**Department of Software Engineering**
**Mehran University of Engineering and Technology, Jamshoro**

| Course: SWE224 – Data Structure & Algorithms | | | |
|---|---|---|---|
| **Instructor** | Mariam Memon | **Practical/Lab No.** | 08 |
| **Date** | | **CLOs** | 3 |
| **Signature** | | **Assessment Score** | 01 |

| **Topic** | **Implementation of Hash Tables** |
|---|---|
| **Objectives** | To implement Hash Tables and its operations in Java. |

## Lab Discussion: Theoretical concepts and Procedural steps

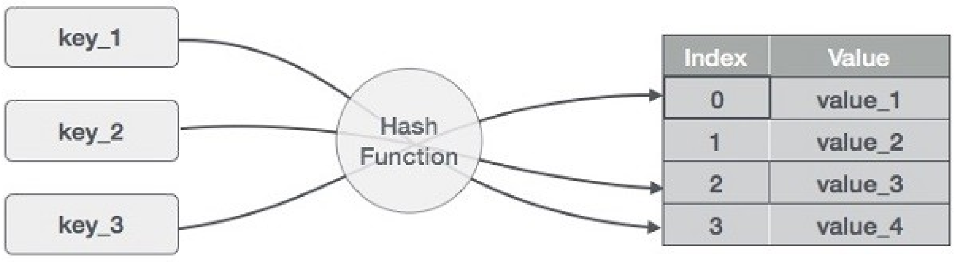
### Hash Tables

Hash Table is a data structure which stores data in an associative manner. In a hash table, data is stored in an array format, where each data value has its own unique index value. Access of data becomes very fast if we know the index of the desired data.

Thus, it becomes a data structure in which insertion and search operations are very fast irrespective of the size of the data. Hash Table uses an array as a storage medium and uses hash technique to generate an index where an element is to be inserted or is to be located from.

### Hashing

Hashing is a technique to convert a range of key values into a range of indexes of an array. We're going to use modulo operator to get a range of key values. Consider an example of hash table of size 20, and the following items are to be stored. Item are in the (key,value) format.



- (1,20)
- (2,70)
- (42,80)
- (4,25)
- (12,44)

- (14,32)
- (17,11)
- (13,78)
- (37,98)

| Sr.No. | Key | Hash | Array Index |
|---|---|---|---|
| 1 | 1 | 1 % 20 = 1 | 1 |
| 2 | 2 | 2 % 20 = 2 | 2 |
| 3 | 42 | 42 % 20 = 2 | 2 |
| 4 | 4 | 4 % 20 = 4 | 4 |
| 5 | 12 | 12 % 20 = 12 | 12 |
| 6 | 14 | 14 % 20 = 14 | 14 |
| 7 | 17 | 17 % 20 = 17 | 17 |
| 8 | 13 | 13 % 20 = 13 | 13 |
| 9 | 37 | 37 % 20 = 17 | 17 |

