**INTERVAL SKIP LIST**

Team members

|  |  |
| --- | --- |
| Name | Roll number |
| Akkina aatesh | CB.EN.U4CSE21404 |
| BVV Satyanarayana | CB.EN.U4CSE21409 |
| Bhavana.N | CB.EN.U4CSE21411 |
| Nadimpalli Ujwal Srimanth Varma | CB.EN.U4CSE21440 |
| Potu Tejaswi | CB.EN.U4CSE21445 |

**Introduction**

Hybrid data structures combine the characteristics and benefits of multiple data structures to address specific problem-solving requirements efficiently. By combining different data structures, hybrid structures can leverage the strengths of each component to optimize various operations and improve overall performance.The significance of hybrid data structures lies in their ability to solve complex problems efficiently by providing a tailored solution that meets specific needs. Here are some key reasons why hybrid data structures are important:**1. Performance Optimization:** Hybrid data structures are designed to optimize specific operations or a combination of operations. They can achieve better time and space efficiency by leveraging the strengths of different data structures. For example, a hybrid structure may use a combination of a hash table and a balanced tree to provide efficient search, insertion, and deletion operations simultaneously.**2. Flexibility and Adaptability:** Complex problems often require multiple operations, and a single data structure may not efficiently handle all of them. Hybrid structures allow for adaptability by combining different data structures in a way that best suits the problem at hand. This flexibility enables the structure to perform efficiently across a wide range of use cases.**3. Trade-offs and Balancing Constraints:** Different data structures have trade-offs in terms of time complexity, space complexity, and other performance characteristics. By combining multiple structures, hybrid data structures can balance these trade-offs to meet specific requirements. For example, a hybrid structure may use a combination of a linked list and an array to balance the need for efficient insertion and removal operations with efficient random access.**4. Domain-Specific Problem Solving:** Hybrid data structures are often designed to solve problems that are specific to certain domains or applications. These structures are optimized for the unique requirements of those problems and may include specialized algorithms and data organization strategies. By combining elements from various structures, hybrid data structures can provide tailored solutions that outperform traditional, single-structure approaches.**5. Scalability and Efficiency:** Hybrid data structures are often designed with scalability in mind, allowing them to efficiently handle large amounts of data or growing datasets. By leveraging the strengths of different structures, hybrid designs can distribute the workload effectively, reduce bottlenecks, and provide better overall efficiency.Overall, hybrid data structures offer a powerful approach to solving complex problems efficiently. By combining and integrating multiple data structures, they can provide optimized solutions that address specific requirements, offer improved performance, and adapt to various use cases.

**Overview:-**

Our data structure is Interval skip list. But all the operations will take a lot of time when the number of intervals in the skiplist are very large (like 10 lakh). If the number of distinct intervals stored in the skip list are 10 lakhs the number of nodes will be surely more than 10 lakhs in the skip list.

The above mentioned scenario is the limitation of skip list . So to overcome this is we made a hybrid data structure by combining a BST and a interval skip list.

An interval skip is a skip list which stores intervals rather than numbers.

So in our Hybrid data structure we have a BST. Each node of a BST stores a interval skip list.

Every node of BST stores 2 things, Range of Intervals and a Skip list.

So for example in a node of BST the range of interval stores is [1-1000] then the node has a skip list with intervals only in range of [1-1000].

25k-50k

32-33

1-100

32-33

1-100

1k-10k

75k-100k

32-33

27-99

1-100

50k-75k

1-1000

All other intervals

10k-50k

The skip list stored in that node

By default the skip list has intervals in sorted order. It is sorted based in the first element of the interval. If the first element of some two are intervals are same then we sort them based on the interval length.

[1-100],[1-50],[2-100] are sorted as [1-50].[1-100],[2-100]

**Advantages:-**

By combining the BST and Interval Skip we combine the advantages of BST and Interval Skip List. Actually we are making the interval skip list more efficient so naturally it has all the advantages of Interval skip list and some extra advantages everything will be listed below.

