UNIT –III

Red-Black Trees, *Splay Trees, Applications.* ***Hash Tables:*** *Introduction, Hash Structure, Hash functions, Linear Open Addressing, Chaining and Applications.*

3.1 Red - Black Tree

Red - Black Tree is another variant of Binary Search Tree in which every node is colored either RED or BLACK. We can define a Red Black Tree as follows...

Red Black Tree is a Binary Search Tree in which every node is colored either RED or BLACK.

In Red Black Tree, the color of a node is decided based on the properties of Red-Black Tree. Every Red Black Tree has the following properties.

Properties of Red Black Tree

- **Property #1:** Red - Black Tree must be a Binary Search Tree.
- **Property #2:** The ROOT node must be colored BLACK.
- **Property #3:** The children of Red colored node must be colored BLACK. (There should not be two consecutive RED nodes).
- **Property #4:** In all the paths of the tree, there should be same number of BLACK colored nodes.
- **Property #5:** Every new node must be inserted with RED color.
- **Property #6:** Every leaf (e.i. NULL node) must be colored BLACK.

Example

Following is a Red-Black Tree which is created by inserting numbers from 1 to 9.

The above tree is a Red-Black tree where every node is satisfying all the properties of Red-Black Tree.

Every Red Black Tree is a binary search tree but every Binary Search Tree need not be Red Black tree.

Insertion into RED BLACK Tree

In a Red-Black Tree, every new node must be inserted with the color RED. The insertion operation in Red Black Tree is similar to insertion operation in Binary Search Tree. But it is inserted with a color property. After every insertion operation, we need to check all the properties of Red-Black Tree.

If all the properties are satisfied then we go to next operation otherwise we perform the following operation to make it Red Black Tree.

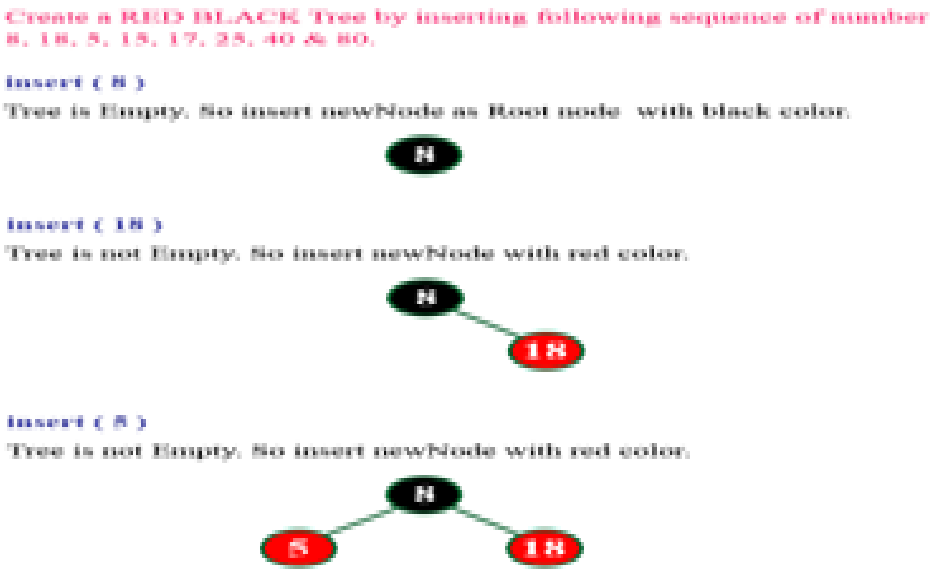
- **1. Recolor**
- **2. Rotation**
- **3. Rotation followed by Recolor**

The insertion operation in Red Black tree is performed using the following steps...

- **Step 1 -** Check whether tree is Empty.
- **Step 2 -** If tree is Empty then insert the **newNode** as Root node with color **Black** and exit from the operation.
- **Step 3 -** If tree is not Empty then insert the newNode as leaf node with color Red.
- **Step 4 -** If the parent of newNode is Black then exit from the operation.
- **Step 5 -** If the parent of newNode is Red then check the color of parentnode's sibling of newNode.
- **Step 6 -** If it is colored Black or NULL then make suitable Rotation and Recolor it.
- **Step 7 -** If it is colored Red then perform Recolor. Repeat the same until tree becomes Red Black Tree.

