LAB MANUAL

DAA Lab

CSE-351P

B.Tech(CSE)



School of Computing Science & Engineering

Version0.1 Date: Drafted By: Lalita Verma

Table of Contents

|  |  |  |
| --- | --- | --- |
| 1. | Course details  1.1.Courseobjective (s) |  |
|  | 1.2. Pre-requisites  1.3.LearningOutcomes |  |
|  | 1.4.Syllabus &References |  |
| 2. | Experimental Setup details for the course. |  |
| 3. | Experiment details |  |

3.1 Write a program to sort given set of numbers in ascending/descending order using Bubble sort and also search a number using binary search.

3.2 Write a program to sort given set of numbers in ascending/descending order using Insertion sort and also search a number using linear search.

3.3 Write a program to sort given set of numbers in ascending/descending order using Quick sort and any other sorting algorithm. Also record the time taken by these two programs and compare them.

3.4 Write a program to sort given set of numbers using Heap sort.

3.5 Write a program to sort given set of numbers Merge Sort.

3.6 Write a program to sort given set of numbers Counting Sort.

3.7 Write a program to implement Strassen's Matrix Multiplication by Divide and Conquer

3.8 Write a program to implement Knapsack using Greedy technique.

3.9 Write a program to implement Knapsack using Dynamic programming.

3.10 Write a program to implement Dijkstra’s Algorithm.

3.11 Write a program to implement Bellman-Ford Algorithm.

3.12 Write a program to implement n-Queen Problem using backtracking.

3.13 Write a program to create B Tree

3.14 Write a program to insert a node in red-black tree.

3.15 Write a program to implement String Matching using Rabin-Karp algorithm

**1 COURSE DETAILS**

**1.1 Course Objective(s)**

Students will gain the ability to implement the computer graphics algorithms in ‘C’. The students will gain the ability to implement basic animation using ‘C’.

**1.2 Pre-requisite**

1. Programming experience of ‘C-programming language’(CSE111)

2. Design and Analysis of Algorithms(CSE351)

**1.3 Learning Outcome**

* To analyze the running time of asymptotic algorithm.
* To develop algorithms for sorting, searching, insertion and matching.
* To identify and apply the concept of computational intractability.
* Apply the algorithms and design techniques to solve problems
* Analyze the complexities of various problems in different domains.

**1.4 Syllabus and References**

**CSE 351 Designs and Analysis of ALGORITHM LAB**

**LIST OF EXPERIMENT**

|  |  |
| --- | --- |
| 1. Write a program to sort given set of numbers in ascending/descending order using Bubble sort and also search a number using binary search. 2. Write a program to sort given set of numbers in ascending/descending order using Insertion sort and also search a number using linear search. 3. Write a program to sort given set of numbers in ascending/descending order using Quick sort and any other sorting algorithm. Also record the time taken by these two programs and compare them. 4. Write a program to sort given set of numbers using Heap sort. 5. Write a program to sort given set of numbers Merge Sort. 6. Write a program to sort given set of numbers Counting Sort.   7. Write a program to implement Strassen's Matrix Multiplication by Divide and Conquer  8. Write a program to implement Knapsack using Greedy technique.  9. Write a program to implement Knapsack using Dynamic programming.  10. Write a program to implement Dijkstra’s Algorithm.  11. Write a program to implement Bellman-Ford Algorithm.  12. Write a program to implement n-Queen Problem using backtracking.  13. Write a program to create B Tree  14. Write a program to insert a node in red-black tree.  15. Write a program to implement String Matching using Rabin-Karp algorithm. | |
| References |  |
| 1. Thomas H. Coreman, Charles E. Leiserson and Ronald L. Rivest, “Introduction to  Algorithms”, Printice Hall of India.  2. RCT Lee, SS Tseng, RC Chang and YT Tsai, “Introduction to the Design and Analysis  of Algorithms”, Mc Graw Hill, 2005.  3. E. Horowitz & S Sahni, "Fundamentals of Computer Algorithms",  4. Berman, Paul,” Algorithms”, Cengage Learning.  5. Aho, Hopcraft, Ullman, “The Design and Analysis of Computer Algorithms” Pearson  Education, 2008. | |

**Mode of Evaluation:** Quiz/Program Execution/Written Examination

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | **Theory** | | **Laboratory** | | **Theory and laboratory** |
| **Components** | **Internal** | **SEE** | **Internal** | **SEE** |
| **Marks** | 50 | 50 | 50 | 50 |
| **Total Marks** | 100 | | 100 | |
| **Scaled Marks** | 75 | | 25 | | 100 |

**Relationship between the Course Outcomes(COs) and Program Outcomes (POs)**

|  |  |  |
| --- | --- | --- |
| **Mapping between Cos and Pos** | | |
| **Sl. No.** | **CourseOutcomes(COs)** | **Mapped**  **Program**  **Outcomes** |
| 1 | To analyze the running time of asymptotic algorithm. | 2,3 |
| 2 | To develop algorithms for sorting, searching, insertion and matching. | 2,3,9,12 |
| 3 | To identify and apply the concept of computational intractability. | 2,3 |
| 4 | Apply the algorithms and design techniques to solve problems   * Analyze the complexities of various problems in different domains. | 2,3,4,9 |
| 5 | Analyze the complexities of various problems in different domains. | 1,4 |

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **`ProgramOutcome→** | | EngineeringKnowledge | Problemanalysis | Design/developmentof solutions | Conductinvestigationsofcomplex problems | Moderntoolusage | Theengineerandsociety | Environmentandsustainability | Ethics | Individualorteamwork | Communication | Projectmanagementandfinance | Life-longLearning |
| Course  Code | CourseName | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
| CSE351 | Design and Analysis of Algorithm Lab | 2 | 3 | 3 | 2 |  |  |  |  | 1 |  |  | 1 |

1=addressed to small extent

2= addressed significantly

3=majorpart ofcourse

Theory

Lab

**Course Lab Outcomes Assessment**

Thelaboratorycomponentstronglycontributestowardstheprogramoutcome**Design/development of solutions-PO(3)**.This evaluation method will be used for the evaluation of lab and program outcomes of this course.

**Direct Measurement Report**

CSE351 Outcome (3) Report Form

**Measure**– percent of students scoring at least 70%marks in lab.

**Target– 70%ofstudents**

**2. EXPERIMENTALSETUPDETAILS FORTHECOURSE**

**Software Requirements**

TurboC2.0/ TurboC++3.0+

**Hardware Requirements**

No specific requirements. Any computer Hardware capable of running DOS can be

Used for this course.