**Efficient interval search**: Interval skip lists excel at interval-based search operations. They allow for efficient retrieval of intervals that overlap with a given query interval. This makes them suitable for applications involving range queries, such as interval-based database queries or event scheduling systems.**Reduced search complexity:** Interval skip lists provide a balanced search structure, similar to traditional skip lists. With multiple levels of indexing, they reduce the search complexity from O(n) in a sorted array or linked list to an average case of O(log n), where n is the number of intervals stored. This faster search capability becomes increasingly advantageous as the size of the data set grows.**Space efficiency:** Interval skip lists do not require additional storage space compared to traditional skip lists. They achieve interval indexing by simply augmenting each node with interval information. This means that the memory overhead is generally low and proportional to the number of intervals stored.**Dynamic updates:** Interval skip lists support efficient dynamic updates, including insertion and deletion of intervals. When an interval is inserted or removed, the structure is adjusted accordingly, ensuring the skip list remains balanced. This property makes them suitable for applications where intervals are frequently added, modified, or removed.**Simplicity of implementation:** Interval skip lists are based on the skip list data structure, which is relatively easy to understand and implement. The core operations, such as searching and updating, are similar to those in skip lists, with only minor modifications needed to handle intervals. This simplicity makes interval skip lists accessible to developers and researchers.**Versatility:** Interval skip lists can be adapted to various data types and use cases. While the primary use is for interval search, they can also be extended to handle other types of queries or customized to suit specific requirements. This flexibility makes them a valuable tool for a wide range of applications.

So the above mentioned are the advantages of interval skip lists.

Now we integrated it with BST we make it further more better:-

1)we reduce the comparisons while searching,insertions,deletion in the Hybrid interval skip lsit compared to a normal interval skip list.

2)we reduce the time complexity by almost half for every operation which is detailly mentioned with an example in performance evaluation.

3)we make it more efficient for larger number of datasets which is the case in an real life scenario

4)But the space we use is more as we accommodate extra space for a BST

**Implementation**

So for implementation of this we used a class interval skip list and a BST. The SkipList class contains all the insertion search and deletion operations of our data structure.

BST class contains an add\_child attribute which creates each of the 7 nodes of a binary tree.

So now lets straightly go to insertion , deletion and search how are they implemented in our code.

Lets start with search. Search\_helper Function call

If root is none

Search operation

Curr=self.head  
finalist=[]

Give the input range you want to search.

NO YES

curr.next.min>min

return

If range in root.range

( in

Calls search\_helper function(root,min,max)

**Search Function call**

curr.next.min <= min and curr.next.max >= max

**NO YES NO**

search(root,max,min)

Search\_helper(root.left,min,max)

Cur=cur.down

**YES**

Finalist.append  
curr=curr.next

**NO**

Curr=curr.next

Search\_helper(root.right,min,max)

**Explanation of search operation:-**

We first input the range we want to search for. The lower bound of the input is stored in a local variable min and the upper bound is stored in the local variable max.

Now a search helper function is called where root of BST ,min ,max are passed as parameters. We know that BST stores two things one self.range (named self.ele in code) and a skiplist(self.skip). So the range element determines which intervals can be stored in the intervals . if range is [1-1000] means only ranges between that can be stored in the linked list.

So we check whether the given range is within the self.range . if yes then we call root.skip.search() function which checks whether the range is present in the skiplist . if the input range is not in the range of self.range we recursively traverse the tree.

Now lets see what happens in the search function:-

In the search function we define a final list which stores in which all intervals the given range is present.

It starts from the self.head of the top most level.

It compares with next element lower bound if the next element lower bound is higher then there of no chance of finding that element in that level so it goes down by one level .

if the next element lower bound is less than the lower bound of the element we are searching for but the upper bound for the search element is more than the next element the upper bound then the range we r searching cannot be there in that interval so we move right as there is a possibility of finding the element in the next level

if the next element lower bound is less than the lower bound of the element we are searching for but the upper bound for the search element is less than or equal to the next element the upper bound then our range is there in that interval so we add it the final list

at last we display the final list

Delete\_helper

delete operation

If root is none

delete function

NO

Give the input range you want to delete

..

Curr=self.head

YES

If range in root.range

( in

Calls delete\_helper function(root,min,max)

return

curr.next.min>=min

YES NO

Search\_helper(root.left,min,max)

curr.next.min <= min and curr.next.max >= max

search(root,max,min)

**NO**

**No Yes**

Search\_helper(root.right,min,max)

Cur=cur.down

Curr.next=curr.next.next

Curr=curr.next

**Explanation of delete function:-**

We first input the range we want to delete for. The lower bound of the input is stored in a local variable min and the upper bound is stored in the local variable max.

