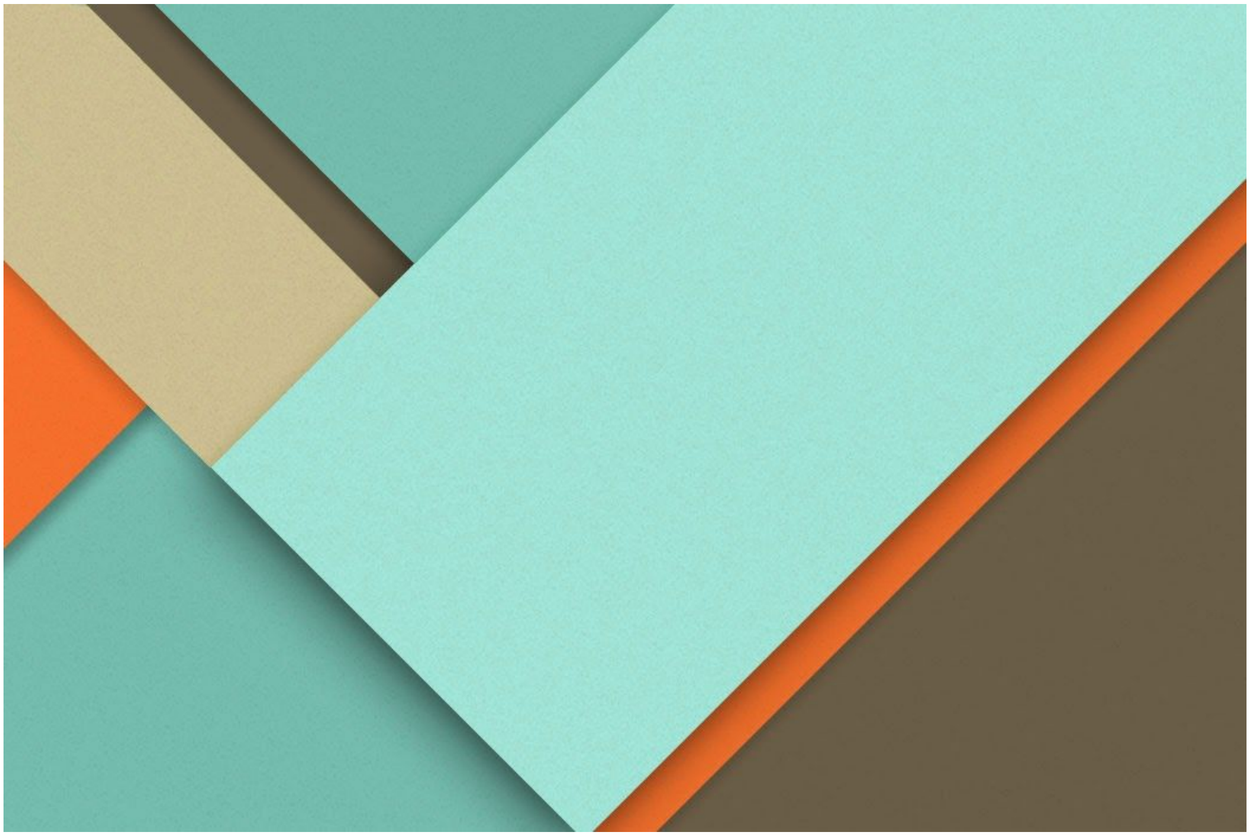


Hopscotch Hashing

18.11.2020

Sridhar M
2018111021
Monsoon 2020
IIIT Hyderabad

Overview

The project is on understanding and implementing Hopscotch hashing in the JAVA language.

Hopscotch hashing is a method of open addressing that builds on the prior work in chained hashing, linear probing, and Cuckoo hashing in order to design a new methodology. The key idea of hopscotch hashing is as follows:

- Assign a global neighborhood size H
- keep the hashed key within this neighborhood

Deliverables

1. Java implementation of Hopscotch hashing
2. Project report
3. Presentation of the project

Algorithmic overview

1. There is a single hash function h
2. Hashed item will always be found either in that home bucket or in one of the next H−1 neighboring bucket (hop)
3. Each bucket includes some hop-information using an H-bit bitmap (in code, represented by- ***hope***), it indicates which of the items in the bucket's neighborhood hashed to the current bucket

Cache Locality

Hopscotch hashing has super-fast query times in practice is because of its excellent cache locality. Complexity for find operation is O(1). Setting neighborhood size to be equal to the size of the cache line means there will be at most 2 cache misses before an object is found.

Comparison: Round Robin Hashing

- Hopscotch hashing is slightly more cache-friendly than Robin Hood variants but requires more complicated operations and implementations.

