Concurrent Skip Lists

B.Tech Project Report

Mentor: R. Govindarajulu

T S V Maneesh

B.Tech CSE

201201183

International Institute of Information Technology, Hyderabad

Hyderabad, India

maneeshts.v@students.iiit.ac.in

Basina Chaitanya

B.Tech CSE

201101163

International Institute of Information Technology, Hyderabad

Hyderabad, India

basinachaitanyavisweswara.rao@students.iiit.ac.in

Srujai Varikuti

B.Tech CSE

201101023

International Institute of Information Technology, Hyderabad

Hyderabad, India

srujai.varikuti@students.iiit.ac.in

Description

An implementation of a skip list data structure in java that is thread-safe for searching and insertion and deletion.

1. Introduction

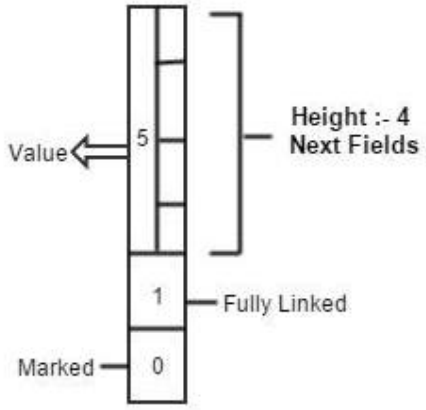
A concurrent data structure is a particular way of storing and organizing data for access by multiple computing threads. The term Concurrent has come to stand mainly for data structures that can be accessed by multiple threads which may actually access the data simultaneously because they run on different processors that communicate with one another.

A skip list [8] is a randomized variant of an ordered linked list with many parallel lists through the data held in the list. When searching for a position in a skip list-to either insert a new item, delete an item, or find a previously inserted item-parallel lists at higher levels skip over large numbers of items. Searching begins at the highest level until a key greater than the one being searched for is encountered. The search shifts to progressively lower levels of parallel lists until the desired location is found. A new item is added by randomly selecting a level, then inserting it in order on the lists for that and all lower levels.

2. Lazy Skip List

In general, there would be far more 'contains ()' operations than 'add ()' or 'delete ()'. So, the idea is to make it fast (wait-free). An algorithm is wait-free if every operation has a bound on the number of steps the algorithm will take before the operation completes.

```
class node
{
      Lock lock;
      int data;
      node[] next;
      boolean mark;
      boolean fulllink;
      int toplevel;
}
```



Node Structure

The core principle behind this implementation is to delete nodes logically (or mark them) before deleting them physically. So, we call it **Lazy Skip List** [2].As we can see in the declaration of the node class above, each node has its own lock and flag for logical removal. Locks are also used to delay access to a node until it is inserted into all levels.

The node class contains fields 'marked' which is set when the node is logically deleted and 'fulllink' which indicates that the node is linked at all levels. If the 'fulllink' field is not set, it means that the node is still in the process of insertion.

A. Find() operation

This find() method is used by the insert and remove methods and is used to find a node.

The find() method is wait-free and does not acquire any locks. It traverses the list and returns:

- The level at which the node was first found (-1 if not found)
- The predecessor nodes of the searched node at all levels
- It returns the node if it is present or returns its successor nodes at all levels if it is not present

```
int find(int data, node[] pred, node[] succ)
{
	int found = -1;
	node prednode = head;
	for(int i = maxheight; i >= 0; i--)
	{
		node currnode = prednode.next[i];
		while(currnode.data < data)
		{
			prednode = currnode;
			currnode = currnode.next[i];
		}

		//Reached the node for the first time
		if(data == currnode.data && found == -1)
		{
			found = i;
		}
		//Recording predecessors and successors
		pred[i] = prednode;
		succ[i] = currnode;
	}
	//Returning the highest level at which the node was found
	return found;
}
```

It traverses the list until a node with value equal to it or greater than the value being searched for is found at all levels. It keeps track of the predecessor node and returns the predecessor nodes at all levels. If the node is found, the successor array contains the node, if it is not found it contains the first node that is greater than the value being searched for at all levels. We can see that the 'currnode' node's data never exceeds the value being searched. So, the 'succ' array points to the node with the value being searched, if it is present. These predecessor and successor nodes are used during insertion and deletion of the nodes.

B. Contains() operation

The contains() method checks if the node is present in the list. It is wait-free.

