**Storing the data**

For n, r1 and r2. We have to maximize the value of n as ‘n +n\*0.1’ as we know that 10% each operation is going to be performed and we would need more 10% size not more than that because both add and delete are being performed. This ‘n +n\*0.1’ is the total size or the total elements our array can have. But the size required to store these total numbers will be different in both approaches.

1-Number store approach: (n < r2-r1+1) Storing number in array

So, the total size for this approach will be equal to ‘n +n\*0.1’. As per assignment we can generate ‘n +n\*0.1’ random numbers.

Now we can sort the array using the qsort function. Sorting makes our time taken by each operation to complete less as instead of searching sequentially we can binary search the array and perform each operation more efficiently in less time.

And after that we have to distribute 10% negative 1 uniformly inside array denoting invalid or empty values so at the end, we will have n random numbers and 10% -1 which might completely get filled after random operations. The best way to add -1 uniformly through our array is to change every 10th element in our array to -1 but we are not allowed to put a negative into a sequence of duplicate numbers like we cannot have a sequence like {…,2,2,2, -1,2,2, 2,….}. This distribution of -1 probably has both pros and cons which are mentioned below.

2-Count store approach: (n>=r2-r1+1)

We know that our array will have duplicates .So instead of storing numbers in our array we can store the counts of numbers but for storing counts we have to make sure that the size of array will be ‘r2-r1+1’ (range) because for range ‘r1’ to ‘r2’ we can have ‘r2-r1+1’ random numbers that can be generated.so the required size will be ‘r2-r1+1’ for storing count of each number.

We can initialize our array of size range with 0 and each index ‘i’ will store the count for ‘i-r1’ value since our range can start from number greater than 0 i.e. r1>0.

for ex: if we have range r1=4, r2=15

Total numbers that can be generated in range = 15-4+1=12

We will need a 12-sized array for storing counts. For instance, if our array after generating the numbers looks like:

ar= {1,0,4,3,4,2,0,1,4,2,3,4}

now each index ‘i+r1’ will denote number.

|  |  |  |
| --- | --- | --- |
| Index: i | Number: i+r1 | count |
| 0 | 4 | 1 |
| 1 | 5 | 0 |
| 2 | 6 | 4 |
| 3 | 7 | 3 |
| 4 | 8 | 4 |
| 5 | 9 | 2 |
| 6 | 10 | 0 |

And similarly for other elements. If we try to look at our real data, it will look like this.

Total count=sum(ar)=28

So, we will have 28 numbers in the range 4-15 including duplicates.

4 6 6 6 6 7 7 7 8 8 8 8 9 9 11 12 12 12 12 13 13 14 14 14 15 15 15 15

*Note that:*

1. *Our data is in sorted format by default because our index denotes the number. This technique is also known as counting sorting which is used to sort limited set of values.*
2. *Our array can store a large amount of data within range. But in our case, we have only limited amount we have to add (n+ n\*0.1) but we can add many.*
3. *We can also implement this approach if our range is greater than n, but it is not memory efficient if our range is so much greater than n.*

*For ex: n=100, r1=1, r2=100000 and range=100000*

*Now in that case we have to allocate 100000 size for storing 100 numbers and only 100 out of 100000 space will be occupied. There will be a lot of memory unused.*

Effects of sorting our array in number store approach

If we look the main reason, it’s because it takes less time in searching elements which would affect our operations:

1. Find- Sequential or linear search takes long time than binary search.  
   And for applying binary search our array must be sorted.
2. Add – The only advantage of not sorting array is in adding operation as we don’t have to care about position where to add the new element. We can add the element whenever we encounter first -1 and we don’t have to take care of shifting elements which will probably take less time.

In the case of sorted array, we have to first find the position where to add element and add it by first shifting the elements if the position is not empty (-1) and after that adding it to the position.

1. Delete- Deletion will be faster in the case of sorted array as we can search the element position faster than in unsorted array.
2. Predecessor and successor- finding predecessor and successor in unsorted array will be bit complex as we have to find the number which is closest smaller element as predecessor and closest larger element as successor. So, we have to traverse whole array in worst case.  
   But in case of sorted array its lot simpler just find the position of element and the predecessor will be first valid element before our targeted number and successor will be first valid element after our targeted number.
3. Min and Max: In case of unsorted form, we have to traverse the whole array comparing the elements and updating our min/max value. But in the case of other its straightforward return the first and last valid elements as min and max.

So, we can conclude that sorting will reduce the overall time performed by operations and hence decreasing the amount of time it takes to completely execute our code.

Effects of including -1 in our array in number store approach:

Pros:

1. As -1 denotes invalid input we can use this advantage in deleting the elements by simply making the position invalid by replacing it with -1.
2. Adding the new elements will probably be fast as -1 can also be considered as empty spaces and secondly if our position where our new element would exist is not empty, we have to find the nearest -1 and shift the elements till that position.