Deletion Operation in Red Black Tree

The deletion operation in Red-Black Tree is similar to deletion operation in BST. But after every deletion operation, we need to check with the Red-Black Tree properties. If any of the properties are violated then make suitable operations like Recolor, Rotation and Rotation followed by Recolor to make it Red-Black .

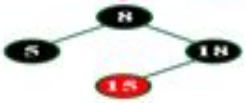


Insert (15)

Tree is not Empty. So insert newNode with red color.

Here there are two consecutive Red nodes (18 & 15).
The newnode's parent sibling color is Red
and parent's parent is root node.
So we use RECOLOR to make it Red Black Tree.

After RECOLOR



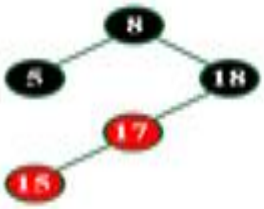
After Recolor operation, the tree is satisfying all Red Black Tree properties.

Insert (17)

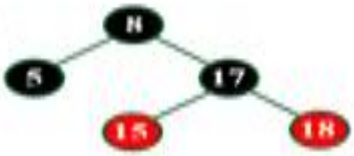
Tree is not Empty. So insert newNode with red color.

Here there are two consecutive Red nodes (15 & 17).
The newnode's parent sibling is NULL. So we need rotation.
Here, we need LR Rotation & Recolor.

After Left Rotation

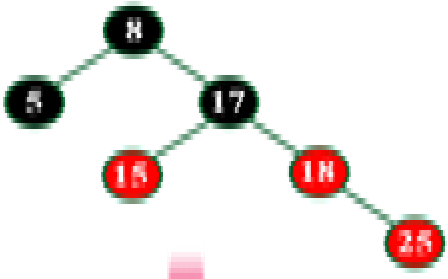


After Right Rotation & Recolor



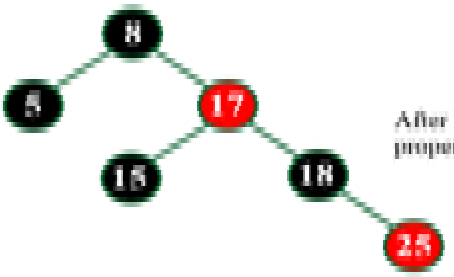
Insert (25)

Tree is not Empty. So insert newNode with red color.



Here there are two consecutive Red nodes (18 & 25).
The newnode's parent sibling color is Red
and parent's parent is not root node.
So we use RECOLOR and Recheck.

After Recolor



After Recolor operation, the tree is satisfying all Red Black Tree properties.

3.2 Splay Tree Data structure

Splay tree is another variant of a binary search tree. In a splay tree, recently accessed element is placed at the root of the tree. A splay tree is defined as follows...

Splay Tree is a self - adjusted Binary Search Tree in which every operation on element rearranges the tree so that the element is placed at the root position of the tree.

In a splay tree, every operation is performed at the root of the tree. All the operations in splay tree are involved with a common operation called "Splaying".

Splaying an element, is the process of bringing it to the root position by performing suitable rotation operations. In a splay tree, splaying an element rearranges all the elements in the tree so that splayed element is placed at the root of the tree. By splaying elements we bring more frequently used elements closer to the root of the tree so that any operation on those elements is performed quickly. That means the splaying operation automatically brings more frequently used elements. . Every operation on splay tree performs the splaying operation. For example, the insertion operation first inserts the new element using the binary search tree insertion process, then the newly inserted element is splayed so that it is placed at the root of the tree. The search operation in a splay tree is nothing but searching the element using binary search process .In splay tree, to splay any element we use the following rotation operations...

