Assignment No -1

Title: Set operations

Problem Statement:

Perform following set operations using arrays –

a) Union
b) Intersection
c) Difference
d) Symmetric Difference

Objective:

This assignment will help the students to realize how the different operation on set operations like union, intersection, difference can be performed.

S/W Packages and H/W apparatus used:

Linux OS: Ubantu/Fedora , Eclipse/codeblocks,
PC with the configuration as Pentium IV 1.7 GHz. 128M.B RAM, 40 G.B HDD,
15''Color Monitor, Keyboard, Mouse

References: *1. 'C programming' by Balguruswami.*
2. ' C' Kanetkar.
3. 'Code complete' Steve Conne

Theory:

Arrays:

An *array* is a data structure that is a collection of variables of one type that are accessed through a common name. Elements are referenced by the array name and an *ordinal* index. Each element

is a *value* [*what*], *index* [*where*] pair. The array indexing begins at zero. The array forms a contiguous list in memory. The name of the array holds the address of the first array element. We specify the array size at compile time, often with a named constant.

e.g. const int size =10;

double setA[SIZE]; // holds up to SIZE setA

int i; // index into array

Array indexes are always *ordinals*, such as integers, characters, enumerated types, etc. The values stored in the array may be any simple type, such as a character, a numeric value, an address (later), etc.

setA: 0 1 2 3

7	**3**	**2**	**0**

setA[0]=7 , setA[1]=3, setA[2]=2 , setA[3]=0

Passing an array to a Function:

At some moment we may need to pass an array to a function as a parameter. In C++ it is not possible to pass a complete block of memory by value as a parameter to a function, but we are allowed to pass its address. In practice this has almost the same effect and it is a much faster and more efficient operation.

In order to accept arrays as parameters the only thing that we have to do when declaring the function is to specify in its parameters the element type of the array, an identifier and a pair of void brackets [].

e.g. The following function accepts a parameter of type "array of int" called arg.

void procedure (int arg[])

In order to pass to this function an array declared as:

int myarray[40]

It would be enough to write a call like this:

procedure (myarray);

Here you have a complete example for arrays as parameters

```
#include <iostream>
using namespace std;

void printarray (int arg[], int length)
{
  int n;
 for( n=0; n<length; n++)
            printf("%d", arg[n]);
}

int main ()
{
 int firstarray[] = {5, 10, 15};
 int secondarray[] = {2, 4, 6, 8, 10};
 printarray (firstarray,3);
 printarray (secondarray,5);
 return 0;
}
// output of this program will be:        5 10 15
                                          2 4 6 8 10
```

As you can see, the first parameter (int arg[]) accepts any array whose elements are of type int, whatever its length. For that reason we have included a second parameter that tells the function

the length of each array that we pass to it as its first parameter. This allows the for loop that prints out the array to know the range to iterate in the passed array without going out of range.

Set Operations:

A **set** is a collection of distinct objects, considered as an object in its own right. Sets are one of the most fundamental concepts in mathematics.

In set theory, the **union** (denoted as ∪) of a collection of sets is the set of all distinct elements in the collection. The union of a collection of sets $S_1, S_2, S_3, \ldots, S_n$gives a set $S_1 \cup S_2 \cup S_3 \cup \ldots \cup S_n$. The union of two sets A and B is the collection of points which are in A or in B (or in both):

$$A \cup B = \{x : x \in A \text{ or } x \in B\}$$

In mathematics, the **intersection** (denoted as ∩) of two sets *A* and *B* is the set that contains all elements of *A* that also belong to *B* (or equivalently, all elements of *B* that also belong to *A*), but no other elements. The intersection of *A* and *B* is written "*A* ∩ *B*". Formally:

In mathematics, the **symmetric difference** of two sets is the set of elements which are in either of the sets and not in their intersection. The symmetric difference of the sets *A* and *B* is commonly denoted by

$$A \Delta B$$

or

$$A \ominus B.$$

For example, the symmetric difference of the sets {1,2,3} and {3,4} is {1,2,4}. The symmetric difference of the set of all students and the set of all females consists of all male students together with all female non-students.

The symmetric difference is equivalent to the union of both relative complements, that is:

$$A \Delta B = (A \setminus B) \cup (B \setminus A),$$

and it can also be expressed as the union of the two sets, minus their intersection:

$$A \Delta B = (A \cup B) \setminus (A \cap B),$$

or with the XOR operation:

$$A \Delta B = \{x : (x \in A) \oplus (x \in B)\}.$$

The symmetric difference is commutative and associative:

$$A \Delta B = B \Delta A,$$
$$(A \Delta B) \Delta C = A \Delta (B \Delta C).$$

Algorithm:

[i] Steps for Union Operation

Precondition: Two 1-D arrays representing 2 sets

Postcondition: Resultant 1-D array after union operation is performed.

Return: Size of resultant array.

Union of Set A & Set B is stored in Set C

1. Copy Set A as it is in Set C.

2. K=limit1, limit3=limit1

3. For J=0 to limit2-1

For I=0 to limit1-1

If B[J]=A[I]

Break

If I=limit1

C[K]=B[J]

K=K+1

Limit3=limit3+1

4.Display C

[ii] Steps for Intersection Operation

Precondition: Two 1-D arrays representing 2 sets

Postcondition: Resultant 1-D array after intersection operation is performed.

Intersection of Set A & Set B is stored in Set C

1. K=0, limit3=0

2. For J=0 to limit2-1

For I=0 to limit1-1

If B[J]=A[I]

C[K]=B[J]

K=K+1

limit3=limit3+1

Break

3.Display C

[iii] Steps for Difference Operation

Precondition: Two 1-D arrays representing 2 sets

Postcondition: Resultant 1-D array after difference operation is done.

Difference of Set A & Set B is stored in Set C

1. K=0, limit3=0

2. For I=0 to limit1-1

For J=0 to limit2-1

If A[I]=B[J]

Break

If J=limit2

C[K]=A[I]

K=K+1

Limit3=limit3+1

3.Display C

[iv] Steps for Symmetric Difference operation

Precondition: Two 1-D arrays representing 2 sets

Postcondition: Resultant 1-D array after symmetric difference operation is done.

Symmetric Difference of Set A & Set B is stored in Set E

1. Find the difference of Set A & Set B in set C

2. Find the difference of Set B & Set A in set D

3. Find Union of Set C & Set D in Set E

4.Display E.

Input:

Input is the data / number in the form of set.

e.g A[3]={2,3,5} B[3]={3,12,13}

Output:

Output is the data/number in the form of set.

e.g. *A ∪ B= {2,3,5,12,13}*

A∩ B= {3}

Test cases:

Check for array out of bound condition.

Check for set member duplication condition.

Check for set elements must belong to universal set condition.

Check for NULL set condition

Application:

Helpful in solving different operations related to set.

FAQs:

- What is Array? Explain types of Array with example.
- Explain memory organization of Array.
- Explain passing of array to function.
- What is Union and its properties?
- What is Intersection and its properties?
- What is Difference, Symmetric Difference and its properties?
- Explain arrays in C.

Instructions for writing journal:

- Title
- Problem Definition
- Concept of arrays
- Pseudo C code for Union, Intersection, Difference, Symmetric Difference
- Analysis of above for time and space complexity
- Program code
- Output for different I/P comparing time complexity
- Conclusion

Assignment No -2

Title: Matrix Operations

Problem Statement: Represent matrix using two dimensional arrays and perform following operations with and without pointer.

a) Addition
b) Multiplication
c) Transpose
d) Saddle point

Objective:

- This assignment will help the students to realize how the different operation on matrices like addition, multiplication, transpose, saddle point works.
- To understand the concept of 2D array,
- To understand the concept of double pointer or pointer of arrays or array of pointers
- Memory Dynamic allocation and deallocation.
- Analyze the above algorithms

S/W Packages and H/W apparatus used:

Linux OS: Ubantu/Fedora , Eclipse/codeblocks,

PC with the configuration as Pentium IV 1.7 GHz. 128M.B RAM, 40 G.B HDD,

15''Color Monitor, Keyboard, Mouse

References: *1. 'C programming' by Balguruswami.*

2. ' C' Kanetkar.

3. 'Code complete' Steve Connel.

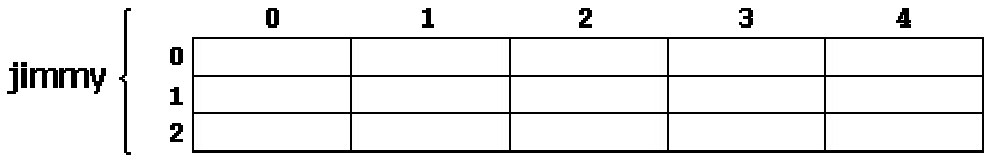
Theory:

In mathematics, a **matrix** (plural **matrices**, or less commonly **matrixes**) is a rectangular array of numbers, such as

$$\begin{bmatrix} 1 & 9 & 13 \\ 20 & 55 & 6 \end{bmatrix}.$$

Multi dimensional arrays in C

Implementation of matrix is done by two-dimensional array. Multidimensional arrays can be described as "arrays of arrays". For example, a 2-dimensional array can be imagined as a 2-dimensional table made of elements, all of them of a same uniform data type.



Arrays can be declared with more than one dimension in C.

e.g.: int jimmy [3][5];

int array2[3][6] = {{4,5,6,7,8,9},

{1,5,6,8,2,4},

```
                    {0,4,4,3,1,1}};
```

Such arrays are accessed like so:

```
array[1][4]= -2;
if (array[2][1] > 0)
{
        printf ("Element [2][1] is %d", array[2][1]);
}
```

Remember that, like ordinary arrays, multi-dimensional arrays are numbered from 0. Therefore, the array above has elements from array[0][0] to array[2][5].

Row major and column major arrays:

Suppose we have a two dimensional array like int A[3][3] We can think of the array A like this:

```
A11   A12   A13
A21   A22   A23
A31   A32   A33
```

The array A must be stored in memory. This causes a problem. There is no such thing as two dimensional memories. Main memory is just like a big 1D array with indices from 0x0 to 0Xffffff. So somehow we have to place our 2D array into 1D memory. Since we want all the locations in an array to be contiguous we have two logical choices for how to line them up. We can store the elements in memory like this:

```
|       A11 A21 A31 A12 A22 A32 A13 A23 A33       |
|                                                 |
0x0          --> Higher Addresses            0xffffff
```

This is **FORTRAN's column major order**, the first array index varies most rapidly.

The other option is to line them up like this:

A11 A12 A13 A21 A22 A23 A31 A32 A33
0x0 --> Higher Addresses 0xffffff

This is **C's row major order**, the last array index varies most rapidly.

The difference lies in how they are stored in memory not in how you interpret the memory.

Passing 2-D array to Functions

In a function declaration it is also possible to include multidimensional arrays. The format for a tridimensional array parameter is:

base_type[ ][depth][depth]

e.g. a function with a multidimensional array as argument could be:

Notice that the first brackets [] are left blank while the following ones are not. This is so because the compiler must be able to determine within the function which is the depth of each additional dimension.

When we pass multi-dimensional arrays to functions or use a prototype, we must include the size of the array in the prototype.

e.g.

```
void process_array (int [3][6]);
void process_array (int array[3][6])
{
        ---
```

--- }

CAUTION: It's easy to become confused here. The above function body defines a function which takes as an argument a 3 by 6 array of int. However, if we call it with:

process_array (array[3][6]);

Then we will cause a problem as this will not pass the array – this will attempt to pass the element [3][6] of the array – which is out of range anyway if the array is [3][6].Multi-dimensional arrays are actually quite rare in C – an array of pointers is more common and more useful.

When we pass a 2D array to a function we must specify the number of columns -- the number of rows is irrelevant. The reason for this is pointers again. C needs to know how many columns in order that it can jump from row to row in memory.

