DS Mid 1 QnA

1. a) Explain the difference between linear data structures and non linear data structures.

b) Explain in detail about double linked list.

a)

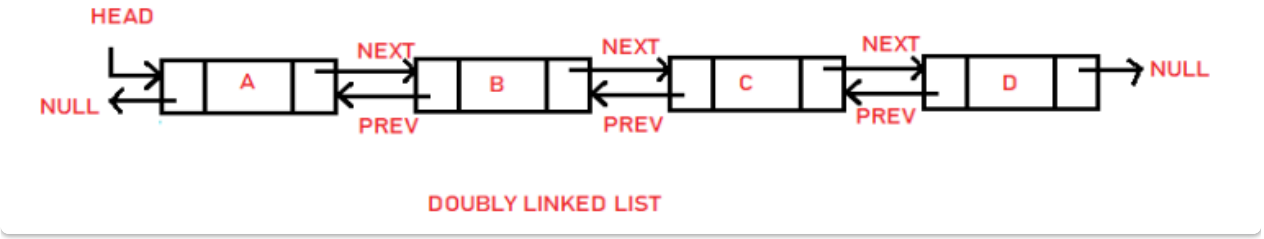
Difference between Linear and Non-linear Data Structures:

S. No.	Linear Data Structure	Non-linear Data Structure
1.	Elements arranged linearly, each attached to previous and next.	Elements attached hierarchically.
2.	Single level involved.	Multiple levels involved.
3.	Implementation is relatively easy.	Implementation is complex.
4.	Elements traversed in a single run.	Elements can't be traversed in a single run.
5.	Memory utilization is less efficient.	Memory is utilized efficiently.
6.	Examples: array, stack, queue, linked list.	Examples: trees and graphs.
7.	Applications in application software development.	Applications in Artificial Intelligence and image processing.

b)

Doubly Linked List:

A doubly linked list is a more complex type of linked list where each node, in addition to storing its data, has two links. The first link points to the previous node in the list, and the second link points to the next node in the list.



Applications of Doubly Linked List:

1. **Browser Navigation:**
   - In web browsers, the back or next function to change tabs utilizes the concept of a doubly-linked list.
2. **Implementation of Other Data Structures:**
   - Doubly linked lists are easily used to implement other data structures like binary trees, hash tables, stacks, etc.
3. **Music Playing System:**
   - In music playing systems, a doubly linked list allows easy navigation between the previous and next songs.
4. **Thread Scheduler in Operating Systems:**
   - Many operating systems use a doubly-linked list in the thread scheduler to maintain a list of all processes running. This facilitates easy movement of a process from one queue to another.
5. **Deck of Cards in Games:**
   - The concept of a doubly linked list is used in games, particularly in representing a deck of cards.

2. a) Write a program to implement stack operations

b) What are the applications of Queues ? Explain.

a)

Stack Operations Algorithms:

1. Push Operation Algorithm:

```
push(int data)
if stack is full
  print error message
else
  top = top + 1
  a[top] = data
```

2. Pop Operation Algorithm:

```
pop()
if stack is empty
  print error message
else
  print a[top]
  top = top - 1
```

3. Traverse (Display) Operation Algorithm:

```
display()
if stack is empty
  print error message
else
  print a[top] to a[0]
```

Auxiliary Stack Operations:

- peek():

```
peek()
if stack is empty
  print error message
```

```
else
  return a[top]
```

- size():

```
size()
return top + 1
```

- isEmpty():

```
isEmpty()
return top == -1
```

- isFull():

```
isFull()
return top == n - 1
```

b)

Queue:

A queue is a linear data structure that follows the First-In-First-Out (FIFO) order. This means that the item inserted first will be the first to be accessed or removed. Queues can be implemented using arrays or linked lists and are commonly used in scenarios where resources are shared among multiple users and served on a first-come, first-serve basis. Examples of queue usage include CPU scheduling and disk scheduling.

Applications of Queue:

1. **Handling Website Traffic:** Queues are used to manage website traffic by handling incoming requests in the order they are received. This ensures fair access to web services.
2. **Playlist Management in Media Players:** Media players use queues to manage playlists. Songs are played in the order they are added to the queue.
3. **Operating Systems and Interrupts:** Queues are employed in operating systems to manage and prioritize interrupt requests from hardware devices, ensuring timely processing.
4. **Resource Scheduling:** Queues help in managing shared resources such as printers, CPUs, or other critical system resources. Tasks are queued and serviced based on priority.
5. **Asynchronous Data Transfer:** Queues are crucial for asynchronous data transfer in computer systems, including communication through pipes, file I/O, and sockets.
6. **Job Scheduling:** Operating systems use queues to schedule processes and jobs, ensuring that tasks are executed in a controlled and efficient manner.
7. **Social Media Uploads:** Social media platforms use queues to manage the upload of multiple photos or videos, ensuring a smooth and organized user experience.
8. **Email Transmission:** Email servers use queues to store and send email messages. Messages are queued and processed in the order they are received.
9. **Handling Website Traffic Simultaneously:** Queues are used to manage website traffic during periods of high load, ensuring that the server can handle multiple requests concurrently.
10. **Windows Operating System:** In Windows and other operating systems, queues are used to switch between multiple applications and manage tasks running on the computer.

