Parul University

FACULTY OF ENGINEERING & TECHNOLOGY

BACHELOR OF TECHNOLOGY

COMPUTATIONAL THINKING FOR STRUCTURED DESIGN - 2

(303105151)

2nd

SEMESTER

COMPUTER SCIENCE & ENGINEERING DEPARTMENT

Laboratory Manual

CERTIFICATE

This is to certify that

Mr./Ms *with*

enrolment no *has successfully completed his/her*

laboratory experiments in the ***Computational Thinking for Structured Design-2(303105151)***

from the department of *during the*

academic year ....

Date of Submission:......................... Staff In charge:...........................

Head of Department:...

TABLE OF CONTENT

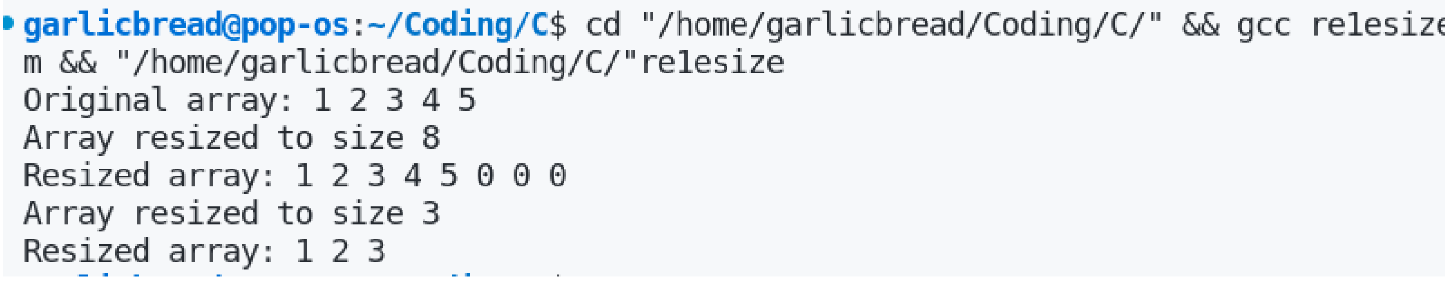
TABLE OF CONTENT

Practical-1

1. Write a c program to increase or decrease the existing size of an 1D array.

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

void resizeArray(int **arr, int *oldSize, int newSize) {
  int *newArray = (int *)malloc(newSize * sizeof(int)); // Allocate memory for new
  array if (newArray == NULL) {
    printf("Memory allocation
    failed!\n"); return;
  }

  for (int i = 0; i < (*oldSize < newSize ? *oldSize : newSize); i++) {
    newArray[i] = (*arr)[i];
  }


  if (*oldSize >
    newSize) {
    free(*arr);
  }


  *arr = newArray;
  *oldSize = newSize;

  printf("Array resized to size %d\n", newSize);
}

int main() {
  int *arr = (int *)malloc(5 * sizeof(int)); // Initial size
  of 5 for (int i = 0; i < 5; i++) {
    arr[i] = i + 1; // Initialize elements (optional)
  }
  int oldSize = 5;
  printf("Original
  array: ");
  for (int i = 0; i < oldSize;
    i++) { printf("%d ", arr[i]);
  }
  printf("\n");

  // Increase
  size int
  newSize =
  8;
  resizeArray(&arr, &oldSize, newSize);

  printf("Resized array: ");
  for (int i = 0; i < newSize;
    i++) { printf("%d ", arr[i]);
  }
  printf("\n");

  // Decrease size
  (optional) newSize =
  3;
  resizeArray(&arr, &oldSize, newSize);

  printf("Resized array: ");
  for (int i = 0; i < newSize;
    i++) { printf("%d ", arr[i]);
  }
  printf("\n");

  free(
  arr);
  retur
  n 0;
}
```

Output:



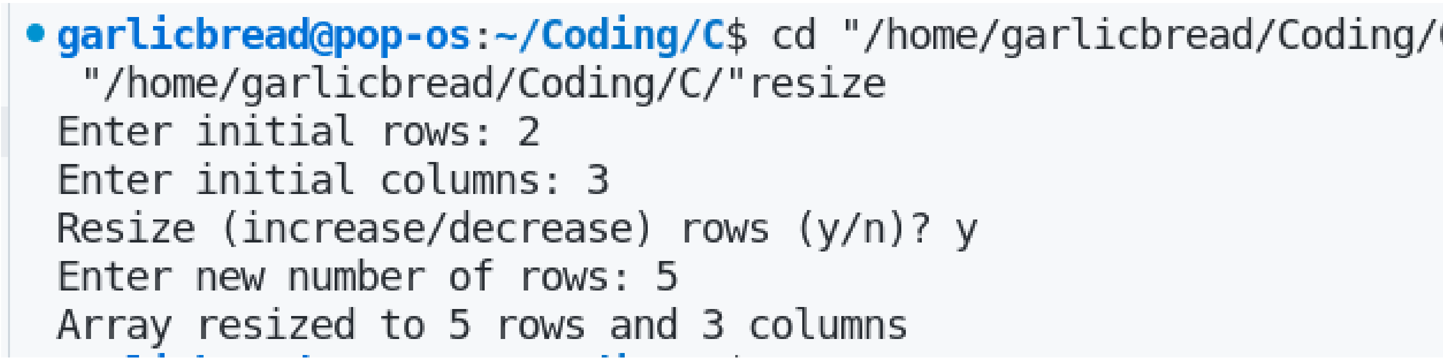
```
garlicbread@pop-os:~/Coding/C$ cd "/home/garlicbread/Coding/C/" && gcc relesize
m && "/home/garlicbread/Coding/C/"relesize
Original array: 1 2 3 4 5
Array resized to size 8
Resized array: 1 2 3 4 5 0 0 0
Array resized to size 3
Resized array: 1 2 3
```

2. Write a c program on 2D array to Increase & Decrease
i) No of subarrays

ii) elements in the subarrays

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

void resizeArray(int ***arr, int *rows, int *cols, int newRows, int
  newCols) { int **newArray = (int **)malloc(newRows * sizeof(int *));
  if (newArray == NULL) {
    printf("Memory allocation
    failed!\n"); return;
  }

  for (int i = 0; i < newRows; i++) {
    newArray[i] = (int *)malloc(newCols *
    sizeof(int)); if (newArray[i] == NULL) {
      printf("Memory allocation
      failed!\n"); return;
    }
  }

  // Copy elements based on size changes
  for (int i = 0; i < (*rows < newRows ? *rows : newRows);
    i++) { for (int j = 0; j < (*cols < newCols ? *cols :
    newCols); j++) { newArray[i][j] = (*arr)[i][j];
    }
  }

  // Free old memory
  for (int i = 0; i < *rows;
    i++) { free((*arr)[i]);
  }
  free(*arr);

  *arr = newArray;
  *rows = newRows;
  *cols = newCols;

  printf("Array resized to %d rows and %d columns\n", newRows, newCols);
}

int main() {
  int **arr, rows, cols;

  printf("Enter initial rows:
  "); scanf("%d", &rows);
  printf("Enter initial columns: ");
  scanf("%d", &cols);

  arr = (int **)malloc(rows *
  sizeof(int *)); for (int i = 0; i < rows;
  i++) {
    arr[i] = (int *)malloc(cols * sizeof(int));
  }


  printf("Resize (increase/decrease) rows (y/n)?
  "); char choice;
  scanf(" %c",
  &choice); if
  (choice == 'y') {
    int newRows;
    printf("Enter new number of
    rows: "); scanf("%d",
    &newRows);
    resizeArray(&arr, &rows, &cols, newRows, cols);
  }

  for (int i = 0; i < rows;
    i++) { free(arr[i]);
  }
  free(arr);

  return 0;
}
```

Output:



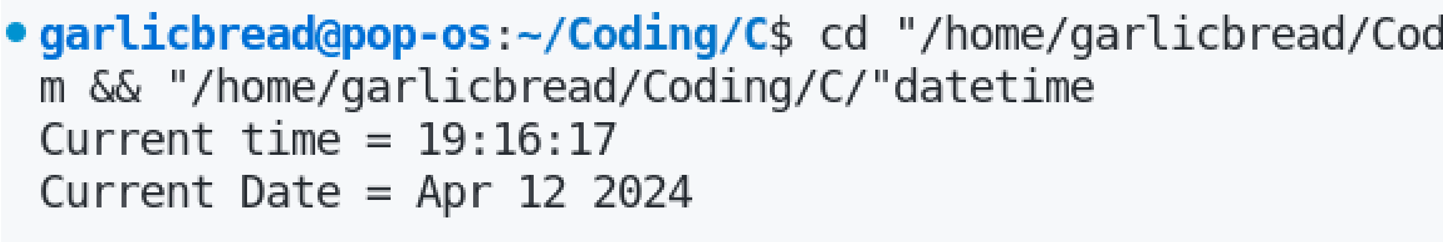
```
garlicbread@pop-os:~/Coding/C$ cd "/home/garlicbread/Coding/
 "/home/garlicbread/Coding/C/"resize
Enter initial rows: 2
Enter initial columns: 3
Resize (increase/decrease) rows (y/n)? y
Enter new number of rows: 5
Array resized to 5 rows and 3 columns
```

Practical-2

1. Write a Code to display present date and time using c language.

```
#include<stdio
.h> int main(){
   printf("Current time = %s\n",__TIME
   ___________________________);
   printf("Current Date = %s\n",__DATE
   ___________________________);
   return 0;
}
```

Output:



```
• garlicbread@pop-os:~/Coding/C$ cd "/home/garlicbread/Cod
  m && "/home/garlicbread/Coding/C/"datetime
  Current time = 19:16:17
  Current Date = Apr 12 2024
```

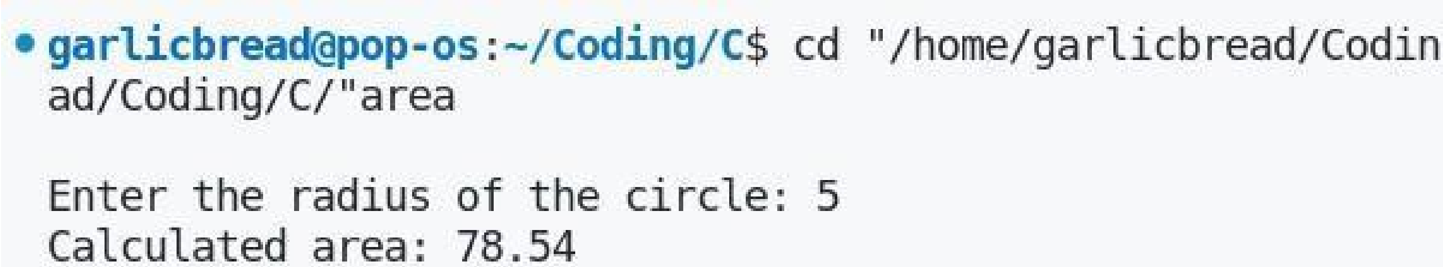
2. Write a c program to demonstrate pre-processor directives i)Macros
ii) Conditional Compilation

i) Macros

```
#include <stdio.h>
#define PI
3.14159 int
main() {
   float radius, area;
   printf("Enter the radius of the
   circle: "); scanf("%f", &radius);
```

```
    area = PI * radius * radius;

    printf("Calculated area: %.2f\n",
     area); return 0;
}
```

Output:



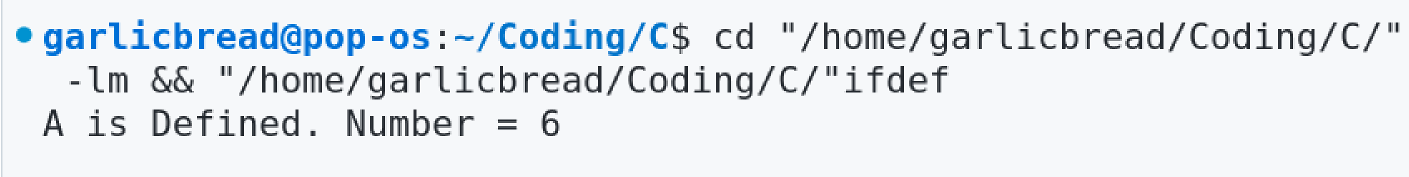
```
• garlicbread@pop-os:~/Coding/C$ cd "/home/garlicbread/Codin
  ad/Coding/C/"area

  Enter the radius of the circle: 5
  Calculated area: 78.54
```

ii) Conditional

Compilation #include

```
<stdio.h> #define A 6

int main() {

    #ifdef A
        printf("A is Defined. Number =
    %d\n",A); #else
        printf("A not defined.\n");
    #endif

    return 0;
}
```

Output:



```
• garlicbread@pop-os:~/Coding/C$ cd "/home/garlicbread/Coding/C/"
   -lm && "/home/garlicbread/Coding/C/"ifdef
  A is Defined. Number = 6
```

Practical-3

1.Write a C program that uses functions to perform the following Operations.

i)Reading a complex number

ii) Writing a complex number

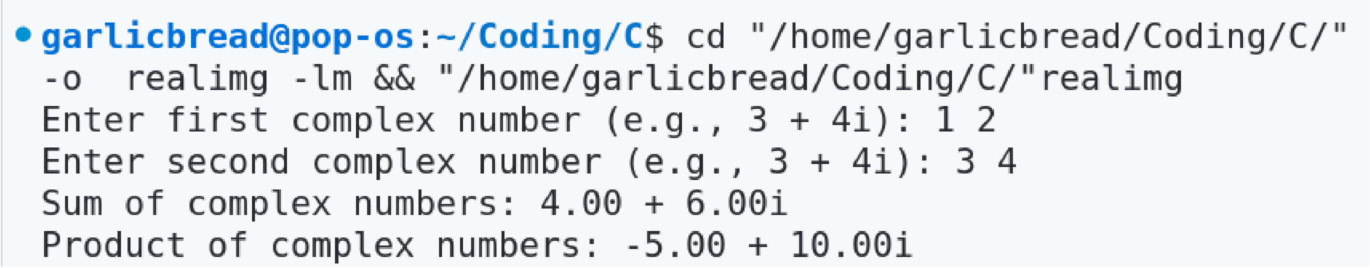
iii) Addition of two complex numbers

iv) Multiplication of two complex numbers

```
#include
<stdio.h>
typedef struct {
   float
   real;
   float
   imag;
} Complex;


void readComplex(Complex *num) {
   printf("(e.g., 3 + 4i): ");
   scanf("%f %f", &num->real, &num->imag);
}


void writeComplex(Complex num) {
   printf("%.2f + %.2fi\n", num.real,
   num.imag);
}


Complex addComplex(Complex num1, Complex
   num2) { Complex sum;
   sum.real = num1.real + num2.real;
   sum.imag = num1.imag +
   num2.imag; return sum;
}
```

```
Complex multiplyComplex(Complex num1, Complex
    num2) { Complex product;
    product.real = (num1.real * num2.real) - (num1.imag *
    num2.imag); product.imag = (num1.real * num2.imag) +
    (num1.imag * num2.real); return product;
}

int main() {
    Complex num1, num2, sum, product;


    printf("Enter first complex
    number ");
    readComplex(&num1);

    printf("Enter second complex
    number "); readComplex(&num2);

    sum = addComplex(num1,
    num2); printf("Sum of complex
    numbers: ");
    writeComplex(sum);

    product = multiplyComplex(num1,
    num2); printf("Product of complex
    numbers: "); writeComplex(product);

    return 0;
}
```

Output:



```
garlicbread@pop-os:~/Coding/C$ cd "/home/garlicbread/Coding/C/"
-o  realimg -lm && "/home/garlicbread/Coding/C/"realimg
Enter first complex number (e.g., 3 + 4i): 1 2
Enter second complex number (e.g., 3 + 4i): 3 4
Sum of complex numbers: 4.00 + 6.00i
Product of complex numbers: -5.00 + 10.00i
```

2. Write a c program to store records of n students based on roll_no, name, gender and 5 subject marks.

i) Calculate percentage each student using 5 Subjects.

ii) Display the student list according to their percentages.

