Date:
Page:

2080

Section 'A'

1) Explain queue as an ADT. write a program to implement linear queue. Compare linear queue with circular queue.

Ans Queue as an ADT

→ A queue is abstract data type (ADT) that represents a linear data structure designed for storing & managing a collection of elements. It follows the first-in-first (FIFO) principle., which means that the first element added to the queue is the first one to be removed. Queues are commonly used in computer science and software development for various applications, including scheduling tasks, handling requests and managing data in a way that ensures fairness & order.

⇒ The primary operations supported by a queue ADT are

1) Enqueue: Add an elements to the back (rear) of the queue.
2) Dequeue: Remove and return the element from the front of the queue.
3) Peek/Front: Retrieve the element at the front of the queue without removing it.

Date:
Page:

4) IsEmpty: check if the queue is empty.
5) size: Get the number of elements in the queue.

* Program to implement linear queue.

```
#include <stdio.h>
#include <stdlib.h>
#define max-size 10

// structure to represent the linear queue
struct Queue {
int items [max-size];
int front;
int rear;
};
// Function to initialize the queue
void initialize (struct Queue *q) {
q -> front = -1;
q -> rear = -1;
}
// Function to check if the queue is empty.
int isEmpty (struct Queue *q)
{
return (q -> front == -1);
}
```

Date:
Page:

// Function to add an element to the rear of the queue.

```
void enqueue (struct Queue *q, int value)
{
  if (isFull(q)) {
printf("queue is full. cannot enqueue.\n");
    return;
  }
if (isEmpty(q))
  {
q->front = 0;
  }
q->rear++;
q->items[q->rear] = value;
printf("Enqueued: %d\n", value);
}
```

// Function to remove and return an element from the front of the queue.

```
int dequeue (struct Queue *q)
{
int item;
if (isEmpty(q)) {
printf("Queue is empty. Cannot dequeue!");
  return -1;
}
```

Date:

Page:

```
item = q -> item [q -> front ];
  IF (q -> front == q -> rear) {
   q -> front = -1;
    }
    else {
   q -> front ++;
    }
     return item;
      }
     Int main () {
   struct Queue q;
    Initialize (&q);
   enqueue (&q, 1);
   Enqueue (&q, 2);
   enqueue (&q, 3);

PrintF ("Dequeued : %d \n", dequeue (&q));
printF ("Dequeued : %d \n", dequeue (&q));

 enqueue (&q, 4);
printF ("Dequeued : %d \n", dequeue (&q));
printF ("Dequeued : %d \n", dequeue (&q));

 return 0;
  }
```

Linear queue	circular queue.
→ Linear queues have a fixed capacity and can become inefficient when the front of the queue has dequeued elements, leading to unused space at the front.	circular queues address the inefficiency of linear queu by utilizing a circular arra structure, where the front and rear "wrap around" when they reach the end of the array.
→ Enqueue and dequeue oper -ations are straight forward but may require shifting element when dequeuing	circular queues elimina the need to shift ele -nts when the front ele -ent is dequeued which makes them more effici
→ Linear queues are simple to implement but may not be efficient for scenarios with frequent enqueue and dequeue operations	circular queues are su -able for scenarios where elements are frequently added and removed such as scheduling tasks in an operating system or managing data buffers.

Date:
Page:

2) Define hash table and hash function. What is collision in hashing? Explain linear probing and quardratic probing with suitable example.

* Hash Table:

A hash table is a data structure that stores key-value pairs in a way that allows fore efficient data retrieval. It uses a tech-nique called hashing, which applies a hash function to key to compute an index into a array of buckets or slots where the corresponding value can be found.

↳ Hash table are used to implement data structure like dictionaries, sets, and associative arrays, providing fast access to data by optimizing the search time.

* Hash Function:

⇒ A hash function is a mathematical function that takes an input (or "key") & returns a fixed sized string of characters, typically a numeric index.

↳ The purpose of a hash function is to convert the input data (e.g. a string or no). into a unique or nearly-unique.

integer value that can be used as an index to access data structure like hash table.

→ The ideal hash function should be deterministic (produce the same output for the same input) efficient to compute, and evenly distribute value across the available slots in the hash table.

