Q 2 (a) Define storage class and its functions. Explain in detail scope, storage allocation and purpose of each storage class.

Answer

'Storage' refers to the scope of a variable and memory allocated by compiler to store that variable. Scope of a variable is the boundary within which a variable can be used. Storage class defines the scope and lifetime of a variable.

From the point view of C compiler, a variable name identifies physical location from a computer where variable is stored. There are two memory locations in a computer system where variables are stored as: Memory and CPU Registers

Functions of storage class:

To determine the location of a variable where it is stored?

Set initial value of a variable or if not specified then setting it to default value.

Defining scope of a variable.

To determine the life of a variable.

Types of Storage Classes:

Storage classes are categorized in 4 (four) types as,

- Automatic Storage Class
- Register Storage Class
- Static Storage Class
- External Storage Class

Storage type	**Created**	**Initialized**	**Scope**	**purpose**
auto	Each time the function or block is called	Can be initialised at the time of declaration	With in the block or function	As variable with in a block
static	First time when the function is	Initialised at the time of declaration	With in the block or	As variables which retain

	called		function	value even after the termination of function.
register	same as auto	same as auto	same as auto	As most frequently used variables
Extern	created in the file where it has been declared as global	initialised in the file where it has been declared as global	Both the files where declared as global and where declared as extern	as variables used by multiple files.

Q2 (b) Differentiate between Static and Dynamic memory allocation

Answer

S.No.	STATIC MEMORY ALLOCATION	DYNAMIC MEMORY ALLOCATION
1.	Memory is allocated before the execution of the program begins. (During Compilation)	Memory is allocated during the execution of the program.
2.	No memory allocation or deallocation actions are performed during Execution.	Memory Bindings are established and destroyed during the Execution.
3.	Variables remain permanently allocated.	Allocated only when program unit is active.
4.	Implemented using stacks and heaps.	Implemented using data segments.
5.	Pointer is needed to accessing variables.	No need of Dynamically allocated pointers.
6.	Faster execution than Dynamic.	Slower execution than static.
7.		

	More memory Space required.	Less Memory space required.

Q3 (a) Define and explain Self Referential Structures in detail. Give suitable example.

Answer

Self Referential Structures:

A structure may have a member whose is same as that of a structure itself. Such structures are called self-referential. A self-referential structure is one which contains a pointer to its own type. Self-Referential Structure are one of the most useful features. They allow to create data structures that contains references to data of the same type as themselves. e.g.

```
struct student
{
int roll;
float marks;
int subject;
struct student *ptr;
};
```

In the above example structure student has a pointer ptr which points to another student structure.Self-referential Structures are used in the area of linear data structures such as linked list, stack, Queue and non-linear data structures such as trees, graphs etc.

Example: Single linked list is implemented using following data structures

```
struct node {

int value;

struct node *next;

   };
```

Q3 (b) How a union is different from a struture? Explain with an example.

Answer

Structure	**Union**
i. Access Members	
We can access all the members of	Only one member of union can be accessed at

structure at anytime.	anytime.
ii. Memory Allocation	
Memory is allocated for all variables.	Allocates memory for variable which variable require more memory.
iii. Initialization	
All members of structure can be initialized	Only the first member of a union can be initialized.
iv. Keyword	
'struct' keyword is used to declare structure.	'union' keyword is used to declare union.
v. Syntax	
struct struct_name { structure element 1; structure element 2; ---------- ---------- structure element n; }struct_var_nm;	**union** union_name { union element 1; union element 2; ---------- ---------- union element n; }union_var_nm;
vi. Example	
struct item_mst {	**union** item_mst {

int rno; char nm[50]; }it;	int rno; char nm[50]; }it;

Q3 (c) List various file opening modes available in 'C'.