Rotations in Splay Tree

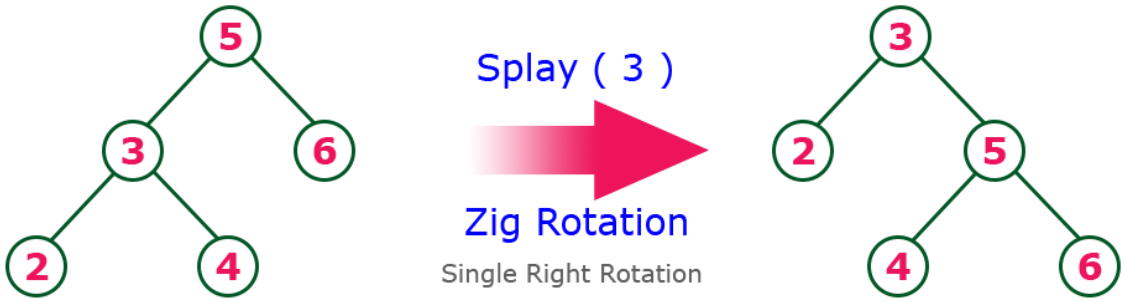
- **1. Zig Rotation**
- **2. Zag Rotation**
- **3. Zig - Zig Rotation**
- **4. Zag - Zag Rotation**
- **5. Zig - Zag Rotation**
- **6. Zag - Zig Rotation**

Example

Zig Rotation

The **Zig Rotation** in splay tree is similar to the single right rotation in AVL Tree rotations. In zig rotation, every node moves one position to the right from its current position.

Consider the following example...

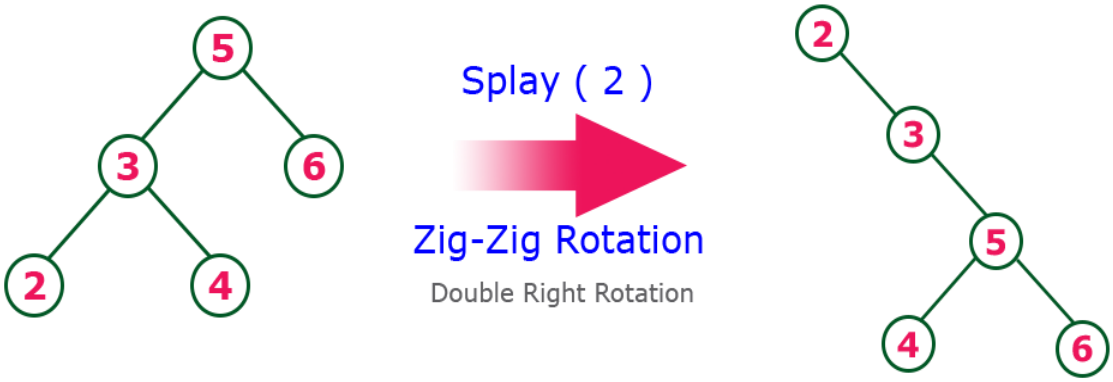


Zag Rotation

The **Zag Rotation** in splay tree is similar to the single left rotation in AVL Tree rotations. In zag rotation, every node moves one position to the left from its current position. Consider the following example...

Zig-Zig Rotation

The **Zig-Zig Rotation** in splay tree is a double zig rotation. In zig-zig rotation, every node moves two positions to the right from its current position. Consider the following example...

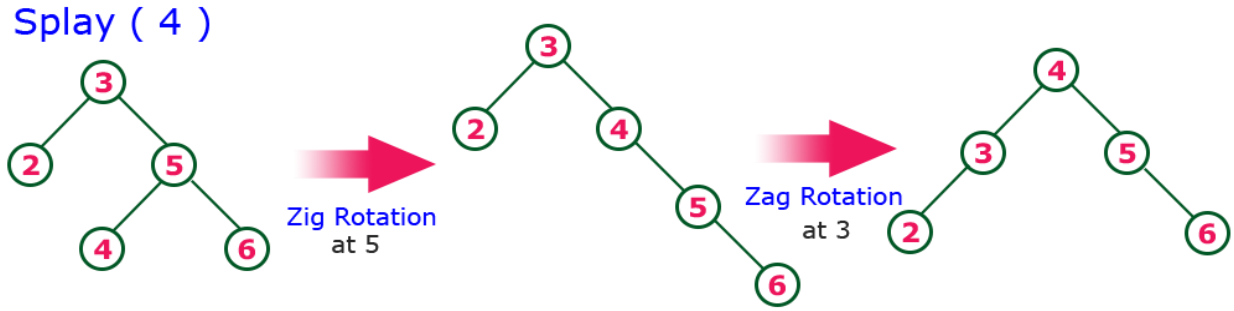


Zag-Zag Rotation

The **Zag-Zag Rotation** in splay tree is a double zag rotation. In zag-zag rotation, every node moves two positions to the left from its current position. Consider the following example...