The contains method uses the find method and

- Returns true, if the node is present in the skip list i.e., the level returned by the find method is valid and the node is not marked for removal and is connected at all levels
- Returns false, if any of the conditions mentioned are not satisfied

```
boolean contains(int data)
{
	node[] preds = new node[maxheight+1];
	node[] succs = new node[maxheight+1];
	int level = find(data,preds,succs);

	//Checking if the node is present and inserted
	//and not marked for deletion
	if(level >= 0 && succs[level].fulllink && !succs[level].mark)
	{
		return true;
	}
	return false;
}
```

It calls the find method and returns true if the successor array contains the node being searched for and if it is not marked for deletion and fully linked. Find() method is wait-free and hence this operation is also wait-free.

C. Insert() operation

The insert method uses the find method and

- Returns true, if the node is inserted by this thread
- Returns false, otherwise

```
boolean add(int data)
{
	//highestlevel: highest level until which the node is inserted
	int highestlevel = randomheight();
	node[] preds = new node[maxheight+1];
	node[] succs = new node[maxheight+1];
	while(true)
	{
		int level = find(data,preds,succs);

		//level >= 0 -> node is found in skip list already
		if(level >= 0)
		{
			node datanode = succs[level];
			if(!datanode.mark)
			{
				//waiting for insertion by other thread
				while(!datanode.fulllink)
				{}
				return false;
			}
			//Node is marked for deletion,
			//so restart the process and insert the node after
			//its deletion
			continue;
		}
```

```
int highestlocked = -1;
                  try
                  {
                        node pred,succ;
                        //valid field tells if the insertion is still
                        //feasible
                        //i.e., if the predecessor and successor are still
                        //in the skip list
                        //and no node is being inserted between them
                        boolean valid = true;
                        for(int i = 0; i <= highestlevel && valid; i++)
                        {
                              pred = preds[i];
                              succ = succs[i];
                              pred.lock();
                              highestlocked = i;
                              valid = !pred.mark && !succ.mark &&
pred.next[i] == succ;
                        }

                        // if valid is false, then either the predecessor
                        //or the successor is
                        //either deleted or being deleted
                        if(!valid)
                        {
                              continue;
                        }

                        //Insertion of new node begins
                        node newnode = new node(data,highestlevel);
                        for(int i = 0; i <= highestlevel; i++)
                        {
                              preds[i].next[i] = newnode;
                              newnode.next[i] = succs[i];
                        }
                        //Node is fully linked at all levels
                        newnode.fulllink = true;
                        return true;
                  }
                  finally
                  {
                        //Unlocking predecessors
                        for(int i = 0; i <= highestlocked; i++)
                        {
                              preds[i].unlock();
                        }
                  }
            }
      }
```

The insert method calls find which returns the predecessor and successor nodes of the value at all levels. If the node is not fully linked, the method waits until the node is fully linked and returns false. If the node is found but marked, waits until the node is deleted and inserts the node again.

If the node is not marked and fully linked, the predecessor nodes previously obtained are validated and then appropriate locks are acquired on them before insertion of the node. If any predecessor node is found to be marked, then the find() method is called again and nodes are obtained. Now the next field of the predecessor node is linked to the node at all levels and the 'fulllink' field is set. This statement is the *linearization point* of a successful insert() call. A *linearization point* is a point or a statement where the method takes effect i.e., appears to have executed.

After the operation is completed, the predecessor nodes are unlocked.

D. Remove() operation

The remove method calls find which returns the predecessor nodes at all levels along with the highest level at which the current node was found if it were present. Remove() operation

- Returns true, if the node was deleted by this thread
- Returns false, otherwise

```
boolean remove(int data)
{
      node deletenode = null;
      int highestlevel = -1;
      boolean marked = false;
      node[] preds = new node[maxheight+1];
      node[] succs = new node[maxheight+1];
      while(true)
      {
            int level = find(data,preds,succs);
            if(level >= 0)
            {
                  //deletenode is the highest node
                  //with the corresponding data value in the list
                  deletenode = succs[level];
            }

            //Checking if deletenode is inserted completely
            //and not already being deleted by another thread
            if(marked | (level >= 0 && !deletenode.mark &&
deletenode.fulllink && deletenode.toplevel == level))
            {
                  //Not yet marked
                  if(!marked)
                  {
                        //Logically marking the node
                        highestlevel = level;
                        deletenode.lock();
                        if(deletenode.mark)
                        {
                              deletenode.unlock();
                              return false;
                        }
                        deletenode.mark = true;
                        marked = true;
                  }
```

```
int highestlocked = -1;
                        try
                        {
                              node pred;
                              boolean valid = true;

                              //Checking if the predecessors obtained are
                              //still valid
                              for(int i=0; i<=highestlevel && valid; i++)
                              {
                                    pred = preds[i];
                                    pred.lock();
                                    highestlocked = i;
                                    valid = !pred.mark && pred.next[i] ==
deletenode;
                              }

                              //if valid is false, then the predecessor
                              //might have been marked
                              //or modified, restarting the process in
                              //this case
                              if(!valid)
                              {
                                    continue;
                              }

                              //Deleting the node
                              for(int i = highestlevel; i >= 0; i--)
                              {
                                    preds[i].next[i] = deletenode.next[i];
                                    deletenode.toplevel--;
                              }
                              deletenode.unlock();
                              return true;
                        }
                        finally
                        {
                              //Unlocking predecessors
                              for(int i = 0; i <= highestlocked; i++)
                              {
                                    preds[i].unlock();
                              }
                        }
                  }
                  else
                  {
                        return false;
                  }
            }
      }
```