Real-Life Examples of Queue:

1. **One-Way Road:** In traffic management, a one-way road follows the queue principle, where vehicles that enter first will exit first. This ensures a smooth flow of traffic.
2. **Ticket Windows:** Ticket counters at various places, such as movie theaters, train stations, or bus terminals, operate on a first-come-first-served basis, resembling a queue.
3. **Cashier Lines:** In stores and supermarkets, customers form queues at cashier lines to complete their transactions. This system ensures fairness and order.
4. **Escalators:** People waiting to use escalators form queues. Those who arrive first board the escalator before those who arrive later.

3. What is single linked list write an algorithm for insertion, deletion and search operations using single linked list.

Singly Linked List

A singly linked list is a data structure comprising nodes, each containing data and a pointer to the address of the next node in the list. The size of linked list elements is not fixed, allowing for efficient memory utilization.

Node Declaration:

```
struct node
{
  int data;
  struct node *next;
};
```

Insertion Operations:

1. **Insertion at the Beginning:**
   - The new node is added before the head of the linked list, becoming the new head.
   - Steps:
     1. Allocate a new node.
     2. Assign data to the new node.
     3. Set the next of the new node as the current head.
     4. Move the head to point to the new node.

```
void insert_begin(struct node *head, int data)
{
  struct node *p = malloc(sizeof(struct node));
  p->data = data;
  p->next = head;
  head = p;
}
```

2. **Insertion After a Given Node:**
   - A new node is inserted after the given node.
   - Steps:
     1. Allocate a new node.
     2. Assign data to the new node.
     3. Find the given previous node in the list.
     4. Set the next of the new node as the next of the given previous node.
     5. Update the next of the previous node to the new node.

```
void insertAfter(struct node *head, int data, int x)
{
  struct node *p = malloc(sizeof(struct node));
  p->data = data;
  struct node *t = head;
```

```
  while (x != t->data)
    t = t->next;
  p->next = t->next;
  t->next = p;
}
```

3. **Insertion at the End:**
   - The new node is added after the last node in the linked list.
   - Steps:
     1. Allocate a new node.
     2. Assign data to the new node.
     3. Traverse the list to the last node.
     4. Set the next of the last node to the new node.

```
void insert_end(struct node *head, int data)
{
  struct node *p = malloc(sizeof(struct node));
  p->data = data;
  p->next = NULL;
  struct node *t = head;
  while (t->next != NULL)
    t = t->next;
  t->next = p;
}
```

Understanding these insertion operations is fundamental for manipulating singly linked lists efficiently.

Delete from a Linked List:

Nodes in a linked list can be deleted in three ways: at the beginning, at the end, or at a specified position in the middle.

Delete at Beginning:

To delete a node at the beginning, update the head to point to the next node and free the memory of the removed node.

```
void del_begin(struct node *head)
{
  struct node *t = head;
  head = head->next;
  free(t);
}
```

Delete at End:

To delete a node at the end, find the last second element, change its `next` pointer to `NULL`, and free the memory of the removed node.

```
void del_end(struct node *head)
{
  struct node *t = head;
  struct node *p;
  while (t->next != NULL)
  {
    p = t;
    t = t->next;
  }
  if (t == head)
    head = NULL;
  else
    p->next = NULL;
  free(t);
}
```

Delete a Given Node:

To delete a node at a specified position, keep track of the pointer (`p`) before the node to delete and the pointer (`t`) to the node to delete. Update the `next` pointer of the previous node or the `head`

if the first node is deleted, and free the memory of the removed node.

```
void del(struct node *head, int x)
{
  struct node *t = head;
  struct node *p;
  while (x != t->data)
  {
    p = t;
    t = t->next;
  }
  if (t == head)
    head = head->next;
  else
    p->next = t->next;
  free(t);
}
```

Displaying a Singly Linked List:

To display the elements of a singly linked list, you can follow these steps:

1. **Check whether the list is Empty:**
   - If `head == NULL`, display "List is Empty!!!" and terminate the function.
2. **If Not Empty:**
   - Define a Node pointer `temp` and initialize it with `head`.
3. **Display Elements:**
   - Keep displaying `temp → data` with an arrow (`--->`) until `temp` reaches `NULL`.

```
t = head;
while (t != NULL)
```

```
{
  printf("%d-->", t->data);
  t = t->next;
}
printf("NULL\n");
```

Applications of Singly Linked List:

1. **Implementing Stacks and Queues:**
   - Fundamental data structures in computer science.
2. **Collision Prevention in Hash Maps:**
   - Singly linked lists are used to handle collisions in hash maps.
3. **Notepad Functions:**
   - Casual notepad applications use singly linked lists for undo, redo, and deleting functions.
4. **Photo Viewer in Slide Show:**
   - Singly linked lists can be used in a photo viewer for continuous slideshow functionality.
5. **Train System:**
   - Analogous to a singly linked list, the system of adding a new train bogie involves either adding it at the end or finding a spot in between bogies to insert.