Cons:

1. Now finding the elements will become complex as for binary search our array needs to be in sorted order and by including -1 it’s breaking our order. We need to modify our binary search algorithm to deal with -1.
2. The other operations min, max, pred and succ are also going to take a little bit of time as we have to find the first valid element.

**Searching algorithm (Binary search with modification):**

There are two basic searching algorithms:

1-Linear Search (Sequential)   
2-Binary Search – Required sorted array

The best way to identify which algorithm to use is to check whether our array is in sorted format or not. If its sorted it’s best to use binary search as it decreases the time complexity from O(n) to O (log n) as it uses divide and conquer algorithm by dividing the array and discarding one half based on condition.

As in the case of linear search it sequentially searches the array until our target element is found.

The real problem:

Since we have -1 one in our sorted array that denotes invalid values, the binary search algorithm will not work as expected.

*So, the real problem is we need an algorithm to search the elements in sorted array which includes -1 denoting empty space.*

Solution

Our binary search works as comparing mid element to the target element and based on that updating our two pointers left and right.  
Now if our mid element is -1, we cannot compare it with target element as it will be always greater than -1 and our left pointer will update. So, we need to put condition we if encounters -1 we do not directly update our pointers. We will reduce our mid until we get a valid element . Now we can compare it and update our pointers.

* If the value at index mid is equal to the target value v, the result is updated to mid, and the right boundary is adjusted to continue searching for a lower index where the same value might appear again.
* If the value at index mid is less than the target value v, the left boundary is adjusted to mid + 1, and any consecutive -1 values are skipped.
* If the value at index mid is greater than the target value v, the right boundary is adjusted to mid - 1, and again, any consecutive -1 values are skipped.

And if our left will be greater than the right pointer our loop will terminate indicating we don’t find our element so we can just return negative of left.

This is the modification we need to add in our binary search algorithm.

**Operations on data**

1. Number store approach:

Adding element:

We know that our array is sorted, and other main thing is it contains -1 which can be filled with valid values. But we cannot add the new element anywhere in the array, we have to keep the sorted order.

So first we have to find the position of the new element using binary search algorithm. Our search algorithm will return negative of position if the target element is not found. So, we convert the position returned to absolute value of position.

After that we need to check if that position is available (it contains -1) and we can also check if position before that is filled with -1. If it is, then we can simply add our element there.

If that’s not the case, we will have to search the position of nearest invalid value and shift the elements from that position either left or right

based on which side we find our position until the target value position.

For ex: if our array is: 2 5 -1 6 9 10 -1 46 47 new: 7

Binary search will return position as -4

So, abs (-4) =4

The position and position-1 are not available as it is filled with 9 and 6. Now we have to find the nearest -1.

The nearest -1 is before 6 which is position 2. So will shift elements after that position to left till one less than new element position i.e., 3.

After shifting, the array will look like this.

2 5 6 6 9 10 -1 46 47

Now the 3 position can be replaced with our new element.

2 5 6 7 9 10 -1 46 47

Deleting element:  
For deleting we need to first search the array for the element if the element position is found we will replace the element at that position with -1 making it invalid.

Ex: 2 5 6 6 -1 8 9 9 9 -1 10 11 delete (6)

Pos=2, we can replace that position with -1.

After replacing: 2 5 6 8 8 9 9 9 9 10 11

Finding element:

The search algorithm returns the position of first occurrence of the number if it exists. For counting total occurrence, we will start traversing the array from that position and incrementing our count flag until we found the element which is not equal to number whose occurrence we are finding.

Ex: 2 2 2 2 5 5 6 6 6 6 -1 10 10 14 14 14 -1 find (6)

Binary search will return 6 from that position we increase our count until

We encounter different numbers. In this case the loop will stop when it encounters -1 returning the count value as 4

Min and max:

These are straightforward just return the first and last valid element.

For finding min we have to traverse the array from start until we found the first valid element.

For finding max we have to traverse the array from back until we found the first valid element which will be the maximum value in the array.

Finding successor and predecessor:

The steps are as follows:

1. Get the position of first occurrence of the number.
2. The predecessor will be the first valid value before that number.

So, traverse the array from that position to left until we find the first valid input.

1. The successor will be the first valid element after that number which is not equal to the number itself. So, traverse the array from that position to right until we find the first valid element which is not equal to the number itself.

2.Count store approach:

Adding:

Adding number in count store approach is simple we just need to increment the count of the index ‘v – r1’ where v is our new number.

Adding the new number will be faster here than the number store approach and we don’t have to find position and we don’t have to perform any shifting.

For example: ar [] = {1,0,3,5,6,2,2,1,0,3,0} r1=5,r2=15

Add (8): The index will be 8-5=3

We can increase the 3rd index value by 1.

After incrementing, our array will be:

1 0 3 6 6 2 2 1 0 3 0

Deletion:  
 For deletion we decrease the count of the index ‘v-r1’ by 1 where v is our number to delete.

This operation is also faster compared to the number store approach as we don’t have find the numbers position in order delete the number.

Finding:  
The index ‘v-r1’ stores the count of the v (number). So, we can simply return the value as the total number of occurrences.

