DATA STRUCTURES – I -UNIT1 NOTES

STRUCTURED PROGRAMMING

A programmers task is to determine what a program is supposed to do and then write a good program that accomplishes the task. There are various programming techniques which help to write programs that are easier to understand ,to test, debug and to modify. The need for structured programming methodology becomes apparent when the program in not small or trivial or it needs later modification.

In the world of computer programming, structured programming is a logical construct that allows for the efficient operation of a program.

Structured programming is a technique for organizing and coding computer programs in which a hierarchy of modules is used, each having a single entry and a single exit point, and in which control is passed downward through the structure without unconditional branches to higher levels of the structure. Three types of control flow are used: sequential, test, and iteration. A style of programming usually associated with languages such as C, FORTRAN, and Pascal and so on. Using structured programming techniques, a problem is often solved using a divide and conquer approach such as stepwise refinement. An initially large problem is broken into several smaller sub-problems. Each of these is then progressively broken into even smaller sub-problems, until the level of difficulty is considered to be manageable. At the lowest level, a solution is implemented in terms of data structures and procedures. The structured program design follows the Top- down Approach.Top down design takes the divide and conquer strategy

The divide-and-conquer strategy consists in breaking a problem into simpler subproblems of the same type, next to solve these subproblems, finally to amalgamate the obtained results into a solution to the problem. Thus it is primarily a recursive method. The algorithms of this type display two parts; the first one breaks the problem into subproblems, the second one merges the partial results into the total result.

Examples where divide-and-conquer strategy is applied : MERGE SORT, QUICK SORT, BINARY SEARCH.

One of the advantages to the implementation of a structured model of programming is the ability to either eliminate or at least reduce the necessity of employing the GOTO statement.

INTRODUCTION TO DATA STRUCTURES

1. In computer science, a Data Structure is a way of storing data in a computer so that it can be used efficiently.
2. Often a carefully chosen data structure will allow the most efficient algorithm to be used.
3. The choice of the data structure often begins from the choice of an abstract data type.
4. A well-designed data structure allows a variety of critical operations to be performed, using as few resources, both execution time and memory space, as possible.
5. Data structures are implemented using the data types, references and operations on them provided by a programming language.

Different kinds of data structures are suited to different kinds of applications, and some are highly specialized to certain tasks. For example, B-trees are particularly well-suited for implementation of databases.

In the design of many types of programs, the choice of data structures is a primary design consideration, as experience in building large systems has shown that the difficulty of implementation and the quality and performance of the final result depends heavily on choosing the best data structure. After the data structures are chosen, the algorithms to be used often become relatively obvious.

Since data structures are so crucial, many of them are included in standard libraries of modern programming languages and environments, such as C++'s Standard Template Library containers, the Java Collections Framework, and the Microsoft .NET Framework.

The fundamental building blocks of most data structures are arrays, records, discriminated unions, and references. For example, the nullable reference, a reference which can be null, is a combination of references and discriminated unions, and the simplest linked data structure, the linked list, is built from records and nullable references.

NEED OF DATA STRUCTURES:

Even though computers can perform literally millions of mathematical computations per second, when a problem gets large and complicated, performance can nonetheless be an important consideration. One of the most crucial aspects to how quickly a problem can be solved is how the data is stored in memory.

To illustrate this point, consider going to the local library to find a book about a specific subject matter. Most likely, you will be able to use some kind of electronic reference or, in the worst case, a card catalog, to determine the title and author of the book you want. Since the books are typically shelved by category, and within each category sorted by author's name, it is a fairly straightforward and painless process to then physically select your book from the shelves.
Now, suppose instead you came to the library in search of a particular book, but instead of organized shelves, were greeted with large garbage bags lining both sides of the room, each arbitrarily filled with books that may or may not have anything to do with one another. It would take hours, or even days, to find the book you needed, a comparative eternity. This is how software runs when data is not stored in an efficient format appropriate to the application.

CLASSIFICATION OF DATA STRUCTURES:

Primitive Data Structure: Basic Structures directly operated upon by the Machine instructions. Eg: Integer, Floating Point number, Character Constants etc.

