DAA LAB MANUAL

EXERCISE-1:

Implement Recursive Binary Search Algorithm

PROGRAM:

```
#include <bits/stdc++.h>
using namespace std;
int binarySearch(vector<int>arr, int l, int r, int x)
{
  while (l <= r)
  {
    int m = l + (r - l) / 2;
    if (arr[m] == x)
    return m;
    else if (arr[m] < x)
    l = m + 1;
    else
    r = m - 1;
   }
   return -1;
}

int main()
{
        int n;
        cout<<"Enter size of Array: "; cin>>n;
        cout<<endl<<"Enter elements: "; vector<int>v;
        for(int i=0;i<n;i++)
        {
      int k; cin>>k;
      v.push_back(k);
     }
        cout<<endl<<"Enter element to be searched:";
        int x;
```

```
        cin>>x;
        int ans = binarySearch(v, 0, n - 1, x);
        if (ans == -1)
                cout << "Element is not present in array"<<endl;
    else
        cout << "Element is present at index " << ans<<endl;
        return 0;
}
```

OUTPUT:

```
Enter size of Array: 3
Enter elements: 1 2 3
Enter element to be searched:2
Element is present at index 1
```

EXERCISE-2:

Implement Recursive Quicksort Algorithm

PROGRAM:

```
#include <iostream>
using namespace std;
int partition(int arr[], int start, int end)
{
  int pivot = arr[start];
  int count = 0;
  for (int i = start + 1; i <= end; i++)
  {
    if (arr[i] <= pivot)
    count++;
  }
  int pivotIndex = start + count;
  swap(arr[pivotIndex], arr[start]);
  int i = start, j = end;
  while (i < pivotIndex && j > pivotIndex)
  {
    while (arr[i] <= pivot)
    {
      i++;
    }
    while (arr[j] > pivot)
    {
      j--;
    }
    if (i < pivotIndex && j > pivotIndex)
    {
     swap(arr[i++], arr[j--]);
    }
  }
  return pivotIndex;
}
```

```
void quickSort(int arr[], int start, int end)
{
  if (start >= end)
    return;
  int p = partition(arr, start, end);
  quickSort(arr, start, p - 1);
  quickSort(arr, p + 1, end);
}
void printArray(int arr[], int size)
{
  int i;
  for (i = 0; i < size; i++)
     cout<<arr[i]<<" ";
}

int main()
{
  int n;
  cout<<"Enter number of elements: "<<endl;
  cin>>n;
  int arr[n];
  cout<<"Enter elements: "<<endl;
  for(int i=0;i<n;i++)
  cin>>arr[i];
  cout<<"Given array: "<<endl;
  for(int i=0;i<n;i++)
  cout<<arr[i]<<" ";
  cout<<endl;
  quickSort(arr, 0, n - 1);
  printf("Sorted array: \n");
  printArray(arr, n);
  return 0;
}
```

OUTPUT:

```
Enter number of elements:
6
Enter elements:
2 5 10 2 5 1
Given array:
2 5 10 2 5 1
Sorted array:
1 2 2 5 5 10
```

EXERCISE-3:

Implement Recursive Mergesort Algorithm

PROGRAM:

```
#include <stdio.h>
 #include <stdlib.h>
void merge(int arr[], int l, int m, int r)
{
  int i, j, k;
  int n1 = m - l + 1;
  int n2 = r - m;
  int L[n1], R[n2];
  for (i = 0; i < n1; i++)
    L[i] = arr[l + i];
  for (j = 0; j < n2; j++)
    R[j] = arr[m + 1 + j];
  i = 0;
  j = 0;
  k = l;
  while (i < n1 && j < n2)
  {
    if (L[i] <= R[j])
    {
        arr[k] = L[i];
        i++;
    }
    else
    {
        arr[k] = R[j];
        j++;
    }
    k++;
  }

  while (i < n1)
```

```
  {
     arr[k] = L[i];
     i++;
     k++;
  }
  while (j < n2)
  {
     arr[k] = R[j];
     j++;
     k++;
  }
}
void mergeSort(int arr[], int l, int r)
{
  if (l < r)
  {
      int m = l + (r - l) / 2;
      mergeSort(arr, l, m);
      mergeSort(arr, m + 1, r);
      merge(arr, l, m, r);
  }
}
void printArray(int A[], int size)
{
   int i;
   for (i = 0; i < size; i++)
   printf("%d ", A[i]);
   printf("\n");
}
int main()
{
   int n,i;
   printf("enter n value\n");
   scanf("%d",&n);
```

```
    int a[n];
    printf("enter array to be sorted\n");
    for(i=0;i<n;i++)
    {
      scanf("%d",&a[i]);
    }
    printf("Given array is \n");
    printArray(a,n);
    mergeSort(a, 0,n-1);
    printf("\nSorted array is \n");
    printArray(a,n);
    return 0;
}
```

