Department of Computer Science & Engineering
IGIT, GGS Indraprastha University, Delhi

LAB MANUAL FOR

DATA STRUCTURES

(Version 1.0)

Odd Semester
(August, 2011)

Prepared by:
Vivekanand Jha
(Assistant Professor)
Deptt. of CSE, IGIT,Delhi

Table of Contents

S. No.	Contents	Page No.
1	Objective	3
2	Prerequisites & Execution mode	3
3	Practical Index & Details of experiment	4-35
4	Submission mode	36

Course Objective

The aim of this course is to provide the practical exposure to data structures implementation, with an emphasis on core concepts of data structures.
At the end of course students will equipped with following:

- Good programming skills of data structures
- Practical exposure to the various algorithms used in data structures
- Able to use most efficient data structures for new applications in computer science
- Strong skill to implement complex algorithm.

Pre-requisites:

- Windows/ Linux platform operating knowledge.
- C programming language.
- Theory course work on Data Structures.

Execution Mode:

- For the given problem statement design flowchart/Algorithm/Logic
- Define variables and functions, which will show the flow of program
- Write C code in the file
- Compile code
  Using gcc compiler for Linux, create a.out executable file.
  Using Windows platform, compile and run .c/.cpp file.
- Test the program using sample input and write down output

PRACTICAL INDEX

Practical #	Name of Practical	Page No	Deadlines
1	To implement Linear Search and Spiral Scanning using Array.		Week 1
2	Implement Single Link List with following operations: i) Insertion of a node at first node, at any position and at end of list. ii) Deletion of a node at start, at middle and at end of list. iii) Display the link list. iv) Count the number of nodes in the link list. v) Search a node in the link list. vi) Sort the link list. vii) Reverse the link list.		Week 2,3
3	Implement Stack with all primitive operations by using Array.		Week 4,5
4	Implement the application of stack for following operations: (i) Infix notation to Postfix notation, (ii) Postfix expression evaluation.		Week 4,5
5	Implement Queue with all primitive operations by using Array.		Week 4,5
6	Implement doubly link list with primitive operations: (i) Create a doubly linked list. (ii Insert a new node to the left of the node. (iii) Delete the node of a given data. (iv) Display the contents of the list.		Week 6
7	Implement Circular link list with primitive operations. (i) Creation of the Circular list (ii) Insertion of the node (iii) Deletion an element (iv) Display the list		Week 7
8	Implement Stack and Queue with all primitive		Week 8

	operations by using link list.		
9	Implement Binary Search Technique.		Week 9
10	Implement Binary Tree and its Traversal.		Week 10
11	Implement BFS & DFS over a graph.		Week 11
12	Implement shortest path algorithms.		Week 12
13	Implement Bubble Sort, Quick Sort, and Merge Sort (as many sorting techniques you can implement).		Week 13, 14
	INTERNAL PRACTICAL TEST		Week 15 onwards
	VIVA VOICE		Week 15 onwards

1. To implement Linear Search and Spiral Scanning using Array.

Linear Search:

Algorithm for **linear search** as

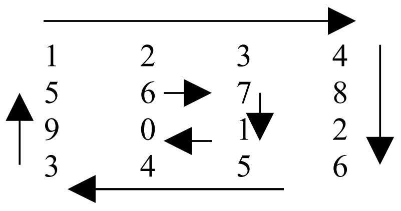
```
a) Read the data to be searched 'X'.
b) Scan the array from left to right.
c) Compare 'X' with the first element.
d) If equal then
Print 'SUCCESS" and return
Else
Compare 'X' with second element and so on
e) The above process is repeated for all the elements in the array.
f) If no value in the array matches with 'X' then Print "NO
MATCH" and return
```

Input: Array having data
Output: Search result for a particular data with correct positions (if available)

Spiral Scanning:

This problem scans 2D-array such that outer elements are scanned first and then inner elements are scanned in spiral manner.

Consider following 2-dimensional matrix below



Spiral scanning for given above matrix will be:
1 2 3 4 8 2 6 5 4 3 9 5
6 7 1 0 (follow the arrow track from outer to inner)

Input: M x N array
Output: Scanning and printing of input array elements in spiral manner.

2. TO implement Single Link List with following operations:

Note: This program student should create in menu driven manner for options (i) to (vii).

(i) Insertion of a node at start, at middle and at end of list.

Algorithm to insert at first node:

```
Algorithm InsertFirst(p, e)
// p - pointer to a linked list
// e - element to be added
// Getnode() returns a new node
q ← Getnode()
info(q) ← e
next(q) ← p
p ← q
return p
end InsertFirst
```

Algorithm to insert at any position in link list:

```
Algorithm InsertAfter(p, x, e)
// p - pointer to a linked list
// Getnode() returns a new node
// x - key node after which e is inserted
// e - element to be added
k ← p
while k ≠ NIL and info(k) ≠ x do // find the key
node
k ← next(k)
if k = NIL
then
        Write "Node not found"
        return p
q ← Getnode()
info(q) ← e
next(q) ← next(k)
next(k) ← q
return p
end InsertAfter
```

Algorithm to insert at Last node:

```
Algorithm InsertLast(p, e)
// p - pointer to a linked list
// Getnode() returns a new node
// e - element to be added
if p = NIL
then
return p
k ← p
while next(k) ≠ NIL do // find the last node
k ← next(k)
q ← Getnode()
info(q) ← e
next(k) ← q
next(q) ← NIL
return p
end InsertLast.
```

Input: User select the options of insertions
Output: Print the content of link list after each operation

(ii)Deletion of a node at start, at middle and at end of list.