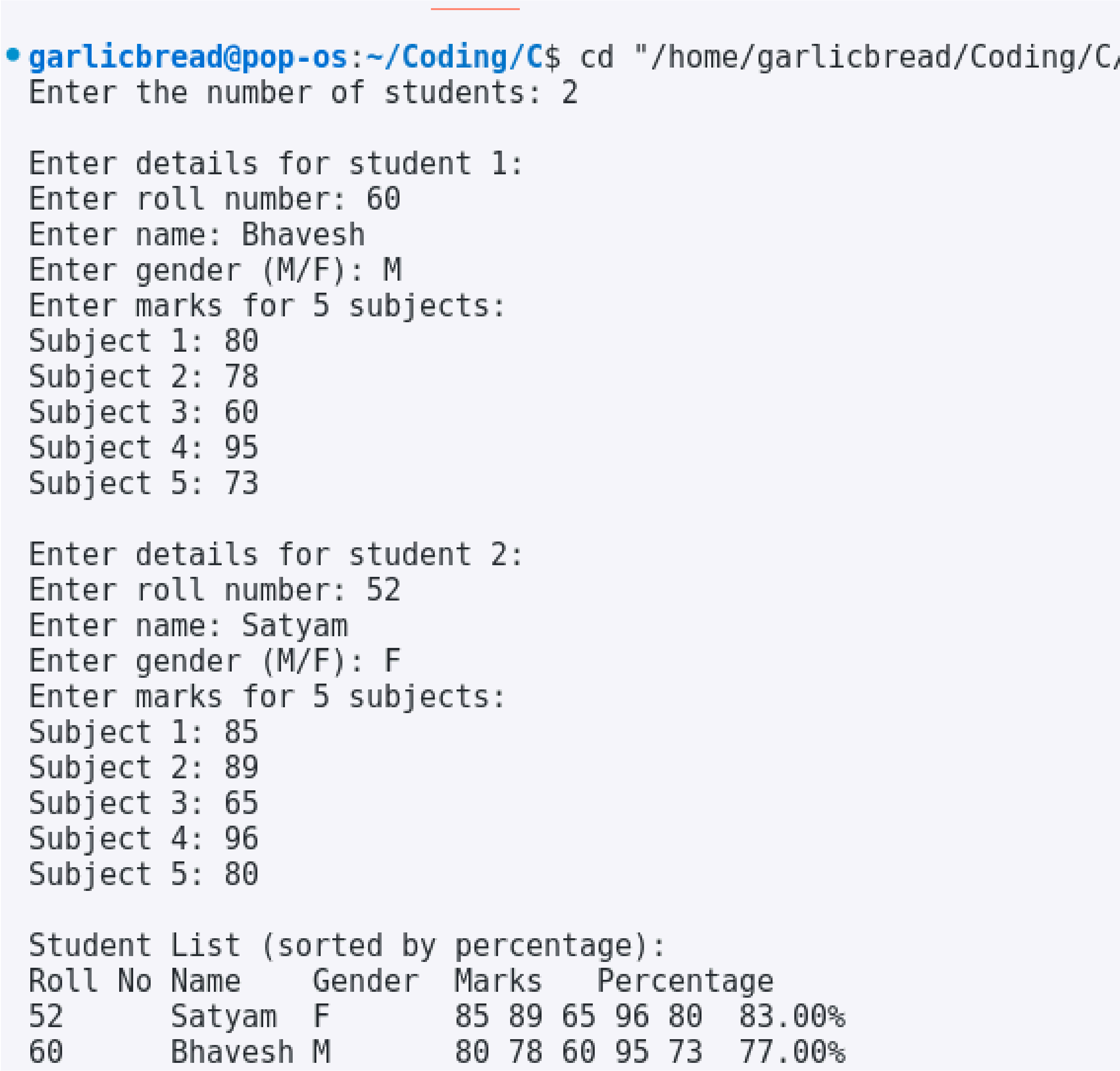
```
#include <stdio.h>
#define MAX_SUBJECTS
5 struct Student {
   int roll_no;
   char
   name[50];
   char
   gender;
   int
   marks[MAX_SUBJECTS];
   float percentage;
};

void readStudent(struct Student
   *student) { printf("Enter roll number:
   ");
   scanf("%d", &student->roll_no);

   printf("Enter name: ");
   scanf("%s",
   student->name);

   printf("Enter gender (M/F): ");
   scanf(" %c", &student->gender);

   printf("Enter marks for %d subjects:\n",
   MAX_SUBJECTS); for (int i = 0; i < MAX_SUBJECTS;
   i++) {
```

```
printf("Subject %d: ", i + 1);
scanf("%d", &student->marks[i]);
```

```
    }
}

void calculatePercentage(struct Student
    *student) { int total_marks = 0;
    for (int i = 0; i < MAX_SUBJECTS;
        i++) { total_marks +=
        student->marks[i];
    }
    student->percentage = (float)(total_marks /MAX_SUBJECTS*100)/100;
}

void swapStudents(struct Student *a, struct Student
    *b) { struct Student temp = *a;
    *a = *b;
    *b = temp;
}

void sortStudentsByPercentage(struct Student students[], int
    n) { for (int i = 0; i < n - 1; i++) {
        for (int j = 0; j < n - i - 1; j++) {
            if (students[j].percentage < students[j +
                1].percentage) { swapStudents(&students[j],
                &students[j + 1]);
            }
        }
    }
}

int main()
    { int n;

    printf("Enter the number of
    students: "); scanf("%d", &n);

    struct Student
```

```
students[n]; for (int i =

0; i < n; i++) {
        printf("\nEnter details for student %d:\n", i +
        1); readStudent(&students[i]);
        calculatePercentage(&students[i]);
    }

    sortStudentsByPercentage(students, n);

    printf("\nStudent List (sorted by percentage):\n");
    printf("Roll
    No\tName\tGender\tMarks\tPercentage\n"); for (int i
    = 0; i < n; i++) {
        printf("%d\t%s\t%c\t", students[i].roll_no, students[i].name,
        students[i].gender); for (int j = 0; j < MAX_SUBJECTS; j++) {
            printf("%d ", students[i].marks[j]);
        }
        printf("\t%.2f%%\n", students[i].percentage);
    }

    return 0;
}
```

Output:



```
garlicbread@pop-os:~/Coding/C$ cd "/home/garlicbread/Coding/C/
Enter the number of students: 2

Enter details for student 1:
Enter roll number: 60
Enter name: Bhavesh
Enter gender (M/F): M
Enter marks for 5 subjects:
Subject 1: 80
Subject 2: 78
Subject 3: 60
Subject 4: 95
Subject 5: 73

Enter details for student 2:
Enter roll number: 52
Enter name: Satyam
Enter gender (M/F): F
Enter marks for 5 subjects:
Subject 1: 85
Subject 2: 89
Subject 3: 65
Subject 4: 96
Subject 5: 80

Student List (sorted by percentage):
Roll No Name    Gender  Marks   Percentage
52      Satyam  F       85 89 65 96 80  83.00%
60      Bhavesh M       80 78 60 95 73  77.00%
```

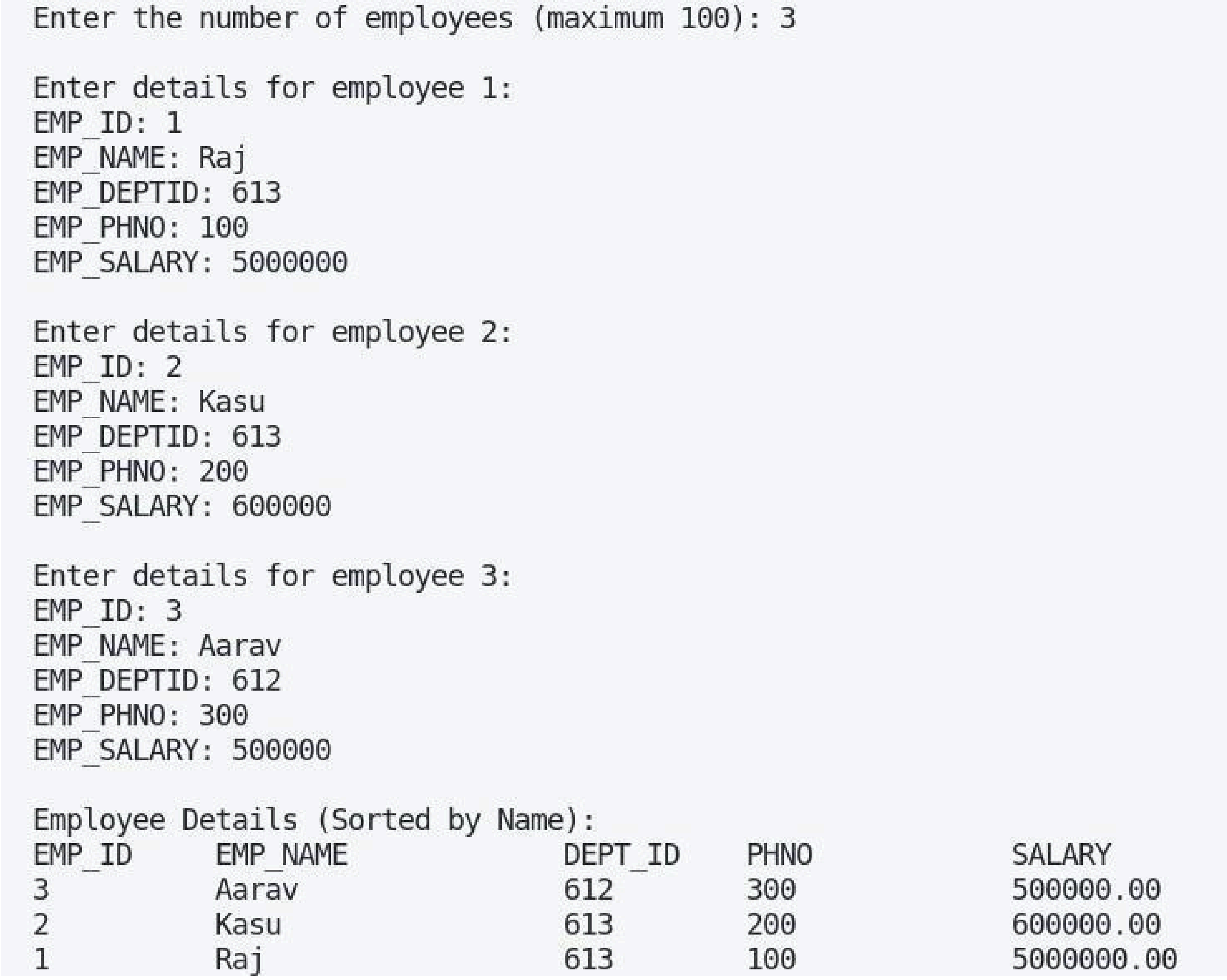
Practical-4

Write a C Program to store n employees records EMP_ID,EMP_NAME,EMP_DEPTID,EMP_PHNO,EMP_SALARY and display all the details of employees using EMP_NAME in sorted order

```
#include
<stdio.h>
#include
<stdlib.h>
#include
<string.h>
#define MAX_EMPLOYEES


100 struct Employee {


   int emp_id;
   char
   emp_name[50];
   int emp_deptid;
   char
   emp_phno[15];
   float emp_salary;
};

int compareEmployees(const void *a, const void *b) {
   const struct Employee *emp1 = (const struct Employee
   *)a; const struct Employee *emp2 = (const struct
   Employee *)b; return strcmp(emp1->emp_name,
   emp2->emp_name);
}

int main() {
   int num_employees, i;

   printf("Enter the number of employees (maximum %d): ",
   MAX_EMPLOYEES); scanf("%d", &num_employees);
```

```
    if (num_employees > MAX_EMPLOYEES) {
                printf("Error: Number of employees exceeds limit
(%d).\n", MAX_EMPLOYEES);
        return 1;
    }
    struct Employee employees[num_employees];


    for (i = 0; i < num_employees; i++) {
        printf("\nEnter details for employee %d:\n", i +
        1); printf("EMP_ID: ");
        scanf("%d", &employees[i].emp_id);
        printf("EMP_NAME: ");
        scanf(" %[^\n]", employees[i].emp_name); // Read entire name with
        spaces printf("EMP_DEPTID: ");
        scanf("%d", &employees[i].emp_deptid);
        printf("EMP_PHNO: ");
        scanf(" %[^\n]", employees[i].emp_phno); // Read entire phone
        number printf("EMP_SALARY: ");
        scanf("%f", &employees[i].emp_salary);
    }


                    qsort(employees,      num_employees,      sizeof(struct
Employee), compareEmployees);

    printf("\nEmployee Details (Sorted by Name):\n");
          printf("%-10s %-20s %-10s %-15s %-10s\n", "EMP_ID", "EMP_NAME",
"DEPT_ID", "PHNO", "SALARY");
    for (i = 0; i < num_employees; i++) {
        printf("%-10d %-20s %-10d %-15s %-10.2f\n",
               employees[i].emp_id, employees[i].emp_name,
               employees[i].emp_deptid,
               employees[i].emp_phno,
               employees[i].emp_salary);
    }

    return 0;
}
```

