EXPERIMENT NO: 9

AIM: Demonstrate how binary trees can be represented and traversed using two different data structures: arrays and linked lists.

THEORY:

1. Binary Trees :

Binary search tree is a data structure that quickly allows us to maintain a sorted list of numbers. It is called a binary tree because each tree node has a maximum of two children.It is called a search tree because it can be used to search for the presence of a number in O(log(n)) time.The properties that separate a binary search tree from a regular binary tree is

1.All nodes of left subtree are less than the root node

2.All nodes of right subtree are more than the root node

3.Both subtrees of each node are also BSTs.

2. Tree Representation

Array Representation :

- In the array representation of a binary tree, elements are stored in a contiguous block of memory (an array). The root node is stored at index 1 (or 0, depending on the implementation), and for any node at index i, its left child is at index $2i$ and its right child is at index $2i + 1$.
- This representation is efficient in terms of memory usage for complete binary trees but may lead to wasted space in sparse trees.

Linked List Representation :

- In the linked list representation, each node is an object that contains the data and pointers to its left and right children. This allows for a more flexible tree structure, as nodes can be added or removed without needing to resize an array.
- This representation is better suited for dynamic binary trees where the number of nodes can vary.

CODE:

USING ARRAY

```
#include <stdio.h>

int max_node = 15;
char tree[] = {'\0', 'D', 'A', 'F', 'E', 'B', 'R', 'T', 'G', 'Q', '\0', '\0', 'V', '\0', 'J', 'L'};

int get_right_child(int index){
  if (tree[index] != '\0' && ((2 * index) + 1) <= max_node)
    return (2 * index) + 1;
  return -1;
}

int get_left_child(int index){
  if (tree[index] != '\0' && (2 * index) <= max_node)
    return 2 * index;
  return -1;
}

void preorder(int index){
  if (index > 0 && tree[index] != '\0'){
    printf(" %c ", tree[index]);
    preorder(get_left_child(index));
    preorder(get_right_child(index));
  }
}

void postorder(int index){
  if (index > 0 && tree[index] != '\0'){
    postorder(get_left_child(index));
    postorder(get_right_child(index));
    printf(" %c ", tree[index]);
  }
}

void inorder(int index){
  if (index > 0 && tree[index] != '\0'){
    inorder(get_left_child(index));
```

```
    printf(" %c ", tree[index]);
    inorder(get_right_child(index));
  }
}

int main() {
  printf("Preorder:\n");
  preorder(1);
  printf("\nPostorder:\n");
  postorder(1);
  printf("\nInorder:\n");
  inorder(1);
  printf("\n");
  return 0;
}
```

OUTPUT:

```
Preorder:
 D  A  E  G  Q  B  F  R  V  T  J  L
Postorder:
 G  Q  E  B  A  V  R  J  L  T  F  D
Inorder:
 G  E  Q  A  B  D  V  R  F  J  T  L


=== Code Execution Successful ===
```

USING LINKED LIST

```
#include <stdio.h>
#include <stdlib.h>

struct node
{
    char data;
    struct node *right_child;
    struct node *left_child;
};
```

```
struct node* new_node(char data)
{
   struct node *temp;
   temp = malloc(sizeof(struct node));
   temp -> data = data;
   temp -> left_child = NULL;
   temp -> right_child = NULL;

   return(temp);
}

void preorder(struct node *root)
{
   if(root != NULL)
   {
      printf(" %c ", root -> data);
      preorder(root -> left_child);
      preorder(root -> right_child);
   }
}

void postorder(struct node *root)
{
   if(root != NULL)
   {
      postorder(root -> left_child);
      postorder(root -> right_child);
      printf(" %c ", root -> data);
   }
}

void inorder(struct node *root)
{
   if(root != NULL)
   {
      inorder(root -> left_child);
      printf(" %c ", root -> data);
      inorder(root -> right_child);
   }
}
```

```
int main()
{
    struct node *root;
    root = new_node('D');

    root -> left_child = new_node('A');
    root -> right_child = new_node('F');
    root -> left_child -> left_child = new_node('E');
    root -> left_child -> right_child = new_node('B');

    printf("Preorder:\n");
    preorder(root);
    printf("\n");
    printf("Postorder:\n");
    postorder(root);
    printf("\n");
    printf("Inorder:\n");
    inorder(root);
    printf("\n");

    return 0;
}
```

OUTPUT:

```
Preorder:
 D  A  E  B  F
Postorder:
 E  B  A  F  D
Inorder:
 E  A  B  D  F


=== Code Execution Successful ===
```

CONCLUSION:

This experiment illustrated two binary tree representations. Arrays provide efficient memory usage for complete trees but may waste space in sparse trees. Linked lists offer flexibility for dynamic trees, allowing easy addition and removal of nodes. Each method has advantages depending on tree structure and application needs.