Zig-Zag Rotation

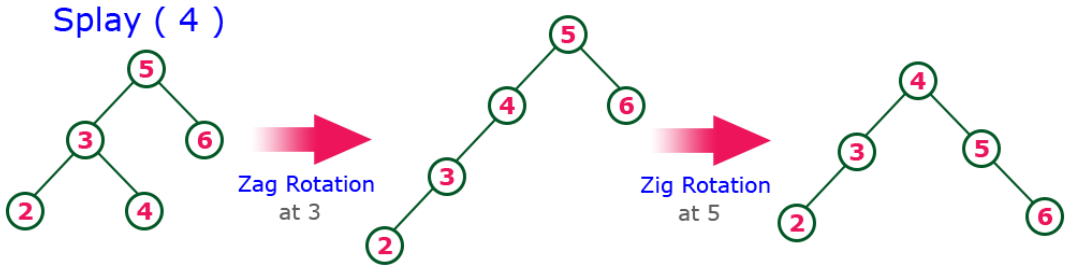
The **Zig-Zag Rotation** in splay tree is a sequence of zig rotation followed by zag rotation. In zig-zag rotation, every node moves one position to the right followed by one position to the left from its current position. Consider the following example...



Zag-Zig Rotation

The **Zag-Zig Rotation** in splay tree is a sequence of zag rotation followed by zig rotation. In zag-zig rotation, every node moves one position to the left followed by one position to the right from its current position.

Consider the following example...



Every Splay tree must be a binary search tree but it is need not to be balanced tree.

Insertion Operation in Splay Tree

The insertion operation in Splay tree is performed using following steps...

- **Step 1 -** Check whether tree is Empty.
- **Step 2 -** If tree is Empty then insert the **newNode** as Root node and exit from the operation.
- **Step 3 -** If tree is not Empty then insert the newNode as leaf node using Binary Search tree insertion logic.
- **Step 4 -** After insertion, **Splay** the **newNode**

Deletion Operation in Splay Tree

The deletion operation in splay tree is similar to deletion operation in Binary Search Tree. But before deleting the element, we first need to **splay** that element and then delete it from the root position. Finally join the remaining tree using binary search tree logic.

Applications:

Decision-based algorithm is used in machine learning which works upon the algorithm of tree. Databases also uses tree data structures for indexing. **Domain Name Server(DNS)** also uses tree structures. File explorer/my computer of mobile/any computer.

Some applications of the trees are:

1. XML Parser uses tree algorithms.
2. Decision-based algorithm is used in machine learning which works upon the algorithm of tree.
3. Databases also uses tree data structures for indexing.
4. Domain Name Server(DNS) also uses tree structures.
5. File explorer/my computer of mobile/any computer
6. BST used in computer Graphics
7. Posting questions on websites like Quora, the comments are child of questions

3.3 Hash Tables :

Introduction:

We've seen searches that allow you to look through data in $O(n)$ time, and searches that allow you to look through data in $O(logn)$ time, but imagine a way to find exactly what you want in $O(1)$ time. Think it's not possible? Think again! Hash tables allow the storage and retrieval of data in an average time of $O(1)$.

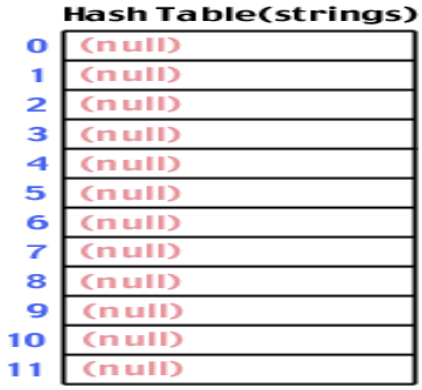
At its most basic level, a hash table data structure is just an array. Data is stored into this array at specific indices designated by a hash function. A hash function is a mapping between the set of input data and a set of integers.

