Appendix D

Solved Question Papers

C Programming and Data Structures

(May/June 2006)

SET 1

1. (a) What is the difference between signed integer and unsigned integer in terms of memory and range?

Type	Memory	Range
1. Signed Integers	In memory it occupies 16 bits.	these are Ranging from -32,678 to 32,677.
2. Unsigned Integers	In memory it occupies 16 bits.	these are Ranging from 0 to 65,535.

For a 16 bit machine, a signed integer uses one for sign and 15 bits for the magnitude of the number. Unlike signed integer, unsigned integers use all the bits for the magnitude of the number and are always positive. Therefore, for a 16 bit machine, the range of unsigned integer numbers will be from 0 to 65,535.

Signed integers are declared as signed int and unsigned integers are declared as unsigned int. Both of these two contain integer storage classes of 3 types, namely, short int, int, long int.

(b) List the entire data types in 'C'. What is the size of these data types?

Storage representations and the machine instructions to handle constants differ from machine to machine. The variety of data types available allow the programmer to application as well as the machine.

'C' supports the following four classes of data types:

1. Primary or fundamental data types
2. Derived data types

3. User-defined data types
4. Empty data types

Primary data types There are four fundamental data types, namely integer (int), character (char), floating point (float) double floating point (double).

Derived data types Generally arrays, functions, structures and pointer will come under the category of derived data types.

User defined data types Type definition (type def) enumerated data type (enum) are the user defined data types. (Structures and unions also come under this category).

Size of data types:-

Type	Size (bits)
1. Char or signed char	8
2. Unsigned char	8
3. Int or signed char	16
4. Unsigned int	16
5. Short int (or) Signed short int	8
6. Long int (or) Signed long int	32
7. Float	32
8. Double	64
9. Long double	80

2. (a) Distinguish between getchar and scanf functions for reading strings.

Getchar:-Reading a single character can be done by using the function getchar.

Syntax:- variable_name = getchar ( );

Variable_name is a valid 'c' name that has been declared as char type. When this statement is encountered, the computer waits until a key is pressed and then assigns this character as a value to getchar function. Since getchar is used on the right hand side of an assignment statement, the character value of getchar is in turn assigned to the variable_name on the left. The getchar function may be called successively to read the characters contained in a line of text. Getchar accepts space character.

Scanf ("control strings",arg1, arg2, ………… argn);

The control string specifies the field format in which the data is to be entered and the arguments arg 1, arg 2, arg *n* specify the address of locations where the data is stored. Scanf does not accept space character.

(b) Write a program to count the number of words, lines and characters in a text.

```
#include<stdio.h>
main( )
{
char line[80],ctr;
int  I,c,end=0,characters=0,words=0,lines=0;
printf(“\n key in text”);
printf(“give one space after each word \n”);
printf(“when completed press ‘enter’ \n\n”);
while (end==0)
{
c=0;
while((ctr=getchar())!=‘\n’)
line[c++]=ctr;
line[c]=‘\0’;
if(line[0]==‘\0’)
break;
else
{
words++;
for(i=0;line[i]!=‘\0’;i++)
if(line[i]==’ ‘ ll line[i]==‘\t’)
words++;
}
lines=lines+1;

characters=characters+strlen(line);

}

printf("\n");
printf("\n number of lines=%d", lines);
printf("\n number of words=%d",words);
printf("\n number of characters=%d",characters);
}
```

3. (a) What is a pointer? List out the reasons for using pointers.

Pointer is a variable which holds the address of another variable, i.e. a pointer is, therefore, nothing but a variable that contains an address which is a location of another variable in memory. Since a pointer is a variable, its value is also stored in the memory in another location.

Example:

variable	value	address
quantity	179	5000
p	5000	5048

Let us assign the address of 'quantity' to a variable '*p*' which is called a "pointer".

