Comparative Data Structure

Mingji Han, Mingyu Chen, Bolun Liu, Haochuan Hu

Ordered Dictionary

- A collection of key-value pairs. Keys are ordered.
- Supported operations: Insertion(Key, Value), Delete(Key), Lookup(Key)
- Widely used in various applications:
- Database System (MongoDB)
- Filesystem (Ceph)
- In-Memory Cache (Redis)

Data Structures

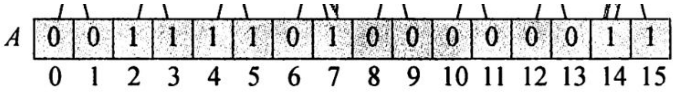
VEB Tree (Mingyu Chen)

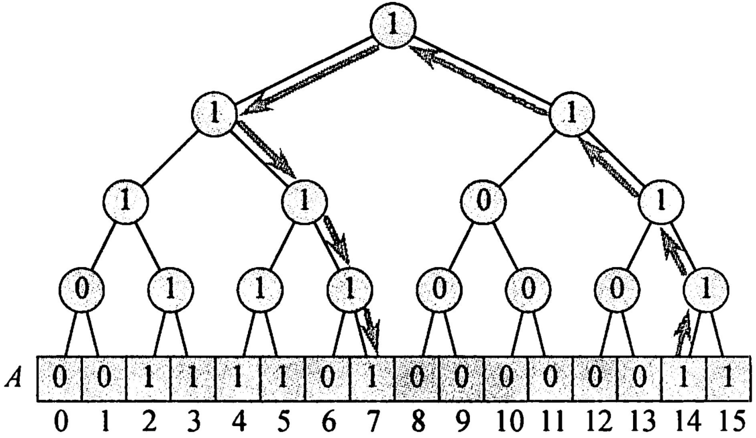
Binary Search Tree (Bolun Liu)

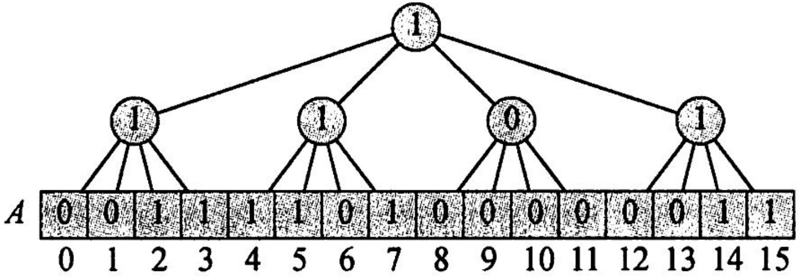
AVL Tree (Haochuan Hu)

Skip-List (Mingji Han)

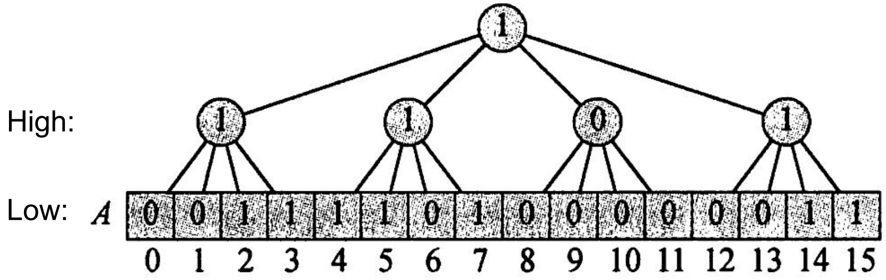
VEB Tree


A
0 0 1 1 1 1 0 1 0 0 0 0 0 0 1 1
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15


A
0 0 1 1 1 1 0 1 0 0 0 0 0 0 1 1
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15


A
0 0 1 1 1 1 0 1 0 0 0 0 0 0 1 1
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15

