Comparativa de la STL

Salvador Romero Cortés y Abel Ríos González

Se van a comparar los siguientes contenedores presentes en la STL

- `vector<int>`
- `list<int>`
- `set<int>`
- `unordered_set<int>`

Inserción de elementos

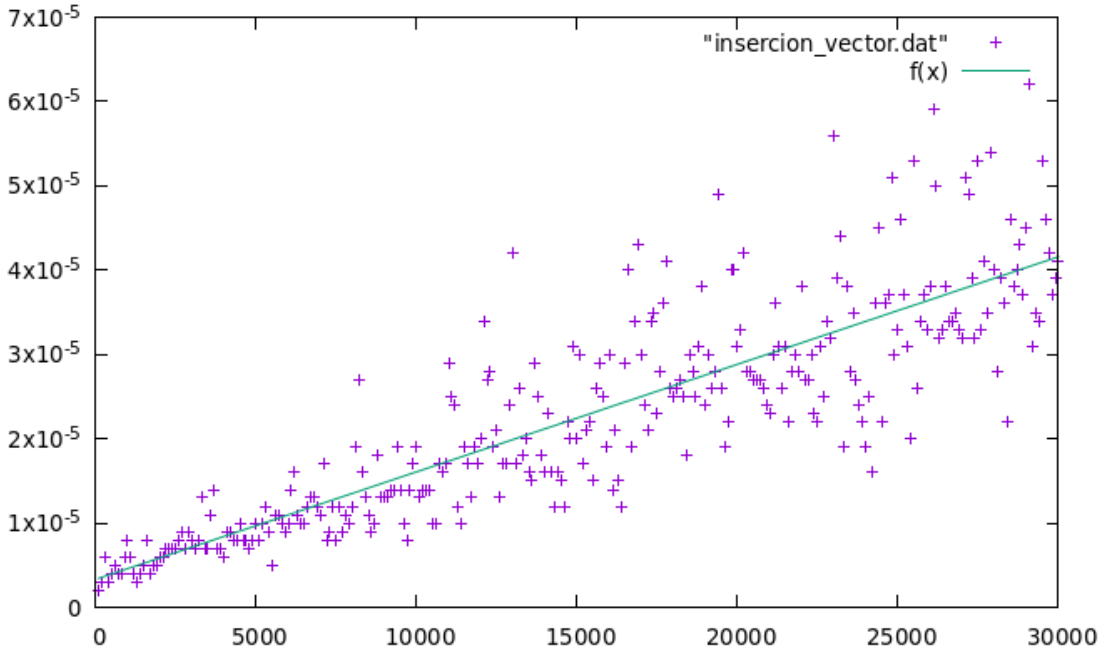
`push_back()` en el caso de `list` y de `vector`; `insert()` en el caso de `set` y de `unordered_set`

- Vector

  En el caso de `vector` cuando hacemos `push_back` pueden ocurrir dos cosas:

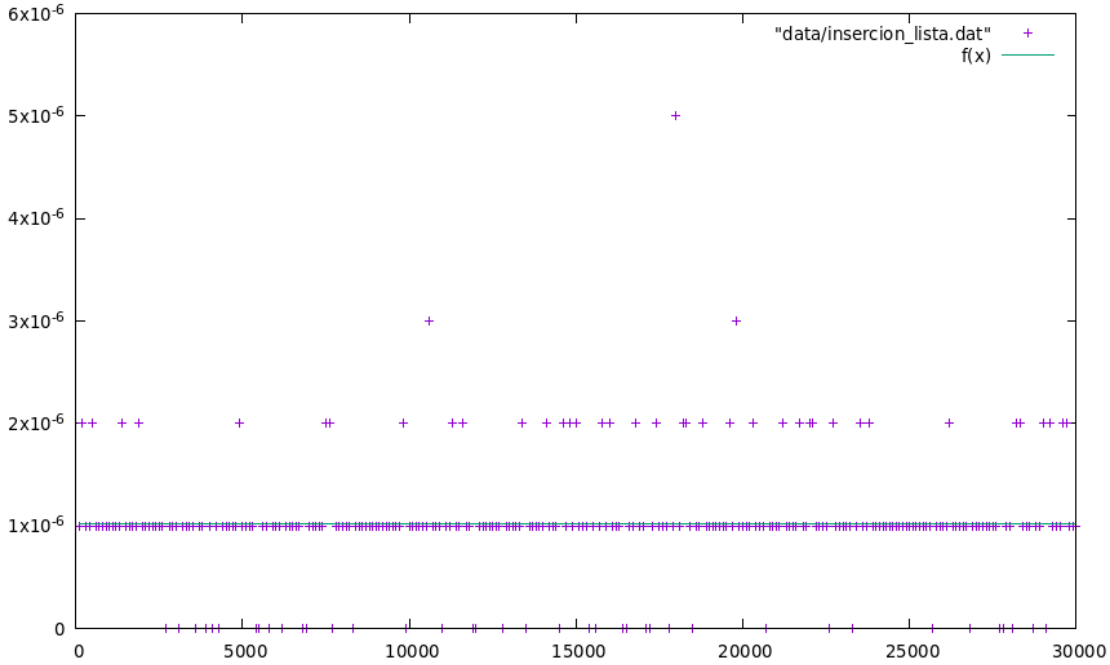
  - Primero, se tiene reservada ya la memoria necesaria por lo que simplemente se inserta en una posición de memoria $O(1)$.
  - O bien, no lo tiene reservado y tiene que redimensionarlo. Esto último es de tiempo $O(n)$.

  En nuestro ejercicio hemos forzado que se redimensione para ver el peor caso posible. Reservamos la memoria para N elementos y luego añadimos uno extra.
  De esta manera si representamos gráficamente los tiempos de inserción vemos que aumenta linealmente con el tamaño del vector.