Consider int a[5][35] to be passed in a function:

We can do:

f (int a[][35]) {.....}

or even:

f(int (*a)[35]) {.....}

We need parenthesis (*a) since [] have a higher precedence than *

So:

int (*a)[35]; declares a pointer to an array of 35 ints.

int *a[35]; declares an array of 35 pointers to ints.

Now lets look at the (subtle) difference between pointers and arrays. Strings are a common application of this.

Consider:

char *name[10];

char Aname[10][20];

We can legally do name[3][4] and Aname[3][4] in C.

However

- Aname is a true 200 element 2D char array.
- access elements via ***20*row + col + base_address*** in memory.
- name has 10 pointer elements.

NOTE: If each pointer in name is set to point to a 20 element array then and only then will 200 chars be set aside (+ 10 elements).

The advantage of the latter is that each pointer can point to arrays be of different length.

Matrix Operations:

i) Matrix Addition

The *sum* **A**+**B** of two *m*-by-*m*atrices **A** and **B** is calculated entrywise:

$(\mathbf{A} + \mathbf{B})_{i,j} = \mathbf{A}_{i,j} + \mathbf{B}_{i,j}$, where $1 \le i \le m$ and $1 \le j \le n$.

ii) Matrix Multiplication

The product of an $i \times k$ matrix A with $k \times j$ matrix B is an $i \times j$ matrix denoted AB whose entries are

$$(AB)_{i,j} = \sum_{k=1}^{p} A_{ik} B_{kj},$$

iii) Transpose of a Matrix

The *transpose* of an *m*-by-*n*matrix **A** is the *n*-by-*m* matrix$\mathbf{A}^T$ (also denoted $\mathbf{A}^{\mathrm{tr}}$ or ${}^{\mathrm{t}}\mathbf{A}$) formed by turning rows into columns and vice versa:

$(\mathbf{A}^T)_{i,j} = \mathbf{A}_{j,i}$.

iv) Saddle point of matrix

Saddle Point of a matrix is the element which is Lowest in the row and the hieghest in the corresponding column. A matrix may have One or Two saddle points or may not have a saddle point

Algorithm:

[i] Matrix addition

Precondition: Two 2-D arrays of same size representing 2 matrices to be added.

Postcondition: Resultant 2-D array (same size as the input) after addition.

A(m:n), B(m:n) are two input, C(m:n) output matrix.

1.For i = 0 to m increment by 1

For j = 0 to n increment by 1

C(i,j) = A(i,j) + B(i,j)

2. Display C.

[ii] Matrix multiplication

Precondition: Two 2-D arrays of mxn and nxl size representing 2 matrices to be multiplied.

Postcondition: Resultant 2-D array (size mxl) after multiplication.

A(m:n), B(n:p) are two input, C(m:p) output matrix.

1.For i = 0 to m increment by 1

For j = 0 to p increment by 1

C(i,j) = 0

For k = 0 to n increment by 1

C(i,j) = C(i,j) + A(i,k)*B(k,j)

2. Display C.

[iii] Transpose of Matrix

Precondition: One 2-D array(size mxn) representing a matrix.

Postcondition: Resultant 2-D array (size nxm) after transposition.

A(m:n) is input, C(n:m) output matrix.

1.For i = 0 to m increment by 1

For j = 0 to n increment by 1

C(j,i) = A(i,j)

2. Display C.

[i] MatrixSaddle point

Precondition: 2-D array of size.

Postcondition: saddle point value with its index if exist otherwise No saddle point .

A(m:n) is input. Saddle point value with location is output.

1.For i = 0 to m increment by 1

Min:=a[i][0]

For j = 0 to n increment by 1

If(A(i,j) <= min)

Min=A(i.j)

Col=j

Endfor

Max=A(0,Col)

For k = 0 to m increment by 1

If(A(k,Col) >= Max)

Max=A(k,Col)

Endfor

2. If Max=Min

Display saddle point at i+1, col+1

3. Return

Input: 2D arrays on which matrix operations to be performed.

e.g. A= {{1,3,1},{1,0,0}} B= {{0,0,5},{7,5,0}}

Output: Resultant 2D array after performing the matrix operation.

i) Matrix Addition

$$\begin{bmatrix}1 & 3 & 1\\ 1 & 0 & 0\end{bmatrix} + \begin{bmatrix}0 & 0 & 5\\ 7 & 5 & 0\end{bmatrix} - \begin{bmatrix}1+0 & 3-0 & 1+5\\ 1+7 & 0 & 5 & 0+0\end{bmatrix} - \begin{bmatrix}1 & 3 & 6\\ 8 & 5 & 0\end{bmatrix}$$

ii) Matrix Multiplication

$$AB = \begin{bmatrix}14 & 9 & 3\\ 2 & 11 & 15\\ 0 & 12 & 17\\ 5 & 2 & 3\end{bmatrix}\begin{bmatrix}12 & 25\\ 9 & 10\\ 8 & 5\end{bmatrix} = \begin{bmatrix}273 & 455\\ 243 & 235\\ 244 & 205\\ 102 & 160\end{bmatrix}$$

iii) Transpose of a Matrix

$$\begin{bmatrix}1 & 2 & 3\\ 0 & -6 & 7\end{bmatrix}^T = \begin{bmatrix}1 & 0\\ 2 & -6\\ 3 & 7\end{bmatrix}$$

iv) Saddle point Matrix

|1 2 3|

|4 5 6| = Saddle point at (2,0)

|7 8 9|

Test Cases:

- Input data must be numeral; if not so error message should be displayed.
- For matrix operations size constraints are to be checked otherwise display message "operation can't be performed due to wrong size of input matrices"

Application:

- *Helpful in solving different operations related to 2D array.*
- *Computer Graphics operations.*
- *Graph Theory*

FAQs :

- *What is 2D array?*
- *How to pass 2D array in functions?*
- *How to allocate memory dynamically?*
- *How to make pointer to 2D array.*
- *What are matrices? Explain different operations on it.*

Instructions for writing journal:

- Title
- Problem Definition
- Concept of 2D arrays
- Pseudo C code for operations on matrices like addition, multiplication, subtraction, transpose
- Concept of double pointer or pointer of arrays or array of pointers
- Memory allocation and deallocation.
- Analysis of above for time and space complexity
- Program code
- Output for different I/P comparing time complexity
- *Conclusion*

Assignment No - 3

Title: String Operation

Problem Statement:

Perform following String operations with and without pointers to arrays (without using the library functions):

a) substring

b) palindrome

c) compare

d) copy

e) Reverse.

Objective:

- To understand the how to use string in c.
- To understand how to handle strings in functions
- To understand how to use pointer for character array for string manipulation.

S/W Packages and H/W apparatus used:

Linux OS: Ubantu/Fedora , Eclipse/codeblocks.

PC with the configuration as Pentium IV 1.7 GHz. 128M.B RAM, 40 G.B HDD, 15''Color Monitor, Keyboard, Mouse

References: 1. 'C programming' by Balguruswami.

2. ' C' Kanetkar.

3. 'Code complete' Steve Connel.

Theory:

String:

- A *sting* is a not a data type in 'C' To handle String in 'C', use character array.
- A character pointer is used instead of character array which stored base address of character array.

- **Declaring Sting:**

  char name[20]; //without pointer

  char *name; //with pointer

Memory Allocation:

For the above example, char name[20] static memory is allocated that is 20 bytes .But in second case for char *name dynamic memory allocation is done and for that whatever the length of a string plus one for null character only that much memory is allocated so memory is saved in case of pointer to string.

Algorithm:

In following algorithms, a and b are strings.

(I) Algorithm length (a, b)

Precondition ***:*** *Accept array of character.*

Postcondition ***:*** *Finding the length of string.*

Return ***:*** *length of string.*

1. *For (i=0 to a[i]!='\0')*
   a. *Increment i by 1;*
2. *Return i.*

3. *End*

(II) Algorithm copy (a, b)

Precondition ***:*** *Accept two array of character*

Postcondition ***:*** *For copy one string into other.*

Return ***:*** *Nil*

1. *for (i=0 to b[i]!='\0')*
    a. *Copy the b array content in a array.*
    b. *Increment i by 1.*
2. *Add null character at the end of a array.*
3. *End loop*
4. *End*

(III) Algorithm palindrome (a)

Precondition ***:*** *Accept array of character.*

Postcondition ***:*** *check whether string is palindrome or not and displayed the Message.*

Return ***:*** *Nil*

1. *Call the length function find of the length of a array.*
2. *Store that length in l variable.*
3. *Decrement l by 1.*
4. *for(i=0 to a[i]==a[l])*
    increment i by 1 and decrement l by 1
    a. *if (l==0)*
    b. *Print "string is palindrome".*
    c. *Else*
    d. *Print "string is not a palindrome"*
    e. *End if*
5. *Return*

(IV) Algorithm reverse (a)

Precondition *: Accept array of character.*

Post condition *: Reverse the content of character array.*

Return *: Nil*

1. *Declare j=0, revStr[max]*
2. *Call length function for a[ ]*
3. *Store length in length*
4. *For (i=length to 0)*

   Revstr[j++]=a[i]
5. *Add null character at the end of revstr array.*
6. *Print revstr array.*
7. *End*

(V) Algorithm compare (a, b)

Precondition *: Accept two array of character*

Postcondition *: For comparing first string with second string.*

Return *: integer value: zero or negative or positive*

1. *Call function length for a array*
2. *Store length in len1*
3. *Call function length for b array*
4. *Store length in len2*
5. *If len1==len2*
   i. *For i=0 to len1*

      If a[i] > b[i]

      Return(1)

      Else if a[i] < b[i]

      Return(-1)

ii. *If i==len1*

Return(0)

(VI) Algorithm concatenate (a, b)

Precondition ***:*** *Accept two array of character*

Postcondition ***:*** *For concatenating first string with second string.*

Return ***:*** *result stored in tempstr[ ] and displayed that one*

Call function length for a array

Store length in len1

Call function length for b array

Store length in len2

Declare tempStr[Max] for storing result

For I=0 to I<len1

i. *tempStr[I]=a[I]*

For J=0 to J<len2

i. *tempStr[I]=b[J]*

ii. *I=I+1*

Store null character at the end of tempStr[I]

Print tempStr[]

End

(VII) Algorithm substring (a, b)

Precondition ***:*** *Accept two array of character*

Postcondition ***:*** *checked whether first string is a substring of second b string.*

Return ***:*** *returns location of substring if found. Otherwise returns -1*

1. *Call function length for a array*

2. *Store length in len1*
3. *Call function length for b array*

4. *Store length in len2*
5. *i=0,j=0*
6. *Repeat thru step 7 while a[i]!= ' '*
7. *while(a[i]!=b[0] && a[i]!=' ')*

```
                    i++;
 if(a[i]==' ')
                    return(-1);
 firstOcc = i;
  while(a[i]==b[j] && a[i]!='' && b[j]!='')
        {
                    i=i+1
                    j=j+1
         }
if(b[j]==' ')
         return(firstOcc);
if(a[i]==' ')
        return(-1);
i = firstOcc + 1;
j = 0;
```

Input:

Input is strings.

MENU:

String1 is: Hello

String2 is: World

Enter Your Choice:

1:Enter New Strings

2:Length

3:Palindrome

4:Compare

5:Copy

6:Reverse

7:SubString

8:Exit

Output:

The Length of String1 is: 5

The Length of String2 is: 5

3:Palindrome

String1 is: madam

String2 is: madam

String1 is a Palindrome

String2 is a Palindrome

4:Compare

String1 is: Hello

String2 is: Hi

First string is alphabetically above than second.

5:Copy

String1 is: Hello

1:Copy String1 on String2

String1 is: Hello

String2 is: Hello

Do you want to use the copied string for further Operations?

