elemento ‘ele‘
// si ‘ele‘ no esta retorna iterador al .end()
itr = st.find(k);
```

Complejidades temporales:

Operacion	Promedio	Peor Caso
insert	$\mathcal{O}(1)$	$\mathcal{O}(n)$
erase	$\mathcal{O}(1)$	$\mathcal{O}(n)$
find	$\mathcal{O}(1)$	$\mathcal{O}(n)$

Complejidad espacial $\theta(n)$

map

```
// incluir map
// map => ordered map
// o sea que esta ordenado
#include< map >
// declaracion de map que almacena
// llave tipo ‘T1‘
// valor tipo ‘T2‘
map< T1, T2 > mp;
// iterador a un map
map< T1, T2 >::iterator itr;
// retorna tamanio del map
mp.size();
// retorna iterador aputnando al primer elemento del map
itr = mp.begin();
// retorna iterador apuntando al final del map
// "justo despues del ultimo elemento"
itr = mp.end();
// inserta el par ‘k‘, ‘v‘
// ‘k‘ es la llave y de tipo ‘T1‘
// ‘v‘ es el valor y de tipo ‘T2‘
// si ya existe la llave no hace nada
mp.insert({k, v});
// elimina el par que tenga la llave ‘k‘
mp.erase(k);
// retorna un iterador al par con llave ‘k‘
// si ‘k‘ no esta retorna iterador al .end()
itr = mp.find(k);
// la llave
itr->first; // o *itr.first
// el valor
itr->second; // o *itr.second
// retorna un iterador al par que tenga como llave ‘k2‘
// y ‘k‘ <= ‘k2‘, y no existe otra llave ‘k3‘ tal que
// ‘k‘ <= ‘k3‘ <= ‘k2‘
itr = mp.lower_bound(k);
// retorna un iterador al par que tenga como llave ‘k2‘
// y ‘k‘ < ‘k2‘, y no existe otra llave ‘k3‘ tal que
// ‘k‘ < ‘k3‘ < ‘k2‘
itr = mp.upper_bound(k);
```

Complejidades temporales:

Operacion	Peor Caso
insert	$\mathcal{O}(\log n)$
erase	$\mathcal{O}(\log n)$
find	$\mathcal{O}(\log n)$
lower_bound	$\mathcal{O}(\log n)$
upper_bound	$\mathcal{O}(\log n)$

Complejidad espacial $\theta(n)$

unordered_map

```
// incluir unordered_map
// o sea que NO esta ordenado
#include< unordered_map >
// declaracion de map que almacena
// llave tipo ‘T1‘
// valor tipo ‘T2‘
unordered_map< T1, T2 > mp;
// iterador a un map
unordered_map< T1, T2 >::iterator itr;
// retorna tamanio del map
mp.size();
// inserta el par ‘k‘, ‘v‘
// ‘k‘ es la llave y de tipo ‘T1‘
// ‘v‘ es el valor y de tipo ‘T2‘
// si ya existe la llave no hace nada
mp.insert({k, v});
// elimina el par que tenga la llave ‘k‘
mp.erase(k);
// retorna un iterador al par con llave ‘k‘
// si ‘k‘ no esta retorna iterador al .end()
itr = mp.find(k);
// la llave
itr->first; // o *itr.first
// el valor
itr->second; // o *itr.second
```

Complejidades temporales:

Operacion	Promedio	Peor Caso
insert	$\mathcal{O}(1)$	$\mathcal{O}(n)$
erase	$\mathcal{O}(1)$	$\mathcal{O}(n)$
find	$\mathcal{O}(1)$	$\mathcal{O}(n)$

Complejidad espacial $\theta(n)$

priority_queue

```
// incluir queue
// dentro tiene la priority_queue
#include< queue >
// declracion de priority_queue que almacena tipo ‘T‘
// por default es un heap que te entrega el maximo
priority_queue< T > pq;
// hacer esto para que te entregue el minimo
priority_queue< T, vector< T >, greater< T > > pq2;
// retorna el tamanio de la priority_queue
pq.size();
// inserta ‘ele‘ a la priority_queue
// ‘ele‘ es de tipo ‘T‘
pq.insert(ele);
// elimina el elemento maximo/minimo de la priority_queue
pq.pop();
// obtiene el elemento maximo/minimo de la priority_queue
pq.top();
```

Complejidades temporales:

Operacion	Peor Caso
insert	$\mathcal{O}(\log n)$
erase	$\theta(1)$
top	$\theta(1)$

Complejidad espacial $\theta(n)$

Algoritmos STL C++

sort