Answer

Modes of opening a file:

Mode	Purpose
"r"	Open a file for reading. If file does not exist, NULL is returned.
"w"	Open a file for writing. If the file does not exist a new file is created. If the file exists, the new contents overwrite the previous contents.
"r+"	Open the file for both reading and writing. If file does not exist, NULL is returned.
"w+"	Open the file for both reading and writing. If file does not exist, the new contents overwrites the previous contents
"a"	Open a file for appending. If the file exists, the new data is written at the end of the file else the new file is created.
"a+"	Open the file for reading and appending. If the file does not exist, a new file is created.

Q4 (a) Write an algorithm to search an element using Binary Search method.

Answer

Binary search:
BINARY(A,LB,UB,ITEM,LOC)

1. Set BEG:=LB,END=UB and MID=INT((BEG+END)/2)
2. Repeat steps 3 &4 while BEG<=END and DATA[MID]≠ ITEM
3. If ITEM < DATA[MID] then

   Set END: =MID-1

else

Set BEG: =MID+1

4. Set MID:=INT((BEG+END)/2)

5. if A[MID]=ITEM then

Set LOC: =MID

else

Set LOC: =NULL

6. Exit

A is a sorted array with lower bound LB and upper bound UB and ITEM is given item of information. Variables BEG, END and MID denote respectively the beginning, end and middle locations of a segment of elements of A.

Q.5 (a) Write a 'C' routine that describes various operations on stack.

Answer

PUSH & POP IN STACK

```
#include<stdio.h>
#include<conio.h>
struct
{
int a[5],top;
}q;
void main()
{
int z,i,t;
clrscr();
for (i=0;i<=5;i++)
{
scanf("%d",&t);
push(t);
}
for(i=1;i<=5;i++)
{
z=pop();
printf("%d",z);
}}
push(int t)
{
q.a[q.top]=t;
return(q.top++);
```

```
}
pop()
{
q.top--;
return(q.a[q.top]);}
```

Q5 (b) what is the advantage of circular queue over linear queue? Write C routines for inserting and deleting an element from the linear queue.

Answer

Circular queue have less memory consumption as compared to linear queue because while doing insertion after deletion operation in a linear queue, it allocate an extra space but in circular queue the first is used as it comes immediate after the last.

INSERT AND DELETE ITEMS IN A QUEUE

```
#include<stdio.h>
struct queue
{ int a[5],front,rear;
} q;
void main()
{
int z,i,t;
for(i=1;i<=5;i++)
{
scanf("%d",&t);
insert(t);
}
for(i=1;i<=5;i++)
{
z=delete();
printf("%d",z);
}
}
insert(int t)
{
q.a[q.front]=t;
q.front++;
}
delete()
{
int p;
p=q.a[q.rear];
q.rear++;
return(p);
}
```

Q6 (a) Explain the a singly linked list? Write a C program for different operations that can be performed on singly linked list.

6 a. Singly Linked List Singly linked list is the most basic linked data structure. In this the elements can be placed anywhere in the heap memory unlike array which uses contiguous locations. Nodes in a linked list are linked together using a next field, which stores the address of the next node in the next field of the previous node i.e. each node of the list refers to its successor and the last node contains the NULL reference. It has a dynamic size, which can be determined only at run time.

Basic operations of a singly-linked list are:

Insert – Inserts a new element at the end of the list.
Delete – Deletes any node from the list.
Find – Finds any node in the list.
Print – Prints the list.

Functions

1. Insert – This function takes the start node and data to be inserted as arguments. New node is inserted at the end so, iterate through the list till we encounter the last node. Then, allocate memory for the new node and put data in it. Lastly, store the address in the next field of the new node as NULL.

2. Delete - This function takes the start node (as pointer) and data to be deleted as arguments. Firstly, go to the node for which the node next to it has to be deleted, If that node points to NULL (i.e. pointer->next=NULL) then the element to be deleted is not present in the list. Else, now pointer points to a node and the node next to it has to be removed, declare a temporary node (temp) which points to the node which has to be removed. Store the address of the node next to the temporary node in the next field of the node pointer (pointer->next = temp->next). Thus, by breaking the link we removed the node which is next to the pointer (which is also temp). Because we deleted the node, we no longer require the memory used for it, free() will deallocate the memory.