Now a search\_helper function is called where root of BST ,min ,max are passed as parameters. We know that BST stores two things one self.range (named self.ele in code) and a skiplist(self.skip). So the range element determines which intervals can be stored in the intervals . if range is [1-1000] means only ranges between that can be stored in the linked list.

So we check whether the given range is within the self.range . if yes then we call root.skip.delete() function which checks whether the range is present in the skiplist . if the input range is not in the range of self.range we recursively traverse the tree.

Now lets see what happens in the delete function:-

We start from the head of the top most node and we search for the element which we want to delete so this will be almost similar to the search function

We start from the head of the top most level

It compares with next element lower bound if the next element lower bound is higher than or equal then it checks whether the the lower bound of the element to delete and the next element is same or not if same even checks for upper bound if same then it deletes it.(for deleting it just changes the pointers simple singly linked list deletion). If upper bound is not equal then it goes down as if it is not equal then it will be greater than that

Else if the next element lower bound is less then it moves to the right.

The entire above 2 process will be in a while loop as we need to delete all the occourences of the range in all the levels.

**Insert Operation**

We first input the range we want to insert . The lower bound of the input is stored in a local variable min and the upper bound is stored in the local variable max.

Now a insert \_elper function is called where root of BST ,min ,max are passed as parameters. We know that BST stores two things one self.range (named self.ele in code) and a skiplist(self.skip). So the range element determines which intervals can be stored in the intervals . if range is [1-1000] means only ranges between that can be stored in the linked list.

So we check whether the given range is within the self.range . if yes then we call root.skip.insert() function which checks whether the range is present in the skiplist . if the input range is not in the range of self.range we recursively traverse the tree.

Now lets see what is in the insert function.

So there is self.ple which is the current number of levels in the skip list. Levels stores in how many levels should the element be inserted. So first a random function is called if it returns less than 0.5 levels gets incremented else they will not be implemented.

Next we check whether levels>self.ple if yes then we have to create new levels to insert the interval into that node . so we have to create (levels-self.ple) levels. So we use a while loop to create levels. Creating levels means creating a head for each level with the interval [none,none]. All the head nodes in each nodes are interconnected using the down pointer.

After we create new level we search for correct place to insert the new value for that we use this code

while curr:

            if curr.next is None or curr.next.min > min:

                path.append(curr)

                curr = curr.down

            elif curr.next.min == min:

                if curr.next.max==max:

                    return

                elif curr.next.max>max:

                    curr=curr

                    path.append(curr)

                    break

                else:

                    while True:

                        if curr.next is None:

                            break

                        elif curr.next.max<max and curr.next.min==min:

                            curr=curr.next

                        else:

                            break

            elif curr.next.min < min:

                curr = curr.next

so we need to insert in multiple levels so the path list stores all the nodes after which the new element is going to be inserted

so while inserting we pop a item prom path list. Lets assume that item name to be curr. So we do curr.next=newnode and curr.down will be pointing to down which is None initially as now we are inserting in the first level and then we make down=newNode. In the second iteration only occours if path has the another element. Then we pop it we perform curr.next=newnode and newnode.down=down here down will be pointing to the node which is inserted in its below level due to the assignment statement down=newNode during the end of the first iteration

so after path is empty all the elements get inserted.

**Practical applications**

This hybrid data structure makes the implementation of interval skip list more efficient so this data structure can be implemented where ever an interval skip list is implemented:-

Hybrid interval skip lists means the data structure we implemented interval skip list + BST