```
// incluir sort
#include< algorithm >
// utilizando un vector que almacena ‘T‘
// ‘T‘ debe tener un metodo de comparacion
// si ‘T‘ es de la STL, deberia tener
vector< T > vec;
// sortear todo el vector
// por default ordena de menor a mayor
sort(vec.begin(), vec.end());
// sortear con funcion de comparacion propia
// ‘comp‘ debe recibir dos parametros tipo ‘T‘ (a, b)
// (recomendacion los parametros usarlos con
// & para que no se pasen como copia)
// comp debe retornar:
//  * ‘true‘: si ‘a‘ va a la izquierda de ‘b‘
//  * ‘falso‘: en caso contrario
// ejemplo de una funcion que ordena de mayor a menor
// bool comp(T &a, T &b) {
//   return a > b;
// }
sort(vec.begin(), vec.end(), comp);
// tambien con arreglos
T arr[n];
sort(arr, arr + n);
```

Complejidad espacial: $\mathcal{O}(1)$

Complejidad temporal: $\theta(\log n)$

lower_bound/upper_bound

```
// incluir lower_bound y upper_bound
#include< algorithm >
// ‘T‘ debe tener un metodo de comparacion
// si ‘T‘ es de la STL, deberia tener
vector< T > vec;
// ‘vec‘ debe estar ordenado
// retorna un iterador apuntando al elemento con valor ‘v2‘
// y ‘v‘ <= ‘v2‘, tal que no exista un elemento ‘v3‘ tal que
// ‘v‘ < ‘v3‘ <= ‘v2‘
lower_bound(vec.begin(), vec.end(), v);
// retorna un iterador apuntando al elemento con valor ‘v2‘
// y ‘v‘ < ‘v2‘, tal que no exista un elemento ‘v3‘ tal que
// ‘v‘ < ‘v3‘ <= ‘v2‘
upper_bound(vec.begin(), vec.end(), v);
```

Complejidad espacial: $\mathcal{O}(1)$

Complejidad temporal: $\theta(\log n)$

EDDs Python

list

```
# asignacion a una variable un list
l = []
# agrega ‘ele‘ al final
l.append(ele)
# inserta en la posicion i del list ‘ele‘
l.insert(i, ele)
# elimina del final
l.pop()
# elimina el elemento en posicion i
l.pop(i)
# accede al elemento en posicion i
l[i]
# retorna una copia de las referencias en la
# sublist de las pociciones i hasta j - 1 del list
l2 = l[i:j]
# retorna tamanio de un list
len(l)
# ordena el list de menor a mayor
l.sort()
# lo da vuelta
l.reverse()
# retorna ‘True‘ si ‘x‘ esta en la lista
# retorna ‘False‘ en caso contrario
x in l
```

Complejidades temporales:

Operacion	Peor Caso
append	$\theta(1)$
pop()	$\theta(1)$
insert	$\mathcal{O}(n)$
pop(i)	$\mathcal{O}(n)$
[i]	$\theta(1)$
[i:j]	$\theta(n)$
sort	$\mathcal{O}(\log n)$
reverse	$\theta(n)$

Complejidad espacial $\theta(n)$

dict

```
# asignacion a una variabe un dict
# dict -> dictionary (es como un unordered_map)
d = dict()
# asignacion a una llave un valor
# si la llave ya existe se reemplaza
d[k] = l
# elimina la llave ‘k‘ y retorna su valor
# si la llave no esta tira error
d.pop(k)
# retorna las llaves del dict
# no retorna una lista
d.keys()
# si quieren una lista hacer
list(d.keys())
# retorna los valores del dict
# no retorna una lista
d.values()
# si quieren una lista hacer
list(d.values())
# retorna tuplas (llave, valor) del dict
# no retorna una lista
d.items()
# si quieren una lista hacer
list(d.items())
```

Complejidades temporales:

Operacion	Promedio	Peor Caso
[k]	$\mathcal{O}(1)$	$\mathcal{O}(n)$
pop	$\mathcal{O}(1)$	$\mathcal{O}(n)$