DESIGN & ANALYSIS OF ALGORITHM LAB
(ACS551)

LIST OF EXPERIMENTS

1. Implementation of Quick Sort and Merge Sort.
2. Implementation of Linear-time Sorting Algorithms.
3. Implementation of Red-Black Tree operations.
4. Implementation of Binomial Heap operations.
5. Implementation of an application of Dynamic Programming.
6. Implementation of an application of Greedy Algorithm.
7. Implementation of Minimum Spanning Tree Algorithm.
8. Implementation of Single-pair shortest path Algorithm.
9. Implementation of All-pair shortest path Algorithm.
10. Implementation of String Matching Algorithm.

1.Implementation of Quick Sort and Merge Sort.

// Quick sort in C

```
#include <stdio.h>
// function to swap elements void
swap(int *a, int *b) {
 int t = *a;
 *a = *b;
 *b = t;
}
// function to find the partition position int
partition(int array[], int low, int high) {
  // select the rightmost element as pivot
 int pivot = array[high];
  // pointer for greater element
 int i = (low - 1);
 // traverse each element of the array
 // compare them with the pivot for
 (int j = low; j < high; j++) {
  if (array[j] <= pivot) {
       // if element smaller than pivot is found
   // swap it with the greater element pointed by i i++;
   // swap element at i with element at j
   swap(&array[i], &array[j]);
  }
 }
 // swap the pivot element with the greater element at i
 swap(&array[i + 1], &array[high]);
 // return the partition point
 return (i + 1);
```

```
}
void quickSort(int array[], int low, int high) { if
  (low < high) {
    // find the pivot element such that
    // elements smaller than pivot are on left of pivot
    // elements greater than pivot are on right of pivot int
    pi = partition(array, low, high);
    // recursive call on the left of pivot
    quickSort(array, low, pi - 1);
    // recursive call on the right of pivot
    quickSort(array, pi + 1, high);
  }
}
// function to print array elements
void printArray(int array[], int size) {
for (int i = 0; i < size; ++i) {
    printf("%d  ", array[i]);
  }
  printf("\n");
}
// main function
int main() {
  int data[] = {8, 7, 2, 1, 0, 9, 6};
  int n = sizeof(data) / sizeof(data[0]);
  printf("Unsorted Array\n");
  printArray(data, n);
  // perform quicksort on data
  quickSort(data, 0, n - 1);
```

```
  printf("Sorted array in ascending order: \n");
  printArray(data, n);
}
```

// Merge sort in C

```
#include <stdio.h>
// Merge two subarrays L and M into arr
void merge(int arr[], int p, int q, int r) {
  // Create L ← A[p..q] and M ← A[q+1..r]
  int n1 = q - p + 1;
  int n2 = r - q;
  int L[n1], M[n2];
  for (int i = 0; i < n1; i++)
    L[i] = arr[p + i];
  for (int j = 0; j < n2; j++)
    M[j] = arr[q + 1 + j];
  // Maintain current index of sub-arrays and main array
  int i, j, k;
  i = 0;
  j = 0;
  k = p;
  // Until we reach either end of either L or M, pick larger among
  // elements L and M and place them in the correct position at A[p..r]
  while (i < n1 && j < n2) {
    if (L[i] <= M[j]) {
      arr[k] = L[i];
      i++;
    } else {
```

```
      arr[k] = M[j];

      j++;
    }
    k++;
  }
  // When we run out of elements in either L or M,
  // pick up the remaining elements and put in A[p..r]
  while (i < n1) {
    arr[k] = L[i];
    i++;
    k++;
  }
  while (j < n2) {
    arr[k] = M[j];
    j++;
    k++;
  }
}
// Divide the array into two subarrays, sort them and merge them
void mergeSort(int arr[], int l, int r) {
  if (l < r) {
    // m is the point where the array is divided into two subarrays
    int m = l + (r - l) / 2;
    mergeSort(arr, l, m);
    mergeSort(arr, m + 1, r);
    // Merge the sorted subarrays
    merge(arr, l, m, r);
  }
}

// Print the array
```

```
void printArray(int arr[], int size) {
  for (int i = 0; i < size; i++)
  printf("%d ", arr[i]);
  printf("\n");
}
// Driver program
int main() {
  int arr[] = {6, 5, 12, 10, 9, 1};
  int size = sizeof(arr) / sizeof(arr[0]);

  printf("Array Before
  MergeSort\n");

  printArray(arr, size);

  mergeSort(arr, 0, size - 1);
  printf("Sorted array: \n");
  printArray(arr, size);
}
```