Output:



```
Enter the number of employees (maximum 100): 3

Enter details for employee 1:
EMP_ID: 1
EMP_NAME: Raj
EMP_DEPTID: 613
EMP_PHNO: 100
EMP_SALARY: 5000000

Enter details for employee 2:
EMP_ID: 2
EMP_NAME: Kasu
EMP_DEPTID: 613
EMP_PHNO: 200
EMP_SALARY: 600000

Enter details for employee 3:
EMP_ID: 3
EMP_NAME: Aarav
EMP_DEPTID: 612
EMP_PHNO: 300
EMP_SALARY: 500000

Employee Details (Sorted by Name):
EMP_ID     EMP_NAME             DEPT_ID    PHNO            SALARY
3          Aarav                612        300             500000.00
2          Kasu                 613        200             600000.00
1          Raj                  613        100             5000000.00
```

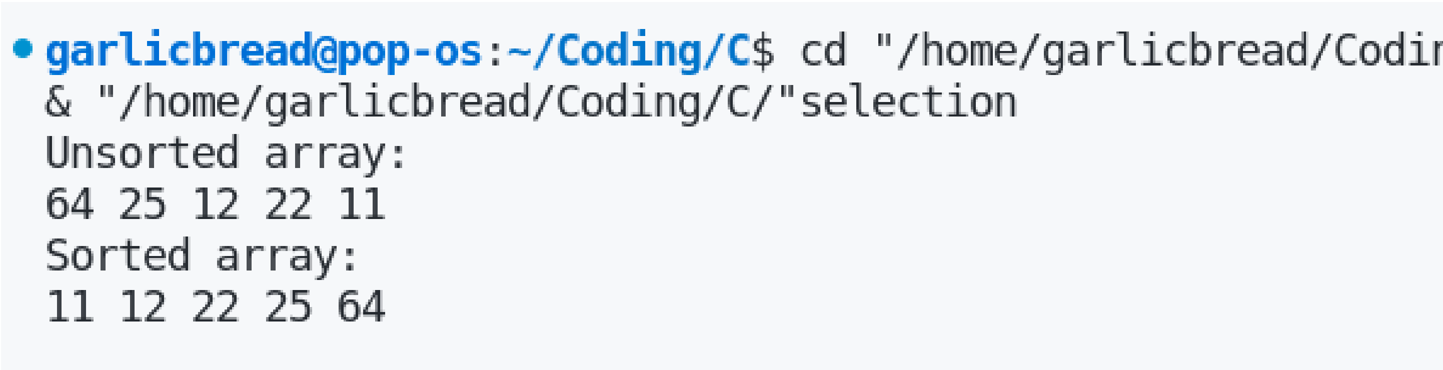
Practical-5

1. Write a c program to implement selection Sort & Bubble sort.

Selection Sort

```
#include

<stdio.h>

void selectionSort(int arr[], int
  n) { int i, j, min_idx;

  for (i = 0; i < n-1;
    i++) { min_idx = i;
    for (j = i + 1; j < n;
      j++) if (arr[j] <
      arr[min_idx])
      min_idx = j;


    if (min_idx != i) {
      int temp =
      arr[min_idx];
      arr[min_idx] = arr[i];
      arr[i] = temp;
    }
  }
}

int main() {
  int arr[] = {64, 25, 12, 22, 11};
  int n = sizeof(arr) / sizeof(arr[0]);

  printf("Unsorted array:
  \n"); for (int i = 0; i < n;
  i++) printf("%d ", arr[i]);
  selectionSort(arr, n);

  printf("\nSorted array:
  \n"); for (int i = 0; i < n;
  i++) printf("%d ", arr[i]);

  printf("\n\
  n");
  return 0;
}
```

Output:



```
● garlicbread@pop-os:~/Coding/C$ cd "/home/garlicbread/Codi
  & "/home/garlicbread/Coding/C/"selection
  Unsorted array:
  64 25 12 22 11
  Sorted array:
  11 12 22 25 64
```

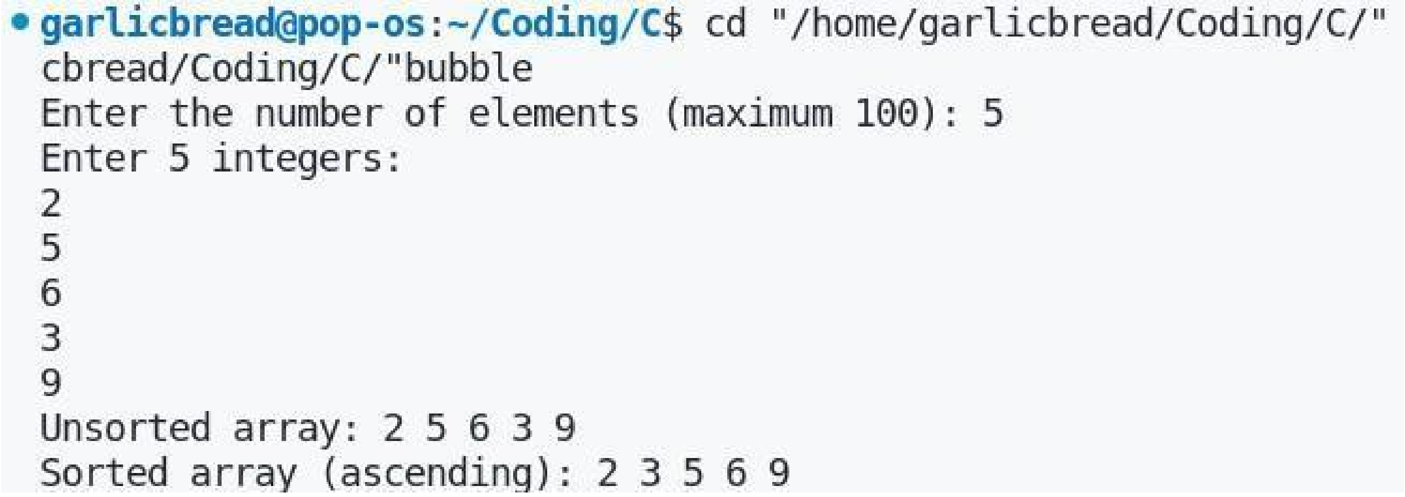
Bubble Sort

```
#include

<stdio.h>

void bubbleSort(int arr[], int
   n) { int i, j, temp,
   swapped;
   for (i = 0; i < n - 1;
      i++) { swapped =
      0;
      for (j = 0; j < n - i - 1; j++) {
         if (arr[j] > arr[j + 1]) {
            // Swap
            elements
```

```
    temp = arr[j];
    arr[j] = arr[j +
    1]; arr[j + 1] =
    temp;
    swapped = 1;
}
```

```
        }
        if (!swapped)
            { break;
        }
    }
}

int main() {
    int arr[100], n, i;

    printf("Enter the number of elements (maximum
    100): "); scanf("%d", &n);

    printf("Enter %d integers:\n",
    n); for (i = 0; i < n; i++) {
        scanf("%d", &arr[i]);
    }

    printf("Unsorted array:
    "); for (i = 0; i < n; i++) {
        printf("%d ", arr[i]);
    }
    printf("\n");

    bubbleSort(arr,

    n);

    printf("Sorted array (ascending):
    "); for (i = 0; i < n; i++) {
        printf("%d ", arr[i]);
    }
    printf("\n");

    return 0;
}
```

Output:



```
• garlicbread@pop-os:~/Coding/C$ cd "/home/garlicbread/Coding/C/"
  cbread/Coding/C/"bubble
  Enter the number of elements (maximum 100): 5
  Enter 5 integers:
  2
  5
  6
  3
  9
  Unsorted array: 2 5 6 3 9
  Sorted array (ascending): 2 3 5 6 9
```

2. Write a C program to reverse the elements within a given range in a sorted list

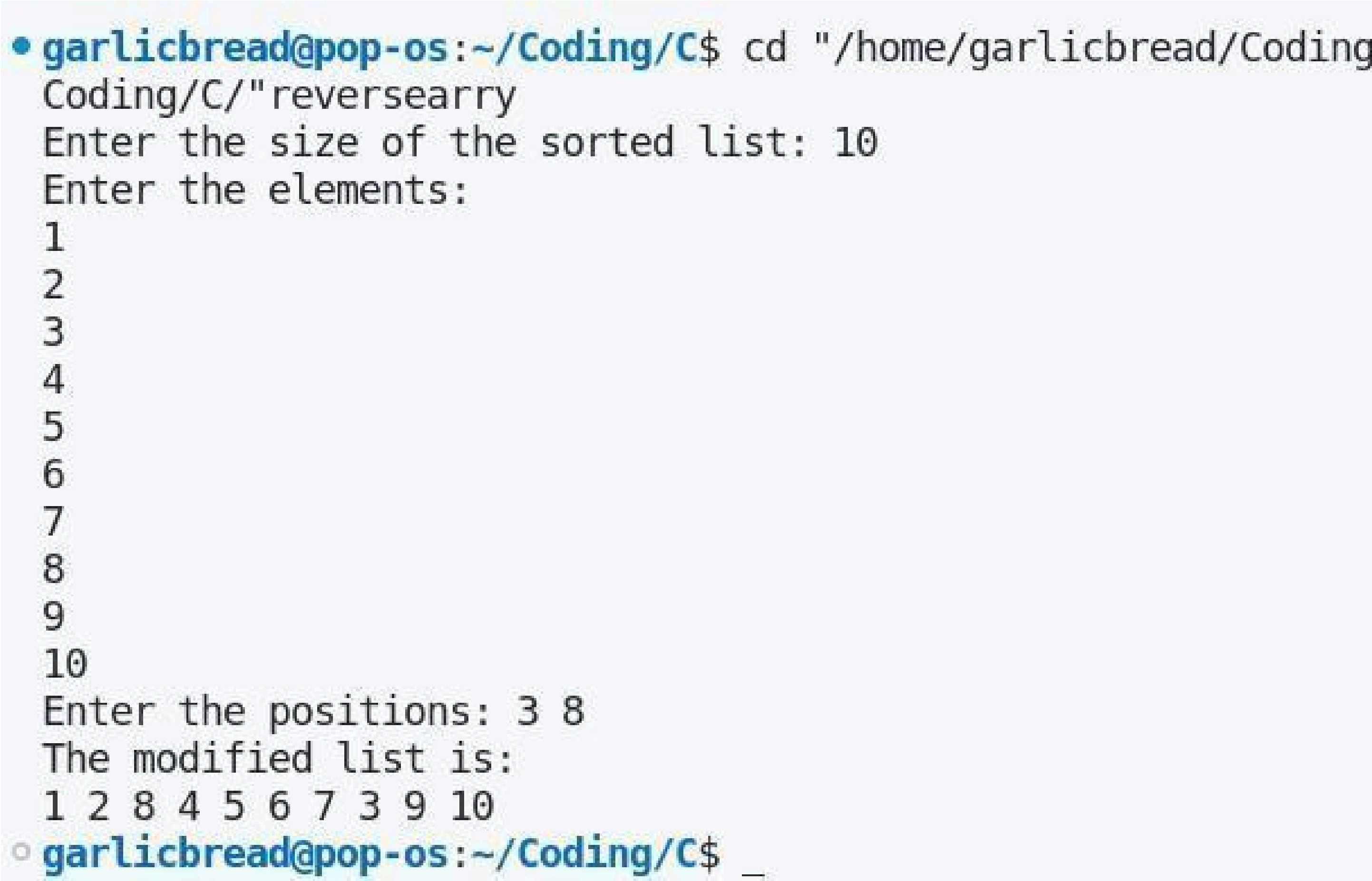
input : 10
9 1 2 4 3 4 6 7 8 10
3 8

output: 1 2 8 7 6 4 4 3 9 10

the sorted list of given array elements is 1 2 3 4 4 6 7 8 9 10 , after reversing the elements with in the range 3 and 8 is 1 2 8 76 4 4 3 9 10.

```
#include
<stdio.h> int
main() {
    int size, i, pos1, pos2, temp;

    printf("Enter the size of the sorted
    list: "); scanf("%d", &size);

    int list[size];

    printf("Enter the elements:\n");
    for (i = 0; i < size;
        i++) { scanf("%d",
        &list[i]);
    }

    printf("Enter the positions: ");
    scanf("%d %d", &pos1, &pos2);

    if (pos1 < 1 || pos1 > size || pos2 < 1 || pos2 > size) {
        printf("Invalid positions. Please enter values between 1 and %d.\n",
        size); return 1;
    }

    pos1--;
    pos2--;

    if (pos1 == pos2) {
        printf("The elements are already at the same position.\n");
    } else {
        temp = list[pos1];
        list[pos1] =
        list[pos2];
        list[pos2] = temp;

    }

    printf("The modified list
    is:\n"); for (i = 0; i < size;
    i++) {
        printf("%d ", list[i]);
    }
    printf("\n");

    return 0;
}
```

Output:



```
• garlicbread@pop-os:~/Coding/C$ cd "/home/garlicbread/Coding
  Coding/C/"reversearry
  Enter the size of the sorted list: 10
  Enter the elements:
  1
  2
  3
  4
  5
  6
  7
  8
  9
  10
  Enter the positions: 3 8
  The modified list is:
  1 2 8 4 5 6 7 3 9 10
○ garlicbread@pop-os:~/Coding/C$ _
```

