Q1. a. Write a C Program to implement Circular queue using arrays

```
#include <stdio.h>
#define SIZE 5  // Define the maximum size of the queue
// Declare the circular queue and related variables
int queue[SIZE];
int front = -1, rear = -1;
// Function to check if the queue is full
int isFull() {
    return (front == 0 && rear == SIZE - 1) || (rear == front - 1);
}
// Function to check if the queue is empty
int isEmpty() {
    return front == -1;
}
// Function to enqueue an element into the circular queue
void enqueue(int element) {
    if (isFull()) {
        printf("Queue is full! Cannot insert %d\n", element);
        return;
    }
    if (front == -1) { // First element to be inserted
        front = rear = 0;
    } else if (rear == SIZE - 1 && front != 0) { // Wrap around
        rear = 0;
    } else { // Normal case
        rear++;
    }
    queue[rear] = element;
    printf("Inserted %d into the queue\n", element);
}
// Function to dequeue an element from the circular queue
int dequeue() {
    if (isEmpty()) {
        printf("Queue is empty! Cannot dequeue\n");
        return -1;
    }
    int element = queue[front];
    if (front == rear) { // Queue becomes empty
        front = rear = -1;
    } else if (front == SIZE - 1) { // Wrap around
        front = 0;
    } else { // Normal case
        front++;
    }
    return element;
}
```

```
// Function to display the elements of the queue
void display() {
    if (isEmpty()) {
        printf("Queue is empty!\n");
        return;
    }
    printf("Queue elements: ");
    if (rear >= front) {
        for (int i = front; i <= rear; i++)
            printf("%d ", queue[i]);
    } else {
        for (int i = front; i < SIZE; i++)
            printf("%d ", queue[i]);
        for (int i = 0; i <= rear; i++)
            printf("%d ", queue[i]);
    }
    printf("\n");
}
// Main function to demonstrate the circular queue
int main() {
    int choice, value;
    while (1) {
        printf("\n--- Circular Queue Operations ---\n");
        printf("1. Enqueue\n");
        printf("2. Dequeue\n");
        printf("3. Display\n");
        printf("4. Exit\n");
        printf("Enter your choice: ");
        scanf("%d", &choice);
        switch (choice) {
            case 1:
                printf("Enter the value to enqueue: ");
                scanf("%d", &value);
                enqueue(value);
                break;
            case 2:
                value = dequeue();
                if (value != -1)
                    printf("Dequeued element: %d\n", value);
                break;
            case 3:
                display();
                break;
            case 4:
                printf("Exiting...\n");
                return 0;
            default:
                printf("Invalid choice! Please try again.\n");
```

```
    }
  }
}
```

==========

Q1. b. Discuss the properties of AVL trees, including their height balance condition. Expl;ain how rotations (single and double) are used to maintain balance during insertions and deletions.

Properties of AVL Trees:

1. **Binary Search Tree Property:**
   - The left subtree contains values less than the root.
   - The right subtree contains values greater than the root.
2. **Height Balance Condition:**
   - The difference in height between the left and right subtrees of any node (known as the **balance factor**) must be between -1 and 1.

$$\text{Balance Factor (BF)} = \text{Height of Left Subtree} - \text{Height of Right Subtree}$$

   - If $|\text{BF}| > 1$, the tree becomes unbalanced, and rotations are required to restore balance.
3. **Logarithmic Height:**
   - Due to its balancing condition, the height of an AVL tree with n nodes is $O(\log n)$, ensuring efficient operations.
4. **Efficient Operations:**
   - Searching, insertion, and deletion operations have a time complexity of $O(\log n)$.

Single Rotations

1. Right Rotation (RR): Used when a node is inserted into the left subtree of the left child, causing a left-heavy tree.
   - Process:
     - The right child of the left subtree becomes the left child of the current node.
     - The left subtree becomes the new root.

   2. Left Rotation (LL): Used when a node is inserted into the right subtree of the right child, causing a right-heavy tree.
   - Process:
     - The left child of the right subtree becomes the right child of the current node.
     - The right subtree becomes the new root.

Double Rotations are combinations of two single rotations. They are used when the imbalance involves two different directions (e.g., left-right or right-left).

1. **Left-Right Rotation (LR)**: Occurs when a node is inserted into the right subtree of the left child.
   - Process:
     - First, perform a left rotation on the left child.

- Then, perform a right rotation on the root.

2. Right-Left Rotation (RL): Occurs when a node is inserted into the left subtree of the right child.

- Process:
  - First, perform a right rotation on the right child.
  - Then, perform a left rotation on the root.

Maintaining Balance During Insertions

1. Insert the new node as in a regular BST.
2. Traverse up the tree and update balance factors.
3. If a balance factor exceeds $|1|$, perform the appropriate rotation(s):
   - LL for left-heavy subtree.
   - RR for right-heavy subtree.
   - LR or RL for mixed imbalances.

Maintaining Balance During Deletions

1. Delete the node as in a regular BST.
2. Traverse up the tree and update balance factors.
3. If the balance factor violates the AVL condition:
   - Perform rotations to restore balance.
   - Ensure the balancing propagates upwards.

==========

Q2. a. Write functions in C Programming language to implement following operations on a Singly Linked List: Deletion (at beginning, end, and a node having a specific data value) and traversal (display all nodes)

```
// Define a structure for the linked list node
struct Node {
    int data;
    struct Node* next;
};
// Function prototypes
void deleteAtBeginning(struct Node** head);
void deleteAtEnd(struct Node** head);
void deleteByValue(struct Node** head, int value);
void traverse(struct Node* head);
void insertAtEnd(struct Node** head, int data); // Helper function to add nodes for demonstration
```

```
// Function to delete a node at the beginning of the linked list
void deleteAtBeginning(struct Node** head) {
    if (*head == NULL) {
        printf("The list is empty. No deletion performed.\n");
        return;
    }
    struct Node* temp = *head;
    *head = (*head)->next; // Move the head pointer to the next node
    free(temp); // Free the memory of the deleted node
    printf("Node deleted at the beginning.\n");
}
// Function to delete a node at the end of the linked list
void deleteAtEnd(struct Node** head) {
    if (*head == NULL) {
        printf("The list is empty. No deletion performed.\n");
        return;
    }
    struct Node* temp = *head;
    // If there's only one node
    if (temp->next == NULL) {
        *head = NULL;
        free(temp);
        printf("Node deleted at the end.\n");
        return;
    }
    // Traverse to the second-last node
    while (temp->next->next != NULL) {
        temp = temp->next;
    }
    free(temp->next); // Free the last node
    temp->next = NULL; // Set the second-last node's next to NULL
    printf("Node deleted at the end.\n");
}
// Function to delete a node by its data value
void deleteByValue(struct Node** head, int value) {
    if (*head == NULL) {
        printf("The list is empty. No deletion performed.\n");
        return;
    }
    struct Node* temp = *head;
    struct Node* prev = NULL;
    // If the node to be deleted is the head
    if (temp != NULL && temp->data == value) {
        *head = temp->next;
        free(temp);
        printf("Node with value %d deleted.\n", value);
        return;
    }
```

```
    // Search for the node to be deleted
    while (temp != NULL && temp->data != value) {
        prev = temp;
        temp = temp->next;
    }
    // If the value was not found
    if (temp == NULL) {
        printf("Node with value %d not found.\n", value);
        return;
    }
    // Unlink the node and free memory
    prev->next = temp->next;
    free(temp);
    printf("Node with value %d deleted.\n", value);
}
// Function to traverse and display all nodes in the linked list
void traverse(struct Node* head) {
    if (head == NULL) {
        printf("The list is empty.\n");
        return;
    }
    struct Node* temp = head;
    printf("Linked List: ");
    while (temp != NULL) {
        printf("%d -> ", temp->data);
        temp = temp->next;
    }
    printf("NULL\n");
}
```

==========

Q2. b. Define a B Tree and explain its structure, including the concepts of order and node capacity. Construct a B Tree of order 5, assuming keys arrive in the following order
1, 12, 8, 2, 25, 5, 14, 28, 17, 7, 52, 16, 48, 68, 3, 26, 29, 53, 55, 45

A B-tree is a self-balancing search tree commonly used in database systems and file systems to store and manage large datasets. It maintains sorted data and allows efficient operations such as search, insertion, and deletion. Its structure ensures logarithmic height and optimal use of disk I/O, making it suitable for large-scale applications.

Properties of a B-Tree

1. **Balanced Height**: The path from the root to any leaf has the same length, ensuring balance.
2. **Key Distribution**: Keys within a node are always sorted, and keys across nodes are distributed in ranges.
3. **Efficient Operations**: Operations such as search, insertion, and deletion require O(logn) time due to balanced height.