**3. EXPERIMENTDETAILS**

|  |  |
| --- | --- |
| **Experiment No:1** | |
| Title | Write a program to sort given set of numbers in ascending/descending order using Bubble sort and also search a number using binary search. |
| Objective | 1.TostudyandImplement Bubble SortAlgorithm  2.TostudyandImplement Binary SearchAlgorithm |
| Pre-requisite | Knowledge of   * Array Data Structure |
| Algorithm | **Bubble Sort Algorithm:**  **Inputtothefunctionis array A: The array to be sorted**  procedure bubbleSort( A : list of sort able items )  repeat  swapped = false  for i = 1 to length(A) - 1 inclusive do:  /\* if this pair is out of order \*/  if A[i-1] > A[i] then  /\* swap them and remember something changed \*/  swap( A[i-1], A[i] )  swapped = true  end if  end for  until not swapped  end procedure  **Binary Search Algorithm:**  **Input to the algorithm is A: Array in which key is to searched, key: the key is to searched, imin: lower index of array, imax: upper index of array.**  int binary\_search(int A[],int key,int imin,int imax)  {  // test if array is empty  if(imax < imin)  // set is empty, so return value showing not found  return KEY\_NOT\_FOUND;  else  {  // calculate midpoint to cut set in half  int imid = midpoint(imin, imax);  // three-way comparison  if(A[imid]> key)  // key is in lower subset  return binary\_search(A, key, imin, imid-1);  elseif(A[imid]< key)  // key is in upper subset  return binary\_search(A, key, imid+1, imax);  else  // key has been found  return imid;  }  } |
| Source Code | #include<stdio.h>  #include<stdlib.h>  #include<conio.h>  void bubble\_sort(int a[],int);  void binary\_search(int a[],int, int);  void main()  {  int data[100],i,n,element,ch;  while(1)  {  clrscr();  printf("\*\*\*\*MENU\*\*\*\*\nEnter 1 to apply bubble sort\nEnter 2 to search an element using Binary Search\nEnter 3 for exit\nEnter your choice\n");  scanf("%d",&ch);  switch(ch)  {  case 1:  printf("Enter the number of elements in array \n");  scanf("%d",&n);  printf("Enter array elements\n");  for(i=0;i<n;i++)  {  printf("enter %d element\n",i+1);  scanf("%d",&data[i]);  }  bubble\_sort(data,n);  break;  case 2:  printf("Enter the number of elements in array \n");  scanf("%d",&n);  printf("Enter array elements\n");  for(i=0;i<n;i++)  {  printf("enter %d element\n",i+1);  scanf("%d",&data[i]);  }  bubble\_sort(data,n);  printf("Enter the element you want to search for\n");  scanf("%d",&element);  binary\_search(data,n,element);  break;  case 3:  exit(0);  default:  printf("You have entered wrong choice\n");  }  getch();  }  }  void bubble\_sort(int a[],int n)  {  int i,j,temp;  for(i=0;i<n;i++)  {  for(j=0;j<n-i-1;j++)  {  if(a[j]>a[j+1])  {  temp=a[j];  a[j]=a[j+1];  a[j+1]=temp;  }  }  }  printf("After sorting array elements are:- \n");  for(i=0;i<n;i++)  printf("%d\n",a[i]);  }  void binary\_search(int a[],int n,int element)  {  int beg,end,mid;  beg=0;  end=n-1;  while(beg<=end)  {  mid=(beg+end)/2;  if(element==a[mid])  {  printf("Element found at location %d\n",mid+1);  return;  }  else if(element>a[mid])  beg=mid+1;  else  end=mid-1;  }  printf("Element not found\n");  return;  } |
| Sample Output |  |
| Post Lab Assignment(If Any) | ---------------------- |

**EXPERIMENTDETAILS**

|  |  |
| --- | --- |
| **Experiment No:2** | |
| Title | Write a program to sort given set of numbers in ascending/descending order using Insertion sort and also search a number using linear search. |
| Objective | 1.TostudyandImplementInsertion Sort Algorithm  2.TostudyandImplementLinear Search Algorithm |
| Pre-requisite | Knowledge of   * Array Data Structure |
| Algorithm | **Insertion Sort Algorithm:**  **Input to the function is array A: The array to be sorted**  **for** i ← 1 **to** length(A)  x ← A[i]  j ← i  **while** j > 0 and A[j-1] > x  A[j] ← A[j-1]  j ← j - 1  A[j] ← x  **Linear Search Algorithm:**  **Input to the algorithm is A: Array in which key is to searched, x: the key is to searched**  LinearSearch(A, x)  Set *i* to *n*.  Repeat this loop:  If *i*< 0, then exit the loop.  If *A*[*i*] = *x*, then exit the loop.  Set *i* to *i* − 1.  Return *i*.  If returned value is –ve then it means item not found otherwise search successful at location i. |
| Source Code | #include<stdio.h>  #include<stdlib.h>  #include<conio.h>  void insertion\_sort(int a[],int);  void linear\_search(int a[],int, int);  void main()  {  int data[100],i,n,element,ch;  while(1)  {  clrscr();  printf("\*\*\*\*MENU\*\*\*\*\nEnter 1 to apply insertion sort\nEnter 2 to search an element using Linear Search\nEnter 3 for exit\nEnter your choice\n");  scanf("%d",&ch);  switch(ch)  {  case 1:  printf("Enter the number of elements in array \n");  scanf("%d",&n);  printf("Enter array elements\n");  for(i=0;i<n;i++)  {  printf("enter %d element\n",i+1);  scanf("%d",&data[i]);  }  insertion\_sort(data,n);  break;  case 2:  printf("Enter the number of elements in array \n");  scanf("%d",&n);  printf("Enter array elements\n");  for(i=0;i<n;i++)  {  printf("enter %d element\n",i+1);  scanf("%d",&data[i]);  }  printf("Enter the element you want to search for\n");  scanf("%d",&element);  linear\_search(data,n,element);  break;  case 3:  exit(0);  default:  printf("You have entered wrong choice\n");  }  getch();  }  }  void insertion\_sort(int a[],int n)  {  int i,j,temp;  for(i=1;i<n;i++)  {  j=i;  temp=a[j];  while(j>0 && temp<a[j-1])  {  a[j]=a[j-1];  j--;  }  a[j]=temp;  }  printf("After sorting array elements are:- \n");  for(i=0;i<n;i++)  printf("%d\n",a[i]);  }  void linear\_search(int a[],int n,int element)  {  int i;  for(i=0;i<n;i++)  {  if(a[i]==element)  {  printf("Element found at location %d\n",i+1);  return;  }  }  printf("Element not found\n");  return;  } |
| Sample Output |  |
| PostLab Assignment(If Any) | ---------------------- |