Understanding these applications showcases the versatility and usefulness of singly linked lists in various scenarios.

4. Define and explain in detail about Binary search tree. Explain insertion operation using one example.

Binary Search Trees:

* A Binary Search Tree is a special kind of binary tree that satisfies the following conditions.

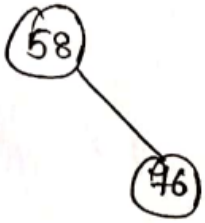
1. The data elements of the left subtree are smaller than the root of the tree.
2. The data elements of the right subtree are greater than or equal to the root of the tree.
3. The left sub tree and right subtree are also the binary search trees. i.e., they must also follow the above two rules.

Construction of a Binary Search Tree (BST):-

* Construct the BST of 58, 76, 14, 63, 63, 78, 43, 6, 11, 61.

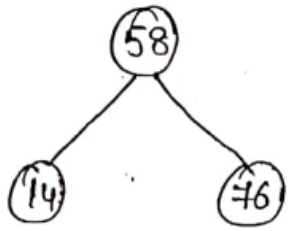
Step 1: Initially, the tree is empty so place the first no at the root.

Step 2: Compare the next number (76) with the root. If the incoming no. is greater than or equal to the root then place it in the right child position. Otherwise place it into the left child position.

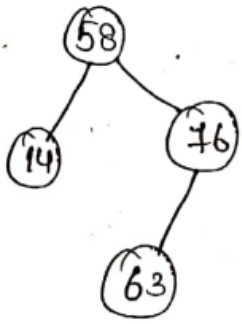


Step 3: Compare the next number (14) with the root number

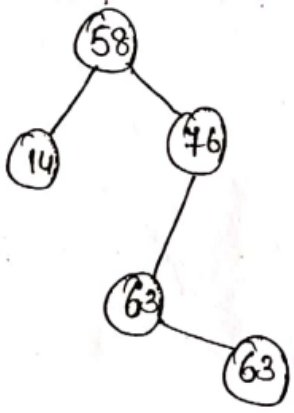
if it is less than the root so examine the left subtree.



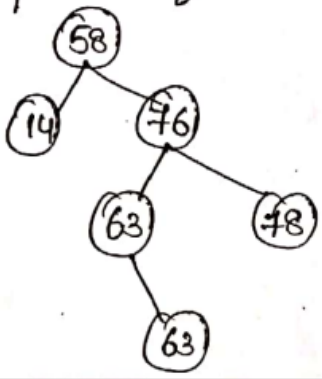
Step 4: Now, the next incoming number 63 is greater than 58 so go to the right subtree where compare it with 76.
Since 63 < 76 so place it in the left child's position of 76.



Step 5: Repeat the process for the next number 63.



Step 6: Repeat the process for the next number (78)



* Insertion into the BST:

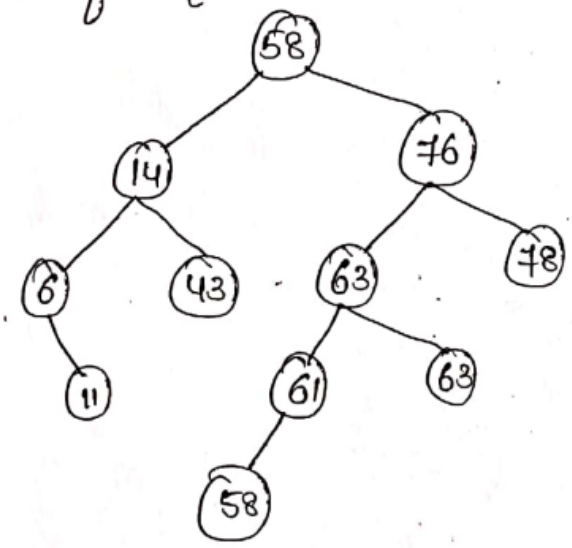
* The insertion of an element into BST follows the process as given below.

① Compare the number with the root of the BST.

② If the number is less than the root value then follow the left subtree.

③ If the number is greater than the root value then follow the right subtree.

④ Apply the same process (1, 2 & 3) to the left subtree and right subtree if required.