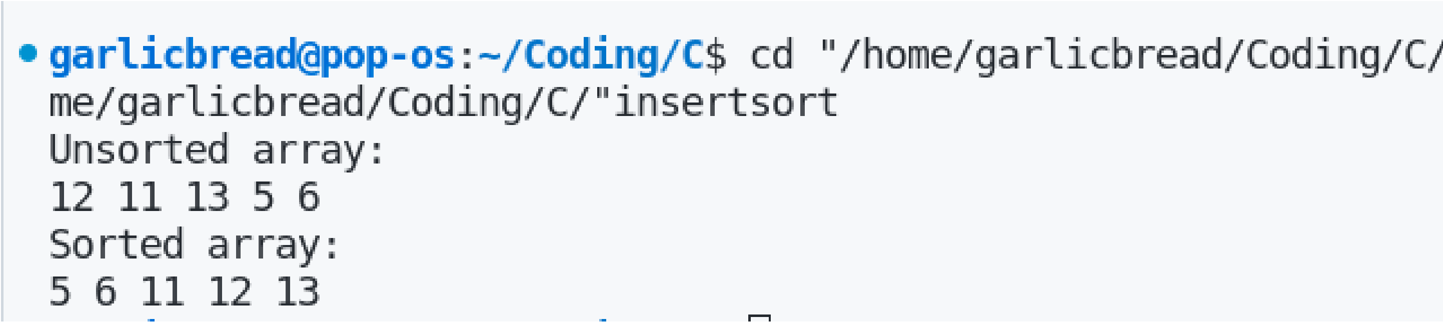
Practical-6

1. Write a c program to implement Insertion Sort & Quick sort.

Insertion Sort:

```
#include <stdio.h>

void insertionSort(int arr[], int
  n) { int i, key, j;


  for (i = 1; i < n;
    i++) { key =
    arr[i];
    j = i - 1;

    while (j >= 0 && arr[j] >
      key) { arr[j + 1] = arr[j];
      j--;
    }
    arr[j + 1] = key;
  }
}


void printArray(int arr[], int
  n) { for (int i = 0; i < n;
  i++) { printf("%d ", arr[i]);
  }
  printf("\n");
}

int main() {
  int arr[] = {12, 11, 13, 5, 6};
  int n = sizeof(arr) / sizeof(arr[0]);
  printf("Unsorted array: \n");
  printArray(arr, n);

  insertionSort(arr, n);

  printf("Sorted array: \n");
  printArray(arr, n);

  return 0;
}
```

Output:



```
garlicbread@pop-os:~/Coding/C$ cd "/home/garlicbread/Coding/C/
me/garlicbread/Coding/C/"insertsort
Unsorted array:
12 11 13 5 6
Sorted array:
5 6 11 12 13
```

Quick Sort

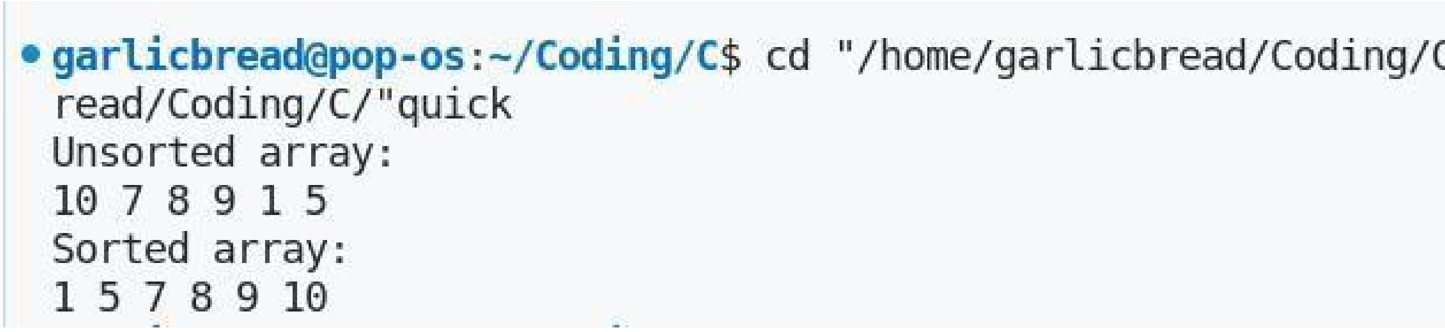
```
#include

<stdio.h>

// Function to swap two
elements void swap(int *a, int
*b) {
   int temp = *a;
   *a = *b;
   *b = temp;
}

int partition(int arr[], int low, int
   high) { int pivot = arr[high];
   int i = (low - 1);
```

```
    for (int j = low; j <= high - 1; j++) {

        if (arr[j] <
            pivot) {
            i++;
            swap(&arr[i], &arr[j]);
        }
    }
    swap(&arr[i + 1],
    &arr[high]); return (i + 1);
}


void quickSort(int arr[], int low, int
    high) { if (low < high) {

        int pi = partition(arr, low, high);


        quickSort(arr, low, pi -
        1); quickSort(arr, pi +
        1, high);
    }
}


void printArray(int arr[], int
    size) { for (int i = 0; i <
    size; i++)
        printf("%d ", arr[i]);
    printf("\n");
}

int main() {
    int arr[] = {10, 7, 8, 9, 1, 5};
    int n = sizeof(arr) / sizeof(arr[0]);
```

```
printf("Unsorted array: \n");
printArray(arr, n);
```

```
    quickSort(arr, 0, n - 1);

    printf("Sorted array: \n");
    printArray(arr, n);

    return 0;
}
```

Output:



```
• garlicbread@pop-os:~/Coding/C$ cd "/home/garlicbread/Coding/C
  read/Coding/C/"quick
  Unsorted array:
  10 7 8 9 1 5
  Sorted array:
  1 5 7 8 9 10
```

2. Write a c program to sort the given n integers and perform following operations

i) Find the products of every two odd position Elements

ii) Find the sum of every two even position elements

Explanation:

The sorted list of given input is 1 2 3 4 5 6 7 8 9, the product of alternative odd position elements is 1*3 = 3,3*5=15,5*7=35… and the sum of two even position elements 2+4 =6,4+6=10.

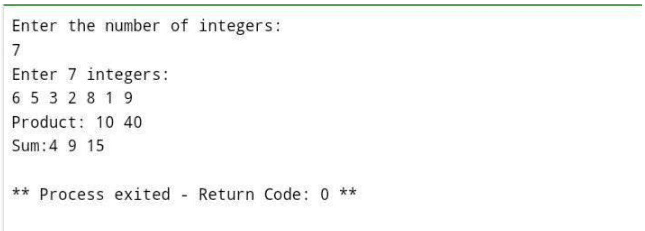
```
#include <stdio.h>
```

```
void bubbleSort(int arr[], int
    n) { for (int i = 0; i < n-1;
    i++) {
        for (int j = 0; j < n-i-1;
            j++) { if (arr[j] >
            arr[j+1]) {
                int temp =
                arr[j]; arr[j] =
                arr[j+1];
                arr[j+1]     =
                temp;
            }
        }
    }
}

int main()
    { int n;
    printf("Enter the number of
    integers: "); scanf("%d", &n);

    int arr[n];
    printf("Enter %d integers:\n",
    n); for (int i = 0; i < n; i++) {
        scanf("%d", &arr[i]);
    }

    // Sorting the
    array
    bubbleSort(arr,
    n);

    printf("Product: ");
    long long product_odd
    = 1; for (int i = 1; i < n; i
    += 2) {
```

```
    product_odd =
    arr[i]*arr[i+2]; if(i+2<n){
    printf("%lld ", product_odd);
    }
    product_odd=1;
}
```

```
    printf("\nSum:");
    long long sum_even = 0;
    for (int i = 0; i < n - 1; i += 2)
        { sum_even = arr[i] + arr[i
        + 2]; if(i+2<n){
        printf("%lld ",sum_even);
        }
        sum_even=0;
    }


    return 0;
}
```

Output:



```
Enter the number of integers:
7
Enter 7 integers:
6 5 3 2 8 1 9
Product: 10 40
Sum:4 9 15

** Process exited - Return Code: 0 **
```

Practica

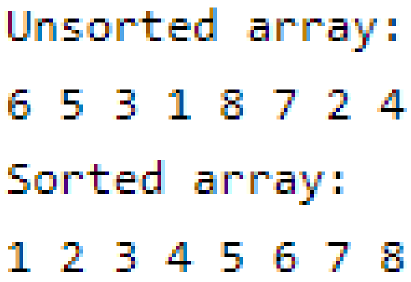
I-7 Write a C Program to implement Merge

Sort. #include <stdio.h>

```
void merge(int arr[], int left, int mid, int
  right) { int size1 = mid - left + 1;
  int size2 = right - mid;
int Left[size1], Right[size2];


for (int i = 0; i < size1; i++) Left[i] = arr[left + i];
for (int j = 0; j < size2; j++) Right[j] = arr[mid + 1 + j];

int i = 0, j = 0, k = left;
while (i < size1 && j <
  size2) { if (Left[i] <=
  Right[j]) {
    arr[k] =
    Left[i]; i++;
  } else {
    arr[k] =
    Right[j]; j++;
  }
  k
  +
  +
  ;
}


while (i <
  size1) {
  arr[k] =
  Left[i]; i++;
  k++;
}


while     (j     <
  size2)         {
  arr[k]         =
  Right[j]; j++;
  k++;
}
```

```
}

void mergeSort(int arr[], int left, int
  right) { if (left < right) {
    int mid = left + (right - left) / 2;
    mergeSort(arr, left, mid);
    mergeSort(arr, mid + 1,
    right);

    merge(arr, left, mid, right);
  }
}

void printArray(int arr[], int n) {
  for (int i = 0; i < n; ++i) printf("%d ",
  arr[i]); printf("\n");
}

int main() {
  int arr[] = {6, 5, 3, 1, 8, 7, 2, 4};
  int n = sizeof(arr) / sizeof(arr[0]);

  printf("Unsorted array: \n");
  printArray(arr, n);

  mergeSort(arr, 0, n - 1);

  printf("Sorted array: \n");
  printArray(arr, n);

  return 0;
}
```

Output:



```
Unsorted array:
6 5 3 1 8 7 2 4
Sorted array:
1 2 3 4 5 6 7 8
```

Practical-8

1. Write a c program to sort in ascending order and reverse the individual row elements of an mxn matrix

```
#include <stdio.h>

void swap(int *a, int
  *b) { int temp = *a;
  *a = *b;
  *b = temp;
}


void selectionSort(int arr[], int
  n) { for (int i = 0; i < n - 1;
  i++) {
    int minIndex = i;
    for (int j = i + 1; j < n;
      j++) { if (arr[j] <
      arr[minIndex])        {
      minIndex = j;
      }
    }
    if (minIndex != i) {
      swap(&arr[i],
      &arr[minIndex]);
    }
  }
}


void reverseArray(int arr[], int
  n) { for (int i = 0; i < n / 2;
  i++) { swap(&arr[i], &arr[n - i
  - 1]);
```

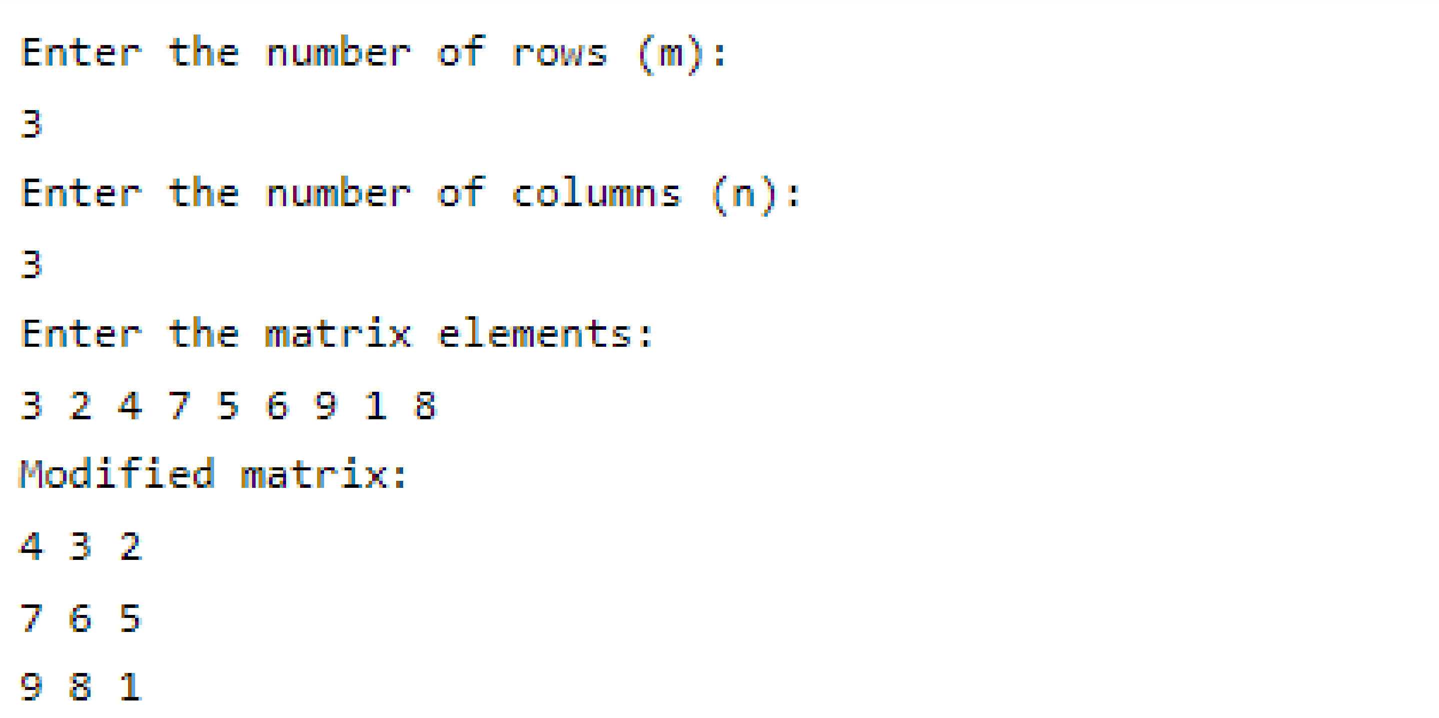
```
    }
  }
```

```
void sortAndReverseRows(int matrix[][100], int m, int
  n) { for (int i = 0; i < m; i++) {
    selectionSort(matrix[i], n); // Sort the row in ascending
    order reverseArray(matrix[i], n);         // Reverse the sorted
    row elements
  }
}


void printMatrix(int matrix[][100], int m, int
  n) { for (int i = 0; i < m; i++) {
    for (int j = 0; j < n; j++) {
      printf("%d ", matrix[i][j]);
    }
    printf("\n");
  }
}

int main()
  { int m,
  n;


  printf("Enter the number of rows
  (m): "); scanf("%d", &m);
  printf("Enter the number of columns
  (n): "); scanf("%d", &n);


  int matrix[m][100];


  printf("Enter the matrix
  elements:\n"); for (int i = 0; i < m;
  i++) {
    for (int j = 0; j < n; j++) {
      scanf("%d", &matrix[i][j]);
```

```
        }
```

```
    }


    sortAndReverseRows(matrix, m, n);

    printf("Modified matrix:\n");
    printMatrix(matrix, m, n);

    return 0;
  }
```