- Hopscotch hashing outperforms Robin Hood hashing in workflows with high thread counts.
- Robin Hood hashing outperforms Hopscotch hashing in update-heavy workflows or workflows with high load factors.
- All in all, Robin Hood and hopscotch are two of the most effective hashing algorithms with their own merits depending on the workflow and memory limitations.

Understanding Source Code

I. Initialization

```java
public class HopscotchHashTable<AnyType> {

    private static final int DEFAULT_TABLE_SIZE = 150;
    private static final int RANGE = 8;
    private HashEntry<AnyType>[] array; // The array of elements
    private int occupied; // The number of occupied cells
    private int theSize;

    public HopscotchHashTable() {
        this(DEFAULT_TABLE_SIZE);
    }

    public HopscotchHashTable(int size) {
        allocateArray(size);
        doClear();
    }

    private void allocateArray(int arraySize) {
        array = new HashEntry[arraySize];
    }

    private void doClear() {
        occupied = 0;
        for (int i = 0; i < array.length; i++)
            array[i] = null;
    }

    /**
     * Generate hash value for the input value.
     * @param x
     * @return
     */
    private int myhash(AnyType x) {
        int hashVal = x.hashCode();

        hashVal %= array.length;
        if (hashVal < 0)
            hashVal += array.length;

        return hashVal;
    }
```

Here I have set default table size = 150, and the range for the "hop" = 8. Also, all the initial values are set to zero.

Hashing of the inserted element is also shown.

II. Insert

```java
public boolean insert(AnyType x) {

    if (!isEmpty()) {
        return false;
    }

    int currentPos = findPos(x);

    if (currentPos == -1) {
        return false;
    }

    if (array[currentPos] != null) {
        x = array[currentPos].element;
        array[currentPos].isActive = true;
    }

    String hope;
    if (array[currentPos] != null) {
        hope = array[currentPos].hope;
        x = array[currentPos].element;
    } else {
        hope = "10000000";
    }

    array[currentPos] = new HashEntry<>(x, hope, true);
    theSize++;

    // Rehash;
    // if (++occupied > array.length / 2)
    // rehash();
    // display();
    return true;
}
```

In the insert function first, we check if there is the availability of position in the table using function ***isEmpty***. If there is a position left, we use ***findpos*** function to search for the position. If no position as per standards of hopscotch hashing is available we return false. And if the position is available we generate the neighborhood range details in a string called ***hope***.

Now, for insertion of an element in the range, we add value to that block, if available, or find the new hashentry for the element

III. display

```java
private void display() {
    System.out.println("\n----------------------Display after insert----
    for (int i = 0; i < array.length; i++) {
        if (array[i] != null) {
            System.out.println("index :" + i + " array item :"
                    + array[i].element + " hope:" + array[i].hope);
        } else {
            System.out.println("index :" + i + " array item :"
                    + "--" + " hope:" + "--");
        }
    }
}
```

Displays all the elements in the table after insertion is done

IV. contains

```java
private boolean contains(AnyType x) {

    System.out.println("\n ---------- contains check for element --------------:"+x);

    int currentPos = myhash(x);
    int count = 0;
    while (array[currentPos] != null && count < RANGE) {

        if (array[currentPos].hope.charAt(count) == '1') {
            if (array[currentPos + count].element.equals(x)) {
                return true;
            }
        }
        count++;
    }

    return false;
}
```

Checks if the query is there in the table using a similar method as insertion. It finds the hash-value and checks the neighborhood range as per hopscotch standards. If available, returns *true* and if not available, returns *false*

V. isEmpty

```java
private boolean isEmpty() {
    for (int i = 0; i < array.length; i++) {
        if (array[i] == null) {
            return true;
        }
    }
    return false;
}
```

Checks if the entire table is empty or not.

VI. findPos

```
private int findPos(AnyType x) {
    int offset = 0;
    int currentPos = myhash(x);
    int startPosition = currentPos;
    int original = startPosition;
    boolean flag = false;
    boolean f = false;
    // && !array[currentPos].element.equals(x)
    while (array[currentPos] != null) {
        currentPos++;

        if (currentPos - startPosition >= 8
                || (((currentPos) < startPosition) && (array.length
                        - startPosition + currentPos) >= 8)) {
            f = true;
            System.out.println("flag:" + f);
        }
        if (f
                && ((myhash(array[startPosition].element) - currentPos) >= RANGE
                        || (currentPos - myhash(array[startPosition].element)) >= RANGE
                        || ((currentPos - myhash(array[startPosition].element)) < 0 && (array.length
                        - myhash(array[startPosition].element) + currentPos) >= 8))) {
            flag = true;
            currentPos = nextJumpPosition(startPosition);
            if (currentPos == -2) {
                return -1;
            }
            startPosition = currentPos;
            offset = 0;
        }
        if (currentPos >= array.length) {
            currentPos = 0;
        }
    }
```