Then the method checks if the node to be deleted is marked already to ensure that the node is marked by some other thread and if it has been partially removed by checking it if it is still present in its top level.

If the above conditions are satisfied, the thread locks the node to be deleted and marks it. This step forms the *linearization point* of a successful remove operation. Now, moving on to the physical deletion.

Locks are obtained over the predecessors (in descending order to prevent deadlocks) and they are validated. If any predecessor is found to be marked, new predecessors are found by calling the find method. If the predecessor still points to the victim node, its next pointer is updated. After the remove operation is completed successfully, all locks are released on the predecessors.

3. Lock Free Skip List

Now, we extend the idea of marking nodes to eliminate locks and make the add() and remove() operations *lock-free* [3]. Lock-free algorithms perform better when some threads or operations hold higher priority over others. For example, if a thread with lower priority holds the lock in the previous case, the higher priority thread has to wait for it. Hence, lock free data structures are used in real-time applications. A *lock-free* algorithm guarantees that the threads do not block each other. They use atomic operations. In this implementation, all list operations are lock-free and use the atomic '*compareandset()*' operation.

```
class node
{
      int data;
      int height;
      AtomicMarkableReference<node>[] next;
}
```

As we can see from the structure above, the next pointer of a node here is an '*AtomicMarkableReference*' class which has a Boolean mark variable along with the next pointer reference. This class contains the compareandset() method along with other methods namely

- getReference(): Returns the enclosed reference
- get(array): Returns the reference along with the mark value in the 0^{th} index of the parameter passed
- compareAndSet(): Atomically sets the value to the given updated value, if the current value is equal (==) to the expected value.
- attemptMark(): Atomically sets the value of the mark to the given update value, if the current reference is equal (==) to the expected reference.

There is a small design change in this implementation wherein the nodes marked by the remove operation are deleted physically by the next find operation which traverses them. The remove() method calls the find() operation to physically delete the node once it has marked the node to be deleted, but it does not restart the process when the physical deletion fails. Failure might occur when the predecessor and successor nodes are modified and the physical deletion will be taken care of by the next find() operation which traverses the nodes. So, this change decreases the thread delay due to a failure in physical deletion. Both insert() and remove() operations call find(), so including this functionality in find() would reflect in physical deletion by both the methods. This behavior is skipped for the contains() method as it has to be implemented to run as fast as possible.

A. Insert() operation

The insert method calls find to get the predecessor and successor nodes and

- Returns true, if the node is inserted by this thread
- Returns false, if the node is already found to be inserted

```
	boolean add(int value)
	{
		//highestlevel: highest level until which the node is inserted
		int highestlevel = randomheight();
		node[] preds = new node[maxheight+1];
		node[] succs = new node[maxheight+1];
		while(true)
		{
			//Getting the predecessor and successor nodes
			boolean present = find(value,preds,succs);

			//If the node is already present, we return false
			if(present)
			{
				return false;
			}
			else
			{
				//Initializing the node
				//and making it point to the successor nodes at
				//all levels
				node curr = new node(value,highestlevel);
				for(int level = 0; level <= highestlevel; level++)
				{
					curr.next[level].set(succs[level], false);
				}

//Using compareandset to add the node to the bottom level
				boolean added =
preds[0].next[0].compareAndSet(succs[0], curr, false, false);

				//If the operation does not succeed
				//The predecessors or successors might be
				//modified, retry
				if(!added)
				{
					continue;
				}

				//Adding node at all levels
				for(int level = 1; level <= highestlevel; level++)
				{
					//retry until added
					while(true)
					{
						boolean temp =
preds[level].next[level].compareAndSet(succs[level], curr, false, false);
						if(temp)
							break;
						//Using find because predecessors and
						//successors are modified
						find(value,preds,succs);
					}
				}
				return true;
			}
		}
	}
```

After checking for the presence of the node in the list, the add method initializes the node and updates its next pointers to point to the successors at all level. It uses the '*compareandset*()' atomic primitive to modify the *'next'* pointers of the predecessor nodes. If the next reference of the predecessor node still points to the successor node then it is updated to point to the node to be inserted at all levels. Adding the node into the bottom level is the *linearization point* of a successful insert operation. Then, the node is added to all remaining levels. If the predecessor node is found to be marked or has changed, find operation is called again to find the new neighbors.