2. Implementation of Linear-time Sorting Algorithms.

// Counting sort: an example of Linear-Time Sorting Algorithm.

```
#include <stdio.h>
 void countingSort(int array[], int size) {
  int output[10];
  // Find the largest element of the array
  int max = array[0];
  for (int i = 1; i < size; i++) {
   if (array[i] > max)
    max = array[i];
  }
  // The size of count must be at least (max+1) but
  // we cannot declare it as int count(max+1) in C as
  // it does not support dynamic memory allocation.
  // So, its size is provided statically.
  int count[10];
  // Initialize count array with all zeros.
  for (int i = 0; i <= max; ++i) {
   count[i] = 0;
  }
  // Store the count of each element
  for (int i = 0; i < size; i++) {

   count[array[i]]++;
  }
  // Store the cumulative count of each array
  for (int i = 1; i <= max; i++) {
   count[i] += count[i - 1];
  }
```

```
    // Find the index of each element of the original array in count array, and
    // place the elements in output array
    for (int i = size - 1; i >= 0; i--) {
    output[count[array[i]] - 1] = array[i];
    count[array[i]]--;
    }
    // Copy the sorted elements into original array
    for (int i = 0; i < size; i++) {
      array[i] = output[i];
    }
}
// Function to print an array
void printArray(int array[], int size) { for (int i = 0; i < size; ++i) { printf("%d ", array[i]);
}
printf("\n");
}
int main() {
int array[] = {4, 2, 2, 8, 3, 3, 1};
int n = sizeof(array) / sizeof(array[0]);
countingSort(array, n);
printf(“Sorted Array:”);
 printArray(array, n);
}
```