Initially setting the boolean for position to false so that it can be changed as per finding for the position according to hopscotch hashing. First, we find a position in the initial range and if not available use the ***nextJumpPosition*** function to find a position after shifts. Here we also handle corner cases line hash value crossing the total length of the hash table, in this case, we start from index zero.

```java
        if (flag == true) {
/           System.out.println("start:"+startPosition+"currentpos"+currentPos);
            array[currentPos] = new HashEntry<>(array[startPosition].element,
                    "00000000", true);
            StringBuilder string = new StringBuilder(
                    array[myhash(array[startPosition].element)].hope);
            if ((currentPos - myhash(array[startPosition].element)) < 0) {
                string.setCharAt(array.length
                        - myhash(array[startPosition].element) + currentPos,
                        '1');
            } else {
                string.setCharAt(currentPos
                        - myhash(array[startPosition].element), '1');
            }
            array[myhash(array[startPosition].element)].hope = string
                    .toString();
            if (array[myhash(array[startPosition].element)].hope
                    .charAt(startPosition
                            - myhash(array[startPosition].element)) == '1') {
                string = new StringBuilder(
                        array[myhash(array[startPosition].element)].hope);
                string.setCharAt(
                        (startPosition - myhash(array[startPosition].element)),
                        '0');
                array[myhash(array[startPosition].element)].hope = string
                        .toString();
            }
            AnyType x1 = x;
            array[startPosition] = new HashEntry<>(x1,
                    array[startPosition].hope, true);
            StringBuilder temp = new StringBuilder(array[myhash(x1)].hope);
            if (startPosition-myhash(x1) < 0) {
                temp.setCharAt(array.length - myhash(x1) + startPosition, '1');
            } else {
                temp.setCharAt(startPosition - myhash(x1), '1');
            }
            array[myhash(x1)].hope = temp.toString();
            currentPos = startPosition;
        } else {
```

In this part after we find a suitable position for the incoming element when the difference between the start and current position is greater than the range. Here, 8.

```java
        } else {
            if (startPosition != currentPos) {
/               System.out.println("=" + currentPos + " start pos"
/                       + startPosition);
                array[currentPos] = new HashEntry<>(x, "00000000", true);
                StringBuilder temp = new StringBuilder(
                        array[startPosition].hope);
                int p;
                if (currentPos - startPosition < 0) {
                    p = array.length - startPosition + currentPos;
                    temp.setCharAt(p, '1');
                } else {
                    temp.setCharAt(currentPos - startPosition, '1');
                }

                AnyType x1 = array[startPosition].element;
                array[startPosition] = null;
                array[startPosition] = new HashEntry<>(x1, temp.toString(),
                        true);
                currentPos = startPosition;
            }
        }

        return currentPos;
    }
```

If the above conditions are not satisfied, a new hashentry is made.

VII. nextJumpPosition

```java
private int nextJumpPosition(int startPosition) {
    int position = startPosition + 1;
    int c = checkHope(position);
    while (c == -1) {
        if (position >= array.length) {
            position = 0;
            c = checkHope(position);
        }
        if (position == startPosition) {
            return -2;
        }
        c = checkHope(position++);
    }
    return c;
}
```

Checks if the next jump is possible or not. If the next jump comes to the start position itself, the function gives a '-2' flag. If the next jump is available the function returns its position.

VIII. checkHope

```java
private int checkHope(int position) {
    if (array[position] != null
            && (!array[position].hope.equals("00000000") && !array[position].hope
                    .equals("00000001"))) {
        for (int i = 0; i < array[position].hope.length(); i++) {
            if (array[position].hope.charAt(i) == '1') {
                return position + i;
            }
        }
    }
    return -1;
}
```

Checks the entry in the string 'hope' for the block of an array for which the function is called.