OUTPUT:

```
enter n value
10
enter array to be sorted
1 4 3 2 5 6 8 97 1 2
Given array is
1 4 3 2 5 6 8 97 1 2

Sorted array is
1 1 2 2 3 4 5 6 8 97
```

EXERCISE-4:

Implement Randomized Quicksort Algorithm

PROGRAM:

```
#include <iostream>
using namespace std;
int partition(int arr[], int low, int high)
{
  int pivot = arr[low];
  int i = low - 1, j = high + 1;
  while (true)
  {
   do
   {
     i++;
   } while (arr[i] < pivot);
   do
   {
     j--;
   } while (arr[j] > pivot);
   if (i >= j)
   return j;
   swap(arr[i], arr[j]);
  }
}
int partition_r(int arr[], int low, int high)
{
   srand(time(NULL));
   int random = low + rand() % (high - low);
   swap(arr[random], arr[low]);
   return partition(arr, low, high);
}
```

```
void quickSort(int arr[], int low, int high)
{
  if (low < high)
  {
     int pi = partition_r(arr, low, high);
     quickSort(arr, low, pi);
     quickSort(arr, pi + 1, high);
  }
}
void printArray(int arr[], int n)
{
   for (int i = 0; i < n; i++)
   printf("%d ", arr[i]);
   printf("\n");
}
int main()
{
   int n;
   cout<<"Enter number of elements: "<<endl;
   cin>>n;
   int arr[n];
   cout<<"Enter elements: "<<endl;
   for(int i=0;i<n;i++)
   cin>>arr[i];
   cout<<"Given array: "<<endl;
   for(int i=0;i<n;i++) cout<<arr[i]<<" ";
   cout<<endl;
   quickSort(arr, 0, n - 1);
   printf("Sorted array: \n");
   printArray(arr, n);
   return 0;
}
```

OUTPUT:

```
Enter number of elements:
7
Enter elements:
7 8 9 6 5 2 3
Given array:
7 8 9 6 5 2 3
Sorted array:
2 3 5 6 7 8 9
```

EXERCISE-5:

Find an optimal solution for a Knapsack Problem

PROGRAM:

```
#include <bits/stdc++.h>

using namespace std;

int main()

{

  int n,mx;

  cout<<"Enter number of elements: "<<endl;

  cin>>n;

  cout<<endl<<"Enter the maximum capacity:"<<endl;

  cin>>mx;

  vector<int>w,p;

  vector<double>d;

  cout<<"Enter weights: "<<endl;

  for(int i=0;i<n;i++)

  {

    int k;

    cin>>k;

    w.push_back(k);

  }

  cout<<endl<<"Enter profits: "<<endl;

  for(int i=0;i<n;i++)

  {

    int k;

    cin>>k;

    p.push_back(k);

  }

  for(int i=0;i<n;i++)

  {

   d.push_back(p[i]/(w[i]*1.0));

  }
```

```
for(int i=0;i<n;i++)
{
  for(int j=i+1;j<n;j++)
  {
    if(d[i]<d[j])
    {
      double tm=d[i]; d[i]=d[j];
      d[j]=tm;
      int t=w[i];
      w[i]=w[j];
      w[j]=t;
      t=p[i];
      p[i]=p[j];
      p[j]=t;
    }
  }

}
double profit=0;
for(int i=0;i<n;i++)
{
  if(mx<w[i])
  {
    profit+=mx*d[i];
    break;
  }
  else
  {
    profit+=p[i];
    mx-=w[i];
  }
 }
```

```
    cout<<endl<<"Maximum Profit is "<<profit<<endl;
    return 0;
}
```

OUTPUT:

```
Enter number of elements:
3
Enter the maximum capacity:
20
Enter weights:
18 15 10
Enter profits:
25 24 15
Maximum Profit is 31.5
```

EXERCISE-6:

Find the shortest path using Single Source Shortest Path Algorithm

PROGRAM:

```
#include <bits/stdc++.h>

using namespace std;

int miniDistance(int dist[], bool visited[],int V)

{

  int min = INT_MAX;

  int min_index;

  for (int v = 0; v < V; v++)

    if (visited[v] == false && dist[v] <= min)

     min = dist[v], min_index = v;

         return min_index;

}


void print(int dist[],int V)

{

  printf("Vertex Distance from Source\n");

  for (int i = 0; i < V; i++)

    cout<<i<<" "<<dist[i]<<endl;

}


void dijkstra(vector<vector<int>>graph, int src,int V)

{

   int dist[V];

   bool visited[V];

   for (int i = 0; i < V; i++)

   dist[i] = INT_MAX, visited[i] = false;

   dist[src] = 0;

   for (int i = 0; i < V - 1; i++)

   {
```

```
      int u = miniDistance(dist, visited,V);
      visited[u] = true;
      for (int v = 0; v < V; v++)
         if (!visited[v] && graph[u][v] && dist[u] != INT_MAX && dist[u] + graph[u][v] < dist[v])
                    dist[v] = dist[u] + graph[u][v];
   }
   print(dist, V);
}

int main()
{
  int V;
  cout<<"Enter number of vertices:"<<endl;
  cin>>V;
  cout<<endl<<"Enter adjacency matrix elements:"<<endl;
  vector<vector<int>>graph;
  for(int i=0;i<V;i++)
  {
    vector<int>v;
    for(int j=0;j<V;j++)
    {
      int k;
      cin>>k;
      v.push_back(k);
    }
    graph.push_back(v);
  }
  dijkstra(graph, 0,V);
  return 0;
}
```

OUTPUT:

```
Enter number of vertices:
5
Enter adjacency matrix elements:
0 10 5 0 0
0 0 0 1 0
0 3 0 9 2
0 0 0 0 0
2 0 0 6 0
Vertex Distance from Source
0 0
1 8
2 5
3 9
4 7
```

EXERCISE-7:

Implement Huffman Coding Technique

PROGRAM:

```
#include <bits/stdc++.h> using namespace std;
struct MinHeapNode
{
 char data;
 int freq;
 MinHeapNode *left, *right;
 MinHeapNode(char data, int freq)
 {
  left = right = NULL;
  this->data = data;
  this->freq = freq;
 }
};

struct compare
{
  bool operator()(MinHeapNode* l, MinHeapNode* r)
  {
    return (l->freq > r->freq);
  }
};
void printCodes(struct MinHeapNode* root, string str)
{
  if (!root) return;
  if (root->data != '$')
    cout << root->data << ": " << str << "\n";
  printCodes(root->left, str + "0");
  printCodes(root->right, str + "1");
```

```
}

void HuffmanCodes(char data[], int freq[], int size)
{
  struct MinHeapNode *left, *right, *top;
  priority_queue<MinHeapNode*, vector<MinHeapNode*>,compare> minHeap;
  for (int i = 0; i < size; ++i)
    minHeap.push(new MinHeapNode(data[i], freq[i]));
  while (minHeap.size() != 1)
  {
    left = minHeap.top();
    minHeap.pop();
    right = minHeap.top();
    minHeap.pop();
    top = new MinHeapNode('$',left->freq + right->freq);
    top->left= left;
    top->right = right;
    minHeap.push(top);
   }
   printCodes(minHeap.top(), "");
}
int main()
{
  int n;
  cout<<"Enter number of characters:"<<endl;
  cin>>n;
  char arr[n];
  int freq[n];
  cout<<endl<<"Enter Characters:"<<endl;
  for(int i=0;i<n;i++)
    cin>>arr[i];
  cout<<endl<<"Enter Frequencies"<<endl;
```

```
  for(int i=0;i<n;i++)
    cin>>freq[i];
  HuffmanCodes(arr, freq, n);
  return 0;
}
```

OUTPUT:

```
Enter number of characters:
5
Enter Characters:
s t u v z
Enter Frequencies
33 12 9 7 22
s: 0
z: 10
t: 110
v: 1110
u: 1111
```

EXERCISE-8:

Implement 0/1 Knapsack Problem

PROGRAM:

```
#include <stdio.h>
int max(int a, int b)
{
  return (a > b) ? a : b;
}

int knapsack(int weights[], int values[], int n, int capacity)
{
   int i, w;
   int dp[n + 1][capacity + 1];
   for (i = 0; i <= n; i++)
   {
      for (w = 0; w <= capacity; w++)
      {
         if (i == 0 || w == 0)
            dp[i][w] = 0;
         else if (weights[i - 1] <= w)
         {
             int p=dp[i-1][w-weights[i-1]+values[i-1];
             dp[i][w] = max(dp[i - 1][w],p);
         }
         else
             dp[i][w] = dp[i - 1][w];
      }
   }
   printf("Selected items: ");
   i = n;
   w = capacity;
   while (i > 0 && w > 0)
   {
      if (dp[i][w] != dp[i - 1][w])
```

```
        {
            printf("%d ", i - 1);
            w -= weights[i - 1];
        }
        i--;
    }
    printf("\n");
    return dp[n][capacity];
}

int main()
{
    int n,i;
    printf("Enter Number of items:");
    scanf("%d",&n);
    int weights[n],profits[n];
    printf("\nEnter Weights:");
    for(i=0;i<n;i++)
    {
      scanf("%d",&weights[i]);
    }
    printf("\nEnter profits:");
    for(i=0;i<n;i++)
    {
      scanf("%d",&profits[i]);
    }
    int capacity;
    printf("\nEnter max capacity:");
    scanf("%d",&capacity);
    int maxValue = knapsack(weights,profits, n, capacity);
    printf("Maximum value: %d\n", maxValue);
    return 0;
}
```

OUTPUT:

```
Enter Number of items:7
Enter Weights:2 3 5 7 1 4 1
Enter profits:10 5 15 7 6 18 3
Enter max capacity:15
Selected items: 5 4 2 1 0
Maximum value: 54
```

EXERCISE-9:

Find the shortest path using All Pairs Shortest Path Algorithm

PROGRAM:

```
#include <bits/stdc++.h>
using namespace std;
void printSolution(vector<vector<int>>dist,int V)
{
  cout << "The shortest distances between every pair of vertices \n";
  for (int i = 0; i < V; i++)
  {
    for (int j = 0; j < V; j++)
    {
          if (dist[i][j] == INT_MAX)
              cout << "∞"<< " ";
          else
              cout << dist[i][j] << " ";
     }
     cout << endl;
   }
}
void floydWarshall(vector<vector<int>>dist,int V)
{
   int i, j, k;
   for (k = 0; k < V; k++)
   {
      for (i = 0; i < V; i++)
      {
            for (j = 0; j < V; j++)
            {
              if (dist[i][j] > (dist[i][k] + dist[k][j]) && (dist[k][j] != INT_MAX && dist[i][k] != INT_MAX))
                      dist[i][j] = dist[i][k] + dist[k][j];
            }
      }
   }
```

```
  printSolution(dist,V);
}
int main()
{
        int V;
        cout<<"Enter number of vertices:";
        cin>>V;
        vector<vector<int>>graph;
        cout<<endl<<"Enter adjacency matrix -1 if no edge exists:";
        for(int i=0;i<V;i++)
        {
           vector<int>v;
           for(int j=0;j<V;j++)
           {
              int k; cin>>k;
              if(k==-1)
              v.push_back(INT_MAX);
              else
              v.push_back(k);
           }
           graph.push_back(v);
        }
        floydWarshall(graph,V);
        return 0;
}
```

OUTPUT:

```
Enter number of vertices:4
Enter adjacency matrix -1 if no edge exists:
0 2 3 -1
-1 0 -1 -1
-1 6 0 4
5 -1 -1 0
The shortest distances between every pair of vertices
0 2 3 7
∞ 0 ∞ ∞
9 6 0 4
5 7 8 0
```

EXERCISE-10:

Implement Travelling Salesman Problem

PROGRAM:

```
#include <bits/stdc++.h>
using namespace std;
int travllingSalesmanProblem(vector<vector<int>>graph, int s,int V)
{
  vector<int> vertex;
  for (int i = 0; i < V; i++)
    if (i != s)
          vertex.push_back(i);
  int min_path = INT_MAX;
  do
  {
      int current_pathweight = 0;
          int k = s;
          for (int i = 0; i < vertex.size(); i++)
      {
              current_pathweight += graph[k][vertex[i]];
              k = vertex[i];
          }
          current_pathweight += graph[k][s];
          min_path = min(min_path, current_pathweight);
   } while (next_permutation(vertex.begin(), vertex.end()));
   return min_path;
}
int main()
{
   vector<vector<int>>graph;
   int V,s;
   cout<<"Enter number of vertices:";
   cin>>V;
   cout<<"Enter adjacency matrix of graph:";
   for(int i=0;i<V;i++)
```

```
    {
      vector<int>v;
      for(int j=0;j<V;j++)
      {
        int k; cin>>k;
            v.push_back(k);
      }
      graph.push_back(v);
    }
    cout<<"Enter Source:";
    cin>>s;
    cout << travllingSalesmanProblem(graph, s,V) << endl;
    return 0;
}
```