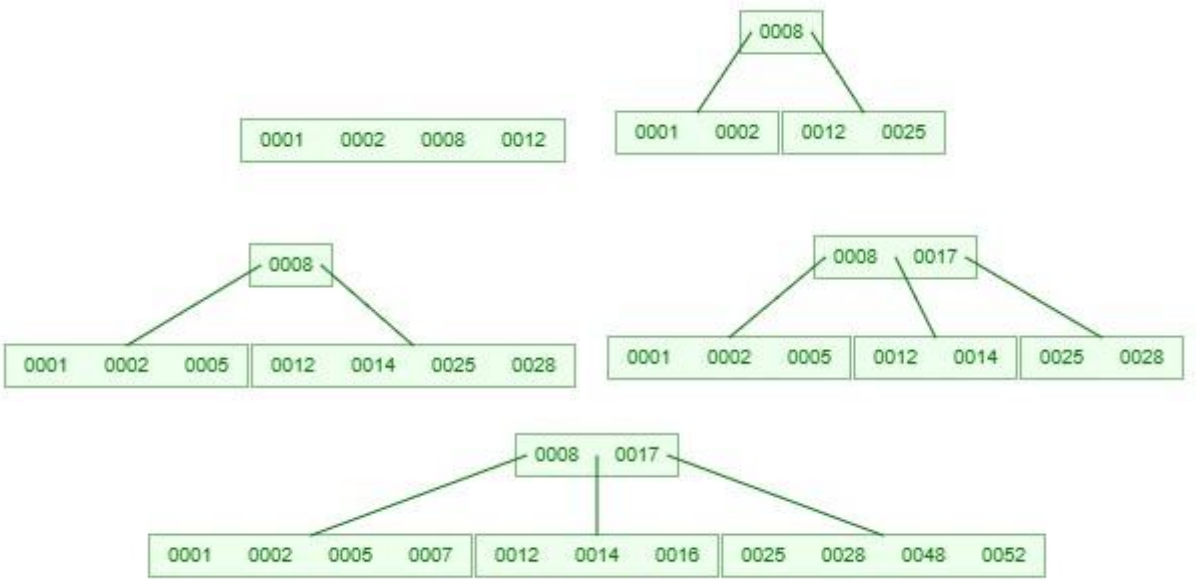
Structure of a B-Tree

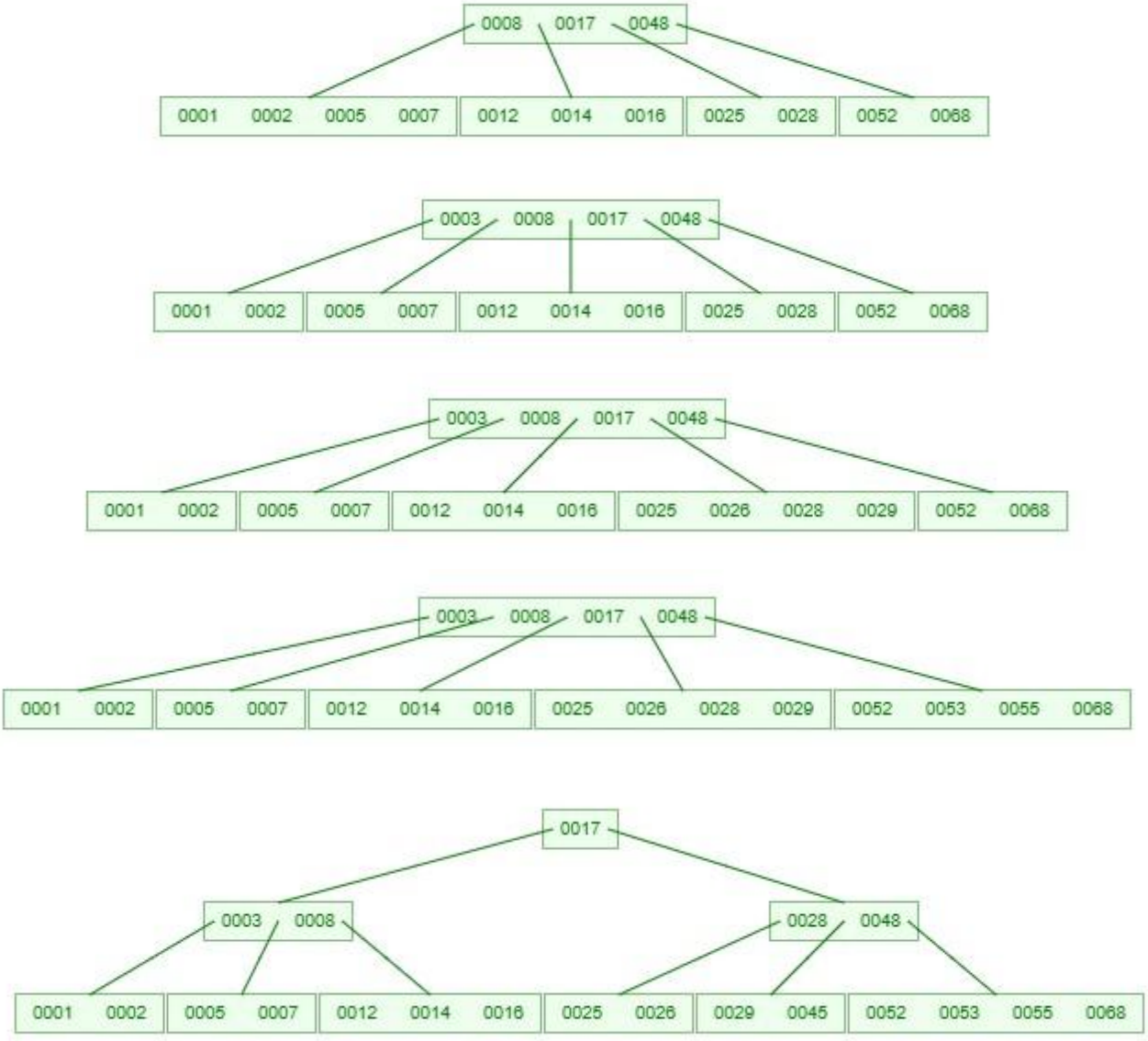
1. Nodes:
    - Each node in a B-tree contains a set of keys and pointers to child nodes (if any).
    - Keys within a node are stored in sorted order.

2. Root Node:
    - The root node may have fewer keys than other nodes.
    - If the tree has only one node, it serves as the root and a leaf.
3. Internal Nodes:
    - Internal nodes have both keys and pointers to child nodes.
    - The keys act as separators that determine the range of values stored in each subtree.
4. Leaf Nodes:
    - Leaf nodes contain only keys and do not have child pointers.
    - All leaf nodes are at the same depth, maintaining balance.

Concepts of Order and Node Capacity

1. **Order (m):**
    - The **order** of a B-tree, denoted as m, determines the maximum number of children each node can have.
    - For a B-tree of order m:
        - A node can have at most $m - 1$ keys.
        - A node (except the root) must have at least $\lceil \frac{m}{2} \rceil - 1$ keys.
        - The root must have at least 1 key (unless it is the only node).
2. **Node Capacity:**
    - The number of keys in a node ranges from $\lceil \frac{m}{2} \rceil - 1$ to $m - 1$.
    - The number of child pointers ranges from $\lceil \frac{m}{2} \rceil$ to m.





==========

Q3. a. Discuss the differences between Linked lists and arrays in terms of memory allocation, size, flexibility, access time, and ease of insertion / deletion operations. Provide examples to illustrate each point.

1. Memory Allocation

- **Array**:
  - **Contiguous Allocation**: Arrays require memory to be allocated in a contiguous block.
  - **Fixed Size**: The size of an array must be defined at the time of its declaration (static allocation in most programming languages).

    ```
    int arr[5];    // Allocates memory for 5 integers in a contiguous block
    ```
  - **Limitation**: Resizing an array may require creating a new array and copying data.
- **Linked List**:
  - **Dynamic Allocation**: Each node is allocated dynamically and may reside in non-contiguous memory locations.

```
struct Node {
    int data;
    struct Node* next;
};
struct Node* node = (struct Node*)malloc(sizeof(struct Node));
// Allocates memory dynamically
```

- **Advantage**: The size can grow or shrink dynamically based on the number of nodes.

2. Size

- **Array**:
  - **Fixed Size**: The size is determined at the time of declaration and cannot be changed during runtime in static arrays.