With hash tables, there always exists the possibility that two data elements will hash to the same integer value. When this happens, a collision results (two data members try to occupy the same place in the hash table array),

and methods have been devised to deal with such situations. In this guide, we will cover two methods, linear probing and separate chaining, focusing on the latter.

A hash table is made up of two parts: an array (the actual table where the data to be searched is stored) and a mapping function, known as a hash function. The hash function is a mapping from the input space to the integer space that defines the indices of the array. In other words, the hash function provides a way for assigning numbers to the input data such that the data can then be stored at the array index corresponding to the assigned number.

Let's take a simple example. First, we start with a hash table array of strings (we'll use strings as the data being stored and searched in this example). Let's say the hash table size is 12:



Next we need a hash function. There are many possible ways to construct a hash function. We'll discuss these possibilities more in the next section. For now, let's assume a simple hash function that takes a string as input. The returned hash value will be the sum of the ASCII characters that make up the string mod the size of the table:

*int hash(char *str, int table_size) { int sum; /* Make sure a valid string passed in */ if (str==NULL) return -1; /*
Sum up all the characters in the string */ for(; *str; str++) sum += *str; /* Return the sum mod the table size */ return sum % table_size; }*

Now that we have a framework in place, let's try using it. First, let's store a string into the table: "Steve".

We run "Steve" through the hash function, and find that hash("Steve",12) yields 3:

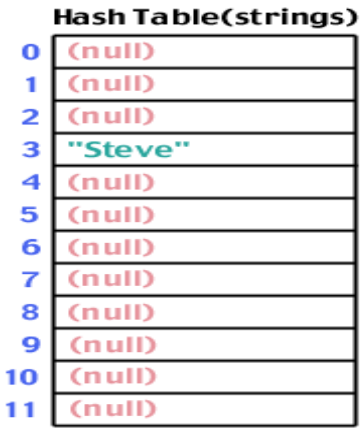


Figure %: The hash table after inserting "Steve"

Let's try another string: "Spark". We run the string through the hash function and find that hash("Spark",12) yields 6. Fine. We insert it into the hash table:

Figure %: T he hash table after inserting "Spark"

Let's try another: "Notes". We run "Notes" through the hash function and find that hash("Notes",12) is 3. Ok. We insert it into the hash table:

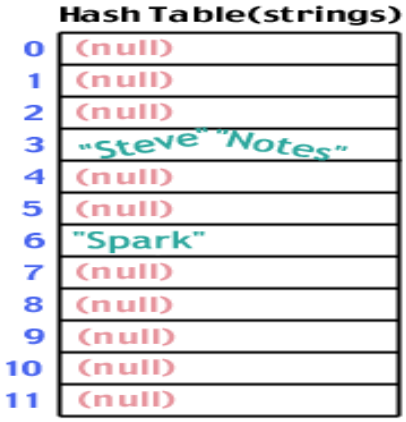


Figure %: A hash table collision

What happened? A hash function doesn't guarantee that every input will map to a different output. There is always the chance that two inputs will hash to the same output. This indicates that both elements should be inserted at the same place in the array, and this is impossible. This phenomenon is known as a collision.

There are many algorithms for dealing with collisions, such as linear probing an d separate chaining. While each of the methods has its advantages, we will only discuss separate chaining here.

Separate chaining requires a slight modification to the data structure. Instead of storing the data elements right into the array, they are stored in linked lists. Each slot in the array then points to one of these linked lists. When an element hashes to a value, it is added to the linked list at that index in the array. Because a linked list has no limit on length, collisions are no longer a problem. If more than one element hashes to the same value, then both

are stored in that linked list.