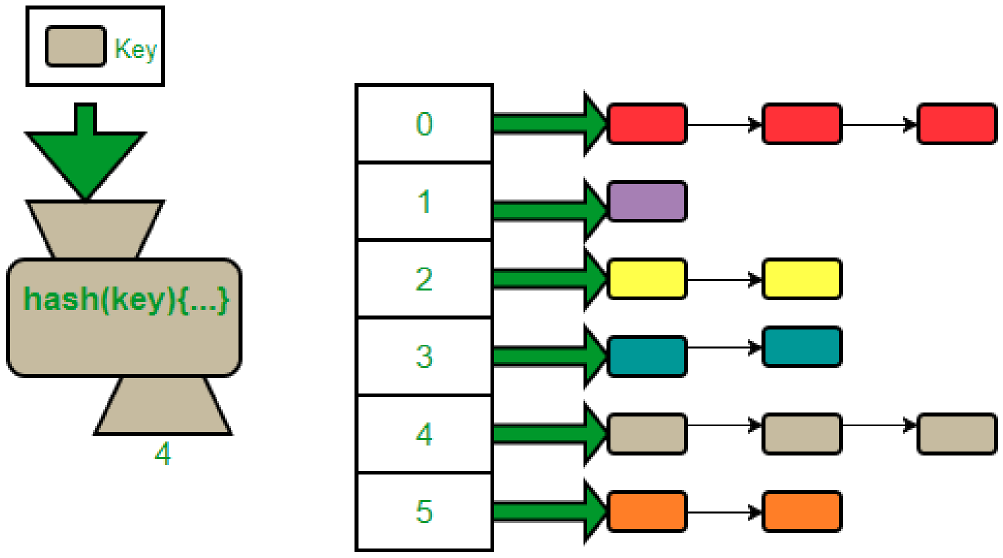
**Linear Probing**

As we can see, it may happen that the hashing technique is used to create an already used index of the array. In such a case, we can search the next empty location in the array by looking into the next cell until we find an empty cell. This technique is called linear probing.

| Sr.No. | Key | Hash | Array Index | After Linear Probing, Array Index |
|---|---|---|---|---|
| 1 | 1 | 1 % 20 = 1 | 1 | 1 |
| 2 | 2 | 2 % 20 = 2 | 2 | 2 |
| 3 | 42 | 42 % 20 = 2 | 2 | 3 |
| 4 | 4 | 4 % 20 = 4 | 4 | 4 |
| 5 | 12 | 12 % 20 = 12 | 12 | 12 |
| 6 | 14 | 14 % 20 = 14 | 14 | 14 |
| 7 | 17 | 17 % 20 = 17 | 17 | 17 |
| 8 | 13 | 13 % 20 = 13 | 13 | 13 |
| 9 | 37 | 37 % 20 = 17 | 17 | 18 |

**Separate Chaining**

The idea is to make each cell of hash table point to a linked list of records that have same hash function value.



**Basic Operations**

- **put(Object key, Object value)** – Adds a key-value pair in the hash table
- **get(Object key)** – finds the value from the hash table using the key and returns it.
- **remove(Object key)** – removes the element from the hash table with key equal to the key passed as argument.
- **size()** – returns the number of elements in the hash table.

**HashTable Implementation in Java with Linear Probing**

```
public class HashTable implements Map
{
      private Entry[] entries;
      private int size, used;
      private float loadFactor;
      private final Entry NIL = new Entry(null,null);
      public HashTable(int capacity, float loadFactor)
      {
            entries = new Entry[capacity];
            this.loadFactor = loadFactor;
      }
      public HashTable(int capacity)
      {
            this(capacity, 0.75F);
```

```
}
public HashTable()
{
      this(101);
}
private class Entry
{
      Object key, value;
      Entry(Object k, Object v){this.key=k; this.value=v;}
}
public Object get(Object key)
{
      int h = hash(key);
      for(int i = 0; i<entries.length; i++)
      {
            int j = nextProbe(h,i);
            Entry entry = entries[j];
            if (entry == null) break;
            if (entry == NIL) continue;
            if(entry.key.equals(key)) return entry.value;
      }
      return null;
}
public Object put(Object key, Object value)
{
      if(used > loadFactor*entries.length) rehash();
      int h = hash(key);
      for(int i=0; i<entries.length; i++)
      {
            int j = nextProbe(h,i);
            Entry entry = entries[j];
            if (entry == null){
                  entries[j] = new Entry(key, value);
                  ++size; ++used;
                  return null;
            }
            if(entry == NIL) continue;
            if(entry.key.equals(key))
            {
                  object oldValue = entry.value;
                  entries[j].value = value;
                  return oldValue;
            }
      }
      return null;
}
public Object remove(Object key)
{
      int h = hash(key);
      for(int i=0; i<entries.length; i++)
      {
            int j = nextProbe(h,i);
            Entry entry = entries[j];
            if (entry == null) break;
            if(entry == NIL) continue;
            if(entry.key.equals(key))
            {
                  object oldValue = entry.value;
                  entries[j] = NIL;
                  --size;
                  return oldValue;
            }
      }
      return null;
}
public int size()
{
      return size;
}

private int hash(object key)
{
      if(key==null) throw new IllegalArgumentException();
      return (key.hashCode() & 0x7FFFFFFF) % entry.length;
}
private int nextProbe(int h, int i)
{
      return (h+i)%entry.length;
}
private void rehash()
{
      Entry[] oldEntries = entries;
      entries = new Entry[2*oldEntries.length+1];
      for(int k = 0; k < oldEntries.length; k++)
      {
            Entry entry = oldEntries[k];
            if(entry == null || entry == NIL) continue;
            int h = hash(entry.key);
            for(int i = 0; i<entries.length; i++)
            {
                  int j = nextProbe(h,i);
                  if(entries[j]==null)
                  {
                        entries[j] = entry;
                        break;
                  }
            }
      }
      used = size;
   }
}
```