    ```
    int arr[10];  // Size fixed at 10
    ```

  - **Limitation**: Wastes memory if the declared size is larger than required.
- **Linked List**:
  - **Dynamic Size**: The size can increase or decrease as nodes are added or removed.

    ```
    struct Node* head = NULL;  // Start with an empty list
    // Add or delete nodes as needed
    ```

3. Flexibility

- **Array**:
  - Less flexible since resizing often requires creating a new array and copying the data.
  - The size must be predefined, leading to potential memory wastage or shortage.

    ```
    // Resize an array
    int* newArr = (int*)malloc(newSize * sizeof(int));
    memcpy(newArr, arr, oldSize * sizeof(int));
    ```

- **Linked List**:
  - Highly flexible since nodes can be added or removed dynamically without resizing or memory reallocation.

    ```
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    newNode->data = value;
    newNode->next = head;
    head = newNode;  // Add at the beginning
    ```

4. Access Time

- **Array**:
  - **Fast Random Access**: Direct access using an index, O(1)

    ```
    int value = arr[2];  // Directly access the third element
    ```

- **Linked List**:
  - **Sequential Access**: Nodes must be traversed sequentially, O(n)

    ```
    struct Node* temp = head;
    while (temp != NULL && count < index) {
        temp = temp->next;
        count++;
    }
    // Access the node at the desired index
    ```

5. Ease of Insertion / Deletion

- **Array**:
  - Insertion or deletion requires shifting elements, which can be time-consuming O(n)
    Example (Insert at position 2):

```
for (int i = n; i > pos; i--) {
    arr[i] = arr[i - 1];
}
arr[pos] = newValue;
```

- **Linked List**:
  - Insertion and deletion are straightforward and efficient, especially at the beginning or end O(1) for head operations, O(n) for general cases).

  Example (Insert at beginning):

```
struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
newNode->data = value;
newNode->next = head;
head = newNode;  // Insert at beginning
```

Comparison Summary

Feature	Array	Linked List
Memory Allocation	Contiguous, static	Non-contiguous, dynamic
Size	Fixed, predefined	Dynamic, grows/shrinks
Flexibility	Less flexible	Highly flexible
Access Time	$O(1)$, fast random access	$O(n)$, sequential access
Insertion/Deletion	$O(n)$, requires shifting	$O(1)$ at head, easier overall

==========

Q3.b. Write a C Program to convert an infix to postfix expression using a Stack

```
#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>
#include <string.h>
#define MAX 100
typedef struct {                    // Define the stack structure
    int top;
    char items[MAX];
} Stack;
// Stack operations
void push(Stack* s, char item) {
    if (s->top == MAX - 1) {
        printf("Stack overflow\n");
        return;
    }
    s->items[++(s->top)] = item;
}
```

```
char pop(Stack* s) {
    if (s->top == -1) {
        printf("Stack underflow\n");
        return '\0';
    }
    return s->items[(s->top)--];
}
char peek(Stack* s) {
    if (s->top == -1) return '\0';
    return s->items[s->top];
}
int isEmpty(Stack* s) {
    return s->top == -1;
}
// Utility functions
int precedence(char op) {
    switch (op) {
        case '+':
        case '-': return 1;
        case '*':
        case '/': return 2;
        case '^': return 3;
        default: return 0;
    }
}
int isOperator(char ch) {
    return ch == '+' || ch == '-' || ch == '*' || ch == '/' || ch == '^';
}
void infixToPostfix(char* infix, char* postfix) {
    Stack s;
    s.top = -1;
    int i, j = 0;
    char ch, temp;
    for (i = 0; infix[i] != '\0'; i++) {
        ch = infix[i];
        if (isdigit(ch) || isalpha(ch)) {
            // Operand: directly add to postfix
            postfix[j++] = ch;
        } else if (ch == '(') {
            // Left parenthesis: push onto stack
            push(&s, ch);
        } else if (ch == ')') {
            // Right parenthesis: pop until '(' is found
            while (!isEmpty(&s) && (temp = pop(&s)) != '(') {
                postfix[j++] = temp;
            }
        } else if (isOperator(ch)) {
            // Operator: pop operators of higher or equal precedence
```

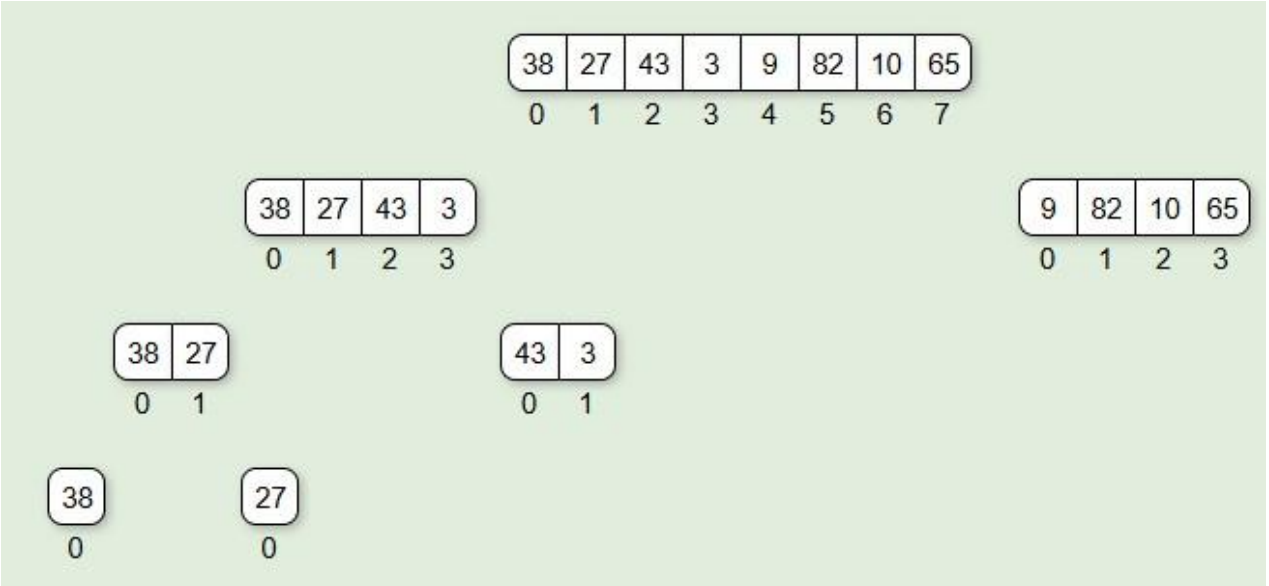
```
            while (!isEmpty(&s) && precedence(peek(&s)) >= precedence(ch)) {
                postfix[j++] = pop(&s);
            }
            push(&s, ch);
        }
    }
    // Pop remaining operators in the stack
    while (!isEmpty(&s)) {
        postfix[j++] = pop(&s);
    }
    postfix[j] = '\0';  // Null-terminate the postfix expression
}
// Main function
int main() {
    char infix[MAX], postfix[MAX];
    printf("Enter an infix expression: ");
    scanf("%s", infix);
    infixToPostfix(infix, postfix);
    printf("Postfix expression: %s\n", postfix);
    return 0;
}
```