**EXPERIMENTDETAILS**

|  |  |
| --- | --- |
| **Experiment No:3** | |
| Title | Write a program to sort given set of numbers in ascending/descending order using Quick sort and any other sorting algorithm. Also record the time taken by these two programs and compare them. |
| Objective | 1.TostudyandImplement Quick SortAlgorithm |
| Pre-requisite | Knowledge of   * Array Data Structure * Divide and Conquer Technique |
| Algorithm | **Quick Sort Algorithm:**  **Input to the function is array A: The array to be sorted**  The following procedure implements quick sort.  **QUICKSORT(A, p, r)**  1 if p < r  2 then q ← PARTITION(A, p, r)  3 QUICKSORT(A, p, q − 1)  4 QUICKSORT(A, q + 1, r)  To sort an entire array A, the initial call is QUICKSORT (A, 1, length[A]).  **Partitioning the array**  The key to the algorithm is the PARTITION procedure, which rearranges the subarray  A [p . . . r] in place.  **PARTITION(A, p, r)**  1 x ← A[r]  2 i ← p − 1  3 for j ← p to r − 1  4 do if A[ j ] ≤ x  5 then i ←i + 1  6 exchange A[i ] ↔ A[ j ]  7 exchange A[i + 1] ↔ A[r]  8 return i + 1 |
| Source Code | #include<stdio.h>  #include<stdlib.h>  #include<conio.h>  #include<time.h>  void insertion\_sort(int a[],int);  void quick\_sort(int a[],int,int,int);  void bubble\_sort(int a[],int);  int partition(int a[],int,int);  void main()  {  int data[1000],i,n,element,ch,a[1000],b[1000],c[1000];  while(1)  {  clrscr();  printf("\*\*\*\*MENU\*\*\*\*\nEnter 1 to apply and compare quick sort,bubble sort,insertion sort\nEnter 2 for exit\nEnter your choice\n");  scanf("%d",&ch);  switch(ch)  {  case 1:  printf("Enter the number of elements in array \n");  scanf("%d",&n);  printf("Enter array elements\n");  for(i=0;i<n;i++)  {  data[i]=rand();  }  for(i=0;i<n;i++)  {  a[i]=data[i];  b[i]=data[i];  c[i]=data[i];  }  quick\_sort(a,0,n-1,n);  //printf("After quick sort array elements are:- \n");  //for(i=0;i<n;i++)  //printf("%d\n",a[i]);  bubble\_sort(b,n);  insertion\_sort(c,n);  break;  case 2:  exit(0);  default:  printf("You have entered wrong choice\n");  }  getch();  }  }  void quick\_sort(int a[],int p,int r,int n)  {  int q,i;  double time;  clock\_t start,end;  if(p==0 && r==n-1)  start=clock();  if(p<r)  {  q=partition(a,p,r);  quick\_sort(a,p,q-1,n);  quick\_sort(a,q+1,r,n);  }  if(p==0 && r==n-1)  {  end=clock();  time=((double)(end-start))/CLK\_TCK;  printf("Time taken by quick sort is %f",time);  }  }  int partition(int a[],int p,int r)  {  int i,j,temp,pivot;  i=p-1;  pivot=a[r];  for(j=p;j<r;j++)  {  if(a[j]<pivot)  {  i++;  temp=a[i];  a[i]=a[j];  a[j]=temp;  }  }  temp=a[i+1];  a[i+1]=pivot;  a[r]=temp;  return i+1;  }  void bubble\_sort(int b[],int n)  {  int i,j,temp;  double time;  clock\_t start,end;  start=clock();  for(i=0;i<n;i++)  {  for(j=0;j<n-i-1;j++)  {  if(b[j]>b[j+1])  {  temp=b[j];  b[j]=b[j+1];  b[j+1]=temp;  }  }  }  end=clock();  time=((double)(end-start))/CLK\_TCK;  printf("Time taken by bubble sort is %f",time);  //printf("After bubble sort array elements are:- \n");  //for(i=0;i<n;i++)  //printf("%d\n",b[i]);  }  void insertion\_sort(int a[],int n)  {  int i,j,temp;  double time;  clock\_t start,end;  start=clock();  for(i=1;i<n;i++)  {  j=i;  temp=a[j];  while(j>0 && temp<a[j-1])  {  a[j]=a[j-1];  j--;  }  a[j]=temp;  }  end=clock();  time=((double)(end-start))/CLK\_TCK;  printf("Time taken by insertion sort is %f",time);  //printf("After insertion sort array elements are:- \n");  //for(i=0;i<n;i++)  //printf("%d\n",a[i]);  } |
| Sample Output |  |
| Post Lab Assignment(if any) | ------------------------------------------------------- |

**EXPERIMENTDETAILS**

|  |  |
| --- | --- |
| **Experiment No:4** | |
| Title | Write a program to sort given set of numbers using Heap sort. |
| Objective | To study and Implement Heap Sort Algorithm. |
| Pre-requisite | Knowledge of   * Array Data Structure * Tree |
| Algorithm/Theory | The heap sort algorithm can be divided into two parts.  In the first step, a [heap](http://en.wikipedia.org/wiki/Heap_(data_structure)) is [built](http://en.wikipedia.org/wiki/Binary_heap#Building_a_heap) out of the data.  In the second step, a sorted array is created by repeatedly removing the largest element from the heap, and inserting it into the array. The heap is reconstructed after each removal. Once all objects have been removed from the heap, we have a sorted array. The direction of the sorted elements can be varied by choosing a min-heap or max-heap in step one.  **Input:-** A is the array to be sorted  **HEAPSORT(A)**  1 BUILD-MAX-HEAP(A)  2 for i ← length[A] down to 2  3 do exchange A[1] ↔ A[i ]  4 heap-size[A] ← heap-size[A] − 1  5 MAX-HEAPIFY(A, 1)  **MAX-HEAPIFY** are an important subroutine for manipulating max-heaps. Its inputs are an array A and an index i into the array. When MAX-HEAPIFY is called, it is assumed that the binary trees rooted at LEFT(i ) and RIGHT(i ) are max-heaps, but that A[i ] may be smaller than its children, thus violating the max-heap property. The function of MAX-HEAPIFY is to let the value at A[i ] “float down” in the maxheap  So that the subtree rooted at index i become a max-heap.  **MAX-HEAPIFY(A, i )**  1 l ← LEFT(i )  2 r ← RIGHT(i )  3 if l ≤ heap-size[A] and A[l] > A[i ]  4 then largest ←l  5 else largest ←i  6 if r ≤ heap-size[A] and A[r] > A[largest]  7 then largest ←r  8 if largest \_= i  9 then exchange A[i ] ↔ A[largest]  10 MAX-HEAPIFY(A, largest)  **BUILD-MAX-HEAP*(A)***  1 heap-size[A] ← length[A]  2 for i ← \_length[A]/2down to 1  3 do MAX-HEAPIFY(A, i ) |
| Source Code | #include<stdio.h>  #include<conio.h>  void main()  {  int heap[10],n,i,j,c,root,temp;  clrscr();  printf("\n Enter no of elements :");  scanf("%d",&n);  printf("\n Enter the nos : ");  for(i=0;i<n;i++)  scanf("%d",&heap[i]);  for(i=1;i<n;i++)  {  c=i;  do  {  root=(c-1)/2;  if(heap[root]<heap[c])  {  temp=heap[root];  heap[root]=heap[c];  heap[c]=temp;  }  c=root;  }while(c!=0);  }  printf("Heap array : ");  for(i=0;i<n; i++)  printf("%d\t", heap[i]);  for(j=n-1;j>=0;j--)  {  temp=heap[0];  heap[0]=heap[j];  heap[j]=temp;  root=0;  do  {  c=2\*root+1;  if((heap[c]<heap[c+1])&&c<j-1)  c++;  if(heap[root]<heap[c]&&c<j)  {  temp=heap[root];  heap[root]=heap[c];  heap[c]=temp;  }  root=c;  }while(c<j);  }  printf("\n The sorted array is : ");  for(i=0;i<n;i++)  printf("\t %d",heap[i]);  getch();  } |
| Sample Output |  |
| Post Lab Assignment(if any) | ------------------------------------------ |