Use the following algorithm to delete an element from link list. Suitable modification in algorithm will support for all class of deletion.

```
Algorithm DeleteNode(p, x)
// p - pointer to linked list
// x - key node to be deleted
// k and pred – temporary variables
k ← p; pred ← NIL
while k ≠ NIL and info(k) ≠ x do // find the key node
pred ← k
k ← next(k)
if k = NIL
then
Write "Node not found"
else
if pred = NIL // only one node in the list
then
p ← next(p)
else
next(pred) ← next(k)
return p
end DeleteNode.
```

Input: node to be deleted
Output: Link list after deletion of a node

(iii)Display the content of link list.

Step1: Create link list
Step2: Scan and print the entire link list elements one by one.

Input: Link list
Output: Print the content of link list

(iv) Count the number of nodes in the link list.

Algorithm:

```
int Length(struct node* head)
{
int count = 0;
struct node* current = head;
while (current != NULL) {
count++;
current = current->next;
}
return(count);
}
```

Input: Link list
Output: Print the total number of nodes in link list

(v) Search a node in the link list.

Algorithm SearchNode(*p*, *x*)

```
// p - pointer to a linked list
// x - key node to be searched
// k - temporary variable
k ← p
while k ≠ NIL and info(k) ≠ x do // find the key node
k ← next(k)
if k = NIL
then
return false // key not found
else
return true // key found
end SearchNode.
```

Input: Link list with element to be search.
Output: Print the position of input element in the link list

(vi) To Sort the link list
One can use the merge sort strategy.The MergeSort strategy is: split into sublists, sort the sublists recursively, merge the twosorted lists together to form the answer.

Algorithm:

```
void MergeSort(struct node** headRef)
 {
struct node* head = *headRef;
struct node* a;
struct node* b;
// Base case -- length 0 or 1
if ((head == NULL) || (head->next == NULL)) {
return;
}
FrontBackSplit(head, &a, &b); // Split head into 'a' and 'b' sublists
// We could just as well use AlternatingSplit()
MergeSort(&a); // Recursively sort the sublists
MergeSort(&b);
*headRef = SortedMerge(a, b);
// answer = merge the two sorted lists together
}
```

Input: Link list to be sorted.
Output: Print sorted link list.

(vii) To reverse the link list

Algorithm

```
/*
Iterative list reverse. Iterate through the list left-right. Move/insert
each node to the front of the result list -- like a Push of the node.
*/
static void Reverse(struct node** headRef) {
struct node* result = NULL;
struct node* current = *headRef;
struct node* next;
while (current != NULL) {
next = current->next; // tricky: note the next node
current->next = result; // move the node onto the result
result = current;
current = next;
}
*headRef = result;
}
```

Input: Link list to be reversed.
Output: Print reverse link list.

3. To implement Stack with all primitive operations by using Array.

Algorithm:

PUSH(STACK , TOP, MAXSTK, ITEM)

This procedure insert an ITEM onto a stack.

```
1. If TOP = MAXST , then Print “OVERFLOW” and return
2. Set TOP = TOP + 1
3. Set STACK[TOP] = ITEM
4. Return
```

POP(STACK , TOP, ITEM)

This procedure delete the top element of STACK and assign it to the variable ITEM.

```
1. If TOP = 0 , then Print “ Under FLOW” and return
2. Set ITEM = STACK[TOP]
3. Set TOP = TOP - 1
4. Return
```

Input: User will give option of push or pop
Output: Print the content of stack after each operation

4.(i) To implement conversion of an infix expression into postfix expression.

Algorithm:

```
1. Define a stack of to hold the characters.
2. Initialize stack top = -1.
3. Read the infix expression.
4. Add ‘)’ to the end of the infix expression.
5. Push ‘(‘ to the stack.
6. Scan the infix expression from left to right and repeat the step 7 for all
the characters in the infix expression.
7. If the character is an operand
        Add it to the postfix expression
If the character is ‘(‘
        Push it to the stack
If the character is ‘)’
        Repeatedly Pop the characters from the stack and add it
        to the postfix expression until ‘)’ is encountered.
        Pop ‘)’
If the character is an operator

        If the precedence of the character is lesser than or equal to the
        precedence of the operator in the top of the stack repeatedly pop
        the characters from the stack and add it to the post fix expression
        till an operator of higher precedence is encountered.

        Push the operator to the top of the stack.

8. Print the postfix expression.
9. Stop.
```

Input: Infix expression
Output: Postfix expression

4.(ii) To implement evaluation of a postfix expression.

