Ordered Map and BST

Ordered Map

Ordered Maps is a data structure similar to that of a map, but with the keys being ordered. This is because, although a Map DS can enable us to find, insert and delete in O(1) time, we are unable to do further analysis in terms of the distribution of items and comparisons between items.

Thus, a notion of ordering is required.

The key operations are:

- Search(x) – Find if a certain element is inside
- Insert(x) – Insert new element
- Remove(x) – Delete existing element
- GetMax
- GetMin
- GetMedian
- ListSorted – Return the list of elements sorted
- Successor(x) – Find the smallest element bigger than x
- Predecessor(x) – Find the largest element smaller than x
- Rank(x) – Find the rank/index of x
- Select(i) – Find the element at rank i
- LargerThan – get all elements greater than this item
- SmallerThan – get all elements smaller than this item

Ordered Map Implementation #1: Array

We can technically use an array to perform operations as required above. However, it can be very inefficient.

Unsorted Array

Operation	Time Complexity	Analysis
Search	O(N)	Scan through all elements in array
Insert	O(1)	Insert to the back since order does not matter
Remove	O(N)	O(N) search and O(N) gap closing after deletion
GetMax	O(N)	Scan through all elements in array
GetMin	O(N)	Scan through all elements in array
GetMedian	O(N log N) / O(N)	Sort the array (N log N) or use QuickSelect (N)
ListSorted	O(N log N)	Comparison-based sorting
Successor	O(N)	Scan through all elements in array
Predecessor	O(N)	Scan through all elements in array
Rank**	O(N log N) / O(N)	Sort the array or use QuickSelect
Select*	O(N log N) / O(N)	Sort the array or use QuickSelect
LargerThan	O(N)	Scan through all elements in array
SmallerThan	O(N)	Scan through all elements in array

*Select(k) returns the value of the kth smallest element (1-based index)
**Rank(v) returns the rank k of element with value v (1-based index)

Sorted Array

Operation	Time Complexity	Analysis
Search	O(log N)	Binary Search
Insert	O(N)	Binary Search + Creation of gap to insert
Remove	O(N)	Binary Search + Close gap
GetMax	O(1)	Return last element
GetMin	O(1)	Return first element
GetMedian	O(1)	Return middle element
ListSorted	O(N)	Still need to iterate through the array
Successor	O(log N)	Binary Search and add 1 to index
Predecessor	O(log N)	Binary Search and minus 1 from index
Rank	O(log N)	Binary Search and return index + 1
Select	O(1)	Access array at the index
LargerThan	O(N)	Binary Search and iterate through array
SmallerThan	O(N)	Binary Search and iterate through array

Ordered Map Implementation #2: Binary Search Tree
A BST is made up of vertices. For every vertex, we define:

- x.left = left child of x
- x.right = right child of x
- x.parent = parent of x
- x.key = value of x

Similar to the Binary Heap, the BST also has a BST Property to maintain:

- For every vertex x and y
    - y.key < x.key if y is in left subtree of x
    - y.key ≥ x.key if y is in right subtree of x

In other words, everything to the right of a certain vertex has a value bigger than itself. Everything to the left has a smaller value. This is unlike a Max Binary Heap, where a parent has a larger value than both of its children.

BST also has a root, leaves and internal vertices.

BST Implementation #1: Sequential Array
One possible way to implement a BST is similar to that of a Binary Heap. We initialise a sufficiently large array, with empty slots having a placeholder value of -1. Using the same positioning logic as with Binary Heap, we can insert the values accordingly.

However, this method has huge limitations:

- Inefficient usage of space, as unlike a binary heap, a BST is unlikely to be a perfect tree, and will have gaps in its branches. This can result in worst case $O(2^n)$ space usage, as all n elements are the right child/left child of each other.
- Cache misses due to large jumps in indices

BST Implementation #2: Parent Array
We can use a similar logic as UFDS and have an array p, with p[i] holding the index of the parent of vertex i. However, this method will work for generic trees, as we cannot easily identify if an element is the right or left child, and will only be allowing traversals upwards towards the parent.

As such, this implementation is not very practical, and does not satisfy the requirements.

BST Implementation #3: Nodes
The BST is best represented using nodes. The concept of a node is similar to that of a LinkedList node – it contains the addresses to its parent, left child and right child. It also contains a key, which is its value.

1. <u>Search</u>

Starting with the root node, we look at the current node value. If the value is too big, we continue searching from the left child. If it's too small, we search from the right. child. If found, then return the value. If the node is null, then return -1 for not found.

- O(h)

2. Insertion

We use the same logic as in search, except that we only stop when we find a null node. Once found, we create a new node and return it, and recursively assign the node as the appropriate child.

- O(h)

3. GetMax

Recursively visit the right child until we reach a node with no right child, then return that node's value.

- O(h)

4. GetMin

Recursively visit the left child until we reach a node with no left child, then return that node's value.

- O(h)

5. Successor(x)

The successor is the immediate node with a value higher than x (which can be found in the tree).

```
if this.right != null return findMin(this.right)
else
      p = this.parent, T = this
      while(p != null && T == p.right)
            T = p, p = T.parent
      if p is null return -1
      else return p
```

The logic is as follows:

- If the node with value x has a right child, then the minimum of the right subtree would be the successor
- If it does not and it has a parent, it needs to keep going up until it finds a parent a that is a left child of its parent b. We then return that parent b's value.
    - Why left child? This is because so long the parent is the right child of its parent, we get smaller and smaller values as we go upwards.
    - However, the moment there is a "right turn", we meet a parent whose value is bigger than everything before it.
    - That parent would then be the successor of the node with value x, as the node with value x would be its predecessor i.e. largest value of its left subtree.
- O(h)

6. Predecessor(x)

The predecessor is the immediate node with a value lower than x (which can be found in the tree).

```
if this.left != null return findMax(this.left)
else
      p = this.parent, T = this
      while(p != null && T == p.left)
            T = p, p = T.parent
      if p is null return -1
      else return p
```

The logic is as follows:

- If the node with value x has a left child, then the maximum of the left subtree would be the predecessor
- If it does not and it has a parent, it needs to keep going up until it finds a parent a that is a right child of its parent b. We then return that parent b's value.
    - Why right child? This is because so long the parent is the left child of its parent, we get larger and larger values as we go upwards.
    - However, the moment there is a "left turn", we meet a parent whose value is smaller than everything before it.
    - That parent would then be the predecessor of the node with value x, as the node with value x would be its successor i.e. smallest value of its right subtree.
- O(h)

7. ListSorted

To list the items in a tree in sorted order, we will need to do inorder traversal:

Inorder Traversal

- If current node is null return
- Visit the left child
- Process the current node value
- Visit the right child

This will return the tree in sorted increasing order.

Other ways of traversal include:

Preorder Traversal

- If current node is null return
- Process the current node value
- Visit the left child
- Visit the right child

Postorder Traversal

- If current node is null return
- Visit the left child
- Visit the right child
- Process the current node value

Overall Complexity: O(3n) = O(n) as we visit each node 3 times

8. Deletion

To delete a node with children effectively, it is best to replace it with another node such that minimal bubbling up is required. In the case of deletion, the successor of the node being deleted is the best candidate to use for replacement.

There are 3 cases when it comes to deletion:

- Vertex v has no children
  - Simply remove vertex v – O(1)
- Vertex v has 1 child
  - Connect child to parent – O(1)
  - Remove v – O(1)
- Vertex v has 2 children
  - First find the successor x of v – O(h)
  - Replace v.key with x.key – O(1)
  - Recursively delete x in v.right – O(h)
- Overall O(h)

But why successor?

- Because the successor has at most 1 child, which makes it easier to delete. It will also not violate the BST property.
  - Vertex x has two children
  - Therefore, vertex x must have a right child
  - Successor of x must be the minimum of the right subtree
  - A minimum element of a BST has no left child
  - So successor of x has at most 1 child

9. Rank

The rank of an element is its position relative to the sorted order, i.e. the kth smallest item would have a rank of k. To calculate that, we need to start from the root:

- Check if value to be ranked is smaller than, larger than or equal to the value of the current node
- If smaller, recursively return rank(curr.left)
- Else if equal, return curr.left.size (number of elements in left subtree) + 1
- Else if larger, return curr.left.size + 1 + rank(curr.right)
- Overall O(h)

10. Select

Select is the reverse of rank. Given you a rank, return the value of the node with that rank. We will need to start checking from the root:

- Check if rank needed is smaller than rank of current node (curr.left.size +1), equal or larger
- If equal, return current node's value
- If smaller, recursively call with curr.left and rank k
- If larger, recursively call with curr.right and rank (k-rank(curr))
- Overall O(h)

11. GetMedian, LargerThan, SmallerThan

With rank and select established, all of these functions are significantly easier to code.

Worst Case for a BST

The worst case for a BST is when height = N. This occurs when we keep inserting increasingly larger numbers or smaller numbers. To prevent this, AVL trees will be needed.

Summary of BST

Operations that modify the BST (*dynamic* data structure):

- insert: O(h)
- delete: O(h)

Query operations (the BST structure remains the same):

- search: O(h)
- findMin, findMax: O(h)
- predecessor, successor: O(h)
- inorder traversal: O(N) – the only one that does not depend on h
- select/rank: O(h)