OUTPUT:

```
Enter number of vertices:4
Enter adjacency matrix of graph:
0 10 15 20
10 0 35 25
15 35 0 30
20 25 30 0
Enter Source:0
80
```

EXERCISE-11:

Implement Sum of Subsets Problem

PROGRAM:

```
#include <bits/stdc++.h>
using namespace std;
int found = 0;
void PrintSubsetSum(int i, int n, vector<int>set, int targetSum,vector<int>& subset)
{
  if (targetSum == 0)
  {
    found = 1;
    cout << "[ ";
    for (int i = 0; i < subset.size(); i++)
    {
        cout << subset[i] << " ";
    }
    cout << "]";
    return;
  }
  if (i == n)
     return;
  PrintSubsetSum(i + 1, n, set, targetSum, subset);
  if (set[i] <= targetSum)
  {
     subset.push_back(set[i]);
     PrintSubsetSum(i + 1, n, set, targetSum - set[i],subset);
     subset.pop_back();
  }
}

int main()
{
   int n;
   cout<<"Enter number of elements:";
```

```
  cin>>n;
  cout<<"Enter elements:";
  vector<int> subset,set;
  for(int i=0;i<n;i++)
  {
    int k; cin>>k;
    subset.push_back(k);
  }
  cout<<"Enter target sum:";
  int targetSum;
  cin>>targetSum;
  PrintSubsetSum(0,n,subset,targetSum,set);
  if(found==0)
   cout<<"No Subset";
  return 0;
}
```

OUTPUT:

```
Enter number of elements:3
Enter elements:1 2 1
Enter target sum:3
[ 2 1 ][ 1 2 ]
```

EXERCISE-12:

Implement N-Queens Problem

PROGRAM:

```
#include <bits/stdc++.h>
using namespace std;
void printSolution(vector<vector<int>>&board,int N)
{
  for (int i = 0; i < N; i++)
  {
    for (int j = 0; j < N; j++)
         if(board[i][j])
           cout << "Q ";
         else
        cout<<". ";
         printf("\n");
  }
}

bool isSafe(vector<vector<int>>&board, int row, int col,int N)
{
  int i, j;
  for (i = 0; i < col; i++)
         if (board[row][i])
           return false;
  for (i = row, j = col; i >= 0 && j >= 0; i--, j--)
         if (board[i][j])
           return false;
  for (i = row, j = col; j >= 0 && i < N; i++, j--)
         if (board[i][j])
            return false;
  return true;
}
```

```
bool solveNQUtil(vector<vector<int>>&board, int col,int N)
{
  if (col >= N)
          return true;
  for (int i = 0; i < N; i++)
  {
    if (isSafe(board, i, col,N))
    {
          board[i][col] = 1;
      if (solveNQUtil(board, col + 1,N))
                  return true;
                  board[i][col] = 0;
    }
  }
  return false;
}
bool solveNQ()
{
  int n;
  cout<<"Enter size of board(n):";
  cin>>n;
  vector<vector<int>>board(n,vector<int>(n,0));
  if (solveNQUtil(board, 0,n) == false)
  {
          cout << "Solution does not exist";
          return false;
   }
   printSolution(board,n);
   return true;
}
int main()
{
          solveNQ();
          return 0;
```

}

OUTPUT:

```
Enter size of board(n):4
. . Q .
Q . . .
. . . Q
. Q . .
```

Longest Common Subsequence Problem :

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

int lcs(char *X, char *Y, int m, int n) {
    int L[m+1][n+1];

    for (int i = 0; i <= m; i++) {
        for (int j = 0; j <= n; j++) {
            if (i == 0 || j == 0) {
                L[i][j] = 0;
            } else if (X[i - 1] == Y[j - 1]) {
                L[i][j] = L[i - 1][j - 1] + 1;
            } else {
                L[i][j] = (L[i - 1][j] > L[i][j - 1]) ? L[i - 1][j] : L[i][j - 1];
            }
        }
    }

    return L[m][n];
}

void printLCS(char *X, char *Y, int m, int n) {
    int L[m+1][n+1];

    for (int i = 0; i <= m; i++) {
        for (int j = 0; j <= n; j++) {
            if (i == 0 || j == 0) {
                L[i][j] = 0;
            } else if (X[i - 1] == Y[j - 1]) {
                L[i][j] = L[i - 1][j - 1] + 1;
            } else {
                L[i][j] = (L[i - 1][j] > L[i][j - 1]) ? L[i - 1][j] : L[i][j - 1];
            }
        }
    }

    int index = L[m][n];
    char lcs[index + 1];
    lcs[index] = '\0';

    int i = m, j = n;
    while (i > 0 && j > 0) {
        if (X[i - 1] == Y[j - 1]) {
            lcs[index - 1] = X[i - 1];
            i--;
            j--;
            index--;
        } else if (L[i - 1][j] > L[i][j - 1]) {
```