**EXPERIMENTDETAILS**

|  |  |
| --- | --- |
| **Experiment No:5** | |
| Title | Write a program to sort given set of numbers Merge Sort. |
| Objective | 1.TostudyandImplement Merge SortAlgorithm |
| Pre-requisite | Knowledge of   * Array Data Structure * Divide and Conquer Technique |
| Algorithm | **Merge Sort Algorithm:**  **Inputtothefunctionis array A: The array to be sorted**  **MERGE-SORT(A, p, r)**  1 if p < r  2 then q ← \_(p + r)/2\_  3 MERGE-SORT(A, p, q)  4 MERGE-SORT(A, q + 1, r)  5 MERGE(A, p, q, r)  **MERGE(A, p, q, r)**  1 n1 ← q − p + 1  2 n2 ←r − q  3 create arrays L[1 . . n1 + 1] and R[1 . . n2 + 1]  4 for i ← 1 to n1  5 do L[i ] ← A[p + i − 1]  6 for j ← 1 to n2  7 do R[ j ]← A[q + j ]  8 L[n1 + 1]←infinity  9 R[n2 + 1]←infinity  10 i ← 1  11 j ← 1  12 for k ← p to r  13 do if L[i ] ≤ R[ j ]  14 then A[k] ← L[i ]  15 i ← i + 1  16 else A[k] ← R[ j ]  17 j ← j + 1 |
| Source Code | #include<stdio.h>  #include<conio.h>  void merge(int [],int,int,int);  void part(int [],int, int);  int main()  {  int a[30],i,size;  clrscr();  printf("\nMerge sorting method\n");  printf("Enter total no. of elements :");  scanf("%d",&size);  for(i=0;i<size;i++)  {  printf("Enter %d element : ",i+1);  scanf("%d",&a[i]);  }  part(a,0,size-1);  printf("\nMerge sorted elements\n");  for(i=0;i<size;i++)  printf("%d\n",a[i]);  getch();  return 0;  }  void part(int a[],int min,int max)  {  int mid;  if(min<max)  {  mid=(min+max)/2;  part(a,min,mid);  part(a,mid+1,max);  merge(a,min,mid,max);  }  }  void merge(int a[],int min,int mid,int max)  {  int temp[30];  int i,j,k,m;  j=min;  m=mid+1;  for(i=min;j<=mid&&m<=max;i++)  {  if(a[j]<=a[m])  {  temp[i]=a[j];  j++;  }  else  {  temp[i]=a[m];  m++;  }  }  if(j>mid)  {  for(k=m;k<=max;k++)  {  temp[i]=a[k];  i++;  }  }  else  {  for(k=j;k<=mid;k++)  {  temp[i]=a[k];  i++;  }  }  for(k=min; k<=max; k++)  a[k]=temp[k];  } |
| Sample Output |  |
| Post Lab Assignment(if any) | --------------------------------------------- |

**EXPERIMENTDETAILS**

|  |  |
| --- | --- |
| **Experiment No:6** | |
| Title | Write a program to sort given set of numbers Counting Sort. |
| Objective | 1.TostudyandImplement Counting SortAlgorithm |
| Pre-requisite | Knowledge of   * Array Data Structure |
| Algorithm | **Insertion Sort Algorithm:**  **Inputtothefunctionis array A: The array to be sorted**  In the code for counting sort, we assume that the input is an array A [1 . . . n], and Thus length [A] = n. We require two other arrays: the array B [1. . . n] holds the sorted output, and the array C[0 . . k] provides temporary working storage.  **COUNTING-SORT(A, B, k)**  1 for i ← 0 to k  2 do C[i ] ← 0  3 for j ← 1 to length[A]  4 do C[A[ j ]] ← C[A[ j ]] + 1  5 C[i] now contains the number of elements equal to i .  6 for i ← 1 to k  7 do C[i ] ← C[i ] + C[i − 1]  8 C[i] now contains the number of elements less than or equal to i .  9 for j ← length[A] down to 1  10 do B[C[A[ j ]]] ← A[ j ]  11 C[A[ j ]] ← C[A[ j ]] − 1 |
| Source Code | #include<stdio.h>  #include<conio.h>  void main()  {  int i,j,n,a[20],b[20],c[20],max,k;  clrscr();  printf("enter no of elements");  scanf("%d",&n);  printf("enter elements between range 0 to 19\n");  for(i=0;i<n;i++)  scanf("%d",&a[i]);  k=19;  for(i=0;i<=k;i++)  c[i]=0;  for(i=0;i<n;i++)  c[a[i]]=c[a[i]]+1;  for(i=1;i<=k;i++)  c[i]=c[i]+c[i-1];  for(j=n-1;j>=0;j--)  {  b[c[a[j]]-1]=a[j];  c[a[j]]=c[a[j]]-1;  }  printf("AFTER SORTING");  for(i=0;i<n;i++)  printf("%d\n",b[i]);  getch();  } |
| Sample Output |  |
| Post Lab Assignment(if any) | ------------------------------------------------------- |

**EXPERIMENTDETAILS**

|  |  |
| --- | --- |
| **Experiment No:7** | |
| Title | Write a program to implement Strassen's Matrix Multiplication by Divide and Conquer |
| Objective | 1. To study and Implement Matrix multiplication problem. |
| Pre-requisite | Knowledge of   * Array concept |
| Algorithm | Begin  Read two arrays A and B of size nxn  MMult(A, B, n)  1. If n = 1 Output A × B  2. Else  3. Compute A11, B11, . . ., A22, B22 % by computing m = n/2  4. X1 ← MMult(A11, B11, n/2)  5. X2 ← MMult(A12, B21, n/2)  6. X3 ← MMult(A11, B12, n/2)  7. X4 ← MMult(A12, B22, n/2)  8. X5 ← MMult(A21, B11, n/2)  9. X6 ← MMult(A22, B21, n/2)  10. X7 ← MMult(A21, B12, n/2)  11. X8 ← MMult(A22, B22, n/2)  12. C 11 ← X1 + X2  13. C 12 ← X3 + X4  14. C 21 ← X5 + X6  15. C 22 ← X7 + X8  16. Output C  17. End If  End |
| Source Code | 1. #include<stdio.h> 2. int main(){ 3. int a[2][2], b[2][2], c[2][2], i, j; 4. int m1, m2, m3, m4 , m5, m6, m7; 6. printf("Enter the 4 elements of first matrix: "); 7. for(i = 0;i < 2; i++) 8. for(j = 0;j < 2; j++) 9. scanf("%d", &a[i][j]); 11. printf("Enter the 4 elements of second matrix: "); 12. for(i = 0; i < 2; i++) 13. for(j = 0;j < 2; j++) 14. scanf("%d", &b[i][j]); 16. printf("**\n**The first matrix is**\n**"); 17. for(i = 0; i < 2; i++){ 18. printf("**\n**"); 19. for(j = 0; j < 2; j++) 20. printf("%d**\t**", a[i][j]); 21. } 23. printf("**\n**The second matrix is**\n**"); 24. for(i = 0;i < 2; i++){ 25. printf("**\n**"); 26. for(j = 0;j < 2; j++) 27. printf("%d**\t**", b[i][j]); 28. } 30. m1= (a[0][0] + a[1][1]) \* (b[0][0] + b[1][1]); 31. m2= (a[1][0] + a[1][1]) \* b[0][0]; 32. m3= a[0][0] \* (b[0][1] - b[1][1]); 33. m4= a[1][1] \* (b[1][0] - b[0][0]); 34. m5= (a[0][0] + a[0][1]) \* b[1][1]; 35. m6= (a[1][0] - a[0][0]) \* (b[0][0]+b[0][1]); 36. m7= (a[0][1] - a[1][1]) \* (b[1][0]+b[1][1]); 38. c[0][0] = m1 + m4- m5 + m7; 39. c[0][1] = m3 + m5; 40. c[1][0] = m2 + m4; 41. c[1][1] = m1 - m2 + m3 + m6; 43. printf("**\n**After multiplication using Strassen's algorithm **\n**"); 44. for(i = 0; i < 2 ; i++){ 45. printf("**\n**"); 46. for(j = 0;j < 2; j++) 47. printf("%d**\t**", c[i][j]); 48. } 50. return 0; 51. } |
| Sample Output |  |
| Post Lab Assignment(if any) | ---------------------------------------------------------- |