5. Explain the operations of skip list representation.

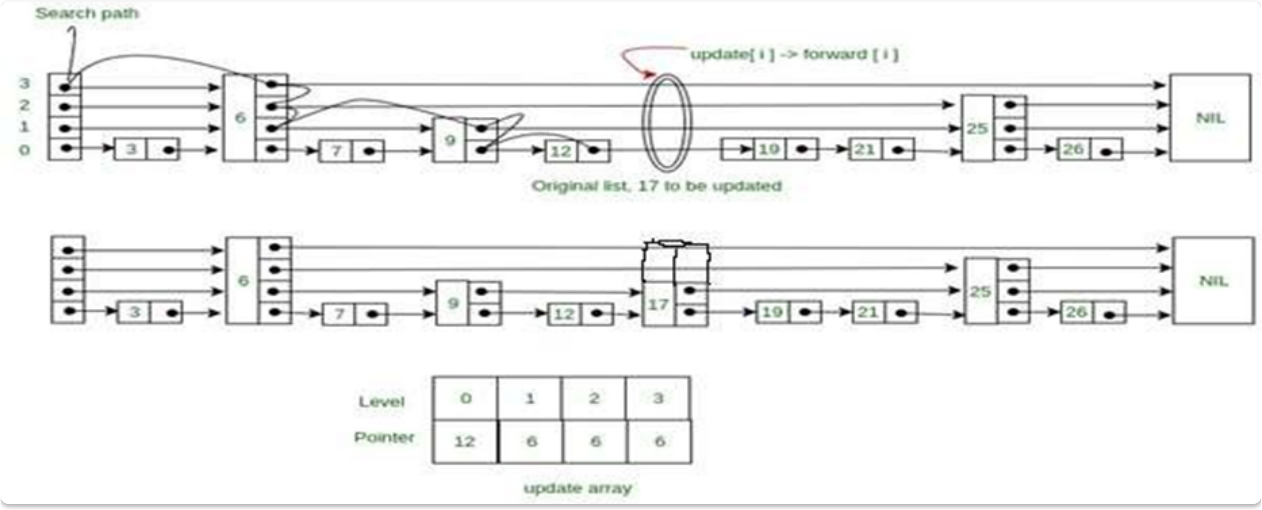
Skip List Operations

Insertion

To insert a given key, the insertion process starts from the highest level, similar to the search operation. An example of inserting the key 17 is illustrated below:

Example

Consider this example where we want to insert key 17.

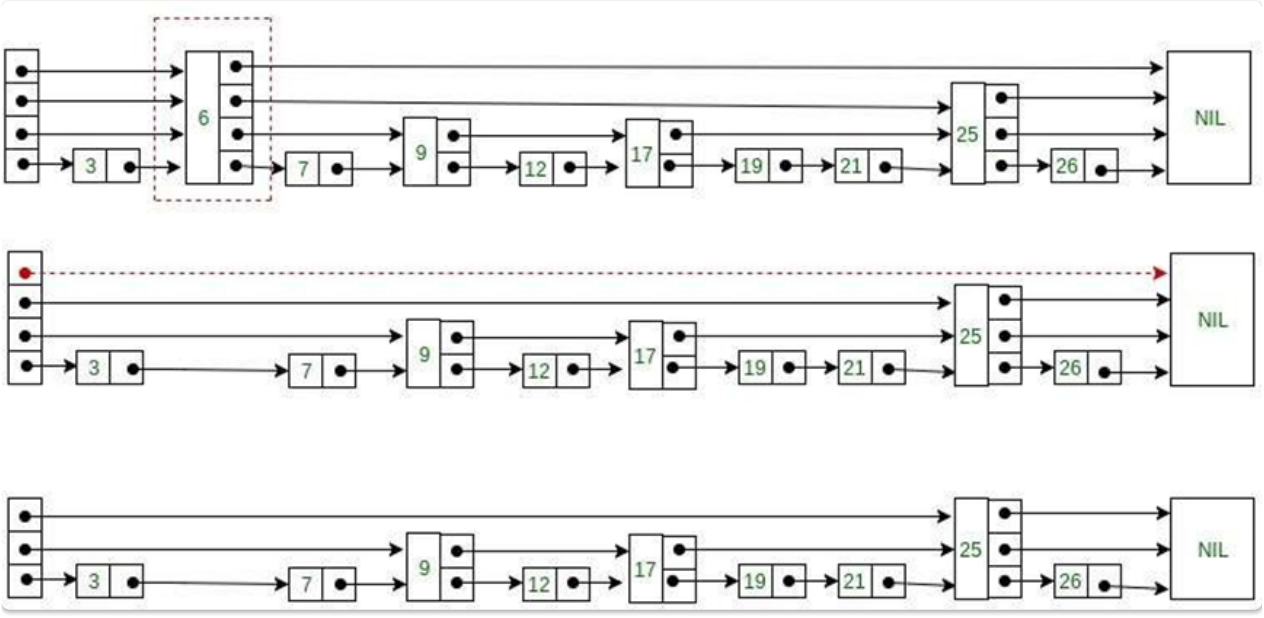


Deletion

Deletion of an element (k) involves locating the element in the skip list using the search algorithm mentioned earlier. Once the element is found, pointers are rearranged to remove the element, akin to a singly linked list. Deletion starts from the lowest level and continues until the element next to `update[i]` is not k. After deletion, there might be levels with no elements, so those levels are removed by decrementing the level of the skip list.