```
            i--;
        } else {
            j--;
        }
    }

    printf("The Longest Common Subsequence is: %s\n", lcs);
}

int main() {
    char X[] = "ABCDEF";
    char Y[] = "AEBDF";

    int m = strlen(X);
    int n = strlen(Y);

    int length = lcs(X, Y, m, n);
    printf("Length of LCS: %d\n", length);

    printLCS(X, Y, m, n);

    return 0;
}
```

Output:
Length of LCS: 4
The Longest Common Subsequence is: ABDF

Heap Sort Technique :

```
#include <stdio.h>

void heapify(int arr[], int n, int i)
{
    int temp, maximum, left_index, right_index;

    maximum = i;
    right_index = 2 * i + 2;
    left_index = 2 * i + 1;

    if (left_index < n && arr[left_index] > arr[maximum])
        maximum = left_index;

    if (right_index < n && arr[right_index] > arr[maximum])
        maximum = right_index;

    if (maximum != i) {
        temp = arr[i];
        arr[i] = arr[maximum];
```

```
        arr[maximum] = temp;
        heapify(arr, n, maximum);
    }
}

void heapsort(int arr[], int n)
{
    int i, temp;

    for (i = n / 2 - 1; i >= 0; i--) {
        heapify(arr, n, i);
    }

    for (i = n - 1; i > 0; i--) {
        temp = arr[0];
        arr[0] = arr[i];
        arr[i] = temp;
        heapify(arr, i, 0);
    }
}

int main()
{
    int arr[] = { 20, 18, 5, 15, 3, 2 };
    int n = 6;

    printf("Original Array : ");
    for (int i = 0; i < n; i++) {
        printf("%d ", arr[i]);
    }

    printf("\n");
    heapsort(arr, n);

    printf("Array after performing heap sort: ");
    for (int i = 0; i < n; i++) {
        printf("%d ", arr[i]);
    }
    return 0;
}
```

Output:
Original Array : 20 18 5 15 3 2
Array after performing heap sort: 2 3 5 15 18 20

B- Tree and its operations :

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

```
#define M 4

struct BTreeNode {
    int num_keys;
    int keys[M-1];
    struct BTreeNode *children[M];
    int is_leaf;
};

struct BTreeNode *createNode(int is_leaf) {
    struct BTreeNode *newNode = (struct BTreeNode *)malloc(sizeof(struct
BTreeNode));
    if (newNode == NULL) {
        perror("Memory allocation failed");
        exit(EXIT_FAILURE);
    }
    newNode->num_keys = 0;
    newNode->is_leaf = is_leaf;
    for (int i = 0; i < M; i++) {
        newNode->children[i] = NULL;
    }
    return newNode;
}

void splitChild(struct BTreeNode *parent, int index) {
    struct BTreeNode *child = parent->children[index];
    struct BTreeNode *newNode = createNode(child->is_leaf);

    newNode->num_keys = M/2 - 1;

    for (int i = 0; i < M/2 - 1; i++) {
        newNode->keys[i] = child->keys[i + M/2];
    }

    if (!child->is_leaf) {
        for (int i = 0; i < M/2; i++) {
            newNode->children[i] = child->children[i + M/2];
        }
    }

    child->num_keys = M/2 - 1;

    for (int i = parent->num_keys; i > index; i--) {
        parent->children[i + 1] = parent->children[i];
    }

    parent->children[index + 1] = newNode;
```

```
    for (int i = parent->num_keys - 1; i >= index; i--) {
        parent->keys[i + 1] = parent->keys[i];
    }

    parent->keys[index] = child->keys[M/2 - 1];
    parent->num_keys++;
}

void insertNonFull(struct BTreeNode *node, int key) {
    int i = node->num_keys - 1;

    if (node->is_leaf) {
        while (i >= 0 && node->keys[i] > key) {
            node->keys[i + 1] = node->keys[i];
            i--;
        }
        node->keys[i + 1] = key;
        node->num_keys++;
    } else {
        while (i >= 0 && node->keys[i] > key) {
            i--;
        }
        i++;