Non-Primitive Data Structure: Derived from primitive data Structures and emphasize on structuring of a group of a homogeneous or Heterogeneous Data items. Eg: Arrays, Lists, Files

PURPOSE OF DATA STRUCTURES:

1. Organization of data
2. Accessing Methods
3. Degree of associativity
4. Processing alternatives for information

VARIOUS OPERATIONS ON DATA STRUCTURES:

1. Insertion
2. Deletion
3. Copying
4. Concatenation
5. Sorting
6. Searching
7. Traversing

THE AREAS IN WHICH DATA STRUCTURES ARE APPLIED EXTENSIVELY

- Compiler Design,
- Operating System,
- Database Management System,
- Statistical analysis package,
- Numerical Analysis,
- Graphics,
- Artificial Intelligence, Simulation

SIMPLE DATA STRUCTURES:
The simplest data structures are primitive variables. They hold a single value, and beyond that, are of limited use. When many related values need to be stored, an array is used.

A somewhat more difficult concept, though equally primitive, are pointers. Pointers, instead of holding an actual value, simply hold a memory address that, in theory, contains some useful piece of data. Pointers "point" somewhere in memory, and do not actually store data themselves.

A less abstract way to think about pointers is in how the human mind remembers (or cannot remember) certain things. Many times, a good engineer may not necessarily know a particular formula/constant/equation, but when asked, they could tell you exactly which reference to check.

Arrays:
Arrays are a very simple data structure, and may be thought of as a list of a fixed length. Arrays are nice because of their simplicity, and are well suited for situations where the number of data

items is known (or can be programmatically determined). Suppose you need a piece of code to calculate the average of several numbers.

An array is a perfect data structure to hold the individual values, since they have no specific order, and the required computations do not require any special handling other than to iterate through all of the values.

The other big strength of arrays is that they can be accessed randomly, by index. For instance, if you have an array containing a list of names of students seated in a classroom, where each seat is numbered 1 through n, then studentName[i] is a trivial way to read or store the name of the student in seat i.

An array might also be thought of as a pre-bound pad of paper. It has a fixed number of pages, each page holds information, and is in a predefined location that never changes.

ARRAY OPERATIONS

DELETION FROM AN ARRAY

```
#include<stdio.h>

#include<conio.h>

int  I, n;

main()

{

int a[100],pos;

void delete(int a[],int,int);

clrscr();

printf("How many elements in the array\n");

scanf("%d",&n);

printf("Enter the element of the array\n");

for(i=0;i<=n-1;i++)

scanf("%d",&a[i]);

printf("On which postion element do you want delete\n");

scanf("%d",&pos);

delete(a,pos,n);

getch();

}


void delete(int a[],int pos,int n)

{

int j,item;

item=a[pos];

for(j=pos;j<n-1;j++)

{

a[j]=a[j+1];
```

```
}
n=n-1;
for(i=0;i<=n-1;i++)
printf("%d\n",a[i]);
}
```

TRAVERSING AN ARRAY

```
#include<stdio.h>
#include<conio.h>
void main()
{
int n,x,i,a[100];
clrscr();
printf("Enter the length of the array");
scanf("%d",&n);
  for(i=0;i<=n;i++)
    scanf("%d",&a[i]);
printf("Traversing of the array\n");
 for(i=0;i<=n;i++)
   printf("%d\n",a[i]);
getch();
}
```

INSERTION IN AN ARRAY

```
#include<stdio.h>
#include<conio.h>
int i,len,pos,num;
void main()
{
int a[100];
void insert(int a[], int, int, int);
clrscr();
printf("Enter integers to be read");
scanf("%d",&len);
printf("Enter integers");
  for(i=0;i<=len;i++)
    {
```

```
        scanf("%d",&a[i]);
    }
printf("Enter integer to be inserted");
scanf("%d",&num);
printf("Enter position in the array for insertion");
scanf("%d",&pos);
--pos;
insert(a,len,pos,num);
}
void insert (int a[], int len, int pos, int num)
{
   for(i=len; i>=pos; i--)
    {
        a[i+1]=a[i];
    }
        a[pos]=num;
         if(pos>len)
           {
                printf("insertion outside the array");
           }
        len++;
        printf("New array");
for(i=0;i<len;i++)
{
printf("%d\n",a[i]);
}
getch();
}
```

Searching

Linear Search
Linear search is one of the searching methods for a list of elements. The linear search performs each element is examined in run time in sequence manner until the element find from the list or end of the list. The linear search is best for unordered list of elements. If the list ordered, we can use different search algorithm for efficient search. The linear search best O (1), at worst O(n), and on average O(n/2).