Let's look at the above example again, this time with our modified data structure:

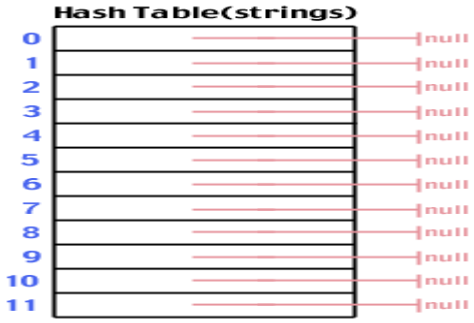


Figure %: Modified table for separate chaining

Again, let's try adding "Steve" which hashes to 3:

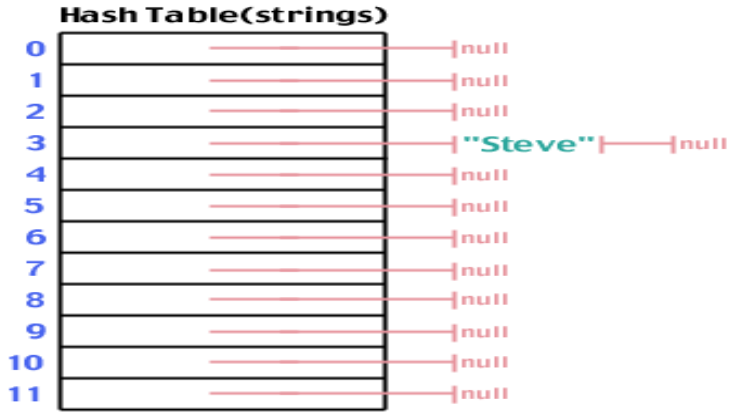


Figure %: After adding "Steve" to the table And "Spark" which hashes to 6:

Problem : How does a hash table allow for $O(1)$ searching? What is the worst case efficiency of a look up in a hash table using separate chainging?

A hash table uses hash functions to compute an integer value for data. This integer value can then be used as an index into an array, giving us a constant time access to the requested data. However, using separate chaining, we won't always achieve the best and average case efficiency of $O(1)$. If we have too small a hash table for the data set size and/or a bad hash function, elements can start to build in one index in the array. Theoretically, all n element could end up in the same linked list. Therefore, to do a search in the worst case is equivalent to looking up a data element in a linked list, something we already know to be $O(n)$ time. However, with a good hash function and a well created hash table, the chances of this happening are, for all intents and purposes, ignorable. **Problem :** The bigger the ratio between the size of the hash table and the number of data elements, the less chance there is for collision. What is a drawback to making the hash table big enough so the chances of collision is ignorable?
Wasted memory space

Problem : How could a linked list and a hash table be combined to allow someone to run through the list from item to item while still maintaining the ability to access an individual element in $O(1)$ time?

Hash Functions

As mentioned briefly in the previous section, there are multiple ways for constructing a hash function. Remember that hash function takes the data as input (often a string), and return s an integer in the range of possible indices into the hash table. Every hash function must do that, including the bad ones. So what makes for a good hash function?

Characteristics of a Good Hash Function

There are four main characteristics of a good hash function:
1) The hash value is fully determined by the data being hashed.
2) The hash function uses all the input data.
3) The hash function "uniformly" distributes the data across the entire set of possible hash values.
4) The hash function generates very different hash values for similar strings. Let's examine why each of these is important:

Rule 1: If something else besides the input data is used to determine the hash, then the hash value is not as dependent upon the input data, thus allowing for a worse distribution of the hash values.
Rule 2: If the hash function doesn't use all the input data, then slight variations to the input data would

cause an inappropriate number of similar hash values resulting in too many collisions.

Rule 3: If the hash function does not uniformly distribute the data across the entire set of possible hash values, a large number of collisions will result, cutting down on the efficiency of the hash table.

Rule 4: In real world applications, many data sets contain very similar data elements.

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 Inex	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

Basic Operations

Following are the basic primary operations of a hash table.

Search − Searches an element in a hash table.

Insert − inserts an element in a hash table.

delete − Deletes an element from a hash table.