        if (node->children[i]->num_keys == M - 1) {
            splitChild(node, i);

            if (node->keys[i] < key) {
                i++;
            }
        }
        insertNonFull(node->children[i], key);
    }
}

void insert(struct BTreeNode **root, int key) {
    struct BTreeNode *node = *root;

    if (node == NULL) {
        *root = createNode(1);
        (*root)->keys[0] = key;
        (*root)->num_keys = 1;
    } else {
        if (node->num_keys == M - 1) {
            struct BTreeNode *new_root = createNode(0);
            new_root->children[0] = node;
            splitChild(new_root, 0);
            *root = new_root;
        }
```

```
        insertNonFull(*root, key);
    }
}

void traverse(struct BTreeNode *root) {
    if (root != NULL) {
        int i;
        for (i = 0; i < root->num_keys; i++) {
            traverse(root->children[i]);
            printf("%d ", root->keys[i]);
        }
        traverse(root->children[i]);
    }
}

int main() {
    struct BTreeNode *root = NULL;

    insert(&root, 10);
    insert(&root, 20);
    insert(&root, 5);
    insert(&root, 6);
    insert(&root, 12);
    insert(&root, 30);

    printf("In-order traversal of the B-tree: ");
    traverse(root);
    printf("\n");

    return 0;
}
```

output :

```
In-order traversal of the B-tree: 5 6 10 12 20 30
```

AVL Trees and Operations :

```
/*
 AVL Tree Program in C
 */

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

// structure of the tree node
struct node
{
    int data;
```

```
    struct node* left;
    struct node* right;
    int ht;
};

// global initialization of root node
struct node* root = NULL;

// function prototyping
struct node* create(int);
struct node* insert(struct node*, int);
struct node* delete(struct node*, int);
struct node* search(struct node*, int);
struct node* rotate_left(struct node*);
struct node* rotate_right(struct node*);
int balance_factor(struct node*);
int height(struct node*);
void inorder(struct node*);
void preorder(struct node*);
void postorder(struct node*);

int main()
{
    int user_choice, data;
    char user_continue = 'y';
    struct node* result = NULL;

    while (user_continue == 'y' || user_continue == 'Y')
    {
        printf("\n\n------- AVL TREE --------\n");
        printf("\n1. Insert");
        printf("\n2. Delete");
        printf("\n3. Search");
        printf("\n4. Inorder");
        printf("\n5. Preorder");
        printf("\n6. Postorder");
        printf("\n7. EXIT");

        printf("\n\nEnter Your Choice: ");
        scanf("%d", &user_choice);

        switch(user_choice)
        {
            case 1:
                printf("\nEnter data: ");
                scanf("%d", &data);
                root = insert(root, data);
                break;
```

```
        case 2:
          printf("\nEnter data: ");
          scanf("%d", &data);
          root = delete(root, data);
          break;

        case 3:
          printf("\nEnter data: ");
          scanf("%d", &data);
          result = search(root, data);
          if (result == NULL)
          {
            printf("\nNode not found!");
          }
          else
          {
            printf("\n Node found");
          }
          break;
        case 4:
          inorder(root);
          break;

        case 5:
          preorder(root);
          break;

        case 6:
          postorder(root);
          break;

        case 7:
          printf("\n\tProgram Terminated\n");
          return 1;

        default:
          printf("\n\tInvalid Choice\n");
      }

      printf("\n\nDo you want to continue? ");
      scanf(" %c", &user_continue);
    }

    return 0;
}

// creates a new tree node
struct node* create(int data)
{
```

```
    struct node* new_node = (struct node*) malloc (sizeof(struct node));

    // if a memory error has occurred
    if (new_node == NULL)
    {
        printf("\nMemory can't be allocated\n");
        return NULL;
    }
    new_node->data = data;
    new_node->left = NULL;
    new_node->right = NULL;
    return new_node;
}

// rotates to the left
struct node* rotate_left(struct node* root)
{
    struct node* right_child = root->right;
    root->right = right_child->left;
    right_child->left = root;

    // update the heights of the nodes
    root->ht = height(root);
    right_child->ht = height(right_child);

    // return the new node after rotation
    return right_child;
}

// rotates to the right
struct node* rotate_right(struct node* root)
{
    struct node* left_child = root->left;
    root->left = left_child->right;
    left_child->right = root;