IX. Input

```java
public static void main(String[] args) {
    HopscotchHashTable<Integer> H = new HopscotchHashTable<>();
    // System.out.println(H.insert(0));
     System.out.println("insert :" +H.insert(1));
     System.out.println("insert :" +H.insert(1));
     System.out.println("insert :" +H.insert(3));
     System.out.println("insert :" +H.insert(2));
     System.out.println("insert :" +H.insert(1));
     System.out.println("insert :" +H.insert(1));
     System.out.println("insert :" +H.insert(1));
     System.out.println("insert :" +H.insert(1));
     System.out.println("insert :" +H.insert(1));
     System.out.println("insert :" +H.insert(1));
     System.out.println("insert :" +H.insert(1));
     System.out.println("insert :" +H.insert(4));
     System.out.println("insert :" +H.insert(5));
     System.out.println("insert :" +H.insert(6));
     System.out.println("insert :" +H.insert(7));
     System.out.println("insert :" +H.insert(8));
     System.out.println("insert :" +H.insert(9));
     System.out.println("insert :" +H.insert(10));
     System.out.println("insert :" +H.insert(25));
     System.out.println("insert :" +H.insert(26));
     System.out.println("insert :" +H.insert(28));

     H.display();

```

```java
        System.out.println("insert :" + H1.insert(73));
        System.out.println("insert :" + H1.insert(73));
        System.out.println("insert :" + H1.insert(76));
        System.out.println("insert :" + H1.insert(75));
        System.out.println("insert :" + H1.insert(74));
        System.out.println("insert :" + H1.insert(72));
        System.out.println("insert :" + H1.insert(73));
        System.out.println("insert :" + H1.insert(73));
        System.out.println("insert :" + H1.insert(148));
        System.out.println("insert :" + H1.insert(148));
        System.out.println("insert :" + H1.insert(148));
        System.out.println("insert :" + H1.insert(148));
        System.out.println("insert :" + H1.insert(73));
        System.out.println("insert :" + H1.insert(73));
        System.out.println("insert :" + H1.insert(73));

        H1.display();
        System.out.println("contains :" + H1.contains(73));
        System.out.println("contains :" + H1.contains(122));

```

For the first image, enters the value to show the input method. For the second image, enters conflicting values to show input and reassignment of positions. Also finally check the values that are in the table and the ones that are not in the table.

X. Output

```
insert :true
insert :true
insert :true
insert :true
insert :true
insert :true
insert :true
insert :true
insert :true
insert :true
insert :false
insert :true
insert :true
insert :true
insert :true
insert :true
insert :true
insert :true
insert :true
insert :true
insert :true

-----------------------Display after insert----------

index :0 array item :-- hope:--
index :1 array item :1 hope:11111111
index :2 array item :1 hope:00000001
index :3 array item :1 hope:00000001
index :4 array item :1 hope:00000001
index :5 array item :1 hope:00000001
index :6 array item :1 hope:00000001
index :7 array item :1 hope:00000001
index :8 array item :1 hope:00000001
index :9 array item :2 hope:00000001
index :10 array item :3 hope:00000001
index :11 array item :4 hope:00000000
index :12 array item :5 hope:00000000
index :13 array item :6 hope:00000000
index :14 array item :7 hope:00000000
index :15 array item :8 hope:00000000
index :16 array item :9 hope:00000000
index :17 array item :10 hope:00000000
index :18 array item :-- hope:--
index :19 array item :-- hope:--
index :20 array item :-- hope:--
index :21 array item :-- hope:--
index :22 array item :-- hope:--
index :23 array item :-- hope:--
index :24 array item :-- hope:--
index :25 array item :25 hope:10000000
index :26 array item :26 hope:10000000
```

Initial insert done, values shown.

```
------------------NEW INSERT--------

insert :true
insert :true
insert :true
insert :true
insert :true
insert :true
insert :true
insert :true
insert :true
insert :true
insert :true
insert :true
insert :true
insert :true
insert :true
```

```
-----------------------Display after insert-

index :0 array item :148 hope:00000000
index :1 array item :148 hope:00000000
index :2 array item :-- hope:--
index :3 array item :-- hope:--
index :4 array item :-- hope:--
index :5 array item :-- hope:--
index :6 array item :-- hope:--
index :7 array item :-- hope:--
index :8 array item :-- hope:--
index :9 array item :-- hope:--
index :10 array item :-- hope:--
index :11 array item :-- hope:--
index :12 array item :-- hope:--
index :13 array item :-- hope:--
index :14 array item :-- hope:--
index :15 array item :-- hope:--
index :16 array item :-- hope:--
index :17 array item :-- hope:--
index :18 array item :-- hope:--
index :19 array item :-- hope:--
index :20 array item :-- hope:--
index :21 array item :-- hope:--
index :22 array item :-- hope:--
index :23 array item :-- hope:--
index :24 array item :-- hope:--
index :25 array item :-- hope:--
index :26 array item :-- hope:--
index :27 array item :-- hope:--
index :28 array item :-- hope:--
index :29 array item :-- hope:--
index :30 array item :-- hope:--
index :31 array item :-- hope:--
index :32 array item :-- hope:--
index :33 array item :-- hope:--
index :34 array item :-- hope:--
index :35 array item :-- hope:--
index :36 array item :-- hope:--
index :37 array item :-- hope:--
index :38 array item :-- hope:--
index :39 array item :-- hope:--
index :40 array item :-- hope:--
index :41 array item :-- hope:--
index :42 array item :-- hope:--
index :43 array item :-- hope:--
index :44 array item :-- hope:--
index :45 array item :-- hope:--
index :46 array item :-- hope:--
index :47 array item :-- hope:--
```