Reasons for using pointers are as follows:

1. A pointer enables us to access a variable that is defined outside the function.
2. Pointers are more efficient in handling the data tables.
3. Pointers reduce the length and complexity of a program.
4. They increase the execution speed.
5. The use of a pointer array to character strings results in saving a data storage space in memory.

(b) Write a 'C' program to illustrate the use of indication operator "*" to access the value pointed by a pointer.

Key board in which the "Employee" structure consists of employee name, code, designation and salary. Construct an array of structures that stores '*n*' employee's information and write a program to carry out operations like inserting a new entry, deleting entry.

```
/*accessing variables using pointers*/

#include<stdio.h>

main( )

{
```

```
int x,y;

int *ptr;

x=10 ;

ptr=&x ;

y=*ptr ;

printf(“ value of x is %d\n\n”,x);
printf(“%d is stored at address %u\n”,x,&x);
printf(“%d is stored at address %u\n”,*&x,&x);
printf(“%d is stored at address  %u\n”,*ptr,ptr);
printf(“%d is stored at address   %u\n”,y,&*ptr);
printf(“%d is stored at address  %u\n”,ptr,&ptr);
printf(“%d is stored at address  %u\n”,y,&y);
*ptr=25;

printf(“\n  now x=%d\n”,x);

}
```

4. **Write a C program to read the information from the keyboard in which the “Employee” structure consists of employee name, code, designation and salary. Construct an array of structures that stores *n* employees information and write a program to carry out operations like inserting a new entry, deleting entry.**

```
#include<stdio.h>

#include<string.h>

Struct employee

{

char ename[20];

int code;

char desi[20];

float salary;
```

```
};
main()
{
int l,j,n,char n[20];
struct employee a[30],*B;    /*pointer for new entry*/
printf("enter number of employees");
scanf("%d",&n);
printf("enter employees name, code, designation,
salary\n");
for(i=0;i<n;i++)
{
scanf("%s%d%s%f",a[i].ename,&a[i].code],a[i].desi,&a[i].salary);
}
/*to insert a new entry*/
printf("To insert a new entry");
B=realloc(sizeof(struct employee));
printf("enter employee name,code,designation and salary");
scanf("%s%d%s%f",*B.enam,e,&*B.code,*B.desi,&*B.salary);
/*To delete an entry*/
printf("enter employee name");
scanf("%s",n);
for( i=0;i<n;i++)
{
if(strcmp(n,a[i].ename)=0)
{
for( j=0;j<n-1n;j++)
{
a[ j]=a[ j+1];
}

}
}
a[j]=*B;
if(strcmp(n,*B.ename)=0)
{
free(B);
}
}
```

5. (a) Explain the command line arguments. What are the syntactic constructs followed in 'C'.

Command line argument is the parameter supplied to a program when the program is invoked.This parameter may represent a file name the program should process. For example, if we want to execute a program to copy the contents of a file named X_FILE to another one name Y_FILE then we may use a command line like

```
c>program X_FILE Y_FILE
```

Program is the file name where the executable code of the program is stored. This eliminates the need for the program to request the user to enter the file names during execution.

The 'main' function can take two arguments called argc, argv and information contained in the command line is passed on to the program to these arguments, when 'main' is called up by the system. The variable argv is an argument vector and represents an array of characters pointers that point to the command line arguments. argc is an argument counter that counts the number of arguments on the command line. The size of this array is equal to the value of argc. In order to access the command line arguments, we must declare the 'main' function and its parameters as follows:

```
main(argc,argv)
int argc;
char *arg[ ];
{
……….
……….
}
```

Generally arg[0] represents the program name.