Output:



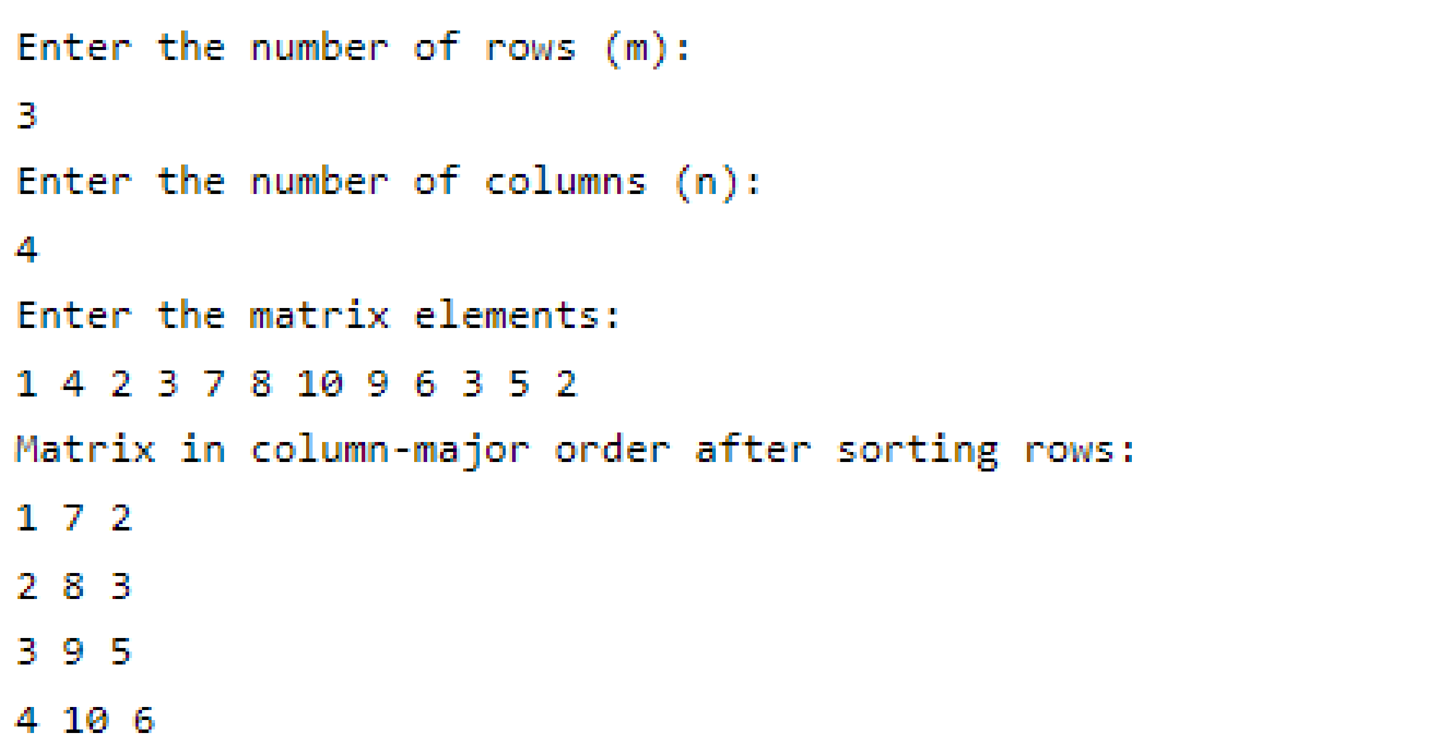
```
Enter the number of rows (m):
3
Enter the number of columns (n):
3
Enter the matrix elements:
3 2 4 7 5 6 9 1 8
Modified matrix:
4 3 2
7 6 5
9 8 1
```

2. Write a c program to sort elements in row wise and print the elements of matrix in Column major order.

```
#include <stdio.h>

void swap(int *a, int
  *b) { int temp = *a;
  *a = *b;
  *b = temp;
}


void selectionSort(int arr[], int
  n) { for (int i = 0; i < n - 1;
  i++) {
    int minIndex = i;
    for (int j = i + 1; j < n;
      j++)   {   if   (arr[j]   <
      arr[minIndex])          {
      minIndex = j;
      }
    }
    if (minIndex != i) {
      swap(&arr[i],
      &arr[minIndex]);
    }
  }
}


void printMatrixColumnMajor(int matrix[][100], int m, int
  n) { for (int j = 0; j < n; j++) {
    for (int i = 0; i < m; i++) {
      printf("%d ", matrix[i][j]);
    }
    printf("\n");
  }
}
```

```
int main()
  { int m,
  n;


  printf("Enter the number of rows
  (m): "); scanf("%d", &m);
  printf("Enter the number of columns
  (n): "); scanf("%d", &n);


  int matrix[m][100];


  printf("Enter the matrix
  elements:\n"); for (int i = 0; i < m;
  i++) {
    for (int j = 0; j < n; j++) {
      scanf("%d", &matrix[i][j]);
    }
  }


  for (int i = 0; i < m; i++) {
    selectionSort(matrix[i], n);
  }


  printf("Matrix in column-major order after sorting
  rows:\n"); printMatrixColumnMajor(matrix, m, n);

  return 0;
}
```

Output:



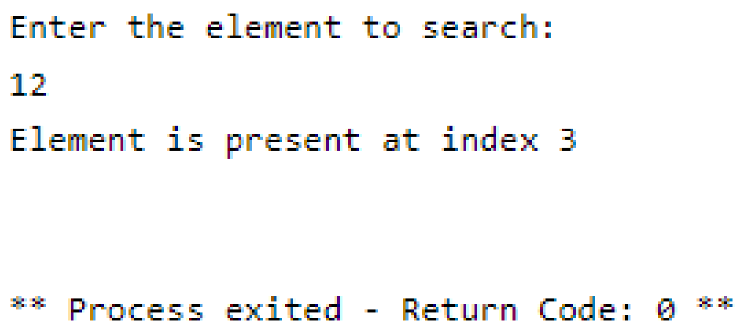
```
Enter the number of rows (m):

3

Enter the number of columns (n):

4

Enter the matrix elements:

1 4 2 3 7 8 10 9 6 3 5 2

Matrix in column-major order after sorting rows:

1 7 2

2 8 3

3 9 5

4 10 6
```

Practical-9

1. Write a c program to perform linear Search.

```
#include <stdio.h>

int linearSearch(int arr[], int n, int
  x) { for (int i = 0; i < n; i++) {
    if (arr[i] ==
      x) {
      return i;
    }
  }
  return -1;
}

int main() {
  int arr[] = {64, 34, 25, 12, 22, 11, 90};
  int n = sizeof(arr) /
  sizeof(arr[0]); int x;

  printf("Enter the element to
  search: "); scanf("%d", &x);
  int result = linearSearch(arr,


  n, x); if (result == -1) {


    printf("Element is not present in array\n");
  } else {
    printf("Element is present at index %d\n", result);
  }

  return 0;
}
```

Output:

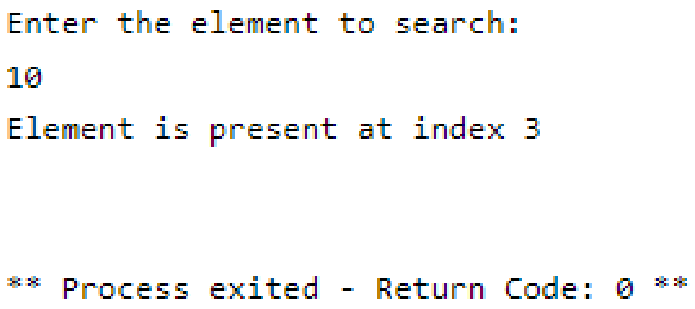


```
Enter the element to search:
12
Element is present at index 3


** Process exited - Return Code: 0 **
```

2. Write a c program to perform binary search.

```
#include <stdio.h>

int binarySearch(int arr[], int left, int right, int
  x) { while (left <= right) {
    int mid = left + (right - left) / 2;


    if (arr[mid] ==
      x) { return
      mid;
    }


    if (arr[mid] <
      x) { left =
      mid + 1;
    }


    else {
      right = mid - 1;
    }
  }
  return -1;
}

int main() {
  int arr[] = {2, 3, 4, 10, 40};
  int n = sizeof(arr) /
  sizeof(arr[0]); int x;

  printf("Enter the element to
  search: "); scanf("%d", &x);
  int result = binarySearch(arr, 0, n -


  1, x); if (result == -1) {


    printf("Element is not present in array\n");
  } else {
    printf("Element is present at index %d\n", result);
  }

  return 0;
}
```

Output:



```
Enter the element to search:
10
Element is present at index 3



** Process exited - Return Code: 0 **
```

Practical-10

Write a c program to Create a single Linked list and perform Following Operations

A. Insertion At Beginning
B. Insertion At End
C. Insertion After a particular node
D. Insertion Before a particular node
E. Insertion at specific position
F. Search a particular node
G. Return a particular node
H. Deletion at the beginning
I. Deletion at the end
J. Deletion after a particular node
K. Deletion before a particular node
L. Delete a particular node
M. Deletion at a specific position

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

// Define the structure for a node in the linked
list struct Node {
   int data;
   struct Node* next;
};

// Function to create a new
node struct Node*
createNode(int data) {
   struct Node* newNode = (struct Node*)malloc(sizeof(struct
   Node)); if (newNode == NULL) {
```

```
printf("Memory allocation failed.\n");
```

```
        exit(1);
    }
    newNode->data =
    data; newNode->next
    = NULL; return
    newNode;
}

// Function to insert a node at the beginning of the
list void insertAtBeginning(struct Node** head, int
data) {
    struct Node* newNode =
    createNode(data); newNode->next =
    *head;
    *head = newNode;
}

// Function to insert a node at the end of the
list void insertAtEnd(struct Node** head, int
data) {
    struct Node* newNode =
    createNode(data); if (*head == NULL) {
        *head =
        newNode; return;
    }
    struct Node* temp =
    *head; while (temp->next
    != NULL) {
        temp = temp->next;
    }
    temp->next = newNode;
}

// Function to insert a node after a particular node
void insertAfterNode(struct Node* prevNode, int
    data) { if (prevNode == NULL) {
        printf("Previous node cannot be
    NULL.\n"); return;
}
struct Node* newNode =
createNode(data); newNode->next =
prevNode->next;
```

```
    prevNode->next = newNode;
}

// Function to insert a node before a particular node
void insertBeforeNode(struct Node** head, struct Node* nextNode, int
    data) { if (nextNode == NULL) {
        printf("Next node cannot be
        NULL.\n"); return;
    }
    struct Node* newNode =
    createNode(data); if (*head ==
    nextNode) {
        newNode->next = *head;
        *head =
        newNode; return;
    }
    struct Node* temp = *head;
    while (temp->next !=
    nextNode) {
        temp = temp->next;
    }
    newNode->next =
    nextNode; temp->next =
    newNode;
}

// Function to insert a node at a specific position
void insertAtPosition(struct Node** head, int position, int
    data) { if (position < 0) {
        printf("Invalid
        position.\n"); return;
    }
    if (position == 0) {
        insertAtBeginning(head, data);
        return;
    }
```

```
struct Node* newNode =
createNode(data); struct Node* temp =
*head;
for (int i = 0; i < position - 1 && temp != NULL; i++) {
        temp = temp->next;
    }
    if (temp == NULL) {
        printf("Position out of
        range.\n"); return;
    }
    newNode->next =
    temp->next; temp->next =
    newNode;
}

// Function to search for a particular node
struct Node* searchNode(struct Node* head, int
    key) { while (head != NULL) {
        if (head->data ==
            key) return head;
        head = head->next;
    }
    return NULL;
}

// Function to delete the first node
void deleteAtBeginning(struct Node**
    head) { if (*head == NULL) {
        printf("List is empty, deletion not
        possible.\n"); return;
    }
    struct Node* temp = *head;
    *head = (*head)->next;
    free(temp);
}

// Function to delete the last node
void deleteAtEnd(struct Node**
    head) { if (*head == NULL) {
        printf("List is empty, deletion not
        possible.\n"); return;
    }
    if ((*head)->next ==
        NULL) { free(*head);
        *head =
        NULL;
        return;
    }
    struct Node* secondLast = *head;
    while (secondLast->next->next !=
        NULL) { secondLast =
        secondLast->next;
    }
    free(secondLast->next)
    ; secondLast->next =
    NULL;
}

// Function to delete a node after a particular
node void deleteAfterNode(struct Node*
prevNode) {
    if (prevNode == NULL || prevNode->next ==
        NULL) { printf("No node to delete.\n");
        return;
    }
    struct Node* temp =
    prevNode->next; prevNode->next
    = temp->next; free(temp);
}