Algorithm linear search
A is an 1-D array having n elements. This algorithm searches the required element KEY using linear search method.
step 1 : Repeat for i= 1 to N
step2 :

If (A[i] = Key) then return i

Next I
Return –1 for not found

Implementation in C:

```
#include< stdio.h>
#define MAX 100
int Linear_Search(int [],int,int);
void main()
{
int total_number=0,key=0,i,arr[MAX],find=0;
printf("Enter the total number of elements:");
scanf("%d",&total_number);
for(i=0;i< total_number;i++)
scanf("%d",&arr[i]);
printf("Enter Key for find:");
scanf("%d",&key);
if( (find  = Linear_Search(arr,total_number,key)) >0)
printf("The element was found in %d location\n", find+1);
else
printf("The element was not found\n");
}

int Linear_Search(int a[],int n, int k)
{
int i=0;
for(i=0;i< n;i++)
{
if(a[i] == k)
return i;
}
return -1;
}
```

Sample run:

Enter the total number of elements: 8

12 23 67 5 0 1 89 7

Enter Key for find: 89

The element was found in 7 location

Binary search

To Search an element in an array using Binary Search

The most common application of binary search is to find a specific value in a sorted list. The binary search begins by comparing the sought value *X* to the value in the middle of the list; because the values are sorted, it is clear whether the sought value would belong before or after that middle value, and the search then continues through the correct half in the same way. binary search is an example of a divide and conquer algorithm.

Algorithm :

```
BinarySearch(A[0..N-1], value)
      1. low = 0
        high = N - 1
      2.while (low <= high)
           mid = (low + high) / 2
           1.if (A[mid] > value)
               high = mid - 1
           2.else if (A[mid] < value)
               low = mid + 1
           3.else
               return mid
        end while
      3. return not_found
```

SORTING

Sorting is a process of arranging elements in Ascending or Descending order. The common sorting algorithms can be divided into two classes by the complexity of their algorithms. Algorithmic complexity is generally written in a form known as Big-O notation, where the *O* represents the complexity of the algorithm and a value *n* represents the size of the set the algorithm is run against.

For example, $O(n)$ means that an algorithm has a linear complexity. In other words, it takes ten times longer to operate on a set of 100 items than it does on a set of 10 items (10 * 10 = 100). If the complexity was $O(n^2)$ (quadratic complexity), then it would take 100 times longer to operate on a set of 100 items than it does on a set of 10 items.

The two classes of sorting algorithms are $O(n^2)$, which includes the bubble, insertion, selection, and shell sorts; and $O(n \log n)$ which includes the heap, merge, and quick sorts.

Bubble Sort

Bubble Sort works by comparing each element of the list with the element next to it and swapping them if required. With each pass, the largest of the list is "bubbled" to the end of the list whereas the smaller values sink to the bottom.

Algorithm for BUBBLESORT

BUBBLESORT (A)
for $i \leftarrow 1$ to length $[A]$ do
  for $j \leftarrow$ length $[A]$ downto $i + 1$ do
    If $A[A] < A[j\text{-}1]$ then
      Exchange $A[j] \leftrightarrow A[j\text{-}1]$

Examine the following table. (Note that each pass represents the status of the array after the completion of the inner for loop, except for pass 0, which represents the array as it was passed to the function for sorting)

```
8 6 10 3 1 2 5 4 } pass 0
6 8 3 1 2 5 4 10 } pass 1
6 3 1 2 5 4 8 10 } pass 2
3 1 2 5 4 6 8 10 } pass 3
1 2 3 4 5 6 8 10 } pass 4
1 2 3 4 5 6 8 10 } pass 5
1 2 3 4 5 6 8 10 } pass 6
1 2 3 4 5 6 8 10 } pass 7
```