**EXPERIMENTDETAILS**

|  |  |
| --- | --- |
| **Experiment No:8** | |
| Title | Write a program to implement Knapsack using Greedy technique. |
| Objective | 1. TostudyandImplementKnapsack Algorithm. |
| Pre-requisite | Knowledge of   * Array Data Structure * Greedy Programming |
| Algorithm | **Knapsack Problem:**   * Two main kinds of Knapsack Problems:   1. 0-1 Knapsack:      + N items (can be the same or different)      + Have **only one** of each      + Must **leave or take**(i.e. 0-1) each item (e.g. ingots of gold)      + DP works, greedy does not   2. Fractional Knapsack:      + N items (can be the same or different)      + Can take **fractional part** of each item (eg bags of gold dust)      + Greedy works and DP algorithms work   **Fractional Knapsack: Greedy Solution**   * Algorithm:   + Assume knapsack holds weight W and items have value vi and weight wi   + Rank items by value/weight ratio: vi / wi     - Thus: vi / wi ≥ vj / wj, for all i ≤ j   + Consider items in order of decreasing ratio   + Take as much of each item as possible * Example: Knapsack Capacity W = 30 and  |  |  |  |  |  | | --- | --- | --- | --- | --- | | Item | A | B | C | D | | Value | 50 | 140 | 60 | 60 | | Size | 5 | 20 | 10 | 12 | | Ratio | 10 | 7 | 6 | 5 |  * Solution:   + All of A, all of B, and ((30-25)/10) of C (and none of D)   + Size: 5 + 20 + 10\*(5/10) = 30   + Value: 50 + 140 + 60\*(5/10) = 190 + 30 = 220 |
| Source Code | /\*Fractional Knapsack using Greedy Method\*/  #include<stdio.h>  #include<conio.h>  void knapsack(int n,float profit[],float weight[],float capacity){  float x[20],total,tp=0;  int i,j;  total=capacity;  printf("\nEntered Items are :\n ");  printf(" Value\t\t\tProfit:");  printf("\n---------------------------\n");  for(i=0;i<n;i++){  printf("%f\t\t\t%f\n",profit[i],weight[i]);  }  for(i=0;i<n;i++)  x[i]=0.0;  for(i=0;i<n;i++){  if(weight[i]>total)  break;  else  {  x[i]=1.0;  tp+=profit[i];  total-=weight[i];  }  }  if(i<n){  x[i]=total/weight[i];  }  tp+=x[i]\*profit[i];  printf("\n\nProfit Vector is : ");  for(i=0;i<n;i++)  printf("%f\t",x[i]);  printf("\n");  printf("\nTotal Profit : %f",tp);  }  int main(){  float weight[20],profit[20],ratio[20],capacity,temp;  int n,i,j;  printf("\nEnter no of items : ");  scanf("%d",&n);  printf("\nEnter Capacity : ");  scanf("%f",&capacity);  printf("\nEnter Weight nd Profit : ");  for(i=0;i<n;i++){  printf("\nEnter Weight nd Profit for item[%d] : ",i);  scanf("%f %f",&weight[i],&profit[i]);  }  for(i=0;i<n;i++)  ratio[i]=profit[i]/weight[i];  for(i=0;i<n;i++)  for(j=i+1;j<n;j++){  if(ratio[i]<ratio[j]){  temp=ratio[j];  ratio[j]=ratio[i];  ratio[i]=temp;  temp=weight[j];  weight[j]=weight[i];  weight[i]=temp;  temp=profit[j];  profit[j]=profit[i];  profit[i]=temp;  }  knapsack(n,profit,weight,capacity);  getch();  return(0);  }  } |
| Sample Output |  |
| Post Lab Assignment(if any) | ----------------------------------------------- |

**EXPERIMENTDETAILS**

|  |  |
| --- | --- |
| **Experiment No:9** | |
| Title | Write a program to implement Knapsack using Dynamic programming. |
| Objective | 1. TostudyandImplement Knapsack Algorithm. |
| Pre-requisite | Knowledge of   * Array Data Structure * Dynamic Programming |
| Algorithm | **Knapsack Problem:**   * Two main kinds of Knapsack Problems:   1. 0-1 Knapsack:      + N items (can be the same or different)      + Have **only one** of each      + Must **leave or take**(i.e. 0-1) each item (e.g. ingots of gold)      + DP works, greedy does not   2. Fractional Knapsack:      + N items (can be the same or different)      + Can take **fractional part** of each item (eg bags of gold dust)      + Greedy works and DP algorithms work   **0-1 Knapsack: Dynamic Solution**  Here is a dynamic programming algorithm to solve the 0-1 Knapsack problem:  **Input:** S, a set of n items as described earlier, W the total weight of the knapsack. (Assume that the weights and values are stored in separate arrays named w and v, respectively.)  **Output:** The maximal value of items in a valid knapsack.  int w, k;  for (w=0; w <= W; w++)  B[w] = 0  for (k=0; k<n; k++) {  for (w = W; w>= w[k]; w--) {  if (B[w – w[k]] + v[k]> B[w])  B[w] = B[w – w[k]] + v[k]  }  } |
| Source Code | #include<stdio.h>  #include<conio.h>  Int sum=0;  int max(int a,int b)  {  if(a>b)  return a;  else  return b;  }  void knapsack(int m,int n,int w[],int p[])  {  int v[100][200],x[10],i,j;  for(i=0;i<=m;i++)  v[0][i]=0;  for(i=1;i<=n;i++)  {  for(j=0;j<=m;j++)  {  if(j>=w[i])  v[i][j]=max(v[i-1][j],v[i-1][j-w[i]]+p[i]);  else  v[i][j]=v[i-1][j];  }  }  for(i=1;i<=n;i++)  x[i]=0;  i=n;  j=m;  while(i>0 && j>0)  {  if(v[i][j]!=v[i-1][j])  {  x[i]=1;  j=j-w[i];  }  i--;  }  printf("\nTHE OPTIMAL SET OF WEIGHTS IS:\n");  for(i=1;i<=n;i++)  {  if(x[i]==1)  {  printf("X%d=1\t",i);  sum=sum+p[i];  }  Else  printf("X%d=0\t",i);  }  printf("\nTotal profit = %d",sum);  }  void main()  {  int w[10],p[10],i,m,n;  printf("\t0/1 KNAPSACK PROBLEM\n\n");  printf("ENTER THE NUMBER OF ITEMS: ");  scanf("%d",&n);  printf("ENTER THE WEIGHTS OF THE ITEMS:\n");  for(i=1;i<=n;i++)  scanf("%d",&w[i]);  printf("ENTER THE PROFITS OF THE ITEMS:\n");  for(i=1;i<=n;i++)  scanf("%d",&p[i]);  printf("ENTER THE CAPACITY OF KNAPSACK: ");  scanf("%d",&m);  knapsack(m,n,w,p);  getch();  } |
| Sample Output |  |
| Post Lab Assignment(if any) | ------------------------------------------------------ |