VEB Tree



1011

High:10, Low:11

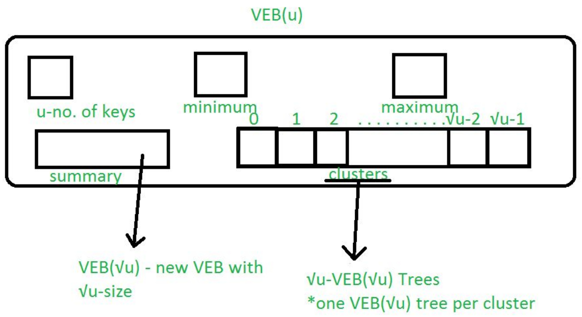
11 = sqrt(16)*2+3

Given n, the number of bits is log(n), the height of the tree is loglog(n)

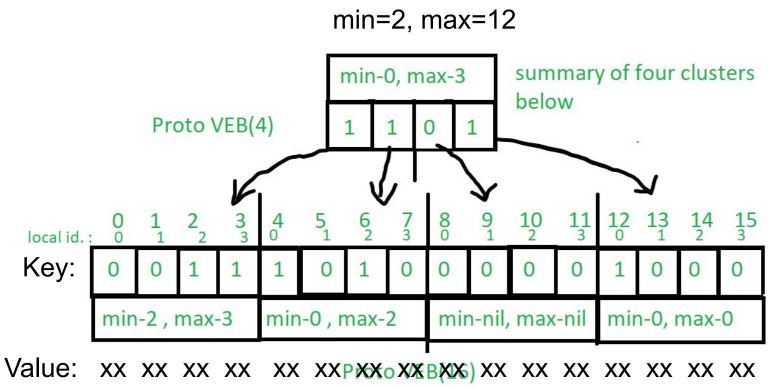
VEB Tree

Van Emde Boas Tree is a recursively defined structure.

1. **u**: Number of keys present in the VEB Tree.
2. **Minimum:** Contains the minimum key present in the VEB Tree and its childrens.
3. **Maximum**: Contains the maximum key present in the VEB Tree and its childrens.
4. **Summary:** Points to new VEB($\sqrt{u}$) Tree which contains overview of keys present in clusters array.
5. **Clusters:** An array of size $\sqrt{u}$ each place in the array points to new VEB($\sqrt{u}$) Tree.



VEB Tree



Algorithm	Average	Worst case
Space	$O(M)$	$O(M)$
Search	$O(\log\log M)$	$O(\log\log M)$
Insert	$O(\log\log M)$	$O(\log\log M)$
Delete	$O(\log\log M)$	$O(\log\log M)$

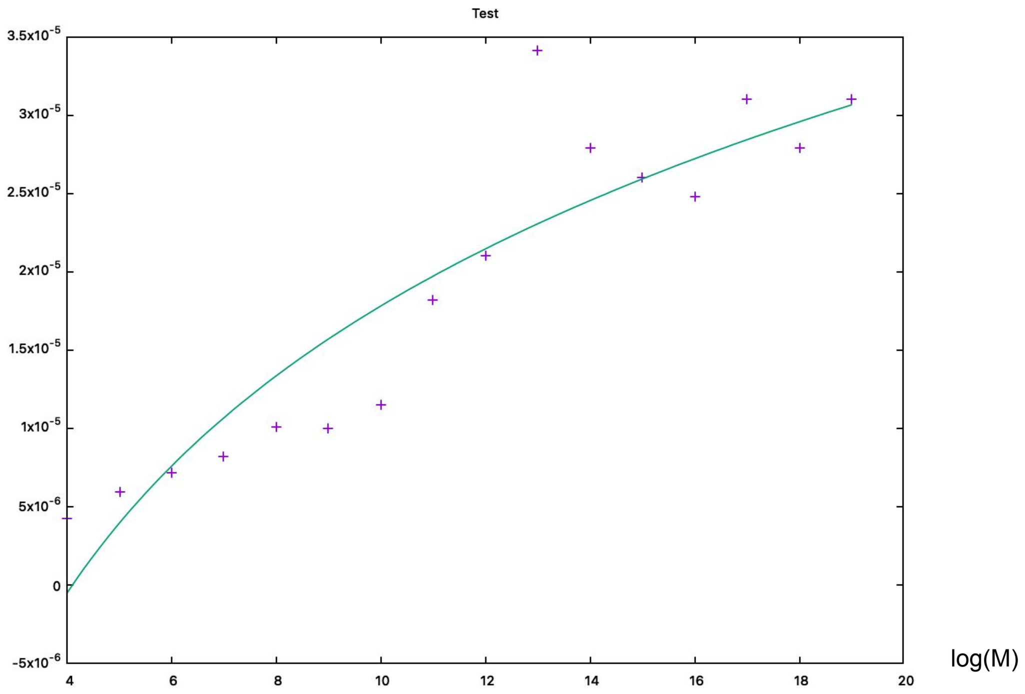
$$T(U) \le T(U^{1/2}) + \Theta(1);\ \text{solves to } \mathbf{O(\log\log}\ \boldsymbol{U}\mathbf{)}$$