3. Implementation of Red-Black Tree operations.

//Red Black Tree Operations

```
#include <stdio.h>
#include <stdlib.h>
enum nodeColor {
RED,
  BLACK
};


struct rbNode {
  int data, color;
  struct rbNode *link[2];
};

struct rbNode *root = NULL;

// Create a red-black tree
struct rbNode *createNode(int data) {
  struct rbNode *newnode;
  newnode = (struct rbNode *)malloc(sizeof(struct rbNode));
  newnode->data = data;
  newnode->color = RED;
  newnode->link[0] = newnode->link[1] = NULL;
  return newnode;
}

// Insert an node
void insertion(int data) {
  struct rbNode *stack[98], *ptr, *newnode, *xPtr, *yPtr;
  int dir[98], ht = 0, index;
  ptr = root;
  if (!root) {
    root = createNode(data);
    return;
  }

  stack[ht] = root;
  dir[ht++] = 0;
  while (ptr != NULL) {
    if (ptr->data == data) {
      printf("Duplicates Not Allowed!!\n");
      return;
    }

    index = (data - ptr->data) > 0 ? 1 : 0;
    stack[ht] = ptr;
    ptr = ptr->link[index];
    dir[ht++] = index;
```

```
}

stack[ht - 1]->link[index] = newnode = createNode(data);
while ((ht >= 3) && (stack[ht - 1]->color == RED)) {
  if (dir[ht - 2] == 0) {
    yPtr = stack[ht - 2]->link[1];
    if (yPtr != NULL && yPtr->color == RED) {
      stack[ht - 2]->color = RED;
      stack[ht - 1]->color = yPtr->color = BLACK;
      ht = ht - 2;
    } else {
      if (dir[ht - 1] == 0) {
        yPtr = stack[ht - 1];
      } else {
        xPtr = stack[ht - 1];
        yPtr = xPtr->link[1];
        xPtr->link[1] = yPtr->link[0];
        yPtr->link[0] = xPtr;
        stack[ht - 2]->link[0] = yPtr;
      }

      xPtr = stack[ht - 2];
      xPtr->color = RED;
      yPtr->color = BLACK;
      xPtr->link[0] = yPtr->link[1];
      yPtr->link[1] = xPtr;
      if (xPtr == root) {
        root = yPtr;
      } else {
        stack[ht - 3]->link[dir[ht - 3]] = yPtr;
      }

      break;
    }

  } else {
    yPtr = stack[ht - 2]->link[0];
    if ((yPtr != NULL) && (yPtr->color == RED)) {
      stack[ht - 2]->color = RED;
      stack[ht - 1]->color = yPtr->color = BLACK;
      ht = ht - 2;
    } else {
      if (dir[ht - 1] == 1) {
        yPtr = stack[ht - 1];
      } else {
        xPtr = stack[ht - 1];
        yPtr = xPtr->link[0];
        xPtr->link[0] = yPtr->link[1];
        yPtr->link[1] = xPtr;
        stack[ht - 2]->link[1] = yPtr;
      }

      xPtr = stack[ht - 2];
      yPtr->color = BLACK;
```

```c
        xPtr->color = RED;
        xPtr->link[1] = yPtr->link[0];
        yPtr->link[0] = xPtr;
        if (xPtr == root) {
          root = yPtr;
        } else {
          stack[ht - 3]->link[dir[ht - 3]] = yPtr;
        }
        break;
      }
    }
  }
  root->color = BLACK;
}

// Delete a node
void deletion(int data) {
  struct rbNode *stack[98], *ptr, *xPtr, *yPtr;
  struct rbNode *pPtr, *qPtr, *rPtr;
  int dir[98], ht = 0, diff, i;
  enum nodeColor color;
  if (!root) {
    printf("Tree not available\n");
    return;
  }

  ptr = root;
  while (ptr != NULL) {
    if ((data - ptr->data) == 0)
      break;
    diff = (data - ptr->data) > 0 ? 1 : 0;
    stack[ht] = ptr;
    dir[ht++] = diff;
    ptr = ptr->link[diff];
  }

  if (ptr->link[1] == NULL) {
    if ((ptr == root) && (ptr->link[0] == NULL)) {
      free(ptr);
      root = NULL;
    } else if (ptr == root) {
      root = ptr->link[0];
      free(ptr);
    } else {
      stack[ht - 1]->link[dir[ht - 1]] = ptr->link[0];
    }
  } else {
```

```
      xPtr = ptr->link[1];
      if (xPtr->link[0] == NULL) {
        xPtr->link[0] = ptr->link[0];
        color = xPtr->color;
        xPtr->color = ptr->color;
        ptr->color = color;

        if (ptr == root) {
          root = xPtr;
        } else {
          stack[ht - 1]->link[dir[ht - 1]] = xPtr;
        }

        dir[ht] = 1;
        stack[ht++] = xPtr;
      } else {
        i = ht++;
        while (1) {
  dir[ht] = 0; stack[ht++] = xPtr;

  yPtr = xPtr->link[0];

           if (!yPtr->link[0])
          break;
          xPtr = yPtr;
        }


        dir[i] = 1;