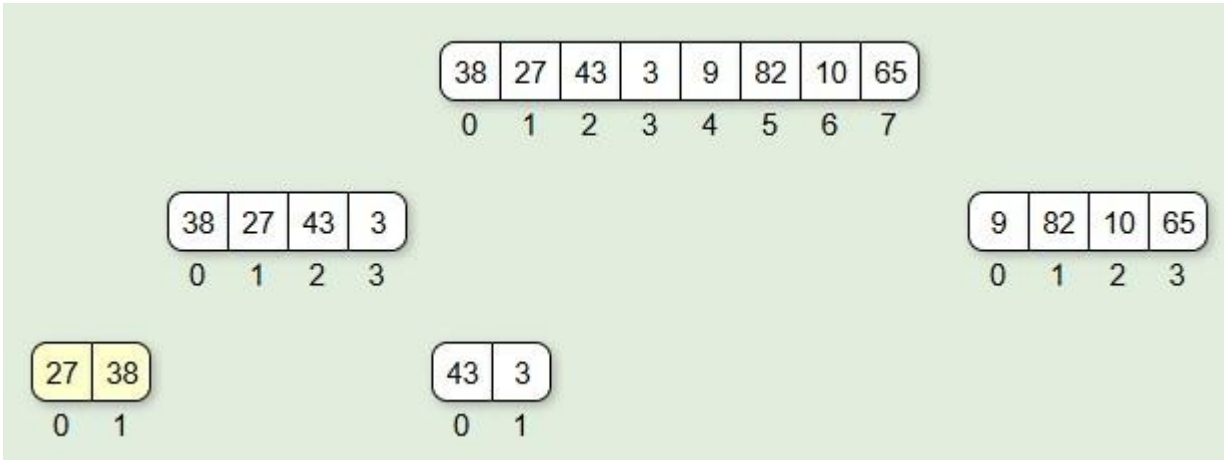
==========

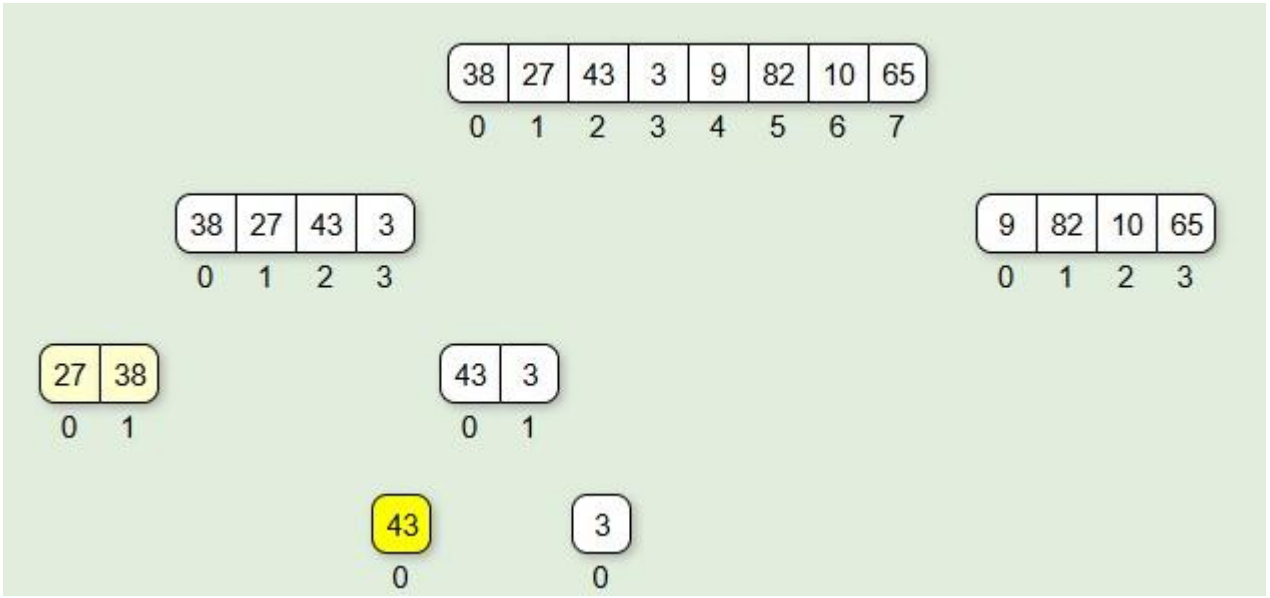
Q 4. a. Write functions in C Program, to implement Merge Sort . Provide a step-by-step example of Merge Sort on the following array. 38, 27, 43, 3, 9, 82, 10, 65

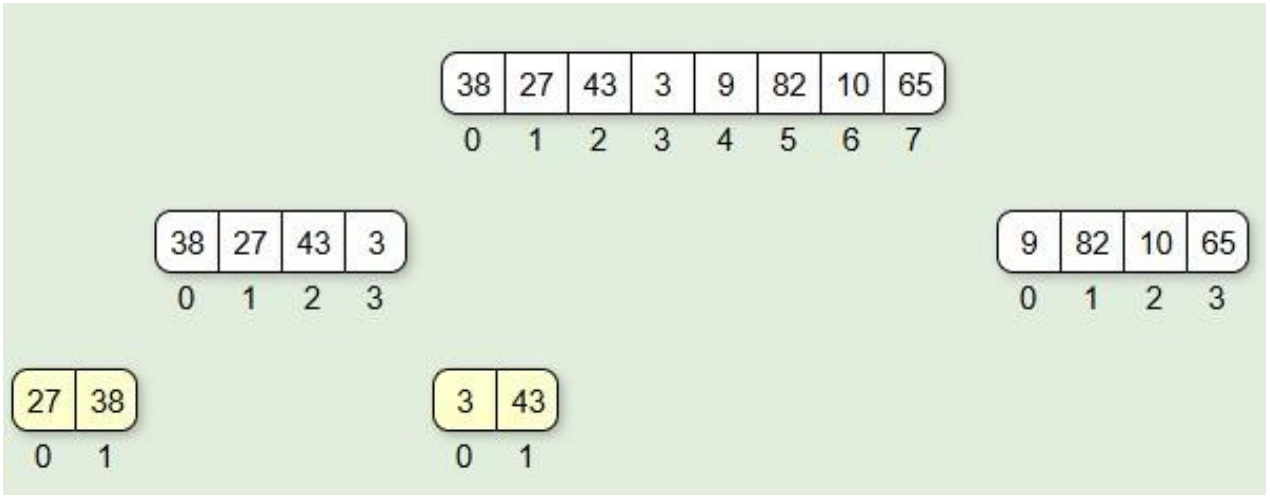
```
#include <stdio.h>
// Function to merge two subarrays into a sorted array
void merge(int arr[], int left, int mid, int right) {
    int i, j, k;
    int n1 = mid - left + 1;  // Size of left subarray
    int n2 = right - mid;     // Size of right subarray
    // Create temporary arrays
    int L[n1], R[n2];
    // Copy data to temporary arrays L[] and R[]
    for (i = 0; i < n1; i++)
        L[i] = arr[left + i];
    for (j = 0; j < n2; j++)
        R[j] = arr[mid + 1 + j];
    // Merge the temporary arrays back into arr[]
    i = 0; // Initial index of left subarray
    j = 0; // Initial index of right subarray
    k = left; // Initial index of merged array
    while (i < n1 && j < n2) {
        if (L[i] <= R[j]) {
            arr[k] = L[i];
            i++;
        } else {
            arr[k] = R[j];
```

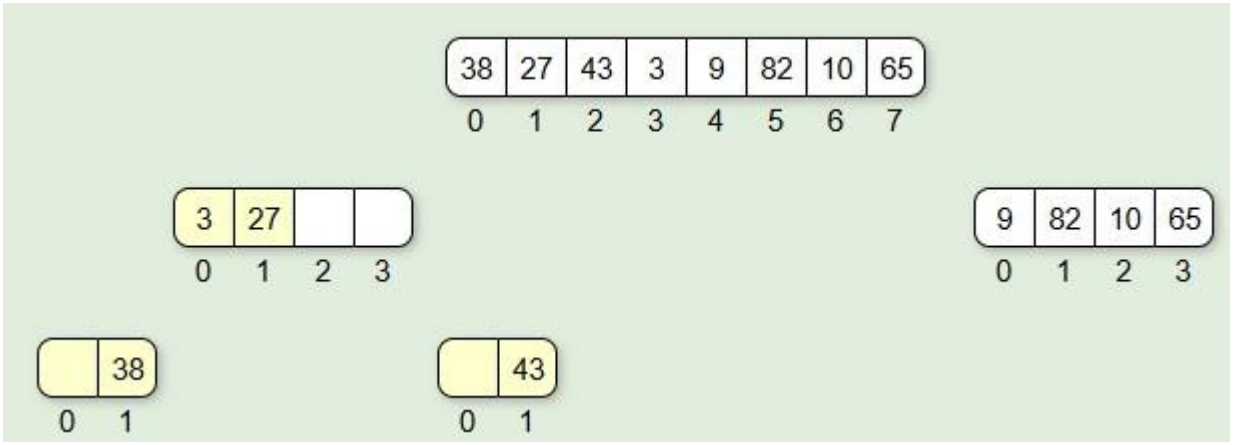
```
            j++;
        }
        k++;
    }
    // Copy any remaining elements of L[]
    while (i < n1) {
        arr[k] = L[i];
        i++;
        k++;
    }
    // Copy any remaining elements of R[]
    while (j < n2) {
        arr[k] = R[j];
        j++;
        k++;
    }
}
// Function to perform Merge Sort
void mergeSort(int arr[], int left, int right) {
    if (left < right) {
        int mid = left + (right - left) / 2;

        // Recursively sort first and second halves
        mergeSort(arr, left, mid);
        mergeSort(arr, mid + 1, right);

        // Merge the sorted halves
        merge(arr, left, mid, right);
    }
}
```