VEB Tree

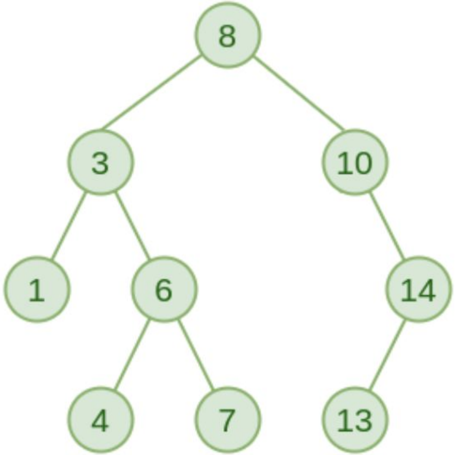
```python
veb = VEB(256)
# Inserting keys
insert(veb, 1, "12")
insert(veb, 0, "123")
insert(veb, 2, "1342")
insert(veb, 4, "1255")
insert(veb, 100, "1275")

#print(isMember(veb, 2))
#print(VEB_predecessor(veb, 4), VEB_successor(veb, 1))
#print(VEB_minimum(veb), VEB_maximum(veb))

start = VEB_minimum(veb)
while start is not None:
    print(start, lookup(veb, start))
    start = VEB_successor(veb, start)

#if isMember(veb, 2):
   # VEB_delete(veb, 2)

#print(isMember(veb, 2))
#print(VEB_predecessor(veb, 4), VEB_successor(veb, 1))
```

```
/Users/mac/PycharmProjects/EC504project/ver
0 123
1 12
2 1342
4 1255
100 1275

Process finished with exit code 0
```

VEB Tree


Test
3.5x10^-5
3x10^-5
2.5x10^-5
2x10^-5
1.5x10^-5
1x10^-5
5x10^-6
0
-5x10^-6
4
6
8
10
12
14
16
18
20
log(M)

Binary Search Tree



Binary Search Tree is a node-based binary tree data structure which has the following properties:

- The left subtree of a node contains only nodes with keys lesser than the node’s key.
- The right subtree of a node contains only nodes with keys greater than the node’s key.
- The left and right subtree each must also be a binary search tree.

Binary Search Tree

This BST implementation for ordered dictionary supports key-value pairs. Each node in the binary search tree contains a key and a value, which are used to store the key-value pairs.

- The `insert` method inserts a new key-value pair into the tree. If the key already exists in the tree, the associated value is updated.
- The `delete` method removes the key-value pair with the given key from the tree.
- The `lookup` method searches for a key-value pair with the given key by traversing the tree in a manner similar to binary search.
- The `findMin` and `deleteMin` methods are helper methods used in the delete method to handle the case where a node to be deleted has two children.

Again, this implementation assumes that the keys are comparable

Binary Search Tree

```java
public void insert(K key, V value) { root = insertNode(root, key, value); }


private Node insertNode(Node node, K key, V value) {
    if (node == null) {
        return new Node(key, value);
    }

    int cmp = key.compareTo(node.key);
    if (cmp < 0) {
        node.left = insertNode(node.left, key, value);
    } else if (cmp > 0) {
        node.right = insertNode(node.right, key, value);
    } else {
        node.value = value;
    }

    return node;
}
```

```java
public void delete(K key) { root = deleteNode(root, key); }

3 usages
private Node deleteNode(Node node, K key) {
    if (node == null) {
        return null;
    }

    int cmp = key.compareTo(node.key);
    if (cmp < 0) {
        node.left = deleteNode(node.left, key);
    } else if (cmp > 0) {
        node.right = deleteNode(node.right, key);
    } else {
        if (node.left == null) {
            return node.right;
        } else if (node.right == null) {
            return node.left;
        } else {
            Node temp = node;
            node = findMin(temp.right);
            node.right = deleteMin(temp.right);
            node.left = temp.left;
        }
    }

    return node;
}
```

```java
public V lookup(K key) {
    Node node = findNode(root, key);
    return node == null ? null : node.value;
}

3 usages
private Node findNode(Node node, K key) {
    if (node == null) {
        return null;
    }

    int cmp = key.compareTo(node.key);
    if (cmp < 0) {
        return findNode(node.left, key);
    } else if (cmp > 0) {
        return findNode(node.right, key);
    } else {
        return node;
    }
}
```

AVL Tree

AVL tree is a tree data structure that each of its subtrees is a balanced binary tree.

In a binary tree the balance factor of a node X is defined to be the height difference.

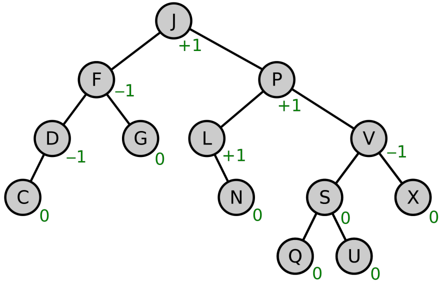
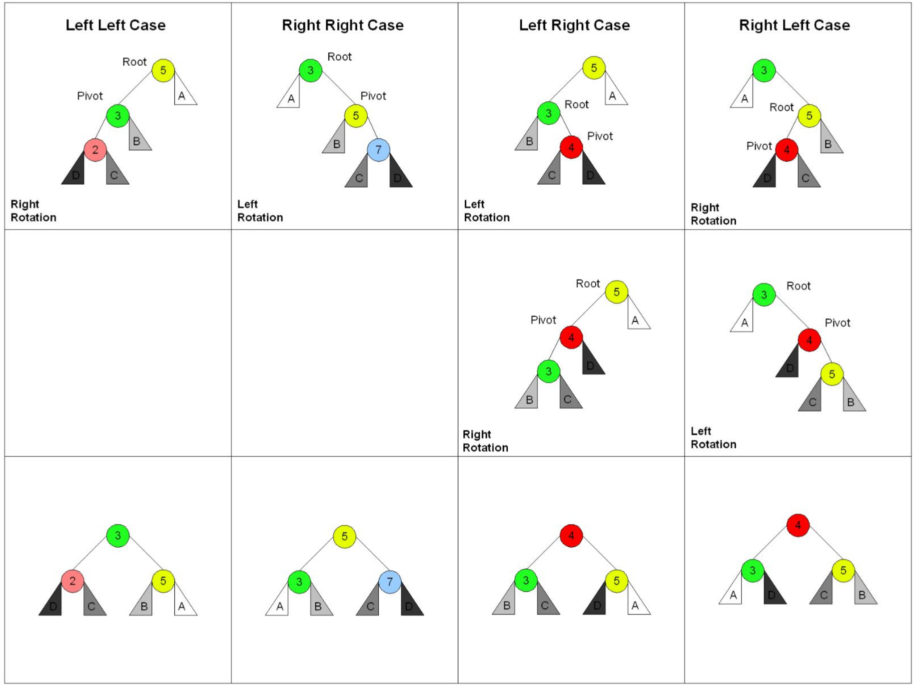


Fig: AVL tree with balance factors (H(R)-H(L))

AVL Tree




```python
def R_rotate(self, node): # LL case
    tmp_tree = node.left
    node.left = tmp_tree.right
    tmp_tree.right = node
    node.height = max(self.height(node.right), self.height(node.left)) + 1
    tmp_tree.height = max(self.height(tmp_tree.left), node.height) + 1
    return tmp_tree

def L_rotate(self, node): # RR case
    tmp_tree = node.right
    node.right = tmp_tree.left
    tmp_tree.left = node
    node.height = max(self.height(node.right), self.height(node.left)) + 1
    tmp_tree.height = max(self.height(tmp_tree.right), node.height) + 1
    return tmp_tree

def RL_rotate(self, node): # RL case
    node.right = self.R_rotate(node.right)
    return self.L_rotate(node)

def LR_rotate(self, node): # LR case
    node.left = self.L_rotate(node.left)
    return self.R_rotate(node)
```

`LL case`: node `5` single R rotate

`RR case`: node `3` single L rotate

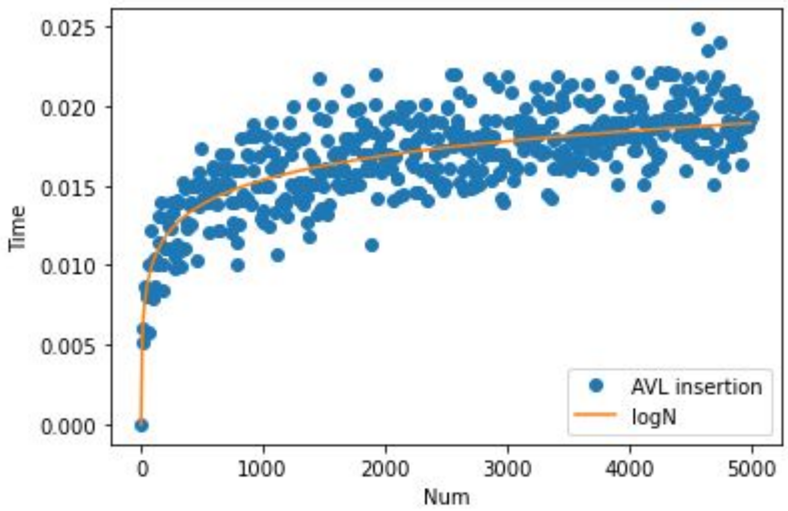
`LR case`: node `3` L rotate, node `5` R rotate

`RL case`: node `5` R rotate, node `3` L rotate

AVL Tree

Operation/case	Average case	Worst case
Insert	O(logN)	O(logN)
Delete	O(logN)	O(logN)
Search	O(logN)	O(logN)

Insertion

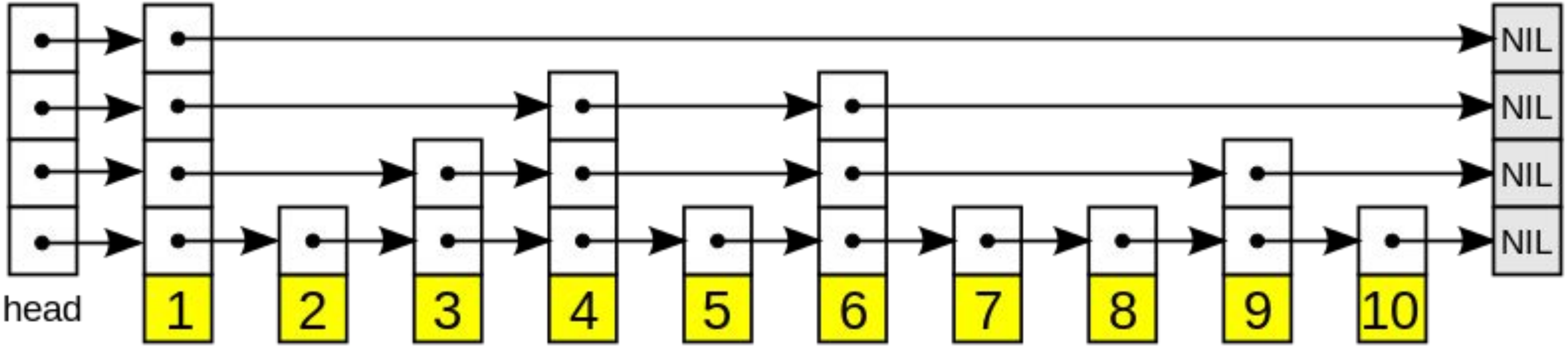


Skip List

Can we use linked list to implement an ordered dictionary with O(logn) insertion / deletion / lookup cost?