        stack[i] = yPtr;
        if (i > 0)
          stack[i - 1]->link[dir[i - 1]] = yPtr;

        yPtr->link[0] = ptr->link[0];

        xPtr->link[0] = yPtr->link[1];
        yPtr->link[1] = ptr->link[1];

        if (ptr == root) {
          root = yPtr;
        }


        color = yPtr->color;
        yPtr->color = ptr->color;
        ptr->color = color;
      }

    }

    if (ht < 1)
      return;

    if (ptr->color == BLACK) {
```

```
while (1) {
 pPtr = stack[ht - 1]->link[dir[ht - 1]];
 if (pPtr && pPtr->color == RED) {
 pPtr->color = BLACK;
  break;
 }


 if (ht < 2)
  break;

 if (dir[ht - 2] == 0) {
  rPtr = stack[ht - 1]->link[1];

  if (!rPtr)
   break;

  if (rPtr->color == RED) {
   stack[ht - 1]->color = RED;
   rPtr->color = BLACK;
   stack[ht - 1]->link[1] = rPtr->link[0];
   rPtr->link[0] = stack[ht - 1];

   if (stack[ht - 1] == root) {
    root = rPtr;
   } else {
    stack[ht - 2]->link[dir[ht - 2]] = rPtr;
   }
   dir[ht] = 0;
   stack[ht] = stack[ht - 1];
   stack[ht - 1] = rPtr;
   ht++;

   rPtr = stack[ht - 1]->link[1];
  }


  if ((!rPtr->link[0] || rPtr->link[0]->color == BLACK) &&
   (!rPtr->link[1] || rPtr->link[1]->color == BLACK)) {
   rPtr->color = RED;
  } else {
   if (!rPtr->link[1] || rPtr->link[1]->color == BLACK) {
    qPtr = rPtr->link[0];
    rPtr->color = RED;
    qPtr->color = BLACK;
    rPtr->link[0] = qPtr->link[1];
    qPtr->link[1] = rPtr;
    rPtr = stack[ht - 1]->link[1] = qPtr;
   }
   rPtr->color = stack[ht - 1]->color;
   stack[ht - 1]->color = BLACK;
```

```
    rPtr->link[1]->color = BLACK;
    stack[ht - 1]->link[1] = rPtr->link[0];
    rPtr->link[0] = stack[ht - 1];
    if (stack[ht - 1] == root) {
      root = rPtr;
    } else {
      stack[ht - 2]->link[dir[ht - 2]] = rPtr;
    }
    break;
  }
} else {
  rPtr = stack[ht - 1]->link[0];
  if (!rPtr)
    break;

  if (rPtr->color == RED) {
    stack[ht - 1]->color = RED;
    rPtr->color = BLACK;
    stack[ht - 1]->link[0] = rPtr->link[1];
    rPtr->link[1] = stack[ht - 1];

    if (stack[ht - 1] == root) {
      root = rPtr;
    } else {
      stack[ht - 2]->link[dir[ht - 2]] = rPtr;
    }
    dir[ht] = 1;
    stack[ht] = stack[ht - 1];
    stack[ht - 1] = rPtr;
    ht++;

    rPtr = stack[ht - 1]->link[0];
  }
  if ((!rPtr->link[0] || rPtr->link[0]->color == BLACK) &&
    (!rPtr->link[1] || rPtr->link[1]->color == BLACK)) {
    rPtr->color = RED;
  } else {
    if (!rPtr->link[0] || rPtr->link[0]->color == BLACK) {
      qPtr = rPtr->link[1];
      rPtr->color = RED;
      qPtr->color = BLACK;
      rPtr->link[1] = qPtr->link[0];
      qPtr->link[0] = rPtr;
      rPtr = stack[ht - 1]->link[0] = qPtr;
    }
    rPtr->color = stack[ht - 1]->color;
    stack[ht - 1]->color = BLACK;
    rPtr->link[0]->color = BLACK;
    stack[ht - 1]->link[0] = rPtr->link[1];
```

```
          rPtr->link[1] = stack[ht - 1];
          if (stack[ht - 1] == root) {
            root = rPtr;
          } else {
            stack[ht - 2]->link[dir[ht - 2]] = rPtr;
          }

          break;
        }

      }

      ht--;
    }

  }

}
```

```
// Print the inorder traversal of the tree


void inorderTraversal(struct rbNode *node) {
  if (node) {
    inorderTraversal(node->link[0]);
    printf("%d  ", node->data);
    inorderTraversal(node->link[1]);
  }

  return;
}


// Driver code
int main() {
int ch, data;
while (1) {
    printf("1. Insertion\t2. Deletion\n");
    printf("3. Traverse\t4. Exit");
    printf("\nEnter your choice:");
    scanf("%d", &ch);
    switch (ch) {
      case 1:
        printf("Enter the element to insert:");
        scanf("%d", &data);
        insertion(data);
        break;
      case 2:
        printf("Enter the element to delete:");
        scanf("%d", &data);
        deletion(data);
        break;
      case 3:
        inorderTraversal(root);
```

```
        printf("\n");
        break;
      case 4:
        exit(0);
      default:
        printf("Not available\n");
        break;
    }

    printf("\n");
  }

  return 0;
}
```