// Function to delete a node before a particular node
void deleteBeforeNode(struct Node** head, struct Node* nextNode) {
    if (*head == NULL || *head == nextNode || (*head)->next ==
        nextNode) { printf("No node to delete.\n");
        return;
    }
    struct Node* temp = *head;
    while (temp->next->next !=
        nextNode) { temp = temp->next;
    }
    struct Node* nodeToDelete = temp->next;
```

```
    temp->next = nextNode;
    free(nodeToDelete);
}

// Function to delete a particular node
void deleteNode(struct Node** head, struct Node*
    keyNode) { if (*head == NULL) {
        printf("List is empty, deletion not
        possible.\n"); return;
    }
    if (*head == keyNode) {
        deleteAtBeginning(he
        ad); return;
    }
    struct Node* temp = *head;
    while (temp->next !=
    keyNode) {
        temp =
        temp->next; if
        (temp == NULL)
        {
            printf("Node not found in the
            list.\n"); return;
        }
    }
    temp->next =
    keyNode->next;
    free(keyNode);
}

// Function to delete a node at a specific position
void deleteAtPosition(struct Node** head, int
    position) { if (*head == NULL || position < 0) {
        printf("List is empty or position is
        invalid.\n"); return;
    }
    if (position == 0) {
```

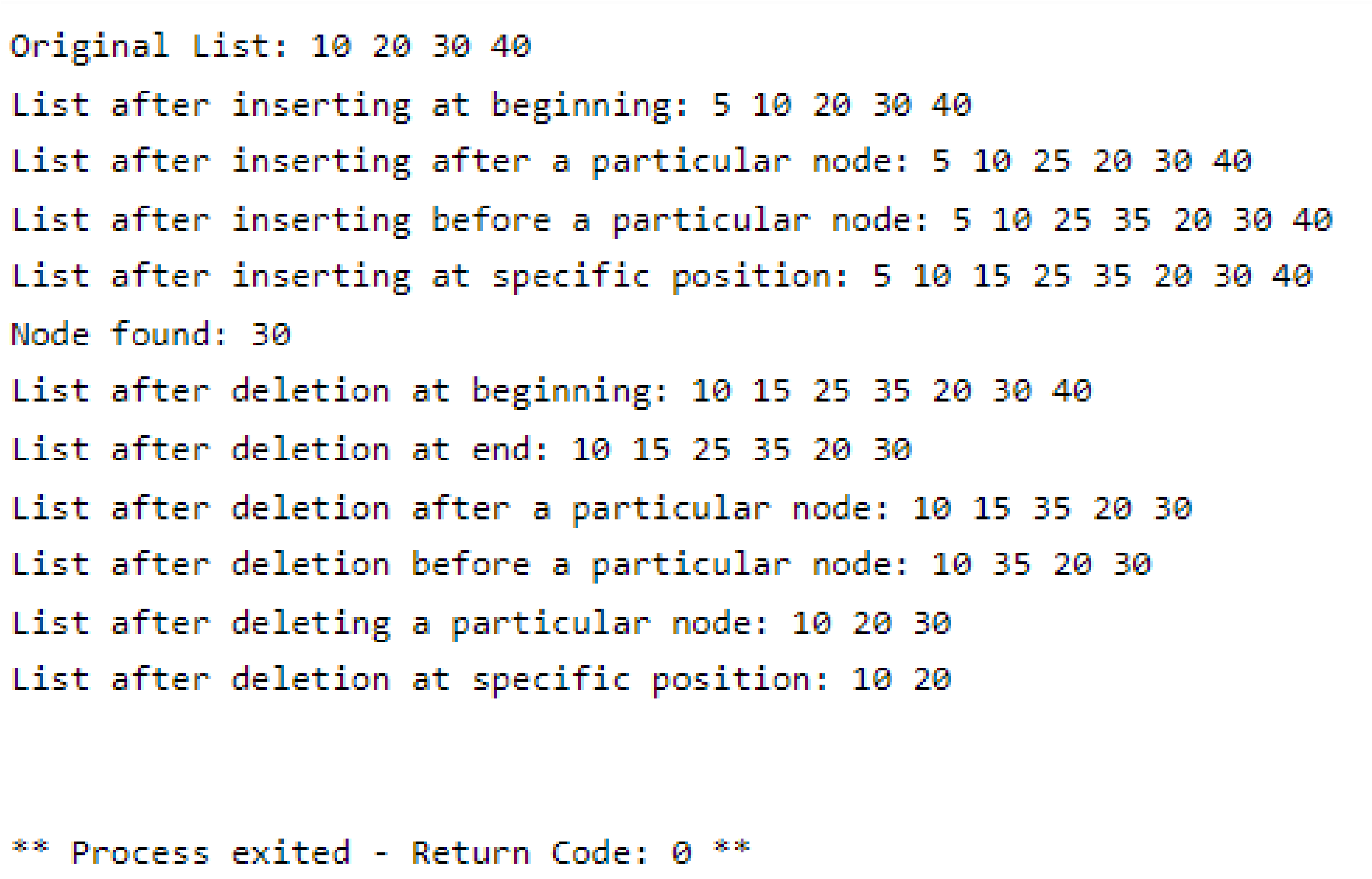
```
    deleteAtBeginning(he
    ad); return;
}
```

```
    struct Node* temp = *head;
    for (int i = 0; temp != NULL && i < position - 1;
        i++) { temp = temp->next;
    }
    if (temp == NULL || temp->next ==
        NULL) { printf("Position out of
        range.\n");
        return;
    }
    struct Node* nodeToDelete =
    temp->next; temp->next =
    nodeToDelete->next;
    free(nodeToDelete);
}

// Function to print the linked
list void printList(struct Node*
head) {
    while (head != NULL) {
        printf("%d ",
        head->data); head =
        head->next;
    }
    printf("\n");
}

// Function to free the memory allocated to the linked
list void freeList(struct Node** head) {
    struct Node* temp;
    while (*head !=
    NULL) {
        temp = *head;
        *head = (*head)->next;
        free(temp);
    }
}
```

```
int main() {
   struct Node* head = NULL;
   insertAtBeginning(&head,
   10);
   insertAtEnd(&head, 20);
insertAtEnd(&head, 30);
insertAtEnd(&head, 40);
printf("Original List: ");
printList(head);

// Perform operations
insertAtBeginning(&head,
5);
printf("List after inserting at
beginning: "); printList(head);

insertAfterNode(head->next, 25);
printf("List after inserting after a particular
node: "); printList(head);

insertBeforeNode(&head, head->next->next->next, 35);
printf("List after inserting before a particular node: ");
printList(head);

insertAtPosition(&head, 2, 15);
printf("List after inserting at specific
position: "); printList(head);

struct Node* searchedNode = searchNode(head,
30); if (searchedNode != NULL)
   printf("Node found: %d\n",
searchedNode->data); else
   printf("Node not found.\n");

deleteAtBeginning(&head);
printf("List after deletion at
beginning: "); printList(head);

deleteAtEnd(&head);
printf("List after deletion at
end: "); printList(head);
```

```
    deleteAfterNode(head->next);
    printf("List after deletion after a particular node: ");
    printList(head);

    deleteBeforeNode(&head, head->next->next);
    printf("List after deletion before a particular
    node: "); printList(head);

    deleteNode(&head, head->next);
    printf("List after deleting a particular
    node: "); printList(head);

    deleteAtPosition(&head, 2);
    printf("List after deletion at specific
    position: "); printList(head);

    // Free the memory allocated to the linked
    list freeList(&head);

    return 0;
}
```

Output:



```
Original List: 10 20 30 40
List after inserting at beginning: 5 10 20 30 40
List after inserting after a particular node: 5 10 25 20 30 40
List after inserting before a particular node: 5 10 25 35 20 30 40
List after inserting at specific position: 5 10 15 25 35 20 30 40
Node found: 30
List after deletion at beginning: 10 15 25 35 20 30 40
List after deletion at end: 10 15 25 35 20 30
List after deletion after a particular node: 10 15 35 20 30
List after deletion before a particular node: 10 35 20 30
List after deleting a particular node: 10 20 30
List after deletion at specific position: 10 20


** Process exited - Return Code: 0 **
```

Practical-11

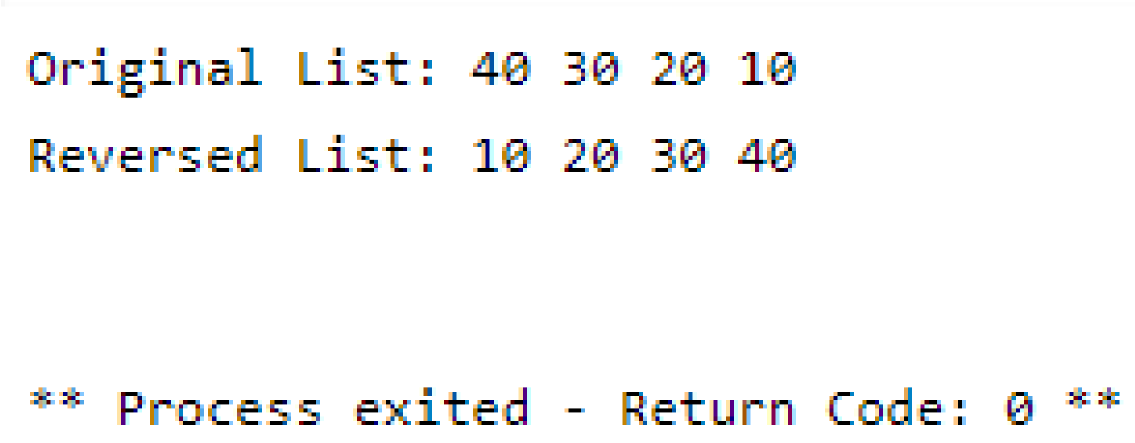
1. Write a program to Reverse a singly Linked list.

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

struct Node
   { int
   data;
   struct Node* next;
};

struct Node* createNode(int data) {
   struct Node* newNode = (struct Node*)malloc(sizeof(struct
   Node)); if (newNode == NULL) {
      printf("Memory allocation failed.\n");
      exit(1);
   }
   newNode->data =
   data; newNode->next
   = NULL; return
   newNode;
}

void insertAtBeginning(struct Node** head, int
   data) { struct Node* newNode =
   createNode(data); newNode->next = *head;
   *head = newNode;
}


void printList(struct Node*
   head) { while (head !=
   NULL) {
    printf("%d ",
    head->data); head =
    head->next;
}
printf("\n");
```

```
}


struct Node* reverseList(struct Node* head) {
   struct Node *prevNode = NULL, *currNode = head, *nextNode =
   NULL; while (currNode != NULL) {
      nextNode =
      currNode->next;
      currNode->next =
      prevNode; prevNode =
      currNode; currNode =
      nextNode;
   }
   return prevNode;
}

int main() {
   struct Node* head = NULL;
   insertAtBeginning(&head,
   10);
   insertAtBeginning(&head, 20);
   insertAtBeginning(&head, 30);
   insertAtBeginning(&head, 40);

   printf("Original List: ");
   printList(head);


   head = reverseList(head);

   printf("Reversed List: ");
   printList(head);


   struct Node* temp;
   while (head !=
   NULL) {
      temp = head;
}
        head = head->next;
        free(temp);
```

```
    }

    return 0;
}
```

Output:



```
Original List: 40 30 20 10
Reversed List: 10 20 30 40



** Process exited - Return Code: 0 **
```

2. Write a c program to check whether the created linked list is palindrome or not

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


struct
    Node {
    char
    data;
    struct Node* next;
};
```

```
struct Node* createNode(char data) {
   struct Node* newNode = (struct Node*)malloc(sizeof(struct
   Node)); if (newNode == NULL) {
      printf("Memory allocation failed.\n");
      exit(1);
   }
   newNode->data =
   data; newNode->next
   = NULL; return
   newNode;
}


void insertAtEnd(struct Node** head, char
   data) { struct Node* newNode =
   createNode(data);
   if (*head == NULL) {
      *head =
      newNode; return;
   }
   struct Node* temp =
   *head; while (temp->next
   != NULL) {
      temp = temp->next;
   }
   temp->next = newNode;
}


void printList(struct Node*
   head) { while (head !=
   NULL) {
      printf("%c ", head->data);
      head = head->next;
   }
   printf("\n");
}
```

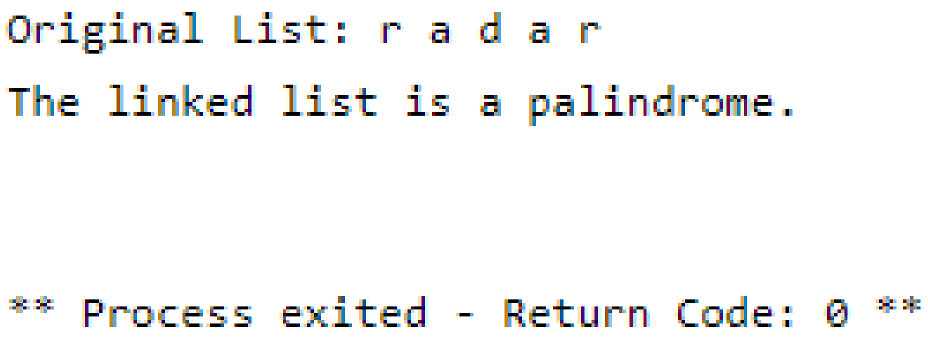
```
struct Node* reverseList(struct Node*
   head) { struct Node* prevNode =
   NULL;
```

```
    struct Node* currNode =
    head; while (currNode !=
    NULL) {
        struct Node* nextNode =
        currNode->next; currNode->next =
        prevNode;
        prevNode =
        currNode;
        currNode =
        nextNode;
    }
    return prevNode;
}

bool isPalindrome(struct Node* head) {
    if (head == NULL || head->next ==
        NULL) { return true;
    }



    struct Node* slow =
    head; struct Node*
    fast = head;
    while (fast->next != NULL && fast->next->next !=
        NULL) { slow = slow->next;
        fast = fast->next->next;
    }
    struct Node* secondHalf =
    reverseList(slow->next); struct Node* firstHalf =
    head;
    while (secondHalf != NULL) {
        if (firstHalf->data !=
            secondHalf->data) { secondHalf =
            reverseList(secondHalf); return
            false;
        }
        firstHalf = firstHalf->next;
        secondHalf = secondHalf->next;
    }
```

```
secondHalf = reverseList(secondHalf);
```

```
    return true;
}

void freeList(struct Node**
    head) { struct Node* temp;
    while (*head !=
        NULL) { temp =
        *head;
        *head = (*head)->next;
        free(temp);
    }
}

int main() {
    struct Node* head =
    NULL;
    insertAtEnd(&head, 'r');
    insertAtEnd(&head, 'a');
    insertAtEnd(&head, 'd');
    insertAtEnd(&head, 'a');
    insertAtEnd(&head, 'r');

    printf("Original List: ");
    printList(head);

    if (isPalindrome(head))
        printf("The linked list is a
    palindrome.\n"); else
        printf("The linked list is not a palindrome.\n");



    freeList(&head);

    return 0;
}
```

Output:



```
Original List: r a d a r
The linked list is a palindrome.