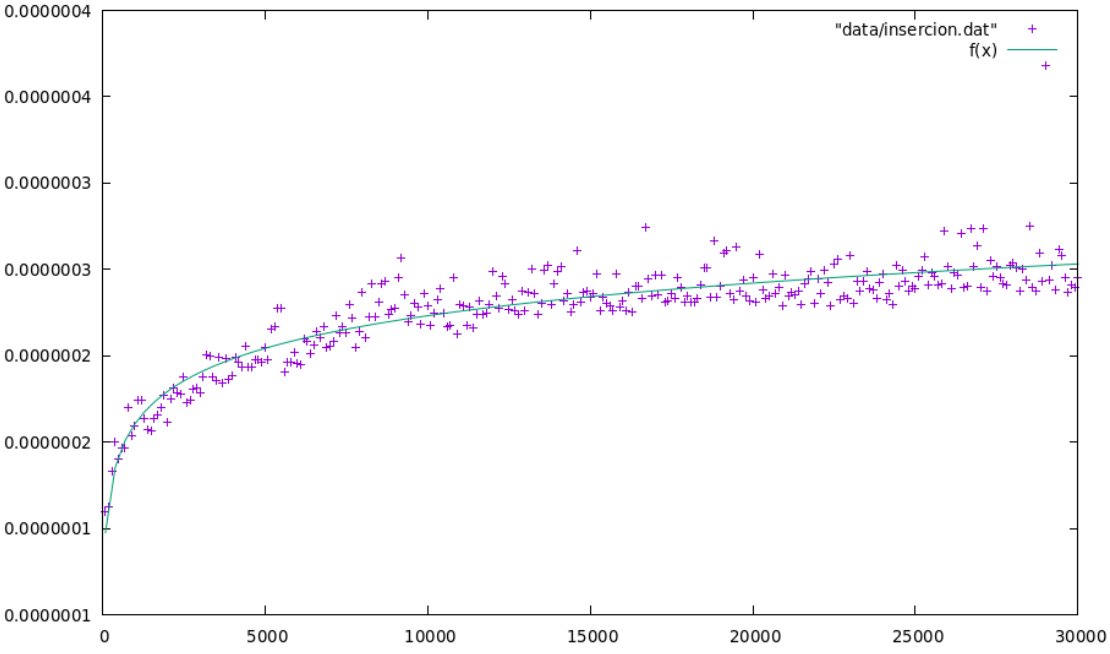
- List

En el caso de `list`, las inserciones son de tiempo constante siempre, es decir $O(1)$. Por tanto, el gráfico es una función constante.



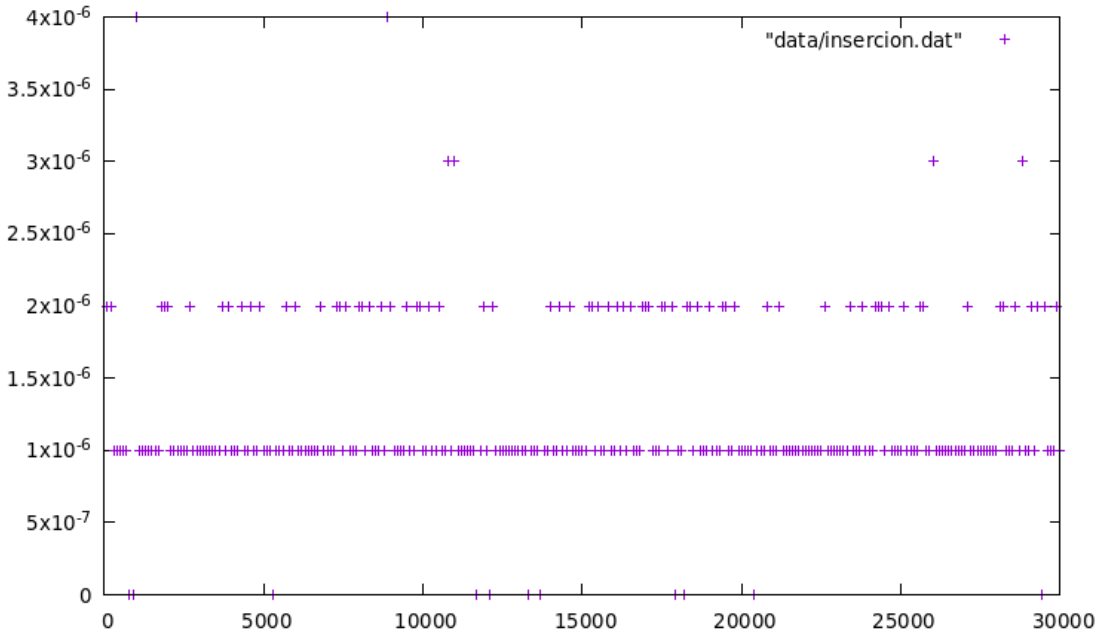
- Set

  Si se inserta un solo elemento, de tamaño logarítmico en general, pero constante amortizado si se da una pista y la posición dada es la óptima.
  En nuestro caso, como solo pasamos como parámetro el elemento a insertar, será $0(log_2(n))$.
  Para poder realizar la gráfica correctamente, hemos hecho cada inserción 1000000 de veces en cada iteración y luego hemos dividido ese tiempo entre 1000000 para intentar conseguir más precisión en los tiempos, puesto que el reloj que usamos no nos ofrece la precisión que necesitamos.
  Una vez hecho eso, ajustamos los resultados a una función logarítmica de base 2.



- Unordered_set

El caso medio tiene complejidad constante $(O(1))$, mientras que el peor caso posible tiene complejidad $(O(n))$.



