(4) Collision Handling

- If 2 values are assigned to same key by the hash fns.

Eg:- Birthday Paradox

23 people in room → 50% same birthday of 2 people

70 people in room → 99.9% same birthday of 2 people.

a) No collision → Perfect Hashing
- very adv method, not in this course.
- should know keys in adv.

b) Handling collision
- 2 methods
  - Chaining
  - Open Addressing

---

(5) Chaining

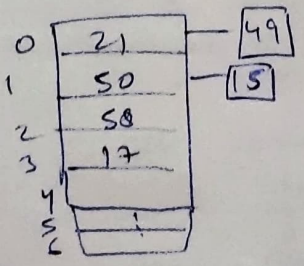
key also called index
Value also called key.

- Use linked list for each key.

∴ If collision happens, we insert the element in the next place of that key's linked list.

Eg:- Hash fn → Value % 7

Values → {50, 21, 58, 17, 15, 49}



(Load factor) $\alpha = \frac{\text{No. of elements}}{\text{No. of buckets}}$

Performance

- Expected time for Search → $O(1+\alpha)$
  " " " Insert/Delete → $O(1+\alpha)$

- Self balancing BST used from Java 8 in Hashmap, instead of LList as they take only log(n) for all operations.

Implementation algorithm

```
Class MyHash
{
  int Bucket;
  ArrayList <LinkedList< Integer>> table;
  MyHash(int b)
  {  Bucket = b;
     table = new ArrayList<LinkedList<Integer>>();
     for (int i=0; i<b; i++)
        { table.add(new LinkedList<Integer>()); }
  }

  void insert(int key)
  { int i = key % Bucket;
    table.get(i).add(key);
  }

  void remove(int key)
  {
    int i = key % Bucket;
    table.get(i).remove((Integer)i);
  }
```

Typecasted to call obj fn of remove

```
boolean search (int key)
{   int i = key % Bucket;
    return table.get(i).contains(key);
}
```

(b) Open Addressing

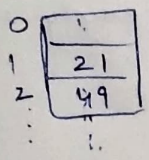
Basic req → No: of slots in hash table ≥ No: of keys to be inserted.

→ Cache friendly.

→ 3 ways to implement → Linear Probing, Double Hashing & Quadratic Probing.

a) Linear Probing

~~Insert~~ → If collision happens, linearily search for next empty slot and insert there.



{ - - - 21, 49 - - - } Hash fn → value % 7.

21 collided to 49. then next free in 2.

* If last slot occupied we go back to first slot & search again.

Searching → Go to index after computing hash fn.

If we find key return true.

else linearly search.

Stop when

- Find key
- Find an empty slot
- Traversed through whole table

Delete

→ while deleting we can't just leave slot empty as it will create problem for next searches.

→ Insert deleted slot which will → won't stop search
→ value can be inserted.

Problems with linear probing

→ Clusters forming, making all operations costly.

---

b) Quadratic Probing

→ hash(key, i) = (h(key) + i²) % m

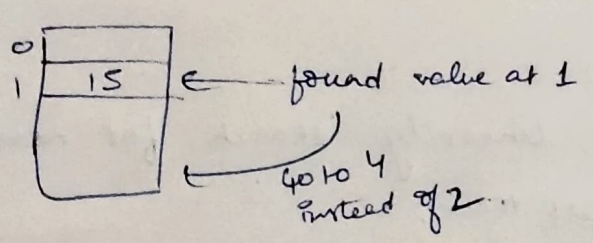
↳ But same problem of clustering.

For linear probing used

hash(key, i)

→ (h(key) + i) % 7

where h(key) = key % 7



→ Problem 2 → won't be able to find empty slot even if present.

---

c) Double Hashing

→ As the name suggests 2 hash functions are used.

$$hash(key, i) = (h_1(key) + i h_2(key)) \% m$$

- No clustering.
- Distributes keys more uniformly.
- If $h_2(key)$ is relatively prime to m you will always find a free slot.

Eg:

49, 63, 56, 52, 54, 48

Q

0	49
1	
2	54
3	63
4	56
5	52
6	48

$m = 7$

$h_1(key) = (key \% 7)$

$h_2(key) = 6 - (key \% 6)$

Sol

49 → h_1 key only. → 0

63 → $6 - (63 \% 6)$

→ $6 - 3$ → 3

∴ $(0 + 1 \times 3) \% 7$

→ 3.

Simi → 56 at 4

54 → $h_1(key)$ → 5

Collision

∴ $6 - (54 \% 6)$

$= 6$ → $(5 + 6) \% 7$ → 4

Again collision, we increment i. in hash fn.

i++

∴ $(5 + (6 \times 2)) \% 7$ → 3

Again collision, i++

$(5 + (6 \times 3)) \% 7$ → 2 → Insert at 2.

52 → $6 - (52 \% 6)$

→ $6 - 4$ → 2

∴ → $(3 + 2) \% 7$

→ 5

Implementation of Open Addressing - (Linear Probing only to keep things simple).

```
Class MyHash {
        int [] arr;
        int cap, size;
        MyHash(int c)
        { cap = c
          size = 0;
          for (int i=0; i<n; i++)
            { arr[i] = -1; }
        }

        int hash(int key)
        { return key % cap; }

        boolean Search (int key)
        {
            int h = hash(key);
            int i = h;
            while (arr[i] != -1)
            { if (arr[i] == key)
                    return true;
              i = (i+1) % cap;
              if (i == h) return false;
            }
        }
```

```
boolean Insert (int key)
{   if (size == cap) { return false; }

    int i = hash(key);
    while (arr[i] != -1 && arr[i] != -2
            && arr[i] != key)
    { i = (i+1) % cap; }

    if (arr[i] == key) { return false; }
    else { arr[i] = key;
           size ++;
           return true;
         }
}
```

for delete, just search and if true make arr[i] = -2.

Chaining is better than open add but more size needed.

dynamic ⟶ Chaining.

Know count of keys ⟶ Open addressing.

Know all keys ⟶ Perfect hashing.