Motivation behind this article-

Well, this article is about a ***super-smart data structure*** which is not yet popular, but it is one of the best data structure I have ever seen (in terms of smartness).

How will you do the following three operations efficiently if there are **Q queries** demanding these operations->
a) **INSERTION**
b) **DELETION**
c) **SEARCHING**
d) **CLEARING/REMOVING ALL THE ELEMENTS**

You will probably use a binary tree (of course, self balancing) to perform all of these operations in **O(log N)** time and hence performing the whole queries in **O(Q.logN)** time.

You can use a hash table to compute the first three operations in amortized time-**O(1). But what about the last operation ?** When you use a hash table you have to **re-initialise** the hash table in each query, thus taking **O(N) time** for the last operation and hence the overall complexity becomes- **O(Q.N)** making it even worse than the binary tree approach.

Can you do better?

Yes, we can slash that **log N** term and reduce the complexity to **O(N).**

Try your luck as much as you can but you can't beat the data structure which is presented below that takes **O(1) time for all the above four operations** and hence the overall time complexity becomes just **O(Q)**.

Now, one can argue that when is this data structure useful ?

"*A common use of this data structure is with register allocation algorithms in compilers, which have a fixed universe(the number of registers in the machine) and are updated and cleared frequently (just like-* ***Q queries****) during a single processing run."*

Hence I got the motivation for this article.

The bonus with this data structure that this is also **super-fast** in calculating **intersection** and **union** of two sets/array.

Computer science is all about **Time-Space Tradeoff** and hence, this data reduces **time** by compromising on **space**

About the article-

Data Structure Used- Sparse Set
Algorithm used- Double Hashing (uses two arrays- **dense[ ], sparse[ ]**)

For convenience of the readers, I would strongly suggest to **classify this article** in two parts-
1st Part, 2nd Part

Purpose of the program-

i) 1st Part-

Design an efficient data structure to perform **Q queries.** Each query demands the below four operations-

a) **INSERTION**
b) **DELETION**
c) **SEARCHING**
d) **CLEARING/REMOVING ALL THE ELEMENTS**

ii) 2nd Part-

Design a set-data structure that perform the typical set operations like-

a) **add-member**
b) **check-membership**
c) **delete-member**
d) **clear_set**
e) **cardinality**
f) **union**
g) **intersect etc.**

<u>About 1st Part-</u>

See the below link for reference-

http://research.swtch.com/sparse

<u>About 2nd Part-</u>

See the below link for reference-

http://codingplayground.blogspot.in/2009/03/sparse-sets-with-o1-insert-delete.html

One may argue that for 2nd Part we can use highly efficient **bit vector** , but this data structure outperforms **bit vector.**

I have attached an image depicting the comparison between **sparse set** and **bit vector-**
Verdict- Sparse Set clearly beats Bit Vector in terms of performance.

Operation	*Bit Vector*	*Sparse*
member	O(1)	O(1)
add-member	O(1)	O(1)
delete-member	O(1)	O(1)
clear-set	O(u)	O(1)
choose-one	O(u)	O(1)
cardinality	O(u)	O(1)
forall	O(u)	O(n)
copy	O(u)	O(n)
compare	O(u)	O(n)
union	O(u)	O(n)
intersect	O(u)	O(n)
difference	O(u)	O(n)
complement	O(u)	O(u)