bubble: A new data structure based on the idea of fibonacci heaps, but works in practice

1st Spiros Maggioros
School of Electrical & Computer Engineering
National Technical University of Athens
Athens, Greece
spirosmag@ieee.org

***Abstract*—bubble is a data structure that is highly influenced from the fibonacci heap structure. It uses an array and AVL trees to store elements and fastly retreieve or insert new ones. The time complexity of insertion, deletion and searching is $O(logn \cdot logm)$ where n is the size of the initial array(or the input bubble size) and m is the number of nodes at the index that the bubble is going to search**

***Index Terms*—Data Structures; Algorithms; Trees; Data**

I. Idea & Structure

The idea of the bubble data structure came directly from the Fibonacci heaps, the similarity is displayed bellow, the only difference is that the nodes are not connected via pointers, but, we now have an array for the initial elements(or key elements, as i like to call them) instead of the roots of the trees.

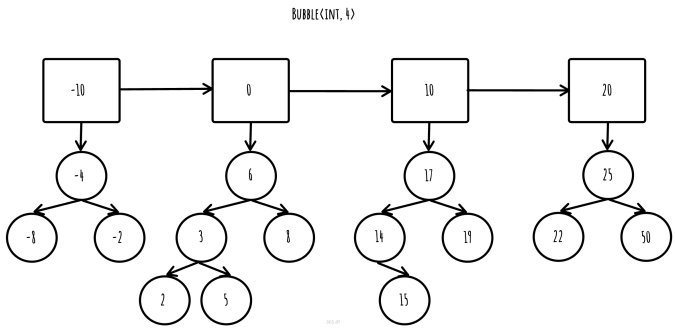

Fig. 1. MY Panel

bubble showed incredible results regarding insertion, deletion and searching and outperformed all the state-of-the-art generic data structures like binary search trees and lists. As explained in the abstract, the time complexity of the three basic operations is $O(logn \cdot logm)$ and the time complexity of the same operations for AVL Trees is $O(logn)$, but what is the n and m each time? For the bubble data structure, n is the initial size and m is the size of the AVL tree on the current index, thus, we can understand that $O(logn \cdot logm)$ can be way smaller than $O(logn)$ if n is the number of all the elements in the case of the AVL Tree alone. The array(first layer) of the structure is implemented with a typical std::vector and have an access time of $O(1)$. The AVL Tree(second layer) is implemented from scratch and is included in the repository, it's a header only file that has only one class and the node architecture is

```
struct node{
    T info;
    int64_t height{0};
    std::shared_ptr<node> left;
    std::shared_ptr<node> right;
    node(T key) : info(key), left(nullptr),
                  right(nullptr) {}
} node;
```

In order to join the array with the AVL Tree together i used std::pair and now the whole structure can be created with only one line

```
std::vector<std::pair<T, std::optional<avl_tree<T
    >>>> bubble;
```

We still have $O(1)$ access time complexity for the key elements, as we can just do bubble[index].first.

II. Insert

The insertion is the first - and most important - basic operation and it has an edge case when **the size of the array(first layer) is not yet full**. When the array is not full, we are just pushing back elements in $O(1)$, there's nothing more we can do. Once the array is full, we have to perform sorting in order for the binary search to work in later stages.

```
if(_size < _SIZE) {
    list.push_back({key, std::nullopt});
    _size++;
    return;
}
if(_size == _SIZE) {
    std::ranges::sort(list, [](const std::pair<T,
        std::optional<avl_tree<T>>> &a, const
    std::pair<T, std::optional<avl_tree<T>>> &b){
        return a.first < b.first;
    });
}
```

I used std::ranges for sorting and implemented a lambda function to support the custom structure for sorting.
If the array is full, then we have to only insert in AVL Trees, binary searching will give us the index as the array is sorted and we can perform std::lower_bound. The lower bound will return us the first element that is bigger than the key we want to insert, so we have to insert it to index - 1 as each index contains elements with value greater than themselves and smaller than the next index. There are a couple of checks that we have to

do while inserting like checking if the element already exists in the structure or if the tree is empty so we have to initialize it. This is why i used std::optional, because for big datasets and a big starting array size, we can't make sure that all the indexes will create a tree.

```
auto it = std::lower_bound(std::ranges::begin(this
    ->list), std::ranges::end(this->list), key,
    [](const std::pair<T, std::optional<avl_tree<T
    >>> &pair, const T &key){ return pair.first <
    key; });

if(it != std::ranges::end(this->list)) {
    int idx = std::distance(std::ranges::begin(
        this->list), it);
    if(this->list[idx].first == key) { return; }

    if(idx == 0){
        if(this->list[0].second == std::nullopt) {
            this->list[0].second = avl_tree<T>();
        }
        this->list[0].second.value().insert(key);
    }
    else{
        if(this->list[idx - 1].second == std::
            nullopt) {
            this->list[idx - 1].second = avl_tree<
                T>();
        }
        this->list[idx - 1].second.value().insert(
            key);
    }
}
else {
    if(this->list[this->list.size() - 1].second ==
         std::nullopt){
        this->list[this->list.size() - 1].second =
             avl_tree<T>();
    }
    this->list[this->list.size() - 1].second.value
        ().insert(key);
}
_size++;
```