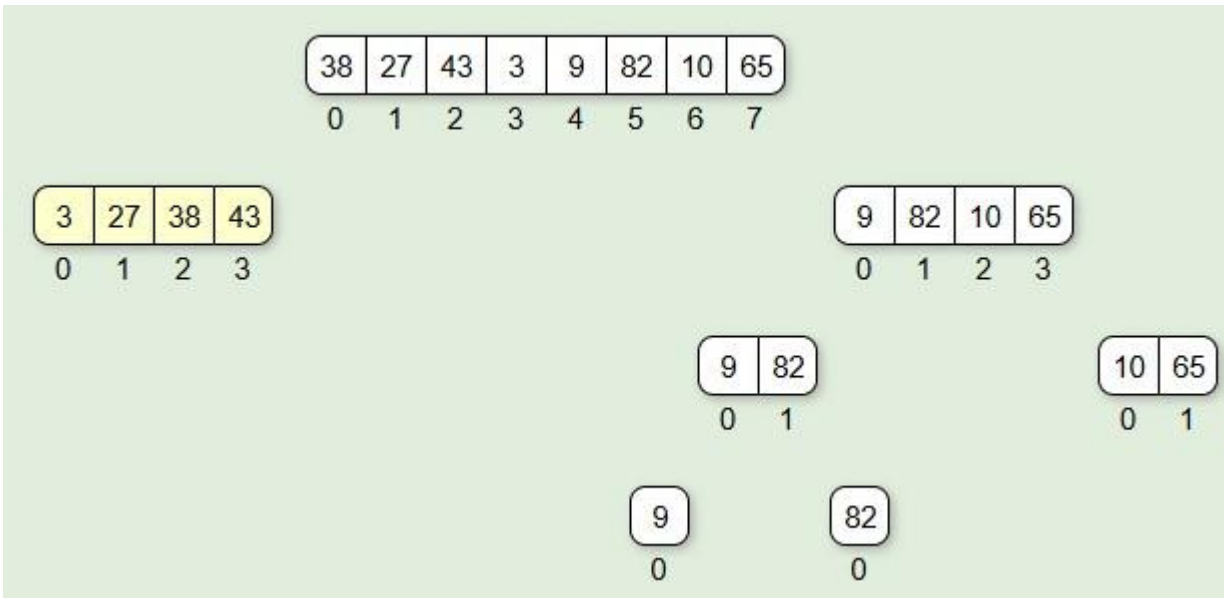


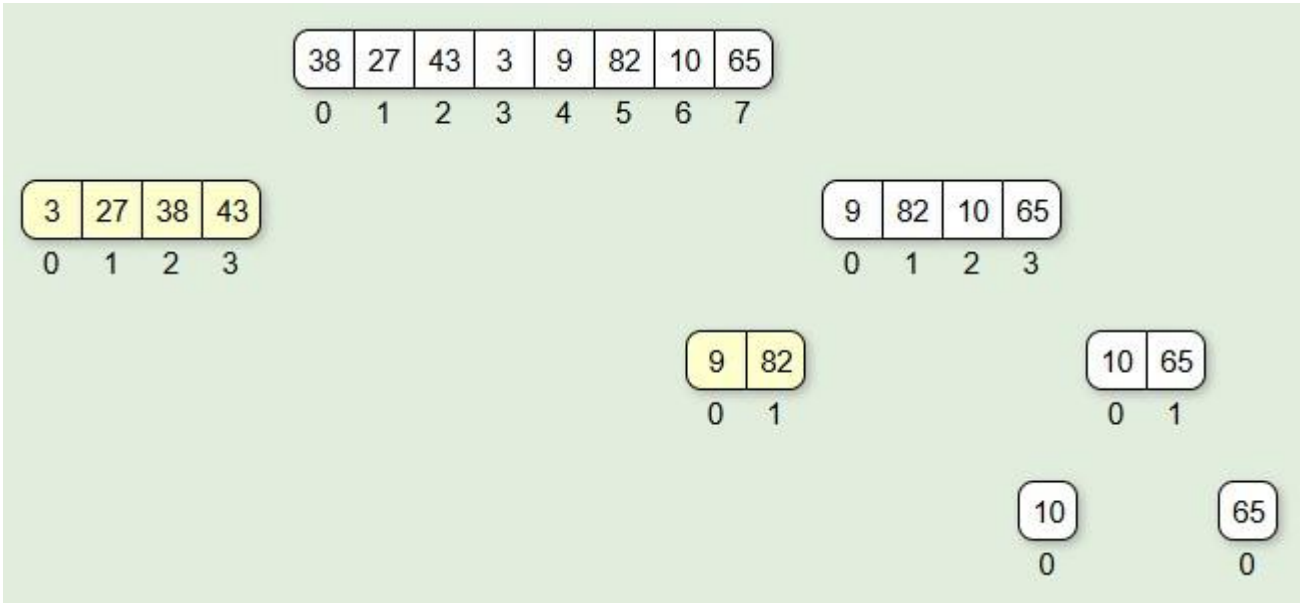

38 27 43 3 9 82 10 65
0 1 2 3 4 5 6 7
38 27 43 3
0 1 2 3
9 82 10 65
0 1 2 3
27 38
0 1
43 3
0 1

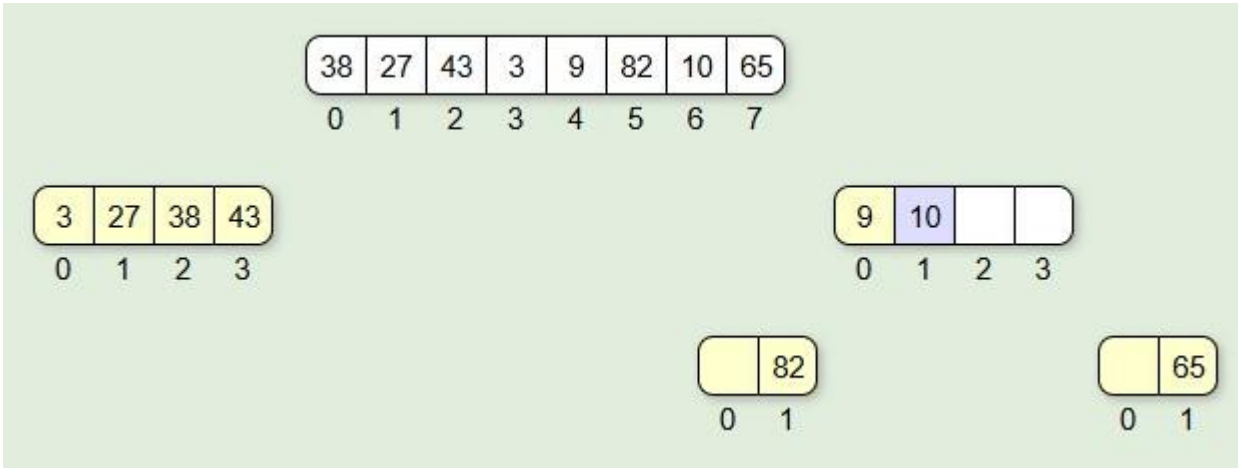

38 27 43 3 9 82 10 65
0 1 2 3 4 5 6 7
38 27 43 3
0 1 2 3
9 82 10 65
0 1 2 3
27 38
0 1
43 3
0 1
43
0
3
0

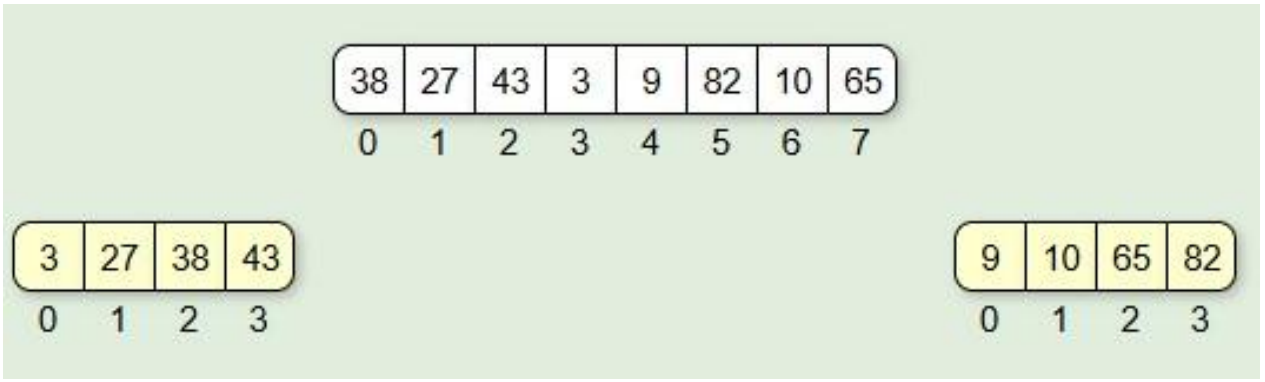

38 27 43 3 9 82 10 65
0 1 2 3 4 5 6 7
38 27 43 3
0 1 2 3
9 82 10 65
0 1 2 3
27 38
0 1
3 43
0 1