**EXPERIMENTDETAILS**

|  |  |
| --- | --- |
| **Experiment No:10** | |
| Title | Write a program to implement Dijkstra’s Algorithm. |
| Objective | 1. TostudyandImplement Dijkstra’s Algorithm. |
| Pre-requisite | Knowledge of   * Tree Data Structure * Graph |
| Algorithm | **Dijkstra’s Algorithm:**  Dijkstra’s algorithm solves the single-source shortest-paths problem on a weighted, directed graph G = (V, E) for the case in which all edge weights are nonnegative. Dijkstra’s algorithm maintains a set S of vertices whose final shortest-path weights from the source s have already been determined. The algorithm repeatedly selects the vertex u € V − S with the minimum shortest-path estimate, adds u to S, and relaxes all edges leaving u. In the following implementation, a min-priority queue Q of vertices is used, keyed by their d values.  **Inputs:**  **G- The Graph**  **w- weight matrix**  **s- source vertex**  **DIJKSTRA(G,w, s)**  1 INITIALIZE-SINGLE-SOURCE(G, s)  2 S ← NULL  3 Q ← V[G]  4 **while** Q != NULL  5 **do** u ← EXTRACT-MIN(Q)  6 S ← S UNION {u}  7 **for** each vertex v € Adj[u]  8 **do** RELAX(u, v,w)  **INITIALIZE-SINGLE-SOURCE(G, s)**  1 **for** each vertex v €V[G]  2 **do** d[v]←∞  3 π[v]← NIL  4 d[s] ← 0  **RELAX(u, v,w)**  1 **if** d[v] > d[u] + w(u, v)  2 **then** d[v] ← d[u] + w(u, v)  3 π[v]← u |
| Source Code | #include "stdio.h"  #include "conio.h"  #define infinity 999  void dij(int n,int v,int cost[10][10],int dist[])  {  int i,u,count,w,flag[10],min;  for(i=1;i<=n;i++)  flag[i]=0,dist[i]=cost[v][i];  count=2;  while(count<=n)  {  min=99;  for(w=1;w<=n;w++)  if(dist[w]<min && !flag[w])  min=dist[w],u=w;  flag[u]=1;  count++;  for(w=1;w<=n;w++)  if((dist[u]+cost[u][w]<dist[w]) && !flag[w])  dist[w]=dist[u]+cost[u][w];  }  }  void main()  {  int n,v,i,j,cost[10][10],dist[10];  clrscr();  printf("\n Enter the number of nodes:");  scanf("%d",&n);  printf("\n Enter the cost matrix:\n");  for(i=1;i<=n;i++)  for(j=1;j<=n;j++)  {  scanf("%d",&cost[i][j]);  if(cost[i][j]==0)  cost[i][j]=infinity;  }  printf("\n Enter the source matrix:");  scanf("%d",&v);  dij(n,v,cost,dist);  printf("\n Shortest path:\n");  for(i=1;i<=n;i++)  if(i!=v)  printf("%d->%d,cost=%d\n",v,i,dist[i]);  getch();  } |
| Sample Output | 999 is representing infinity. It means no paths exist.  0 in the input cost matrix denotes no direct paths exist. |
| Post Lab Assignment(if any) | ------------------------------------------------------------- |

**EXPERIMENTDETAILS**

|  |  |
| --- | --- |
| **Experiment No:11** | |
| Title | Write a program to implement Bellman-Ford Algorithm. |
| Objective | 1. TostudyandImplement Dijkstra’s Algorithm. |
| Pre-requisite | Knowledge of   * Tree Data Structure * Graph |
| Algorithm | **Bellman-For**d **Algorithm:**  The Bellman-Ford algorithm solves the single-source shortest-paths problem in the general case in which edge weights may be negative. Given a weighted, directed graph G = (V, E) with source s and weight function w : E → R, the Bellman-Ford algorithm returns a Boolean value indicating whether or not there is a negative-weight cycle that is reachable from the source. If there is such a cycle, the algorithm indicates that no solution exists. If there is no such cycle, the algorithm produces the shortest paths and their weights. The algorithm uses relaxation, progressively decreasing an estimate d[v] on the weight of a shortest path from the source s to each vertex v € V until it achieves the actual shortest-path weight δ(s, v). The algorithm returns TRUE if and only if the graph contains no negative-weight cycles that are reachable from the source.  **BELLMAN-FORD(G,w, s)**  1 **INITIALIZE-SINGLE-SOURCE(G, s)**  2 for i ← 1 to |V[G]| − 1  3 do for each edge (u, v) € E[G]  4 do **RELAX(u, v,w)**  5 for each edge (u, v) € E[G]  6 do if d[v] > d[u] + w(u, v)  7 then return FALSE  8 return TRUE  **INITIALIZE-SINGLE-SOURCE(G, s)**  1 **for** each vertex v € V[G]  2 **do** d[v]←∞  3 π[v]← NIL  4 d[s] ← 0  **RELAX(u, v, w)**  1 **if** d[v] > d[u] + w(u, v)  2 **then** d[v] ← d[u] + w(u, v)  3 π[v]← u |
| Source Code | #include <stdio.h>  /\* Relax edge (u,v) with weight w. \*/  void relax(int u, int v, double w, double d[], int pi[]) {  if (d[v] > d[u] + w) {  d[v] = d[u] + w;  pi[v] = u;  }  }  /\* Initialize a single-source shortest-paths computation. \*/  void initialize\_single\_source(double d[], int pi[], int s, int n) {  int i;  for (i = 1; i <= n; ++i) {  d[i] = 1000000000.0;  pi[i] = 0;  }  d[s] = 0.0;  }  /\* Run the Bellman-Ford algorithm from vertex s. Fills in arrays d  and pi. \*/  int bellman\_ford(int first[], int node[], int next[], double w[], double d[],  int pi[], int s, int n) {  int u, v, i, j;  initialize\_single\_source(d, pi, s, n);  for (i = 1; i <= n-1; ++i) {  for (u = 1; u <= n; ++u) {  j = first[u];  while (j > 0) {  v = node[j];  relax(u, v, w[j], d, pi);  j = next[j];  }  }  }  for (u = 1; u <= n; ++u) {  j = first[u];  while (j > 0) {  v = node[j];  if (d[v] > d[u] + w[j])  return 0;  j = next[j];  }  }  return 1;  }  int main(void) {  int first[6], node[11], next[11], pi[6];  double w[11], d[6];  int s;  int i;  int ok;  first[1] = 1;  first[2] = 3;  first[3] = 6;  first[4] = 7;  first[5] = 9;  node[1] = 2;  node[2] = 4;  node[3] = 3;  node[4] = 4;  node[5] = 5;  node[6] = 2;  node[7] = 3;  node[8] = 5;  node[9] = 1;  node[10] = 3;  w[1] = 6.0;  w[2] = 7.0;  w[3] = 5.0;  w[4] = 8.0;  w[5] = -4.0;  w[6] = -2.0;  w[7] = -3.0;  w[8] = 9.0;  w[9] = 2.0;  w[10] = 7.0;  next[1] = 2;  next[2] = 0;  next[3] = 4;  next[4] = 5;  next[5] = 0;  next[6] = 0;  next[7] = 8;  next[8] = 0;  next[9] = 10;  next[10] = 0;  printf("Enter source node: ");  scanf("%d", &s);  ok = bellman\_ford(first, node, next, w, d, pi, s, 5);  printf("bellman\_ford returns ");  printf("%d\n\n", ok);  for (i = 1; i <= 5; ++i) {  printf("%d: %f %d\n", i, d[i], pi[i]);  }  return 0;  } |
| Sample Output |  |
| Post Lab Assignment(if any) | --------------------------------------- |