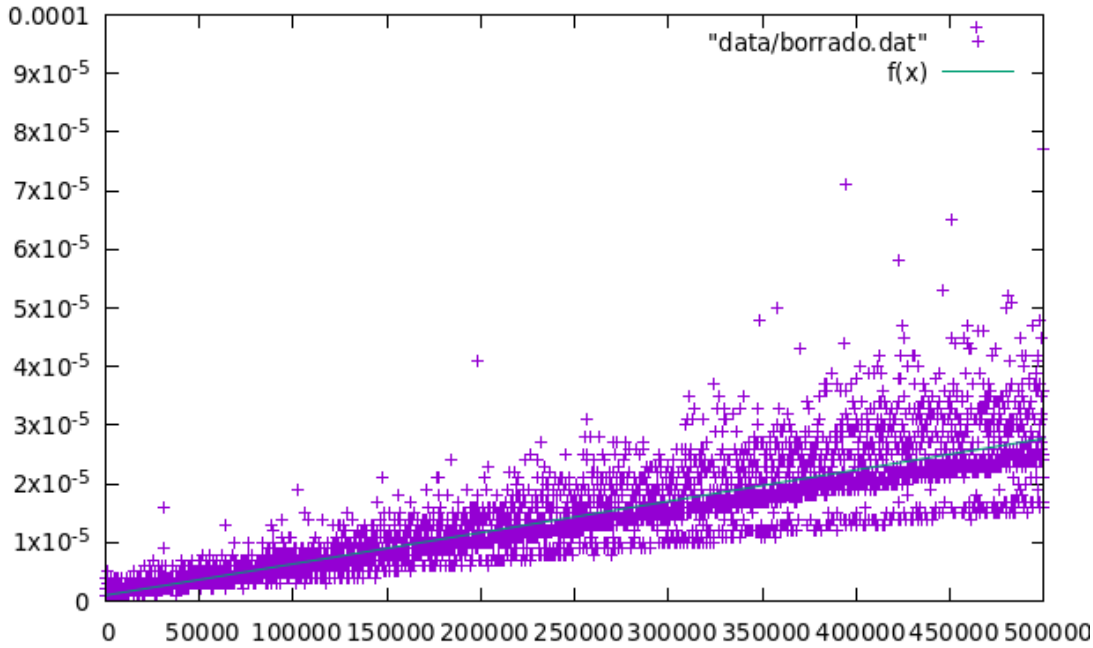
Vemos que el tiempo es constante y no depende del tamaño. Hay varias líneas de tiempos que se repiten (las distintas alturas en el gráfico) pero creemos que se puede deber a factores externos como faltas de caché.

Borrado del elemento central de la estructura

Se borrará el elemento en la posición $[\frac{N}{2}]$ en el caso de `vector` y `list`, y el elemento con clave $[\frac{N}{2}]$ en el caso de `set` y `unordered_set`.

- Vector

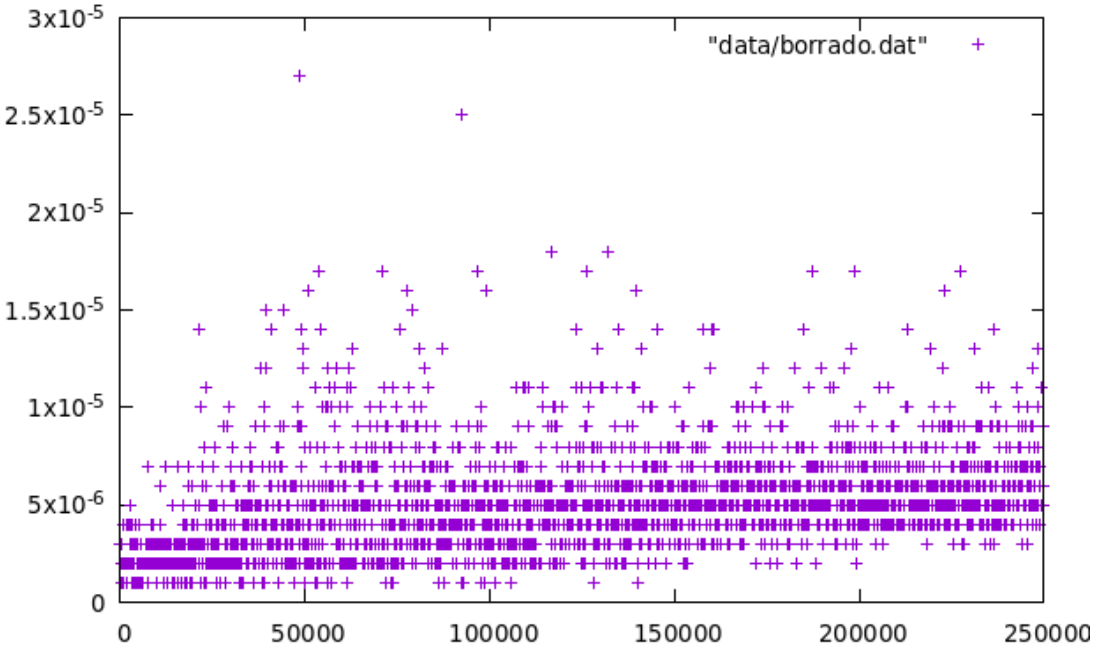
  La complejidad de borrar un elemento es lineal con respecto al número de elementos borrados (en este caso, un elemento) y con respecto al número de elementos tras el elemento que se borra, puesto que se desplazan.
  Entonces, en este ejercicio la complejidad será lineal es decir $O(n)$, en concreto $(O(\frac{n}{2}))$.



Para el borrado hemos hecho un vector con hasta 500000 elementos para mostrar la evolución lineal de los borrados.