4. Implementation of Binomial Heap operations.

Binomial Heap Operation

```
// BinomialHeap
#include<stdio.h>
#include<malloc.h>
struct node {
 int n;
 int degree;
 struct node* parent;
 struct node* child;
 struct node* sibling;
};
struct node* MAKE_bin_HEAP();
int bin_LINK(struct node*, struct node*);
struct node* CREATE_NODE(int);
struct node* bin_HEAP_UNION(struct node*, struct node*);
struct node* bin_HEAP_INSERT(struct node*, struct node*);
struct node* bin_HEAP_MERGE(struct node*, struct node*);
struct node* bin_HEAP_EXTRACT_MIN(struct node*);
int REVERT_LIST(struct node*);
int DISPLAY(struct node*);
struct node* FIND_NODE(struct node*, int);
int bin_HEAP_DECREASE_KEY(struct node*, int, int);
int bin_HEAP_DELETE(struct node*, int);
int count = 1;
struct node* MAKE_bin_HEAP() {
 struct node* np;
 np = NULL;
 return np;
}
struct node * H = NULL;
struct node *Hr = NULL;
int bin_LINK(struct node* y, struct node* z) {
 y->parent = z;
 y->sibling = z->child;
 z->child = y;
 z->degree = z->degree + 1;
}
struct node* CREATE_NODE(int k) {
 struct node* p;
//new node;
 p = (struct node*) malloc(sizeof(struct node));
 p->n = k;
 return p;
}
struct node* bin_HEAP_UNION(struct node* H1, struct node* H2) {
```

```
 struct node* prev_x;
 struct node* next_x;
 struct node* x;
 struct node* H = MAKE_bin_HEAP();
 H = bin_HEAP_MERGE(H1, H2);
 if (H == NULL)
 return H;
 prev_x = NULL;
 x = H;
 next_x = x->sibling;
 while (next_x != NULL) {
 if ((x->degree != next_x->degree) || ((next_x->sibling != NULL)
 && (next_x->sibling)->degree == x->degree)) {
 prev_x = x;
 x = next_x;
 } else {
 if (x->n <= next_x->n) {
   x->sibling = next_x->sibling;
 bin_LINK(next_x, x);
 } else {
 if (prev_x == NULL)
 H = next_x;
 else
 prev_x->sibling = next_x;
 bin_LINK(x, next_x);
 x = next_x;
 }
 }
 next_x = x->sibling;
 }
 return H;
}
struct node* bin_HEAP_INSERT(struct node* H, struct node* x) {
 struct node* H1 = MAKE_bin_HEAP();
 x->parent = NULL;
x->child = NULL;
 x->sibling = NULL;
 x->degree = 0;
 H1 = x;
 H = bin_HEAP_UNION(H, H1);
 return H;
}
struct node* bin_HEAP_MERGE(struct node* H1, struct node* H2) {
 struct node* H = MAKE_bin_HEAP();
 struct node* y;
 struct node* z;
 struct node* a;
 struct node* b;
```

```
 y = H1;
 z = H2;
 if (y != NULL) {
 if (z != NULL && y->degree <= z->degree)
 H = y;
 else if (z != NULL && y->degree > z->degree)
 /* need some modifications here;the first and the else conditions can be merged together!!!!
*/
 H = z;
 else
 H = y;
 } else
 H = z;
 while (y != NULL && z != NULL) {
 if (y->degree < z->degree) {
y = y->sibling;
 } else if (y->degree == z->degree) {
 a = y->sibling;
 y->sibling = z;
 y = a;
 } else {
 b = z->sibling;
 z->sibling = y;
 z = b;
 }
 }
 return H;
}
int DISPLAY(struct node* H) {
 struct node* p;
 if (H == NULL) {
 printf("\nHEAP EMPTY");
 return 0;
 }
 printf("\nTHE ROOT NODES ARE:-\n");
 p = H;
 while (p != NULL) {
 printf("%d", p->n);
 if (p->sibling != NULL)
 printf("-->");
 p = p->sibling;
 }
 printf("\n");
}
struct node* bin_HEAP_EXTRACT_MIN(struct node* H1) {
 int min;
 struct node* t = NULL;
 struct node* x = H1;
```

```
 struct node *Hr;
 struct node* p;
 Hr = NULL;
 if (x == NULL) {
 printf("\nNOTHING TO EXTRACT");
 return x;
 }
// int min=x->n;
 p = x;
 while (p->sibling != NULL) {
 if ((p->sibling)->n < min) {
 min = (p->sibling)->n;
 t = p;
 x = p->sibling;
 }
 p = p->sibling;
 }
 if (t == NULL && x->sibling == NULL)
 H1 = NULL;
 else if (t == NULL)
 H1 = x->sibling;
 else if (t->sibling == NULL)
 t = NULL;
 else
 t->sibling = x->sibling;
 if (x->child != NULL) {
 REVERT_LIST(x->child);
 (x->child)->sibling = NULL;
 }
 H = bin_HEAP_UNION(H1, Hr);
 return x;
}
int REVERT_LIST(struct node* y) {
 if (y->sibling != NULL) {
 REVERT_LIST(y->sibling);
 (y->sibling)->sibling = y;
 } else {
 Hr = y;
 }
}
struct node* FIND_NODE(struct node* H, int k) {
 struct node* x = H;
 struct node* p = NULL;
 if (x->n == k) {
 p = x;
 return p;
 }
 if (x->child != NULL && p == NULL) {
```

```
 p = FIND_NODE(x->child, k);
 }
 if (x->sibling != NULL && p == NULL) {
 p = FIND_NODE(x->sibling, k);
 }
 return p;
}
int bin_HEAP_DECREASE_KEY(struct node* H, int i, int k) {
 int temp;
 struct node* p;
 struct node* y;
 struct node* z;
 p = FIND_NODE(H, i);
 if (p == NULL) {
 printf("\nINVALID CHOICE OF KEY TO BE REDUCED");
 return 0;
 }
 if (k > p->n) {
 printf("\nSORY!THE NEW KEY IS GREATER THAN CURRENT ONE");
 return 0;
 }
 p->n = k;
 y = p;
 z = p->parent;
 while (z != NULL && y->n < z->n) {
 temp = y->n;
 y->n = z->n;
 z->n = temp;
 y = z;
 z = z->parent;
 }
 printf("\nKEY REDUCED SUCCESSFULLY!");
 }
int bin_HEAP_DELETE(struct node* H, int k) {
 struct node* np;
 if (H == NULL) {
 printf("\nHEAP EMPTY");
 return 0;
 }
 bin_HEAP_DECREASE_KEY(H, k, -1000);
 np = bin_HEAP_EXTRACT_MIN(H);
 if (np != NULL)
 printf("\nNODE DELETED SUCCESSFULLY");
}
int main() {
 int i, n, m, l;
 struct node* p;
 struct node* np;
```

```
char ch;
printf("\nENTER THE NUMBER OF ELEMENTS:");
scanf("%d", &n);
printf("\nENTER THE ELEMENTS:\n");
for (i = 1; i <= n; i++) {
scanf("%d", &m);
np = CREATE_NODE(m);
H = bin_HEAP_INSERT(H, np);
}
DISPLAY(H);
do {
printf("\nMENU:-\n");
printf("\n1)INSERT AN ELEMENT\n2)EXTRACT THE MINIMUM KEY NODE\n3)DECREASE A NODEKEY\n
4)DELETE A NODE\n5)QUIT\n");
scanf("%d", &l);
switch (l) {
case 1:
do {
printf("\nENTER THE ELEMENT TO BE INSERTED:");
scanf("%d", &m);
p = CREATE_NODE(m);
H = bin_HEAP_INSERT(H, p);
printf("\nNOW THE HEAP IS:\n");
DISPLAY(H);
printf("\nINSERT MORE(N/Y)= \n");
fflush(stdin);
scanf("%c", &ch);
} while (ch == 'Y' || ch == 'y');
break;
case 2:
do {
printf("\nEXTRACTING THE MINIMUM KEY NODE");
p = bin_HEAP_EXTRACT_MIN(H);
if (p != NULL)
printf("\nTHE EXTRACTED NODE IS %d", p->n);
printf("\nNOW THE HEAP IS:\n");
DISPLAY(H);
printf("\nEXTRACT MORE(N/Y)\n");
fflush(stdin);
scanf("%c", &ch);
} while (ch == 'Y' || ch == 'y');
break;
case 3:
do {
printf("\nENTER THE KEY OF THE NODE TO BE DECREASED:");
scanf("%d", &m);
printf("\nENTER THE NEW KEY : ");
scanf("%d", &l);
```

```
bin_HEAP_DECREASE_KEY(H, m, l);
printf("\nNOW THE HEAP IS:\n");
DISPLAY(H);
printf("\nDECREASE MORE(N/Y)\n");
fflush(stdin);
scanf("%c", &ch);
} while (ch == 'Y' || ch == 'y');
break;
case 4:
do {
printf("\nENTER THE KEY TO BE DELETED: ");
scanf("%d", &m);
bin_HEAP_DELETE(H, m);
printf("\nDELETE MORE(N/Y)\n");
fflush(stdin);
scanf("%c", &ch);
} while (ch == 'y' || ch == 'Y');
break;
case 5:
printf("\nTHANK U SIR\n");
break;
default:
printf("\nINVALID ENTRY...TRY AGAIN....\n");
}
} while (l != 5);
}
```