Sample code:

```
void BubbleSort(int a[], int array_size)
{
     int i, j, temp;
     for (i = 0; i < (array_size - 1); ++i)
     {
          for (j = 0; j < array_size - 1 - i; ++j )
          {
               if (a[j] > a[j+1])
               {
                    temp = a[j+1];
                    a[j+1] = a[j];
                    a[j] = temp;
               }
          }
     }
}
```

A single, complete "bubble step" is the step in which a maximum element is bubbled to its correct position. This is handled by the inner for loop.

```
for (j = 0; j < array_size - 1 - i; ++j )
{
     if (a[j] > a[j+1])
     {
          temp = a[j+1];
          a[j+1] = a[j];
          a[j] = temp;
     }
}
```

Selection Sort

In Selection Sort first find the smallest in the array and exchange it with the element in the first position, then find the second smallest element and exchange it with the element in the second position, and continue in this way until the entire array is sorted. Selection sort is among the simplest of sorting techniques and it work very well for small files. Furthermore, despite its evident "naïve approach "Selection sort has a quite important application because each item is actually moved at most once, Section sort is a method of choice for sorting files with very large objects (records) and small keys.

SELECTION_SORT (A)

```
for i ← 1 to n-1 do
   min j ← i;
   min x ← A[i]
   for j ← i + 1 to n do
      If A[j] < min x then
                min j ← j
         min x ← A[j]
   A[min j] ← A [i]
   A[i] ← min x
```

Consider the following table. (Note that each pass represents the status of the array after the completion of the inner for loop, except for pass 0, which represents the array as it was passed to the function for sorting)

```
8 6 10 3 1 2 5 4 } pass 0
1 6 10 3 8 2 5 4 } pass 1
1 2 10 3 8 6 5 4 } pass 2
1 2 3 10 8 6 5 4 } pass 3
1 2 3 4 8 6 5 10 } pass 4
1 2 3 4 5 6 8 10 } pass 5
1 2 3 4 5 6 8 10 } pass 6
1 2 3 4 5 6 8 10 } pass 7
```

At pass 0, the list is unordered. Following that is pass 1, in which the minimum element 1 is selected and swapped with the element 8, at the lowest index 0. In pass 2, however, only the sublist is considered, excluding the element 1. So element 2, is swapped with element 6, in the 2nd lowest index position. This process continues till the sub list is narrowed down to just one element at the highest index (which is its right position).

Insertion Sort

If the first few objects are already sorted, an unsorted object can be inserted in the sorted set in proper place. This is called insertion sort. An algorithm consider the elements one at a time, inserting each in its suitable place among those already considered (keeping them sorted). Insertion sort is an example of an incremental algorithm; it builds the sorted sequence one number at a time.

INSERTION_SORT (A)

```
1. For j = 2 to length [A] do
2.     key = A[j]
3.     {Put A[j] into the sorted sequence A[1 . . j-1]
4.     i ← j -1
5.     while i > 0 and A[i] > key do
6.         A[i+1] = A[i]
7.             i = i-1
8.     A[i+1] = key
```

Examine the following table. (Note that each pass represents the status of the array after the completion of the inner for loop, except for pass 0, which represents the array as it was passed to the function for sorting)

```
8 6 10 3 1 2 5 4 } pass 0
6 8 10 3 1 2 5 4 } pass 1
6 8 10 3 1 2 5 4 } pass 2
3 6 8 10 1 2 5 4 } pass 3
1 3 6 8 10 2 5 4 } pass 4
1 2 3 6 8 10 5 4 } pass 5
1 2 3 5 6 8 10 4 } pass 6
1 2 3 4 5 6 8 10 } pass 7
```

The pass 0 is only to show the state of the unsorted array before it is given to the loop for sorting.