B. Remove() operation

The remove method also calls the find method to get the predecessor nodes and

- Returns true, if the node to be deleted is removed by this thread
- Returns false, otherwise

```
boolean remove(int value)
{
	node[] preds = new node[maxheight+1];
	node[] succs = new node[maxheight+1];
	while(true)
	{
		//Getting predecessor and successor nodes
		boolean present = find(value,preds,succs);

		//Checking for presence of node in skip list
		if(!present)
		{
			return false;
		}

		else
		{
			node curr = succs[0];

			//Marking all levels except bottom level
			for(int level = curr.getheight(); level >= 1; level--)
			{
				//temp: contains the logical 'mark' value
				boolean[] temp = {false};
				node succ = curr.next[level].get(temp);

				//marking the nodes while they are not marked
				while(!temp[0])
				{
					curr.next[level].attemptMark(succ, true);
					succ = curr.next[level].get(temp);
				}
			}
```

```
                    boolean[] temp = {false};
                    node succ = curr.next[0].get(temp);
                    while(true)
                    {
                          //Marking the bottom level
                          boolean marked = curr.next[0].compareAndSet(succ,
succ, false, true);

                          //calling find method to clear out marked nodes
                          //can be skipped
                          if(marked)
                          {
                                find(value,preds,succs);
                                return true;
                          }
                          else
                          {
                                succ = curr.next[0].get(temp);

                                //Node is already marked, returning false
                                if(temp[0])
                                      return false;
                          }
                    }
              }
        }
}
```

After calling the find method, it checks if the node is present and it is not yet marked for deletion before proceeding. It then starts marking the node from the top level descending downwards. It uses the '*attemptMark()*' to mark the nodes. If any predecessor nodes at a level are changed, find() method is called to get the new predecessors. Once all the levels except the bottom level are marked, the bottom node is marked. Marking the node at the bottom level marks the *linearization point* of a successful remove operation. This is achieved by using the '*compareandset()*' method. The reason behind using the '*attemptMark()'* method earlier and *'compareandset()'* method for marking the last node is that the *'attemptMark()'* method does not take the expected mark value into account and *'compareandset()*' does. If we use '*attemptMark()'* to mark the node at the bottom level and it is already true, then it might already be in the process of physical deletion and this method again calls a find method to clean up which becomes a waste. So, overhead can be decreased. After deletion, a find method can also be called to clean up and complete physical deletion of the node.

If it is not successful in deleting a node, then it might already be deleted by some other thread or its neighbor might be deleted. If its neighbor is deleted, it automatically gets the successor after its successor's physical deletion.

C. Find() operation

The find method just as before returns

- The presence of the node
- The predecessor nodes of the searched node at all levels
- It returns the node if it is present or returns its successor nodes at all levels if it is not present

```
boolean find(int value,node[] pred,node[] succ)
{
	node before = null,curr = null,after = null;
	while(true)
	{
		before = head;
		boolean done = true;

		//Traversing the list
		for(int level = maxheight; level >= 0 && done; level--)
		{
			curr = before.next[level].getReference();
			while(true)
			{
				boolean[] temp = {false};
				after = curr.next[level].get(temp);

				//If temp[0] is true, node is marked
				while(temp[0])
				{
					//Physically deleting the node
					done =
before.next[level].compareAndSet(curr, after, false, false);

					//If deleting was unsuccessful,
					//restarting the process
					if(!done)
						break;
					curr = before.next[level].getReference();
					after = curr.next[level].get(temp);
				}
				if(curr.getdata() < value && done)
				{
					before = curr;
					curr = after;
				}
				else
					break;
			}

			//Recording predecessor
			//and successor nodes
			if(done)
			{
				pred[level] = before;
				succ[level] = curr;
			}
			else
				break;
		}
		if(done)
			return curr.getdata()==value;
	}
}
```

The find method physically deletes a logically deleted node. It traverses the list deleting any logically marked node along the way using the *'compareandset()'* method. It records the predecessor and successor nodes at all levels which are used by add() and delete() operations.