Y:Yes

N:No

N

6: Reverse

String1 is: Hello

Reverse of String1 is: olleH

7:Substring

String1 is: HelloWorld

String 2: World

String 2 is substring of string1.

Application:

This assignment is helpful in solving different operations related to strings.

FAQs :

- What is String?
- How to pass string to functions?
- How to make pointer to string and how to pass it to function?
- How to initialize string?
- How to allocate memory dynamically for array of string.

Instructions for writing journal:

- Title
- Problem Definition

- Concept of character arrays
- Pseudo C code for finding Length,compare, palindrome, reverse, substring, copy of strings.
- Analysis of above for time and space complexity
- Program code
- Output for different I/P comparing time complexity
- Conclusion

Assignment No - 4

Title: Structure Operations

Problem Statement: Create a Database using array of structures and perform following operations on it:

a) Create Database
b) Display Database
c) Add record
d) Search a record
e) Modify a record.
f) Delete a record.

Objective:

- To understand the concept of structures.
- To understand how to access members of structure.
- To understand the concept of array of structure & structure as function arguments.
- To understand the concept of passing pointer of structure to function.

S/W Packages and H/W apparatus used:

Linux OS: Ubantu/Fedora , Eclipse/codeblocks,

PC with the configuration as Pentium IV 1.7 GHz. 128M.B RAM, 40 G.B HDD, 15''Color Monitor, Keyboard, Mouse

References:

1. 'C programming' by Balguruswami.
2. 'LET US C' By Yashwant Kanetkar.
3. 'Pointers in C' By Yashwant Kanetkar.

Theory:

Structure:

- A *struct* is a derived data type composed of members that are each fundamental or derived data types.
- A single *struct* would store the data for one object. An array of *struct*s would store the data for several objects.
- A *struct* can be defined in several ways as illustrated in the following examples:

Declaring Structure:

Does Not Reserve Space

struct my_example

```
{
        int label;
        char letter;
        char name[20];
} ;             /* The name "my_example" is called a structure tag    */
```

Reserves Space

```
struct my_example
{
        int label;
        char letter;
        char name[20];
} mystruct ;
```

Memory Allocation:

For the above example, mystruct is the variable of structure. The memory allocated is the sum of memory requirement of all the members of structure. Memory allocated for mystruct is 23bytes(2 bytes for lable,1 for letterm 20 for array name[]).

Typedef:

Often, t*ypedef* is used in combination with *struct* to declare a synonym (or an alias) for a structure:

```
        typedef struct              /* Define a structure */
        {
          int label ;
          char letter;
          char name[20] ;
        } Some_name ;                       /* The "alias" is Some_name */
```

Some_name mystruct ; /* Create a struct variable */

Accessing Struct Members:

- Individual members of a *struct* variable may be accessed using the structure member operator (the dot, "."):

  mystruct.letter ;

- Or , if a pointer to the *struct* has been declared and initialized

  Some_name *myptr = &mystruct ;

  by using the structure pointer operator (the "->"):

  myptr -> letter ;

  which could also be written as:

  (*myptr).letter ;

Algorithm:

```
Structure declaration:

typedef struct  {

          char name[15];

          char address[20];

          float balance;

          int acno;

  } Bankcust;
```

(VIII) Algorithm add (struct Bankcust cust[], int no)

Precondition ***:*** *Accept array of Structures.*

Postcondition ***:*** *Add new record to the array.*

Return ***:*** *current no of records.*

1. *if (no=Max)*
    b. *display list full*
    c. *goto 6*
3. *End if*
4. *increment no.*
5. *Accept details and add in the array at cust[no].*
6. *return no.*

(IX) Algorithm display_list (struct Bankcust cust[], int no)

Precondition ***:*** *Accept array of Structures*

Postcondition ***:*** *All records that exist are displayed*

Return ***:*** *Nil*

1. *for(t=0 to t<no)*
    c. *if (cust[t].acno not equal to 0)*
        I. *display contents of the customer record.*
    d. *End if*
5. *End loop*
6. *End*

(X) Algorithm search (struct Bankcust cust[], int no, int ano)

Precondition ***:*** *Accept array of Structures and account no.*

Postcondition ***:*** *Record with corresponding acno is found*

and displayed

Return ***:*** *If record found, return index of the record else return –1*

6. *for(t=0 to t<no)*
   a. *if (cust[t].acno= ano)*
      i. *return t*
   b. *End if*
7. *End loop*
8. *return -1*
9. *End*

(XI) Algorithm modify (struct Bankcust cust[], int ano, int no)

Precondition ***:*** *Accept array of Structures and customer ano.*

Postcondition ***:*** *Record with corresponding acno is modified*

Return ***:*** *Nil*

8. *found=search(cust, no, ano)*
   a. *if (found=-1)*
      i. *display- record not found*
      ii. *goto 2*
   b. *else*
      i. *loc=found*
      ii. *modify the contents of record at cust[loc]*
   c. *End if*

9. *End*

(XII) Algorithm disp_rec(struct Bankcust cust[], int no, int Ano)

Precondition *: Accept array of Structures and ano.*

Postcondition *: Record with corresponding acno is displayed*

Return *: Nil*

1. *found=search(cust, no, ano)*
   a. *if (found=-1)*
      i. *display- record not found*
      ii. *goto 2*
   b. *else*
      i. *loc=found*
      ii. *display the contents of record at cust[loc]*
   c. *End if*
2. *End*

(XIII) Algorithm delet (struct Bankcust cust[],int no, int acno)

Precondition *: Accept array of Structures and employee id.*

Postcondition *: Corresponding acno is set to 0*

Return *: updated no of records(if deleted) in no.*

1. *found=search(cust,no, acno)*
2. *if (found=-1)*
   a. *display- record not found*
   b. *goto 5*
3. *else*
   a. *loc=found*
   b. *for i=loc to no*
      1. *cust[i]=cust[i+1]*
   d. *return no-1;*
4. *End if*
5. *End*

Input:

Input is the data/element that is to be added in fields like name, address, balance account no. Account number should be auto-generated. Display records in tabular format.

MENU:

1. **NEW ACCOUNT (ADD RECORDS)**
2. **MODIFY ACCOUNT INFO**
3. **DELETE ACCOUNT**
4. **SEARCH ACCOUNT DETAILS**
5. **DISPLAY RECORDS**
6. **EXIT**

NEW ACCOUNT (INPUT/ADD RECORDS):

Account no	Name	Address	Balance
1001234	ABC PQR	PQR Street, Pune	1000.00
1001235	BBC PQR	PQR Street, Pune	2000.00
1001236	CBC PQR	PQR Street, Pune	3000.00
1001237	DBC PQR	PQR Street, Pune	4000.00
1001238	EBC PQR	PQR Street, Pune	5000.00

Output:

All operations such as add records, modify records, delete records, search records and display records are applied on bank database and the output should be displayed in tabular format.

MODIFY RECORDS:

Accept Account no: 1001236 ←

OUTPUT:

Account no 1001236 Found.

Change Name? Y/N: Y←

New Name: XBC PQR

Change Balance? Y/N: Y←

New Balance : 6000

DISPLAY RECORDS :

Account no	**Name**	**Address**	**Balance**
1001234	ABC PQR	PQR Street, Pune	1000.00
1001235	BBC PQR	PQR Street, Pune	2000.00
1001236	XBC PQR	PQR Street, Pune	6000.00
1001237	DBC PQR	PQR Street, Pune	4000.00
1001238	EBC PQR	PQR Street, Pune	5000.00

DELETE RECORDS:

Accept Account no: 1001236 ←

OUTPUT:

Account no 1001236 Found.

Delete Record? Y/N: Y←

DISPLAY RECORDS :

Account no	**Name**	**Address**	**Balance**
1001234	ABC PQR	PQR Street, Pune	1000.00
1001235	BBC PQR	PQR Street, Pune	2000.00

1001237	DBC PQR	PQR Street, Pune	4000.00
1001238	EBC PQR	PQR Street, Pune	5000.00

INPUT/ADD RECORDS:

Name of Account Holder: FBC PQR ←

Address: MNP Street, Pune ←

Initial Balance: 1000 ←

Account Created !!!

WELCOME TO XYZ BANK !

Your Account No is: 1001239

DISPLAY RECORDS :

Account no	**Name**	**Address**	**Balance**
1001234	ABC PQR	PQR Street, Pune	1000.00
1001235	BBC PQR	PQR Street, Pune	2000.00
1001237	DBC PQR	PQR Street, Pune	4000.00
1001238	EBC PQR	PQR Street, Pune	5000.00
1001239	FBC PQR	MNP Street, Pune	1000.00

Test cases:

1. *Name should be character and should be of minimum 8 characters.*
2. *Minimum Balance for account opening is 1000.00 Rs.*
3. *Account no should not be modified.*
4. *After deletion of account, the account no should not be reallocated.*
5. *Mobile number must be of 10 characters only.*

6.

Application:

This assignment is helpful in solving different operations related to structures. This application effectively can be used with files, where database can be stored.

Practice Assignments:

1) Write a menu driven program that depicts the working of a library. The menu options should be :

a) Add book information.
b) Display book information.
c) List all books of given Author.
d) List the title of specified book.
e) List the counts of books in the library.
f) Exit.

2) Write a program to declare structure data members are real and imaginary.Write a function add (),mul() to perform the given operations.

FAQs :

- What is Structure, array of structure?
- How to pass structures in functions.
- How to allocate dynamic memory for structure variables.
- What is difference between structure and union. Explain with example.

Instructions for writing journal:

- Title
- Problem Definition
- Theory: Concept of structure, fuctions, passing structure to function, structure using pointers, array of structure

- Pseudo C code for Insert, display, modify and delete records.
- Analysis of above for time and space complexity
- Program code
- Output for different I/P comparing time complexity
- Conclusion

Assignment No : 5

Title: FILE operations

Problem Statement:

a) Write C program to implement TYPE and COPY commands of DOS using command line arguments.

b) Implement sequential file and perform following operations on any database:

i. Display
ii. Add records
iii. Search record
iv. Modify record
v. Delete record

Objective:

- To understand FILE structure
- To understand the concept of sequential files
- To know different modes to access file
- To understand create, read, write, append, modify delete, search Creations on sequential file.

- This assignment gives us another way to do implement basic DOS commands with help of command line arguments.

Outcome:

Students will be able to learn and implement sequential file concept.

S/W Packages and H/W apparatus used:

Linux OS: Ubantu/Fedora , Eclipse/codeblocks,

PC with the configuration as Pentium IV 1.7 GHz. 128M.B RAM, 40 G.B HDD,

15''Color Monitor, Keyboard, Mouse

References :

- "C programming " Balguruswamy.
- "Fundamentals of datastructures" sartaj sahani.
- Data structures " Tananbum

Theory:

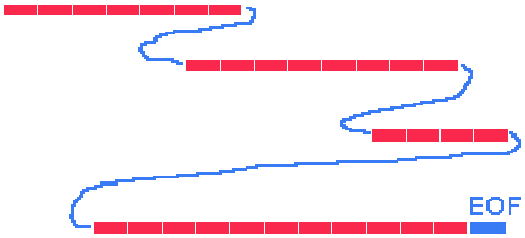
A file is a named area of secondary storage. Secondary storage is permanent. The content of a file, unlike the contents of primary memory, which is volatile, is accessible after we have turned the power off and back on.

A file is not necessarily stored contiguously on the storage device: the file may be fragmented. The operating system controls the fragmentation, if any.

All peripheral devices allow files to be processed sequentially: you start at the beginning of the file and work through each record in turn. One important advantage of sequential files is that different records can have different lengths; the minimum record length is zero but the maximum is system-dependent.