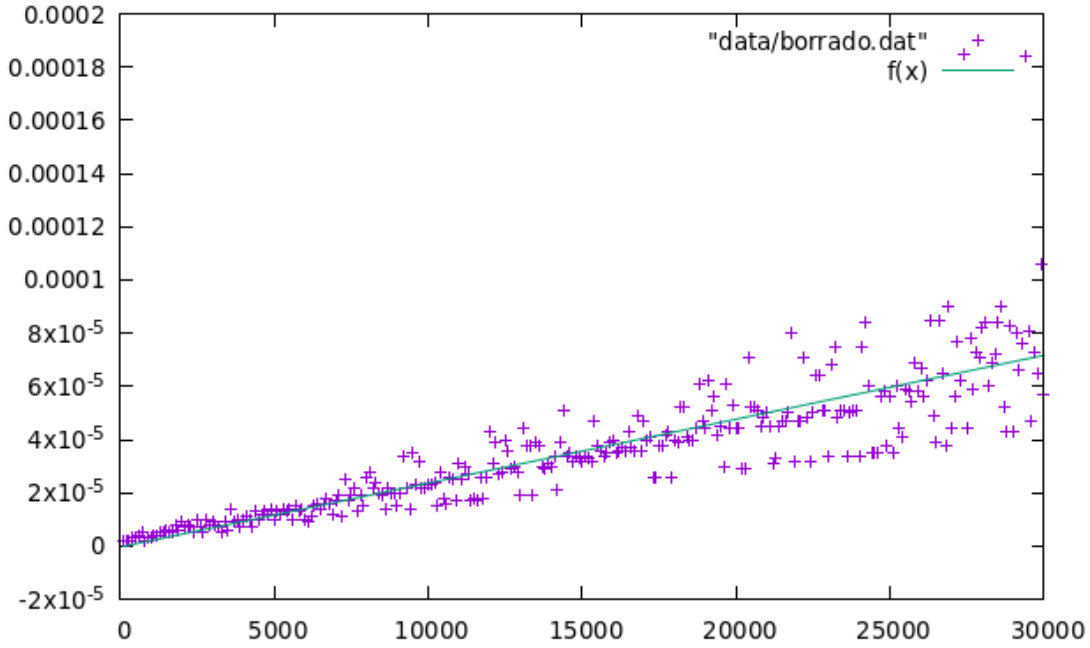
- List

  La complejidad es lineal según el número de elementos borrados, en este caso: uno.



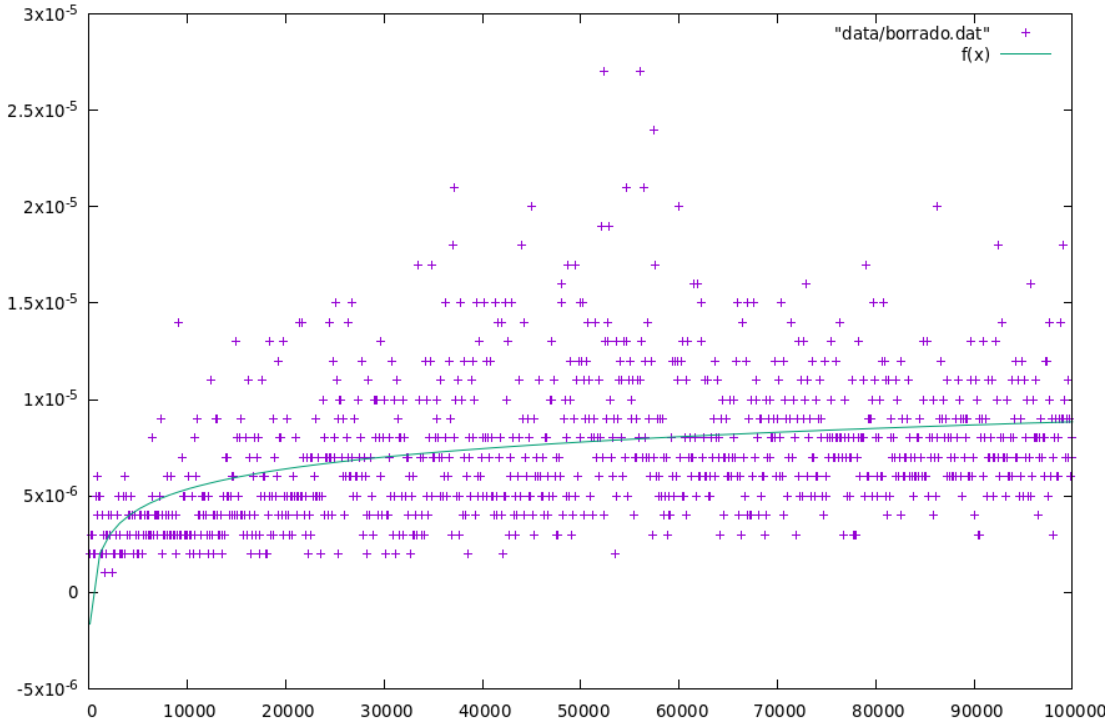
  Vemos que en una vista general, es constante aunque con tiempos variantes.

  Si tenemos en cuenta el tiempo que se tarda en encontrar el punto medio para borrarlo, la complejidad es $O(\frac{n}{2})$, es decir, lineal $(O(n))$.
  Gráfica incorporando los tiempos de búsqueda del punto medio.

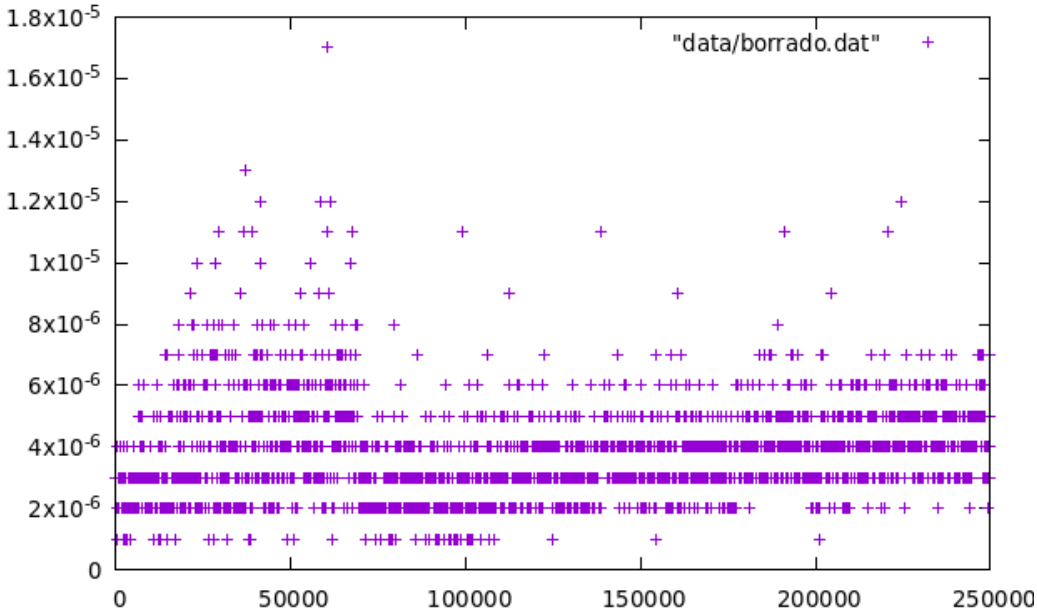


- Set

Según la documentación debe tener una complejidad logarítmica en función del tamaño del contenedor.