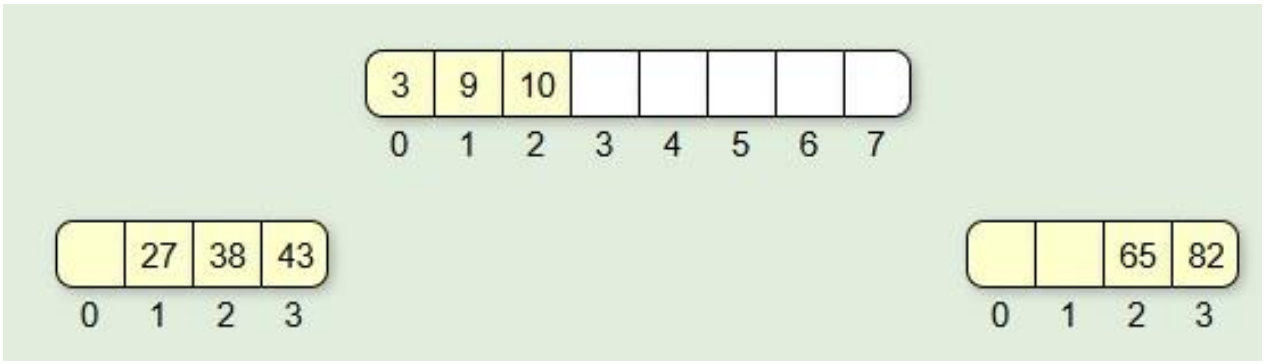

38 27 43 3 9 82 10 65
0 1 2 3 4 5 6 7
3 27
0 1
9 82 10 65
0 1 2 3
38
0 1
43
0 1

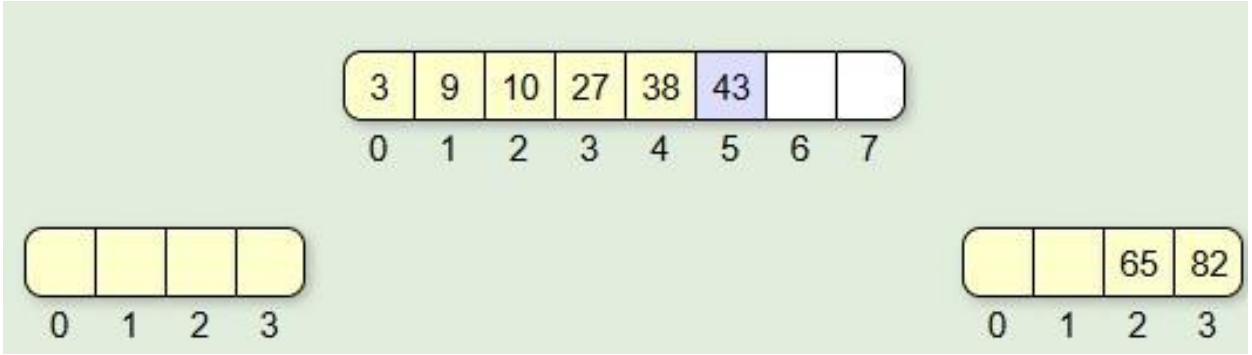

38 27 43 3 9 82 10 65
0 1 2 3 4 5 6 7
3 27 38 43
0 1 2 3
9 82 10 65
0 1 2 3
9 82
0 1
10 65
0 1
9
0
82
0

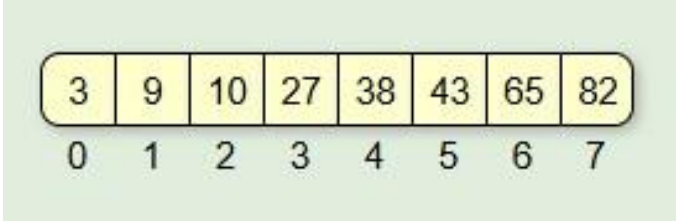

38 27 43 3 9 82 10 65
0 1 2 3 4 5 6 7
3 27 38 43
0 1 2 3
9 82 10 65
0 1 2 3
9 82
0 1
10 65
0 1
10
0
65
0


38 27 43 3 9 82 10 65
0 1 2 3 4 5 6 7
3 27 38 43
0 1 2 3
9 10
0 1 2 3
82
0 1
65
0 1


38 27 43 3 9 82 10 65
0 1 2 3 4 5 6 7
3 27 38 43
0 1 2 3
9 10 65 82
0 1 2 3


3 9 10
0 1 2 3 4 5 6 7
27 38 43
0 1 2 3
65 82
0 1 2 3


3 9 10 27 38 43
0 1 2 3 4 5 6 7
0 1 2 3
65 82
0 1 2 3


3 9 10 27 38 43 65 82
0 1 2 3 4 5 6 7

==========

Q 4. b. Describe what an Abstract Data Type (ADT) is and explain its importance in data structure design. Provide an example to illustrate your explanation.

An **Abstract Data Type (ADT)** is a mathematical model for a data structure that defines the data type and the operations that can be performed on it, independent of implementation details. It focuses on *what* operations are supported and not *how* these operations are implemented.

Key Characteristics:

1. **Encapsulation**:
    - ADT encapsulates the data and the operations that can be performed, separating the interface from the implementation.
2. **Abstraction**:
    - Users of an ADT interact with it through its interface, without needing to know how it is implemented.
3. **Flexibility**:
    - The implementation of an ADT can be changed without affecting the program using it, as long as the interface remains consistent.

Importance in Data Structure Design:

1. **Modularity**:
    - ADTs promote modular design, enabling separation of concerns between the definition of a data structure and its implementation.
2. **Reusability**:
    - ADTs can be reused across different programs, as they provide a standard interface.
3. **Ease of Maintenance**:
    - Changes in the implementation can be made without altering the programs that use the ADT.
4. **Implementation Independence**:
    - Allows the same ADT to be implemented differently depending on the requirements, e.g., stacks can be implemented using arrays or linked lists.
5. **Improved Code Clarity**:
    - Focusing on what the ADT does makes code easier to understand and use.

Example: Stack as an ADT

A **stack** is a commonly used ADT that follows the **LIFO (Last In, First Out)** principle. Its operations are defined as follows:

1. **Operations**:
    - `push(x)`: Add an element xxx to the top of the stack.
    - `pop()`: Remove the top element of the stack.
    - `peek()`: View the top element without removing it.
    - `isEmpty()`: Check if the stack is empty.
2. **Interface**:
    - The user interacts with the stack through the above operations, without needing to know whether the stack is implemented using an array or a linked list.
3. **Implementation**:
    - The stack can be implemented:
        - **Using arrays**:
            - Fixed-size array, with a pointer indicating the top.
        - **Using linked lists**:
            - Dynamically allocated nodes, with a pointer to the top node.

==========

Q4 c. How singly linked lists can be used in polynomial representation and addition?

Singly Linked Lists in Polynomial Representation and Addition

A polynomial can be represented as a sum of terms, each having a coefficient and an exponent, such as:

$$P(x) = 4x^3 + 3x^2 + 2x + 1$$

Using a **singly linked list**, each node can store:

1. **Coefficient** of the term.
2. **Exponent** of the term.
3. A pointer to the **next node**, linking terms in order of descending exponents.

Structure of a node

```
struct Node {
    int coeff;      // Coefficient of the term
    int exp;        // Exponent of the term
    struct Node* next;  // Pointer to the next term
};
```

Polynomial Representation

Each term of the polynomial is stored as a node in the linked list:

- The **head** of the list points to the first term.
- Terms are stored in descending order of exponents to simplify operations like addition.

For example:

$$P(x) = 4x^3 + 3x^2 + 2x + 1$$

Polynomial Addition Using Singly Linked Lists

When adding two polynomials, terms with the **same exponent** are combined by adding their coefficients. If a term exists in one polynomial but not the other, it is directly added to the result.

Algorithm for Polynomial Addition

1. Traverse both polynomials using pointers (`p1` for the first and `p2` for the second).
2. Compare the exponents of the current terms:
   - If `exp(p1) > exp(p2)` : Add the term from `p1` to the result and move `p1` forward.
   - If `exp(p1) < exp(p2)` : Add the term from `p2` to the result and move `p2` forward.
   - If `exp(p1) == exp(p2)` : Add their coefficients, create a new term, and move both `p1` and `p2` forward.
3. Append remaining terms from `p1` or `p2` to the result if one polynomial has extra terms.
4. Return the resulting linked list.