Sequential files behave as if there were a pointer attached to the file which always indicates the next record to be transferred. On devices such as terminals and printers you can only read or write in strict sequential order, but when a file is stored on disc or tape it is possible to use the REWIND statement to reset this pointer to the start of the file, allowing it to be read in

again or re-written. On suitable files the BACKSPACE statement can be used to move the pointer back by one record so that the last record can be read again or over-written.



BYTES

The fundamental unit of a file is a byte. A file concludes with a special mark called the end of file mark. This mark is defined as **EOF**, typically with a value of **-1**. Under the C standard, the end of file mark may be any negative integer value.

FORMAT

We store data in a file in either of two formats:

- binary format, or
- text format.

In binary format, the data on the file is identical to the data stored in primary memory. Each byte on the file is a direct image of the corresponding byte in memory. In other words, there is no translation.

In text format, the data on the file is in a form that we can display and modify using a text editor. The data on the file is a translated form of the data stored in primary memory. In text

format, we may introduce some approximation. Consider the floating-point number 12.345678901234 stored as a **double** in memory. Let us convert this value using the **%.2lf** specifies. The converted value is 12.35. This is the value that the program stores in the file. If subsequently the program reads 12.35 from the file under the **%lf** conversion specifier, it stores the value in primary memory as 12.350000000000. The value stored in primary memory is clearly different from the original 12.345678901234! In other words, although text format provides the ability to edit text, there may be some loss of accuracy with floating point numbers.

Aside from readability, another advantage of text format is portability. We can move the data from one platform to another as long as the characters in the file belong to a standard character set shared by the platforms. The standard character set (IEC/ISO 646-1083 Invariant Code Set) consists of

- 52 upper and lower case alphabetic characters
- 10 digits
- space
- horizontal tab, vertical tab and form feed
- 29 graphic characters: ! # % ^ & * (_) - + = ~ [ ] ' | \ ; : " { } , . < > / ?

ACCESS

A program accesses a file as a stream of bytes. Typically, the file consists of records. The program can access the records in either of two ways

- randomly (like CD's) or
- sequentially (like Cassette Tapes).

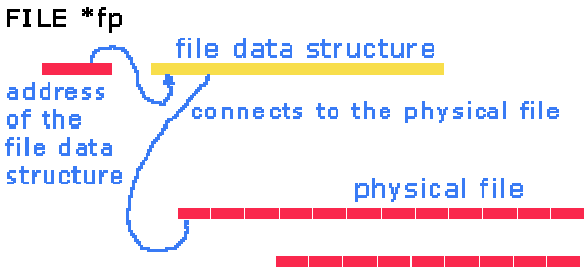
To access records randomly, we need to fix their size. With the record sizes fixed, we can determine the location of the record that we are seeking, jump to that location and access that record directly. In other words, we can skip the intervening records without reading them.

Under sequential access, we access the records in the order of creation. In this case, the records can vary in size. We do not need to know their sizes beforehand.

In this subject, we focus on sequential text files.

CONNECTING

We connect a file to our program using a FILE data structure. This data structure holds the connection information for the file.



We allocate space for the address of this data structure in a declaration of the form

FILE *identifier;

identifer is the name of a pointer that will hold the address of the data structure. The data type **FILE** is defined in **<stdio.h>**.

For example, to allocate memory for the address of a file data structure, we write

```
#include <stdio.h>

FILE *fp = NULL;
```

We initialize the pointer **fp** to **NULL** as a precaution against inadvertent premature dereferencing.

If, for some reason, our program tries to access data at **fp** before our program has assigned an address to **fp**, our program will fail because dereferencing a **NULL** address is prohibited. **NULL** is defined as the null address in **<stdio.h>**.

OPENING

The library function **fopen()** connects a specific file to a program. **fopen()** returns the address of the file connection data structure for the named file. The prototype for **fopen()** is

```
FILE *fopen(char file_name[], char mode[]);
```

The first parameter is a null-byte terminated string containing the name of the file. The second parameter is a null-byte terminated string containing the connection mode.

The most common connection modes are

- **"r"** - read from the file,
- **"w"** - write to the file: if the file exists, truncate its contents and then write; if the file does not exist, create a new file and then write to that file,
- **"a"** - write to the end of the file: if the file exists, append to the end of the file; if the file does not exist, create it and then write.

The mode parameter is a null-byte terminated string (NOT A CHARACTER). The other connection modes for text files are

- **"r+"** - opens the file for reading and possibly writing,
- **"w+"** - opens the file for writing and possibly reading; if the file exists, truncates its contents and then writes to the file; if the file does not exist, creates a new file and then writes to that file,
- **"a+"** - opens the file for writing to the end of the file and possibly reading; if the file exists, appends to the end of the file; if the file does not exist, creates it and then writes to the file.

fopen returns **NULL** if the attempt to connect to the file fails. **fopen** can fail for lack of permission, premature removal of the secondary storage medium, device full, etc.

Closing

The library function **fclose()** disconnects a file from a program. **fclose()** takes as its only parameter the address of the file data structure. The prototype for **fclose()** is

int fclose(FILE *);

If the program opened the file for writing, **fclose()** writes any data remaining in the file stream's buffer to the file and concludes b appending an end of file mark immediately after the last character. If the program opened the file for reading, **fclose()** ignores any data left in the file

stream's buffer and closes the connection. For example, to open a file named **alpha.txt** for writing and then to close the file, we write

Opening and closing a file

```
#include <stdio.h>

int main( ) {

   FILE *fp = NULL;

   fp = fopen("alpha.txt","w");

   if (fp != NULL) {

      /* we will add statements here later */

      fclose(fp);

   } else

      printf("Failed to open file\n");

  return 0;

}
```

fclose() returns 0 if successful, **EOF** if unsuccessful. **fclose()** can fail if the secondary storage medium is full, an I/O error occurs or the medium has been prematurely removed.

Writing to a File

The library function **fprintf()** sends data from primary memory to a connected file under format control. The prototype for **fprintf()** is

int fprintf(FILE *, char [], ...);

The first parameter receives the address of the file connection data structure. The second parameter is the format string containing the text to be written directly to the file and the conversion specifiers, if any, to be applied to the data values received in the parameters following the format string. Note the similarity between the **fprintf()** and the **printf()** library functions.

For example:

```
 #include <stdio.h>

int main( ) {

    FILE *fp = NULL;

    char phrase[] = "My name is Arnold";

    fp = fopen("alpha.txt","w");

    if (fp != NULL) {

        fprintf(fp, "%s\n", phrase);

        fclose(fp);

    } else if (fp != NULL) {

        fprintf(fp, "%s\n", phrase);

        fclose(fp);

    }

else
```

```
        printf("Failed to open file\n");
    return 0;
}
```

The library function **fputc()** writes a single character to a file. The prototype for **fputc()** is:

```
int fputc(int ch, FILE *fp);
```

ch is the character to be written and **fp** is the address of the connection data structure for the destination file. **fputc()** returns the character written, or **EOF** in the event of an error.

The library function **fputs()** writes a null-byte terminated character to a file. The prototype for **fputs()** is:

```
int fputs(char str[], FILE *fp);
```

str is the string to be written and **fp** is the address of the connection data structure for the destination file. **fputs()** returns a non-negative value if successful, or **EOF** in the event of an error.

Reading from the file

The library function **fscanf()** reads a sequence of bytes from a connected file into primary memory under format control. The prototype for **fscanf()** is

```
int fscanf(FILE *, char [], ...);
```

The first parameter receives the address of the file connection data structure. The second parameter is the format string containing the conversion specifiers to be applied to the subsequent parameters while converting the byte subsets into specific data types.

For example:

Read From File

```
#include <stdio.h>
 int main( )
{
    FILE *fp = NULL;
    char phrase[61];
    fp = fopen("alpha.txt","r");
    if (fp != NULL) {
       fscanf(fp, "%60[^\n]\n", phrase);
       printf("You read : %s\n", phrase);
       fclose(fp);
    } else
       printf("Failed to open file\n");
    return 0;
 }
```

The library function **fgetc()** extracts a single character from a file. The prototype for **fgetc()** is

```
int fgetc(FILE *fp);
```

fp is the address of the connection data structure for the file. **fgetc()** returns the next character from **fp**, or **EOF** in the event of an error.

The library function **fgets()** extracts a contiguous series of bytes from a file. The prototype for **fgets()** is

```
char* fgets(char str[], int max, FILE *fp);
```

The string **str** will hold the set of bytes ending in a newline character up to **max-1** bytes from **fp**. **fgets** appends the null byte to the string stored in primary memory. **fgets()** returns the address of **str**, or **NULL** in the event of an end of file or an I/O error.

Algorithms:

I. COPY Command ( )

i. start
ii. Open the Source File (eg "Data1.txt") in Reading Mode.
iii. Open the destination File (eg."Data2.Txt") in Writing Mode.
iv. While not end of file encountered
a. Read Content of Data1.Txt Record by Record Or character by character
b. Copy it to Dtata2.Txt in the same order.
// End while
v. Close the both Source and Destination files.
vi. End of COPY Command

II. TYPE command( ).

i. Start
ii. Open the File (eg "Data2.txt") in Reading Mode.
iii. While not end of file encountered
a. Read Content of Data2.Txt Record by Record Or character by character.
b. Display the same on the output screen.
// End while
iv. Close Dtat2.Txt file.
v. end of TYPE command.

III. Implement sequential file and perform following operations on any database:

i. **Display**
ii. Add records
iii. Search record
iv. Modify record
v. Delete record

File_operation( )

i. Start

ii. If (File is not exisiting)

a. Open a file in write mode (if no i/p file).

b. add records in the file .

c. Close a file.

d. Goto step iii

// file is already ready to read.

iii. a. Open a file in read mode

b display the records from the file

c. Close the file

iv. a. Open a file in write mode

b. search the record from the file tobe modified

c. enter the new data for the record.

d. display the records with modified record

v. a. Open a file in write mode

b. search the record from the file tobe deleted

c. delete the record

d. shift all the record above

d. display the

Input:

1 : command line argument
Copy and Type command

- Input to this program is already exiting files.

2. Reading file to count characters, words etc.

- Two options for i/p
  Use already exiting files.
  Create New text file and insert the text data .

Output:

1. Command Line Argument :
   - For Copy Command Show the status of copy command.
   - For Type command Display content of file on the output screen
2. Read File

- Output is stored in the another file.

Input Validations :

1. Command line Argument

- *Make sure i/p file is already present*
- *User enter a file which is not existing show appropriate error massage.*

2. If File is existing

- *It shout not be null or Empty.*

Application :

It is an simulation of Dos commands. Used in handling different types of files.

FAQS :

- ✍ *How it is useful.*
- ✍ *What are the different ways use files.*
- ✍ *What are the Different types of files.*

Instructions for writing journal:

- Title
- Problem Definition
- Concept of 2D arrays
- Pseudo C code for file operations
- Memory allocation and deallocation.
- Analysis of above for time and space complexity
- Program code

Assignment No.6

Title: Sorting & Searching operations

Problem Statement: Accept student information (e.g. RollNo, Name, Percentage etc.).

a) Display the data in ascending order of name (Bubble Sort)

b) Display the data in descending order of name(Selection sort)

c) Display data for RollNo specified by user (Binary search)

d) Display the number of passes and comparisons for different test cases (Worst, Average, Best case).

Objective :

- This assignment is designed to implement different sorting methods.
- Analyze the given algorithms w.r.t Time and space requirement.

Outcome:

Students will be able to select the searching algorithm and also apply sorting techniques.

S/W Packages and H/W apparatus used:

Linux OS: Ubantu/Fedora , Eclipse/codeblocks,

PC with the configuration as Pentium IV 1.7 GHz. 128M.B RAM, 40 G.B HDD,

15''Color Monitor, Keyboard, Mouse

References: 1. 'C programming' by Balguruswami.