* Collision in Hashing:

Collision in hashing occurs when two different key hash to the same index in the hash table. Since hash functions aim to map keys to unique locations, collisions represent a challenges. There are several techniques for handling collisions and two of them are: linear probing & quadratic probing.

* Linear probing:

Linear probing is a collision resolution technique where, when a collision occurs, the algorithm searches for the next available slot in a linear manner (incrementing by a constant interval) until an empty slot is found. This ensures that keys with collisions are stored in consecutive locations

Date:

Page:

Suppose you have a hash table with 10 slots, and the hash function is (key % 10, where modulo operator). You want to insert the keys 25, 35, 45 and 55 into the hash table.

→ key 25 hashes to index 5

→ key 35 hashes to index 5 (collision)

→ Linear probing looks for the next available slot, which is index 6.

→ key 45 hashes to index 5 (collision) but is stored at index 7.

→ key 55 hashes to index 5 (collision) but is stored at index 8.

The Final state of the hash table might look like this.

[25, 35, 45, 55, _, _, _, _, _, _]

* Quadratic probing:

→ Quadratic probing is another collision resolution technique that, when a collision occurs, searches for the next available slot in a quadratic manner (incrementing by quadratic formula) until an empty slot is found. This help

Date:
Page:

avoid clustering issues that can occure with linear probing.

→ Quadratic probing reduces clustering because it explores slots further away from the orginal hash positions. It uses a probing sequence (c1 * i^2 + c2 * i) to find the next available slots)

suppose we have a hash table with 10 slots, and the hash function is. (key % 10).

- key 25 hashes to index 5.
- key 35 hashes to index 5 (collision)
  Quadratic probing with c1 = 1 and c2 = 1 looks for the next available slots:
  (1 * 2^2 + 1 * 1) = 2 slots away from index 5, so it checks index 7, it's empty, so it stores 35 there.
- key 45 hashes to index 5 (collision).
  quadratic probing with c1 = 1 and c2 = 1 continues searching for the next available slots (1 * 2^2 + 1 * 2) = 10 slots away from index 5, so, it checks index 5 (circularly). It's occupied;
  (1 * 3^2 + 1 * 3) = 18 slot away from index 5, so it checks index 8, it's empty, so it stores 45 there.

Date:

Page:

drawback: key 55 hashes to index 5 (collision) quadratic probing with c1=1 and c2=1 continues searching for the next available slots:

(1*4^2 + 1*4) = 32 slots away from index 5, so, it checks index 2, it's empty. so, it stores 55 there.

* The final state of the hash table after quadratic probing might look like this.

[25, _, 55, 35, _, _, 45, _]

3) Explain AVL tree with example. Also, explain balancing algorithm for this tree.

→ AVL Tree:

An AVL tree (Adelson-Velsky and Landis tree) is a self-balancing binary search tree. It is named after the inventors of the tree structure, G.M. Adelson, Velsky & E.M. Landis. AVL trees are designed to maintain a balance between the left & right subtrees of node., ensuring that the tree remains approximately balanced.

Date
Page

This balance property ensures that the height of the tree is logarithmic, resulting in efficient search, insert, and delete operations with time complexity of O(log n).

In AVL tree, the balance Factor of each node, which is the difference in height between the left and right subtrees, must be within the range of -1, 0 or 1. If the balance Factor of any node exceeds this range, the tree is considered unbalanced and rotation operation are applied to restore balance.

Balancing Algorithm For AVL tree.

The primary balancing operation in an AVL tree is rotation. There are four types of rotation used to balance the tree.

1) Left Rotation (LL Rotation)

It occurs when the balance Factor of a node is greater than 1 (i.e left-heavy) and the left child of that node is also left-heavy. To balance the tree, a single left rotation is performed. This help move the right child of the unbalanced node up.

Date: _____

Page: _____

Right Rotation (RR Rotation).

It occurs when the balance factor of a node is less than -1 (i.e right-heavy) and the right child of that node is also right-heavy. A single right rotation is performed to balance the tree, moving the left child of the unbalanced node up.