DataItem

Define a data item having some data and key, based on which the search is to be conducted in a hash table.

```
struct DataItem {
int data;
  int key;
};
```

Hash Method

Define a hashing method to compute the hash code of the key of the data item.

```
int hashCode(int key){
   return key % SIZE;
}
```

Search Operation

Whenever an element is to be searched, compute the hash code of the key passed and locate the element using that hash code as index in the array. Use linear probing to get the element ahead if the element is not found at the computed hash code.

Insert Operation

Whenever an element is to be inserted, compute the hash code of the key passed and locate the index using that hash code as an index in the array. Use linear probing for empty location, if an element is found at the computed hash code.

Delete Operation

Whenever an element is to be deleted, compute the hash code of the key passed and locate the index using that hash code as an index in the array. Use linear probing to get the element ahead if an element is not found at the computed hash code. When found, store a dummy item there to keep the performance of the hash table intact.

OpenAddressing

Like separate chaining, open addressing is a method for handling collisions. In Open Addressing, all elements are stored in the hash table itself. So at any point, the size of the table must be greater than or equal to the total number of keys (Note that we can increase table size by copying old data if needed).

Insert(k): Keep probing until an empty slot is found. Once an empty slot is found, insert k.

Search(k): Keep probing until slot's key doesn't become equal to k or an empty slot is reached.

Delete(k): ***Delete operation is interesting***. If we simply delete a key, then the search may fail. So slots of deleted keys are marked specially as "deleted". The insert can insert an item in a deleted slot, but the search doesn't stop at a deleted slot.

Open Addressing is done in the following ways:

a) Linear Probing: In linear probing, we linearly probe for next slot. For example, the typical gap between two probes is 1 as seen in the example below.

Let **hash(x)** be the slot index computed using a hash function and **S** be the table size

If slot hash(x) % S is full, then we try (hash(x) + 1) % S

If (hash(x) + 1) % S is also full, then we try (hash(x) + 2) % S

If (hash(x) + 2) % S is also full, then we try (hash(x) + 3) % S

Let us consider a simple hash function as "key mod 7" and a sequence of keys as 50, 700, 76, 85, 92, 73, 101.

Challenges in Linear Probing :

1. **Primary Clustering:** One of the problems with linear probing is Primary clustering, many consecutive elements form groups and it starts taking time to find a free slot or to search for an element.
2. **Secondary Clustering*:*** Secondary clustering is less severe, two records only have the same collision chain (Probe Sequence) if their initial position is the same.

b) Quadratic Probing We look for i^2'th slot in i'th iteration.

let hash(x) be the slot index computed using hash function.

If slot hash(x) % S is full, then we try (hash(x) + 1*1) % S

If (hash(x) + 1*1) % S is also full, then we try (hash(x) + 2*2) % S

If (hash(x) + 2*2) % S is also full, then we try (hash(x) + 3*3) % S

c) Double Hashing We use another hash function hash2(x) and look for i*hash2(x) slot in i'th rotation.

let hash(x) be the slot index computed using hash function.

If slot hash(x) % S is full, then we try (hash(x) + 1*hash2(x)) % S

If (hash(x) + 1*hash2(x)) % S is also full, then we try (hash(x) + 2*hash2(x)) % S

If (hash(x) + 2*hash2(x)) % S is also full, then we try (hash(x) + 3*hash2(x)) % S

Comparison:

Linear probing has the best cache performance but suffers from clustering. One more advantage of Linear probing is easy to compute. Quadratic probing lies between the two in terms of cache performance and clustering.

Double hashing has poor cache performance but no clustering. Double hashing requires more computation time as two hash functions need to be computed.

