

QUESTION BANK SOLUTION**MODULE 1****Introduction to Data Structures**

1. What is a pointer variable? How pointers are declared & initialized in C? Can we have multiple pointer to a variable? Explain Lvalue and Rvalue expression.(june/jul 2014) (Dec 2014/Jan 2015)

Soln: 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.

2. Give atleast 2 differences between : i. Static memory allocation and dynamic memory allocation.

ii. Malloc() and calloc(). (Dec 2014/Jan 2015)(Dec/Jan-16)

Soln:

| Static memory allocation | Dynamic memory allocation |
|--|--|
| 1.Memory is allocated during compilation time. | 1. Memory is allocated during execution time. |
| 2. Used only when the data size is fixed and known in advance before processing. | 2. Used only for unpredictable memory requirement. |

| malloc() | calloc() |
|---|---|
| 1.The syntax of malloc() is: ptr = (data_type*)malloc(size); The required number of bytes to be allocated is specified as argument i.e., size in bytes. | 1.The syntax of calloc() is: ptr = (data_type*)calloc(n,size); takes 2 arguments – n number of blocks to be allocated and size is number of bytes to be allocated for each block. |
| 2. Allocated space will not be initialized. | 2. Each byte of allocated space is initialized to zero. |

3. What is dangling pointer reference & how to avoid it? (june/jul 2014)

Soln: Dangling pointers are pointers that do not point to a valid object of the appropriate type. These are special cases of memory safety violations. More generally, **dangling references** are references that do not resolve to a valid destination, and include such phenomena as link rot on the internet. Dangling pointers arise during object destruction, when an object that has an incoming reference is deleted or deallocated, without modifying the value of the pointer, so that the pointer still points to the memory location of the deallocated memory. As the system may reallocate the previously freed memory to another process, if the original program then dereferences the (now) dangling pointer, *unpredictable behavior may result*, as the memory may now contain completely different data. This is especially the case if the program writes data to memory pointed to by a dangling pointer, a silent corruption of unrelated data may result, leading to subtle bugs that can be extremely difficult to find, or cause segmentation faults (UNIX, Linux) or general protection faults (Windows)

Avoiding: In C, the simplest technique is to implement an alternative version of the free() (or alike) function which guarantees the reset of the pointer. However, this technique will not clear other pointer variables which may contain a copy of the pointer. These uses can be masked through #define directives to construct useful macros, creating something like a metalanguage or can be embedded into a tool library apart. In every case, programmers using this technique should use the safe versions in every instance where free() would be used; failing in doing so leads again to the problem. Also, this solution is limited to the scope of a single program or project, and should be properly documented. Another approach is to use the Boehm garbage collector, a conservative garbage collector that replaces standard memory allocation functions in C and C++ with a garbage collector. This approach completely eliminates dangling pointer errors by disabling frees, and reclaiming objects by garbage collection.

4. With suitable example, explain dynamic memory allocation for 2-D arrays. (june/jul 2014)(Dec/Jan16)(jun/Jul16)

Soln: We've seen that it's straightforward to call malloc to allocate a block of memory which can simulate an array, but with a size which we get to pick at run-time. Can we do the same sort of thing to simulate multidimensional arrays? We can, but we'll end up using pointers to pointers. If we don't know how many columns the array will have, we'll clearly allocate memory for each row (as many columns wide as we like) by calling malloc, and each row will therefore be represented by a pointer. How will we keep track of those pointers? There are, after all, many of them, one for each row. So we want to simulate an array of pointers, but we don't know how

many rows there will be, either, so we'll have to simulate that array (of pointers) with another pointer, and this will be a pointer to a pointer.

This is best illustrated with an example:

```
#include <stdlib.h>

int **array;

array = malloc(nrows * sizeof(int *));

if(array == NULL)
{
    fprintf(stderr, "out of memory\n");
    exit or return
}

for(i = 0; i <nrows; i++)
{
    array[i] = malloc(ncolumns * sizeof(int));

    if(array[i] == NULL)
    {
        fprintf(stderr, "out of memory\n");
        exit or return
    }
}
```

array is a pointer-to-pointer-to-int: at the first level, it points to a block of pointers, one for each row. That first-level pointer is the first one we allocate; it has n rows elements, with each element big enough to hold a pointer-to-int, or int *. If we successfully allocate it, we then fill in the pointers (all n rows of them) with a pointer (also obtained from malloc) to n columns number of ints, the storage for that row of the array. If this isn't quite making sense, a picture should make everything clear:

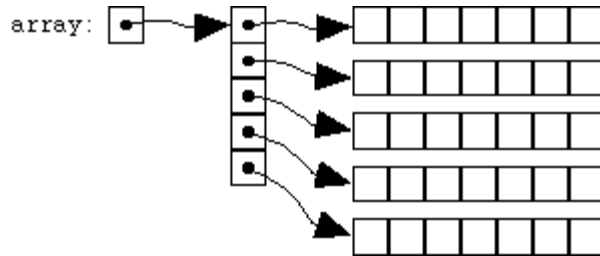


Fig1: representation of array

Once we've done this, we can (just as for the one-dimensional case) use array-like syntax to access our simulated multidimensional array. If we write

```
array[i][j]
```

we're asking for the *i*'th pointer pointed to by array, and then for the *j*'th element pointed to by that inner pointer. (This is a pretty nice result: although some completely different machinery, involving two levels of pointer dereferencing, is going on behind the scenes, the simulated, dynamically-allocated two-dimensional "array" can still be accessed just as if it were an array of arrays, i.e. with the same pair of bracketed subscripts.). If a program uses simulated, dynamically allocated multidimensional arrays, it becomes possible to write "heterogeneous" functions which *don't* have to know (at compile time) how big the "arrays" are. In other words, one function can operate on "arrays" of various sizes and shapes. The function will look something like

```
func2(int **array, intnrows, intncolumns)
{
}
```

This function does accept a pointer-to-pointer-to-int, on the assumption that we'll only be calling it with simulated, dynamically allocated multidimensional arrays. (We must not call this function on arrays like the "true" multidimensional array *a2* of the previous sections). The function also accepts the dimensions of the arrays as parameters, so that it will know how many "rows" and "columns" there are, so that it can iterate over them correctly. Here is a function which zeros out a pointer-to-pointer, two-dimensional "array":

```
void zeroit(int **array, intnrows, intncolumns)
{
    int i, j;
    for(i = 0; i < nrows; i++)
    {
```

```
        for(j = 0; j < ncolumns; j++)  
            array[i][j] = 0;  
    }  
}
```

Finally, when it comes time to free one of these dynamically allocated multidimensional ``arrays," we must remember to free each of the chunks of memory that we've allocated. (Just freeing the top-level pointer, array, wouldn't cut it; if we did, all the second-level pointers would be lost but not freed, and would waste memory.) Here's what the code might look like:

```
    for(i = 0; i < nrows; i++)  
        free(array[i]);  
    free(array);
```

5. Define a structure for the employee with the following fields: Emp_Id(integer), Emp_Name(string), Emp_Dept(string) & Emp_age(integer) Empid, DOJ (date,month,year) and salary (Basic, DA,HRA). Write the following functions to process the employee data: (i) Function to read an employee record. (ii) Function to print an employee record. (june/jul 2014) (dec 2014/jan 2015)

Soln:

```
#include <stdio.h>  
#include <conio.h>
```

```
struct details  
{  
    char name[30];  
    int age;  
    char address[500];  
    float salary;  
    char dept[30];  
    int emp_id;  
    char DOJ[20];  
};
```

```
int main()  
{  
    struct details detail;  
    clrscr();  
    printf("\nEnter name:\n");  
    gets(detail.name);
```

```

printf("\nEnter age:\n");
scanf("%d",&detail.age);
printf("\nEnter Address:\n");
gets(detail.address);
printf("\nEnter Salary:\n");
scanf("%f",&detail.salary);
printf("\nEnter dept, emp_id, DOJ);
scanf("%s%d%s", &detail.dept, &detail.emp_id,&detail.DOJ);

printf("\n\n\n");
printf("Name of the Employee : %s \n",detail.name);
printf("Age of the Employee : %d \n",detail.age);
printf("Address of the Employee : %s \n",detail.address);
printf("Salary of the Employee : %f \n",detail.salary);
printf("department of the Employee: %s\n",detail. Dept);
printf("ID of the Employee :%d\n",detail. emp_id);
printf(" Date of joining of the Employee: %s\n",detail.DOJ);

getch();
}

```

6. What is a structure ? Give three different ways of defining structure & declaring variables & method of accessing members of structures using student structure with roll number, names & marks in 3 subjects as members of that structure as example. (dec 2014/jan 2015) (june/jul 2014)

Soln: Structure:

A structure is a user-defined data type. You have the ability to define a new type of data considerably more complex than the types we have been using. A structure is a collection of one or more variables, possibly of different types, grouped together under a single name for convenient handling. Structures are called “records” in some languages, notably Pascal. Structures help organize complicated data, particularly in large programs, because they permit a group of related variables to be treated as a unit instead of as separate entities.