2. ‘LET US C’ By Yashwant Kanetkar.

3. ‘Pointers in C’ By Yashwant Kanetkar.

Theory:

Sorting : A process that organizes a collection of data into either ascending or descending order.

- Data items to be sorted can be
  - Integers
  - Character strings
  - Objects

- Sort key
  - The part of a record that determines the sorted order of the entire record within a collection of records

Bubble Sort Technique:

- Compare each element (except the last one) with its neighbor to the right
  - If they are out of order, swap them
  - This puts the largest element at the very end
  - The last element is now in the correct and final place
- Compare each element (except the last *two*) with its neighbor to the right
  - If they are out of order, swap them
  - This puts the second largest element next to last
  - The last two elements are now in their correct and final places
- Compare each element (except the last *three*) with its neighbor to the right
  - Continue as above until you have no unsorted elements on the left

Example:

Linear Search:

A linear search is the most basic of search algorithm you can have. A linear search sequentially moves through your collection (or data structure) looking for a matching value.

Example:

Search 6 in sequence {2, 5, 6, 3, 8}

Result: Found at 3'rd position

The worst case performance scenario for a linear search is that it needs to loop through the entire collection; either because the item is the last one, or because the item isn't found. In other words, if you have *N* items in your collection, the worst case scenario to find an item is *N* iterations. This is known as *O(N)* using the Big O Notation. The speed of search grows linearly with the number of items within your collection.

Linear searches don't require the collection to be sorted.

Selection sort: is one of the O(n2) sorting algorithms, which makes it quite inefficient for sorting large data volumes. The idea of algorithm is quite simple. Array is imaginary divided into two parts - sorted one and unsorted one. At the beginning, sorted part is empty, while unsorted one contains whole array. At every step, algorithm finds minimal element in the unsorted part and adds it to the end of the sorted one. When unsorted part becomes empty, algorithm stops.

When algorithm sorts an array, it swaps first element of unsorted part with minimal element and then it is included to the sorted part. This implementation of selection sort in not stable. In case of linked list is sorted, and, instead of swaps, minimal element is linked to the unsorted part, selection sort is stable.

Given an array of length n,

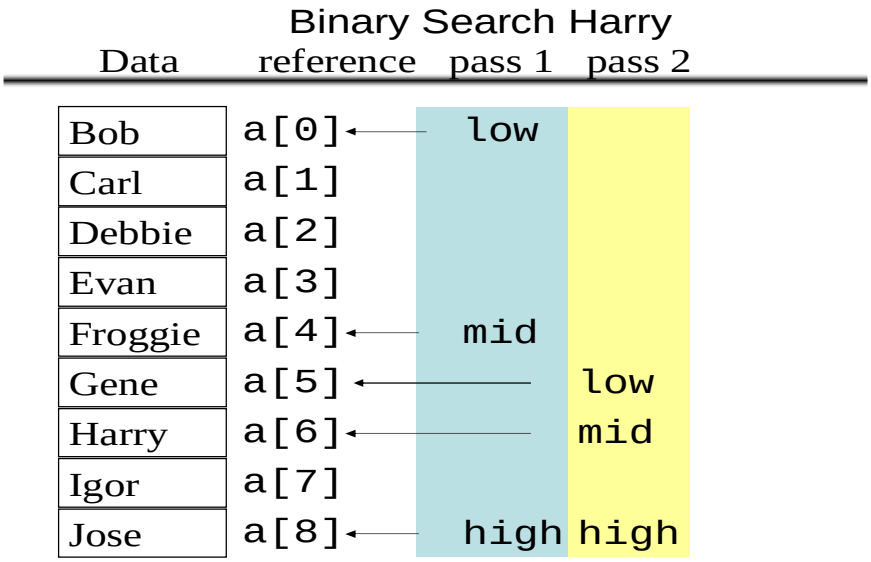
- Search elements 0 through n-1 and select the smallest

        - Swap it with the element in location 0
    - Search elements 1 through n-1 and select the smallest
        - Swap it with the element in location 1
    - Search elements 2 through n-1 and select the smallest
        - Swap it with the element in location 2
    - Search elements 3 through n-1 and select the smallest
        - Swap it with the element in location 3
    - Continue in this fashion until there's nothing left to search

Binary Search: To find a value in unsorted array, we should look through elements of an array one by one, until searched value is found. In case of searched value is absent from array, we go through all elements. In average, complexity of such an algorithm is proportional to the length of the array.

Situation changes significantly, when array is sorted. If we know it, random access capability can be utilized very efficiently to find searched value quick. Cost of searching algorithm reduces to binary logarithm of the array length. For reference, $\log_2(1\ 000\ 000) \approx 20$. It means, that **in worst case**, algorithm makes 20 steps to find a value in sorted array of a million elements or to say, that it doesn't present it the array.

- Binary Search works only on sorted arrays like this
    - Compare the element in the middle
    - if that's the target, quit and report success
    - if the key is smaller, search the array to the left
    - otherwise search the array to the right
- This process repeats until the target is found or there is nothing left to search
- Each comparison narrows search by half



Algorithm:

```
Structure declaration:

typedef struct  {
          int  Rollno;
          char Name[20];
          float percent;
  }student;
```

Algorithm:

1 . Function Linear Search(student s[], int rno, int N): rno is the number to be searched, s[N] is the Array of elements N, I is the index of array.

1. I=1

2. while (s[I].Rollno != rno)
I=I+1

3. If I=N

A. Then Print ("Unsuccessful Search")

a. Return(0)

B. Else Print ("Successful Search")

Print ("No of comparisons are", I)

Return(I)

2. Function BubbleSort(student s[], int N): s[N] is the Array of elements N which we want to sort in descending order of percentage, I is the index of array.

Precondition : Accept array of Structures & no of records.
Postcondition : Records sorted in ascending order of name.
Return : none.

```
1. LAST=N

2. for (Pass= 1;Pass<N;Pas=Pass+1)
        EXCH=0
        For( I=1;I<LAST;I++)
                If( s[I]. percent < s[I+1].percent)
                        T=s[I].percent
                        s [I].percent=s[I+1].percent
                        s [I+1].percent=T
                        EXCH=EXCH+1

            End If
            If EXCH== 0
                Print("Total No of passes are",Pass);
                return
            Else
                Display(s,N)
                LAST=LAST-1
3. Return
```

3. Algorithm selection(struct student u[10],int count)

Precondition : Accept array of Structures & no of records.
Postcondition : Records sorted in ascending order of name.
Return : none.

```
1. for (i=0 to no_rec-1)
    a. k=i;
    b. for(j=i+1;j<no_rec;j++)
        i. comp++;
       ii. if((strcmp(u[j]->name,u[k]->name))<0)
```

1. *k=j;*

c. *if(k!=i)*

i. *t=u[i];*

ii. *u[i]=u[k];*

iii. *u[k]=t;*

end.

4. Algorithm binary_search (struct student u[10],int no_rec,char key)

Precondition : Accept array of Structures, no of records & key to be searched.
Postcondition : mobile no(key) is searched & result is displayed.
Return : none.

1. *low=0;*
2. *high=rec_no-1;*
3. *found=-1;*
4. *while(low<=high)*
   i. *mid=(low+high)/2;*
   ii. *if(strcmp(key,u[mid]->mobno)>0)*
      1. *high=mid-1;*
   iii. *else if(strcmp(key,u[mid]->mobno)<0)*
      1. *low=mid+1;*

   else
      2. *found=mid;*
      3. *goto step5;*

5. *if(found==-1)*
   i. *display "Record not found"*
6. *else*
   i. *display "Record not found"*
7. *end.*

Input:

Data in the form an array of structure.

Menu:

Enter Your Choice:

1:Accept Data

2:Display Data

3:Sort By Percentage

4:Search

5:Exit

Output :

For each algorithm output should be in the form of:

1. Original List of numbers.
2. Number to be searched with its position.

1.Accept :

Enter Your Choice:
1:Accept Data
2:Display Data
3:Sort By Percentage
4:Search
5:Exit
1

Enter the no. of Students to Add:
5
Enter Data for Roll No. 2501
Enter Name:
amit
Enter Marks in Physics:
60
Enter Marks in Chemistry:
70
Enter Marks in Maths:
80
Percentage: 70.00%

Enter Data for Roll No. 2502

Enter Name:
sushan
Enter Marks in Physics:
55
Enter Marks in Chemistry:
65
Enter Marks in Maths:
72
Percentage: 64.00%

Enter Data for Roll No. 2503
Enter Name:
shlok
Enter Marks in Physics:
56
Enter Marks in Chemistry:
67
Enter Marks in Maths:
70
Percentage: 64.33%

Enter Data for Roll No. 2504
Enter Name:
arnav

Enter Marks in Physics:
60
Enter Marks in Chemistry:
71
Enter Marks in Maths:
75
Percentage: 68.67%

Enter Data for Roll No. 2505
Enter Name:
arya
Enter Marks in Physics:
64

Enter Marks in Chemistry:
76
Enter Marks in Maths:
81
Percentage: 73.67%
Data Accepted!

2:Display Data

Enter Your Choice:
1:Accept Data
2:Display Data
3:Sort By Percentage
4:Search
5:Exit
2

Roll No	Name	Phy	Chem	Maths	Percent
2501	amit	60	70	80	70.00%
2502	sushan	55	65	72	64.00%
2503	shlok	56	67	70	64.33%
2504	arnav	60	71	75	68.67%
2505	arya	64	76	81	73.67%

Enter Your Choice:
1:Accept Data
2:Display Data
3:Sort By Percentage
4:Search
5:Exit
3
Bubble Sort:

2502	sushan	55	65	72	64.00%
2503	shlok	56	67	70	64.33%
2504	arnav	60	71	75	68.67%

No of Comparisons : 6
No of Exchanges : 5

Pass 2:

Roll No	Name	Phy	Chem	Maths	Percent
2505	arya	64	76	81	73.67%
2501	amit	60	70	80	70.00%
2504	arnav	60	71	75	68.67%
2502	sushan	55	65	72	64.00%
2503	shlok	56	67	70	64.33%

No of Comparisons : 5
No of Exchanges : 2
Pass 3:

Roll No	Name	Phy	Chem	Maths	Percent
2505	arya	64	76	81	73.67%
2501	amit	60	70	80	70.00%
2504	arnav	60	71	75	68.67%
2503	shlok	56	67	70	64.33%
2502	sushan	55	65	72	64.00%

No of Comparisons : 4
No of Exchanges : 1

After Sorting:

Roll No	Name	Phy	Chem	Maths	Percent
2505	arya	64	76	81	73.67%
2501	amit	60	70	80	70.00%
2504	arnav	60	71	75	68.67%
2503	shlok	56	67	70	64.33%
2502	sushan	55	65	72	64.00%

Total Passes : 4

Total Comparisons : 18

Total Exchanges : 8

It is an Average Case!
4:Search

Enter Your Choice:
1:Accept Data
2:Display Data
3:Sort By Percentage
4:Search
5:Exit
4

Enter Roll No. to Search:
2504

Element Found at : Position 2

Roll No. : 2504
Name : arnav

Physics : 60
Chemistry : 71
Maths : 75
Percentage : 68.67%

Total Comparisons : 3
It is an Average Case!

Application:

Useful in managing large amount of data.

FAQ:

- What is the need of sorting
- What is searching?
- How to get output after each pass?
- What do you mean by PASS?
- What is recursion?
- Where the values are stored in recursion?
- Explain the notation Ω, θ, O for time analysis.
- Explain the time complexity of bubble sort and linear search, binary search, selection sort.