Left-Right Rotation (LR Rotation)

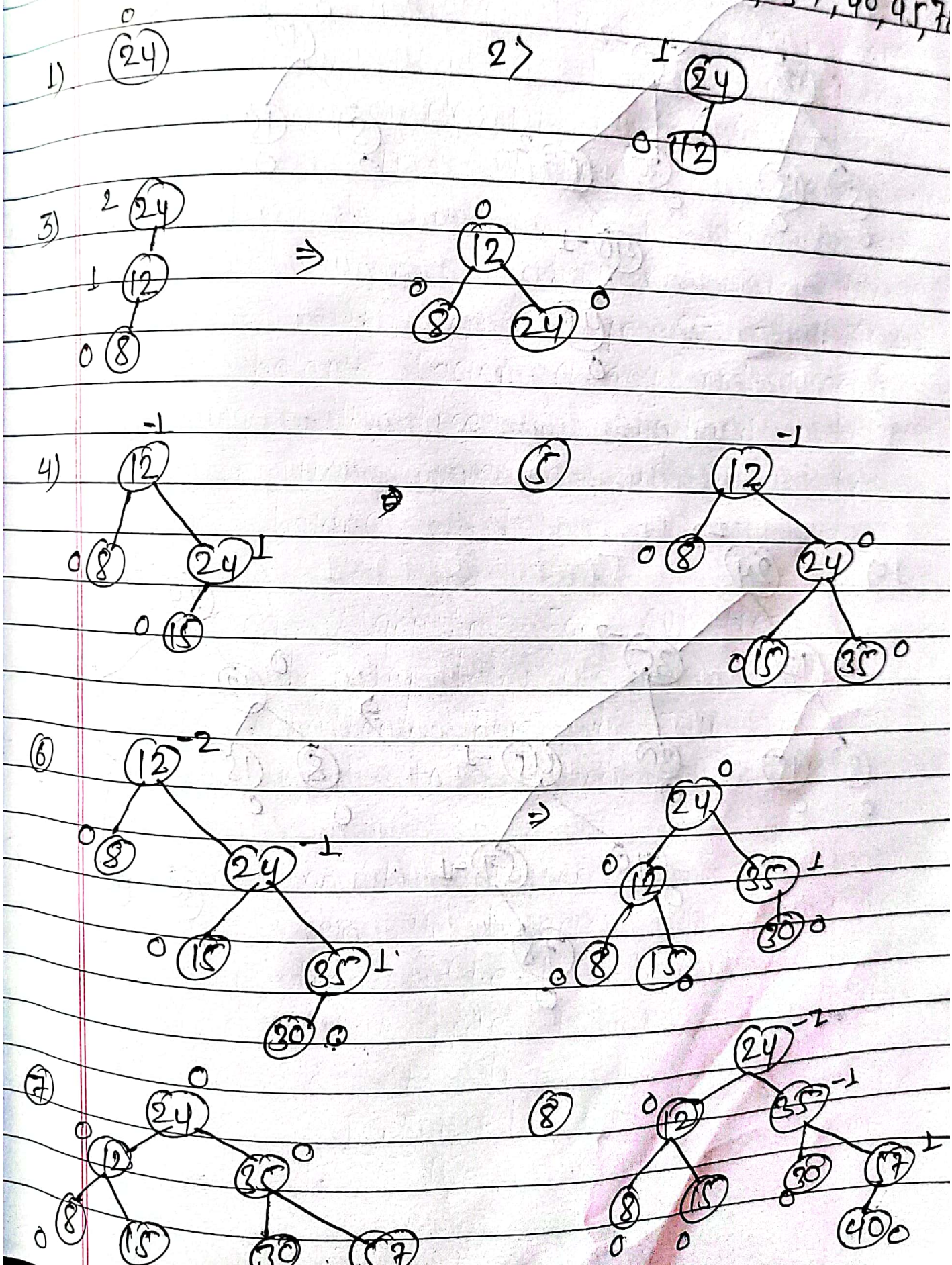
It occurs when the balance factor of a node is greater than 1 (left-heavy) and the left child is right-heavy. It involves performing a left rotation on the left child followed by a right rotation on the unbalanced node. This combination of rotations balances the tree.