3. Find - This function takes the start node (as pointer) and data value of the node (key) to be found as arguments. First node is dummy node so, start with the second node. Iterate through the entire linked list and search for the key. Until next field of the pointer is equal to NULL, check if pointer->data = key. If it is then the key is found else, move to the next node and search (pointer = pointer -> next). If key is not found return 0, else return 1.

4. Print - function takes the start node (as pointer) as an argument. If pointer = NULL, then there is no element in the list. Else, print the data value of the node (pointer->data) and move to the next node by recursively calling the print function with pointer->next sent as an argument.

Performance:

1. The advantage of a singly linked list is its ability to expand to accept virtually unlimited number of nodes in a fragmented memory environment.
2. The disadvantage is its speed. Operations in a singly-linked list are slow as it uses sequential search to locate a node.

Single Linked List - C Program source code

```
#include<stdio.h>
#include<stdlib.h>
typedef struct Node
{
	int data;
	struct Node *next;
}node;
void insert(node *pointer, int data)
{
	/* Iterate through the list till we encounter the last node.*/
	while(pointer->next!=NULL)
	{
		pointer = pointer -> next;
	}
	/* Allocate memory for the new node and put data in it.*/
	pointer->next = (node *)malloc(sizeof(node));
	pointer = pointer->next;
	pointer->data = data;
	pointer->next = NULL;
}
int find(node *pointer, int key)
{
	pointer =  pointer -> next; //First node is dummy node.
	/* Iterate through the entire linked list and search for the key. */
	while(pointer!=NULL)
	{
		if(pointer->data == key) //key is found.
		{
			return 1;
		}
		pointer = pointer -> next;//Search in the next node.
	}
	/*Key is not found */
	return 0;
}
```

```
void delete(node *pointer, int data)
{
	/* Go to the node for which the node next to it has to be deleted */
	while(pointer->next!=NULL && (pointer->next)->data != data)
	{
		pointer = pointer -> next;
	}
	if(pointer->next==NULL)
	{
		printf("Element %d is not present in the list\n",data);
		return;
	}
	/* Now pointer points to a node and the node next to it has to be removed */
node *temp;
	temp = pointer -> next;
	/*temp points to the node which has to be removed*/
	pointer->next = temp->next;
	/*We removed the node which is next to the pointer (which is also temp) */
	free(temp);
	/* Beacuse we deleted the node, we no longer require the memory used for it .
	   free() will deallocate the memory.
	 */
	return;
}
void print(node *pointer)
{
	if(pointer==NULL)
	{
		return;
	}
	printf("%d ",pointer->data);
	print(pointer->next);
}
int main()
{
	/* start always points to the first node of the linked list.
	   temp is used to point to the last node of the linked list.*/
	node *start,*temp;
	start = (node *)malloc(sizeof(node));
	temp = start;
	temp -> next = NULL;
	/* Here in this code, we take the first node as a dummy node.
	   The first node does not contain data, but it used because to avoid handling special
cases
	   in insert and delete functions.
	 */
```

```
printf("1. Insert\n");
printf("2. Delete\n");
printf("3. Print\n");
printf("4. Find\n");
while(1)
{
	int query;
	scanf("%d",&query);
	if(query==1)
	{
		int data;
		scanf("%d",&data);
		insert(start,data);
	}
	else if(query==2)
	{
		int data;
		scanf("%d",&data);
		delete(start,data);
	}
	else if(query==3)
	{
		printf("The list is ");
		print(start->next);
		printf("\n");
	}
	else if(query==4)
	{
		int data;
		scanf("%d",&data);
		int status = find(start,data);
		if(status)
		{
			printf("Element Found\n");
		}
		else
		{
			printf("Element Not Found\n");

		}
	}
}
```

Q6 (b) what is the main advantage and disadvantage of using Linked List over an Array?