D. Contains() operation

The contains method traverses the list and returns

- True, if the node is present in the list
- False, if the node is not present in the list

```
boolean contains(int value)
{
	node before = null,curr = null,after = null;
	while(true)
	{
		before = head;

		//Traversing the list
		for(int level = maxheight; level >= 0; level--)
		{
			curr = before.next[level].getReference();
			while(true)
			{
				boolean[] temp = {false};
				after = curr.next[level].get(temp);

				//If temp[0] is true, node is marked
				//Skipping over the marked nodes
				while(temp[0])
				{
					before = curr;
					curr = before.next[level].getReference();
					after = curr.next[level].get(temp);
				}

				if(curr.getdata() < value)
				{
					before = curr;
					curr = after;
				}
				else
					break;
			}
		}
		return curr.getdata()==value;
	}
}
```

The contains method works very similarly to the find method in this implementation. The only difference being that the contains method does not physically delete marked nodes but ignores them while traversing.

Performance Comparison

We compared our *'lazy skip list'* and *'lock free skip list'* with other existing implementations of concurrent skip lists. The non-blocking skip list implementation [5] of Doug-Lea which was built into the java concurrency library as *'ConcurrentSkipListSet'* and the skip list variant '*coarse_list' was* built based on the coarse grained linked list data structure from The Art of Multi-Processor Programming book. A *'coarse_list'* is one in which the whole list is locked before the operation is done and it is released after completion of the operation.

The following tests were run on a quad core i7-4700MQ processor with a frequency of 2.40GHz and 4 threads were used.

We tested our skip lists on data of five different categories.

- Category 1 : 20% add(), 10% remove(), 70% contains()
- Category 2 : 8% add(), 2% remove(), 90% contains()
- Category 3 : 33.33% add(), 33.33% remove(), 33.33% contains()
- Category 4 : 50% add(), 50% remove(), 0% contains()
- Category 5 : 20% add(), 0% remove(), 80% contains()

We had two types of test data, one of size 200,000 operations and another of size 2,000,000 operations, which were generated randomly every time for each category on each implementation.

	Coarse List	Skip List Set	Lazy Skip List	Lock Free Skip List
Category 1	0.853	0.945	0.982	1.078
Category 2	1.257	1.443	1.477	1.522
Category 3	0.725	0.615	0.659	0.757
Category 4	0.966	0.876	0.951	1.011
Category 5	1.278	1.268	1.279	1.352

Performance Comparison Table 1

	Coarse List	Skip List Set	Lazy Skip List	Lock Free Skip List
Category 1	42.6	40.9	39.8	40.6
Category 2	76.0	77.6	76.3	79.0
Category 3	32.7	32.0	32.3	32.7
Category 4	51.2	49.0	49.7	50.2
Category 5	62.9	65.3	63.1	67.1

Performance Comparison Table 2

The results shown in the above tables are the running times in seconds of the algorithms on the input data of the corresponding category. *Table 1* corresponds to the running times of the implementations for the smaller data set (of size 200,000) and *Table 2* corresponds to the running times of the implementations for the larger data set (of size 2,000,000). Improvement over the *'coarse_list'* which is the basic skip list implementation is not evidently visible here but that is because the overhead cancels out the performance improvement of the four threads but it is clear from the implementation that the *'coarse_list'* performs almost the same with increase in number of threads while our *'lazy skip list'* and *'lock free list'* algorithms throughput would increase linearly. On average, we were able to see a performance increase of 4-5% from the basic implementation of concurrent skip list on 4 threads for the *'Lazy skip* list' and no significant improvement for *'lock free skip* list'. Also, our algorithms provide almost the same performance as the *'ConcurrentSkipListSet'* of the java library. The *'lock free list'* did not produce expected performance results. Unfortunately the current reality is that most lock free algorithms are complex, slow. This gap between theoretical and practical performance can be attributed to the lack of support for the atomic primitive operations from the hardware or the overhead caused to emulate it by the hardware or the operating system. Although this problem has been reduced by the latest processors, implementing the structure using this primitive produces some overhead. Also memory management is another key issue, the general purpose garbage collectors are not non-blocking and hence they may produce some delay. Even with all these limitations, the *'lock-free skip list'* might be able to give better performance than a *'coarse_list'* when run on higher number of threads.

Performance Improvements

The Lazy Skip List [2] algorithm was improved for better performance

1. Dynamically changing maximum height was implemented for the skip list. Therefore, overhead of traversing to empty levels decreases.

2. The add() function and the contains() function called find() which traversed all levels of the list even if the node was found. But now, the find() returns to add and contains on finding the node at its highest level and the add() function can consequently return false if the node is not marked and fully linked and contains() function can return true. This improvement gets rid of the overhead incurred due to traversing the extra bottom levels after finding the node.