(b) Write a ‘C’ program to read the input file from command prompt, using command line arguments.

```
#include<stdio.h>
main(int arg,char *argv[])
{
FILE *fp;
int l;
char word[15];
fp=fopen(arg[1],“w”);   /*to open file with name agr[1]*/
printf(“\n number of lines in command line = %d\n\n”,argc);
for(i=2;i<argc;i++)

fprintf(fp,””%s”,argv[i]);     /*write to file argv[1]*/
fclose(fp);

/*writing contents of the file to screen*/
printf(“contents of %s file \n\n”,argv[1]);

fp=fopen(argv[1],“r”);
for(i=2;i<argc;i++)
{
fscanf(fp,”%s”,word);
printf(“%s”,word);
}

fclose(fp);
printf(“\n\n”);
/*writing the arguments from memory*/
for(i=0;i<argc;i++)
printf(“%*s\n”,i*5,argv[i]);
}
```

6. Define a data structure. What are the different types of data structures? Explain each of them with suitable example.

Data structure is the Mathematical or logical organization of data

Data structure operations are

1. Insertion
2. Deletion
3. Search
4. Sorting
5. Traversing

There are two types of data structures:

1. Linear data structure:

If there exists a linear relationship between elements present in data structure, then the data structure is known as linear data structure.

These are mainly arrays, stacks, and queues.

ARRAYS: An array is a group of related data items that share a common name.

ex:-salary[10];

In this example 'salary' represent a set of salaries of a group of employees. A particular number is indicated by writing a number called Index number of subscript in brackets after the array name. The example salary[10] represent the salary of the tenth employee.

STACKS: It is a linear data structure in which insertion and deletion takes place from only one end called top of the stack.

Example: Insert 10, 2, 15, 9, 6 with size = 4.

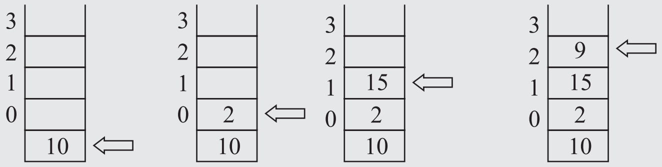

Fig. Set 1.6.(a)

In Fig. Set 1.6(a) the "arrow" mark indicates the current element that is inserted. If we try to insert '6' we come across overflow condition, i.e, overflow = size–1.

Delete the elements from above stack will be

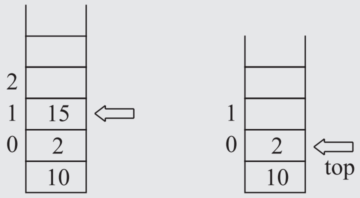

Fig. Set 1.6 (b)

If we try to delete after top Fig. set 1.6 (b) = –1, we come across under flow.

Stack is also known an LIFO (LAST-IN-FIRST-OUT) or pile or push down list.

QUEUES: It is a non linear data structure in which insertion takes place at the rear end and the deletion takes place at the front end.

Example: Insert 10, 33 with size = 5.

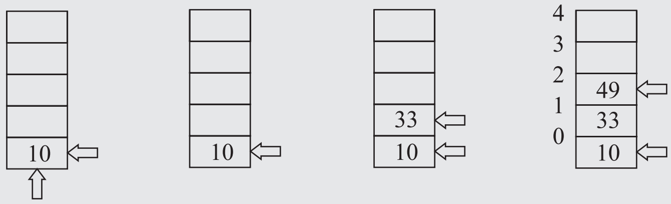

Fig. set 1.6 (c)

Here the first arrow on the right side in queue I represents the front and the arrow on the down represents the rear side.

If we want to insert elements more than the given size, then the condition is called overflow condition, i.e. rear = size–1.

Deletion of elements from above queue will be:

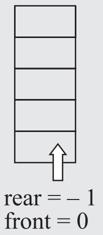

Fig. Set 1.6 (d)

After deleting 10, if we try to delete another element it will come across under flow condition, i.e. rear = –1 and front = 0.

2. NON LINEAR DATA STRUCTURES:

If there exists random relationship between the elements of a data structure, then that data structure is called non-linear data structure.

Trees and Graphs are known as non-linear data structures.