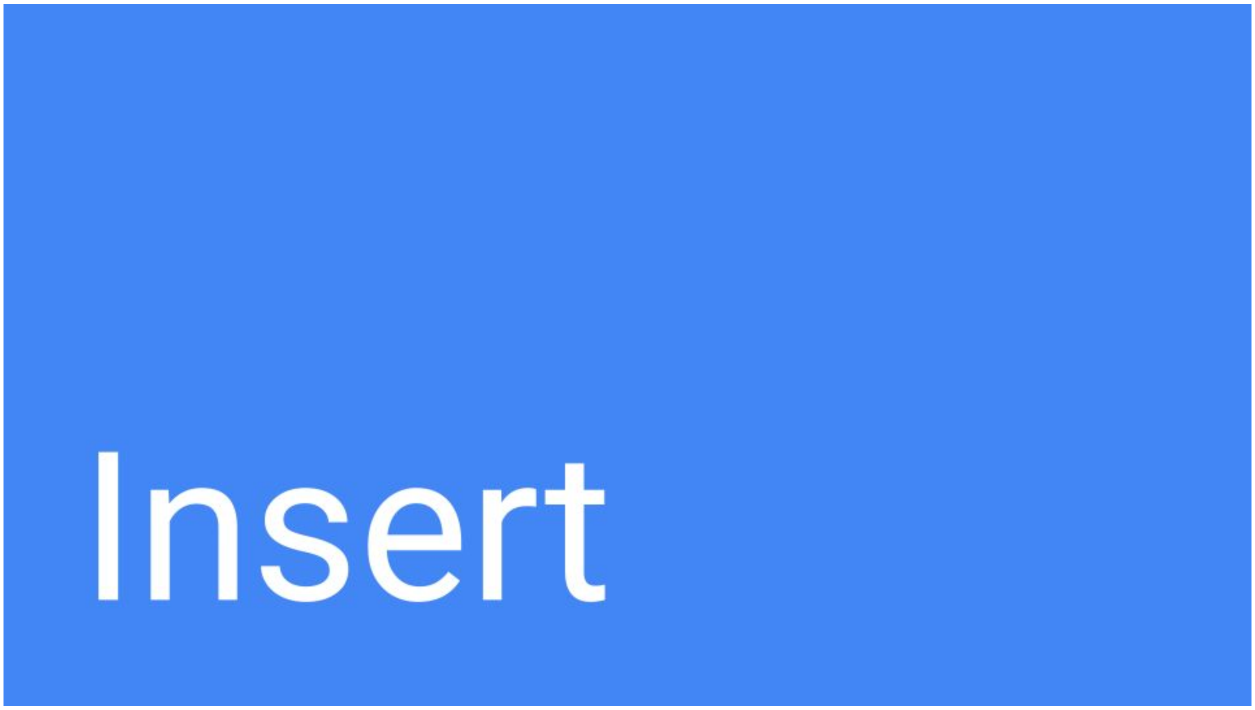
```
index :48 array item :-- hope:--
index :49 array item :-- hope:--
index :50 array item :-- hope:--
index :51 array item :-- hope:--
index :52 array item :-- hope:--
index :53 array item :-- hope:--
index :54 array item :-- hope:--
index :55 array item :-- hope:--
index :56 array item :-- hope:--
index :57 array item :-- hope:--
index :58 array item :-- hope:--
index :59 array item :-- hope:--
index :60 array item :-- hope:--
index :61 array item :-- hope:--
index :62 array item :-- hope:--
index :63 array item :-- hope:--
index :64 array item :-- hope:--
index :65 array item :-- hope:--
index :66 array item :-- hope:--
index :67 array item :-- hope:--
index :68 array item :-- hope:--
index :69 array item :-- hope:--
index :70 array item :-- hope:--
index :71 array item :-- hope:--
index :72 array item :72 hope:10000000
index :73 array item :73 hope:11101111
index :74 array item :73 hope:00000001
index :75 array item :73 hope:00000001
index :76 array item :76 hope:10000000
index :77 array item :73 hope:00000000
index :78 array item :73 hope:00000000
index :79 array item :73 hope:00000000
index :80 array item :73 hope:00000000
index :81 array item :74 hope:00000000
index :82 array item :75 hope:00000000
index :83 array item :-- hope:--
index :84 array item :-- hope:--
index :85 array item :-- hope:--
index :86 array item :-- hope:--
index :87 array item :-- hope:--
index :88 array item :-- hope:--
index :89 array item :-- hope:--
index :90 array item :-- hope:--
index :91 array item :-- hope:--
index :92 array item :-- hope:--
index :93 array item :-- hope:--
index :94 array item :-- hope:--
```