Right-Left Rotation (RL rotation)

It occurs when the balance factor of a node is less than -1 (right-heavy) and the right child is left-heavy. It involves performing a right rotation on the right child followed by a left rotation on the unbalanced node. This combination of rotations balances the tree.

Date:
Page:

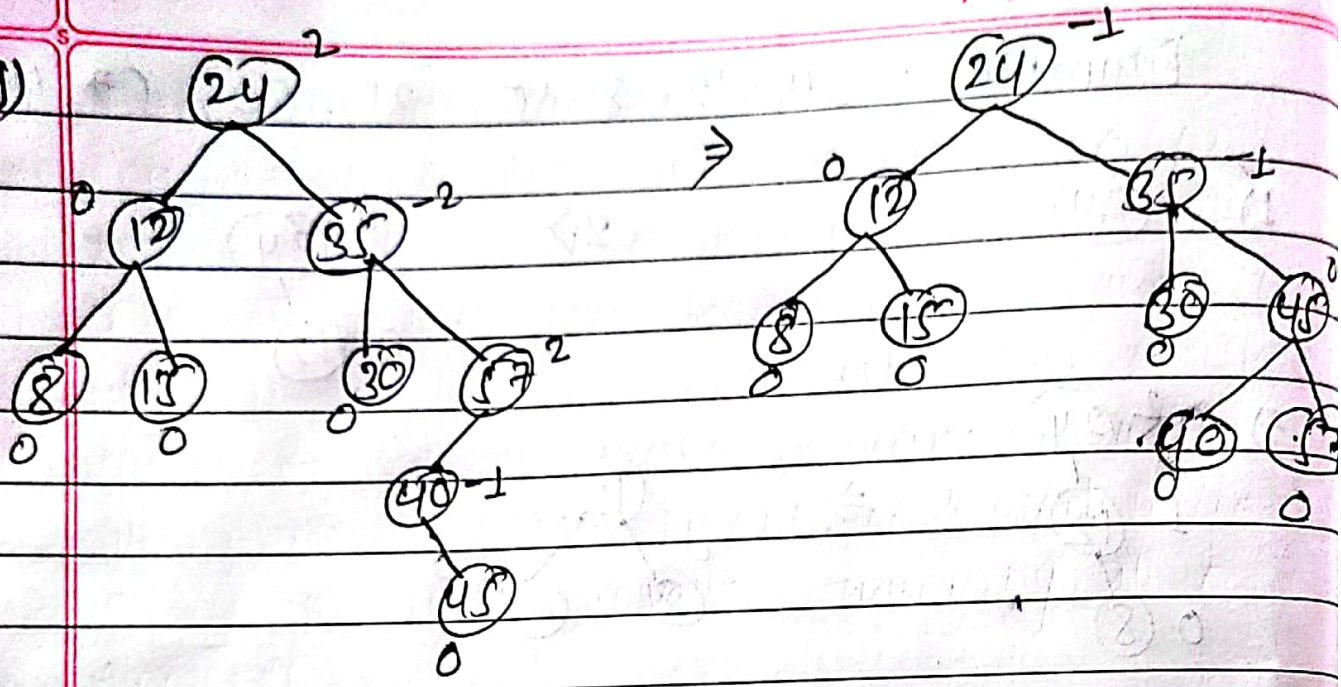
Example : 24, 12, 8, 15, 35, 30, 57, 40, 45, 7