Example

Consider this example where we want to delete element 6. At level 3, there is no element after deleting element 6, so the level of the skip list is decremented by 1.



Advantages of Skip List

- The skip list is solid and trustworthy.
- Adding a new node is extremely quick.
- Easy to implement compared to hash tables and binary search trees.
- As the number of nodes increases, the likelihood of the worst-case scenario decreases.
- Requires only Θ(logn) time on average for all operations.
- Finding a node in the list is relatively straightforward.

Disadvantages of Skip List

- Requires more memory than a balanced tree.
- Reverse search is not permitted.
- Searching is slower than a linked list.
- Skip lists are not cache-friendly as they don't optimize the locality of reference.

6. Is linear probing and Open addressing same ? Justify your answer.

Open Addressing:

- A generic term for a method dealing with collisions in hash tables.
- When a collision occurs (two keys hash to the same index), the algorithm looks for the next available slot in the hash table.
- Includes various techniques for finding the next available slot, such as linear probing, quadratic probing, and double hashing.

Linear Probing:

- A specific form of open addressing.
- When a collision occurs, linear probing searches for the next available slot by probing consecutively in a linear fashion.

- If the slot at the hashed index is occupied, it checks the next slot, then the next, and so on, until it finds an empty slot.

Hence, Linear probing is a type of open addressing.
Not all open addressing techniques involve linear probing; others include quadratic probing, double hashing, etc.
Linear probing is one strategy within the broader category of open addressing.

7. a) What is hashing? Discuss the hash function.

b) List and explain Advantages of extendable hashing.

a)

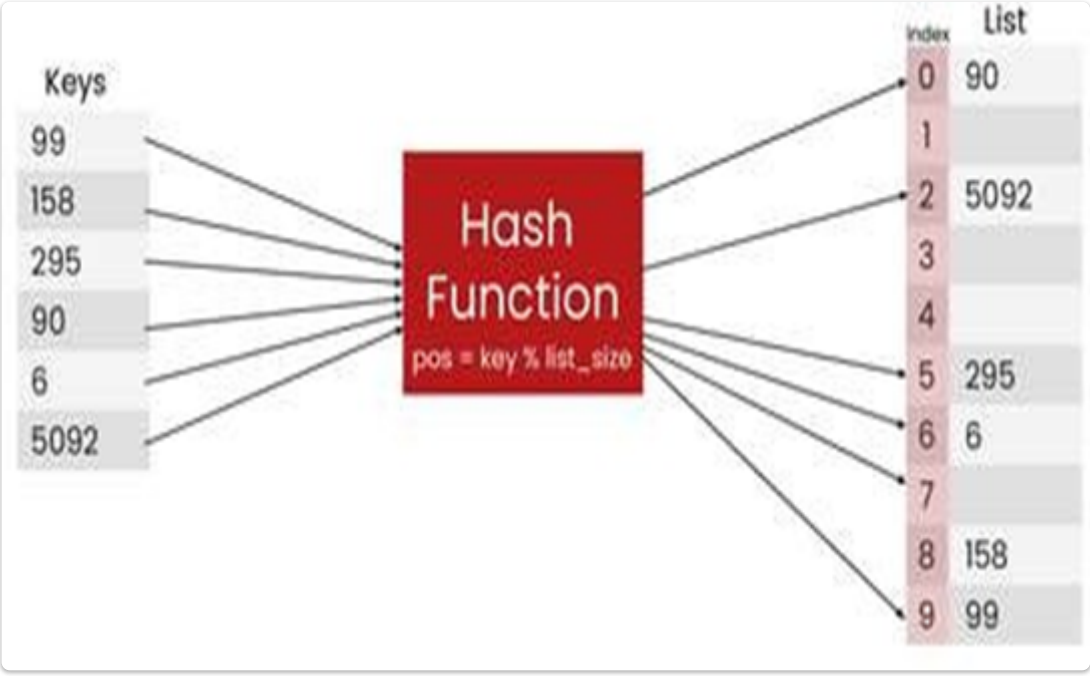
> ### Hashing
>
> Hashing involves generating a fixed-size output from an input of variable size using mathematical formulas known as hash functions. This technique is used to determine an index or location for the storage of an item in a data structure.

> ## Hash Functions
>
> The hash function establishes a mapping between a key and a value through the use of mathematical formulas. The result is known as a hash value or hash, representing the original string of characters but usually smaller.
>
> ### Types of Hash Functions
>
> There are various hash functions for numeric or alphanumeric keys:
>
> - Division Method:
>   Involves dividing the element with the size of the hash table and using the remainder as the index. Formula: H(key) = k mod m, where m is the size of the hash table.