**EXPERIMENTDETAILS**

|  |  |
| --- | --- |
| **Experiment No:12** | |
| Title | Write a program to implement n-Queen Problem using backtracking. |
| Objective | 1.TostudyandImplementn-Queen Problem |
| Pre-requisite | Knowledge of   * Array Data Structure * Backtracking |
| Algorithm/Theory | **n-Queen Problem :**  The n-queens problem consists in placing n non-attacking queens on an n-by-n chess board. A queen can attack another queen vertically, horizontally, or diagonally. E.g. placing a queen on a central square of the board blocks the row and column where it is placed, as well as the two diagonals (rising and falling) at whose intersection the queen was placed.  The algorithm to solve this problem uses backtracking, but we will unroll the recursion. The basic idea is to place queens column by column, starting at the left. New queens must not be attacked by the ones to the left that have already been placed on the board. We place another queen in the next column if a consistent position is found. All rows in the current column are checked. We have found a solution if we placed a queen in the rightmost column.  Following is a solution for 4 Queen problem. |
| Source Code | #include<stdio.h>  #include<conio.h>  #include<math.h>  char a[10][10];  int n;  void printmatrix()  {  int i,j;  printf("n");  for(i=0;i < n;i++)  {  for(j=0;j < n;j++)  printf("%c\t",a[i][j]);  printf("n\n");  }  }  int getmarkedcol(int row)  {  int i,j;  for(i=0;i < n;i++)  if(a[row][i]=='Q')  {  return(i);  break;  }  }  int feasible(int row, int col)  {  int i,tcol;  for(i=0;i < n;i++)  {  tcol=getmarkedcol(i);  if(col==tcol || abs(row-i)==abs(col-tcol))  return 0;  }  return 1;  }  void nqueen(int row)  {  int i,j;  if(row < n)  {  for(i=0;i < n;i++)  {  if(feasible(row,i))  {  a[row][i]='Q';  nqueen(row+1);  a[row][i]='.';  }  }  }  else  {  printf("\nThe solution is:- ");  printmatrix();  }  }  voidmain()  {  int i,j;  clrscr();  printf("n Enter the no. of queens:- ");  scanf("%d",&n);  for(i=0;i < n;i++)  for(j=0;j < n;j++)  a[i][j]='.';  nqueen(0);  getch();  } |
| Sample Output | Enter the no. of queens:- 4  The solution is:-  . Q . .  . . . Q  Q . . .  . . Q .  The solution is:-  . . Q .  Q . . .  . . . Q  . Q . . |
| Post Lab Assignment(if any) | ------------------------------------------------------ |

**EXPERIMENTDETAILS**

|  |  |
| --- | --- |
| **Experiment No:13** | |
| Title | Write a program to create B Tree |
| Objective | To study and Implement B tree |
| Pre-requisite | Knowledge of   * Tree |
| Algorithm/ Theory | Begin  insert newEntry in the appropriate leaf;  currentNode = leaf;  while (currentNode overflow)  split the currentNode into two nodes  on the same level, and promote median key up  to the parent of currentNode;  currentNode := parent of currentNode;  End |
| Source Code | #include<stdlib.h>  #include<stdio.h>  #define M 5   struct node{      int n; /\* n < M No. of keys in node will always less than order of B tree \*/      int keys[M-1]; /\*array of keys\*/      struct node \*p[M];   /\* (n+1 pointers will be in use) \*/  }\*root=NULL;   void insert(int key);  void display(struct node \*root,int);   int main()  {      int key;      int choice;      printf("Creation of B tree for node %d\n",M);      while(1)      {          printf("1.Insert\n");          printf("2.Display\n");          printf("3.Quit\n");          printf("Enter your choice : ");          scanf("%d",&choice);           switch(choice)          {              case 1:                  printf("Enter the key : ");                  scanf("%d",&key);                  insert(key);                  break;               case 2:                  printf("Btree is :\n");                  display(root,0);                  break;              case 3:                  exit(1);              default:                  printf("Wrong choice\n");                  break;          }/\*End of switch\*/      }/\*End of while\*/      return 0;  }/\*End of main()\*/   void insert(int key)  {      struct node \*newnode;      int upKey;      enum KeyStatus value;      value = ins(root, key, &upKey, &newnode);      if (value == Duplicate)          printf("Key already available\n");      if (value == InsertIt)      {          struct node \*uproot = root;          root=malloc(sizeof(struct node));          root->n = 1;          root->keys[0] = upKey;          root->p[0] = uproot;          root->p[1] = newnode;      }/\*End of if \*/  }/\*End of insert()\*/    void display(struct node \*ptr, int blanks)  {      if (ptr)      {          int i;          for(i=1;i<=blanks;i++)              printf(" ");          for (i=0; i < ptr->n; i++)              printf("%d ",ptr->keys[i]);          printf("\n");          for (i=0; i <= ptr->n; i++)              display(ptr->p[i], blanks+10);      }/\*End of if\*/  }/\*End of display()\*/ |
| Sample Output |  |
| Post Lab Assignment(if any) | --------------------------------------------- |