Performance Comparison

The following tests were run on a quad core i7-4700MQ processor with a frequency of 2.40GHz and Intel hyper-threading was used for running 6 threads and 8 threads. We compared our implementations with the *'Coarse_List'* implementation built from *'Coarse_Linked_List'* [1] proposed by Maurice Herlihy and the *'ConcurrentSkipListSet'* from the JDK8.

We had two types of test data, one of size 200,000 operations and another of size 2,000,000 operations, which were generated randomly every time for each category on each implementation.

We tested our skip lists on data of five different categories.

- Category 1: 20% add(), 10% remove(), 70% contains()
- Category 2: 8% add(), 2% remove(), 90% contains()
- Category 3: 33.33% add(), 33.33% remove(), 33.33% contains()
- Category 4: 50% add(), 50% remove(), 0% contains()
- Category 5: 20% add(), 0% remove(), 80% contains()

All reported values are runtimes in seconds.

	Coarse list	SkiplistSet	Lazy list	Lock free	Height	Functions	Both
1.	0.881	0.811	0.815	0.892	0.817	0.815	0.816
2.	1.117	1.163	1.162	1.165	1.174	1.172	1.168
3.	0.603	0.494	0.508	0.532	0.506	0.505	0.504
4.	0.802	0.708	0.712	0.747	0.722	0.714	0.718
5.	1.041	0.984	0.992	0.989	1.008	0.993	0.993

4 Threads on Small Dataset

	Coarse list	SkiplistSet	Lazy list	Lock free	Height	Functions	Both
1.	0.769	0.699	0.711	0.718	0.702	0.705	0.708
2.	1.037	1.001	1.01	1.004	1.005	1.006	1
3.	0.581	0.433	0.434	0.447	0.432	0.429	0.429
4.	0.698	0.615	0.62	0.634	0.622	0.633	0.62
5.	0.88	0.836	0.85	0.85	0.852	0.848	0.851

6 Threads on Small Dataset

	Coarse list	SkiplistSet	Lazy list	Lock free	Height	Functions	Both
1.	0.711	0.649	0.645	0.651	0.639	0.639	0.641
2.	0.935	0.917	0.915	0.919	0.913	0.908	0.912
3.	0.502	0.407	0.401	0.412	0.399	0.395	0.399
4.	0.635	0.55	0.559	0.568	0.555	0.555	0.557
5.	0.789	0.723	0.731	0.735	0.733	0.728	0.728

8 Threads on Small Dataset

	SkiplistSet	Lazy list	Lock free	Height	Functions	Both
1.	51.7	51.8	51.3	50.8	51.2	50.5
2.	77.8	78.5	78.7	78.1	78.5	77.4
3.	31.8	32	32.5	31.6	32.2	31.9
4.	50	48.7	50.2	48.5	48.3	48.1
5.	66.5	65.6	65.2	65	64.8	64.7

4 Threads on Big Dataset

	SkipistSet	Lazy list	Lock free	Height	Functions	Both
1.	40.6	39.1	39.2	37.9	38.5	38.5
2.	0	56.8	57.3	56.6	56.5	56.2
3.	25.4	25.1	25.3	25	24.7	25
4.	36.4	36.3	37.5	36.4	36.3	35.9
5.	48.4	47.9	47.4	47.1	47.5	47.4

8 Threads on Big Dataset

The overhead caused by the dynamic height improvement is covered due to more number of operations in the larger dataset. The improvement made to the functions performs very well on the smaller dataset in most cases while it is not as effective on the larger dataset. The improvement is utilized at its best when there are redundant insertions and searching of values which are present in the dataset. But due to the large range of values in the larger dataset, it might not have found those many redundant insert operations.

Deviation from Ideal Skip List Height Generation

The height of a node to be inserted into the skip list is determined by tossing a coin. So, statistically over a reasonable number of insertions into the list, only half of the nodes from the layer below are found in the layer above it. This property is vital to a skip list and is the reason behind the skip list giving O(log n) performance on insertion, deletion and searching.

We tested our lazy skip list implementation to check for the deviation from this property. We used the random.nextInt() function from the java library to generate random heights for our nodes. On monitoring the number of nodes in the 0^{th} level, 1^{st} level and 2^{nd} level of our skip list over multiple runs, we found that the ratio of number of nodes in the 1^{st} level to the number of nodes in the 0^{th} level was ranging from 0.4 to 0.58 and the maximum number of nodes in the 0^{th} level between two nodes in the 1^{st} layer was generally 6 but we recorded a maximum of 9 nodes for a runtime once. On average over all runs, the ratio of 1^{st} level nodes to 0^{th} level nodes was 0.49. But a maximum of 6 nodes and the range 0.4-0.58 of the ratio meant that there was room for further improvement in the randomheight function.