    // update the heights of the nodes
    root->ht = height(root);
    left_child->ht = height(left_child);

    // return the new node after rotation
    return left_child;
}

// calculates the balance factor of a node
int balance_factor(struct node* root)
{
    int lh, rh;
    if (root == NULL)
```

```
        return 0;
    if (root->left == NULL)
        lh = 0;
    else
        lh = 1 + root->left->ht;
    if (root->right == NULL)
        rh = 0;
    else
        rh = 1 + root->right->ht;
    return lh - rh;
}

// calculate the height of the node
int height(struct node* root)
{
    int lh, rh;
    if (root == NULL)
    {
        return 0;
    }
    if (root->left == NULL)
        lh = 0;
    else
        lh = 1 + root->left->ht;
    if (root->right == NULL)
        rh = 0;
    else
        rh = 1 + root->right->ht;

    if (lh > rh)
        return (lh);
    return (rh);
}

// inserts a new node in the AVL tree
struct node* insert(struct node* root, int data)
{
    if (root == NULL)
    {
        struct node* new_node = create(data);
        if (new_node == NULL)
        {
            return NULL;
        }
        root = new_node;
    }
    else if (data > root->data)
    {
        // insert the new node to the right
```

```
        root->right = insert(root->right, data);

        // tree is unbalanced, then rotate it
        if (balance_factor(root) == -2)
        {
            if (data > root->right->data)
            {
                root = rotate_left(root);
            }
            else
            {
                root->right = rotate_right(root->right);
                root = rotate_left(root);
            }
        }
    }
    else
    {
        // insert the new node to the left
        root->left = insert(root->left, data);

        // tree is unbalanced, then rotate it
        if (balance_factor(root) == 2)
        {
            if (data < root->left->data)
            {
                root = rotate_right(root);
            }
            else
            {
                root->left = rotate_left(root->left);
                root = rotate_right(root);
            }
        }
    }
    // update the heights of the nodes
    root->ht = height(root);
    return root;
}

// deletes a node from the AVL tree
struct node * delete(struct node *root, int x)
{
    struct node * temp = NULL;

    if (root == NULL)
    {
        return NULL;
    }
```

```
if (x > root->data)
{
   root->right = delete(root->right, x);
   if (balance_factor(root) == 2)
   {
      if (balance_factor(root->left) >= 0)
      {
         root = rotate_right(root);
      }
      else
      {
         root->left = rotate_left(root->left);
         root = rotate_right(root);
      }
   }
}
else if (x < root->data)
{
   root->left = delete(root->left, x);
   if (balance_factor(root) == -2)
   {
      if (balance_factor(root->right) <= 0)
      {
         root = rotate_left(root);
      }
      else
      {
         root->right = rotate_right(root->right);
         root = rotate_left(root);
      }
   }
}
else
{
   if (root->right != NULL)
   {
      temp = root->right;
      while (temp->left != NULL)
         temp = temp->left;

      root->data = temp->data;
      root->right = delete(root->right, temp->data);
      if (balance_factor(root) == 2)
      {
         if (balance_factor(root->left) >= 0)
         {
            root = rotate_right(root);
         }
```

```
            else
            {
               root->left = rotate_left(root->left);
               root = rotate_right(root);
            }
         }
      }
      else
      {
         return (root->left);
      }
   }
   root->ht = height(root);
   return (root);
}

// search a node in the AVL tree
struct node* search(struct node* root, int key)
{
   if (root == NULL)
   {
      return NULL;
   }

   if(root->data == key)
   {
      return root;
   }

   if(key > root->data)
   {
      search(root->right, key);
   }
   else
   {
      search(root->left, key);
   }
}

// inorder traversal of the tree
void inorder(struct node* root)
{
   if (root == NULL)
   {
      return;
   }

   inorder(root->left);
   printf("%d ", root->data);
```

```
    inorder(root->right);
}

// preorder traversal of the tree
void preorder(struct node* root)
{
    if (root == NULL)
    {
        return;
    }

    printf("%d ", root->data);
    preorder(root->left);
    preorder(root->right);
}

// postorder traversal of the tree
void postorder(struct node* root)
{
    if (root == NULL)
    {
        return;
    }

    postorder(root->left);
    postorder(root->right);
    printf("%d ", root->data);
}
```

output:

```
------- AVL TREE --------
1. Insert
2. Delete
3. Search
4. Inorder
5. Preorder
6. Postorder
7. EXIT
Enter Your Choice: 1
Enter data: 2
Do you want to continue? y
```