5. Implementation of an application of Dynamic Programming.

0/1 Knapsack problem program

//DP is an algorithmic technique used in computer science and mathematics to solve complex problems by breaking them down into smaller overlapping subproblems.

```
#include <stdio.h>

// A utility function that returns
// maximum of two integers
int max(int a, int b) { return (a > b) ? a : b; }
// Returns the maximum value that can be
// put in a knapsack of capacity W
int knapSack(int W, int wt[], int val[], int n)
{
// Base Case
if (n == 0 || W == 0)
return 0;
// If weight of the nth item is more than
// Knapsack capacity W, then this item cannot
// be included in the optimal solution
if (wt[n - 1] > W)
return knapSack(W, wt, val, n - 1);

// Return the maximum of two cases:
// (1) nth item included
// (2) not included
else
return max( val[n - 1] + knapSack(W - wt[n - 1], wt, val, n - 1),
knapSack(W, wt, val, n - 1));
}

// Driver code
int main()
{
int val[] = { 60, 100, 120 };
int wt[] = { 10, 20, 30 };
int W = 50;

int n = sizeof(val) / sizeof(val[0]);
printf("%d", knapSack(W, wt, val, n));
return 0;
}
```

6. Implementation of an application of Greedy Algorithm.

```
#include <stdio.h>
#include <stdlib.h>
#define MAX_TREE_HT 50
struct MinHNode {
char item; unsigned freq;
struct MinHNode *left, *right;
};

struct MinHeap { unsigned size; unsigned capacity;
struct MinHNode **array;
};

// Create nodes
struct MinHNode *newNode(char item, unsigned freq) {
struct MinHNode *temp = (struct MinHNode *)malloc(sizeof(struct MinHNode));

temp->left = temp->right = NULL; temp->item = item;
temp->freq = freq;

return temp;
}

// Create min heap
struct MinHeap *createMinH(unsigned capacity) {
struct MinHeap *minHeap = (struct MinHeap *)malloc(sizeof(struct MinHeap)); minHeap->size = 0;
minHeap->capacity = capacity;

minHeap->array = (struct MinHNode **)malloc(minHeap->capacity * sizeof(struct MinHNode *)); return
minHeap;
}
// Function to swap
void swapMinHNode(struct MinHNode **a, struct MinHNode **b) {

struct MinHNode *t = *a;
*a = *b;
*b = t;
}
// Heapify
void minHeapify(struct MinHeap *minHeap, int idx) { int smallest = idx;
int left = 2 * idx + 1; int right = 2 * idx + 2;

if (left < minHeap->size && minHeap->array[left]->freq < minHeap->array[smallest]->freq) smallest = left;
```

```
if (right < minHeap->size && minHeap->array[right]->freq < minHeap->array[smallest]->freq) smallest =
right;

if (smallest != idx) {
swapMinHNode(&minHeap->array[smallest], &minHeap->array[idx]); minHeapify(minHeap, smallest);
}
}

// Check if size if 1
int checkSizeOne(struct MinHeap *minHeap) { return (minHeap->size == 1);
}

// Extract min
struct MinHNode *extractMin(struct MinHeap *minHeap) { struct MinHNode *temp = minHeap-
>array[0];
minHeap->array[0] = minHeap->array[minHeap->size - 1];

--minHeap->size; minHeapify(minHeap, 0);
return temp;
}

// Insertion function
void insertMinHeap(struct MinHeap *minHeap, struct MinHNode *minHeapNode) {
++minHeap->size;
int i = minHeap->size - 1;

while (i && minHeapNode->freq < minHeap->array[(i - 1) / 2]->freq) { minHeap->array[i] = minHeap-
>array[(i - 1) / 2];
i = (i - 1) / 2;
}

minHeap->array[i] = minHeapNode;
}
// Print the array
void printArray(int arr[], int n) { int i;
for (i = 0; i < n; ++i) printf("%d", arr[i]);

printf("\n");
}
void buildMinHeap(struct MinHeap *minHeap) { int n = minHeap->size - 1;
int i;

for (i = (n - 1) / 2; i >= 0; --i) minHeapify(minHeap, i);
```

```
}

int isLeaf(struct MinHNode *root) { return !(root->left) && !(root->right);
}
struct MinHeap *createAndBuildMinHeap(char item[], int freq[], int size) { struct MinHeap *minHeap =
createMinH(size);

for (int i = 0; i < size; ++i)
minHeap->array[i] = newNode(item[i], freq[i]);

minHeap->size = size; buildMinHeap(minHeap);

return minHeap;
}

struct MinHNode *buildHuffmanTree(char item[], int freq[], int size) { struct MinHNode *left, *right,
*top;
struct MinHeap *minHeap = createAndBuildMinHeap(item, freq, size);

while (!checkSizeOne(minHeap)) { left = extractMin(minHeap); right = extractMin(minHeap);
top = newNode('$', left->freq + right->freq); top->left = left;
top->right = right;

insertMinHeap(minHeap, top);
}
return extractMin(minHeap);
}
void printHCodes(struct MinHNode *root, int arr[], int top) { if (root->left) {
arr[top] = 0;

printHCodes(root->left, arr, top + 1);
}
if (root->right) { arr[top] = 1;
printHCodes(root->right, arr, top + 1);
}
if (isLeaf(root)) {
printf(" %c | ", root->item);
printArray(arr, top);
}
}

// Wrapper function
void HuffmanCodes(char item[], int freq[], int size) {
struct MinHNode *root = buildHuffmanTree(item, freq, size); int arr[MAX_TREE_HT], top = 0;
```

```
printHCodes(root, arr, top);
}


int main() {
char arr[] = {'A', 'B', 'C', 'D'};
int freq[] = {5, 1, 6, 3};

int size = sizeof(arr) / sizeof(arr[0]);
printf(" Char | Huffman code "); printf("\n \n");

HuffmanCodes(arr, freq, size);
}
```