Instructions for writing journal:

- Title
- Problem Definition
- Concept of arrays
- Algorithms for Bubble sort, linear search.
- Analysis of above for time and space complexity
- Program code
- Output for different I/P comparing time complexity
- Conclusion

Assignment No - 7

Title: Quick Sort

Problem Statement: Accept Mobile user information (e.g. MobileNo, Name, BillAmount etc.).

a) Display the data in descending order of Name (Quicksort, recursive)
b) Display the data in ascending order of MobileNo (Quicksort nonrecursive)
c) Display pivot position and its corresponding list in each pass.
d) Display the number of passes and comparisons for different test cases (Worst, Average, Best case).

Objective:

- To understand the concept of quick Sort and divide and conquer.
- To understand complexity of quick sort.

Outcome:

Students will be able to apply quick sort.

S/W Packages and H/W apparatus used:

Windows 2000, Turbo C++,

PC with the configuration as Pentium IV 1.7 GHz. 128M.B RAM, 40 G.B HDD,

15''Color Monitor, Keyboard, Mouse

References:

1. 'Fundamentals of Data Structures' by Horowitz and Sahani.

2. 'LET US C' By Yashwant Kanetkar.

Theory:

Divide and Conquer

1. **Base Case**, solve the problem **directly** if it is small enough.
2. **Divide** the problem into two or more **similar and smaller** subproblems.
3. **Recursively** solve the subproblems.
4. **Combine** solutions to the subproblems.

Quick Sort

1. **Divide**:
    a. Pick any element **p** as the **pivot**, e.g, the first element
    b. Partition the remaining elements into
        i. **FirstPart, which contains all elements < p**
        ii. **SecondPart, which contains all elements ≥ p**
2. **Recursively sort** the FirstPart and SecondPart
3. **Combine**: no work is necessary since sorting is done in place.

Complexity of Quick-Sort: Most of the work done in partitioning

- Average case takes **Θ(n log(n))** time
- Worst case takes **$\Theta(n^2)$** time
- **Θ(1) space**

Algorithm:

1. ***Algorithm Quick-Sort(A, left, right)***

Precondition ***:*** *Accept unsorted array in array A*
Postcondition ***:*** *Sorted array in A.*
Return ***:*** *Sorted array.*

1. ***if left ≥ right return***
2. ***else***

i. ***middle ← Partition(A, left, right)***
ii. ***Quick-Sort(A, left, middle–1)***
iii. ***Quick-Sort(A, middle+1, right)***

3. ***end if***

2. ***Algorithm Partition(A, left, right)***

Precondition : *Accept part array in array A*
Postcondition : *Division of A w.r.t. i such that elements left of i are less than i and elements right of i are greater than i .*
Return : *.position for next partition in i.*

1. ***x ← A[left]***
2. ***i ← left***
3. ***for j ← left+1 to right***
a. ***if A[j] < x then***
i. ***i ← i + 1***
ii. ***swap(A[i], A[j])***
b. ***end if***
4. ***end for j***
5. ***swap(A[i], A[left])***
6. ***return i***

Input**:**

Input no of elements to be sorted using quick sort.

MENU:

7. **Quick Sort**
8. **EXIT**

Output**:**

Enter the Number of Elements: 0

Invalid Entry!

Enter the Number of Elements: -10

Invalid Entry!

Enter the Number of Elements: 10

Enter the Elements of the Array: 56 -90 80 78 234 654 432 12 0 -11

The Entered Array is:

56 -90 80 78 234 654 432 12 0 -11

Pivot=56

Left Part:12 -90 -11 0

Right Part:654 432 234 78 80

Pivot=12

Left Part:0 -90 -11

Right Part:--

Pivot=0

Left Part:-11 -90

Right Part:--

Pivot=-11

Left Part:-90

Right Part:--

Pivot=654

Left Part:80 432 234 78

Right Part:--

Pivot=80

Left Part:78

Right Part:234 432

Pivot=234

Left Part:--

Right Part:432

The Sorted Array is:

-90 -11 0 12 56 78 80 234 432 654

The No. of Passes is: 6

TEST CASES:

7. **Sorting on valid integers.If no of elements are 0 or <0**
8. **Display error if wrong option gets selected.**

Application:

Sorting

Practice Assignments:

Write a program to sort list of alphabets/ strings using Quick Sort. Display division and result of every pass.

FAQs :

- What is Divide and conquer methodology?
- How the pivot is selected and partition is done in quick sort?
- What is the complexity of quick sort?

Instructions for writing journal:

- Title
- Problem Definition
- Theory: Quick sort technique.
- Pseudo C code for Quick Sort
- Analysis of above for time and space complexity
- Program code
- Output for different I/P comparing time complexity
- Conclusion

Assignment No - 8

Title: Sparse Matrix

Problem Statement:

Implement Sparse matrix and perform following operations on it:

- Addition
- Simple Transpose
- Fast Transpose

Objective:

- To understand the concept sparse matrix.
- To understand complexity of sparse matrix addition.
- To understand complexities of sparse matrix transpose.
- To understand the difference between simple transpose and fast transpose.

Outcome:

Students will be able to implement sparse matrix.

S/W Packages and H/W apparatus used:

Windows 2000, Turbo C++,

PC with the configuration as Pentium IV 1.7 GHz. 128M.B RAM, 40 G.B HDD,

15''Color Monitor, Keyboard, Mouse

References: 1. 'Fundamentals of Data Structures' by Horowitz and Sahani.

2. 'LET US C' By Yashwant Kanetkar.

Theory:

A sparse matrix is a matrix populated primarily with zeros.

If we perform different operations on such matrix with regular matrix representation, then memory and time can be wasted.

Sparse Matrix Representation:
Sparse matrix can be represented using triplet <row, col, value>.

Row	Col	value
4	6	7
0	0	1
0	5	2
1	2	3
1	3	1
2	4	5
3	0	6
3	4	7

Algorithm:

```
Structure declaration:

typedef struct  {

           int row;

           int col;

           int value;

  } sparseMat;
```

(XIV) Algorithm Transpose (SP1, SP2)

Precondition ***:*** *Accept Sparse matrix SP1.*
Postcondition ***:*** *Transpose of Sparse matrix SP1 in sparse matrix SP2.*
Return ***:*** *none.*

*(m,n,t)← (SP1(0,1),SP1(0,2),SP1(0,3)) //**m**,n,t are initialized with total no of Row, column, Nonzero values in the input matrix*
(SP2(0,1),SP2(0,2),SP2(0,3)) ← (n,m,t),

If t<=0 then return
q←1
for col←1 to n do *// column wise scan the SP1*
for p← 1 to t do *// row no in the col*

```
            if SP1(p,2)=col                    // do the transpose columwise
            then [ (SP2(q,1),SP2(q,2),SP2(q,3)) ← (SP1(p,2),SP1(p,1),SP1(p,3))
                    q←q+1 ]
            end
        end
    end TRANSPOSE
```

Why Fast Transpose ?

Drawback with simple Transpose is overcome with Fast Transpose.

(XV) Algorithm Fast Transpose (SP1,SP2)

Precondition : Accept Sparse matrix SP1.
Postcondition : FastTranspose of Sparse matrix SP1 in sparse matrix SP2.
Return : none.

```
Declare Start_Pos (1:n), No_of_ Terms(1,n)  //  These are the temporary array

// Start_Pos(1:n ) :: - To Hold Starting position of each reow in the transposed matrix
SP2.
// No_of_Terms(1:n) :: - To Hold no of elements in the each columnn
// n  :: - Is the No of nonzero elements in the SP1.
// store the no. of rows, columns, nonzero values  of input matrix SP1 to SP2
(m,n,t)← (SP1(0,1),SP1(0,2),SP1(0,3))        // m= no of rows, n=no of columns
(SP2(0,1),SP2(0,2),SP2(0,3)) ← (n,m,t)      // t= no of terms

If t<=0 then  return -1  // empty  input matrix SP1
for i←0 to n do
No_Of_Terms(i) ← 0      end // initialize No_of _Terms( i ) with 0

for i←1 to t do
  No_of_Terms(SP1(i,2)) ← No_of_Terms(SP1(i,2))+1
end
Start_Pos(0) ← 1   //Starting  position of first row in SP2 is always 1.
for i←1 to n-1 do
  Start _Pos(i) ← Start_Pos(i-1)+S(i-1) // calculate starting position of each row
                                          In resultant matrix.
end

for i←1 to t do
  j ← A(1,2)  //take the column for copy
  (SP2(T(j),1),SP2(T(j),2),B(T(j),3)) ← (SP1(i,2),SP1(i,1),SP1(i,3)) //cop
  T(j) ← T(j)+1 // increment to next position for same col/row  element
end
```

//end FAST TRANSPOSE

Input:

Input simple matrix and convert the matrix into sparse form.

MENU:

9. INPUT MATRIX

10. DISPLAY SPARSE MATRIX

11. SIMPLE TRANSPOSE

12. FAST TRANSPOSE

13. ADDITION

14. EXIT

Output:

All operations such as add records, modify records, delete records, search records and display records are applied on bank database and the output should be displayed in tabular format.

Sample I/O :

Input matrix:

Enter the Rows and Columns of Matrix1:

3 3

Enter the Elements:

0 0 0

0 9 0

0 7 0

Enter the Rows and Columns of Matrix2:

3 3

Enter the Elements:

0 0 0

6 3 0

0 7 0

Sparse Matrix1 is:

Row	Col	Value
3	3	2
1	1	9
2	1	7

Sparse Matrix2 is:

Row Col Value

3 3 3

1 0 6

1 1 3

2 1 7

Menu Addition:

The Addition is:

Row Col Value

3 3 3

1 0 6

1 1 12

2 1 14

Simple Transpose:

Input sparse matrix A is:

Row Col value

6 6 8

0 0 15

0 3 22

0 5 -15

1 1 11

1 2 3

2 3 -6

4 0 91

5 2 28

Simple Transpose of Matrix A is:

Row Col Value

6 6 8

0 0 15

0 4 91

1 1 11

2 1 3

2 5 28

3 0 22

3 2 -6

5 0 -15

Fast Transpose of Matrix A is:

Row Col Value

6 6 8

0 0 15

0 4 91

1 1 11

2 1 3

2 5 28

3 0 22

3 2 -6

5 0 -15

TEST CASES:

1. Sparse matrix should be generated from regular matrix.
2. In addition of two sparse matrices, if result is zero then don't include in resultant sparse.
3. Display error if wrong option gets selected.
4. Sparse cannot be displayed if input matrix is not taken.

Application:

This assignment is helpful in different applications related to image processing and graphs.

Practice Assignments:

Write a program to enter the matrix and convert it into sparse matrix and perform the different operations on it.

FAQs :

- What is sparse matrix?
- How to convert simple matrix into sparse matrix?
- What is the use of sparse matrix?
- What is the structure of sparse matrix?
- Discuss various Representation for sparse.
- Explain various operations of sparse matrix.
- Compare simple and fast transpose.

Instructions for writing journal:

- Title
- Problem Definition
- Theory: Concept of Sparse and need of sparse.
- Algorithms for Simple transpose and fast transpose
- Analysis of above for time and space complexity
- Program code
- Output for different I/P comparing time complexity
- Conclusion

Assignment No - 9

Title: Singly linked list

Problem Statement: Create a singly linked list with options:

a) Insert (at front, at end, in the middle),

b) Delete (at front, at end, in the middle),

c) Display,

d) Display Reverse,

e) Revert the SLL

Objective:

- To understand the concept of singly linked list.
- To understand the concept of dynamic memory allocation.
- To understand how to use single linked list to implement other data structures.

Outcome:-

S/W Packages and H/W apparatus used:

Linux OS: Ubantu/Fedora , Eclipse/codeblocks,

PC with the configuration as Pentium IV 1.7 GHz. 128M.B RAM, 40 G.B HDD,

15''Color Monitor, Keyboard, Mouse

References:

1. 'C programming' by Balguruswami.
2. ' C' Kanetkar.
3. 'Code complete' Steve Connel.