We found that the ratio of 2nd level nodes to 1st level nodes was slightly better ranging from 0.46-0.53 and an average of 0.505 over multiple runs of the program. And the maximum number of nodes in the 1st level between two nodes of the 2nd level was generally 4-5. It can be inferred from this data that the greater the number of nodes in the level, the greater the deviation from idea skip list behaviour.

We tested for the same property in the *'ConcurrentSkipListSet'* [9] class from the java JDK8 version. This class had the ratio of 1st level nodes to 0th level nodes ranging from 0.53-0.58 and on average had 0.54 on several runs. However, the maximum number of nodes in the 0th level between two nodes in the first level was kept at 1 in all the runs. And the ratio of number of nodes at the 2nd level to the number of nodes at the 1st level ranged from 0.53 to 0.57 and had an average of 0.55 approximately. And, the maximum number of nodes at the 1st level between two nodes which are at the second level was anywhere between 6 and 12. Therefore, it could be deduced that their randomheight function is better than our randomheight function for a small dataset as it is better at lower levels than the higher levels. It might have attributed to the better performance of the *'ConcurrentSkipListSet'* than our lazy list algorithm for the smaller dataset.

Concurrent Dictionary (Map)

- Implemented on top of the concurrent skip list.
- Contains Key-Value Pairs and skip list is sorted in the order of keys.
- Keys are considered to be unique.

Node Structure

```
public class node
{
        private final Lock lock = new ReentrantLock();
        public int key;
        public int value;
        public node[] next;
        public boolean mark = false;
        public boolean fulllink = false;
        int toplevel;
}
```

Dictionary Functions

- **Insert(Key, Value)**
    - Inserts the corresponding key value pair into the dictionary and returns true.
    - Returns false if the key is already present in the dictionary.

```
public boolean insert(int key, int value)
{
	boolean key_add = true;
	key_add = keys.add(key, value);
	if(key_add)
	{
		length++;
		return true;
	}
	return false;
}
```

- **Delete(Key)**
  - Removes the node with the corresponding key from the dictionary.
  - Returns false if the key is not found in the dictionary or already deleted.

```
public boolean delete(int key)
{
	boolean key_rem = true;
	key_rem = keys.remove(key);
	if(key_rem)
	{
		length--;
		return true;
	}
	return false;
}
```

- **Contains_Key(Key)**
  - Returns true if the key is present in the dictionary and false otherwise
  - Directly calls the contains() function of the skiplist
- **Get_Value(Key)**
  - Returns the value associated with the key
  - If key is not present, returns -2^{31}.
  - Finds the keys and returns the value if the node is fully linked and not marked.

```
public int get_value(int key)
{
        node[] preds = new node[maxheight+1];
        node[] succs = new node[maxheight+1];
        int level = keys.find(key, preds, succs);

        if(level >= 0 && succs[level].fulllink && !succs[level].mark)
        {
                return succs[level].value;
        }
        return Integer.MIN_VALUE;
}
```

- **Replace_key(int key, int new_key)**
    - Replaces the current key value with the new_key and returns true
    - Returns false if new_key cannot be inserted or key is not present.
    - Finds the node and locks it and collects the value and height corresponding to that node and then deletes the node before calling the add_with_height() helper function to add the node with the same value at the same height as the previously deleted node.
    - If the node with the new_key cannot be added, the older key is added back and the function returns false.

```
public boolean replace_key(int key,int new_key)
{node[] preds = new node[maxheight+1];
        node[] succs = new node[maxheight+1];
        int level = keys.find(key, preds, succs);

        if(level >= 0 && succs[level].fulllink && !succs[level].mark)
        {
                succs[level].lock();
                int value = succs[level].value;
                int height = succs[level].toplevel;
                succs[level].unlock();
                boolean key_del = delete(key);
                if(key_del)
                {
                        boolean key_add = keys.add_with_height(new_key, value, height);
                        if(key_add)
                                return true;
                        else
                        {
                                keys.add_with_height(key, value, height);
                                return false;
                        }
                }
                else
                        return false;
        }
        return false;
}
```