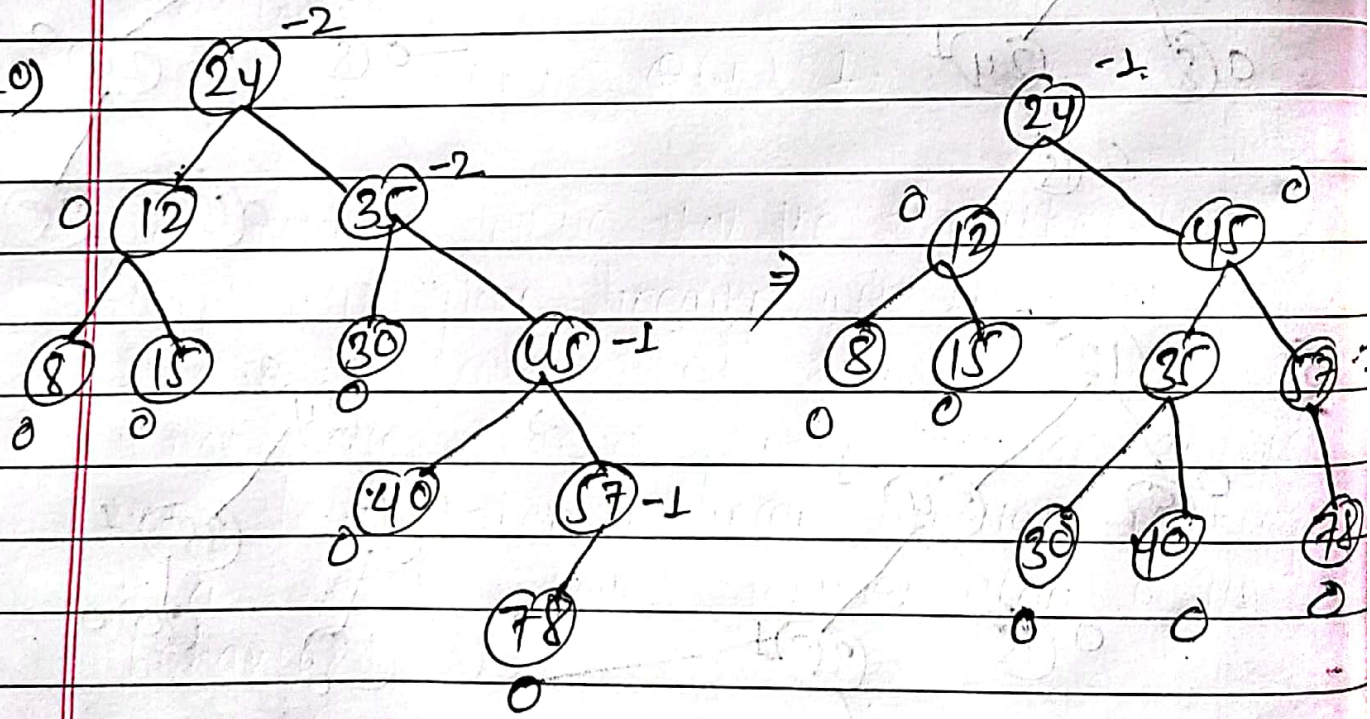
Date: _____

Page: _____

9)



10)



Date:
Page:

Section 'B'

4) What is asymptotic Analysis? Explain ~~tete~~ theta notation with example:

→ Asymptotic analysis is a method in computer science & mathematics to analyze the efficiency or performance of algorithms and data structures in terms of their growth rates as the input size becomes arbitrarily large. It provides a high-level understanding of how the time and space complexity of algorithms scales, with input size allowing us to make meaningful comparisons between different algorithms without getting bogged down in the details of constant factors.

Example of Theta notations.

→ Let's consider a simple example to demonstrate Theta notation.

Suppose we have an algorithm that performs a linear search in array of n elements. The time complexity of a linear search is O(n) because, in the worst case, it may need to transverse the entire array.

Date:

Page:

In this case, we can also say that the time complexity of the linear search is $\Theta(n)$ because. It has both an upper and lower bound of n. The best-case scenario occurs when the elements is found in the first position (constant time), & the worst-case scenario is when the element is not present or located in the last position, which involves traversing all n elements.

So, for the linear search algorithm, we can express its time complexity using Theta notation as:

$$T(n) = \Theta(n) = 1$$

This notation tell us that the algorithms time complexity grows linearly with the input size. It also indicates that algorithm's performance is not worse than a linear function and is not better than a linear function, making it a tight bound representation of its behavior.

Date
Page

5) Explain push and pop operations of stack. what are different applications of stack.?

→ A stack is a linear data structure that follows the Last-In-First out (LIFO) principle, Meaning the last item added to the stack is first one to be removed. It supports two primary operation PUSH and pop.