Example

If the size of the hash table (m) is 10 and the key is {99, 158, 295, 90, 6, 5092}:

- H(99) = 99 % 10 = 9
- H(158) = 158 % 10 = 8
- H(295) = 295 % 10 = 5
- H(90) = 90 % 10 = 0
- H(6) = 6 % 10 = 6
- H(5092) = 5092 % 10 = 2



- **Mid Square Method:**
  Squares the key value and extracts the middle r digits as the hash value, where r is determined by the size of the hash table.

  ### Example

  If the hash table has 100 memory locations, and r=2, then the hash value is derived by squaring the key.
- **Digit Folding Method:**
  Breaks the key into groups of digits, and the sum of these groups contributes to the hash code. Two folding methods are used: Fold Shift and Fold Boundary.

- **Fold Shift:**
  Divides the key into parts matching the size of the required address, and the parts are added to get the address.
- **Fold Boundary:**
  Divides the key into parts, applying folding, except for the middle part.

- **Multiplication Method:**
  Uses the formula h(k) = floor(m (kA mod 1)). Both k and A are multiplied, and their fractional part is separated, then multiplied with m to get the hash value.

  Example

  If the key is multiplied by a constant value A between 0 and 1.

Properties of a Good Hash Function

- Efficiently computable.
- Uniformly distributes keys.
- Minimizes collisions.
- Low load factor (number of items in the table divided by the size of the table).

b)
advantages of extendable hashing:

1. **Dynamic Size Adjustment:**
   - Adjusts hash table size dynamically based on the number of keys.
2. **No Fixed Size Limit:**
   - Grows or shrinks dynamically, adapting to changing workloads.
3. **Efficient Space Utilization:**

- Compact representation reduces wasted space and memory overhead.

4. **Balanced Load Distribution:**
   - Maintains balanced distribution of keys, preventing excessive collisions.
5. **Minimal Reorganization:**
   - Requires only partial bucket splits or merges during resizing.
6. **Support for Incremental Growth:**
   - Allows gradual expansion or contraction in response to key changes.
7. **Low Overhead for Updates:**
   - Fast insertions and deletions with minimal bucket modifications.
8. **Query Efficiency:**
   - Balanced load and controlled bucket size contribute to uniform access time.

8. a) Write down the differences between STACK and QUEUE.

b) Distinguish between Linked list and skip list.

a) The differences between a Stack and a Queue:

Characteristic	Stack	Queue
Order of Elements	Last In, First Out (LIFO)	First In, First Out (FIFO)
Insertion Operation	1. Push: Adds an element to the top of the stack.	1. Enqueue: Adds an element to the rear of the queue.
Deletion Operation	2. Pop: Removes the element from the top of the stack.	2. Dequeue: Removes the element from the front of the queue.

Characteristic	Stack	Queue
Access	Limited to the top element.	Limited to the front element.
Additional Operations	3. Peek/Top: View the top element without removing it.	3. Front: View the front element without removing it.
Real-world Analogy	Pile of plates or books, where you can only add or remove from the top.	Line of people waiting in a queue, where the first person to join is the first to be served.
Usage	Function call management (call stack), undo mechanisms, expression evaluation.	Task scheduling, print spooling, breadth-first search algorithms.

b) distinguishing between Linked Lists and Skip Lists:

Characteristic	Linked List	Skip List
Basic Structure	- Linear collection of nodes where each node has a data field and a reference (pointer) to the next node.	- Multiple layers of linked lists, where each layer is a separate linked list. Nodes on higher levels have references to nodes on lower levels.
Search Operation	- O(n) in the worst case, as you may need to traverse the list sequentially.	- O(log n) on average. The multiple layers allow for "skipping" ahead in the structure, similar to a binary search. Uses probabilistic balancing to maintain logarithmic height.
Insertion and Deletion	- Efficient for insertions and deletions at any position, as these	- Efficient (O(log n)) for insertions and deletions. Balancing ensures logarithmic height and