## HashTable Implementation in Java with Separate Chaining

```
public class HashTable2 implements Map
{
      private Entry[] entries;
      private int size;
      private float loadFactor;
      public HashTable(int capacity, float loadFactor)
      {
            entries = new Entry[capacity];
            this.loadFactor = loadFactor;
      }
      public HashTable(int capacity)
      {
            this(capacity, 0.75F);
      }
      public HashTable()
      {
            this(101);
      }
      private class Entry
      {
            Object key, value;
            Entry next;
            Entry(Object k, Object v, Entry n)
            {this.key=k; this.value=v; this.next=n;}
      }
      public Object get(Object key)
      {
            int h = hash(key);
            for(Entry e=entries[h]; e!=null; e=e.next)
```

```
        {
                if(e.key.equals(key)) return e.value;
        }
        return null;
}
public Object put(Object key, Object value)
{
        int h = hash(key);
        for(Entry e=entries[h]; e!=null; e=e.next)
        {
                if(e.key.equals(key))
                {
                        Object oldValue = e.value;
                        e.value = value;
                        return oldValue;
                }
        }
        entries[h]=new Entry(key, value, entries[h]);
        ++size;
        if(size > loadFactor*entries.length) rehash();
        return null;
}
public Object remove(Object key)
{
        int h = hash(key);
        for(Entry e=entries[h], prev=null; e!=null; prev=e, e=e.next)
        {
                if(e.key.equals(key))
                {
                        Object oldValue = e.value;
                        if(prev==null)entries[h] = e.next;
                        else prev.next = e.next;
                        --size;
                        return oldValue;
                }
        }
        return null;
}
public int size()
{
        return size;
}

private int hash(Object key)
```

```
        {
                if(key==null) throw new IllegalArgumentException();
                return (key.hashCode() & 0x7FFFFFFF) % entries.length;
        }
        private void rehash()
        {
                Entry[] oldEntries = entries;
                entries = new Entry[2*oldEntries.length+1];
                for(int k = 0; k < oldEntries.length; k++)
                {
                        for(Entry old = oldEntries[k]; old!=null; )
                        {
                                Entry e = old;
                                old = old.next;
                                int h = hash(e.key);
                                e.next = entries[h];
                                entries[h] = e;
                        }
                }
        }
}
```

## Lab Tasks

**Task 1:** Use quadratic probing to overcome collision problem of hashing.

**Task 2:** Write a program that compares linear probing and quadratic probing. Insert at least 10 items in the hash table. Each time the data is saved, the number of collisions that were required to save it using linear and quadratic probing should be displayed.

**Task 3:** Explore java.util.HashTable class and use any 5 of its methods in your code.

## Lab Rubrics for Evaluation

| **Rubrics** | **Proficient** | **Adequate** | **Poor** |
|---|---|---|---|
| **Programming Algorithms** | Completely accurate and efficient design of algorithms **(0.4)** | Accurate but inefficient algorithms **(0.2)** | Unable to design algorithms for the given problems **(0.0)** |

| | | | |
|---|---|---|---|
| **Originality** | Submitted work shows large amount of original thought **(0.3)** | Submitted work shows some amount of original thought **(0.2)** | Submitted work shows no original thought **(0.0)** |
| **Troubleshooting** | Easily traced and corrected faults **(0.3)** | Traced and corrected faults with some guidance **(0.2)** | No troubleshooting skills **(0.0)** |