Q. Implement HashMap

Implement put(), get(), remove(), containsKey() ~~keySet(),~~ size() all in O(1) time complexity and keySet()

- Hashmap

    i) is a ~~bucket~~ (array) of LLs

    ii) Time complexity of each type of operation is O(const)

    ↳ not quite correct

    ↳ avg. TC of each type of operation is $O(\lambda)$

    λ: loading factor ~~of~~

    $\lambda = \frac{n}{N} = \frac{\text{no. of elements}}{\text{no. of buckets or cells in array}}$ = average no. of elements per bucket

cuz we search not in all buckets but only in a specific bucket

We keep λ under a threshold (say K)

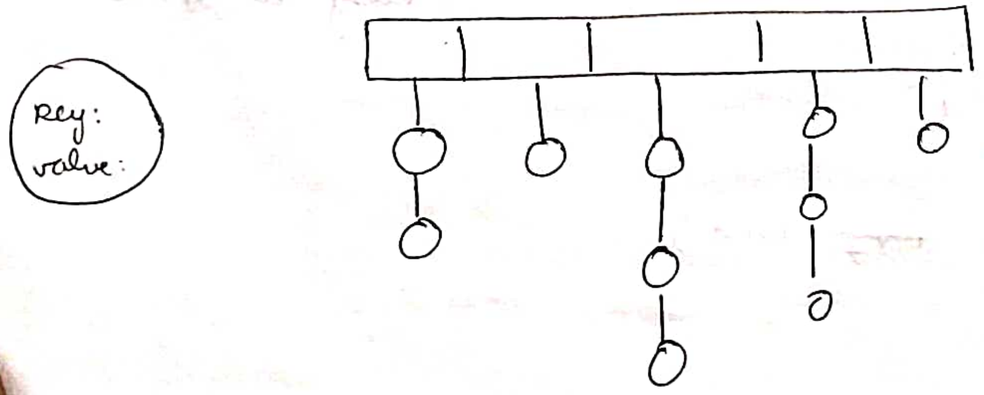
So we can now safely say

<u>average</u> TC of any type of operation in a hashmap is O(const)

↳ note: worst TC is O(n)

all elements in the same bucket

- Typical HashMap structure



• How do we ensure λ always stays ~~within~~ under a threshold ?

Let's say K is the threshold

On addition (insertion not updation) of any element in hashmap, $n \uparrow$ (N stays the same)

$\therefore \lambda \uparrow$ λ might become $> K$

$ { when λ becomes $> K$, we double the number of buckets and reshuffle the elements in the new number of buckets.
↳ costly

Elements ~~in~~ which were in same bucket in old Hashmap might not be in same bucket in new hashmap

Thus by doing a costly operation once we ensure ~~∅~~ ~~TC~~ const TC in average case. next reshuffling will not take place immediately cuz λ is now $\sim \frac{K}{2}$

$ is called rehashing

• How do we know which element should be placed in which bucket ?

$$\begin{tikzcd}
\text{key} \arrow[r] & \boxed{\text{hash function}} \arrow[r] \arrow[d] & \text{always unique } \underline{\text{valid}} \text{ bucket index } \sout{\text{twithin}} \\
\text{i/p} & \text{works in } O(1) & \text{o/p}
\end{tikzcd}$$

```
~~pv~~
class   HashMap1 < K, V>
{
oo      ~~HMNode~~
        class   HMNode
        {
              K   key ;
              V   value ;
        ~~}~~
            HMNode( K Key, V value)
            {
                this.key = key;
                this.value= value;
            }
        }


        int   size ;  //n
        LinkedList <HMNode > buckets [] ;
        // N = buckets. length


        HashMap~~1~~1 ()
        {
             initbuckets (4);
             size =0 ;
        }


       ~~init~~
       void  initbuckets (int N)
       {
         1 buckets =   new   LinkedList [N];

          for (int bi =0 ; bi < buckets.length; bi++)
                  buckets [bi] =  new  LinkedList <>();
        2
       }
```

```
↳ void  put (K key, V value)
  {
      int bi = hashFunction(key);
      int di = getIndexWithinBucket(bi, key);

      if (di == -1)
      {
          HMNode node = new HMNode(key, value);
          buckets[bi].addLast(node);  // O(1)
          size++;   // don't miss
      }
      else
      {
          HMNode node = buckets[bi].get(di);
          node.value = value;
      }

      double lambda
      = (size * 1.0)/buckets.length;

      if(lambda > 2)   // K = 2
          rehashing();
  }

↳ V  get (K key)
  {
      int bi = hashFunction(key);
      int di = getIndexWithinBucket(bi, key);
      if (di == -1)
          return null;
      else
          return buckets[bi].get(di).value;
  }
```