III. Remove

To remove an element, we follow the same steps basically. We have the same edge case when the array is not yet full, so the only thing we can do is a simple removal of an element from an array. I used std::ranges::remove_if to perform this operation using again a lambda function for it to work with the custom structure.

```
if(this->_size <= _SIZE) {
    auto [begin, end] = std::ranges::remove_if(
        this->list, [&](const std::pair<T, std::
        optional<avl_tree<T>>> &t) { return t.
        first == key; });
    list.erase(begin, end);
    _size--;
}
```

This operation will take $O(n)$ time complexity as it's better to save time with not sorting the array at each insertion. If the array is full, we have to do the same as the insertion, std::lower_bound will give us the index and if the key is not in the array, we have to remove it from the tree of index - 1. But, if the key we want to remove exist in the array, we have to remove the root from the tree of the current index and put the value of the root to the same index on the array. We still have to check if the AVL Tree is not yet initialized.

```
auto it = std::lower_bound(std::ranges::begin(this
    ->list), std::ranges::end(this->list), key,
    [](const std::pair<T, std::optional<avl_tree<T
    >>> &pair, const T &key){ return pair.first <
    key; });
if(it != std::ranges::end(this->list)) {
    size_t idx = std::ranges::distance(std::ranges
        ::begin(this->list), it);

    if(this->list[idx].first == key && this->list[
        idx].second != std::nullopt) {
        T curr_root = this->list[idx].second.value
            ().get_root();
        this->list[idx].second.value().remove(
            curr_root);
        this->list[idx].first = curr_root;
        return;
    }

    if(idx == 0){
        if(this->list[0].second == std::nullopt){
            return;
        }
        this->list[0].second.value().remove(key);
    }
    else{
        if(this->list[idx - 1].second == std::
            nullopt) {
            return;
        }
        this->list[idx - 1].second.value().remove(
            key);
    }
}
else {
    if(this->list[this->list.size() - 1].second ==
         std::nullopt) {
        return;
    }
    this->list[this->list.size() - 1].second.value
        ().remove(key);
}
_size--;
```

This will take $O(logn \cdot logm)$ as it's pretty much the same as the insertion, std::lower_bound takes $O(logn)$ to find the index and then we need $O(logm)$ to remove an element from the tree.

IV. Search

Searching is a much easier operation as we either have to search on the array or an AVL Tree. I used std::lower_bound once again to find the first element that is greater or equal then the key we want to search and we have to check if the key exist in the array, otherwise we have to search in the AVL Tree of index - 1.

```
auto it = std::lower_bound(std::ranges::begin(this
    ->list), std::ranges::end(this->list), key,
    [](const std::pair<T, std::optional<avl_tree<T
    >>> &pair, const T &key){ return pair.first <
    key; });
if (it != std::ranges::end(this->list)){
    size_t idx = std::ranges::distance(std::ranges
        ::begin(this->list), it);
    if(idx == 0){
        if(this->list[0].first == key) { return
            true; }
```

```
        if(this->list[0].second == std::nullopt) {
             return false; }
        if(this->list[0].second.value().search(key
            ) == true) { return true; }
    }
    else {
        if(this->list[idx].first == key) { return
            true; }
        if(this->list[idx - 1].second == std::
            nullopt) { return false; }
        if(this->list[idx - 1].second.value().
            search(key) == true) { return true; }
    }
}
else {
    if(this->list[this->list.size() - 1].first ==
        key) { return true; }
    if(this->list[this->list.size() - 1].second ==
         std::nullopt) { return false; }
    if(this->list[this->list.size() - 1].second.
        value().search(key) == true) { return true
        ; }
}
return false;
```

V. Examples & Tests

Let's use the container itself to test the three basic operations. To initialize a bubble we do

```
bubble<T, SIZE> b;
```

Where T is the type of elements and SIZE is the initial size of the array. Let's create a bubble with an array size of 5 with integers.

```
bubble<int, 5> b;
```

Let's fill the array by inserting 5 elements

```
b.insert(-50, -20, 0, 20, 50);
```

From now on, if we insert any more elements, AVL Trees are going to be initialized, let's insert some more elements.

```
b.insert(24, 28, 21, 42, -22, -25, 4, -3, 55, -59)
    ;
```

In order to search for an element, we have to use the search function

```
assert(b.search(24) == true && b.search(-25) ==
    true && b.search(100) == false);
```

To remove an element, we have to use the remove function

```
b.remove(24);
assert(b.search(24) == false);
b.remove(-20);
assert(b.search(-20) == false);
```

Note that we can only use multiple arguments for insert and remove functions. Search function only takes one parameter.