**Database systems**: Hybrid Interval skip lists can be utilized in databases to efficiently handle range queries. They allow for quick retrieval of records that overlap with a given time interval or spatial range. This is particularly useful in applications such as event scheduling, temporal databases, and geospatial databases.**Calendar and scheduling systems:** Hybrid Interval skip lists can facilitate the efficient management of events and appointments. They enable fast lookup of overlapping events or available time slots, allowing for optimized scheduling algorithms and conflict resolution.**Computational geometry:** Hybrid Interval skip lists are valuable in computational geometry algorithms that involve interval intersection or overlap tests. They can be used in tasks like range searching, line segment intersection detection, and interval-based geometric computations.**Traffic management systems:** Hybrid Interval skip lists can be applied in traffic management systems to handle queries related to traffic intervals, such as finding intersections between time intervals of different traffic flows or detecting congested intervals in a road network.**Financial systems:** Interval skip lists can be employed in financial systems to efficiently handle time-based queries for stock prices, trade intervals, or portfolio valuation. They enable quick retrieval of relevant financial data based on specific time ranges.**Event-driven simulations:** Hybrid Interval skip lists can be used in event-driven simulations, such as simulations of physical systems or network protocols. They help manage and search for events that occur within specific time intervals, enabling efficient simulation algorithms.**Multimedia applications:** Hybrid Interval skip lists can be utilized in multimedia applications, such as video indexing or audio processing. They can facilitate operations like searching for video segments within a specific time range or processing audio signals within particular time intervals.

**An example real life scenario:-**

Suppose you have a hotel with multiple rooms, and customers can book rooms for specific time intervals. The hotel reservation system needs to handle queries like finding available rooms for a given check-in and check-out date.Here's how an interval skip list can be applied in this scenario:**Data representation:** Each node in the interval skip list represents a room reservation, storing the interval of time for which the room is booked (e.g., check-in date and time to check-out date and time). The intervals are ordered based on the start time.**Searching for available rooms:** When a customer wants to book a room for a specific time interval, the interval skip list can efficiently identify available rooms that do not overlap with any existing reservations within the desired time range. By traversing the skip list and comparing intervals, the system can quickly identify available rooms or determine if there is a conflict.**Insertion and deletion of reservations:** When a new reservation is made, the interval skip list is updated accordingly by inserting the new reservation interval into the appropriate position in the list. If a reservation is canceled or modified, the corresponding interval can be deleted or updated in the skip list. These operations can be performed efficiently while maintaining the balanced structure of the skip list.**Efficient time range queries:** The interval skip list allows for efficient retrieval of reservations within a specific time range. For example, if the hotel wants to generate a report of all reservations between two dates, the skip list can be traversed to find all intervals that overlap with the given time range.By utilizing interval skip lists in the hotel reservation system, the process of searching for available rooms and managing reservations becomes more efficient and scalable. It enables faster response times for queries, reduces the likelihood of booking conflicts, and provides a streamlined experience for both customers and hotel administrators.

**Performance analaysis**

**Time complexity:-**

The data structure we used is a hybrid of BST and a interval skip list. We made a hybrid data structure to optimize time complexity of operations compared to a interval skip list. The main difference is seen when intervals range between larger numbers and there are larger number intervals (may be one lakh or a million or crore)

The time complexity for insertion is O(logn.logm)=>O(log(n+m)).   
Here n is the number of nodes or elements in a BST and m is the number of intervals in a skiplist.  
so there may be a question the worst case time complexity of a normal interval skip list is O(logm) where m is the number of intervals in a interval skip list.  
The reason for that is pretty straight forward. So lets take a example to properly understand it.  
if we have intervals (1-100),(2-101),(3-102)…….(9901-10000),(10001-11000),(11000-12000)….(99000-100000),(100000-110000),(110000-120000)…..(990000-1000000) [these intervals may be repeated in the subsequent levels which all depends on a coin toss so we cant predict the total number of nodes in a interval skip list]

Here there are a lot of intervals ranging 10000-infinte as the insertion of a intervals on the upper levels is dependent on the tossing of a coin . so for now after insertion lets assume we get some 20000 intervals means each node has only occurred twice in the skiplist.  
first scenario:- Just using a interval skip list  
we store all the elements in a interval skip list. We inserted all the elements. Now lets assume we need to include a interval in this skip list.  
so the worst case time is O(logn) where n is the number of intervals. So to insert it takes O(log(20000)) which is a pretty large value

Now lets consider the same scenario for the hybrid.  
lets take a case where we have a BST of 3 levels and 7 nodes.  
each node stores a skip list.  
So the root node consists of a skiplist where the intervals in the skiplist are only allowed to be ranging between 50000-100000.  
root.left consists of a skiplist where the intervals in the skiplist are only allowed to be ranging between 1000-10000  
root.left.left consists of a skiplist where the intervals in the skiplist are only allowed to be ranging between 1-1000  
root.left.right consists of a skiplist where the intervals in the skiplist are only allowed to be ranging between 10000-50000  
root.right consists of a skiplist where the intervals in the skiplist are only allowed to be ranging between 500000-1000000  
root.right.left consists of a skiplist where the intervals in the skiplist are only allowed to be ranging between 1-500000.  
For there may be a interval like 75000-120000 which any of these nodes will not store these kind of intervals will be stored in the skip list of the last node which is root.right.right.

