Q.2a. Define generic data abstract types and its notations.

Sol: ADTs are used to define a new type from which instances can be created. As shown in the list example, sometimes these instances should operate on other data types as well. For instance, one can think of lists of apples, cars or even lists. The semantical definition of a list is always the same. Only the type of the data elements change according to what type the list should operate on.

This additional information could be specified by a *generic parameter* which is specified at instance creation time. Thus an instance of a *generic ADT* is actually an instance of a particular variant of the ADT. A list of apples can therefore be declared as follows:

```
List<Apple> listOfApples;
```

The angle brackets now enclose the data type for which a variant of the generic ADT *List* should be created. *listOfApples* offers the same interface as any other list, but operates on instances of type *Apple*.

Notations:

As ADTs provide an abstract view to describe properties of sets of entities, their use is independent from a particular programming language. We therefore introduce a notation here which is adopted from [3]. Each ADT description consists of two parts:

- **Data**: This part describes the structure of the data used in the ADT in an informal way.
- **Operations**: This part describes valid operations for this ADT, hence, it describes its interface. We use the special operation **constructor** to describe the actions which are to be performed once an entity of this ADT is created and **destructor** to describe the actions which are to be performed once an entity is destroyed. For each operation the provided *arguments* as well as *preconditions* and *postconditions* are given.

As an example the description of the ADT *Integer* is presented. Let *k* be an integer expression:

ADT *Integer* is
Data

A sequence of digits optionally prefixed by a plus or minus sign. We refer to this signed whole number as *N*.
Operations
constructor
Creates a new integer.
add(k)

Creates a new integer which is the sum of *N* and *k*.

Consequently, the *postcondition* of this operation is *sum* = *N*+*k*. Don't confuse this with assign statements as used in programming languages! It is rather a mathematical equation which yields ``true" for each value *sum*, *N* and *k* after *add* has been performed.

sub(k)
Similar to *add*, this operation creates a new integer of the difference of both integer values. Therefore the postcondition for this operation is*sum* = *N*-*k*.
set(k)
Set *N* to *k*. The postcondition for this operation is *N* = *k*.
...
end

The description above is a *specification* for the ADT *Integer*. Please notice, that we use words for names of operations such as ``add". We could use the more intuitive ``+" sign instead, but this may lead to some confusion: You must distinguish the operation ``+" from the mathematical use of ``+" in the postcondition. The name of the operation is just *syntax* whereas the *semantics* is described by the associated pre- and postconditions. However, it is always a good idea to combine both to make reading of ADT specifications easier.

Real programming languages are free to choose an arbitrary *implementation* for an ADT. For example, they might implement the operation *add* with the infix operator ``+" leading to a more intuitive look for addition of integers.

b. How the initialization of pointer can be defined?

Sol: The initializer is an = (equal sign) followed by the expression that represents the address that the pointer is to contain. The following example defines the variablestime and speed as having type **double** and amount as having type pointer to a **double**. The pointer amount is initialized to point to total:

```
double total, speed, *amount = &total;
```

The compiler converts an unsubscripted array name to a pointer to the first element in the array. You can assign the address of the first element of an array to a pointer by specifying the name of the array. The following two sets of definitions are equivalent. Both define the pointer student and initialize student to the address of the first element in section:

```
int section[80];
```

```
    int *student = section;
```

is equivalent to:

```
    int section[80];
    int *student = &section[0];
```

You can assign the address of the first character in a string constant to a pointer by specifying the string constant in the initializer.

The following example defines the pointer variable string and the string constant "abcd". The pointer string is initialized to point to the character a in the string"abcd".

```
    char *string = "abcd";
```

The following example defines weekdays as an array of pointers to string constants. Each element points to a different string. The pointer weekdays[2], for example, points to the string "Tuesday".

```
    static char *weekdays[ ] =
        {
         "Sunday", "Monday", "Tuesday", "Wednesday",
         "Thursday", "Friday", "Saturday"
        };
```

A pointer can also be initialized to null using any integer constant expression that evaluates to 0, for example char * a=0;. Such a pointer is a *null pointer*. It does not point to any object.

c. How dynamic memory management can be done in data structure.