- **Replace_value(Key,New_Value)**
    - Replaces the current value associated with the key with the new value.
    - Returns false if the key is not present in the dictionary.
    - Finds the node and locks it if it is valid i.e., not marked for deletion and fully linked and then changes the value.

```
public boolean replace_value(int key,int new_value)
{
        node[] preds = new node[maxheight+1];
        node[] succs = new node[maxheight+1];
        int level = keys.find(key, preds, succs);

        if(level >= 0 && succs[level].fulllink && !succs[level].mark)
        {
                succs[level].lock();
                succs[level].value = new_value;
                succs[level].unlock();
                return true;
        }
        return false;
}
```

- **Keys()**
  - Returns all the keys present in the dictionary in sorted order.
  - This method calls the get_keys() helper method added to the skip list data structure.
  - get_keys() method moves through the bottom level of the list and stores all nodes which are fullyinserted and not yet marked into the keys array and returns it.

```
public ArrayList<Integer> get_keys()
{
	node curr = head.next[0];
	ArrayList<Integer> keys = new ArrayList<Integer>();
	while(curr != tail)
	{
		if(curr.fulllink && !curr.mark)
		{
			keys.add(curr.key);
		}
		curr = curr.next[0];
	}
	return keys;
}
```

- **Values()**
  - Returns the values present in the dictionary in sorted order of keys.
  - This method calls the get_values() helper method added to the skip list data structure.
  - The get_values() does the same operations as the get_keys() method. It checks the bottom layer of the skiplist for nodes which are not yet marked and fullylinked and returns the set of values.

```
public ArrayList<Integer> get_values()
{
	node curr = head.next[0];
	ArrayList<Integer> values = new ArrayList<Integer>();
	while(curr != tail)
	{
		if(curr.fulllink && !curr.mark)
		{
			values.add(curr.value);
		}
		curr = curr.next[0];
	}
	return values;
}
```

- **Size()**
  - Returns the size of the dictionary i.e., the number of elements present in the dictionary.

- **Clear()**
  - Clears the dictionary i.e., deletes all elements present in the dictionary.

Performance Comparison of Concurrent Dictionary

The following tests were run on a quad core i7-4700MQ processor with a frequency of 2.40GHz and Intel hyper-threading was used for running 6 threads and 8 threads. We tested our implementation against the *'ConcurrentSkipListMap'*[8] from the JDK8.

The dataset consisted of 200,000 operations which were generated randomly every time the program is run. We tested our dictionary in data sets of two different categories.

Category 1: 20% insert(), 10% remove(), 25% get_value(), 30% replace_value(), 10% contains_key(), 5% keys() and values().

Category 2: 25% insert(), 10% remove(), 25% get_value(), 30% replace_value(), 10% contains_key().

All reported values are runtimes of the programs in seconds.

	Concurrent Dictionary	ConcurrenSkipListMap
1.	1.682	0.397
2.	0.407	0.399

4 Threads

	Concurrent Dictionary	ConcurrenSkipListMap
1.	1.301	0.350
2.	0.347	0.349

6 Threads

	Concurrent Dictionary	ConcurrenSkipListMap
1.	1.128	0.335
2.	0.327	0.326

8 Threads

The keys() and values() functions of our concurrent dictionary were bottlenecking the throughput and it is clear through the difference in runtimes of category 1 and 2 of the concurrent dictionary. Not considering the keys() and values() functions, our concurrent dictionary fares very well and gives almost the same performance as the *'ConcurrentSkipListMap'* in the JDK8 version. It might be because the keys() and values() functions are O(n) implementations which go through the bottom layer of the skiplist and store keys and values every time they are called.

A better and more efficient implementation would have been to maintain a keys array and a

values array and add and remove values from them during the insert() and remove() operations. Then insert() and remove() would incur overhead of the inserting and removing values from these arrays but the keys() and values() functions would give O(1) performance. Another issue would be maintaining the integrity of the data while it is accessed concurrently. It can be done by maintaining locks for access to the array. Doing this will reduce the situation to a readers-writers problem. This implementation, in our opinion, would greatly reduce the runtime of our concurrent dictionary and possibly reduce the runtime to around the running time of the *'ConcurrentSkipListMap'*.

References

[1] Maurice Herlihy and Nir Shavit, The Art of Multi-Processor Programming

[2] Maurice Herlihy, A provably correct scalable concurrent skip list

[3] Kier Fraser, Practical Lock Freedom, 2004

[4] William Pugh, Concurrent Maintenance of Skip Lists, 1990

[5] Doug Lea, No Hot Spot Non-Blocking Skip List

[6] The Art of Multi-Processor Programming book slides

[7] William Pugh, Skip Lists: A Probabilistic Alternative to Balanced Trees

[8] https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/ConcurrentSkipListMap.html

[9] https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/ConcurrentSkipListSet.html