Answer

The principal benefit of a linked list over a conventional array is that the list elements can easily be inserted or removed without reallocation or reorganization of the entire structure because the data items need not be stored contiguously in memory or on disk. Linked lists allow insertion and removal of nodes at any point in the list, and can do so with a constant number of operations if the link previous to the link being added or removed is maintained during list traversal.

On the other hand, simple linked lists by themselves do not allow random access to the data, or any form of efficient indexing. Thus, many basic operations — such as obtaining the last node of the list (assuming that the last node is not maintained as separate node reference in the list structure), or finding a node that contains a given datum, or locating the place where a new node should be inserted — may require scanning most or all of the list elements

Q7 (a) Define and explain doubly linked list. Write a 'C' routine to insert a node after the specified node in a doubly linked list.

Answer

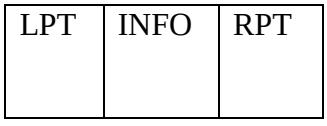
Doubly Linked List

Doubly-linked list is a more sophisticated form of linked list data structure. Each node of the list contain two references (or links) – one to the previous node and other to the next node. The previous link of the first node and the next link of the last node points to NULL. In comparison to singly-linked list, doubly-linked list requires handling of more pointers but less information is required as one can use the previous links to observe the preceding element. It has a dynamic size, which can be determined only at run time.

Performance

1. The advantage of a doubly linked list is that we don't need to keep track of the previous node for traversal or no need of traversing the whole list for finding the previous node.

2. The disadvantage is that more pointers needs to be handled and more links need to updated



```
void insert_given_node()
{
struct node *ptr,*cpt;
int m;
ptr=(struct node * ) malloc(size of (struct node));
if (ptr==NULL)
{
printf("OVERFLOW");
return;
}
printf("Input new node information");
scanf("%d",&ptr ->info);
printf("Input node information after which insertion is to be done");
scanf("%d",&m);
cpt=first;
while(cpt->info!=m)
cpt=cpt->rpt;
tpt=cpt->rpt;
cpt->rpt=ptr;
ptr->lpt=cpt;
ptr->rpt=tpt;
tpt->lpt=ptr;
printf("\n Insertion is complete");
```

}

Q7 (b) Define and explain Binary Search Tree (BST). Write 'C' function for counting the number of nodes in a BST.

Answer

Binary Search Tree In computer science, a **binary search tree** (**BST**) is a node based binary tree data structure which has the following properties:

- The left subtree of a node contains only nodes with keys less than the node's key.
- The right subtree of a node contains only nodes with keys greater than the node's key.
- Both the left and right subtrees must also be binary search trees.

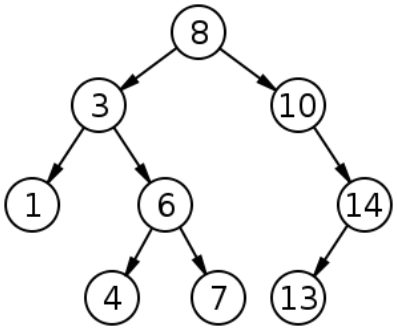
From the above properties it naturally follows that:

- Each node (item in the tree) has a distinct key.

Generally, the information represented by each node is a **record** rather than a single data element. However, for sequencing purposes, nodes are compared according to their **keys** rather than any part of their their associated records.

The major advantage of binary search trees over other data structures is that the related sorting algorithms and search algorithms such as in-order traversal can be very efficient.

Binary search trees are a fundamental data structure used to construct more abstract data structures such as sets, multisets, and associative arrays.



How to count number of nodes in a Binary Search Tree?