Sol: String processing requires memory to be allocated for string storage. New strings may be input or created, old strings discarded, and strings in general may expand or contract during this processing. This requires some means of allocating storage in segments of variable size, and "recycling" unused space for new data.

The Forth language has only two structures for allocation and de-allocation: the dictionary, and the stack. Both of these are Last-In First-Out allocators, releasing memory only in the reverse order of its original allocation.

Since this is an unacceptable restriction, patternForth requires a dynamic memory manager.

1. Design

There are four design criteria for a memory allocator.

a) Efficiency of representation. Data should be stored in a form which can be efficiently processed by the end application.

b) Speed of allocation. The time involved to obtain a new segment of memory would preferably be finite and deterministic; ideally, short and invariant.

c) Speed of "recycling". Likewise, the time make a segment of memory available for re-use, would preferably be finite and deterministic; ideally, short and invariant.

d) Utilization of memory. The amount of memory which is made available for data should be maximized. This implies:

* memory should be "parceled out" in such a manner as to minimize "wasted space";

* the allocator should use as little storage as possible for its own internal data;

* the amount of memory which is made available through recycling should be maximized. Ideally, storage which is recycled should be completely re-usable. This is particularly significant since the allocate-recycle- allocate cycle will repeat endlessly.

A memory allocator should solve, or at least address, the problem of "fragmentation." This is the division of memory so as to leave small, unusable segments.

Q.3 a. What is the difference between a queue and a stack?

Sol: A queue is typically FIFO (priority queues don't quite follow that) while a stack is LIFO. Elements get inserted at one end of a queue and retrieved from the other, while the insertion and removal operations for a stack are done at the same end.

b.What is a pointer variable? Can we have multiple pointers to a variable? Explain Lvalue and Rvalue expression.

Sol: A variable which holds the address of another variable is called a pointer variable. In other words, pointer variable is a variable which holds pointer value (i.e., address of a variable.)

```
ex: int *p; /*declaration of a pointer variable */
int a=5;
p = &a;
```

Yes, we can have multiple pointers to a variable.

```
Ex: int *p , *q , *r;
int a=10;
p = &a;
q = &a;
r = &a;
```

Here, p ,q, r all point to a single variable 'a'.
An object that occurs on the left hand side of the assignment operator is called Lvalue. In other words, Lvalue is defined as an expression or storage location to which a value can be assigned.
Ex: variables such as a,num,ch and an array element such as a[0], a[i], a[i][j] etc.
Rvalue refers to the expression on right hand side of the assignment operator.
The R in Rvalue indicates read the value of the expression or read the value of the variable.
Ex: 5, a+2, a[2]+4, a++, --y etc.

c.What is the advantage of circular queue over ordinary queue? Mention applications of queue also.

Sol: Advantages of circular queue over ordinary queue:

• Rear insertion is denied in an ordinary queue even if room is available at the front end. Thus, even if free memory is available, we cannot access these memory locations. This is the major disadvantage of linear queues.
• In circular queue, the elements of a given queue can be stored efficiently in an array so as to “wrap around” so that end of the queue is followed by the front of queue.
• This circular representation allows the entire array to store the elements without shifting any data within the queue.

Applications of queues:

• Queues are useful in time sharing systems where many user jobs will be waiting in the system queue for processing. These jobs may request the services of the CPU, main memory or external devices such as printer etc. All these jobs will be given a fixed time for processing and are allowed to use one after the other. This is the case of an ordinary queue where priority is the same for all the jobs and whichever job is submitted first, that job will be processed.
• Priority queues are used in designing CPU schedulers where jobs are dispatched to the CPU based on priority of the job.

Q.4 a. List out any two applications of linked list and any two advantages of doubly linked list over singly linked list.

Sol: **Applications of linked list:**

• Arithmetic operations can be easily performed on long positive numbers.
• Manipulation and evaluation of polynomials is easy.
• Useful in symbol table construction(Compiler design).

Singly linked list:

1) Traversing is only in one direction.
2) While deleting a node, its predecessor is required and can be found only after traversing from the beginning of list.
3) programs will be lenghty and need more time to design.

Doubly Linked List:

1) Traversing can be done in both directions.
2) While deleting a node x, its predecessor can be obtained using link of node x. No need to traverse the list.
3) Using circular linked list with header, efficient and small programs can be written and hence design is easier.

b.Give an algorithm to insert a node at a specified position for a given singly linked list.

Sol: Algorithm insert_pos

```
        {
step 1: create a node and get the item from the user. Let the node be temp.
Temp = getnode;
info(temp) = item;
step 2: If the list is empty (first == NULL) and given position = 1, return temp;
first = temp;
step 3: if list is empty and pos != 1, then report invalid position.
Step 4: if pos = 1
link(temp) = first
first = temp;
step 5: When none of the above cases hold, find the appropriate position:
count = 1
prev = NULL
cur = first
while(cur != NULL and count != pos)
{
prev = cur
cur = link(cur)
count = count+1
}
step 6: If count == pos,
link(prev) = temp
link(temp) = cur
else
report invalid position.
}
```

c. Write a C program to copy one string to another, using pointers and without using library functions.

Sol:
```
#include<stdio.h>
void my_strcpy(char *dest, char *src)
{
/*Copy the string */
while(*src != '\0')
*dest++ = *src++;
/* Attach null character at end */
*dest = '\0';
}
void main()
{
char src[20] , dest[20];
printf("Enter some text\n");
scanf("%s",src);
my_strcpy(dest,src);
printf("The copied string = %s\n",dest);
}
```

Q.5a. Write the algorithm for bucket sort and give the complexity of bucket sort.

Sol: Bucket sort (a, n)
Here a is a linear array of integers with n elements, the variable digit count is used to store the number of digits in the largest number in order to control the number of passes to be performed.

```
Begin
find the largest number of the array
set digit count=no. of digits of the largest
no. for pass=1 to digit count by 1 do
initialize buckets
for i=1 to n-1 by 1 do
set digit=obtain digit no. pass of
a[i] put a[i] in bucket no. digit
```

```
increment bucket count for bucket no.
digit endfor
collect all the numbers from buckets in
order endfor
end
```

Complexity of bucket sort:
Suppose that the number of digits in the largest number of the given array is s and the number of passes to be performed is n. Then, the number of comparisons, f(n),needed to sort the given array are:
f(n)=n*s*10, where 10 is the decimal number base.
Though s is independent of n, but if s=n, then in worst case,
f(n)=O(n2). But on the other hand, if s=log10 n, then f(n)=O(n log10 n).
From the above discussion, we conclude that bucket sort performs well only when the number of digits in the elements are very small.

b. Write a Sub Algorithm to Find the Smallest Element in the array.
Sol: Smallest element(a, n, k, loc)
Here a is linear array of size n. This sub algorithm finds the location loc of smallest element among a[k-1],a[k+1],a[k+2]…a[n-1]. A temporary variable "small" is used to hold the current smallest element and j is used as the loop control variable.

```
Begin
set small=a[k-
1] set loc=k-1
for j=k to (n-1) by 1
do if(a[j]<small) then
set small = a[j]
set
loc=j endif
endfor
end
```

c. Explain Dijkstra's algorithm and Give mode of operation in Dijkstra's algorithm.

Sol: Djikstra's algorithm (named after its discover, E.W. Dijkstra) solves the problem of finding the shortest path from a point in a graph (the *source*) to a destination. It turns out that one can find the shortest paths from a given source to *all* points in a graph in the same time, hence this problem is sometimes called the **single-source shortest paths problem.**
Dijkstra's algorithm keeps two sets of vertices:
S the set of vertices whose shortest paths from the source have
already been determined *and*

V-S the remaining vertices.
The other data structures needed are:
d array of best estimates of shortest path to each vertex
pi an array of predecessors for each vertex
The basic mode of operation is:
1. Initialise **d** and **pi**,
2. Set **S** to empty,
3. While there are still vertices in **V-S**,
    i. Sort the vertices in **V-S** according to the current best estimate of their distance from the source,
    ii. Add **u**, the closest vertex in **V-S**, to **S**,
    iii. **Relax** all the vertices still in **V-S** connected to **u**

Q.6 a. What is Hashing? Explain any three hash functions.

b. Program to add a new node to the ascending order linked list.

Sol:
```
#include <conio.h>
#include <alloc.h>
/* structure containing a data part and link part */
struct node
{
int data ;
struct node *link ;
} ;
void add ( struct node **, int ) ;
void display ( struct node * ) ;
int count ( struct node * ) ;
void delete ( struct node **, int ) ;
void main( )
{
struct node *p ;
p = NULL ; /* empty linked list */
add ( &p, 5 ) ;
add ( &p, 1 ) ;
add ( &p, 6 ) ;
add ( &p, 4 ) ;
add ( &p, 7 ) ;
clrscr( ) ;
display ( p ) ;
printf ( "\nNo. of elements in Linked List = %d", count ( p ) ) ;
}
/* adds node to an ascending order linked list */
void add ( struct node **q, int num )
```

```
{
struct node *r, *temp = *q ;
r = malloc ( sizeof ( struct node ) ) ;
r -> data = num ;
/* if list is empty or if new node is to be inserted before the first node */
if ( *q == NULL || ( *q ) -> data > num )
{
*q = r ;
( *q ) -> link = temp ;
}
else
{
/* traverse the entire linked list to search the position to insert the
new node */
while ( temp != NULL )
{
if ( temp -> data <= num && ( temp -> link -> data > num ||
temp -> link ==
NULL ))
{
r -> link = temp -> link ;
temp -> link = r ;
return ;
}
temp = temp -> link ; /* go to the next node */
}
}
}
/* displays the contents of the linked list */
void display ( struct node *q )
{
printf ( "\n" ) ;
/* traverse the entire linked list */
while ( q != NULL )
{
printf ( "%d ", q -> data ) ;
q = q -> link ;
}
}
/* counts the number of nodes present in the linked list */
int count ( struct node *q )
{
int c = 0 ;
/* traverse the entire linked list */
while ( q != NULL )
{
q = q -> link ;
c++ ;
}
return c ;
}
```

c. Explain circularly linked lists and minimum spanning tree.

Sol: By ensuring that the tail of the list is always pointing to the head, we can build a circularly linked list. If the external pointer (the one in struct t_node in our implementation), points to the current "tail" of the list, then the "head" is found trivially via tail->next, permitting us to have either LIFO or FIFO lists with only one external pointer. In modern processors, the few bytes of memory saved in this way would probably not be regarded as significant. A circularly linked list would more likely be used in an application which required "round-robin" scheduling or processing.

Minimum spanning tree
If a cost, **cij**, is associated with each edge, **eij = (vi,vj)**, then the minimum spanning tree is the set of edges, **Espan**, forming a spanning tree, such that:
C = sum(cij | all **eij** in **Espan**) is a minimum.

Q.7 a. Write the process of selection sort.

Sol : Pass1.
1. Find the location loc of the smallest element in the entire array,
i.e. a[0],[1],a[2]…a[n-1].
2. Interchange a[0] & a[loc]. Then, a[0] is trivially sorted.
Pass2.
1. Find the location loc of the smallest element in the entire array,
i.e. a[1],a[2]…a[n-1].
2. Interchange a[1] & a[loc]. Then a[0], a[1] are sorted.
Pass k.
1. Find the location loc of the smallest element in the entire array, i.e.
a[k],a[k+1],a[k+2]…a[n-1].
2. Interchange a[k] & a[loc]. Then a[0],a[1],a[2],…a[k] are sorted.
Pass n-1.
1. Find the location loc of the smaller of the elements a[n-2],a[n-1].
2. Interchange a[n-2] & a[loc]. Then elements a[0],a[1],a[2]….a[n-1] are sorted.
3. sorted.

b. Program that implements depth first search algorithm.
Sol:

```
#include <stdio.h>
#include <conio.h>
#include <alloc.h>
#define TRUE 1
#define FALSE 0
#define MAX 8
struct node
{
int data ;
struct node *next ;
} ;
int visited[MAX] ;
void dfs ( int, struct node ** ) ;
struct node * getnode_write ( int ) ;
void del ( struct node * ) ;
void main( )
{
struct node *arr[MAX] ;
struct node *v1, *v2, *v3, *v4 ;
int i ;
clrscr( ) ;
v1 = getnode_write ( 2 ) ;
arr[0] = v1 ;
v1 -> next = v2 = getnode_write ( 3 ) ;
v2 -> next = NULL ;
v1 = getnode_write ( 1 ) ;
arr[1] = v1 ;
v1 -> next = v2 = getnode_write ( 4 ) ;
v2 -> next = v3 = getnode_write ( 5 ) ;
v3 -> next = NULL ;
v1 = getnode_write ( 1 ) ;
arr[2] = v1 ;
v1 -> next = v2 = getnode_write ( 6 ) ;
v2 -> next = v3 = getnode_write ( 7 ) ;
v3 -> next = NULL ;
v1 = getnode_write ( 2 ) ;
arr[3] = v1 ;
v1 -> next = v2 = getnode_write ( 8 ) ;
```

```
v2 -> next = NULL ;
v1 = getnode_write ( 2 ) ;
arr[4] = v1 ;
v1 -> next = v2 = getnode_write ( 8 ) ;
v2 -> next = NULL ;
v1 = getnode_write ( 3 ) ;
arr[5] = v1 ;
v1 -> next = v2 = getnode_write ( 8 ) ;
v2 -> next = NULL ;
v1 = getnode_write ( 3 ) ;
arr[6] = v1 ;
v1 -> next = v2 = getnode_write ( 8 ) ;
v2 -> next = NULL ;
v1 = getnode_write ( 4 ) ;
arr[7] = v1 ;
v1 -> next = v2 = getnode_write ( 5 ) ;
v2 -> next = v3 = getnode_write ( 6 ) ;
v3 -> next = v4 = getnode_write ( 7 ) ;
v4 -> next = NULL ;
dfs ( 1, arr ) ;
for ( i = 0 ; i < MAX ; i++ )
del ( arr[i] ) ;
getch( ) ;
}
void dfs ( int v, struct node **p )
{
struct node *q ;
visited[v - 1] = TRUE ;
printf ( "%d\t", v ) ;
q = * ( p + v - 1 ) ;
while ( q != NULL )
{
if ( visited[q -> data - 1] == FALSE )
dfs ( q -> data, p ) ;
else
q = q -> next ;
}
}
struct node * getnode_write ( int val )
{
struct node *newnode ;
newnode = ( struct node * ) malloc ( sizeof ( struct node ) ) ;
newnode -> data = val ;
return newnode ;
```

```
}
void del ( struct node *n )
{
struct node *temp ;
while ( n != NULL )
{
temp = n -> next ;
free ( n ) ;
n = temp ;
}
}
```

c. Program that creates random numbers in a given file.

Sol:
```
#include <stdio.h>
#include <conio.h>
#include <stdlib.h>
void main( )
{
FILE *fp ;
char str [ 67 ] ;
int i, noofr, j ;
clrscr( ) ;
printf ( "Enter file name: " ) ;
scanf ( "%s", str ) ;
printf ( "Enter number of records: " ) ;
scanf ( "%d", &noofr ) ;
fp = fopen ( str, "wb" ) ;
if ( fp == NULL )
{
printf ( "Unable to create file." ) ;
getch( ) ;
exit ( 0 ) ;
}
randomize( ) ;
for ( i = 0 ; i < noofr ; i++ )
{
j = random ( 1000 ) ;
fwrite ( &j, sizeof ( int ), 1, fp ) ;
printf ( "%d\t", j ) ;
}
fclose ( fp ) ;
printf ( "\nFile is created. \nPress any key to continue." ) ;
```

getch( ) ;
}

Q.8 a. What is a file pointer? Explain with syntax fopen(), fread() and fwrite() functions.

Sol: A file pointer fp is a pointer to a structure of type FILE.
FILE *fp;
• fopen() - The file should be opened before reading a file or before writing into
a file. The syntax to open a file for either read/write operation is shown below:
#include<stdio.h>
FILE *fp;
....
....
fp = fopen(char *filename , char *mode);
where, fp is a file pointer of type FILE.
Filename holds the name of the file to be opened.
Mode infoms the library function about the purpose of opening a file.
Mode = r,w,a,r+,w+,a+,rb,wb ....
Return values: The function may return the followingfile
pointer of type FILE if successful
NULL if unsuccessful.
Ex: fp = fopen("t.c","rb");
• **fread()** - This function is used to read a block of data from a given file. The prototype of fread() is shown below:
int fread(void *ptr, int size , int n , FILE *fp);
The parameters passed are:
Ø fp is a file pointer of an opened file from where the data has to be read.
Ø Ptr : The data read from the file should be stored in the memory. For this
purpose, it is required to allocate the sufficient memory and address of the first byte is stored in ptr.
Ø N is the number of items to be read from the file.
Ø Size is the length of each item in bytes.
Ex: #include<stdio.h>
void main()
{
int a[10];
....
....
fread(a,sizeof(int),5,fp);

....
}

• **fwrite()** - This function is used to write a block of data into a given file. The prototype of fwrite() is shown below:

int fwrite(void *ptr, int size, int n, FILE *fp); where,

▪ fp is a file pointer of an opened file into which the data has to be
written.

▪ Ptr : ptr points to the buffer containing the data to be written into the file.

▪ N is the number of bytes to be written into the file.

▪ Size is the length of each item in bytes.

Ex:
```
#include<stdio.h>
void main()
{
int a[8] = { 10,20,30,40,50,60,70,80};
........
.........
fwrite(a,sizeof(int),5,fp);
.........
..........
}
```

b. How do you define a data structure? How is stack a data structure? Give a C program to construct a stack of integers and perform all the necessary operations on it.

Sol: A data structure is a method of storing data in a computer so that it can be used efficiently. The data structures mainly deal with :

• The study of how data is organised in the memory.

• How efficiently can data be stored in the memory.

• How efficiently data can be retrieved and manipulated.

• The possible ways in which different data items are logically related.

A stack is a special type of a data structure where elements are inserted from one end and elements are deleted from the same end. This position from where elements are inserted and from where elements are deleted is called top of stack.

Using this approach, the last element inserted is the first element to be deleted out, and hence stack is also called Last In First Out(LIFO) data structure.

The various operations that can be performed on stacks are :

• push – Inserting an element on top of stack

• pop – Deleting an element from the top of stack

• display – Display contents of stack

Since a stack satisfies all the requirements mentioned in the definition of a data structure, stack as a type of data structure.

```
/* C program to construct a stack of integers */
#include<stdio.h>
#include<conio.h>
```

```
#include<process.h>
void push( int item, int *top, int s[])
{
/*check for overflow of stack */
if(*top == stack_size-1)
{
printf("Stack overflow \n");
return;
}
*top = *top+1;
s[*top] = item;
}
int pop(int *top, int s[])
{
int item_deleted;
/*stack underflow */
if(*top == -1) return 0;
item_deleted = s[*top];
*top -= 1;
return item_deleted;
}
void display(int top, int s[])
{
int i;
/* is stack empty */
if(top==-1)
{
printf("Stack is empty\n");
```

```
return;
}
/*display contents of stack */
printf("Contents of stack are\n");
for(i=0 ; i<=top ; i++)
printf("%d\n",s[i]);
}
void main()
{
int top = -1;
int s[10];
int item;
int item_deleted;
int choice;
clrscr();
```

```
while(1)
{
printf("1: Push 2: Pop\n");
printf("3: Display 4: Exit\n");
printf("Enter your choice\n");
scanf("%d",&choice);
switch(choice)
{
case 1: printf("Enter the item to be inserted\n");
scanf("%d",&item);
push(item , &top , s);
break;
case 2: item_deleted = pop(&top , s);
if(item_deleted == 0)
printf("Stack is empty\n");
else
printf("Item deleted = %d\n",item_deleted);
break;
case 3: display(top,s);
break;
default: exit(0);
}
}
}
```

c. Discuss about representation of a graph into memory.

Q.9 a. Explain the traversal of a binary tree
b. What is binary search tree? Explain the insertion and deletion of an element into binary search tree using suitable example.
c. Write advantages and disadvantages of tree data structures.

Text Book

1. Data Structures using C & C++, Rajesh K. Shukla, Wiley India. 2009