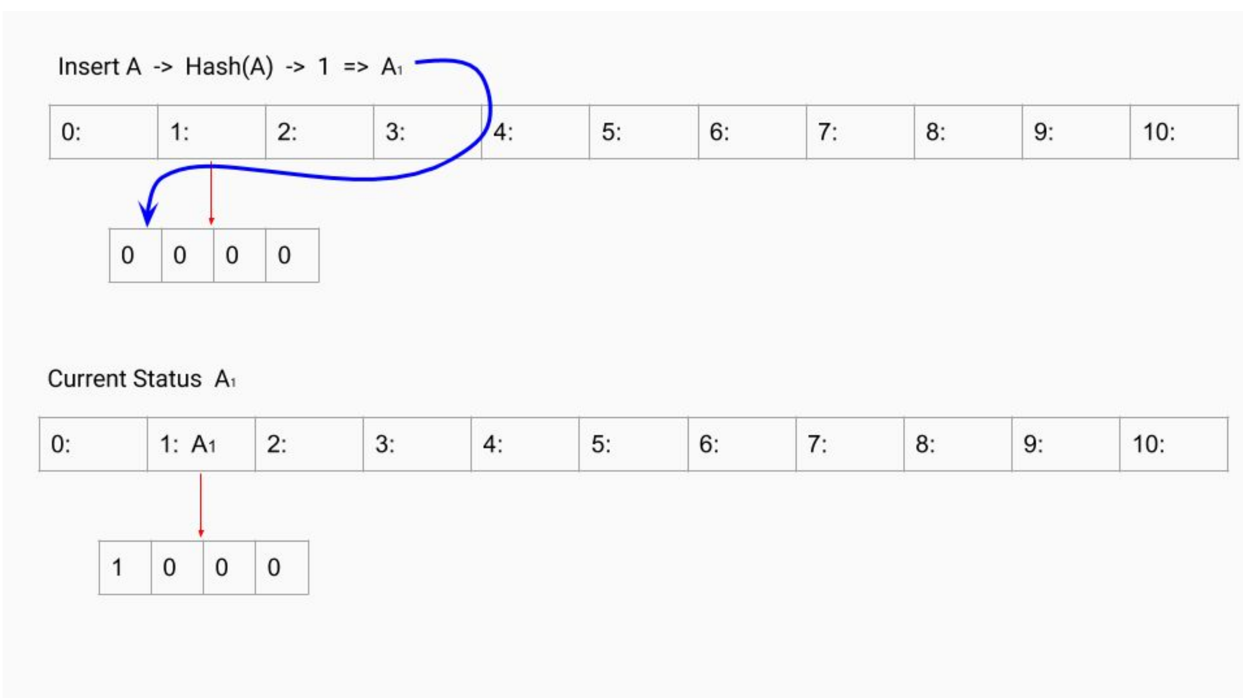
Final inserts

```
index :146 array item :-- hope:--
index :147 array item :-- hope:--
index :148 array item :148 hope:11110000
index :149 array item :148 hope:00000000

 ----------- contains check for element ---------------:73
contains :true

 ----------- contains check for element ---------------:122
contains :false
```

We can see the corner case handling, proper working of ‘hope’ or bit-value position storage, element finding in the table.

XI. Understanding the Concept using Graphics





Insert B -> Hash(B) -> 1 => B_1
Insert C -> Hash(C) -> 1 => C_1
Insert D -> Hash(D) -> 1 => D_1

0:	1: A_1	2:	3:	4:	5:	6:	7:	8:	9:	10:

1	0	0	0

Current Status

0:	1: A_1	2: B_1	3: C_1	4: D_1	5:	6:	7:	8:	9:	10:

1	1	1	1

Insert E -> Hash(E) -> 3 => E_3
Insert F -> Hash(F) -> 5 => F_5

0:	1: A_1	2: B_1	3: C_1	4: D_1	5:	6:	7:	8:	9:	10:

1	1	1	1

0	0	0	0

0	0	1	0

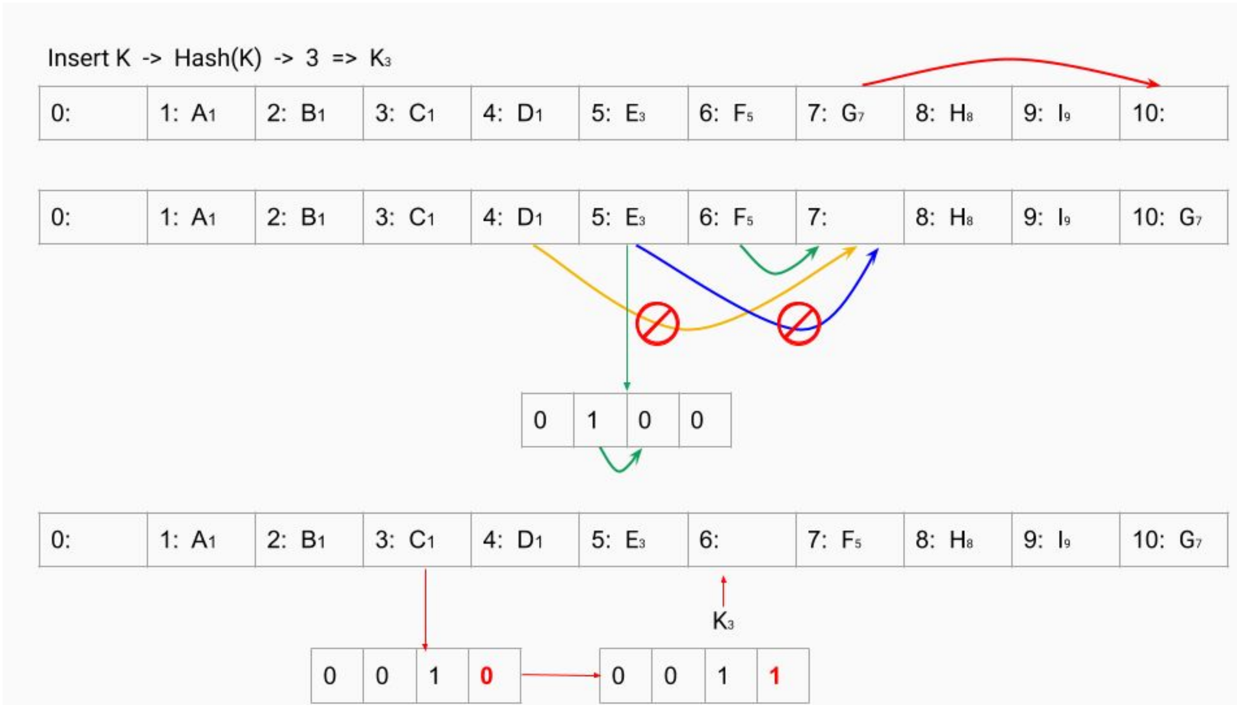
Current Status (after assume G, H, I are inserted)

0:	1: A_1	2: B_1	3: C_1	4: D_1	5: E_3	6: F_5	7: G_7	8: H_8	9: I_9	10:

1	1	1	1

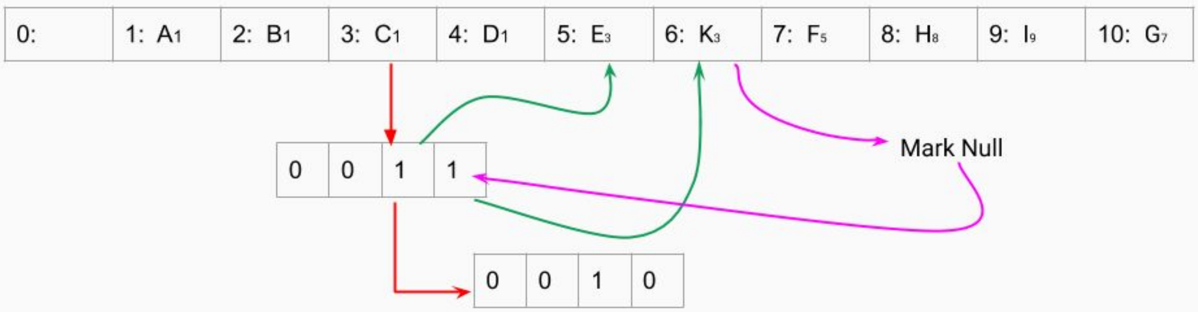
0	1	0	0

0	0	1	0


Insert K -> Hash(K) -> 3 => K_3
0: | 1: A_1 | 2: B_1 | 3: C_1 | 4: D_1 | 5: E_3 | 6: F_5 | 7: G_7 | 8: H_8 | 9: I_9 | 10:
0: | 1: A_1 | 2: B_1 | 3: C_1 | 4: D_1 | 5: E_3 | 6: F_5 | 7: | 8: H_8 | 9: I_9 | 10: G_7
0 | 1 | 0 | 0
0: | 1: A_1 | 2: B_1 | 3: C_1 | 4: D_1 | 5: E_3 | 6: | 7: F_5 | 8: H_8 | 9: I_9 | 10: G_7
K_3
0 | 0 | 1 | 0
0 | 0 | 1 | 1