S.No.	**Separate Chaining**	**Open Addressing**
1.	Chaining is Simpler to implement.	Open Addressing requires more computation.
2.	In chaining, Hash table never fills up, we can always add more elements to chain.	In open addressing, table may become full.
3.	Chaining is Less sensitive to the hash function or load factors.	Open addressing requires extra care to avoid clustering and load factor.
4.	Chaining is mostly used when it is unknown how many and how frequently keys may be inserted or deleted.	Open addressing is used when the frequency and number of keys is known.
5.	Cache performance of chaining is not good as keys are stored using linked list.	Open addressing provides better cache performance as everything is stored in the same table.
6.	Wastage of Space (Some Parts of hash table in chaining are never used).	In Open addressing, a slot can be used even if an input doesn't map to it.
7.	Chaining uses extra space for links.	No links in Open addressing

PerformanceofOpenAddressing:

Like Chaining, the performance of hashing can be evaluated under the assumption that each key is equally likely to be hashed to any slot of the table (simple uniform hashing)

Applications of Hashing

- Message Digest.
- Password Verification.
- Data Structures(Programming Languages)
- Compiler Operation.
- Rabin-Karp Algorithm.
- Linking File name and path together.

PART-A (2 Marks)

1. Define splay tree.

Ans: Splay tree is a binary search tree in which restructuring is done using a scheme called splay. The splay is a heuristic method which moves a given vertex v to the root of the splay tree using a sequence of rotations.

2. What is the idea behind splaying?

Ans: Splaying reduces the total accessing time if the most frequently accessed node is moved towards the root. It does not require to maintain any information regarding the height or balance factor and hence saves space and simplifies the code to some extent

3. Define hashing function.

Ans: A hashing function is a key-to-transformation, which acts upon a given key to compute the relative position of the key in an array. A simple hash function HASH(KEY_Value)= (KEY_Value)mod(Table-size).

4. What is open addressing?

Ans: Open addressing is also called closed hashing, which is an alternative to resolve the collisions with linked lists. In this hashing system, if a collision occurs, alternative cells are tired until an empty cell is found.

5. What are the collision resolution methods?

Ans: The following are the collision resolution methods:

- Separate chaining
- Open addressing
- Multiple hashing

6. Define separate chaining It is an open hashing technique.

Ans: A pointer field is added to each record location, when an overflow occurs, this pointer is set to point to overflow blocks making a linked list. In this method, the table can never overflow, since the linked lists are only extended upon the arrival of new keys.

7. Define Hashing.

Ans: Hashing is the transformation of string of characters into a usually shorter fixed length

value or key that represents the original string. Hashing is used to index and retrieve items in a database because it is faster to find the item using the short hashed key than to find it using the original value.

8. What do you mean by hash table?

Ans: The hash table data structure is merely an array of some fixed size, containing the keys. A key is a string with an associated value. Each key is mapped into some number in the range 0 to tablesize-1 and placed in the appropriate cell.

9. Write the importance of hashing.

Ans: • Maps key with the corresponding value using hash function.

• Hash tables support the efficient addition of new entries and the time spent on searching for the required data is independent of the number of items stored.

10. What do you mean by collision in hashing?

Ans: When an element is inserted, it hashes to the same value as an already inserted element, and then it produces collision.

11. What are the collision resolution methods?

Ans: • Separate chaining or External hashing

• Open addressing or Closed hashing.

12. What do you mean by open addressing?

Ans: Open addressing is a collision resolving strategy in which, if collision occurs alternative cells are tried until an empty cell is found. The cells h0(x), h1(x), h2(x),…. are tried in succession, where hi(x)=(Hash(x)+F(i))mod Tablesize with F(0)=0. The function F is the collision resolution strategy.

13. List the limitations of linear probing.

Ans: • Time taken for finding the next available cell is large.

• In linear probing, we come across a problem known as clustering.

14. Mention one advantage and disadvantage of using quadratic probing.

Ans: Advantage: The problem of primary clustering is eliminated.

Disadvantage: There is no guarantee of finding an unoccupied cell once the table is nearly half

full.

15. what are the properties of red black tree?

Ans: **Each tree node is colored either red or black. The root node of the tree is always black**. Every path from the root to any of the leaf nodes must have the same number of black nodes. No two red nodes can be adjacent, i.e., a red node cannot be the parent or the child of another red node.

PART-B (10 Marks)

1. Explain Operations on Red Black tree.
2. Explain Operations on splay tree.
3. Explain Chaining method and Open Addressing with an example
4. Explain double hashing and Bucket hashing with an example
5. Explain difference between Linear Probing and Quadratic Probing with an example
6. Explain Rehashing and Extendible hashing with an example