7. Implementation of Minimum Spanning Tree Algorithm.

Minimum Spanning Tree using Prim's Algorithm

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

#define V 5

// Function to find the vertex with the minimum key value
int minKey(int key[], int mstSet[]) {
   int min = INT_MAX, min_index;

   for (int v = 0; v < V; v++)
      if (mstSet[v] == 0 && key[v] < min)
         min = key[v], min_index = v;

   return min_index;
}

// Function to print the constructed MST
void printMST(int parent[], int n, int graph[V][V]) {
   printf("Edge   Weight\n");
   for (int i = 1; i < V; i++)
      printf("%d - %d    %d \n", parent[i], i, graph[i][parent[i]]);
}

// Function to construct and print MST using Prim's algorithm
void primMST(int graph[V][V]) {
   int parent[V]; // Array to store the constructed MST
   int key[V];    // Key values used to pick minimum weight edge
   int mstSet[V]; // To represent set of vertices included in MST

   // Initialize all keys as infinite and mstSet[] as false
   for (int i = 0; i < V; i++) {
      key[i] = INT_MAX;
      mstSet[i] = 0;
   }

   // Always include the first vertex in MST
   key[0] = 0;     // Make key 0 to pick it first
   parent[0] = -1; // First node is always the root of MST
```

```c
    // MST will have V vertices
    for (int count = 0; count < V - 1; count++) {
        // Pick the minimum key vertex not yet included in MST
        int u = minKey(key, mstSet);

        // Add the picked vertex to the MST set
        mstSet[u] = 1;

        // Update key value and parent index of adjacent vertices
        for (int v = 0; v < V; v++) {
            // Update key only if graph[u][v] is non-zero, v is not in mstSet,
            // and the new weight is smaller than the current key
            if (graph[u][v] && mstSet[v] == 0 && graph[u][v] < key[v]) {
                parent[v] = u;
                key[v] = graph[u][v];
            }
        }
    }

    // Print the constructed MST
    printMST(parent, V, graph);
}
int main() {
    /* Let us create the following graph:
          2    3
      (0)--(1)--(2)
       |   / \   |
      6|  8   5  |7
       | /     \ |
      (3)-------(4)
           9
    */
    int graph[V][V] = {
        { 0, 2, 0, 6, 0 },
        { 2, 0, 3, 8, 5 },
        { 0, 3, 0, 0, 7 },
        { 6, 8, 0, 0, 9 },
        { 0, 5, 7, 9, 0 }
    };
    // Run Prim's algorithm
    primMST(graph);
    return 0;
}
```

8. Implementation of Single-pair shortest path Algorithm.

Find Shortest Path using Dijkstra's Algorithm