1) Push:

... The push operation is used to add an element onto the top of the stack when we push element onto stack, it become the ~~top~~ new top element. This operation increases the stack's size by one. In programming, we can typically implement the push operation using a function like Push (element).

2) Pop:

The pop operation is used to remove and retrieve the top element from the stack. After a pop operation, the stack's size decreases by one and the next element in the stack becomes the new top element. In most programming languages we can implement the pop operation using a function like pop ().

Date: _____

Page: _____

Application of stacks are:

1) Function call management:
Stack are used to manage function call in programming language. Each function call is pushed onto the stack, and when a function returns, it is popped of the stack, allowing the program to return to the previous state.

2) Expression Evaluation:
→ Stacks are used to evaluate mathematical expression, such as infix expression, by converting them to postfix (or prefix) notation and then using a stack to perform the evaluation.

3) Backtracking:
Stack are used in algorithms like depth-first search (DFS) to keep track the visited nodes or states, allowing the algorithm to backtrack when necessary.

4) Undo Mechanisms:
→ Many applications such as text editors, & graphic design software, use stacks to implement undo & redo functionality. Each change is pushed onto the undo stack,

Date:
Page:

and undo operations pop the changes from stack

5) Syntax parsing:
Stack are used in parsing and interpreting programming languages to manages the parsing process. They can help with checking the correctness of nested constructs like parentheses and braces.

6) Memory management
7) Expression conversion:
8) Task management
9) History management

(6) Explain tail recursion with example compare recursion with iteration.

→ Tail Recursion:
Tail recursion is a specific from of recursion where the recursive call is the last operation in a function before it returns its result. In other words, in a tail-recursive function, there is no additional computation or processing performed after the recursive call.

This has an important implication in terms of optimization because certain programming languages & compilers can optimize tail

Date: _____

Page: _____

- recursive calls by reusing the current function's stack frame, which eliminates the risk of stack overflow due to excessive recursion.

```
def Factorial-tail-recursive (n, result = 1):
    if n == 0:
        return result
    else
    return Factorial-tail-recursive (n-1, n * result)
```

Recursion	Iteration.
→ Recursion is a technique where a function calls it self to solve a problem	Iteration is a technique where a loop is used to repeatedly execute a block of code to solve a problem
→ It is often more concise & easier to understand for problems that can be naturally divided into smaller, similar subproblems.	It can be more efficient for some problems especially those that don't naturally break down into subproblems.
→ Some problems are naturally expressed in recursive terms, making recursion a more suitable choice.	Iteration is generally more space-efficient as does not rely on the call stack.
→ Recursion may lead to stack overflow errors for deep recursion if not optimized (except for tail recursion.	Most programming languages provide good support for iteration with loops, making it easy to implement.
→ Recursion can have higher memory overhead due to the call stack.	Iteration often used when performance is primary concern as it avoids the overhead of function calls

Date:

Page:

7) Trace selection sort algorithm with array of numbers 2, 81, 6, 45, 11, 21, 23, 41 and 11.

Solution:

→ Step1 : [2, 81, 6, 45, 11, 21, 23, 41, 11]
Orginal array.
→ The minimum value in the unsorted portion is 2, so we swap it with the first element

Step2: [2, 81, 6, 45, 11, 21, 23, 41, 11] (2 is in its Final sorted position)
→ Now the remaining unsorted portion is [81, 6, 45, 11, 21, 23, 41, 11].
→ The minimum value is this portion is 6, so we swap it with the second element.

Step3: [2, 6, 81, 45, 11, 21, 23, 41, 11] (2 and 6 are in their Final sorted partitions)
→ The remaining unsorted portion is [81, 45, 11, 21, 23, 41, 11].
→ The minimum value is 11, so we swap it with the third element.

Step4: [2, 6, 11, 45, 81, 21, 23, 41, 11) (2, 6 and 11 are in their final sorted positions)

→ The remaining unsorted portion is [81, 45, 21, 23, 41, 11]

→ The minimum value is 11 (again), so we swap it with the Fourth element.