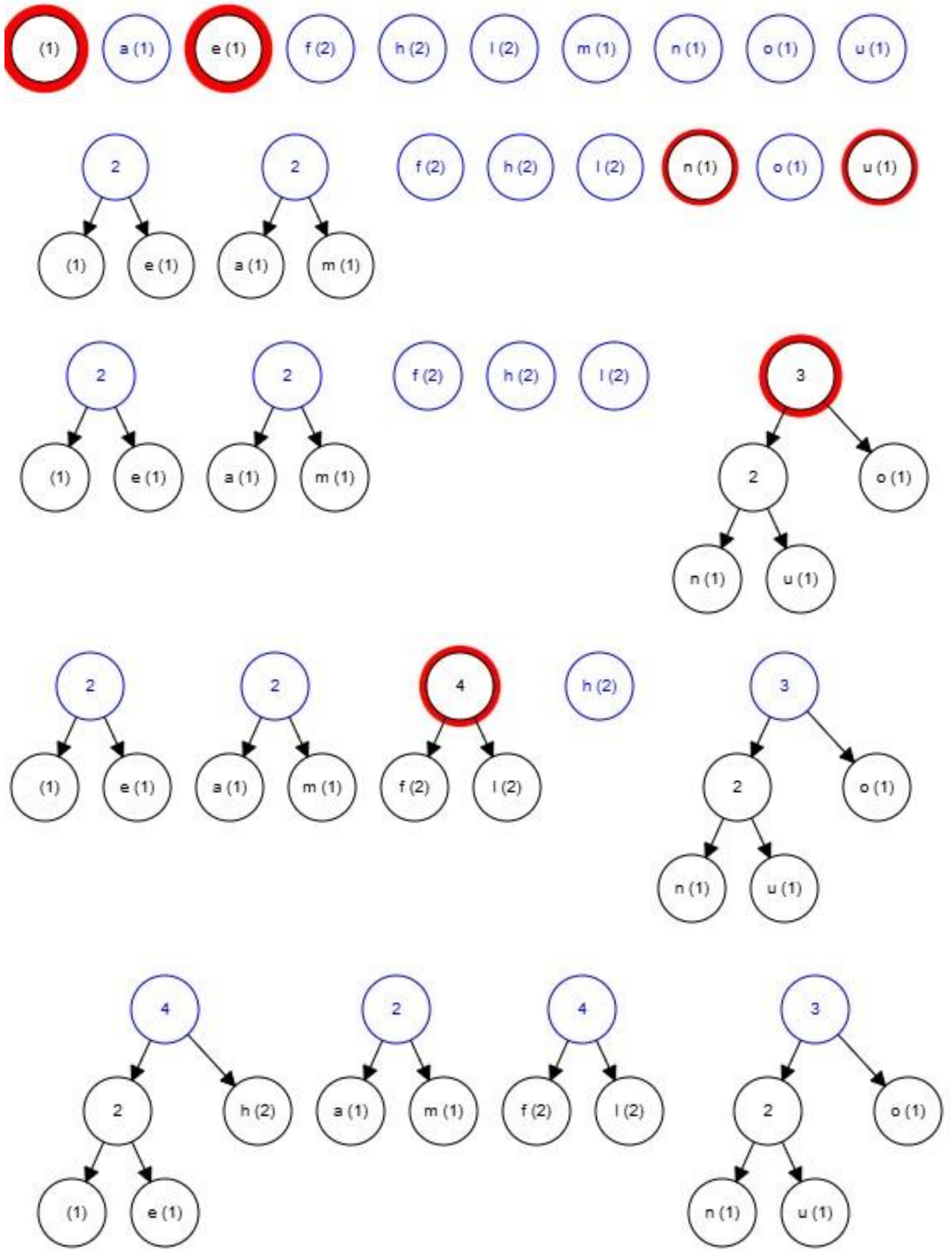
First Polynomial: 4x^3 + 3x^2 + 2x + 1
Second Polynomial: 3x^3 + x^2 + 5
Resultant Polynomial: 7x^3 + 4x^2 + 2x + 6

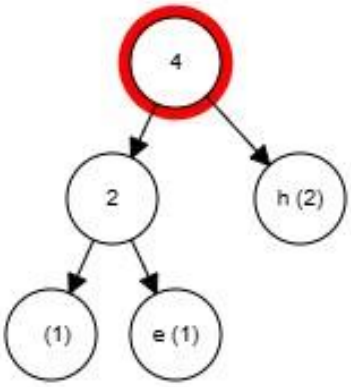
==========

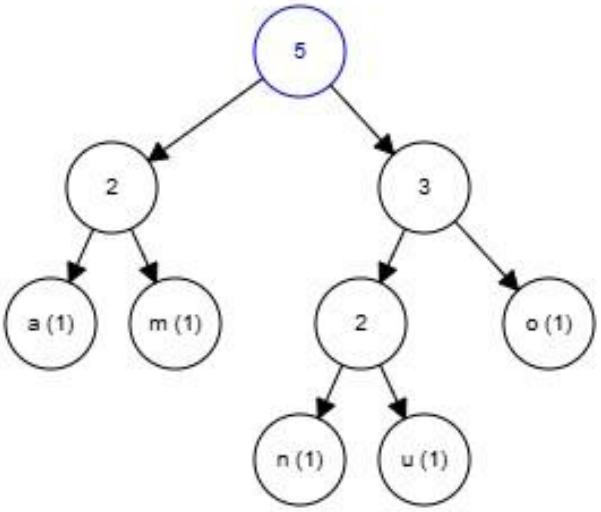
Q 5. a. With the help of the given diagram, explain following terms with respect to a tree - Root, siblings, internal node, external node, ancestors of a node, descendant of a node, depth of a node, height, degree of a node, degree of a tree

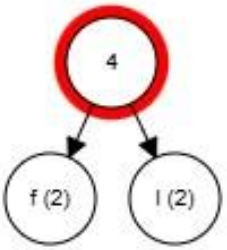
Term	Description
Root	The topmost node of the tree.
Siblings	Nodes with the same parent.
Internal Node	A node with at least one child.
External Node	A node with no children (leaf node).
Ancestors	Nodes along the path from the root to a specific node.
Descendants	Nodes originating from a specific node (children, grandchildren, etc.).
Depth	Number of edges from the root to the node.
Height	Number of edges on the longest path from a node to a leaf.
Degree of Node	Number of children of the node.
Degree of Tree	Maximum degree of any node in the tree.

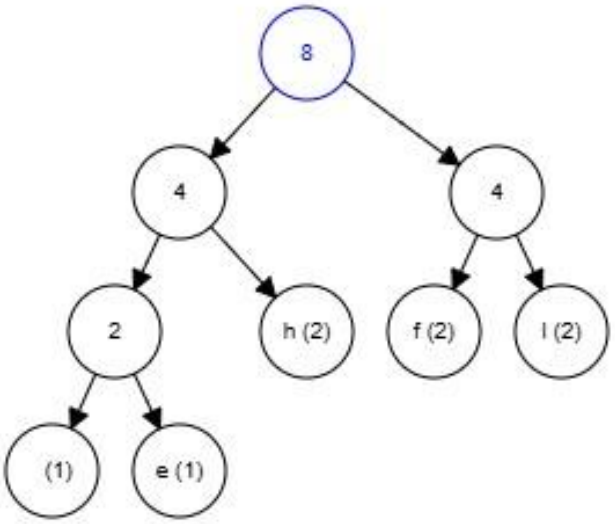
Q5.b. Build Huffman Tree for the given string : hello huffman