```
#include <stdio.h>
#include <limits.h>
// Number of vertices in the graph
#define V 9
// A utility function to find the vertex with minimum distance value, from
// the set of vertices not yet included in shortest path tree
int minDistance(int dist[], int sptSet[]) {
 // Initialize min value
 int min = INT_MAX, min_index;
 int v;
 for (v = 0; v < V; v++)
 if (sptSet[v] == 0 && dist[v] <= min)
 min = dist[v], min_index = v;
 return min_index;
}
// A utility function to print the constructed distance array
void printSolution(int dist[], int n) {
 printf("Vertex Distance from Source\n");
 int i;
 for (i = 0; i < V; i++)
 printf("%d \t\t %d\n", i, dist[i]);
}
// Funtion that implements Dijkstra's single source shortest path algorithm
// for a graph represented using adjacency matrix representation
void dijkstra(int graph[V][V], int src) {
 int dist[V]; // The output array. dist[i] will hold the shortest
 // distance from src to i
 int sptSet[V]; // sptSet[i] will 1 if vertex i is included in shortest
 // path tree or shortest distance from src to i is finalized
 // Initialize all distances as INFINITE and stpSet[] as 0
 int i, count, v;
 for (i = 0; i < V; i++)
 dist[i] = INT_MAX, sptSet[i] = 0;
 // Distance of source vertex from itself is always 0
 dist[src] = 0;
 // Find shortest path for all vertices
 for (count = 0; count < V - 1; count++) {
 // Pick the minimum distance vertex from the set of vertices not
```

```
// yet processed. u is always equal to src in first iteration.
int u = minDistance(dist, sptSet);
// Mark the picked vertex as processed
sptSet[u] = 1;
// Update dist value of the adjacent vertices of the picked vertex.
for (v = 0; v < V; v++)
// Update dist[v] only if is not in sptSet, there is an edge from
// u to v, and total weight of path from src to v through u is
// smaller than current value of dist[v]
if (!sptSet[v] && graph[u][v] && dist[u] != INT_MAX && dist[u]
+ graph[u][v] < dist[v])
dist[v] = dist[u] + graph[u][v];
}
// print the constructed distance array
printSolution(dist, V);
}
// driver program to test above function
int main() {
/* Let us create the example graph discussed above */
int graph[V][V] = {{0, 4, 0, 0, 0, 0, 0, 8, 0},
{4, 0, 8, 0, 0, 0, 0, 11, 0},
{0, 8, 0, 7, 0, 4, 0, 0, 2},
{0, 0, 7, 0, 9, 14, 0, 0, 0},
{0, 0, 0, 9, 0, 10, 0, 0, 0},
{0, 0, 4, 0, 10, 0, 2, 0, 0},
{0, 0, 0, 14, 0, 2, 0, 1, 6},
{8, 11, 0, 0, 0, 0, 1, 0, 7},
{0, 0, 2, 0, 0, 0, 6, 7, 0}
};
dijkstra(graph, 0);
return 0;
}
```

9. Implementation of All-pair shortest path Algorithm.

Implement Floyd Warshall Algorithm

// all pair Shorting path Algorithm using Floyd Warshall Algorithm

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

void floydWarshall(int **graph, int n)
{

    int i, j, k;
    for (k = 0; k < n; k++)
    {

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

            for (j = 0; j < n; j++)
            {

                if (graph[i][j] > graph[i][k] + graph[k][j])
                    graph[i][j] = graph[i][k] + graph[k][j];
            }
        }
    }
}


int main(void)
{

    int n, i, j;
    printf("Enter the number of vertices: ");
    scanf("%d", &n);
    int **graph = (int **)malloc((long unsigned) n * sizeof(int *));
    for (i = 0; i < n; i++)
    {

        graph[i] = (int *)malloc((long unsigned) n * sizeof(int));
    }

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

        for (j = 0; j < n; j++)
        {
```

```c
            if (i == j)
                graph[i][j] = 0;
            else
                graph[i][j] = 100;
        }
    }

    printf("Enter the edges: \n");
    for (i = 0; i < n; i++)
    {

        for (j = 0; j < n; j++)
        {

            printf("[%d][%d]: ", i, j);
            scanf("%d", &graph[i][j]);
        }
    }

    printf("The original graph is:\n");
    for (i = 0; i < n; i++)
    {

        for (j = 0; j < n; j++)
        {

            printf("%d ", graph[i][j]);
        }

        printf("\n");
    }

    floydWarshall(graph,  n);
    printf("The shortest path matrix is:\n");
    for (i = 0; i < n; i++)
    {

        for (j = 0; j < n; j++)
        {

            printf("%d ", graph[i][j]);
        }

        printf("\n");
    }

    return 0;
}
```

10. Implementation of String Matching Algorithm.

```
#include<stdio.h>
int main() {
 char str1[30], str2[30];
 int i;
 printf("\nEnter two strings :");
 gets(str1);
 gets(str2);
 i = 0;
 while (str1[i] == str2[i] && str1[i] != '\0')
 i++;
 if (str1[i] > str2[i])
 printf("str1 > str2");
 else if (str1[i] < str2[i])
 printf("str1 < str2");
 else
 printf("str1 = str2");
 return (0);
}
```