TREE: A tree is either empty or it may have a special node or a set of branches. Each branch in turn represents a tree.

Example: Fig. set 1.6 (e)

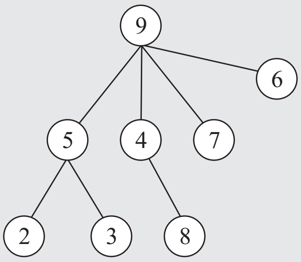

Fig. set 1.6(e)

GRAPHS: Set of vertices and set of edges combined together is known as graph. It is denoted by (G (V, E)).

Example: Fig. set 1.6(f)

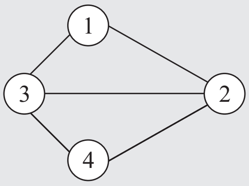

Fig. set 1.6(f)

V = {1, 2, 3, 4}

E = {(1,2), (1,3), (3,4), (2,4), (2,3)}

7. Write a ‘C’ program to insert and delete the elements from circular doubly linked list.

```
/* Circular linked list */

#include<stdio.h>

#include<alloc.h>

#define null 0

struct linked_list

{
```

```
struct linked_list *pre;

int data;

struct linked_list *next;

}*first,*fresh,*ptr,*last;

typedef struct linked_list node;

main()

{

int ch,a;

clrscr();

while(1)

{

printf("\n   MIAN MENU    \n");

printf("1.insert element\n2.delete element\n3.view
contents\n");

printf("4.exit\n");

printf("enter your choice\n");

scanf("%d",&ch);

switch(ch)

{

case 1:fresh=malloc(sizeof(node));
       printf("enter the element to be inserted\n");
       scanf("%d",&fresh->data);
       printf("where do you want to insert\n");
       printf("1.begining\n2.end\n3.middle\n");
       scanf("%d",&ch);
       switch(ch)
       {
```

```
case 1:
    Ibegin();      break;
case 2:
    Iend(); break;
case 3:
    printf("enter position\n");
    scanf("%d",&a);
    Imiddle(a);
    break;
    }

    break;

case 2:
    printf("where do you want to delete\n");
    printf("1.begining\n2.end\n3.required position\n");
    scanf("%d",&ch);
    switch(ch)
    {

    case 1:
          Dbegin();
          break;
    case 2:
          Dend();
          break;
    case 3:
          printf("enter position\n");
          scanf("%d",&a);
          Dmiddle(a);
          break;
    }

    break;

case 3:
          view();break;
```

```
case 4:
          exit(1);       break;
    default:
          printf("wrong choice\n");    break;
}
}
getch();
}
/* Insertion function*/
lbegin()
{
if(first==null)
{
first=fresh;
first->pre=first;
first->next=first;
last=first;
}
else
{
fresh->next=first;
first->pre=fresh;
first=fresh;
last->next=first;
}
}
lend()
{
if(first==null)
{
first=fresh;
first->pre=first;
first->next=first;
last=first;
```

```
}
else
{
fresh->next=first;
last->next=fresh;
fresh->pre=last;
last=fresh;
first->pre=last;
}
}
Imiddle(int n)
{
int i=1;
if(first==null)
{
first=fresh;
first->pre=first;
first->next=first;
last=first;
}
else
{
ptr=first;
while(ptr->next!=first)
i++,ptr=ptr->next;
if(i<=n)
Iend();
else if(i>n)
{
for(i=1;i<n;i++)
ptr=ptr->next;
fresh->next=ptr->next;
ptr->next=fresh;
fresh->pre=ptr;
fresh->next->pre=fresh;
}
}
}

/*   Deletion function   */
Dbegin()
{
ptr=first;
if(ptr->next==first)
{
if(first==null)
{
puts("list is empty,deletion not possible");
return;
}
else
{
printf("\nthe deleted element is %d\n",ptr->data);
first=null;
last=null;
free(first);
}
}
else
{
printf("\nthe deleted element is %d\n",ptr->data);
first=ptr->next;
last->next=first;
first->pre=last;
free(ptr);
}
}
Dend()
{
```

```
ptr=first;
if(ptr->next==first)
{
if(first==null)
{
puts(“list is empty,deletion not possible”);
return;
}
else
{
printf(“deleted element is %d\n”,ptr->data);
first=null;
last=null;
free(first);
}
}
else
{
while(ptr->next!=last)
ptr=ptr->next;
printf(“\n deleted element is %d\n”,last->data);
free(last);
ptr->next=first;
first->pre=ptr;
last=ptr;
}
}
Dmiddle(int n)
{
int i;
ptr=first;
if(ptr->next==first)
{
if(first==null)
{
```

```
puts(“list is empty,deletion not possible”);
return;
}
else
{
printf(“\ndeleted element is %d\n”,ptr->data);
first=null;
free(first);
}
}
else
{
for (i=1;i<n-1;i++)
ptr=ptr->next;
fresh=ptr->next;
ptr->next=ptr->next->next;
ptr->next->pre=ptr;
printf(“\ndeleted element is %d\n”,fresh->data);
fresh->pre=null;
fresh->next=null;
free(fresh);
}
}
```