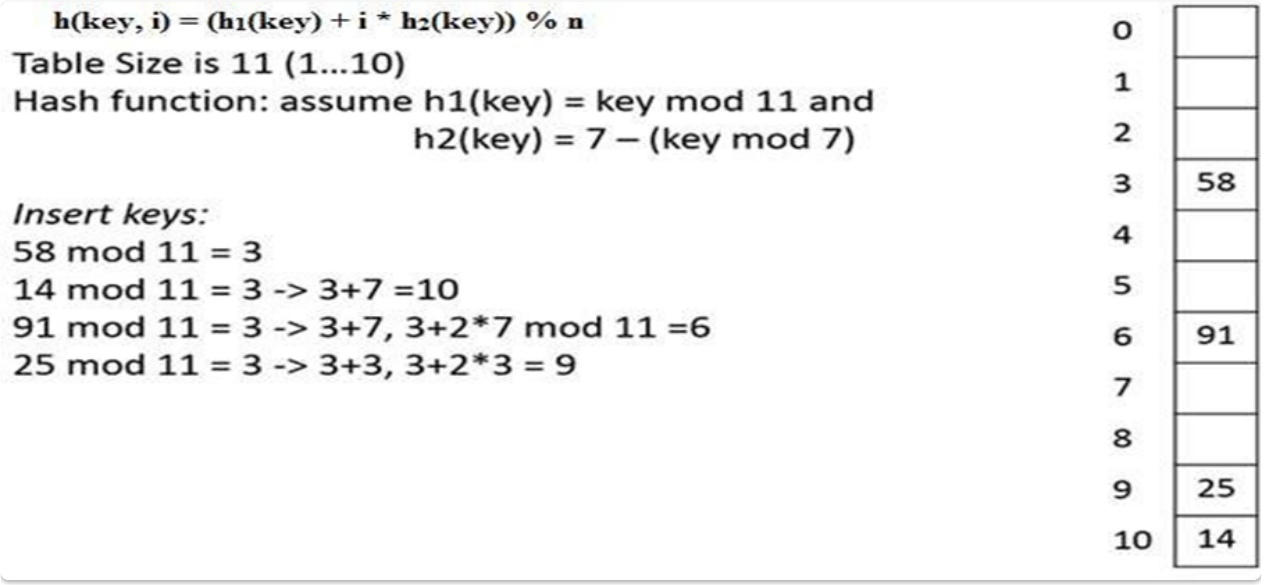
Characteristic	Linked List	Skip List
	operations involve updating pointers.	maintains overall structure integrity.
Memory Usage	- Can be memory-efficient, as nodes only need to store data and a single reference.	- May use more memory due to the additional structure of multiple layers. The trade-off is made for faster search times.

- **Balancing:** Skip lists achieve logarithmic search times through probabilistic balancing. This means that the structure maintains balance on average, but individual instances may differ.
- **Randomization:** Skip lists use a degree of randomization to determine the height of each node, contributing to their efficient performance.
- **Adaptability:** Skip lists adapt well to dynamic datasets, where insertions and deletions are frequent, maintaining efficient search times.
- **Complexity:** While skip lists offer better average-case search times compared to linked lists, they are more complex to implement and may require additional bookkeeping for maintaining their structure.

9. Explain double hashing? Explain the insertion operation with example.

> ### 2.c) Double Hashing
>
> Double hashing uses two hash functions. The first hash function determines the initial location, and the second one is used if a collision occurs.
>
> #### Example
>
> Insert keys {27, 43, 692, 72} into a hash table of size 7 with first hash function h1(k) = k mod 7 and second hash function h2(k) = 1 + (k

mod 5). Detailed steps and examples provided for each key.



10. a) Explain about load factor in hashing.

b) What is collision? Explain in detail about collision.

a)

Load Factor in Hashing

The load factor of a hash table is a crucial metric that indicates the relationship between the number of items stored in the hash table and its size. It is calculated by dividing the number of items in the hash table by the total size of the hash table. The load factor becomes particularly significant when considering whether to rehash a previous hash function or when adding more elements to an existing hash table.

Importance of Load Factor:

1. Efficiency of the Hash Function
2. Rehashing Decision
3. Dynamic Sizing

Load Factor Formula:

The load factor (LF) is calculated using the formula:

$$\text{Load Factor (LF)} = \frac{\text{Number of Items}}{\text{Size of Hash Table}}$$

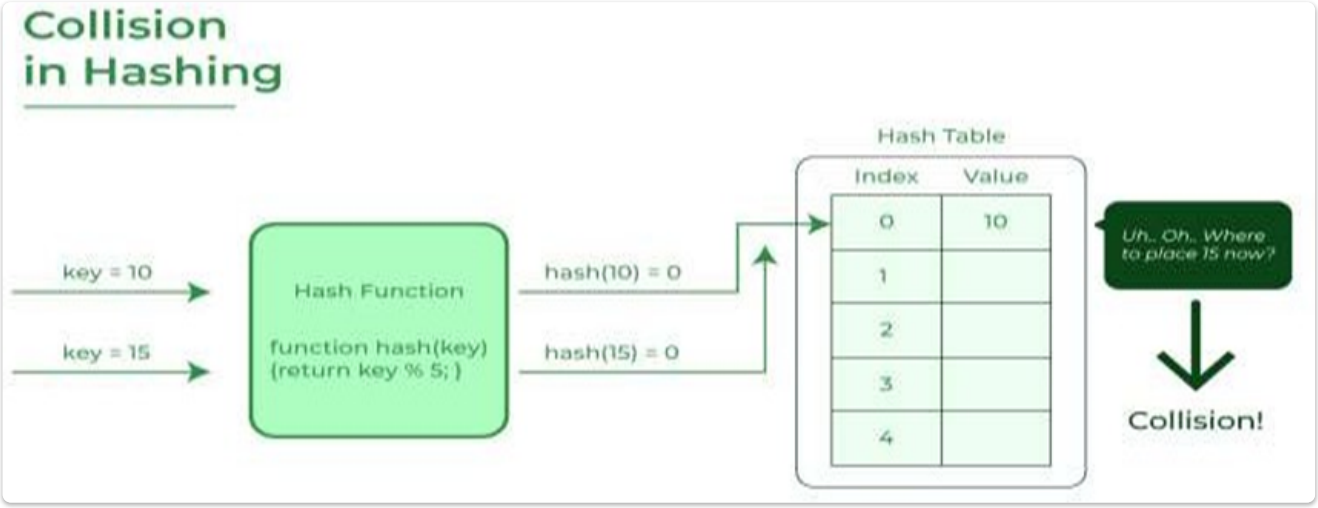
Where:

- **Number of Items:** The current count of elements stored in the hash table.
- **Size of Hash Table:** The total capacity or size of the hash table.
- **Low Load Factor (LF < 1):**
  - Indicates that the hash table has ample space relative to the number of items.
  - Hash collisions may be less frequent, resulting in better performance.
- **High Load Factor (LF > 1):**
  - Suggests that the hash table is becoming crowded.
  - Increased likelihood of hash collisions, potentially leading to performance degradation.
  - May trigger rehashing to maintain efficiency.

b)

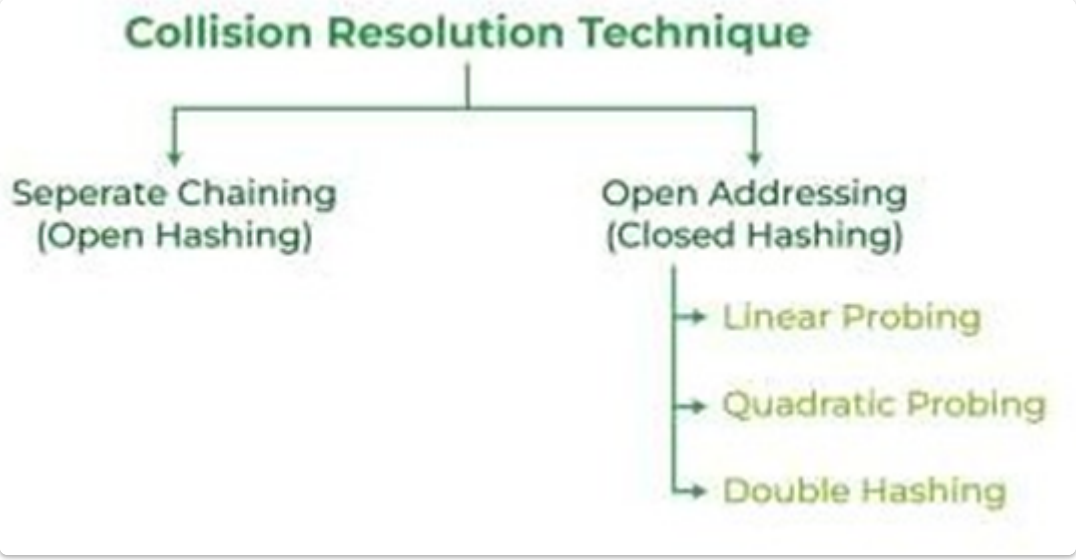
Collision

In hashing, the process of generating a small number for a large key may lead to the possibility of two keys producing the same value. This situation is known as a collision, and various techniques are used to handle it.



Collision Resolution

There are two main methods to handle collisions:



1. **Chaining Collisions:**
   - In chaining, each slot in the hash table is the head of a linked list. When a collision occurs, the collided keys are added to the linked list at that slot.
   - Chaining provides a simple and effective way to handle collisions. It allows multiple keys with the same hash value to coexist at the same index by forming a linked list of collided keys.
   - **Advantages:**
     - No limit on the number of elements with the same hash value.
     - Simple to implement.
   - **Disadvantages:**
     - Memory overhead due to the storage of linked lists.
     - Cache inefficiency as elements are scattered in memory.
2. **Open Addressing Collisions:**
   - In open addressing, when a collision occurs, the algorithm looks for the next available slot in the hash table to place the collided key.

- Various techniques exist within open addressing, including linear probing, quadratic probing, and double hashing. These methods determine the sequence in which the algorithm searches for the next available slot.
- **Advantages:**
  - No additional memory overhead for linked lists.
  - Better cache performance as elements are stored contiguously.
- **Disadvantages:**
  - Can lead to clustering, where consecutive slots are occupied, increasing the likelihood of more collisions.
  - Difficulty in finding an efficient probing sequence.

Collision Resolution:

- **Separate Chaining:** In chaining collisions, the hash table slots contain linked lists to handle multiple elements with the same hash value.
- **Linear Probing:** In open addressing collisions, the algorithm looks for the next available slot in a linear sequence.
- **Quadratic Probing:** The algorithm uses a quadratic function to determine the next slot in open addressing collisions.
- **Double Hashing:** Another hash function determines the step size for probing in open addressing.