QUICK SORT

Quicksort is a *divide-and-conquer* style algorithm. A divide-and-conquer algorithm solves a given problem by splitting it into two or more smaller subproblems, recursively solving each of the subproblems, and then combining the solutions to the smaller problems to obtain a solution to the original one.. Typically, quicksort is significantly faster in practice than other $\Theta(n \log n)$ algorithms, because its inner loop can be efficiently implemented on most architectures

As a first step, Quick Sort chooses as pivot one of the items in the array to be sorted. Then array is then partitioned on either side of the pivot. Elements that are less than or equal to pivot will move toward the left and elements that are greater than or equal to pivot will move toward the right.

Algorithm:

partition(*A*, *p*, *r*)

1. $x \leftarrow A[p]$
2. $i \leftarrow p-1$
3. $j \leftarrow r+1$
4. while TRUE do
5. Repeat $j \leftarrow j-1$
6. until $A[j] \leq x$
7. Repeat $i \leftarrow i+1$
8. until $A[i] \geq x$
9. if $i < j$
10. then exchange $A[I] \leftrightarrow A[j]$
11. else return j

Partition selects the first key, $A[p]$ as a pivot key about which the array will partitioned:

Keys $\leq A[p]$ will be moved towards the left .
Keys $\geq A[p]$ will be moved towards the right

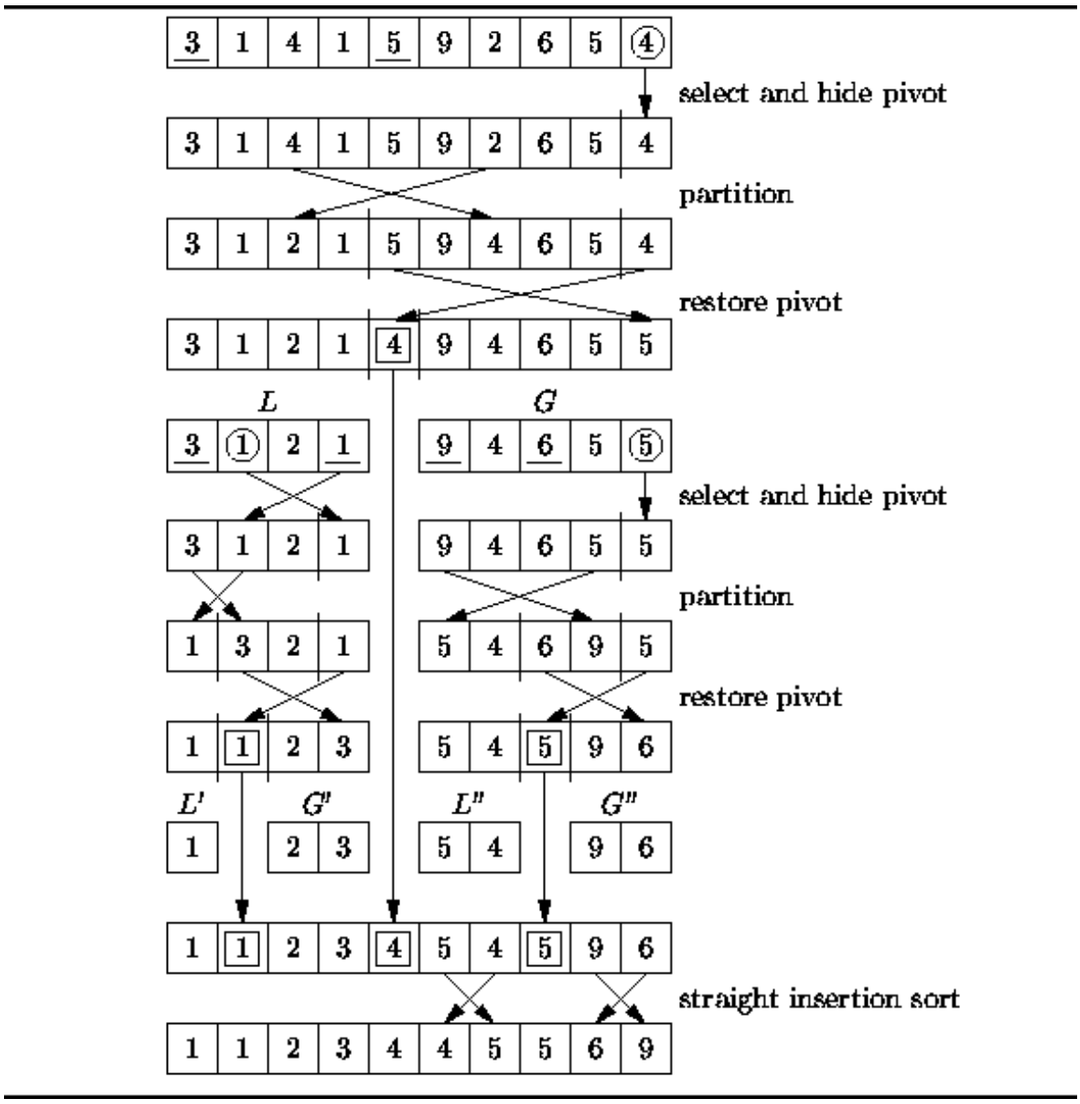

Figure: ``Quick'' sorting.

After the partitioning, the pivot is inserted between the two sequences. This is called *restoring* the pivot. To restore the pivot, we simply exchange it with the first element of *G*. Notice that the 4 is in its correct position in the sorted sequence and it is not considered any further.

Now the quicksort algorithm calls itself recursively, first to sort the sequence $L = \{3, 1, 2, 1\}$; second to sort the sequence $G = \{9, 4, 6, 5, 5\}$. The quicksort of *L* selects 1 as the pivot, and creates the two subsequences $L' = \{1\}$ and $G' = \{2, 3\}$. Similarly, the quicksort of *G* uses 5 as the pivot and creates the two subsequences $L'' = \{5, 4\}$ and $G'' = \{9, 6\}$.

program in C for MERGE SORT

```
mergesort (int a[], int low, int high)
{
int mid;
if(low<high)
{
mid=(low+high)/2;
mergesort(a,low,mid);
mergesort(a,mid+1,high);
merge(a,low,high,mid);
}
return(0);
}

merge(int a[], int low, int high, int mid)
{
int i, j, k, c[50];
i=low;
j=mid+1;
k=low;
while((i<=mid)&&(j<=high))
{
if(a[i]<a[j])
{
c[k]=a[i];
k++;
i++;
}
else
{
c[k]=a[j];
k++;
j++;
}
}
while(i<=mid)
{
c[k]=a[i];
k++;
i++;
}
while(j<=high)
{
c[k]=a[j];
k++;
j++;
}
for(i=low;i<k;i++)
{
a[i]=c[i];
}
}
```

Example: 3 5 2 6 8

MergeSort uses divide & conquer strategy to sort the elements.
Here, you go on dividing till only one element is left.
Then this element will be merged with the next one and both will be put in correct order and since this is a recursive function, it will be repeated till you complete the merge and come out.

So, here is the sequence of calls:

a[] = {3, 5, 2, 6, 8}
low = 0; high = 4;
mergesort(a, 0, 4)
mid = 0 + 4 / 2 = 2
mergesort(a, 0, 2)
mergesort(a, 3, 4)
merge(a, 0, 4, 2)

So, each of the mergesort() calls will again go back and call mergesort() and merge(), this goes on till: (low < high) is satisfied.

```
#include<stdio.h>

#include<conio.h>

void main()

{

int p,q,m,n,c;

int array1[100],array2[100],array3[200];

clrscr();

puts("Enter number of element of the first sorted array");

scanf("%d",&p);

puts("Enter element of the first sorted array");

for(m=0;m<=p;m++)

scanf("%d",&array1[m]);

puts("Enter number of element of the second sorted array");

scanf("%d",&q);

puts("Enter element of the second array");

for(n=0;n<=q;n++)

scanf("%d",&array2[n]);

c=0;

m=0;

n=0;

while((m<q)&&(n<q))

{

if(array1[m]<=array2[n])

array3[c]=array1[m++];

else

array3[c]=array2[n++];

c++;

}

while(m<p)

{

array3[c]=array1[m];

c++;
```

```
m++;
}
while(n<q)
{
array3[c]=array2[n];
c++;
n++;
}
puts("merged array in ascending order");
for(m=0;m<=c;m++)
printf("%d",array3[m]);
getch();
}
```