Step5: [2, 6, 11, 81, 21, 23, 41, 45] (2, 6, 11, and 11 are in their Final sorted positions).

→ The remaining unsorted portion is [81, 21, 23, 41, 45)

→ The minimum value in this portion is 21, so we swap it with the fifth element.

Step6: [2, 6, 11, 11, 21, 81, 23, 41, 45] (2, 6, 11, 11 and 21 are in their Final sorted positions)

→ The remaining unsorted portion is [81, 23, 41, 45]

→ The minimum value is 23, so we swap it with sixth element.

Step7: [2, 6, 11, 11, 21, 23, 81, 41, 45] (2, 6, 11, 11, 21, and 23 are in their Final sorted positions)

→ The remaining unsorted portion is [81, 41, 45)

→ The minimum value is 41, so we swap it with the seventh element.

Date: _____

Page: _____

Step 8: [2, 6, 11, 11, 21, 23, 41, 81, 45] (2, 6, 11, 11, 21, 23, and 41 are in their final sorted position)

→ The remaining unsorted portion is [81, 45]

→ The minimum value is 45, so we swap it with the eighth element.

Step 9: [2, 6, 11, 11, 21, 23, 41, 45, 81] All elements are in their final sorted positions)

The selection sort algorithm is complete, and the array is now sorted in ascending order.

Date:
Page:

8) Explain binary search with an example. What is the time complexity of binary search?

→ Binary search:

It is a fast and efficient searching algorithm that is used to search for a specific element in a sorted array or list. It works by repeatedly dividing the search interval in half which reduces the search space and narrows down the possible location of the target element until it's found.

Here, How binary search works

1) Initalize two pointers, left and right, to the first and last element of the sorted array respectively.
2) Calculate the middle index, mid, as the average of left and right.
3. Compare the element at middle index, arr [mid], with the target element
   → If arr [mid] is equal to the target, the element is found, and its index is mid.
   → If arr [mid] is greater than the target, update right to mid - 1, effectively eliminating the right half of the search interval.

Date:

Page:

If arr [mid] is less than the target, update left to mid + 1, effectively eliminating the left half of the search interval.

Repeat step 2 and 3 untill the element is found or the search interval is empty (i.e. left > right).

Example

Input : Array [1, 2, 5, 7, 9, 11, 13, 15]

Target:

```
#include <stdio.h>
int binarysearch (int arr[], int size, int target)
{
    int left = 0;
    int right = size - 1;
    while (left <= right)
    {
    int mid = left + (right - left) / 2;

    IFF (arr [mid] == target)
    {
    return mid;
    }
    else IF (arr [mid] < target) {
    left = mid + 1;
```

Date:
Page:

```
}
else {
right = mid - 1;
    }
    }
return -1;
    }
int main () {
int arr [] = {1, 3, 5, 7, 9, 11, 15};
   int target = 7;
int size = size of (arr) / size of (arr [0]);
 int result = binary_search (arr, size, target);
  if (result != -1)
      {
printf ("Element %d found at index %d \n",
        target, result);
    }
    else {
       printf (" Element %d not found \n",
            target);
       }
       return 0;
       }
```

Output:

Elemet 7 found at index 3.

Date:
Page:

Time complexity:
→ O (log n) because the size of the search range is halved with each step.

9) Write Dijkstra's algorithm to find shortest path between any two vertices of graph

→ Dijstra's algorithm is used to find the shortest path in a weighted graph from a source vertex to all other vertices. Here are the step to execute Dijkstra's algorithm.

Algorithm.

1) Create graph with vertices and edges. each edge should have a weight representing the cost or distance between two vertices.

2. Select a source vertex From which we want to find the shortest path to all other vertices

3. Create a set of unvisited vertices and initialize the distance to the source vertex as 0 & all other vertices as infinity.

Date:
Page:

4) Create ~~a set of unvisited vertices and~~ a data structure (e.g. priority queue or a min-heap) to store vertices and their tentative distance from the source.

5) While there are unvisited vertices:
   a. Select the unvisited vertex with the smallest tentative distance.
   b → For the selected vertex, calculate the tentative distance to all of its unvisited neighbors.
   c → Update the tentative distances for the neighbors if a shorter path is found.
   d → Mark the selected vertex as visited to avoid revisiting it.