Theory:

Linked list:

- Linked list is one of the fundamental data structure, and can be used to implement other data structures.
- It consists of a sequence of nodes, each containing arbitrary data fields and one or two references ("links") pointing to the next and/or previous nodes

- Head pointer to the first node
- The last node points to NULL

Singly Linked List:

- The simplest kind of linked list is a singly-linked list which has one link per node.
- This link points to the next node in the list, or to a null value or empty list if it is the final node.
- A singly linked list's node is divided into two parts. The first part holds or points to information about the node, and second part holds the address of next node.
- A singly linked list travels one way.

Memory allocation functions:

Malloc-

- The malloc function is one of the functions in standard C to allocate memory without initialization.
- Its function prototype is -

  *void *malloc(size_t size);* which allocates *size* bytes of memory.

  - If the allocation succeeds, a pointer to the block of memory is returned, else a null pointer is returned.
  - malloc returns a void pointer (void *), which indicates that it is a pointer to a region of unknown data type so we have to cast the value to the type of the destination pointer.
  - Memory allocated via malloc is persistent: it will continue to exist until the program terminates or the memory is explicitly deallocated by the programmer.

Calloc-

- The calloc function is one of the functions in standard C to allocate memory with initialization.
- Its function prototype is -

  *void *calloc(size_t nelements, size_t bytes);* which allocates a region of memory large enough to hold nelements of size bytes each.
- The allocated region is initialized to zero.
- If the allocation succeeds, a pointer to the block of memory is returned, else a null pointer is returned.

Realloc-

- It changes the size of a block of memory already assigned to a pointer.
- Its function prototype is -

  *void * realloc (void * pointer, size_t size);*

  Pointer parameter receives a pointer to an already assigned memory block or a null pointer, and size specifies the new size that the memory block shall have.
- realloc behaves like malloc if the first argument is NULL:

```
void *p = malloc(42);
void *p = realloc(NULL, 42);        /* equivalent */
```

Free-

- It releases a block of dynamic memory previously assigned using malloc, calloc or realloc.
- Its function prototype is -

  *void free (void * pointer);*
- This function must only be used to release memory assigned with functions malloc, calloc and realloc.

Algorithm:

Structure declaration:

```
struct node {
        int data;
        struct node *link;

};
```

(XVI) Algorithm create (struct node *head)

Postcondition ***:*** *create new linked list.*

Return ***:*** *pointer to first node.*

1. *Allocate memory for the node current and scan data make link field as Null*
2. *If head == Null*

   head = current
7. *End if*
8. *Else traverse the list to get last node as first*

   Make first->link=current
9. *End else*
10. *Perform 1 to 5 for more node.*
11. *return head.*

(XVII) Algorithm display_list (struct node *head)

Precondition : Create linked list
Postcondition : All nodes that exist are displayed
Return : Nil

1. if (head==Null)

   Display list is empty

   End if

7. Else while head!=Null

   Display nodes using head->data

8. End else

(XVIII) Algorithm insert (struct node *head, int data)

Postcondition : Record with corresponding data is inserted either at first, in between or end
Return : Return pointer to first node

At front-

1. *Allocate memory for the node current and scan data make link field as Null*
2. *If head == Null*

   head = current

   End if

3. *Else make current->link=current and head=current*
4. *End else*
5. *return head.*

At the end-

1. *Allocate memory for the node current and scan data make link field as Null*
2. *If head == Null*

head = current

End if

3. *Else traverse list while last node as first then make first->link=current*
4. *End else*
5. *Return head.*

At the middle

1. *Allocate memory for the node current and scan data make link field as Null*
2. *Scan after which node want to insert*
3. *Find that node as first into the list*
    3. *Then make current->link= first->link and first->link=current*
    4. *Return head.*

(XIX) Algorithm delete (struct node *head)

Precondition : Create list of node.
Postcondition : Node with corresponding data is deleted from the linked list
Return : Return pointer to first node

10. *Search for node into the list as first*
11. *Search for its previous node as prev*
12. *Then make prev->link = first->link*
13. *Free first*
14. *Return head*

(XX) Algorithm display_reverse(struct node * head)

Precondition : Create list of nodes.
Postcondition : Display liat in reverse order
Return : Nil

3. *While head not equal to null*
4. *Call same function*
5. *Print data*
6. *End*

(XXI) Algorithm revert (struct node *head)

Precondition : Create list of nodes.
Postcondition : Revert the list
Return : Pointer to first node.

1. *Initialize pointers as*

   temp=null;

   result=null;

   current=head;

2. *While current!=Null make temp=current->next; current->next=result;*

   result=current; and current=temp.

3. *Then make head=result;*
4. *Return head.*

Input Output and Test Cases:

Enter No of nodes to insert: 3

Enter Data-1

Enter Data-2

Enter Data-3

Linked list- 1 2 3

Insert Menu

1. At front

2. In the middle

3. At end

4. Return

Enter choice-1

Insert At front

Enter data-0

Linked list- 0 1 2 3

Insert At end

Enter data-4

Linked list- 0 1 2 3 4

Insert In the middle

Enter data- 5

After which node want to insert-2

Linked list- 0 1 2 5 3 4

Delete Menu

1. At front

2. In the middle

3. At end

4. Return

Enter choice-1

Linked list- 0 1 2 5 3 4

Delete At front

Linked list- 1 2 5 3 4

Delete At end

Linked list- 1 2 5 3

Delete In the middle

Linked list- 1 2 5 3

Enter node want to delete-2

Linked list- 1 5 3

Display reverse

Original Linked list- 1 5 3

Reverse Linked list- 3 5 1

Revert Linked List

Original Linked list- 1 2 5 3

Reverse Linked list- 3 5 2 1

Display Menu

Linked List- 3 5 2 1

Test Cases:

- When the list is empty, for delete option, display reverse, revert option, and display option, display message “The linked list is empty”
- When user trying to delete number not in the list, display “Data not found!”.
- If user is trying to select option not in the list, display message, “wrong Option!”

Application:

This assignment is helpful in implementing other data structures.

FAQs :

- What is singly linked list?
- What are applications of linked list?
- How to allocate dynamic memory for node.
- How to implement other data structure using linked list

Instructions for writing journal:

- Title
- Problem Definition
- Concept of linked list
- Pseudo C code for create, insert, delete, display reverse and revert
- Analysis of above for time and space complexity
- Program code
- Output for different I/P comparing time complexity

- Conclusion

Assignment No - 10

Title: Operations on single variable polynomial using circular linked lists.

Problem Statement: Implement polynomial using CLL and perform

a) Addition of Polynomials

b) Multiplication of polynomials

c) Evaluation of polynomial.

Objective:

- To understand the concept of structures.
- To understand the concept of Linked list.
- To understand the concept of Array.
- To understand the concept of circular Linked List

Outcome:

S/W Packages and H/W apparatus used:

Linux OS: Ubantu/Fedora , Eclipse/codeblocks,

PC with the configuration as Pentium IV 1.7 GHz. 128M.B RAM, 40 G.B HDD,

15''Color Monitor, Keyboard, Mouse

References: 1. 'C programming' by Balguruswami.

2. ' C' Kanetkar.

3. 'Code complete' Steve Connel.

Theory:

Linked list:

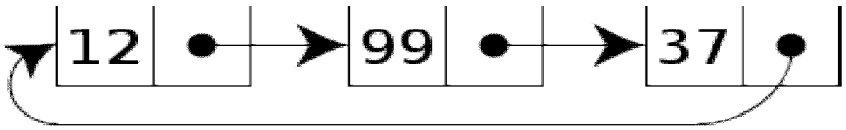
- Linked list is one of the fundamental data structure, and can be used to implement other data structures.

- It consists of a sequence of nodes, each containing arbitrary data fields and one or two references ("links") pointing to the next and/or previous nodes

- Head pointer to the first node
- The last node points to Head

Singly Circular Linked List:

- The simplest kind of linked list is a singly-linked list which has one link per node.
- This link points to the next node in the list, or to a null value or empty list if it is the final node.
- A singly linked list's node is divided into two parts. The first part holds or points to information about the node, and second part holds the address of next node.
- A singly linked list travels one way.



Memory allocation functions:

Malloc-

- The malloc function is one of the functions in standard C to allocate memory without initialization.
- Its function prototype is -

  *void *malloc(size_t size);* which allocates *size* bytes of memory.

  - If the allocation succeeds, a pointer to the block of memory is returned, else a null pointer is returned.

- malloc returns a void pointer (void *), which indicates that it is a pointer to a region of unknown data type so we have to cast the value to the type of the destination pointer.
- Memory allocated via malloc is persistent: it will continue to exist until the program terminates or the memory is explicitly deallocated by the programmer.

Calloc-

- The calloc function is one of the functions in standard C to allocate memory with initialization.
- Its function prototype is -

  *void *calloc(size_t nelements, size_t bytes);* which allocates a region of memory large enough to hold nelements of size bytes each.
- The allocated region is initialized to zero.
- If the allocation succeeds, a pointer to the block of memory is returned, else a null pointer is returned.

Realloc-

- It changes the size of a block of memory already assigned to a pointer.
- Its function prototype is -

  *void * realloc (void * pointer, size_t size);*

  Pointer parameter receives a pointer to an already assigned memory block or a null pointer, and size specifies the new size that the memory block shall have.
- realloc behaves like malloc if the first argument is NULL:

```
void *p = malloc(42);
void *p = realloc(NULL, 42);        /* equivalent */
```

Free-

- It releases a block of dynamic memory previously assigned using malloc, calloc or realloc.
- Its function prototype is -

  *void free (void * pointer);*

- This function must only be used to release memory assigned with functions malloc, calloc and realloc.

Algorithm:

Structure declaration:

```
struct node {
        int pow;
       float coeff;
        struct node *link;

};
```

(I) Algorithm create (struct node *head)

Postcondition ***:*** *create new linked list.*

Return ***:*** *pointer to first node.*

1. *Allocate memory for the node current and scan data make link field as Null*
2. *If head == Null*

   head = current

   End if

12. *Else traverse the list to get last node as first*

Make first->link=current

End else

13. *Perform 1 to 5 for more node.*
14. *return head.*

(II) Algorithm display_list (struct node *head)

Precondition : Create linked list
Postcondition : All nodes that exist are displayed
Return : Nil

1. *if (head==Null)*

Display list is empty

End if

9. *Else while temp!=head*

Display nodes using temp ->coeff

Display nodes using temp ->pow

10. *return*

(III) Algorithm insert (struct node *head)

Postcondition : Record with corresponding data is inserted either at first, in between or end
Return : Return pointer to first node

3. *Allocate memory for the node current and scan data make link field as Null*
4. *If head == Null*

head = current

head->coef=coe;

head->pow=power;

head->next=head;

Else traverse list with first node as p while(p->next != head)

p=p → next;

5. *current->coef=coe;*
   current ->pow=power;
   p ->next=current;
   current ->next=head;

6. *Return head.*

Algorithm Addpoly (struct node *head1, struct node *head2, struct node *result)

Precondition ***:*** *Create list of node.*
Postcondition ***:*** *Node with ploy1 & poly2 are added*
Return ***:*** *Return pointer to resultant node*

15. *Repeat thru step 2 while (poly1 < > head1 AND poly2 < > head2)*
16. *if(poly1->pow > poly2->pow)*
    add poly1 terms to result;
    increment poly1;
    end if
    if(poly1->pow < poly2->pow)
    add poly2 terms to result;
    increment poly2;
    end if
    else
    add term with poly1->coef+poly2->coef and poly1->pow to result;
    increment poly1;
    increment poly2;
    end if.
17. *while(poly1 != head1)*

add remaining poly1 terms to result;

18. while(poly2 != NULL)

add remaining poly2 terms to result;

19. return

(IV) Algorithm Mulply (struct node *head1, struct node *head2, struct node *result)

Precondition ***:*** *Create list of node.*
Postcondition ***:*** *Node with ploy1 & poly2 are added*
Return ***:*** *Return pointer to resultant node*

1. *Allocate memory dynamically for result*
2. *Repeat thru step 2 while (poly1 < > head1)*
3. *Repeat while (poly2 < > head2)*

```
result->coe=poly1->coef*poly2->coef;
result->pow = poly1->pow+ poly2->pow;
insert_node(result);
```

4. *Return*

(V) Algorithm Eval (struct node *head1,int x)

Precondition ***:*** *Create list of node.*
Postcondition ***:*** *Node with sum as result after evaluation*
Return ***:*** *Return evaluated answer*

1. *poly1=head1*
2. *Repeat while (poly1 < > head1)*

```
sum= sum + poly1->coef * power(x, ploy1->exp);
poly1=poly1->next;
```

3. *Return sum*

Input Output and Test Cases:

Input :

Enter sample input here=>

Enter first polynomial :3x^2+2x+5

Enter second polynomial :8x^4+10

Output :

Show sample output for sample input here=>

Addition :8x^4+3x^2+2x+15

Multiplication :24x^6+16x^5+40x^4+30x^2+20x+50

Evaluation :3x^2+2x+5

Enter value of x :2

Answer : 21

Application:

This assignment is helpful in implementing other data structures.