** Process exited - Return Code: 0 **
```

Practical-12

Write a c program to Create a Circular Linked list and perform Following Operations
A. Insertion At Beginning
B. Insertion At End
C. Insertion After a particular node
D. Insertion Before a particular node
E. Insertion at specific position
F. Search a particular node
G. Return a particular node
H. Deletion at the beginning
I. Deletion at the end
J. Deletion after a particular node
K. Deletion before a particular node
L. Delete a particular node
M. Deletion at a specific position

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

struct Node {
   int data;
   struct Node* next;
};

struct Node* createNode(int data) {
   struct Node* newNode = (struct Node*)malloc(sizeof(struct Node)); if
   (newNode == NULL) {
      printf("Memory allocation failed.\n"); exit(1);
   }
   newNode->data =
   data; newNode->next
   = NULL; return
   newNode;
```

```
}

void insertAtBeginning(struct Node** head, int data) {
   struct Node* newNode = createNode(data);
   if (*head == NULL) {
      *head = newNode; newNode-
      >next = newNode; return;
   }
   struct Node* temp = *head;
   while (temp->next != *head) {
      temp = temp->next;
   }
   temp->next =
   newNode;
   newNode->next =
   *head;
   *head = newNode;
}


void insertAtEnd(struct Node** head, int
   data) { struct Node* newNode =
   createNode(data); if (*head == NULL) {
      *head = newNode;
      newNode->next = newNode; // For
      circularity return;
   }
   struct Node* temp = *head;
   while (temp->next != *head) {
      temp = temp->next;
   }
   temp->next =
   newNode;
   newNode->next =
   *head;
}

void insertAfterNode(struct Node* prevNode, int data) {
    if (prevNode == NULL) {
        printf("Previous node cannot be NULL.\n"); return;
    }
    struct Node* newNode = createNode(data);
    newNode->next = prevNode->next;
    prevNode-
    >next = newNode;
}


void insertBeforeNode(struct Node** head, struct Node* nextNode, int
    data) { if (*head == NULL || nextNode == NULL) {
        printf("List is empty or next node cannot be
        NULL.\n"); return;
    }
    struct Node* newNode =
    createNode(data); struct Node* temp =
    *head;
    while (temp->next != nextNode && temp->next != *head) {
        temp = temp->next;
    }
    if (temp->next == *head) {
        printf("Next node not found in the list.\n");
        return;
    }
    newNode->next =
    nextNode; temp->next =
    newNode;
if (temp == *head) {
   *head = newNode;
    }
}

void insertAtPosition(struct Node** head, int position, int data) {
    if (position < 0) {
        printf("Invalid position.\n");
        return;
    }
    if (position == 0) {
        insertAtBeginning(head,
        data); return;
    }
    struct Node* newNode =
    createNode(data); struct Node* temp =
    *head;
    for (int i = 0; i < position - 1 && temp != NULL;
        i++) { temp = temp->next;
    }
    if (temp == NULL) {
        printf("Position out of
        range.\n"); return;
    }
    newNode->next =
    temp->next; temp->next
    = newNode;
}

struct Node* searchNode(struct Node* head, int key) {
    if (head == NULL) {
        printf("List is empty.\n");
        return NULL;
    }
    struct Node* temp = head; do
    {
        if (temp->data ==
            key) { return
            temp;
        }
            temp = temp->next;
      } while (temp != head);
    printf("Node not found in the list.\n");
    return NULL;
}

struct Node* getNodeAtPosition(struct Node* head, int position) {
    if (head == NULL || position < 0) {
        printf("List is empty or invalid position.\n"); return
        NULL;
    }
    struct Node* temp =
    head; int count = 0;
    do {
        if (count ==
            position) { return
            temp;
        }
        temp =
        temp->next;
        count++;
    } while (temp != head);
    printf("Position out of
    range.\n"); return NULL;
}

void deleteAtBeginning(struct Node** head)
    { if (*head == NULL) {
        printf("List is empty, deletion not possible.\n"); return;
    }
    struct Node* temp =
    *head; if (temp->next ==
    *head) {
        *head = NULL;
    } else {
        struct Node* lastNode = *head;
        while (lastNode->next != *head)
        {
            lastNode = lastNode->next;
        }
        *head =
        temp->next;
        lastNode->next =
        *head;
    }
    free(temp);
```

```
}
```

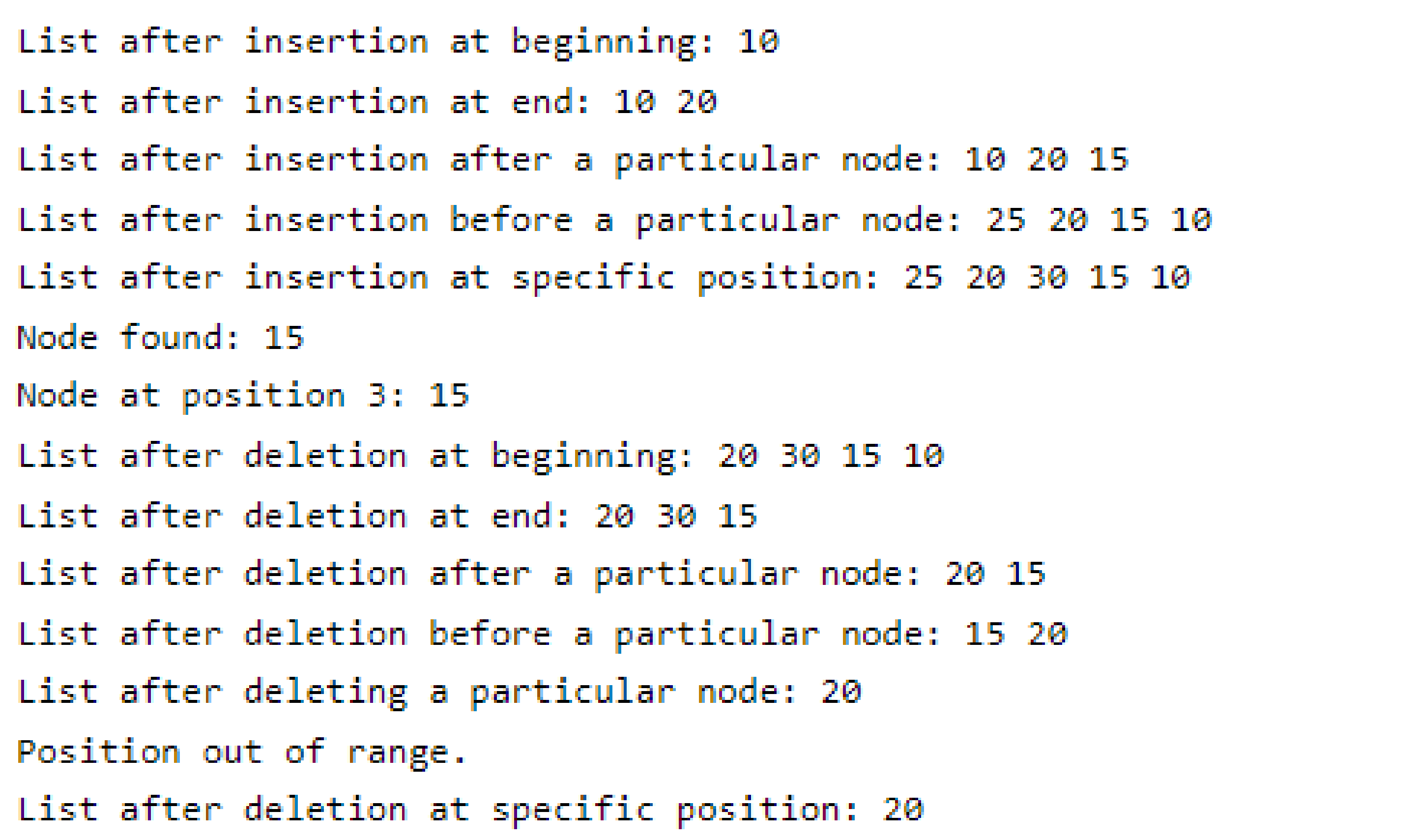
```
void deleteAtEnd(struct Node** head)
   { if (*head == NULL) {
      printf("List is empty, deletion not possible.\n"); return;
   }
   struct Node* temp = *head;
   struct Node* prevNode =
   NULL; while (temp->next !=
   *head) {
      prevNode =
      temp; temp =
      temp->next;
   }
   if (temp == *head) {
      *head = NULL;
   } else {
      prevNode->next = *head;
   }
   free(temp);
}

void deleteAfterNode(struct Node* prevNode) {
   if (prevNode == NULL || prevNode->next ==
      NULL) { printf("No node to delete.\n");
      return;
   }
   struct Node* temp =
   prevNode->next;
   prevNode->next = temp->next;
   free(temp);
}

void deleteBeforeNode(struct Node** head, struct Node* nextNode) { if
   (*head == NULL || nextNode == NULL || nextNode->next == NULL) {
      printf("No node to
      delete.\n"); return;
   }
```

```
    struct  Node*  temp  =  *head;
    struct   Node*   prevNode   =
    NULL;  while  (temp->next  !=
    nextNode) {
      prevNode = temp;
      temp =
      temp->next; if
      (temp == *head) {
        printf("Node not found in the list.\n"); return;
      }
    }
if (temp == *head) {
   *head = nextNode;
    } else {
      prevNode->next = nextNode;
    }
    free(temp);
  }

  void deleteNode(struct Node** head, int key)
    { if (*head == NULL) {
      printf("List is empty, deletion not possible.\n"); return;
    }
    struct Node* temp = *head;
    struct Node* prevNode =
    NULL; while (temp->data !=
    key) {
      prevNode = temp;
      temp =
      temp->next; if
      (temp == *head) {
        printf("Node not found in the list.\n"); return;
      }
    }
    if (temp == *head) {
      if (temp->next == *head) {
            *head = NULL;
        } else {
            struct Node* lastNode = *head;
            while (lastNode->next !=
            *head) {
                lastNode = lastNode->next;
            }
            *head =
            temp->next;
            lastNode->next =
            *head;
        }
    } else {
        prevNode->next = temp->next;
    }
    free(temp);
}


void deleteAtPosition(struct Node** head, int
    position) { if (*head == NULL || position < 0) {
        printf("List is empty or position is invalid.\n"); return;
    }
    if (position == 0) {
        deleteAtBeginning(he
        ad); return;
    }
    struct Node* temp = *head;
    struct Node* prevNode =
    NULL; int count = 0;
    do {
        if (count ==
            position) { if
            (temp == *head) {
                if (temp->next == *head) {
                    *head = NULL;
                } else {
                    struct Node* lastNode = *head;
                    while (lastNode->next != *head) {
                        lastNode = lastNode->next;
                    }
                    *head =
                    temp->next;
                    lastNode->next =
                    *head;
                }
            } else {
                prevNode->next = temp->next;
            }
            free(te
            mp);
            return;
        }
        prevNode =
        temp; temp =
        temp->next;
        count++;
    } while (temp != *head);
    printf("Position out of
    range.\n");
}

void printList(struct Node* head) { if
    (head == NULL) {
        printf("List is empty.\n"); return;
    }
    struct Node* temp = head; do
    {
        printf("%d ", temp->data); temp
        = temp->next;
    } while (temp != head);
    printf("\n");
}

void freeList(struct Node** head) { if
    (*head == NULL) {
        return;
    }
    struct Node* temp = *head;
    struct Node* prevNode =
    NULL; do {
        prevNode =
        temp; temp =
        temp->next;
        free(prevNode)
        ;
    } while (temp != *head);
    *head = NULL;
}

int main() {
    struct Node* head = NULL;

    // Perform operations
    insertAtBeginning(&head, 10);
    printf("List after insertion at
    beginning: "); printList(head);

    insertAtEnd(&head, 20);
    printf("List after insertion at
    end: "); printList(head);

    struct Node* secondNode = head->next;
    insertAfterNode(secondNode, 15);
    printf("List after insertion after a particular node: "); printList(head);

    insertBeforeNode(&head, secondNode, 25); printf("List
    after insertion before a particular node: ");
    printList(head);

    insertAtPosition(&head, 2, 30);
    printf("List after insertion at specific position: "); printList(head);
```

```
struct Node* searchedNode = searchNode(head, 15); if
(searchedNode != NULL) {
   printf("Node found: %d\n", searchedNode->data);
}

struct Node* returnedNode = getNodeAtPosition(head, 3); if
(returnedNode != NULL) {
   printf("Node at position 3: %d\n", returnedNode->data);
}

deleteAtBeginning(&head);
printf("List after deletion at beginning: ");
printList(head);

deleteAtEnd(&head);
printf("List after deletion at end: ");
printList(head);

deleteAfterNode(secondNode);
printf("List after deletion after a particular node: "); printList(head);

deleteBeforeNode(&head, secondNode->next);
printf("List after deletion before a particular node: ");
printList(head);

deleteNode(&head, 15);
printf("List after deleting a particular node:
"); printList(head);

deleteAtPosition(&head, 2);
printf("List after deletion at specific position: "); printList(head);
```

```
    freeList(&head);

    return 0;
}
```