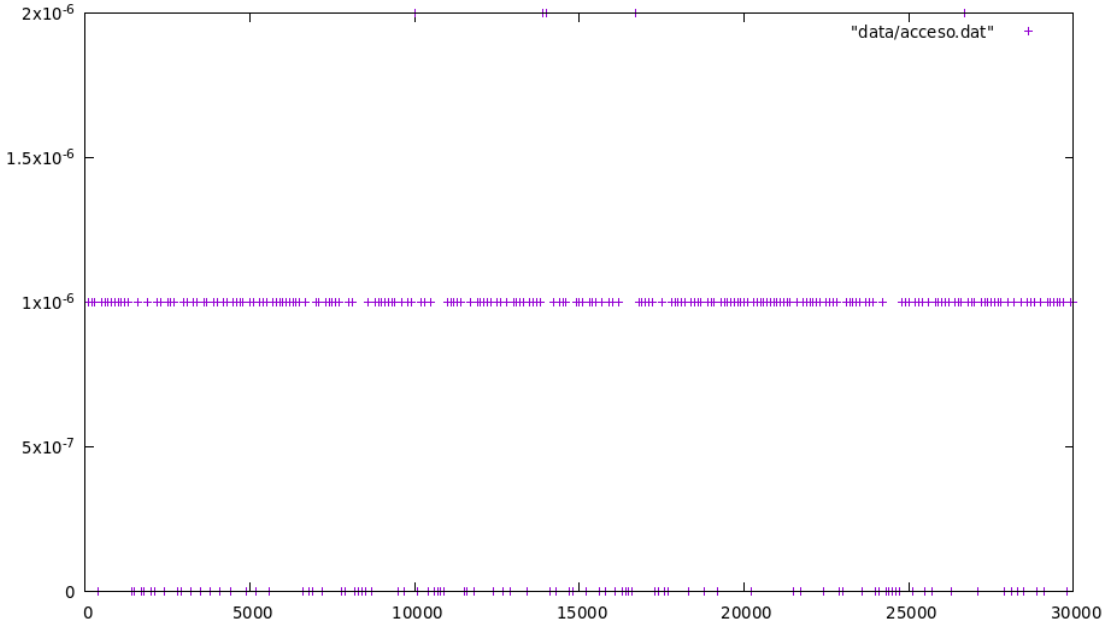
- Unordered_set
  Complejidad teórica:
  - Caso promedio: Lineal en el número de elementos eliminados (que es constante en nuestro caso).
  - Peor de los casos: lineal en función del tamaño del contenedor.



  Vemos que el resultado es similar al de `list`.

Acceso al elemento central de la estructura

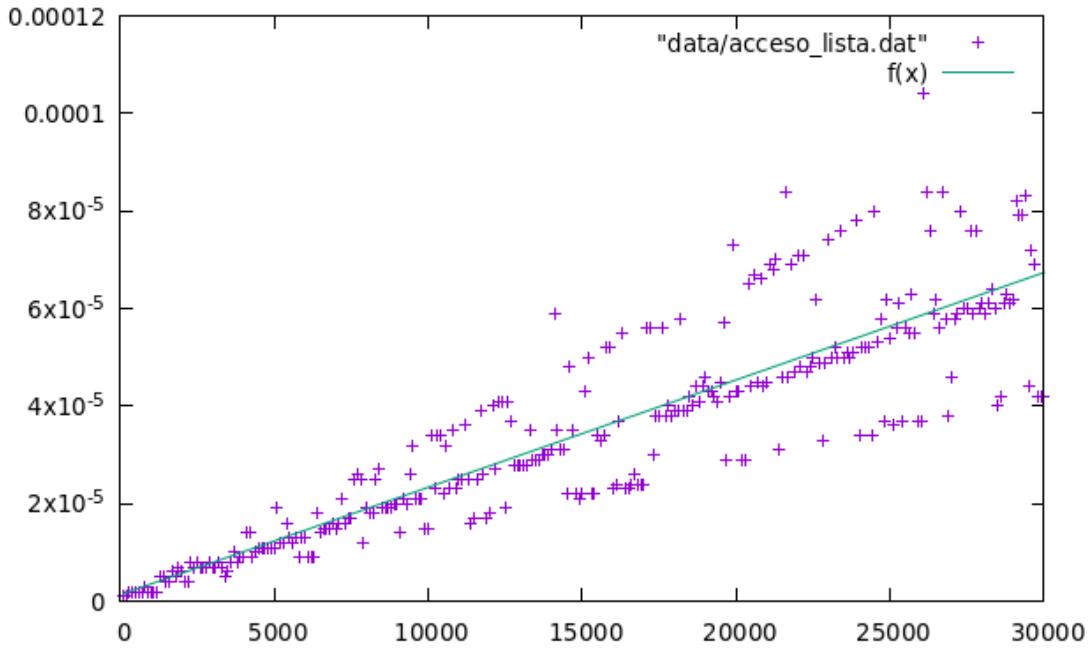
- Vector
  En el caso del vector usamos el operador `[]`. Teóricamente el tiempo de acceso es constante, es decir $O(1)$. Si dibujamos los tiempos que ha tardado en acceder al elemento $\frac{N}{2}$ vemos que es una función constante.