FAQs :

- What is singly circular linked list?
- What are applications of CLL?
- How to allocate dynamic memory for node.
- How to implement other data structure using linked list

Instructions for writing journal:

- Title
- Problem Definition
- Concept of linked list
- Pseudo C code for create, insert, delete, display reverse and revert
- Analysis of above for time and space complexity
- Program code
- Output for different I/P comparing time complexity
- Conclusion

Assignment No - 11

Title: Doubly linked list

Problem Statement:

Accept input as a string and construct a Doubly Linked List for the input string with each node contains, as a data one character from the string and perform:

d) Insert

e) Delete,

f) Display forward

g) Display backward.

Objective:

- To understand the concept of doubly linked list.
- To understand the concept of dynamic memory allocation.
- To understand how to use single linked list to implement other data structures.

Outcome:-

S/W Packages and H/W apparatus used:

Linux OS: Ubantu/Fedora , Eclipse/codeblocks,

PC with the configuration as Pentium IV 1.7 GHz. 128M.B RAM, 40 G.B HDD,

15''Color Monitor, Keyboard, Mouse

References: 1. 'C programming' by Balguruswami.

2. ' C' Kanetkar.

3. 'Code complete' Steve Connel.

Theory:

Linked list:

- Linked list is one of the fundamental data structure, and can be used to implement other data structures.
- It consists of a sequence of nodes, each containing arbitrary data fields and one or two references ("links") pointing to the next and/or previous nodes

- Head pointer to the first node
- The last node points to NULL

Doubly Linked List:

- Also called as two-way linked list.
- Each node has two links: one points to the previous node, or points to a null value or empty list if it is the first node; and one points to the next, or points to a null value or empty list if it is the final node.
- Convenient to traverse lists backwards.

Memory allocation functions:

Malloc-

- The malloc function is one of the functions in standard C to allocate memory without initialization.

- Its function prototype is -

  *void *malloc(size_t size);* which allocates *size* bytes of memory.

  - If the allocation succeeds, a pointer to the block of memory is returned, else a null pointer is returned.
  - malloc returns a void pointer (void *), which indicates that it is a pointer to a region of unknown data type so we have to cast the value to the type of the destination pointer.
  - Memory allocated via malloc is persistent: it will continue to exist until the program terminates or the memory is explicitly deallocated by the programmer.

Calloc-

- The calloc function is one of the functions in standard C to allocate memory with initialization.
- Its function prototype is -

  *void *calloc(size_t nelements, size_t bytes);* which allocates a region of memory large enough to hold nelements of size bytes each.
- The allocated region is initialized to zero.
- If the allocation succeeds, a pointer to the block of memory is returned, else a null pointer is returned.

Realloc-

- It changes the size of a block of memory already assigned to a pointer.
- Its function prototype is -

  *void * realloc (void * pointer, size_t size);*

  Pointer parameter receives a pointer to an already assigned memory block or a null pointer, and size specifies the new size that the memory block shall have.
- realloc behaves like malloc if the first argument is NULL:

```
void *p = malloc(42);
void *p = realloc(NULL, 42);        /* equivalent */
```

Free-

- It releases a block of dynamic memory previously assigned using malloc, calloc or realloc.
- Its function prototype is -

  *void free (void * pointer);*
- This function must only be used to release memory assigned with functions malloc, calloc and realloc.

Algorithm:

Structure declaration:

```
struct node {
        char data;
        struct node *next;
        struct node *prev;

};
```

(I) Algorithm create (struct node *head)

Postcondition : create new linked list.
Return : Pointer to first node.

1. *Accept string*
2. *Allocate memory for the node current and data as one character make both link field as Null*
3. *If head == Null*

   head = current

   End if

   Else traverse the list to get last node as first

   Make first->next=current and current->prev=first

15. *End else*
16. *Perform 1 to 5 for all characters from string.*

17. Return head.

(II) Algorithm display_list_forward (struct node *head)

Precondition : Create linked list
Postcondition : All nodes that exist are displayed
Return : Nil

1. *if (head==Null)*

Display list is empty

11. End if

12. Else while head!=Null

Display nodes using head->data

13. End else

(III) Algorithm insert (struct node *head, int data)

Postcondition : Record with corresponding data is inserted either at first, in between or end
Return : Return pointer to first node

At front-

1. *Allocate memory for the node current and scan data make both link field as Null*
2. *If head == Null*

head = current

End if

3. *Else make current->next=head and head->prev=current and head=current*
4. *End else*
5. *return head.*

At the end-

1. *Allocate memory for the node current and scan data make both link field as Null*
2. *If head == Null*

i. *head = current*

ii. *End if*

3. *Else traverse list while last node as first then make first->next=current and current->prev=first*

4. *End else*
5. *Return head.*

At the middle

1. *Allocate memory for the node current and scan data make link field as Null*
2. *Scan after which node want to insert*
3. *Find that node as first into the list*
4. *Then make current->next= first->next, first->next->prev=current,*
    i. *current->prev=first and first->next=current*
5. *Return head.*

(IV) Algorithm delete (struct node *head)

Precondition : Create list of node.
Postcondition : Node with corresponding data is deleted from the linked list
Return : Return pointer to first node

1. *Search for node into the list as first*
2. *Then make first->prev->next = first->next and first->next->prev=first->prev*
3. *Free first*
4. *Return head*

(V) Algorithm display_backword(struct node * head)

Precondition : Create list of nodes.
Postcondition : Display list in reverse order
Return : Nil

1. Traverse the list while end of list using next pointer
2. Let last node be first
3. Traverse list using prev and print data while first->prev not equal to null
4. End

Input Output and Test Cases:

Enter character string to create list - PICT

Linked list- P I C T

Menu

1. Insert Character

2. Delete Character

3. Display Forward

4. Display Backward

5. Exit

Enter your choice- 1

Insert Menu

1. At front

2. In the middle

3. At end

4. Return

Enter choice-1

Insert At front:

Enter data- S

Linked list- S P I C T

Insert in the middle:

Before Insert: Linked list- S P I C T

Enter character after which you want to insert: I

Enter new character: J

After Insert: Linked list- S P I J C T

Insert at end:

Before Insert: Linked list- S P I J C T

Enter new character: M

After Insert: Linked list- S P I J C T M

Display Forward:

Linked list- S P I C T

Delete Menu

1. At front

2. In the middle

3. At end

4. Return

Enter choice-1

Delete At front

Before Delete: Linked list- S P I C T

After Delete: Linked list- P I C T

Delete in the middle

Before Delete: Linked list- P I C T

Enter Data to delete-C

After Delete: Linked list- P I T

Delete at end

Before Delete: Linked list- P I T

After Delete: Linked list- P I

Display backward :

Linked list- I P

Test Cases:

- When the list is empty, for delete option, display forward option, and display backward option display message "The linked list is empty"
- When user trying to delete character not in the list, display "Character not found!".
- If user is trying to select option not in the list, display message, "wrong Option!"

Application:

This assignment is helpful in implementing other data structures and applications. Linked lists are used as a building block for many other data structures, such as stacks, queues and their variations. Linked list is also used for implementation of nonlinear data structure such as tree.

FAQs :

- What is doubly linked list?
- What are applications of doubly linked list?
- How to allocate dynamic memory for node.
- How to implement other data structure using linked list

Instructions for writing journal:

- Title
- Problem Definition
- Concept of doubly linked list
- Pseudo C code for create, insert, delete, display reverse and revert
- Analysis of above for time and space complexity
- Program code
- Output for different I/P comparing time complexity
- Conclusion

Assignment No - 12

Title: Generalized linked list

Problem Statement:

Implement Generalized Linked List to create and display the book index.

Objective:

- To understand the concept of doubly linked list.
- To understand the concept of dynamic memory allocation.
- To understand how to use single linked list to implement other data structures.

S/W Packages and H/W apparatus used:

Linux OS: Ubantu/Fedora , Eclipse/codeblocks,

PC with the configuration as Pentium IV 1.7 GHz. 128M.B RAM, 40 G.B HDD,

15''Color Monitor, Keyboard, Mouse

References:

1. 'C programming' by Balguruswami.
2. ' C' Kanetkar.
3. 'Code complete' Steve Connel.

Theory:

Generalized Linked List:
A Linear list ,we define as finite sequence of n- elements $a_1,a_2….a_n$, and we represent as $A=(a_1,a_2,…,a_n)$. Here we noticed that each element a belongs to A has a unique structural property having single position i.e a_1+1 succeeds a1 and each a_1 is an atom. Many applications do need that there should be a separate structural property of their own. To facilitate this,we go in for generalized linked list.

Definition:
"A generalized linked list "A" is a finite sequence of 'n' elements (a_1,a_2,…a_n) where a_i may be either atoms or list. The elements a_i which are not atoms are called sublist of A ".
Note that the elements if all are atoms then it turns out to be a simple list.
The node Structure for Generalized link list is as shown,

Tag=true/false	Data/link	Link

The link field is used as pointer to the list,while the "Data/link" field can hold an atom in case head (A) is an atom or be a pointer to the list representation of head(A) in case it is a list.The 'Tag' will denote link or data.

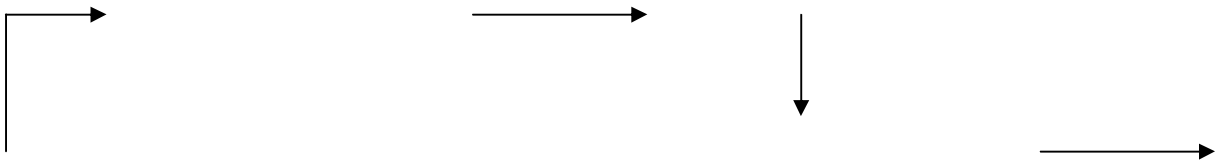
Polynomial Representation using GLL:
Suppose if we are having a polynomial with three variables then we may think of representing it by using simple list. Here we might think of having the node structure as:

Coef	Exp x	Exp y
Exp. z	Link	

But this would mean that polynomial in different number of variables would need a different number of variables would need a different number of fields, adding another difficulty to the sequential representation. Thus, these nodes would have to vary in size depending on the number of variables .It makes storage management difficult. Hence, we make use of G.L.L to represent the polynomial. The general idea is to factorize the given polynomial so that each element of polynomial either becomes an atom or an list itself.

Representations of Generalized Lists

Example 1: $3X^2Y$



Example 2: Representation of P(x, y, z)

Example 3: