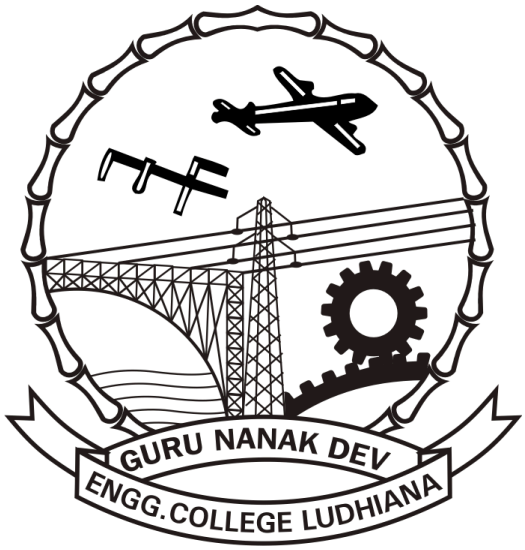
INSTRUCTION MANUAL

DATA STRUCTURES LAB

(BTCS-306)



Prepared by

Er. Gurjit Kaur
Assistant Professor (CSE)

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING
GURU NANAK DEV ENGINEERING COLLEGE
LUDHIANA – 141006

DECLARATION

This Manual of Data Structures Lab (BTCS-306) has been prepared by me as per syllabus of Data Structures lab (BTCS-306).

Signature

SYLLABUS

List of practical exercises, to be implemented using object-oriented approach in C++ Language.

1. Write a menu driven program that implements following operations (using separate functions) on a linear array:

Insert a new element at end as well as at a given position
Delete an element from a given whose value is given or whose position is given
To find the location of a given element
To display the elements of the linear array

2. Write a menu driven program that maintains a linear linked list whose elements are stored in on ascending order and implements the following operations (using separate functions):

Insert a new element
Delete an existing element
Search an element
Display all the elements

3. Write a program to demonstrate the use of stack (implemented using linear array) in converting arithmetic expression from infix notation to postfix notation.
4. Program to demonstrate the use of stack (implemented using linear linked lists) in evaluating arithmetic expression in postfix notation.
5. Program to demonstration the implementation of various operations on a linear queue represented usinga linear array.
6. Program to demonstration the implementation of various operations on a circular queue represented using a linear array.
7. Program to demonstration the implementation of various operations on a queue represented using a linear linked list (linked queue).
8. Program to illustrate the implementation of different operations on a binary search tree.
9. Program to illustrate the traversal of graph using breadth-first search.
10. Program to illustrate the traversal of graph using depth-first search.
11. Program to sort an array of integers in ascending order using bubble sort.
12. Program to sort an array of integers in ascending order using selection sort.
13. Program to sort an array of integers in ascending order using insertion sort.
14. Program to sort an array of integers in ascending order using radix sort.
15. Program to sort an array of integers in ascending order using merge sort.
16. Program to sort an array of integers in ascending order using quick sort.
17. Program to sort an array of integers in ascending order using heap sort.
18. Program to sort an array of integers in ascending order using shell sort.
19. Program to demonstrate the use of linear search to search a given element in an array.
20. Program to demonstrate the use of binary search to search a given element in a sorted array in ascending order.

INDEX

PRACTICAL 1

Title. Algorithm for Inserting an element in linear array

Let A be a collection of data elements in the memory of the computer. "Inserting" refers to the operation of adding element to the collection A.

Inserting an element at the "end" of a linear array can be done easily provided the memory space allocated for the array is large enough to accommodate the additional element. On the other hand, suppose we need to insert an element in the middle of the array. Then, on the average, half of the elements must be moved downward to new locations to accommodate the new elements and keep the order of the other elements.

ALGORITHM

(Inserting into a Linear Array) INSERT (LA, N, K, ITEM)

Here LA is a linear array with N elements and K is a positive integer such that K<= N. This algorithm inserts an element ITEM into the Kth position in LA.

Step 1: initialize counter set J=N
Step 2: repeat step 3 & 4 while J>K
Step 3: set LA(J+1)=LA(J)
Step 4: set J=J-1
Step 5: set LA(K)=ITEM
Step 6: N=N+1
Step 7:Exit

Title. Algorithm for Deleting an element from linear array

"Deleting " refers to the operation of removing one of the elements from A. Deleting an element at the end of an array presents no difficulties, but deleting an element somewhere in the middle of the array would require that each subsequent element be moved one location upward in order to " fill up" the array.

ALGORITHM

(Deleting from a Linear Array) DELETE(LA, N, K, ITEM)

Here LA is a linear array with N elements and K is a Positive integer such that K <= N. This algorithm deletes Kth element from LA.

Step 1:set ITEM=LA(K)
Step 2:repeat for J=K to N-1 set LA(J)=LA(J+1)

Step 3:set N=N-1
Step 4:Exit

Title. Algorithm to search the element from linear array

ALGORITHM

Linear search is the simplest algorithm. It search every element from A[0] on until x is found or until the end of the array is reached.
Linear Search uses a loop to sequentially look through a set of data and return the first instance of the value being searched for. Before we start coding, let's take a look at the pseudo code for this algorithm:

ALGORITHM
Step 1.set INDEX to 0
Step 2.set POSITION to -1
Step 3.set FOUND to false
Step 4.while INDEX is less than the number of elements and the value hasn't been found
if the element in the subscript index is equal to value
set FOUND to true
set POSITION to INDEX
increment INDEX
return POSITION

PRACTICAL 2

Title. **Algorithm for inserting node after a given node list & at beginning of the list**

ALGORITHM

Step 1:[OVERFLOW?] if AVAIL=NULL then write OVERFLOW and EXIT
Step 2:Set NEW=AVAIL and AVAIL = Link[AVAIL]
Step 3:Set Info[NEW]=ITEM
Step 4:if LOC=NULL then
Step 5:Set Link[NEW]=START and START = NEW
Step 6:else Set Link[NEW]=Link[LOC] and Link[LOC]=NEW
Step 7:EXIT

Inserting at the beginning of the list requires a separate function. This requires updating firstNode.
function insertBeginning

(list LIST, Node NEW NODE)
{ // insert node before current first node
newNode.NEXT list.FIRSTNODE
list.FIRSTNODE NEWNODE
}
Similarly, we have functions for removing the node after a given node, and for removing a node from the beginning of the list. The diagram demonstrates the former. To find and remove a particular node, one must again keep track of the previous element.

Title. **Algorithm for deleting node from the list**

ALGORITHM
Step 1.Delete FIRST ( ):
Description. Here START is a pointer variable which contains the address of first node. ITEM is the value to be deleted.
Step 2. if (START == NULL) Then [Check whether list is empty]
Step 3. Print. Linked-List is empty.
Step 4. else
Step 5. PTR = START
Step 6. ITEM = START->Info [Assign INFO of first node to ITEM]
Step 7. START == START->Link [START now points to 2nd node]

Step 8. Delete PTR [Delete first node].
Step 9.Print. ITEM deleted
[End of if]
Step 10. Exit

Title. **Algorithm for searching element from the list**

The simplest case of a selection algorithm is finding the minimum (or maximum) element by iterating through the list, keeping track of the running minimum – the minimum so far – (or maximum) and can be seen as related to the selection sort. Conversely, the hardest case of a selection algorithm is finding the median, and this necessarily takes *n*/2 storage. Sequential search is the most common search used on linked list structures.

ALGORITHM

Step 1:insert item at the end of data.set data(N+1)=ITEM
Step 2:initialise counter set LOC=1
Step 3:search for ITEM. repeat while data (LOC)!=ITEM
set LOC =LOC+1
Step 4:(successful?). if LOC=N+1,set LOC=0
Step 5:Exit

PRACTICAL 3

Stack (data structure) and Queue

In computer science, a stack is a Temporary abstract data type and data structure based on the principle of LAST In FIRST Out (LifO). Stacks are used extensively at every level of a modern computer system. For example, a modern PC uses stacks at the architecture level, which are used in the basic design of an operating system for interrupt handling and operating system function calls. Among other uses, stacks are used to run a Java Virtual Machine, and the Java language itself has a class called "Stack", which can be used by the programmer. The stack is ubiquitous.

Abstract data type

As an abstract data type, the stack is a container of nodes and has two basic operations. push and pop. Push adds a given node to the top of the stack leaving previous nodes below. Pop removes and returns the current top node of the stack. A frequently used metaphor is the idea of a stack of plates in a spring loaded cafeteria stack. In such a stack, only the top plate is visible and accessible to the user, all other plates remain hidden. As new plates are added, each new plate becomes the top of the stack, hiding each plate below, pushing the stack of plates down. As the top plate is removed from the stack, they can be used, the plates pop back up, and second plate becomes the top of the stack. Two important principles are illustrated by this metaphor, the LAST In FIRST Out principle is one. The second is that the contents of the stack are hidden. Only the top plate is visible, so to see what is on the third plate, the first and second plates will have to be removed.

Operations

In modern computer languages, the stack is usually implemented with more operations than just "push" and "pop". The length of a stack can often be returned as a parameter. Another helper operation top (also known as peek and peak) can return the current top element of the stack without removing it from the stack.

This section gives pseudocode for adding or removing nodes from a stack, as well as the length and top functions. Throughout we will use null to refer to an end-of-list marker or sentinel value, which may be implemented in a number of ways using pointers.

Title . Algorithm for converting arithmetic expression from infix notation to postfix notation.

POSTFIX (Q, P)

Suppose Q is an arithmetic expression written in infix notation. This algorithm finds the equivalent postfix expression P.

ALGORITHM

Step1. Push "(" onto STACK and add '')'' to the end of Q.
Step2. Scan Q from left to right and repeat step 3 to 6 for each element of Q, until the STACK is empty.
Step3. if an operand is encountered, push it to STACK.
Step4. if a left parenthesis is encountered, push it to STACK.
Step5. if an operator X is encountered, then
a) Repeatedly pop from STACK and add to P each operator (Top of Stack) which has same precedence as or higher precedence than X.
b) Add X to STACK.
Step6. if a right parentheses is encountered, then;
a) Repeatedly pop from STACK and add to P each operator (top of stack) until a left parentheses is encountered.
b) Remove the left parentheses from stack [Do not add it to P]
Step7. Exit.

Title . Algorithm for evaluating arithmetic expression in postfix notation

ALGORITHM

Let P is an expression written in postfix notation.
Step 1: STACK=empty stack.
Step 2: Scan P from left to right and repeat step 3 and 4 for each symbol in P until end of expression.
Step 3: if an operand is encountered, push it on STACK.
Step 4: if an operator X encountered then;
a) Operand 2= pop (STACK).
b) Operand 1= pop (STACK).
c) Value= operand1 X operand 2. Data Structure and Algorithms
By. Surya Bam Page 4
d) Push value on STACK.
Step 5: Return the value at top of the STACK.
Step 6: Exit.

PRACTICAL 4

Title . Algorithm to perform various operations on a linear queue using a linear array

ALGORITHM

Algorithm for Insertion in Queue

Step 1: if (REAR == N) Then [Check for overflow]
Print. Overflow
Step 2: else
if (FRONT and REAR == 0) Then [Check if QUEUE is empty]
(a) Set FRONT = 1
(b) Set REAR = 1
Step 3: else
Set REAR = REAR + 1 [Increment REAR by 1]
[End of Step 4 if]
Step 4: QUEUE[REAR] = ITEM
Print. ITEM inserted
[End of Step 1 if]
Step 5:Exit

Algorithm for Deletion in Queue

Step 1: if (FRONT == 0) Then [Check for underflow]
Step 2: Print. Underflow
Step 3: else
Step 4: ITEM = QUEUE[FRONT]
Step 5: if (FRONT == REAR) Then [Check if only one element is left]
(a) Set FRONT = 0
(b) Set REAR = 0
Step 6: else
Step 7: Set FRONT = FRONT + 1 [Increment FRONT by 1]
[End of Step 5 if]
Step 8: Print. ITEM deleted
[End of Step 1 if]
Step 9: Exit

Title. **Algorithm to perform various operations on a circular queue usinga linear array**

ALGORITHM

Algorithm for Insertion in a circular queue

Insert CircularQueue ( )

Step 1: if (FRONT == 1 and REAR == N) or (FRONT == REAR + 1) Then
Step 2: Print. Overflow
Step 3: else
Step 4: if (REAR == 0) Then [Check if QUEUE is empty]
(a) Set FRONT = 1
(b) Set REAR = 1
Step 5: elseif (REAR == N) Then [if REAR reaches end if QUEUE]
Step 6: Set REAR = 1
Step 7: else
Set REAR = REAR + 1 [Increment REAR by 1]
[End of Step 4 if]
Step 8:Set QUEUE[REAR] = ITEM
Step 9: Print. ITEM inserted
[End of Step 1 if]
11:Exit

Title. Algorithm to perform various operations on a queue using a linear linked list

ALGORITHM

Algorithm to Insert an ITEM into a linked queue.

Step 1: if AVAIL=NULL, then write OVERFLOW and Exit.
Step 2: set NEW = AVAIL and AVAIL= link[AVAIL].
Step 3:Set info[NEW]=ITEM and link[NEW]=NULL
Step 4:if FRONT=NULL then FRONT= REAR=NEW.
else set link[REAR]=NEW and REAR=NEW.
Step 5:Exit.

Algorithm to delete the front element of the linked queue and stores it in ITEM
Step 1: if(FRONT=NULL) then write. UNDERFLOW and Exit
Step 2:Set TEMP=FRONT

Step 3:ITEM=info[TEMP]
Step 4:FRONT=link[TEMP]
Step 5:link[TEMP]=AVAIL and AVAIL=TEMP.
Step 6:EXIT

PRACTICAL 5

Title. **Algorithm to perform different operations on a binary search tree**

To create a nonempty tree, first create an empty tree, then insert nodes

B-TREE-CREATE(T)
Step 1: X ← allocate-node()
Step 2: leaf[X] ← true
Step 3: n[X] ← 0
Step 4: disk-write(X)
Step 5: root[T] ← X

Algorithm For Splitting a node in B-Tree

split-child(X,I,Y)
Z<-allocate-node();
leaf[Z] <-leaf[Y];
n[Z] <- T-1;
for J<-1 to T-1
do $key_j[Z]$ <- $key_{j+t}[Y]$
if not leaf[Y]
than for J <- 1 to T
do $c_j[Z]$<-$c_{j+t}[Y]$
n[Y] <-T-1;
for J<- n[X] +1 down to I+1
do $C_{j+1}[X]$<-$C_j[X]$
$C_{(i+1)}[X]$<-Z;
for J<- n[X] down to 1
do $key_{j+1}[X]$ <- $key_j[X]$
key <- key[X];
n[X]<-n[x]+1;
disk-write(Y);
disk-write(Z);
disk-write(X)

B-TREE-INSERT(T, k)
Step 1 R ← root[T]
Step 2 if n[R] = 2T - 1
Step 3 then S ← allocate-note()

Step 4 root[T] ← S
Step 5 leaf[S] ← false
Step 6 n[S] ← 0
Step 7 c1[S] ← R
Step 8 B-TREE-SPLIT-CHILD(S, 1, R)
Step 9 B-TREE-INSERT-NONFULL(S, K)
Step 10 else B-TREE-INSERT-NONFULL(R, K)

PRACTICAL 6

Title . Algorithm for Sorting an array using bubble sorting

In computer science and mathematics, a sorting algorithm is an algorithm that puts elements of a list in a certain order. The most-used orders are numerical order and lexicographical order. Efficient sorting is important to optimizing the use of other algorithms (such as search and merge algorithms) that require sorted lists to work correctly; it is also often useful for canonicalizing data and for producing human-readable output. More formally, the output must satisfy two conditions:

1.The output is in non-decreasing order (each element is no smaller than the previous element according to the desired total order);

2.The output is a permutation, or reordering, of the input.

Bubble Sort

Bubble sort is a simple sorting algorithm. It works by repeatedly stepping through the list to be sorted, comparing two items at a time and swapping them if they are in the wrong order. The pass through the list is repeated until no swaps are needed, which means the list is sorted. The algorithm gets its name from the way smaller elements "bubble" to the top (i.e. the beginning) of the list via the swaps. (Another opinion. it gets its name from the way greater elements "bubble" to the end.) Because it only uses comparisons to operate on elements, it is a comparison sort. This is the easiest comparison sort to implement.

ALGORITHM

Step 1:Repeat steps 2& 3 for K=1to N-1
Step 2:PTR=1
Step 3:repeat while PTR<N-K
a) if data(PTR)>data(PTR+1)
interchange data(PTR) & data (PTR+1)
b)PTR=PTR+1
Step 4:Exit

```
procedure bubbleSort( A . list of sortable items ) defined as:
  do
swapped = false
    for each I in 0 to length( A ) - 2 do:
```

```
if A[ I ] > A[ I + 1 ] then
      swap( A[ I ], A[ I + 1 ] )
      swapped = true
    end if
  end for
```

Title. **Algorithm to sort an array of integers in ascending order using selection sort.**

ALGORITHM

Step 1 the smallest element put it in the first position

Step2 find the next smallest element in the remaining elements

Step3 put it in the second position

Step4 And so on, until we get to the end of the array

Title . Algorithm for Sorting an array using insertion sorting

In abstract terms, every iteration of an insertion sort removes an element from the input data, inserting it at the correct position in the already sorted list, until no elements are left in the input. The choice of which element to remove from the input is arbitrary and can be made using almost any choice algorithm.

Sorting is typically done in-place. The resulting array after k iterations contains the first k entries of the input array and is sorted. In each step, the first remaining entry of the input is removed, inserted into the result at the right position, thus extending the result becomes with each element x copied to the right as it is compared against x.

The most common variant, which operates on arrays, can be described as:
Suppose we have a method called insert designed to insert a value into a sorted sequence at the beginning of an array. It operates by starting at the end of the sequence and shifting each element one place to the right until a suitable position is found for the new element. It has the side effect of overwriting the value stored immediately after the sorted sequence in the array.

To perform insertion sort, start at the left end of the array and invoke insert to insert each element encountered into its correct position. The ordered sequence into which we insert it is stored at the beginning of the array in the set of indexes already examined. Each insertion overwrites a single value, but this is okay because it's the value we're inserting.
A simple pseudocode version of the complete algorithm follows, where the arrays are zero-based:

ALGORITHM

(Insertion Sort) INSERTION (A, N)

This algorithm sorts the array A with N elements.

Step 1:set A(0)=-infinity [Initializes sentinel element]
Step 2:reapet step 3 to 5 for K=2,3,4.....n
Step 3:set TEMP=A(K) &PTR=K-1
Step 4:repeat while TEMP<A(PTR)
a)PTR+1=A(PTR)
b)PTR=PTR-1
Step 5:set A(PTR+1)=TEMP
[End of Step 2 loop]
Step 6:return.

Title. Algorithm to sort an array of integers in ascending order using radix sort.

ALGORITHM

Step 1:[find maximum width of integers]
Call M = find width[A,M]
Step 2:for PASS = 1 to M repeat step 3 -7
Set DIV =1
Step 3: for K = 0 to 9 repeat step 4
COUNT[K]=0
Step 4: for I=0 to N-1
Compute R =(A[I]/DIV)%10
BUCKET[1]*COUNT[R]=A[I]
COUNT[R] = COUNT[R]+1
End of loop started in step 4
End of loop started in step 3
Step5: set I=0
Step6 : for K=0 to 9 repeat step 7
Step 7: for J=0 to COUNT[K]
A[I++]=BUCKET[K][J]
Set DIV = DIV/10
End of loop started in step 7
End of loop started in step 6
End of loop started in step 2

Step8: Exit

Title. Algorithm to sort an array of integers in ascending order using merge sort.

ALGORITHM

MERGE (A,P,Q,R)
Step 1:set NA=1 , NB =1 , PTR = 1
Step 2:Repeat while NA<=R and NB<=S
Step 3:if A[NA]<B[NB] than
a)set C[PTR]=A[NA]
b)set PTR=PTR+1 and NA=NA+1
else
a)set C[PTR]=B[NA]
b)Set PTR=PTR+1 and NB=NB+1
[end of if structure]
[end of loop]
Step 4: assign remaining elements to c
If NA>R than
Repeat for K=0,1,2,3,4…..n
Set C[PTR+k]=B[NB+K]
End of loop
Else
Repeat for K=0,1,2,3,4…..n
Set C[PTR+k]=A[NA+K]
End of loop
End of if structure
Step5 : Exit

Title . Algorithm for Sorting an array using quick sorting

Quick-sort sorts by employing a divide and conquer strategy to divide a list into two sub-lists. The steps are:

Pick an element, called a pivot, from the list.

Reorder the list so that all elements which are less than the pivot come before the pivot and so that all elements greater than the pivot come after it (equal values can go either way). After this partitioning, the pivot is in its final position. This is called the partition operation.

1. Recursively sorts the sub-list of lesser elements and the sub-list of greater elements.

2. The base case of the recursion are lists of size zero or one, which are always sorted. The
Algorithm always terminates because it puts at least one element in its final place on each iteration (the loop invariant).

Search algorithm

In computer science, a search algorithm, broadly speaking, is an algorithm that takes a problem as input and returns a solution to the problem, usually after evaluating a number of possible solutions. Most of the algorithms studied by computer scientists that solve problems are kinds of search algorithms. The set of all possible solutions to a problem is called the search space. Brute-force search or "naïve"/uninformed search algorithms use the simplest, most intuitive method of searching through the search space, whereas informed search algorithms use heuristics to apply knowledge about the structure of the search space to try to reduce the amount of time spent searching.
In simple pseudocode, the algorithm might be expressed as:

ALGORITHM

Step 1: [Is List is valid?]
If n<=1 then Print "It is not a valid list"
Exit
Else
Step 2: set TOS = TOS + 1
Set START[TOS] = 1 and
Set END[TOS] = n
Step 3: Repeat step 4 to 7 while TOS != NULL
Step 4: set FIRST = START[TOS] and LAST = END[TOS]
Set TOS = TOS -1
Step 5: [call procedure RLMOVE]
Set LOC = RLMOVE(A,FIRST,LAST)
Step 6: if FIRST < LOC – 1 than [push left sublist]
Set TOS = TOS+1
Set START[TOS] = FIRST and END[TOS]=LOC-1
ENDIF
Step 7: if LAST > LOC+1 than[push right sublist]
Set TOS = TOS+1
Set START[TOS] = LOC+1 and END[TOS] = LAST
Endif
End of loop started in step 3
Step 8 : Exit

Title . Algorithm to sort an array of integers in ascending order using heap sort.

ALGORITHM

MAX-HEAPifY(A,I)
Step l = LEFT(I)
Step 2: R = RIGHT(I)
Step 3:if l <= A.heap-size and A[l] > A[I]
Step 4: LARGEST = l
Step 5:else LARGEST = I
Step 6:if R<= A.heap-size and A[R] > A[LARGEST]
Step 7: LARGEST = R
Step 8:if LARGEST != I
Step 9: exchange A[I] with A[LARGEST]
Step 10:MAX-HEAP ifY(A,LARGEST)

BUILD-MAX-HEAP(A)
Step 1: A.heap-size = A.length
Step 2: for I = [A.length/2] downto 1
Step 3: MAX-HEAPifY(A,I)

HEAPSORT(A)
Step 1: BUILD-MAX-HEAP(A)
Step 2: for I = A.length downto 2
Step 3: exchange A[1] with A[I]
Step 4: A.heap-size = A.heap-size - 1
Step 5: MAX-HEAPifY(A, 1)

Title . Algorithm to sort an array of integers in ascending order using shell sort.

ALGORITHM

Step 1:N = length of the list, increment = N/2

Step 2: Do the following until increment>0 I = increment until I<N do the following

Store the value at index I in a Temporary variable(TEMP)

J= I until J>= increment do the following

Step 3: ifTEMP is less than the value at index J-increment

Replace the value at index J with the value at index J-increment and

Decrease J by increment.

Step 4: else break out of the J loop.

Replace the value at index J with TEMP and increase I by 1.

Divide increment by 2.

PRACTICLE 7

Title -Algorithm for linear search of an element in an array

Arrays

Arrays contain a linear table of values, which must all be of the same type, called the base type. These values are called the array's elements, and any element can be specified by its position in the array, called its index. In this section we will discuss C++'s built-in arrays, together with typical array operations like copying and searching.

Searching

You will find that you often need to search for a value in a table of numbers. You want to know the index of that value within that table, or even simply whether the value is present. The simplest method is to run along until you find the value, or key. if you don't find the key, you just return -1 to indicate that the key is not present in the array, as in the following example, where the function linear_search() is defined:

This method is fast enough for small tables, but it isn't adequate for large tables that need to be processed quickly. A much better method is to use a binary search, but the elements of that table must be in ascending order. To perform a binary search, you first divide the table in two; the key must be either in the first half or the second half. You compare the key to the value in the middle; if the key is less than the middle value, you choose the first half, and if the key is greater than the middle value, you choose the second half. Then you repeat the process, dividing the chosen half into halves and comparing the key, until either the key is equal to the value or until you cannot divide the table any further. Here is a C++ example of performing a binary search:

ALGORITHM

Linear Search

Step 1:insert item at the end of data.set data(N+1)=ITEM
Step 2:initialise counter set LOC=1
Step 3:search for ITEM. repeat while data (LOC)!=ITEM
set LOC =LOC+1
Step 4:(successful?). if LOC=N+1,set LOC=0
Step 5:Exit

Title - Algorithm for binary search of an element in an array

Theory:-

How much faster is a binary search than a linear search? Well, if there are 1,000 entries in a table, a binary search will take only about 30 tries (on average) to find the key. The catch here is that the table must be sorted. if it's already sorted, then it seems a good idea to insert any new value in its proper place. Finding the place is easy. You run along and stop when the key is less than the value. The code for finding the position is similar to the code for a linear search, but -1 now means that the key was larger than all the existing values and can simply be appended to the table:

ALGORITHM

Step 1:initialize segment variables
set BEG=LB and END=UB and MID =int (BEG+END)/2
Step 2: repeat steps 3 and 4 while BEG<END
Step 3:ifITEM<data(MID)
set END =MID-1
else
set BEG =MID +1
Step 4: set MID =int(BEG+END)/2
Step 5:if data(MID)=ITEM
set LOC=MID
else set LOC =NULL
Step 6:Exit