Root has a skip list with almost 1000 nodes if we assume that each node is only repeated twice like the last scenario.  
Root.left has a skip list with nearly 8000 nodes   
Root.right has a skip list with around 1000 nodes  
Root.left.right has skiplist with 2000 nodes  
root.right.left has a skiplist with around 3000 nodes  
root.left.left has almost 4000 nodes   
root.right,right has almost 1000 nodes

So those 20000 elements in a single interval skip list is now being divided into 7 different skip list.  
so now we want to insert a element [24000-900000] so this will be inserted in the skip list of the last node of BST   
so traverse to the last node it takes log(3) as the number of levels in the tree are just 3  
and for insertion it takes log(1000) as that skip list just has 1000 nodes  
so in total the insertion just takes O(log(1000+3))=O(log(1003))  
so we are reducing the time by 20 times.

We can even reduce the time for all the the operations if we narrow down the intervals and increase the number of levels in the BST so as BST with 15 nodes(means 15 skiplist , that means 20000 intervals will be split into 15 parts rather than 8 which reduces the number of elements in each skiplist) takes much lesser time than BST with 7 nodes .

But the inly issue with increase in the number of nodes in BST is space if there 31 nodes in a BST(each has 5 levels). So each nodes stores an interval skip list so the space complexity is very high compared to the normal interval skip list is used without any hybrid. So if we want to save a lot of time increase number of nodes in a BST , if we want space have lesser BST levels

For deletion time complexity is O(log(n+m)) where n is the number of levels in the BST and m is the number of intervals in the skip list

Same time complexity for search also O(log(n+m))

How it is better compared to a single skip list without BST is already explained with an example above near insertion the same logic goes for deletion and searching also

So basically insertion and deletion time complexity are equal to search as for insertion first we need to search where to insert and when search is completed then just we need to change the pointers of the nodes , same for deletion so all insertion , deletion ,searching takes the same time complexity.

**SPACE COMPLEXITY:-**

So a normal randomized interval skip list 0(n) where n is the number of nodes in an interval skip list. In the hybrid randomized interval skip which we implemented takes O(nm + n) space in its worst case. where n is the number of nodes in a BST and m is the number of nodes in a interval skip list.

The space complexity is almost the n+m where n is number of nodes in BST and m is number of intervals which we will insert in all the skip lists. As when we give an input itself node is created. So if we don’t have data to input no extra space will be untilized

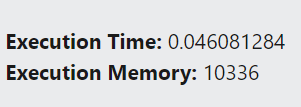
**Experimental Evaluation:**

So we will compare insertion function on less number of intervals and more number of intervals fow our a normal interval skip list and a hybrid interval skip list(combining with BST).

1. Space and time for insertion of intervals followed by a search of numberfor a small data set some 3-4 inputs in a hybrid interval skip list



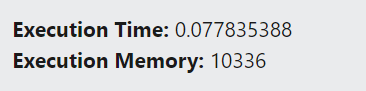
1. Space and time for insertion of some elements and searching a element for a small data set some 3-4 inputs in a normal interval skip list



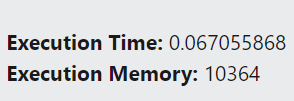
For smaller data sets the time and space taken for insertion and searching a element is almost the same for both the things. But the extra memory occupied is for the binary search tree.

For a little larger data set if 10 intervals

This is the normal skip list



This is a hybrid skip list



So just by adding extra 5 elements we can see that the time taken to search a element by the hybrid skip list is fast than a normal skip list. The extra space is to accommodate a BST . So what we can just assume if we add 10 crore + intervals we make the execution time way more faster than the normal interval skip list.

So to perform insertion , deletion first we need to search . so search is the basic operation we are doing for performing operations in a skip list so we made the search way more efficient and better so this even optimizes other operations like insertion and deletion in our hybrid data structure as compared to a normal interval skip

Even the space complexity is almost the same there is not much extra space needed.

so in a implementation I have done is I have splitted the code in 15 nodes of a BST giving the same 10 inputs

lets see the time and space now



We could optimize the execution time by bit when we splitted the interval skip list into 15 parts than 7. More number of divisons is directly propotional to the efficiency of code.

So searching and insertion almost takes same time as to perform insertion we need to search the node after which we need to insert and direct single step insertion can be done.

Same case with deletion we first perform search and then a single step deletion so deletion and search almost takes the same amount of time

**Discussion**

In scheduling systems, there is often a need to efficiently manage and query intervals of time, such as booking slots, event schedules, or resource allocation. The Interval Skip List can be highly practical and effective in such scenarios due to the following reasons:**1.** **Efficient range queries**: The Interval Skip List allows for fast range queries on intervals. For example, if we want to find all available time slots within a specific range, we can use the Interval Skip List to efficiently retrieve overlapping intervals in O(log n + k) time, where n is the number of intervals in the list, and k is the number of intervals in the output. This enables quick identification of available time slots for scheduling purposes.**2. Dynamic updates:** Scheduling systems often require dynamic updates, such as adding or removing events, adjusting time slots, or modifying existing bookings. The Interval Skip List supports efficient insertion and deletion operations, taking O(log n) time on average. This ensures that the scheduling system can quickly respond to changes and maintain an up-to-date representation of the intervals.**3. Flexibility in interval attributes:** The Interval Skip List is flexible in handling various interval attributes. In scheduling systems, intervals can have different attributes like start time, end time, duration, priority, or other custom properties. The Interval Skip List can accommodate intervals with different attributes, allowing for versatile scheduling scenarios.**4. Performance with large datasets:** Scheduling systems often deal with a significant number of intervals, especially in scenarios where multiple resources or locations need to be managed. The Interval Skip List's average time complexity of O(log n) for common operations ensures efficient performance even with large datasets, making it a practical choice for scaling scheduling systems.**5. Query optimization:** The Interval Skip List can be further optimized by storing additional information in each node, such as the maximum endpoint of intervals within its subtree. This additional information enables quicker pruning and improves the efficiency of range queries, reducing the search space and enhancing performance for complex scheduling queries.**6. Integration with other features:** The Interval Skip List can be integrated with other scheduling system features, such as conflict detection or resource allocation. By efficiently identifying overlapping intervals, the scheduling system can detect conflicts between events or efficiently allocate shared resources without manual checks or exhaustive searches.

By combining skip with BST we can make it more efficient by effectively reducing the time 1/10th in average scenario. So there are still some limitation that we are occupying extra space for a BST kind of implementation this can be considered as one of the limitation.

Another major limitation of skip list and even our hybrid skip list is if the random function always returns a value less than 0.5 then we would have infinte number of levels though it is almost impossible for that event to occur but still there is a minor chance of having that. So skip list or our skip list is randomized due to which the output or elements stored in levels differ at each and very execution so it is tough to predict its nature at times.

So now we took a BST and made it efficient in future we would try to reduce the space complexity of our newly made hybrid data structure. If that is possible then our new hybrid data structure (BST+skiplist) will replace the normal interval skip list

**Conclusion**

There are some of findings which we found.

Skip list or an interval skip takes just O(logn) time to search which is much faster than a linked list which takes O(n). But the space interval skip list takes more space compared to normal skip list

We can further more optimize the interval skip list by dividing the large skip lists into parts and storing each part in a node of BST. So by this implementation we are reducing a lot of time

Interval skip list is used in many of the practical applications which are already mentioned and is very useful in the real world as many important applications use interval skip list. The more divisions you make to the skip list the more time we reduce so more optimize we make it

Due to its speed in search it is a very good data structure , it has a very good Range query performance, scalabilty(The hybrid which we implemented is even more efficient than the normal skip list if we scale it ) and query optimization which makes it efficient.

In conclusion, the Interval Skip List is a practical and effective data structure for efficiently managing and querying interval-based data. It offers efficient time complexity for common operations, including insertion, deletion, and search, with an average time complexity of O(log n). The ability to perform range queries in O(log n + k) time makes it well-suited for scenarios where interval overlap detection is crucial.