Find

Delete K -> Find K -> Hash(K) -> 3 => K_3

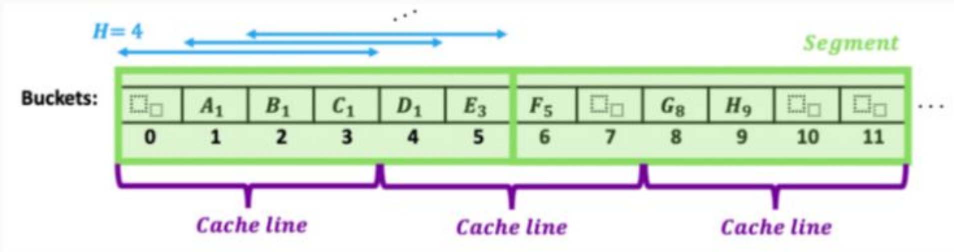


What is special about this **DELETE**

- Super fast update ~O(1)

Cache Support

Basic Structure of Hopscotch table



- For the concurrent version, each segment is protected by a lock, indicated by the green boxes of size 6
- For a cache-optimized variant of the algorithm, The associated cache lines, which are units of data that are fetched from main memory

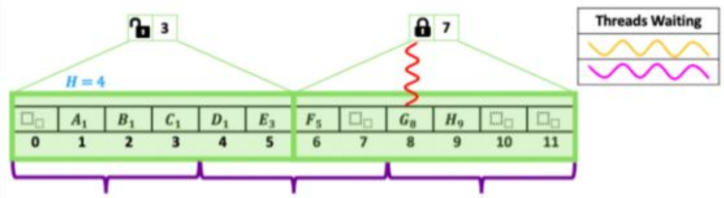
Concurrent Hopscotch Hashing

Concept

We will group buckets together into segments

Each segment will also contain its own lock and timestamp

Each time a segment is modified, that segment's timestamp is incremented by 1



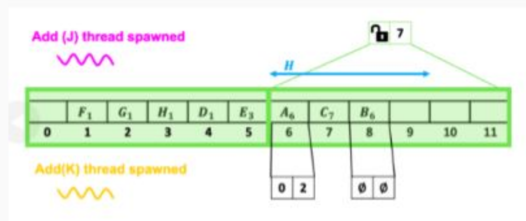
Concurrent hopscotch hash table outline

1. there cannot be any race conditions in the accessing or modification of the timestamp
2. it is possible for the contains operation to loop infinitely

Function: add()

The add operation is identical to the sequential add(k) except for two key differences:

- While traversing the hash table the relevant segment must be locked. This ensures that there are no race conditions during multiple concurrent add/remove calls to the hash table.
- Before returning, the add(k) function will increment the timestamp of the relevant segment and release the lock. The time stamp update is necessary to ensure that concurrent contains(k) re-check the updated hash table.
- worst-case complexity is O(n)



Bibliography

https://en.wikipedia.org/wiki/Hopscotch_hashing | Hopscotch hashing - Wikipedia

https://programming.guide/hopscotch-hashing.html | Hopscotch Hashing | Programming.Guide

https://tessil.github.io/2016/08/29/hopscotch-hashing.html | Hopscotch hashing

http://codecapsule.com/2013/08/11/hopscotch-hashing/ | Hopscotch hashing | Code Capsule

https://medium.com/@michelle.bao1/hopscotch-hashing-2045e7cc176b | Overview of Hopscotch Hashing | Medium

https://medium.com/@michelle.bao1/an-in-depth-analysis-of-hopscotch-hashings-locking-mechanisms-77b1a6f04262 | An In-Depth Analysis of Hopscotch Hashing's Locking Mechanisms | by Michelle Bao | Medium

https://medium.com/better-programming/what-are-hash-tables-and-why-are-they-amazing-89cf52246f91 | What Are Hash Tables and Why Are They Amazing? | by Jonathan Hsu | Better Programming | Medium

https://github.com/udand/HopscotchHashTable/blob/master/HopscothchNew/src/HopscotchHashTable.java

Thanks!

Sridhar M
2018111021
sridhar.m@research.iiit.ac.in