Finding occurrence is also faster here because we don’t have to deal with finding the position of the element and traversing the array for counting.

Min and Max:  
Our array is in sorted order by default we have to find first non-zero element in the array and return the ‘index+r1’ as minimum value and similarly for max value we will find the last non-zero element and return the ‘index+r1’ as maximum value.

Time taken for finding minimum and maximum is probably same for both the cases as both have to traverse the array from start or back until they find a valid value (non-zero or value other then -1).

Predecessor and successor:  
For predecessor and successor of element ‘v’ we will find the first non-zero value after the index ‘v-r1’ and similarly for predecessor we will find the first non-zero element before the index ‘v-r1’ and return the result   
as ‘index+r1’, where ‘index’ is first non-zero element index.

The time taken to perform this action is faster than the number store approach as in that approach we have to first find the position of the element ‘v’.

*Conclusion:*

*The performance of the count store approach is better than number store approach in both the aspects time and space because it performs operations faster and can store a large amount of data in less space.*

*We can clearly see the difference in the performance test outputs.  
Sample1 shows the result for the count store approach and Sample2  
shows the result for the number store approach.*

*Insert the results for n=100000000 r1=1 r2=100000 sample1*

*And for sample 2 n=100000 r1=1 r2=1000000*

*As we can see the amount of data in sample1 is 1000 times more than sample 2 and still memory usage is less than sample2 and there is not much difference in overall time taken to perform each operation.*

**Data Compression techniques:**

Run-Length Encoding (RLE):

RLE is a simple and effective compression technique for data with long runs of the same value. In sorted data with duplicates, you often have consecutive values that are the same. RLE encodes these runs as a single value followed by a count. For example, instead of storing "99999," you store "9" followed by "5.

Example:

Original: 1 1 1 1 1 2 2 3 4 4 4 5 5

RLE: 1 5 2 2 3 1 4 3 5 2

Since this technique changes the array and makes it unsorted, we can’t perform binary search in it. For adding and deleting operations we have to perform sequential search and update our array.

And similarly, for finding, predecessor and successor operations we have to perform sequential search.

But in case of finding min and max it’s probably the same returning second last and first element.

The main advantage of using this technique is the space utilized will be reduced as we can see clearly in the example.

Delta Encoding:

In sorted data, neighboring values often have small differences. Delta encoding encodes the difference between each value and the previous one. This technique is particularly effective if the differences between values are small.

Example:

Original: 10 15 17 18 21 22 25

Delta: 10 5 2 1 3 1 3

Again, here the we cant perform binary search as the elements are completely changed we have to perform sequential search and we also will have to keep track of the sum of elements while traversing and searching for target element.

Adding and deleting:

For adding a new element to array we have to first find the position where I would exist and after that adding it there and also updating one element after it. And it’s similar for deletion as well finding the position, deleting the element and updating one element after it.

For updating the next element, we only need to add or decrease the new added or deleted element.

In above example if 19 is being added then the Delta array will look like this

10 5 2 1 1 2 1 3

Pred and succ:

For these two operations while performing search if we found our target element, we can add next element in the sum and for predecessor we can decrease sum by our element.

Finding occurrence:

If we have duplicates in the data our delta array will have 0 denoting the number is being repeated so, if we find our element we have to count the total zeroes after it and add 1 to it for counting the total occurrences.

Max and min:

Min will be the first element and max will be sum of all elements.

Dictionary-Based Compression:

For data with many duplicates, we can use a dictionary to store unique values and replace duplicates with references to the dictionary. This technique works well when you have a relatively small set of unique values.

Example:

Sorted Array: 2 2 3 3 3 4 4 5 5 5

Dictionary:

Index Value

0 2

1 3

2 4

3 5

Compressed Array: 0 0 1 1 1 2 2 3 3 3

The efficiency of dictionary-based compression on a sorted array can impact various operations differently:

Adding Elements: When adding new elements to the sorted array, you need to ensure that the array remains sorted. This operation can be relatively expensive because inserting a new element while maintaining the sorted order might require shifting a portion of the array to make space for the new element. Additionally, if the added element is not in the dictionary, you need to update the dictionary, which can also introduce some overhead.

Deleting Elements: Deleting an element from the sorted array requires finding the element, which can be done efficiently using binary search due to the sorted nature of the array. After finding the element, you'll need to remove it from the array, which may involve shifting elements to fill the gap. Deleting an element can also require updating the dictionary if the deleted element is not present elsewhere in the array.

Finding Count of Elements: Since the array is sorted and encoded, finding the count of a specific element involves searching for it in the encoded array, which can be done efficiently using binary search. Once you find the element, you can count its occurrences.

Predecessor and Successor: Finding the predecessor (the largest element smaller than a given element) and successor (the smallest element greater than a given element) can also be efficiently done using binary search on the sorted, encoded array.

Min and Max Operations: The minimum and maximum elements can be quickly found in a sorted array without having a significant impact on efficiency. You can directly access the first and last elements of the sorted array.