專題2
Making Binary Search Dynamic

110502518陳文獻

Explanation Of Implementing

I use an 2D vector in c++ (named as "Vectors") to implement this data structure.

Mark n+m as N := maximum amount of data

Vectors size = $\lceil \log(N) \rceil$

Vectors[i] size = 2^i where $0 <= i < \log(N)$

And I divide s, I, d three operator to three function:

Search(); Insert(); Delete();

also, Binary_Search(); Merge(); to implement functions above mentioned.

p.s. I just use Insert() to initialize data

Vectors

I use an struct that combine value and mark as a unit in Vector[i].

I also assign two vectors, “is_full” and “mark_couter”, which is parallel to “Vectors”, indicate if is full or not, how many marked elements in the ith sub-vector respectively.

Search(value: int)

for every full sub-vector, use common binary search to check if element in sub-vectors.

```
for i <- 0 to N do
    if is_full[i]
        if value in Vectors[i] // using binary search
            return i, index_of(value, Vectors[i])
// means position of value in Vectors
```

Insert(value: int)

First search() to check if value can insert or not.

if is, find the first empty sub-vector Vectors[i]. merge Vectors[0 to i-1], and put them to Vector[i].

```
while(is_full[i]) i += 1;
Vectors[i][0] = value
for j <- 0 to i-1
    Merge(Vectors[i], Vectors[j])
```

Delete(value: int)

First search() to check if value can delete or not

if is, mark the element.

if amount of marked points > (2 ^ i / 2) then clean marked element.

Cleaning:

if Vectors[i-1] is full, merge Vectors[i-1] and Cleaned Vectors[i] and put into Vectors[i].

else just put cleaned Vectors[i] to Vector[i-1]

Pseudo Code Of Delete(value: int)

```
i, j = Search(value)
Vector[i][j].mark = false
mark_counter[i] += 1
if (mark_counter[i] > size_of(Vector[i]) / 2)
    Clean(Vector[i])
    if (is_full[i-1])
        Vector[i] = Merge(Vector[i], Vector[i-1])
        mark_counter[i] = mark_couter[i-1]
        is_full[i-1] = False
    else
        Vector[i-1] = Vector[i]
        swap(is_full[i-1], is_full[i])
    mark_couter[i-1] = 0
```

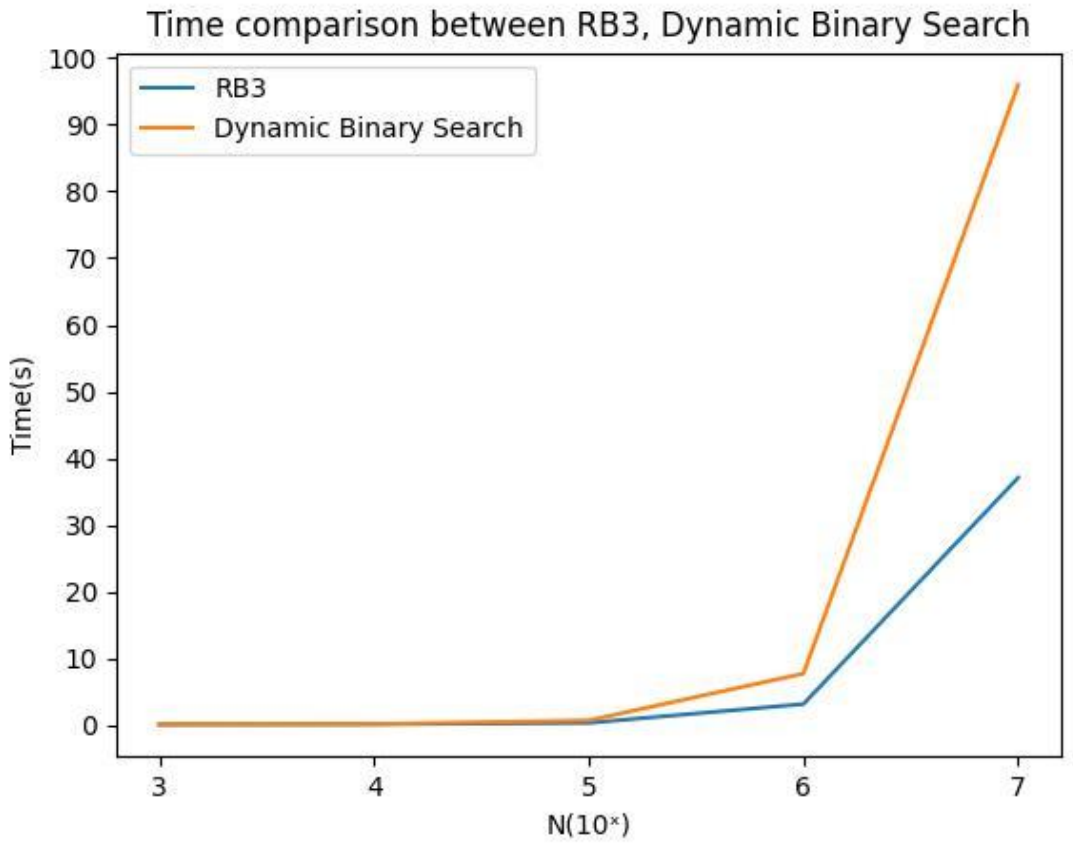
Time complexity comparison

Data Structure	DBS(Dynamic Binary Search) *if we ignore search before insert, delete			RB3(Red Black Tree, using c++ set)		
method	Search	Insert	Delete	Search	Insert	Delete
worst case	logn*logn	logn	logn	logn	logn	logn
avg case	logn*logn	logn	1	logn	logn	logn

Time complexity comparison

RB3(Red-Black Tree, using c++ set)
Dynamic Binary Search (This topic)

They all have great performance until N=10^5.(RB3 slightly better)
RB3 looks better in extreme large cases.



Conclusion

In common using case, I would rather use red-black tree because is always in library, but this data structure not.

Only If it is necessary to build it by my own, I would choose this data structure that much more intuitive and not too complicate for me.