Dynamic Array Set

Individual Project

Instructor: Armen Kostanyan
Student: Aram Tatalyan

Introduction

Let's consider two widely applicable data structures Array and Red Black Tree. Let's compare their characteristics

	Array	Red Black Tree
insert	O(n)	O(log(n))
find (using key)	O(1)	O(log(n))
erase	O(n)	O(log(n))

Array data structure is compact and simple and it's the best on access time using the known key. However inserting and deleting can take up to N operation. Red Black Tree is balanced on all operations.

The aim of this project is to try to implement data structure that has better characteristics on insert and erase slightly sacrificing on find (using key) operation performance.

Definition

Let's name the new data structure Dynamic Array Set.

Let's define container of Array(s) each of size N^2 where N = 0, 1, 2, …

0 index Array will have 2^0 = 1 size
1 index Array will have 2^1 = 2 size
2 index Array will have 2^2 = 4 size
3 index Array will have 2^3 = 8 size
N index Array will have 2^N size

Each Array within array container has these properties

- Each Array is identified by it's index that also defines it's size using 2^(index) formula.
- Each Array has 2 conditions, enabled which means that it is fully filled with stored elements or is disabled which means that it has no stored elements at all
- Array condition depends on total number of keys stored in "Dynamic Array Set" data structure.
- Each Array is sorted

Let's define how Array condition changes from disabled to enabled and vice-versa. Let's consider that "Dynamic Array Set" has total N elements stored.

In this case with below specified algorithm we can determine which Array(s) within the container should be enabled for specified data structure size;

```
unsigned remaining = N;
while (remaining) {
        unsigned enabled_array_index = log2(N / 2);
        remaining = N % 2;
}
```

If we represent number of elements stored in Dynamic Array Set in binary format then indexes of "1"s will identify enabled Array(s) and indexes of "0"s will indicate disabled Array(s).

Decimal Value	5			
Bit Value	0	1	0	1
Bit Index	3	2	1	0

Let's say we have 4 keys, x1, x2, x3, x4, x5
Total 4 keys.
We have 2 Arrays enabled
0 index Array with size 2^0=1 and 2 index Array with size 2^2=4.
All these 5 elements from x1-x5 should be stored in enabled Arrays.

On each insert operation when we insert single element into the data structure the size is increased by 1. It means that at exactly 1 Array should be enabled on insert operation and at least 1 should be disabled.
For example, in case if size is equal to 0111 in binary format and we increase it by 1 we will have 1000. Which means that index 3 Array is enabled and 0,1,2 Arrays are disabled.
In case of erase operation we remove 1 element decreasing the size by 1. That means that on each erase we disable exactly 1 Array and enable 0 or more Arrays.
In case if size is 1000 in binary format, decreasing its size by 1 we will have 0111. Disabling index 3, enabling indexes 0,1,2.
In case of 0001 invoking erase we will have 0000, which means that Dynamic Array Set data structure is empty.

Complexity Evaluation

Find

Let's consider that N is the number of elements stored in Dynamic Array Set. Even if we have to traverse through all Arrays to find seeking element the number of Arrays is not greater than log2(N). On each Array we have search for the element using dividing method, which is equal to Array size. Even if we traverse through all Arrays the maximum traverse will not exceed N, it means that log2(N) will be the sum of all Arrays where we will search for key using dividing method. So we have worst time log2(N)*log2(N) complexity for each find operation.

Insert

In the worst case we need N+1 operation to move all existing data from smaller arrays into new big one.
Example 0111 size in binary format if we add 1 we will have 1000 moving data from smaller Arrays with 0,1,2, indexes into new bigger Arrays with index 3.

In the worst case we need 1 operation.
Example in case of 1110 size in binary format if we add 1 we will have 1111. We just need to enable Array with index 0 adding only 1 element.

Delete

Same for delete.
The worst case is N+1 so O(N).
The best case is 1 so O(1).

The average complexity yet should be calculated.

Implementation

Implementation of the data structure can be found at specified address on GitHub.

https://github.com/mister-crow/dynamic-array-set

Data structure is implemented using C++ programming language.
It is almost compliant to STL.
insert, erase, find methods signatures are similar to those found in std::set container.

It uses BlockAllocator class to allocate memory blocks for Arrays. For small arrays it caches memory blocks in order to not often disturb memory allocator for small memory blocks, which supposed to greatly improve overall performance of the data structures insert and delete operations.

There you can find a test application that compares new data structure performance with Red Black Tree data structure.
STL std::set container is used as a Red Black Tree data structure.
In the next page you can find testing results.

Testing results

Here is the output of testing application for number of operation 100,000 and 1,000,000 and 10,000,000.
The duration is specified in milliseconds.

```
Arams-MacBook-Pro:src aram$ ../bin/test_app
Enter the number of tests for each operation
100000
Number of iterations 100000

RB Tree insert duration 52
Dynamic Array insert duration 15

RB Tree find duration 15
Dynamic Array find duration 53

RB Tree erase duration 16
Dynamic Array erase duration 54
```

```
Arams-MacBook-Pro:src aram$ ../bin/test_app
Enter the number of tests for each operation
1000000
Number of iterations 1000000

RB Tree insert duration 547
Dynamic Array insert duration 183

RB Tree find duration 180
Dynamic Array find duration 763

RB Tree erase duration 185
Dynamic Array erase duration 741
```

```
Arams-MacBook-Pro:src aram$ ../bin/test_app
Enter the number of tests for each operation
10000000
Number of iterations 10000000

RB Tree insert duration 5880
Dynamic Array insert duration 2580

RB Tree find duration 1986
Dynamic Array find duration 9449

RB Tree erase duration 1962
Dynamic Array erase duration 9464
```

Conclusions

We can see that Red Black Tree behavior is stable. We increase the load 10 times and the time required to process the load increases exactly 10 times.
Meanwhile on Dynamic Array Set it increases more than 10 times which could indicate problems related to code optimization.
Further research should be done and also there is a room for code optimization and evaluate performance of Dynamic Array Set data structure.
On each insert operation std::sort is used.
We must make sure that Quick Sort will not be used since we deal with semi-sorted arrays where Quick Sort perform inefficient. This also could be the cause why 10 times increasing the load increases processing time more than 10 times.
Since it is known that std::sort uses one algorithm for short arrays and another algorithm for larger arrays. It means that we must use something else instead of std::sort.