8. (a) Write a C program to search a tree for a given element in the integer array using binary search?

```
#include<stdio.h>
main()
{
int a[10],n,I,s,m,1,u,f=o;
printf(“enter the value of n”);
scanf(“%d”,&n);
printf(“enter n sorted elements”);
for(i=0;i<n;i++)
scanf(“%d”,&a[i]);
printf(“enter the searching element”):
```

```
scanf("%d",&s);
1=0;
u=n-1;
while(1<=u)
{
m=(1+u)/2;
if(a[m]==s)
{
printf("%d is found at %d position",s,m+1);
f=1;
break;
}
else
if(s<a[m])
u=m-1;
else
1=m+1;
}
if(f=0)
printf("element is not found");
}
```

(b) Derivation of Time complexity of Binary Search-Analysis:

To analyze the time complexity of Binary Search, first we have to compute the number of comparisons expected for successful Binary search.

So, consider I, such that

$$2^i >= (N+1)$$

Thus, 2^{i-1}-1 is the maximum number of comparisons that are left with first comparison. Similarly 2^{i-2}-1 is maximum number of comparisons left with second comparison.

In general we say that 2^{i-k}-1 is the maximum number of comparisons that are left after '*k*' comparisons. If the number of elements are left then the desired data item is obtained by i^{th} comparisons.

Thus, the maximum number of comparisons required for successful search is:

$$C_{ms} = i$$

or

$$C_{ms} = \log_2 (N+1)$$

For unsuccessful search, the maximum number of comparisons:

$$C_{ms} = C_{mv}$$

for, computing average number of comparisons 'Cs' indicates successful search.

$$C_s = C/N$$

$$C = i*2^i - (2^0 + 2^1 + 2^2 + \ldots\ldots.2(i\text{-}1))$$

$$= i + 2^i\ (i\text{-}1)$$

Now, consider that the probability for searching a requested data item is $1/n$, then,

$$C_s = (1 + 2^i\ (1\text{-}\ i)/n)$$

$$C_s = [1\text{+}\ (N+1)(\log 2\ (N+1)\text{-}1)]/n$$

$$C_s = \log_2 N \text{ (for large } N\text{)}$$

It can be easily observed that for successful and unsuccessful search the expected number of cases is given by:

$$C_s = C_4 = O\ (\log_2 N)$$

It should be noted that Binary Search provides to be more efficient than the sequential search. But when implemented with linked lists it would not be efficient.

On the basis of the above analysis the time complexity of Binary Search is: $E(n) = [\log_2 n] +1$, it is actually $2^{E(\mathrm{a}})>n$, that is $O(\log_2 n)$.