Hay algunos datos con tiempo 0 que creemos que se debe a algún error con la precisión del reloj o a factores externos.

- List

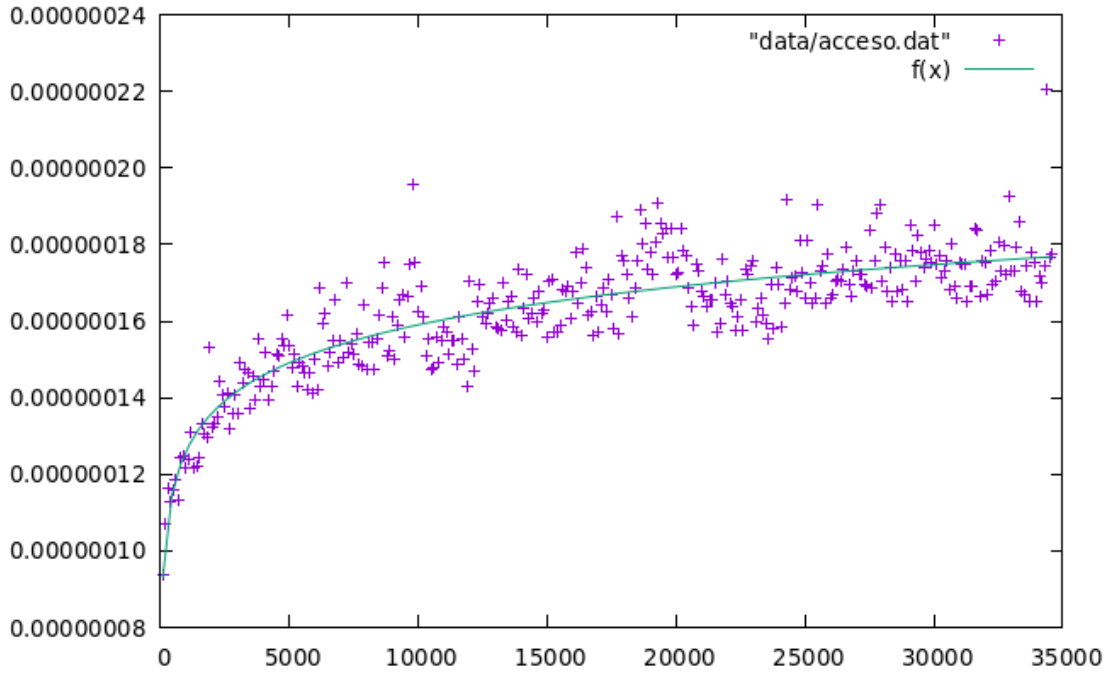
  Con la lista, no disponemos del operador `[]` y la única manera que tenemos de acceder al elemento $\frac{N}{2}$ es iterando hasta la mitad de la lista y desde ahí acceder al elemento. Por tanto, la complejidad es de $O(\frac{n}{2})$. Realmente, la constante $\frac{1}{2}$ no afecta mucho a la hora de estudiar la complejidad y seguimos diciendo que es una complejidad lineal ($O(n)$).



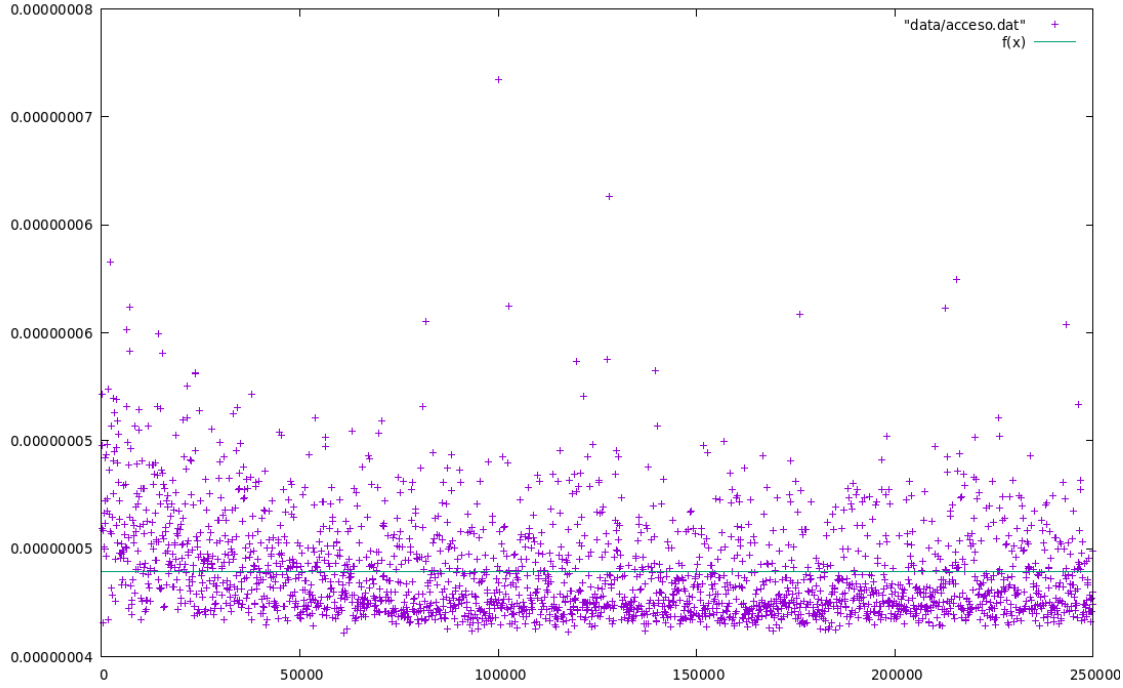
- Set

  Al usar la orden find, que encuentra el elemento con un algoritmo con complejidad

$O(log_2(n))$, el acceso al elemento medio también tiene una complejidad logarítmica



- Unordered_set
  Al usar la orden find, que encuentra de media en tiempo constante, podemos ver que a rasgos generales, la función de los tiempos también es constante.