To count the number of nodes in a given binary search tree, the tree is required to be traversed recursively until a leaf node is encountered. When a leaf node is encountered, a count of 1 is returned to its previous activation (which is activation for its parent), which takes the count returned from both the children's activation, adds 1 to it, and returns this value to the activation of its parent. This way, when the activation for the root of the binary search tree returns, it returns the count of the total number of the nodes in the binary tree.

```
int count(struct tnode *p)
                {


 if(p==NULL)
                return(0);
        else
              if( p->lchild == NULL && p->rchild == NULL)
                return(1);
              else
                return(1 + (count(p->lchild) + count(p->rchild)));
}
```

Q 8 (a) Define and explain Graph Traversal. Describe in detail various Graph Traversal Strategies with the help of example.

Answer

Graph Traversal

To traverse a graph is to process every node in the graph exactly once. Because there are many paths leading from one node to another, the hardest part about traversing a graph is making sure that you do not process some node twice. There are two general solutions to this difficulty:

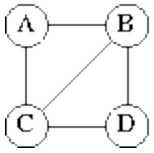
1. When you first encounter a node, mark it as REACHED. When you visit a node, check if it's marked REACHED; if it is, just ignore it. This is the method our algorithms will use.
2. When you process a node, delete it from the graph. Deleting the node causes the deletion of all the arcs that lead to the node, so it will be impossible to reach it more than once.

   At the heart of our traversal algorithm is a list of nodes that we have reached but not yet processed. We will call this the READY list.

General Traversal Strategy

1. Mark all nodes in the graph as NOT REACHED.
2. pick a starting node. Mark it as REACHED and place it on the READY list.
3. pick a node on the READY list. Process it. Remove it from READY. Find all its neighbours: those that are NOT REACHED should marked as REACHED and added to READY.
4. repeat 3 until READY is empty.

Example:



- Step 1: A = B = C = D = NOT REACHED.
- Step 2: READY = {A}. A = REACHED.
- Step 3: Process A. READY = {B, C}. B = C = REACHED.
- Step 3: Process C. READY = {B, D}. D = REACHED.
- Step 3: Process B. READY = {D}.
- Step 3: Process D. READY = {}.

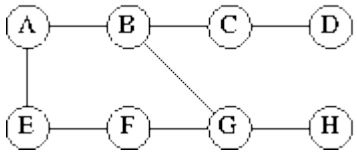
In fact this will traverse only a connected graph. To traverse a graph that might be unconnected, whole procedure is repeated until all nodes are marked as REACHED.

There are two *choice points* in this algorithm: in step 2, how do we pick the initial starting node, and in step 3 how do we pick a node from READY? The answer is, it

depends on what you are trying to accomplish. If all you want to do is print out nodes, or count them, or do any other processing that is order-independent, then any selection will do. The two most common traversal patterns are *breadth-first* traversal and *depth-first* traversal. In breadth-first traversal, READY is a QUEUE, not an arbitrary list. Nodes are processed in the order they are reached (FIFO). This has the effect of processing nodes according to their distance from the initial node. First, the initial node is processed. Then all its neighbours are processed. Then all of the neighbours' neighbours etc.

In depth-first traversal, READY is a STACK; the most recently reached nodes are processed before earlier nodes.

Let us compare the two traversal orders on the following graph:



Initial Steps: READY = [A]. Process A. READY = [B,E]. Process B.

It is at this point that two traversal strategies differ. Breadth-first adds B's neighbours to the *back* of READY; depth-first adds them to the *front*:

Breadth First:

- READY = [E,C,G].
- process E. READY = [C,G,F].
- process C. READY = [G,F,D].
- process G. READY = [F,D,H].
- process F. READY = [D,H].
- process D. READY = [H].
- process H. READY = [].

Depth First:

- READY = [C,G,E].
- process C. READY = [D,G,E].
- process D. READY = [G,E].
- process G. READY = [H,F,E].
- process H. READY = [F,E].
- process F. READY = [E].

Text Book

C& Data Structures, P.S. Deshpande and O.G. Kakde, Dreamtech Press, 2007