Output:



```
List after insertion at beginning: 10
List after insertion at end: 10 20
List after insertion after a particular node: 10 20 15
List after insertion before a particular node: 25 20 15 10
List after insertion at specific position: 25 20 30 15 10
Node found: 15
Node at position 3: 15
List after deletion at beginning: 20 30 15 10
List after deletion at end: 20 30 15
List after deletion after a particular node: 20 15
List after deletion before a particular node: 15 20
List after deleting a particular node: 20
Position out of range.
List after deletion at specific position: 20
```

Practical-13

Write a c program to Create a Circular single Linked list and perform Following Operations

A. Insertion After a particular node
B. Insertion Before a particular node
C. Search a particular node
D. Return a particular node
E. Deletion before a particular node
F. Delete a particular node

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


struct Node
   { int
   data;
   struct Node* next;
};


struct Node* createNode(int value) {
   struct Node* newNode = (struct Node*)malloc(sizeof(struct
   Node)); newNode->data = value;
   newNode->next =
   NULL; return
   newNode;
}


void insertAfter(struct Node* prevNode, int
   value) { if (prevNode == NULL) {
```

```
printf("Previous node cannot be NULL.\n");
```

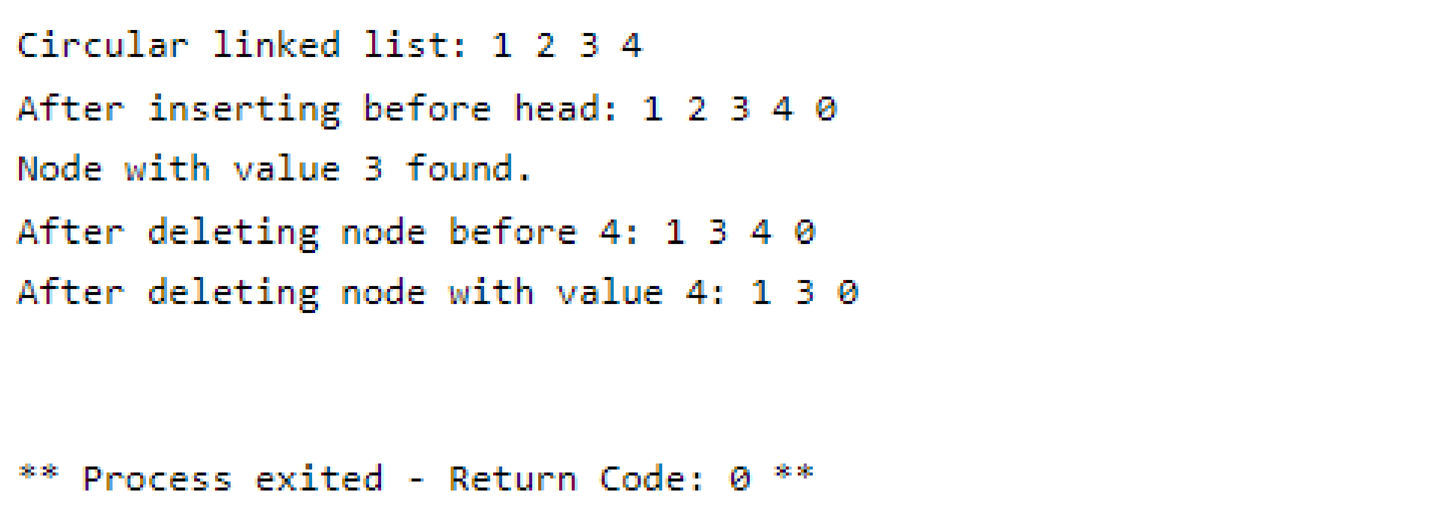
```
        return;
    }
    struct Node* newNode =
    createNode(value); newNode->next =
    prevNode->next; prevNode->next =
    newNode;
}


void insertBefore(struct Node** headRef, struct Node* nextNode, int value) {
    struct Node* newNode = createNode(value);
    if (*headRef == NULL) {
        *headRef = newNode;
        newNode->next =
        newNode; return;
    }
    struct Node* current =
    *headRef; while (current->next
    != nextNode) {
        current =
        current->next; if
        (current == *headRef)
        {
            printf("Node not found.\n");
            return;
        }
    }
    newNode->next =
    nextNode; current->next
    = newNode;
    if (current == *headRef)
        *headRef = newNode;
}


struct Node* searchNode(struct Node* head, int
    key) { if (head == NULL)
        return NULL;
```

```
struct Node* current =
head; do {
   if (current->data == key)
```

```
        return current;
      current =
      current->next;
   } while (current !=
   head); return NULL;
}


void deleteBefore(struct Node** headRef, struct Node*
   nextNode) { if (*headRef == NULL || (*headRef)->next ==
   nextNode) {
      printf("No node to delete before the given
      node.\n"); return;
   }
   struct Node* current = *headRef;
   while (current->next->next !=
      nextNode) { current =
      current->next;
      if (current->next ==
         *headRef) { printf("Node
         not found.\n"); return;
      }
   }
   struct Node* temp =
   current->next; current->next =
   nextNode; free(temp);
}


void deleteNode(struct Node** headRef, struct Node*
   delNode) { if (*headRef == NULL) {
      printf("List is empty.\n");
      return;
   }
   struct Node* current = *headRef;

   if (current == delNode) {
      struct Node* temp =
*headRef; while
(temp->next != *headRef)
```

```
            temp = temp->next;
        temp->next = (*headRef)->next;
        *headRef = (*headRef)->next;
        free(current);
        return;
    }

    while (current->next !=
        delNode) { current =
        current->next;
        if (current->next ==
            *headRef) { printf("Node
            not found.\n"); return;
        }
    }
    current->next =
    delNode->next;
    free(delNode);
}


void displayList(struct Node*
    head) { if (head == NULL) {
        printf("List is empty.\n");
        return;
    }
    struct Node* current =
    head; do {
        printf("%d ",
        current->data); current =
        current->next;
    } while (current !=
    head); printf("\n");
}

int main() {
    struct Node* head = NULL;
    head = createNode(1);
    head->next = head; // Circular reference

    insertAfter(head, 2);
    insertAfter(head->next, 3);
    insertAfter(head->next->next, 4);

    printf("Circular linked list: ");
    displayList(head);


    insertBefore(&head, head, 0);
    printf("After inserting before
    head: "); displayList(head);


    int key = 3;
    struct Node* foundNode = searchNode(head,
    key); if (foundNode != NULL)
        printf("Node with value %d found.\n",
    key); else
        printf("Node with value %d not found.\n", key);


    deleteBefore(&head, head->next->next);
    printf("After deleting node before 4: ");
    displayList(head);


    deleteNode(&head,
    head->next->next); printf("After
    deleting node with value 4: ");
    displayList(head);

    return 0;
}
```

Output:



```
Circular linked list: 1 2 3 4
After inserting before head: 1 2 3 4 0
Node with value 3 found.
After deleting node before 4: 1 3 4 0
After deleting node with value 4: 1 3 0


** Process exited - Return Code: 0 **
```

Practical-14

Write a c program to Create a Circular DoubleLinked list and perform Following Operations

A. Insertion After a particular node
B. Insertion Before a particular node
C. Search a particular node
D. Return a particular node
E. Deletion before a particular node
F. Delete a particular node

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


struct Node
   { int
   data;
   struct Node*
   next; struct
   Node* prev;
};


struct Node* createNode(int data);
void insertAfter(struct Node** head_ref, int value, int
key); void insertBefore(struct Node** head_ref, int
value, int key); struct Node* searchNode(struct Node*
head, int key); struct Node* returnNode(struct Node*
head, int key);
void deleteBefore(struct Node** head_ref, int
key); void deleteNode(struct Node** head_ref,
int key); void displayList(struct Node* head);
```

```
struct Node* createNode(int data) {
    struct  Node*  newNode  =  (struct  Node*)malloc(sizeof(struct
    Node)); newNode->data = data;
    newNode->next       =
    NULL;
    newNode->prev       =
    NULL;               return
    newNode;
}


void insertAfter(struct Node** head_ref, int value, int
    key) { struct Node* newNode = createNode(value);
    if (*head_ref == NULL) {
        printf("List is empty.\n");
        return;
    }
    struct Node* temp =
    *head_ref; while
    (temp->data != key) {
        temp = temp->next;
        if (temp == *head_ref) {
            printf("Key not found in the
            list.\n"); return;
        }
    }
    newNode->prev = temp;
    newNode->next =
    temp->next;
    temp->next->prev =
    newNode; temp->next =
    newNode;
}


void insertBefore(struct Node** head_ref, int value, int
    key) { struct Node* newNode = createNode(value);
    if (*head_ref == NULL) {
        printf("List is empty.\n");
    return;
}
struct Node* temp = *head_ref;
```

```
    while (temp->data !=
        key) { temp =
        temp->next;
        if (temp == *head_ref) {
            printf("Key not found in the
            list.\n"); return;
        }
    }
    newNode->prev =
    temp->prev;
    newNode->next = temp;
    temp->prev->next =
    newNode; temp->prev =
    newNode;
    if (temp == *head_ref)
        *head_ref = newNode;
}


struct Node* searchNode(struct Node* head, int
    key) { if (head == NULL) {
        printf("List is empty.\n");
        return NULL;
    }
    struct Node* temp =
    head; do {
        if (temp->data ==
            key) return temp;
   temp = temp->next;
} while (temp != head);
    printf("Key not found in the
    list.\n"); return NULL;
}


struct Node* returnNode(struct Node* head, int
    key) { return searchNode(head, key);
}
```

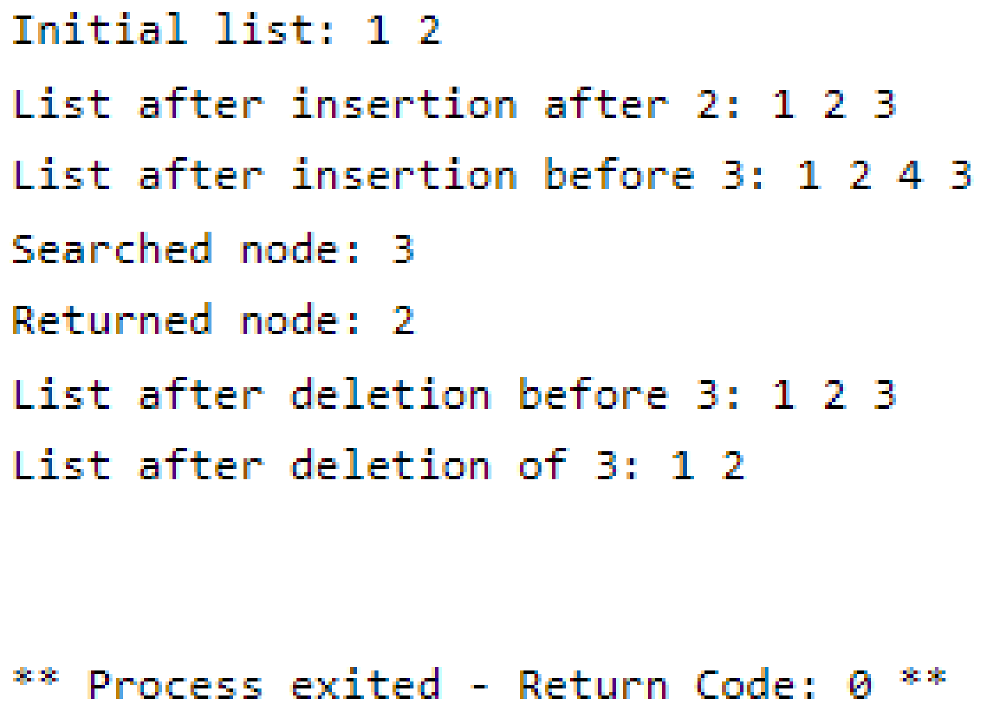
```
void deleteBefore(struct Node** head_ref, int
   key) { if (*head_ref == NULL) {
      printf("List is empty.\n");
      return;
   }
   struct Node* temp =
   *head_ref; while
   (temp->data != key) {
      temp = temp->next;
      if (temp == *head_ref) {
         printf("Key not found in the
         list.\n"); return;
      }
   }
   struct Node* delNode =
   temp->prev;
   delNode->prev->next = temp;
   temp->prev =
   delNode->prev; if
   (delNode == *head_ref)
      *head_ref =
   temp;
   free(delNode);
}


void deleteNode(struct Node** head_ref, int
   key) { if (*head_ref == NULL) {
      printf("List is empty.\n");
      return;
   }
   struct Node* temp =
   *head_ref; while
   (temp->data != key) {
      temp = temp->next;
      if (temp == *head_ref) {
         printf("Key not found in the
        list.\n"); return;
    }
}
```

```
    if (temp == *head_ref) {
        *head_ref = temp->next;
    }
    temp->prev->next                =
    temp->next;
    temp->next->prev                =
    temp->prev; free(temp);
}



void displayList(struct Node*
    head) { if (head == NULL) {
        printf("List is empty.\n");
        return;
    }
    struct Node* temp =
    head; do {
        printf("%d ",
        temp->data); temp =
        temp->next;
    } while (temp !=
    head); printf("\n");
}

int main() {
    struct Node* head = NULL;


    head = createNode(1);
    head->next =
    createNode(2);
    head->next->prev =
    head; head->next->next
    = head; head->prev =
    head->next;


    printf("Initial list: ");
```

```
displayList(head);
```

```
    insertAfter(&head, 3, 2);
    printf("List after insertion after
    2: "); displayList(head);


    insertBefore(&head, 4, 3);
    printf("List after insertion before
    3: "); displayList(head);


    struct Node* searchedNode = searchNode(head,
    3); printf("Searched node: %d\n",
    searchedNode->data);


    struct Node* returnedNode = returnNode(head,
    2); printf("Returned node: %d\n",
    returnedNode->data);


    deleteBefore(&head, 3);
    printf("List after deletion before
    3: "); displayList(head);


    deleteNode(&head, 3);
    printf("List after deletion of
    3: "); displayList(head);

    return 0;
}
```

Output:



```
Initial list: 1 2
List after insertion after 2: 1 2 3
List after insertion before 3: 1 2 4 3
Searched node: 3
Returned node: 2
List after deletion before 3: 1 2 3
List after deletion of 3: 1 2


** Process exited - Return Code: 0 **
```