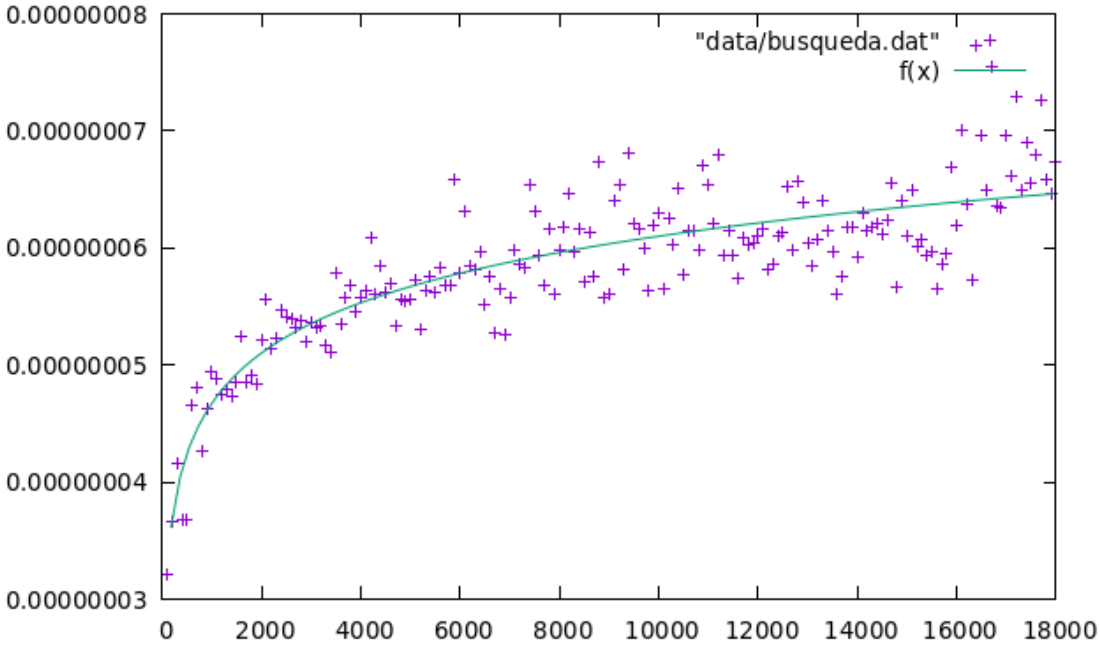


Búsqueda del elemento con clave/valor N en la estructura

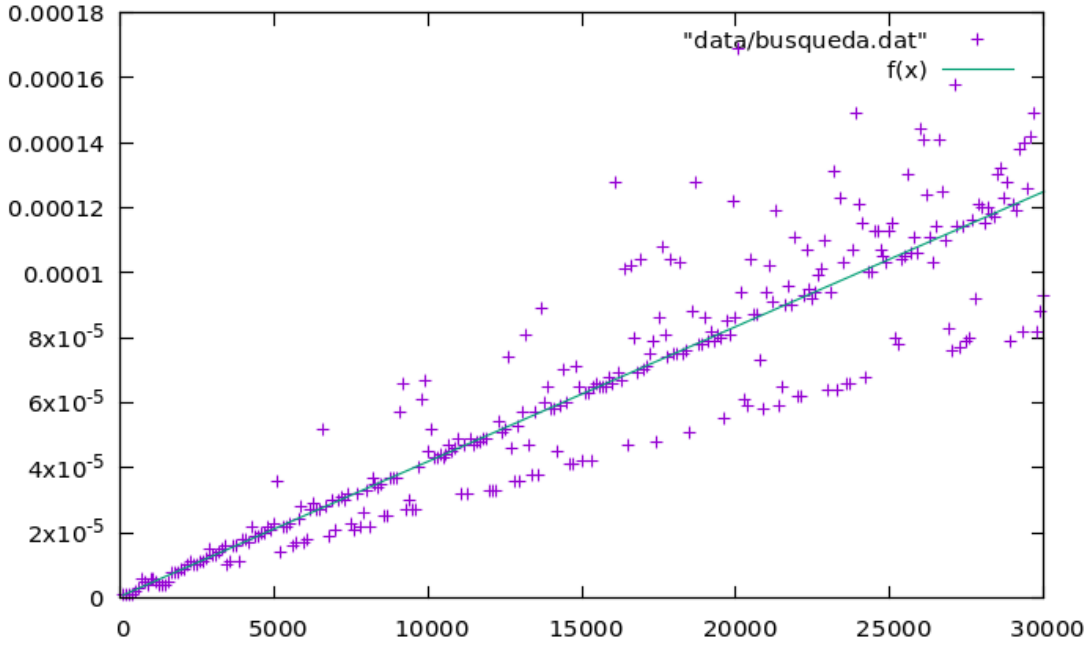
En el caso de `vector` se implementa una búsqueda binaria.
En el caso de `list`, recorriendo la lista.
En el caso de `set` y `unordered_set` usaremos el método `find()`.