Algorithm:

```
1.  Declare the structure for the stack.

2.  Read the postfix expression.

3. Repeatedly execute the following for all the characters in the expression
from left to right

        If the character is a number

            Then convert the character to integer by subtracting '0'
            from the character.

            Push the integer value into the stack.

        If the character is a operator

            Pop two values from the stack and perform the operation
            and store the  result to stack.

4. The stack top has the result. Pop it and Print.

5. Stop.
```

Input: Postfix expression
Output: Value of expression

5. To implement Queue with all primitive operations by using Array.

Algorithm:

QINSERT(QUEUE,N,FRONT,REAR,ITEM)

```
1. If FRONT =1 and REAR=N or if FRONT=REAR+1 then :
Print : Overflow and return
2. IF FRONT = NULL then
Set FRONT =1 and REAR =1
Else If REAR =N then
Set REAR = 1
Else
Set REAR =REAR +1
3. Set QUEUE[REAR]=ITEM
4. RETURN
```

QDELETE(QUEUE,N,FRONT,REAR,ITEM)

```
1. If FRONT =NULL then :
Print : Underflowflow and return
2. Set ITEM = QUEUE[FRONT]
3. IF FRONT = REAR then
Set FRONT =0 and REAR =0
Else If FRONT =N then
Set FRONT = 1
Else
Set FRONT =FRONT +1
4. RETURN
```

Input: Queue with operations insert or delete
Output: Content of queue after each operation

(6) Implement doubly link list with primitive operations.

(i) Create a doubly linked list.
(ii) Insert a new node to the left of the node.
(iii) Delete the node of a given data.
(iv) Display the contents of the list.

Algorithm InsertFront(*p*, *e*)

```
// $p$ – list pointer
// $e$ - element to be pushed
$q \leftarrow$ Getnode()
$info(q) \leftarrow e$
$prev(q) \leftarrow$ NIL
$next(q) \leftarrow p$
if $p \neq$ NIL // not the first node
then
    $prev(p) \leftarrow q$
$p \leftarrow q$
return $p$
end InsertFront.
```

Algorithm InsertAfter(*p*, *x*, *e*)

```
// $p$ - pointer to a linked list
// $x$ - key node after which e is inserted
// $e$ - element to be added
$k \leftarrow p$
while $k \neq$ NIL and $info(k) \neq x$ do // find the key node
    $k \leftarrow next(k)$
if $k$ = NIL
then
    Write "Node not found"
    return $p$
$q \leftarrow$ Getnode()
$info(q) \leftarrow e$
if $next(k) \neq$ NIL
then
    $next(q) \leftarrow next(k)$
    $prev(q) \leftarrow k$
    $next(k) \leftarrow q$
    $prev(next(k)) \leftarrow q$
else // key node is at
the end
    $next(q) \leftarrow$ NIL
    $next(k) \leftarrow q$
    $prev(q) \leftarrow k$
end InsertAfter
```

```
Algorithm DeleteNode(p, x)
    // p - pointer to a linked list
    // x - key node to be deleted
    // k – address of key node
    k ← p
    while k ≠ NIL and info(k) ≠ x do // find the key node
        k ← next(k)
    if k = NIL
    then
        Write "Node not found"
        return p // key found
    if k = p and next(k) = NIL // only one node in the list
    then
        p ← NIL
        return p
    if next(k) = NIL // last node is key
    then
        next(prev(k)) ← next(k)
    else
        if prev(k) = NIL // first node is key
        then
            p ← next(p)
            prev(p) ← NIL
        else
        next(prev(k)) ← next(k)
        prev(next(k)) ← prev(k)
    return p
    end DeleteNode.
```

Input: Menu driven input based on (i), (ii), (iii), and (iv)
Output: Show content of link list after each operation

(7) To implement circular link list with operations:

(i) Creation of the Circular list

(ii) Insertion of the node

(iii) Deletion an element

(iv) Display the list

Algorithm for the Creation of the Circular list

CREATE (* TEMPHEAD)
[This function creates the circular list and TEMPHEAD is the pointer variable which points the first element of the list]

```
1.[Save the address of the first element]
SAVE = TEMPHEAD
2. [Repeat thru step 5]
Repeat while Choice! = ‘n’
3. [Allocate the New node]
NEW NODE()
4. [Initialize the fields of new node]
INFO (NEW) = X
LINK (SAVE) = NEW
SAVE = NEW
5. [Want to insert another node]
Read (Choice)
6. [Set the LINK field of Last inserted element]
LINK (SAVE) = TEMPHEAD
7. [Finished]
Return
```

Algorithm for the insertion of the node in the circular list

INSERT (* TEMPHEAD, KEY)
[This Function inserts an element after the node which have the info field equal to the KEY variable and TEMPHEAD is the pointer which points the first element of the list and SAVE is the temp variable for the store address of the first element]

```
1. [Allocate the Memory for the NEW node]
       NEW NODE( )
2. [Set fields of the NEW node]
       INFO (NEW) = X
       LINK (NEW) = NULL
3. [Save address of the first node]
       FIRST = TEMPHEAD
4. [Insertion as first node and find last element of the list]
       Repeat while LINK (TEMPHEAD)! = NULL
5. [Insert the node]
       LINK (TEMPHEAD) = NEW
       LINK (NEW) = FIRST
       FIRST = NEW
       Return (FIRST)
6. [Insert in the list other than the first node]
       Repeat while INFO (LINK (TEMPHEAD)) = KEY
7. [Set the link for the NEW node]
       LINK (NEW) = LINK (TEMPHEAD)
       LINK (TEMPHEAD) = NEW
8. [Finished]
       Return (FIRST)
```

Algorithm for the Deletion an element from the circular list

DELETE (*TEMPHEAD, KEY)
[This Function deletes an element from the circular list]

```
1. [Check for the empty list]
       If TEMPHEAD = NULL
       Then write ("Empty List")
2. [List contain Single node]
       if LINK (TEMPHEAD) = TEMPHEAD
       Return NULL
       Free (TEMPHEAD)
3. [Save the address of the first node]
       FIRST = TEMPHEAD
4. [Deletion of the first node]
       Repeat while LINK (TEMPHEAD)! =NULL
5. [Delete the node]
       LINK (TEMPHEAD) = LINK (FIRST)
       LINK (FIRST) = FIRST
       Return (FIRST)
```

```
6. [Finding desire node]
        Repeat while INFO (LINK (TEMPHEAD)) = KEY
7. [Deletes the node]
        TEMP = LINK (TEMPHEAD)
        LINK (TEMPHEAD) = LINK (LINK (TEMPHEAD))
        Free (TEMP)
8. [Finished]
        Return (FIRST)
```

Algorithm for display the element of the circular link list

```
DISPLAY (*TEMPHEAD)
        1. [Check for the empty list]
                If TEMPHEAD = NULL
                Then write ("Empty list")
                Return
        2. [Print the desire node]
                Repeat while LINK (TEMPHEAD)! = TEMPHEAD
                Write (INFO (TEMPHEAD))
        3. [Finished]
                Return
```

Input: Menu driven input based on (i), (ii), (iii), and (iv)
Output: Show content of link list after each operation

(8) Implement Stack and Queue with all primitive operations by using link list.

STACK:	PUSH, POP
QUEUE:	ENQUEUE, DEQUEUE

```
Algorithm Push(p, e)
        // p – stack pointer
        // e - element to be pushed
        q ← Getnode()
        info(q) ← e
        next(q) ← p
        p ← q
        return p
        end Push.

Algorithm Pop(p, e)
        // p – stack pointer
        // e – returned as popped element
        if p = NIL
        then Write "Stack Underflow"
        else
        e ← info(p)
        next(p) ← p
        return p
        end Pop.

Algorithm enqueue(f, r, e)
        // f and r – front and rear pointers
        // e - element to be inserted
        q ← Getnode()
        info(q) ← e
        next(q) ← NIL
        if f = NIL
        then f ← r ← q
        else
        next(r) ← q
        r ← q;
        end enqueue.

Algorithm dequeue(f, r, e)
        // f and r – front and rear pointers
        // e – returned element
        if f = NIL
        then Write "Queue Underflow"
        else
        e ← info(f)
        f ← next(f)
        return e
```

end dequeue.

(9) To implement Binary Search Technique.

Algorithm steps for **Binary Search** :

```
a) Sort the array in ascending order.
b) Let lb=0 and ub=n-1
c) Read the data to be searched 'X'.
d) Find the mid position of the given array
        Mid=(lb+ub)/ 2 (N --- No.of Elements in the array)
e) Compare X with a[mid]
        If equal then
                Goto step (g)
        Else
                If X less than a[mid] then ub=mid-1
                If 'X' greater than a[mid] then lb=mid+1
f) If lb<=ub
        Repeat steps (d) and (e) for the sub array lb to ub
   Else
        Goto step (g)
g) If (lb>ub)
        Print "Search Success"
    Else
        Print "Search Failed"
f) Return.
```

Input: Array containing data, user enters data to be searched.
Output: Successful search with data position / Unsuccessful search

(10) To implement Binary Search Tree and its Traversal using link list

Algorithm CreateTree(*p*, *e*)

```
    // p – pointer to the root
    // e – element to be inserted
    if p = NIL
    then
        p = Getnode()
        info(p) ← e
        left(p) ← right(p) ← NIL
    else
        if e = info(p)
        then
            Write "Duplicate element. Insertion can't be
        done"
        else
            if e < info(p)
            then
                left(p) ← CreateTree(left(p), e)
            else
                right(p) ← CreateTree(right(p), e)
    return p
    end CreateTree.
```

Algorithm Preorder(*p*)

```
        // p – pointer to the root
        if p ≠ NIL
        then
            Write "Node visited: ", info(p)
            Preorder(left(p))
            Preorder(right(p))
        end Preorder.
```

Algorithm Inorder(*p*)

```
        // p – pointer to the root
        if p ≠ NIL
        then
            Inorder(left(p))
            Write "Node visited: ", info(p)
            Inorder(right(p))
        end Inorder.
```

Algorithm Postorder(*p*)

```
        // p – pointer to the root
        if p ≠ NIL
        then
            Postorder(left(p))
            Postorder(right(p))
            Write "Node visited: ", info(p)
        end Postorder.
```

(11) To implement BFS & DFS over a graph.

(i) Breadth First Search.
Suppose G = (V;E) is directed graph implemented via adjacency lists. Given a source vertex s we want to find the length of a shortest path from s to each other vertex. We use Breadth First Search. The algorithm makes use of a queue Q, and an array d[u] indexed by the vertices; at any point d[u] is an estimate of the distance of s to u. An auxiliary array of colors, white, gray and black, is used to indicate: unseen, being processed, finished.

```
BFS[G, s] ( )
1. for each v in V
2. d[v]=infinity;
3. color[v]=white;
4. d[s]=0;
5. Q={s};
6. color[s]=gray;
7. while Q is not empty {
8. u=dequeue[Q];
9. for each v on Adj[u]
10. if (color[v]==white) {
11. d[v]=d[u]+1;
12. color[v]=gray;
13. enqueue[v]; }
14. color[v]=black; }
15. return d[ ].
```

(ii)Depth First Search
This is another strategy for graph exploration, which in effect uses a stack, rather than a queue for storing vertices being processed. It makes use of colors, a global clock time, and computes a discovery time d[u] and finishing time f[u] for each vertex u. The algorithm uses a recursive subroutine Depth First Visit[u].

```
DFS[G]( ) {
for each vertex u
color[u]=white;
time=0;
for each vertex u
if (color[u]==white)
DFV[u];
}

==================

DFV[u]( ){
color[u]=gray;
d[u]=++time;
for each v on Adj[u]
if (color[v]==white) (*)
DFV[v];
color[u]=black;
f[u]=++time;
}
```

(12) To implement shortest path algorithm

In graph theory, the shortest path problem is the problem of finding a path between two vertices (or nodes) in a graph such that the sum of the weights of its constituent edges is minimized. An example is finding the quickest way to get from one location to another on a road map; in this case, the vertices represent locations and the edges represent segments of road and are weighted by the time needed to travel that segment.

Dijkstra's Algorithm

```
1 Function Dijkstra(G, w, s)
2 for each vertex v in V[G] // Initialization
3 do d[v] := infinity
4 previous[v] := undefined
5 d[s] := 0
6 S := empty set
7 Q := set of all vertices
8 while Q is not an empty set
9 do u := Extract-Min(Q)
10 S := S union {u}
11 for each edge (u,v) outgoing from u
12 do if d[v] > d[u] + w(u,v) //Relax( u,v)
13 then d[v] := d[u] + w(u,v)
14 previous[v] := u
```

Input: Weighted graph with source node.
Output: Shortest path (from source node to any node in given graph).

(13) To implement different Sorting techniques

One of the fundamental problems of computer science is ordering a list of items. There's a plethora of solutions to this problem, known as sorting algorithms. Some sorting algorithms are simple and intuitive, such as the bubble sort. Others, such as the quick sort are extremely complicated, but produce lightning-fast results.

Different Sorting Techniques

1. Bucket Sort

Bucket sort is possibly the simplest distribution sorting algorithm. The essential requirement is that the size of the universe from which the elements to be sorted are drawn is a small, fixed constant, say *m*.

For example, suppose that we are sorting elements drawn from **{0, 1, . . ., m-1}**, i.e., the set of integers in the interval **[0, *m*-1]**. Bucket sort uses ***m*** counters. The ***i***th counter keeps track of the number of occurrences of the ***i***th element of the universe. The figure below illustrates how this is done.

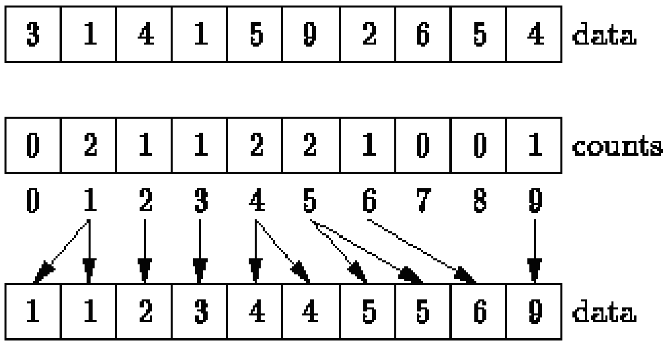


Figure: Bucket Sorting

In the figure above, the universal set is assumed to be **{0, 1, . . ., 9}**. Therefore, ten counters are required-one to keep track of the number of zeroes, one to keep track of the number of ones, and so on. A single pass through the data suffices to count all of the elements. Once the counts have been determined, the sorted sequence is easily obtained. E.g., the sorted sequence contains no zeroes, two ones, one two, and so on.

Program Implementation

```
void bucketSort(dataElem array[], int array_size)
        {
                int i, j;
                dataElem count[array_size];

                for(i =0; i < array_size; i++)
                        count[i] = 0;
                for(j =0; j < array_size; j++)
                        ++count[array[j]];
                for(i =0, j=0; i < array_size; i++)
                        for(; count[i]>0; --count[i])
                                array[j++] = i;

        }
```

2. Selection Sorting

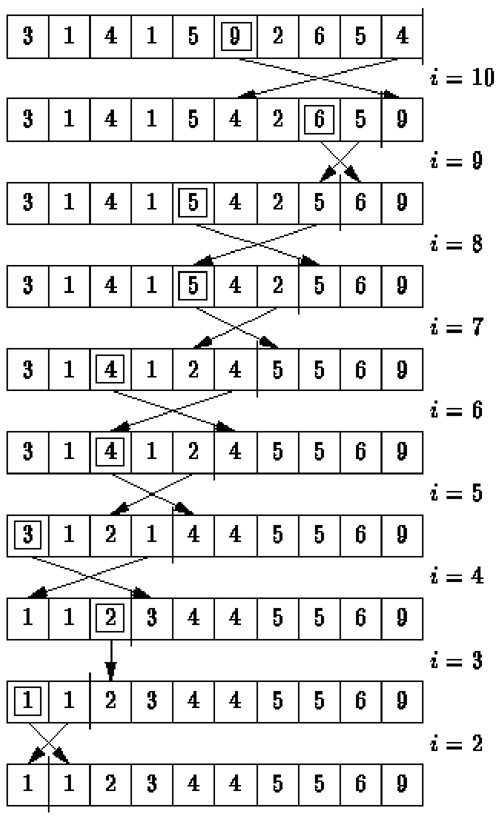
The selection sort algorithms constructs the sorted sequence one element at a time by adding elements to the sorted sequence *in order*. At each step, the next element to be added to the sorted sequence is selected from the remaining elements.

Because the elements are added to the sorted sequence in order, they are always added at one end. This is what makes selection sorting different from insertion sorting. In insertion sorting elements are added to the sorted sequence in an arbitrary order. Therefore, the position in the sorted sequence at which each subsequent element is inserted is arbitrary.

Both selection and insertion sorts sort the arrays *in place*. Consequently, the sorts are implemented by exchanging array elements. Nevertheless, selection differs from exchange sorting because at each step we *select* the next element of the sorted sequence from the remaining elements and then we move it into its final position in the array by exchanging it with whatever happens to be occupying that position.

At each step of the algorithm, a linear search of the unsorted elements is made in order to determine the position of the largest remaining element. That element is then moved into the correct position of the array by swapping it with the element which currently occupies that position.

For example, in the first step shown in the Figure below, a linear search of the entire array reveals that 9 is the largest element. Since 9 is the largest element, it belongs in the last array position. To move it there, we swap it with the 4 that initially occupies that position. The second step of the algorithm identifies 6 as the largest remaining element and moves it next to the 9. Each subsequent step of the algorithm moves one element into its final position.



Selection Sort Program Implementation

```
void selectionSort( dataElem array[ ], int array_size)
   {
      int i, j;
      int min, temp;

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

3. Insertion Sorting

The insertion sort works just like its name suggests - it inserts each item into its proper place in the final list. The simplest implementation of this requires two list structures - the source list and the list into which sorted items are inserted. To save memory, most implementations use an in-place sort that works by moving the current item past the already sorted items and repeatedly swapping it with the preceding item until it is in place.

Insertion Sort Program Implementation

```
void insertionSort(int array[], int array_size)
    {
       int i, j, index;

       for (i=1; i < array_size; i++)
          {
               index = array[i];
               j = i;
               while ((j > 0) && (array[j-1] > index))
                   {
                       array[j] = array[j-1];
                       j = j - 1;
                   }
               array[j] = index;
          }
     }
```

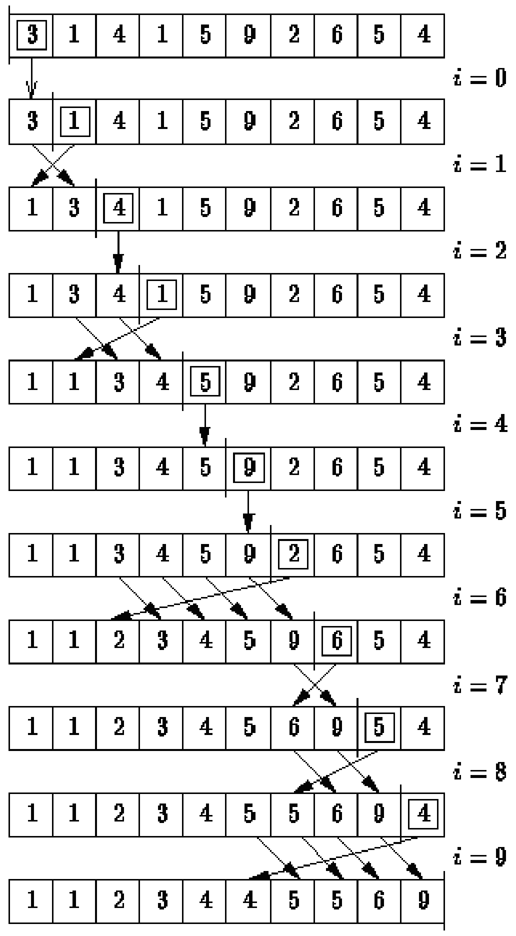


Figure: Insertion Sorting

4. Bubble Sort

The bubble sort is the oldest and simplest sort in use. Unfortunately, it's also the slowest.

The bubble sort works by comparing each item in the list with the item next to it, and swapping them if required. The algorithm repeats this process until it makes a pass all the way through the list without swapping any items (in other words, all items are in the correct order). This causes larger values to "bubble" to the end of the list while smaller values "sink" towards the beginning of the list.

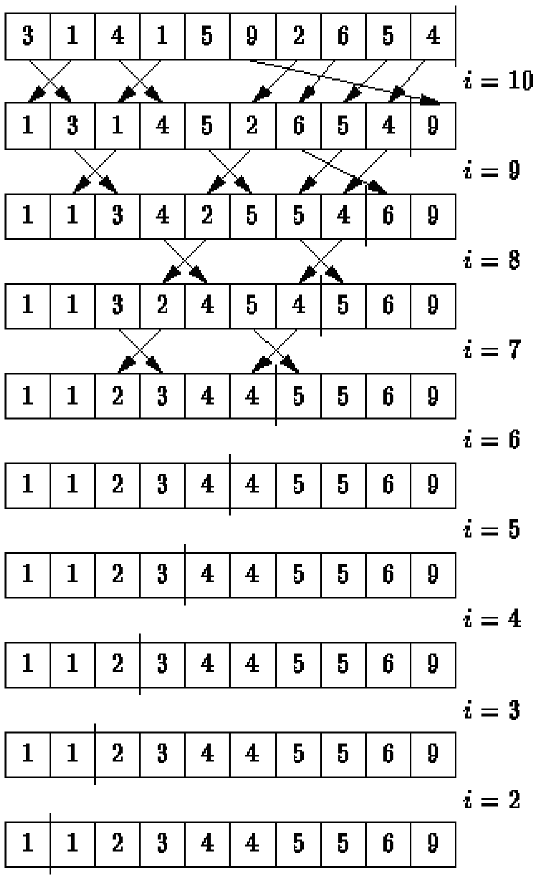


Figure: Bubble Sorting

Bubble Sort Program Implementation

```
void bubbleSort(int array[], int array_size)
   {
      int i, j, temp;

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

5. Quicksort

The quick sort is an in-place, divide-and-conquer, massively recursive sort. The algorithm is simple in theory, but very difficult to put into code (computer scientists tied themselves into knots for years trying to write a practical implementation of the algorithm, and it still has that effect on university students).

The recursive algorithm consists of four steps:

1. If there is one or less element in the array to be sorted, return immediately.
2. Pick an element in the array to serve as a "pivot" point. (Usually the left-most element in the array is used.)
3. Split the array into two parts - one with elements larger than the pivot and the other with elements smaller than the pivot.
4. Recursively repeat the algorithm for both halves of the original array.

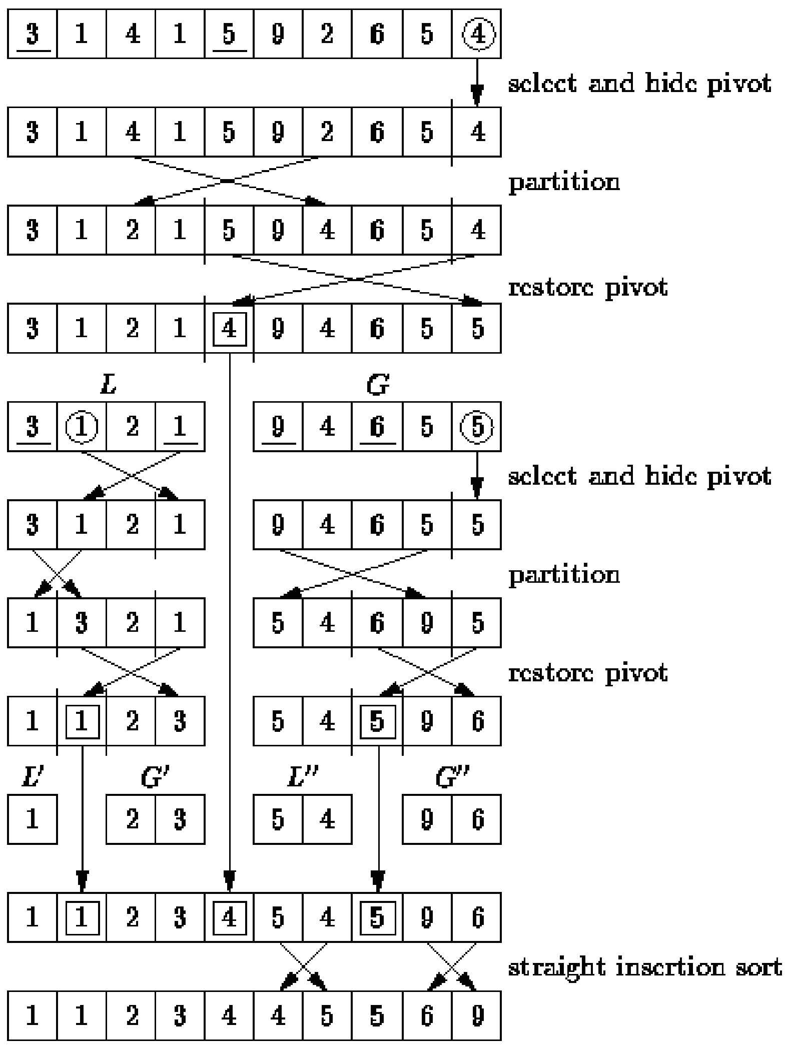


Figure: "Quick" Sorting

Quick Sort Program Implementation

```
void q_sort(int array[], int left, int right)
    {
        int pivot, l_hold, r_hold;

        l_hold = left;
        r_hold = right;
        pivot = array[left];
        while (left < right)
          {
                while ((array[right] >= pivot) && (left < right))
                right--;
                if (left != right)
                   {
                        array[left] = array[right];
                        left++;
                   }
                while ((array[left] <= pivot) && (left < right))
                     left++;
                if (left != right)
                     {
                        array[right] = array[left];
                        right--;
                     }
          }
        array[left] = pivot;
        pivot = left;
        left = l_hold;
        right = r_hold;
        if (left < pivot)
        q_sort(array, left, pivot-1);
        if (right > pivot)
            q_sort(array, pivot+1, right);
  }

void quickSort(int array[], int array_size)
    {
        q_sort(array, 0, array_size - 1);
    }
```

6. Heap Sort

The heap sort is the slowest of the fast sorting algorithms, but unlike algorithms such as the merge and quick sorts it does not require massive recursion or multiple arrays to work. This makes it the most attractive option for *very* large data sets of millions of items.

The heap sort works as it name suggests - it begins by building a heap out of the data set, and then removing the largest item and placing it at the end of the sorted array. After removing the largest item, it reconstructs the heap and removes the largest remaining item and places it in the next open position from the end of the sorted array. This is repeated until there are no items left in the heap and the sorted array is full. Elementary implementations require two arrays - one to hold the heap and the other to hold the sorted elements.

To do an in-place sort and save the space the second array would require, the algorithm below "cheats" by using the same array to store both the heap and the sorted array. Whenever an item is removed from the heap, it frees up a space at the end of the array that the removed item can be placed in.

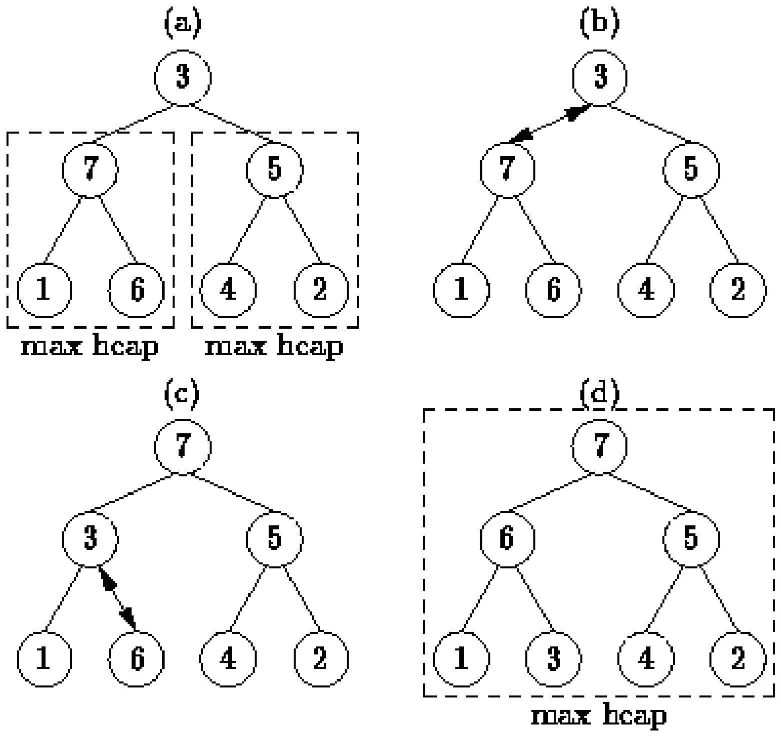


Figure: Combining Heaps by Percolating Values

```
void siftDown(int array[], int root, int bottom)
    {
        int done, maxChild, temp;

        done = 0;
        while ((root*2 <= bottom) && (!done))
           {
              if (root*2 == bottom)
```

```
                maxChild = root * 2;
            else if (array[root * 2] > array[root * 2 + 1])
                    maxChild = root * 2;
                 else
            maxChild = root * 2 + 1;

            if (array[root] < array[maxChild])
                {
                   temp = array[root];
                   array[root] = array[maxChild];
                   array[maxChild] = temp;
                   root = maxChild;
                }
            else
                 done = 1;
        }
  }


void heapSort(int array[], int array_size)
    {
        int i, temp;

        for (i = (array_size / 2)-1; i >= 0; i--)
             siftDown(array, i, array_size);

        for (i = array_size-1; i >= 1; i--)
           {
             temp = array[0];
             array[0] = array[i];
             array[i] = temp;
             siftDown(array, 0, i-1);
           }
    }
```

==

Submission Mode

1. Handwritten file.

2. One side of page to be used for writing.

3. File format is:

(i) File heading
(ii) Table of Contents
(iii) Algorithm
(iv) Program (Proper aligned and Commented at right place)
(v) Input
(vi) Output
(vii) Constraints / Remarks if any

GOOD LUCK