bi: bucket index

di: data index within a particular bucket

note: We've never get element using index in LL previously

```
↳ boolean containsKey (K key)
  {
      int bi = hashFunction (key);
      int di = getIndexWithinBucket (bi, key);

      if (di == -1)
            return false;
      else
            return true;
  }

↳   V   remove (K key)
    {
      int bi = hashFunction (key);
      int di = getIndexInBucket (bi, key);

      if (di == -1)
          return null;
      else
      {
          HMNode node = buckets[bi] . remove(d
          size --;   // don't miss
          return node. value;
      }
    }

↳ ArrayList < K >   keySet ()
  {
      ArrayList <K> list = new ArrayList <>()

      for (int bi = 0 ; bi < buckets. length ; bi++)
      {
            for (HMNode node : buckets [bi])
                  list. add (node. key);
      }
      return list;
```

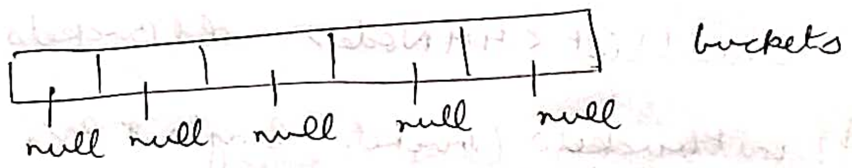
```
↳ int size()
  {
      return size;
  }


↳ private int hashFunction (K key)
  {
      return Math.abs(key.hashCode()) %
                                    buckets.length;


      // key.hasCode() not hashCode(key)
      // hashCode can return -ve integer as well
  }


↳ private void rehashing()
  {
      LinkedList<HMNode> oldBuckets[] = buckets;
  $$ initbuckets(buckets.length * 2);
  /* ~~// too~~ 'buckets' now points to twice the
     old size memory in heap
  */

      size = 0;  // very easy to miss

      for(int bi=0; bi< oldBuckets.length; bi++)
      {
                              oldBuckets
          for(HMNode node : ~~buckets~~ [bi])
          {  ~~if(node.key.equals(key))~~ ~~//key can be String~~
                ~~return idx;~~             ~~.equals()~~
              $$ put(node.key, node.value);
          }
```

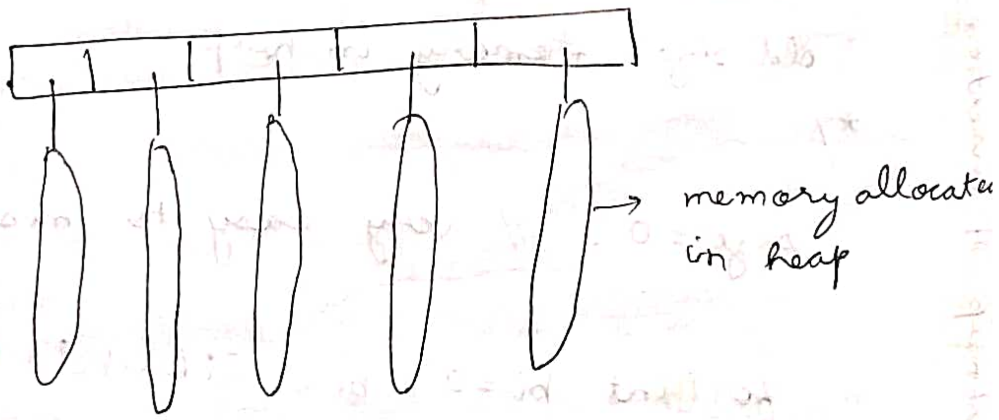
$$ नए hashmap में insertion

```
↳ private int getIndexWithinBucket (int bi, K ke
{
        int idx = 0;

        for ( HMNode node : buckets [bi])
        {
            if(node. key . equals (key))
                    return idx;

                idx ++;
        }

        return -1;
}
```

key can be String ∴ equals()

}
oo

1 

2. 

1 LL का न होना और
2 खाली LL होना अलग - अलग बातें है