- Vector
  En el caso del vector, al usar un algoritmo de búsqueda binaria la complejidad esde

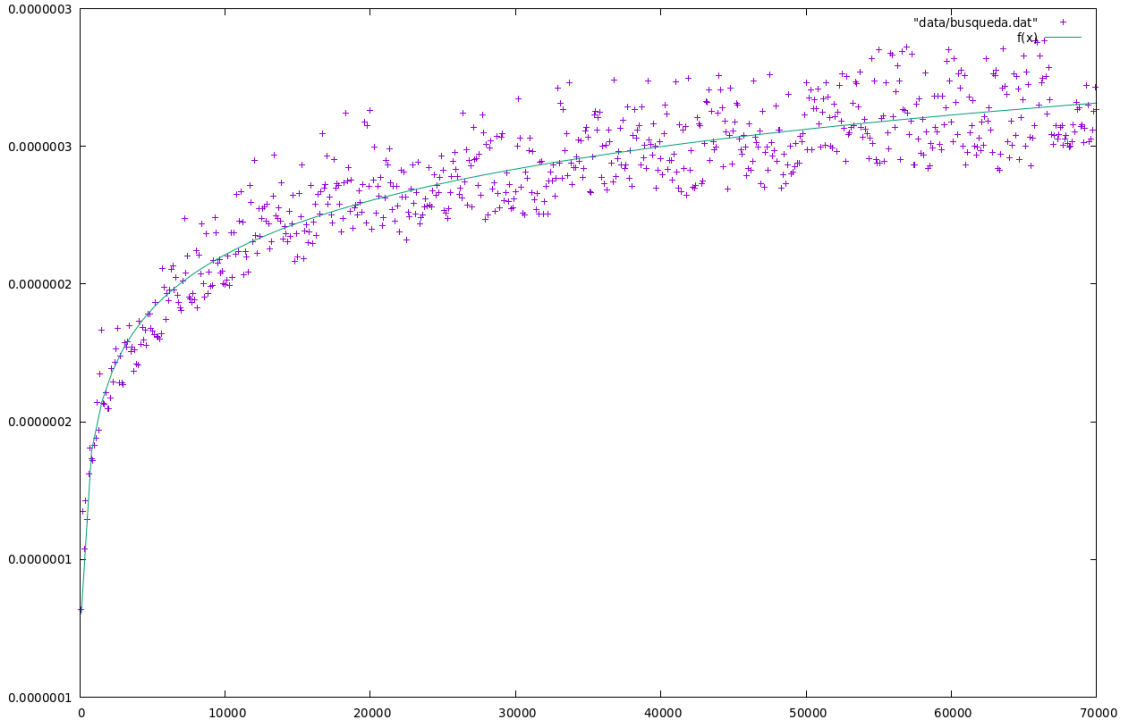
$O(log_2(n))$.



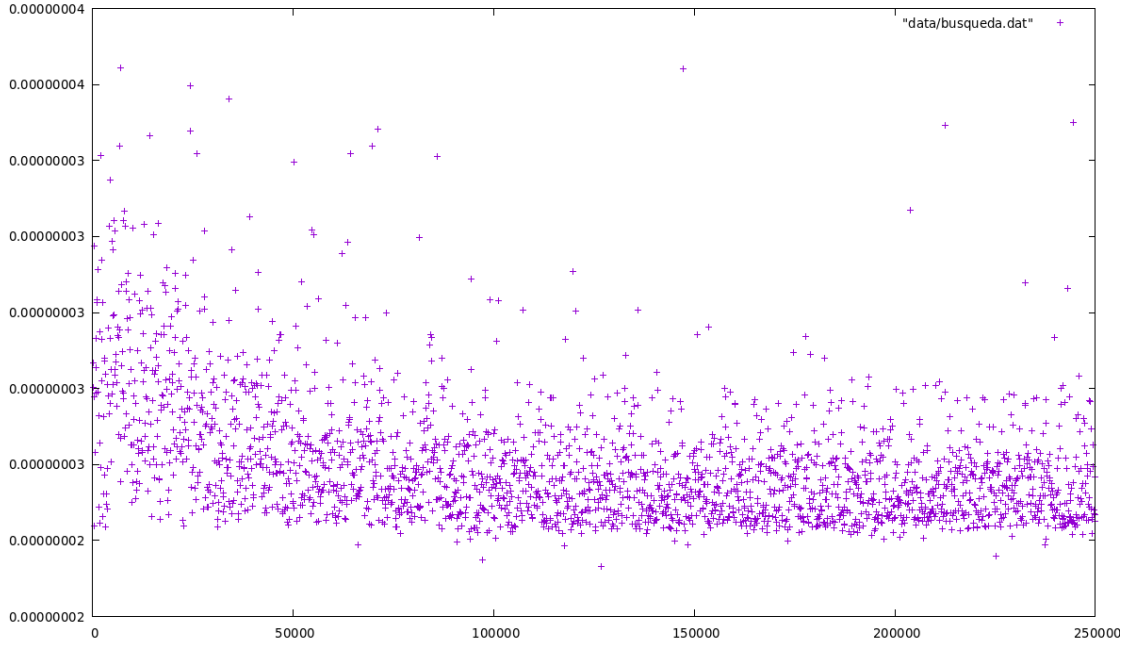
- List
  En el caso de la lista tenemos que iterar sobre ella para localizar el elemento. Como el elemento que buscamos es el n-ésimo la complejidad es de $O(n)$.



- Set
  La búsqueda de set tiene complejidad logarítmica: $O(log_2(n))$.



- Unordered_set
  La búsqueda con `unordered_set` es tiempo constante($O(1)$) en el caso medio y en el peor caso es lineal ($O(n)$).



  Podemos ver que a lo general, los tiempos son relativamente constantes, no van incrementando con el tamaño del contenedor.

Conclusión

En la inserción de elementos hemos visto como `list` es la más óptima, quedando como segundo lugar `unordered_set` al ser la media constante.

En el borrado de elementos, `set` es la más eficiente, siguiendo el resto una tendencia lineal. Sin embargo, no todos son lineales con respecto al número de elementos, sino que algunos son lineales con respecto al número de elementos borrados con una llamada (`list` y `unordered_set`). Por tanto, el segundo puesto estaría empatado entre estos dos.

En el acceso a alementos `vector` es la ideal, seguido de `unordered_set`.

Finalmente, en la búsqueda de elementos, `unordered_set` es la clara elección puesto que de media el tiempo es constante.

Podemos observar, como realmente no hay una que sea siempre superior al resto. Todas tienen algún punto fuerte y otros donde flaquean. Si por ejemplo nos interesa que los accesos a los datos sean muy rápidos usaremos la clase `vector` mientras que si nos interesa insertar elementos ágilmente usaremos una lista.