**EXPERIMENTDETAILS**

|  |  |
| --- | --- |
| **Experiment No:14** | |
| Title | Write a program to insert a node in red-black tree. |
| Objective | To study and Implement Red Black Tree algorithm. |
| Pre-requisite | Knowledge of   * Tree |
| Algorithm/Theory | Insertion begins by adding the node as any [binary search tree insertion](https://en.wikipedia.org/wiki/Binary_search_tree#Insertion) does and by coloring it red. Whereas in the binary search tree, we always add a leaf, in the red–black tree, leaves contain no information, so instead we add a red interior node, with two black leaves, in place of an existing black leaf.  What happens next depends on the color of other nearby nodes. The term *uncle node* will be used to refer to the sibling of a node's parent, as in human family trees. Note that:   1. The label **N** will be used to denote the current node (colored red). In the diagrams **N** carries a blue contour. At the beginning, this is the new node being inserted, but the entire procedure may also be applied recursively to other nodes (see case 3). **P** will denote **N**'s parent node, **G** will denote **N**'s grandparent, and **U** will denote **N**'s uncle. In between some cases, the roles and labels of the nodes are exchanged, but in each case, every label continues to represent the same node it represented at the beginning of the case. 2. If a node in the right (target) half of a diagram carries a blue contour it will become the current node in the next iteration and there the other nodes will be newly assigned relative to it. Any color shown in the diagram is either assumed in its case or implied by those assumptions. 3. A numbered triangle represents a subtree of unspecified depth. A black circle atop a triangle means that black-height of subtree is greater by one compared to subtree without this circle.   There are several cases of red–black tree insertion to handle:   * **N** is the root node, i.e., first node of red–black tree * **N**'s parent (**P**) is black * **N**'s parent (**P**) and uncle (**U**) are red * **N** is added to right of left child of grandparent, or **N** is added to left of right child of grandparent (**P** is red and **U** is black) * **N** is added to left of left child of grandparent, or **N** is added to right of right child of grandparent (**P** is red and **U** is black)   Each case will be demonstrated with example [C](https://en.wikipedia.org/wiki/C_(programming_language)) code. The uncle and grandparent nodes can be found by these functions:  **Case 1:** The current node **N** is at the root of the tree. In this case, it is repainted black to satisfy property 2 (the root is black). Since this adds one black node to every path at once, property 5 (all paths from any given node to its leaf nodes contain the same number of black nodes) is not violated.  **Case 2:** The current node's parent **P** is black, so property 4 (both children of every red node are black) is not invalidated. In this case, the tree is still valid. Property 5 (all paths from any given node to its leaf nodes contain the same number of black nodes) is not threatened because the current node **N** has two black leaf children, but because **N** is red, the paths through each of its children have the same number of black nodes as the path through the leaf it replaced, which was black, and so this property remains satisfied.  **Case 3:** If both the parent **P** and the uncle **U** are red, then both of them can be repainted black and the grandparent**G** becomes red (to maintain property 5 (all paths from any given node to its leaf nodes contain the same number of black nodes)). Now, the current red node **N** has a black parent. Since any path through the parent or uncle must pass through the grandparent, the number of black nodes on these paths has not changed. However, the grandparent **G**may now violate properties 2 (The root is black) or 4 (Both children of every red node are black) (property 4 possibly being violated since **G** may have a red parent). To fix this, the entire procedure is recursively performed on **G** from case 1. Note that this is a tail-recursive call, so it could be rewritten as a loop; since this is the only loop, and any rotations occur after this loop, this proves that a constant number of rotations occur.  **Case 4:** The parent **P** is red but the uncle **U** is black; also, the current node **N** is the right child of **P**, and **P** in turn is the left child of its parent **G**. In this case, a [left rotation](https://en.wikipedia.org/wiki/Tree_rotation) on**P** that switches the roles of the current node **N** and its parent **P** can be performed; then, the former parent node**P** is dealt with using case 5 (relabeling **N** and **P**) because property 4 (both children of every red node are black) is still violated. The rotation causes some paths (those in the sub-tree labelled "1") to pass through the node **N** where they did not before. It also causes some paths (those in the sub-tree labelled "3") not to pass through the node **P** where they did before. However, both of these nodes are red, so property 5 (all paths from any given node to its leaf nodes contain the same number of black nodes) is not violated by the rotation. After this case has been completed, property 4 (both children of every red node are black) is still violated, but now we can resolve this by continuing to case 5.  **Case 5:** The parent **P** is red but the uncle **U** is black, the current node **N** is the left child of **P**, and **P** is the left child of its parent **G**. In this case, a [right rotation](https://en.wikipedia.org/wiki/Tree_rotation) on **G** is performed; the result is a tree where the former parent **P**is now the parent of both the current node **N** and the former grandparent **G**. **G** is known to be black, since its former child **P** could not have been red otherwise (without violating property 4). Then, the colors of **P** and **G** are switched, and the resulting tree satisfies property 4 (both children of every red node are black). Property 5 (all paths from any given node to its leaf nodes contain the same number of black nodes) also remains satisfied, since all paths that went through any of these three nodes went through **G** before, and now they all go through **P**. In each case, this is the only black node of the three. |
| Source Code | **#include<stdio.h>**  **#include<conio.h>**  **struct** node \*grandparent(**struct** node \*n)  {  **if** ((n != NULL) && (n->parent != NULL))  **return** n->parent->parent;  **else**  **return** NULL;  }  **struct** node \*uncle(**struct** node \*n)  {  **struct** node \*g = grandparent(n);  **if** (g == NULL)  **return** NULL; *// No grandparent means no uncle*  **if** (n->parent == g->left)  **return** g->right;  **else**  **return** g->left;  }  void insert\_case1(**struct** node \*n)  {  **if** (n->parent == NULL)  n->color = BLACK;  **else**  insert\_case2(n);  }  void insert\_case2(**struct** node \*n)  {  **if** (n->parent->color == BLACK)  **return**; */\* Tree is still valid \*/*  **else**  insert\_case3(n);  }  void insert\_case3(**struct** node \*n)  {  **struct** node \*u = uncle(n), \*g;  **if** ((u != NULL) && (u->color == RED)) {  n->parent->color = BLACK;  u->color = BLACK;  g = grandparent(n);  g->color = RED;  insert\_case1(g);  } **else** {  insert\_case4(n);  }  }  void insert\_case4(**struct** node \*n)  {  **struct** node \*g = grandparent(n);  **if** ((n == n->parent->right) && (n->parent == g->left)) {  rotate\_left(n->parent);  */\**  *\* rotate\_left can be the below because of already having \*g = grandparent(n)*  *\**  *\* struct node \*saved\_p=g->left, \*saved\_left\_n=n->left;*  *\* g->left=n;*  *\* n->left=saved\_p;*  *\* saved\_p->right=saved\_left\_n;*  *\**  *\* and modify the parent's nodes properly*  *\*/*  n = n->left;  } **else** **if** ((n == n->parent->left) && (n->parent == g->right)) {  rotate\_right(n->parent);  */\**  *\* rotate\_right can be the below to take advantage of already having \*g = grandparent(n)*  *\**  *\* struct node \*saved\_p=g->right, \*saved\_right\_n=n->right;*  *\* g->right=n;*  *\* n->right=saved\_p;*  *\* saved\_p->left=saved\_right\_n;*  *\**  *\*/*  n = n->right;  }  insert\_case5(n);  }  void insert\_case5(**struct** node \*n)  {  **struct** node \*g = grandparent(n);  n->parent->color = BLACK;  g->color = RED;  **if** (n == n->parent->left)  rotate\_right(g);  **else**  rotate\_left(g);  } |
| Sample Output |  |
| Post Lab Assignment(if any) | ---------------------------------------------------- |

**EXPERIMENTDETAILS**

|  |  |
| --- | --- |
| **Experiment No:15** | |
| Title | Write a program to implement String Matching using Rabin-Karp algorithm. |
| Objective | 1.TostudyandImplementString Matching using Rabin-Karp algorithm |
| Pre-requisite | Knowledge of   * String Data Structure |
| Algorithm/Theory | The Rabin–Karp algorithm seeks to speed up the testing of equality of the pattern to the substrings in the text by using a [hash function](http://en.wikipedia.org/wiki/Hash_function). A hash function is a function which converts every string into a numeric value, called its hash value; for example, we might have hash ("hello") =5. Rabin–Karp exploits the fact that if two strings are equal, their hash values are also equal. Thus, it would seem all we have to do is compute the hash value of the substring we're searching for, and then look for a substring with the same hash value.  However, there are two problems with this. First, because there are so many different strings, to keep the hash values small we have to assign some strings the same number. This means that if the hash values match, the strings might not match; we have to verify that they do, which can take a long time for long substrings. Luckily, a good hash function promises us that on most reasonable inputs, this won't happen too often, which keeps the average search time good.  **The algorithm is as shown:**  function RabinKarp(string s[1..n], string sub[1..m])  hsub := hash(sub[1..m]); hs := hash(s[1..m])  for i from 1 to n-m+1  if hs = hsub  if s[i..i+m-1] = sub  return i  hs := hash(s[i+1..i+m])  return not found |
| Source Code | #include<stdio.h>  #include<string.h>  // d is the number of characters in input alphabet  #define d 256  /\* pat -> pattern  txt -> text  q -> A prime number  \*/  void search(char \*pat, char \*txt, int q)  {  int M = strlen(pat);  int N = strlen(txt);  int i, j;  int p = 0; // hash value for pattern  int t = 0; // hash value for txt  int h = 1;  // The value of h would be "pow(d, M-1)%q"  for (i = 0; i < M-1; i++)  h = (h\*d)%q;  // Calculate the hash value of pattern and first window of text  for (i = 0; i < M; i++)  {  p = (d\*p + pat[i])%q;  t = (d\*t + txt[i])%q;  }  // Slide the pattern over text one by one  for (i = 0; i <= N - M; i++)  {  // Chaeck the hash values of current window of text and pattern  // If the hash values match then only check for characters on by one  if ( p == t )  {  /\* Check for characters one by one \*/  for (j = 0; j < M; j++)  {  if (txt[i+j] != pat[j])  break;  }  if (j == M) // if p == t and pat[0...M-1] = txt[i, i+1, ...i+M-1]  {  printf("Pattern found at index %d \n", i);  }  }  // Calulate hash value for next window of text: Remove leading digit,  // add trailing digit  if ( i < N-M )  {  t = (d\*(t - txt[i]\*h) + txt[i+M])%q;  // We might get negative value of t, converting it to positive  if(t < 0)  t = (t + q);  }  }  }  /\* Driver program to test above function \*/  int main()  {  char \*txt = "GEEKS FOR GEEKS";  char \*pat = "GEEK";  int q = 101; // A prime number  search(pat, txt, q);  getchar();  return 0;  } |
| Sample Output |  |
| Post Lab Assignment(if any) | --------------------------------------------------------------- |