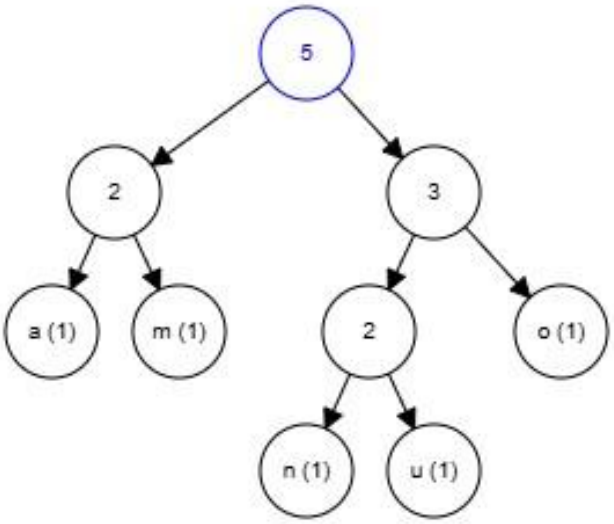


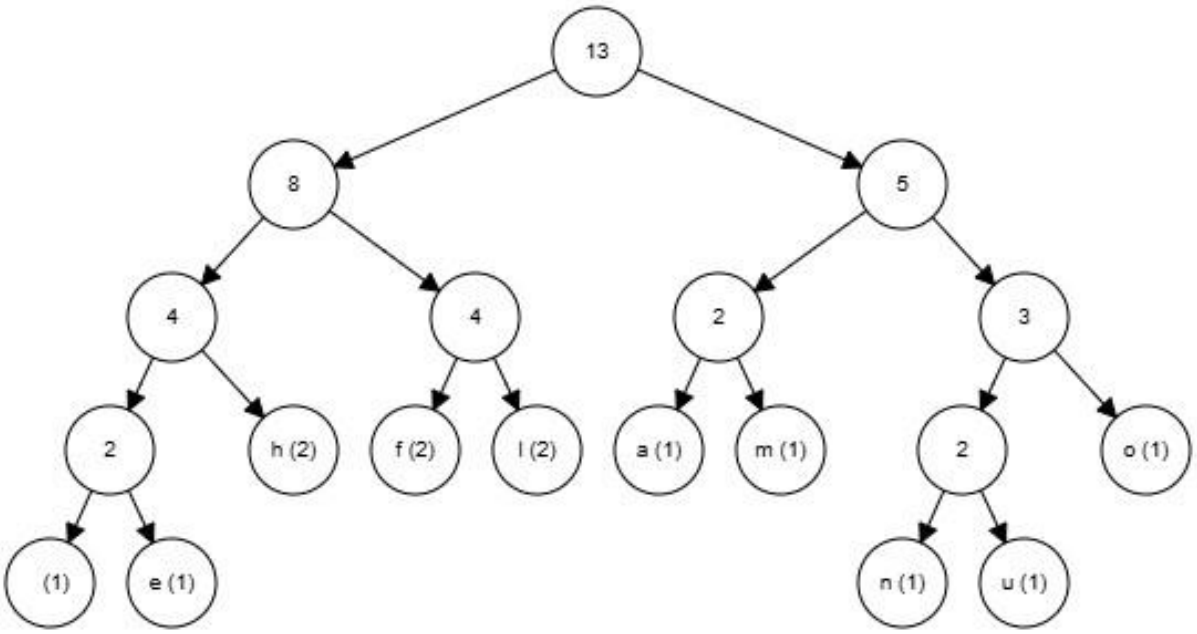

4
2
h (2)
(1)
e (1)

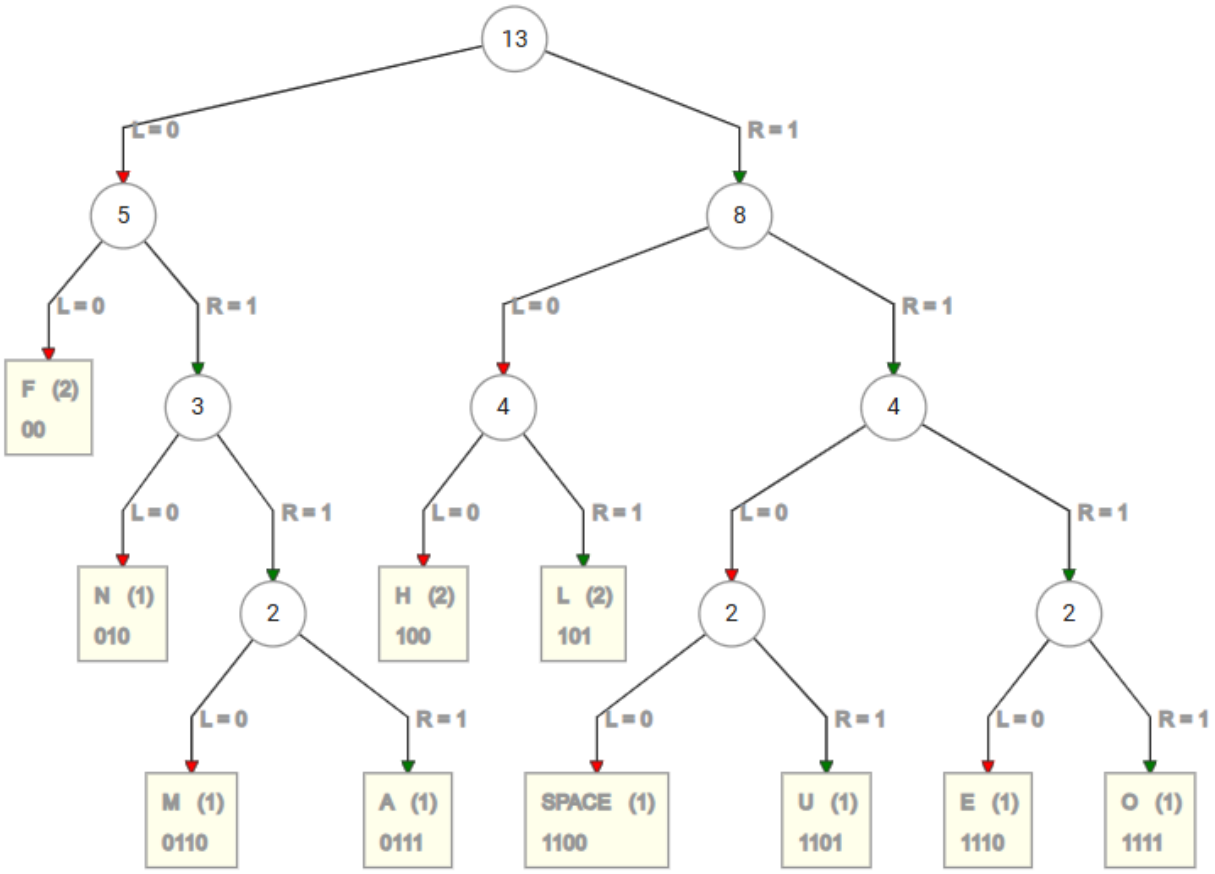

5
2
3
a (1)
m (1)
2
o (1)
n (1)
u (1)


4
f (2)
l (2)


8
4
4
2
h (2)
f (2)
l (2)
(1)
e (1)


5
2
3
a (1)
m (1)
2
o (1)
n (1)
u (1)


13
8
5
4
4
2
3
2
h (2)
f (2)
l (2)
a (1)
m (1)
2
o (1)
(1)
e (1)
n (1)
u (1)



==========

Q5c. What are Tries, and how are they used in searching for strings?

A Trie (pronounced as "try"), also known as a prefix tree, is a tree-like data structure used to efficiently store and retrieve keys in a dataset of strings. It is especially useful for operations involving strings, such as searching, insertion, and prefix matching.

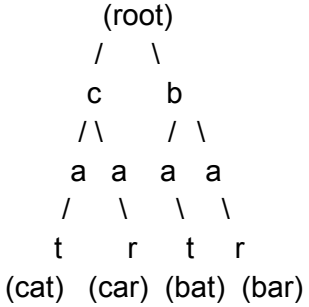
Structure of a Trie

1. Nodes: Each node represents a character of the string. The root node is empty and does not contain any character.
2. Edges: connect nodes and represent transitions between characters in the string.
3. End of Word Marker: A special marker or flag is used at the node to signify the end of a valid string.

Key Properties

1. Hierarchical Storage: Strings are stored by sharing common prefixes.
2. Space-Efficient: Common prefixes are stored only once, reducing redundancy.
3. Time Complexity: Insertion and Search: O(L)O(L)O(L), where LLL is the length of the string.

Consider inserting the words: `cat`, `car`, `bat`, and `bar` into a Trie.



Operations on a Trie

1. Insertion
    - Start at the root.
    - For each character in the string:
        - If the character does not exist as a child node, create a new node.
        - Move to the child node corresponding to the character.
    - Mark the last node as the end of the word.
2. Searching
    - Start at the root.
    - For each character in the string:
        - Check if the character exists as a child node.
        - If not, the string is not present.
        - If all characters are found and the last node is marked as the end of the word, the string is present.
3. Prefix Matching
    - Similar to search, but stop after processing all characters of the prefix.

Tries are especially efficient for searching strings due to their hierarchical structure and ability to share prefixes.

1. Full String Search
    - Follow the path of characters from the root to the last node.
    - Check if the end marker is present.
2. Prefix Search
    - Follow the path of characters corresponding to the prefix.
    - If the path exists, all words in the subtree starting from the current node share that prefix.