Yes! Skip List = Linked List + multiple level pointers.

Redis uses skip list for ordered data structure.

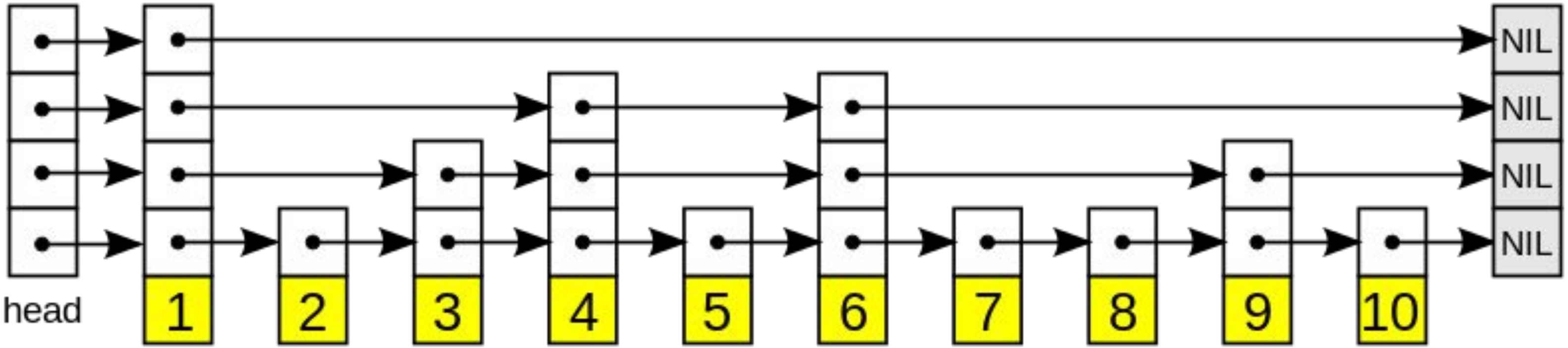


Skip List

Insertion: add one more level with probability until we fail. (Geometric distribution)

Delete: just like linked list

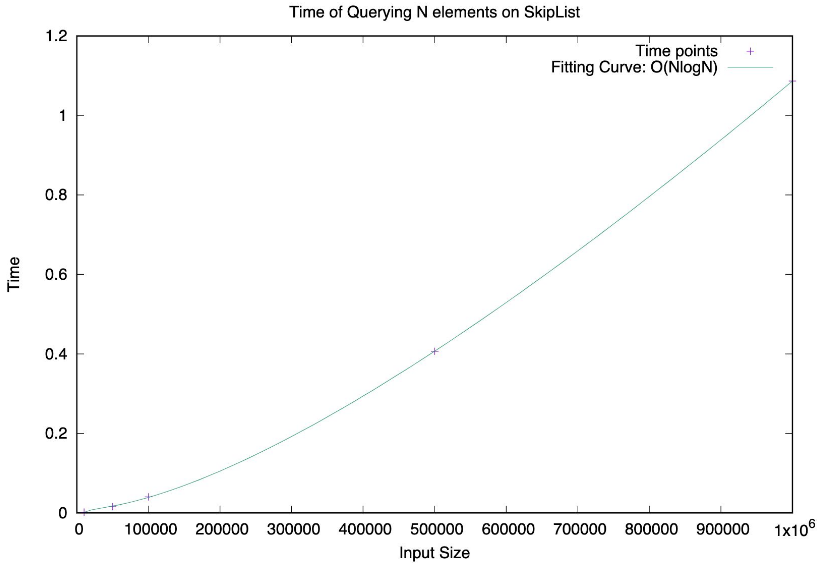
Lookup: search from the highest level to the lowest level



Skip List

Implement Skip List in C++ 17

Evaluation it on a server with a Intel(R) Xeon(R) Gold 5317 CPU and 256GB memory, Ubuntu 20.04



Conclusion

There are various data structures can support ordered dictionary

Choose the most suitable one for your application