6) When all vertices have been visited or if the target vertex (if specific) has been reached, the algorithm terminates.

7) If we want to find the actual shortest path from the source to a specific target vertex, backtrack from the target vertex to the source vertex using the recorded shortest path.

Date:

Page:

8) The algorithm provides the shortest distances From the source. Vertex to all other vertices of the graph. Additionally, if you perform backtracking, you can obtain the actual shortest path From the source to a specific target vertex.

10) Write a program to Implement Insertion sort.

Program

```
#include <stdio.h>
void insertionSort(int arr[], int
{
  int i, j, key;
 for (i=1; i<n; i++)
     {
       key = arr[i];
       j= i-1;
// move elements of arr[0...i-1] that are
greater than key,
// to one position ahead of their
current position.
```

Date:

Page:

```
while (j<=0 && arr[j]>key) {
    arr[j+1]= arr[j];
    j = j-1
    }
    arr[j+1] = key;
    }
    }
    int main() {
    int arr[] = {12, 11, 13, 05, 6};
int n= sizeof(arr) / size-of(arr[0]);
    printf(" Original array:");
    for (int i= 0; i<n; i++)
        {
        printf ("%d ", arr[i]);
        }
        printf ("\n");
    insertion sort (arr, n);
    printf ("sorted Array :");
    For (int i=0; i<n; i++)
        {
        printf ("%d ", arr[i]);
        }
        printf ("\n");
        return 0;
        }
```

Date:

Page:

How can you use linked list to implement stack ? Explain

We can use a linked list to implement a stack by maintaining a singly linked list where each node represents an element in the stack. The key idea is to perform all stack operations (push and pop) at the front of the linked list, which ensures that the last element inserted becomes the first to be removed, following the Last-In-First-Out (LIFO) principle of a stack.

Push operation

To push an element onto the stack, create a new node and insert it at the beginning of the linked list.

update the top pointer to point to the newly added node.

```
Void push (struct stack * stack, int data)
{
struct Node* newNode = (struct Node *) malloc (size_of (struct Node));
NewNode -> data = data;
```

Date:

Page:

New Node → ~~Next~~ next = stack → top

stack → top = ~~New~~ new Node;

}

Pop operation:

→ To pop operation an element From the stack, remove the node at the top of the linked list

→ update the top pointer to point to the next node

→ Return the data from the removed node.

→ check For stack underFlow (when the stack is empty) before popping.

```
int pop (struct stack * stack)
{
    if (stack → top == NULL) {
    printf (" stack underFlow \n");
    exit (1);    // Handle underFlow as needed
    }
struct Node * poppedNode = stack → top;
int data = poppedNode → data;
stack → top = poppedNode → ~~dex~~ next;
free (poppedNode);
return data;
}
```

Date:
Page:

Other operations:

We can also implement additional stack operations like checking if the stack is empty and retrieving the top element without removing it.

```
int isEmpty (struct stack* stack)
{
return (stack -> top == NULL);
}
int peek (struct stack *. stack) {
if (stack -> top == NULL) {
printf ("stack is empty .\n");
exit (1);
}
return stack -> top -> data;
}
```

Initialization:

Before using the stack, initialize it by creating an empty linked list (setting top to NULL) & allocating memory for the stack structure.

```
struct stack * stack = (struct stack *)
Malloc (size of (struct stack));
stack -> top = NULL;
```

Date:
Page:

12) Write short note on.
i) Abstract data type.
ii) Circular linked list

i) solution

→ An abstract data type is a high-level description of a data structure that defines a set of operations and the behavior of those operations without specifying the implementation details.
→ It focuses on what operations can be performed on the data structure and what constraints or rule should be followed, rather than how those operation are implement.
→ Example ; lists, stacks, queues, trees, and graph.

ii) Circular Linked List:
A circular linked list is a variation of a singly linked list where the last node in the list points back to the first node, creating a closed loop. In a traditional singly linked list, the ~~closed loop~~ last node's "next" pointer is typically set to NULL, indicating the end of the list.