Today’s application requires complex data structures to support them. A structure is a collection of related elements where element belongs to a different type. Another way to look at a structure is a template – a pattern. For example graphical user interface found in a window requires structures typical example for the use of structures could be the file table which holds the key data like logical file name, location of the file on disc and so on. The file table in C is a type

defined structure - FILE. Each element of a structure is also called as field. A field has a manycharacteristic similar to that of a normal variable. An array and a structure are slightlydifferent. The former has a collection of homogeneous elements but the latter has a collection of heterogeneous elements. The general format for a structure in C is shown

```
struct{field_list} variable_identifier;
```

```
structstruct_name
```

```
{
```

```
type1 fieldname1;
```

```
type2 fieldname2;
```

```
.
```

```
.
```

```
.
```

```
typeNfieldnameN;
```

```
};
```

```
structstruct_name variables;
```

The above format shown is not concrete and can vary, so different `

flavours of structure declaration is as shown.

```
struct
```

```
{
```

```
....
```

```
} variable_idenfifer;
```

Example

```
structmob_equip;
```

```
{
```

```
long int IMEI;
```

```
char rel_date[10];
```

```
char model[10];
```

```
char brand[15];
```

```
};
```

The above example can be upgraded with *typedef*. A program to illustrate the working of the structure is shown in the previous section.

```
typedef struct mob_equip;
```

```
{
```

```
long int IMEI;
```

```
char rel_date[10];
```

```
char model[10];
```

```
char brand[15];
```

```
int count;
```

```
} MOB; MOB m1;
```

```
struct tag
```

```
{
```

```
.....
```

```
};
```

```
struct tag variable_identifiers;
```

```
typedef struct
```

```
{
```

```
.....
```

```
} TYPE_IDENTIFIER;
```

```
TYPE_IDENTIFIER variable_identifiers;
```

Accessing a structure

DATA STRUCTURES AND APPLICATIONS

15CS33

A structure variable or a tag name of a structure can be used to access the members of a structure with the help of a special operator '.' –also called as member operator. In our previous example To access the idea of the IMEI of the mobile equipment in the structure mob_equip is done like this Since the structure variable can be treated as a normal variable All the IO functions for a normal variable holds good for the structure variable also with slight. The scanf statement to read the input to the IMEI is given below

```
scanf ("%d",&m1.IMEI);
```

Increment and decrement operation are same as the normal variables this includes postfix and prefix also. Member operator has more precedence than the increment or decrement. Say suppose in example quoted earlier we want count of student then

```
m1.count++; ++m1.count
```

```
#include<stdio.h>
struct student
{
    char name[10];
    char rollno[10];
    char gender;
    int marks[3];
    float avg;
};
int main()
{
    int i,sum=0;
    struct student s1;
    printf("\n\t\t\tEnter the students details : ");
    printf("\n\n Enter the name of the student : ");
    fflush(stdin);
    gets(s1.name);
    printf(" Enter the Rollno of the student : ");
    gets(s1.rollno);
```

7. What is the purpose of free()? With an example explain the problem when it is not used. (Dec-Jan-16)(Jun-Jul 16)

The C library function void free(void *ptr) deallocates the memory previously allocated by a call to calloc, malloc, or realloc.

Declaration

Following is the declaration for free() function.

```
void free(void *ptr)
```

Parameters

ptr -- This is the pointer to a memory block previously allocated with malloc, calloc or realloc to be deallocated. If a null pointer is passed as argument, no action occurs.

Return Value

This function does not return any value.

Example

The following example shows the usage of free() function.

```
#include <stdio.h>
```

```
#include <stdlib.h>
```

```
#include <string.h>
```

```
int main()
```

```
{
```

```
    char *str;
```

```
    /* Initial memory allocation */
```

```
    str = (char *) malloc(15);
```

```
    strcpy(str, "tutorialspoint");
```

```
printf("String = %s, Address = %u\n", str, str);
```

```
/* Reallocating memory */
```

```
str = (char *) realloc(str, 25);
```

```
strcat(str, ".com");
```

```
printf("String = %s, Address = %u\n", str, str);
```

```
/* Deallocate allocated memory */
```

```
free(str);
```

```
return(0);
```

```
}
```

8. Explain how memory can be allocated using realloc()?(Dec/Jan -16)(Jun/Jul-16)

Description

The C library function `void *realloc(void *ptr, size_t size)` attempts to resize the memory block pointed to by `ptr` that was previously allocated with a call to `malloc` or `calloc`.

Declaration

Following is the declaration for `realloc()` function.

```
void *realloc(void *ptr, size_t size)
```

Parameters

`ptr` -- This is the pointer to a memory block previously allocated with `malloc`, `calloc` or `realloc` to be reallocated. If this is `NULL`, a new block is allocated and a pointer to it is returned by the function.

size -- This is the new size for the memory block, in bytes. If it is 0 and ptr points to an existing block of memory, the memory block pointed by ptr is deallocated and a NULL pointer is returned.

Return Value

This function returns a pointer to the newly allocated memory, or NULL if the request fails.

Example

The following example shows the usage of realloc() function.

```
#include <stdio.h>

#include <stdlib.h>

int main()
{
    char *str;

    /* Initial memory allocation */
    str = (char *) malloc(15);
    strcpy(str, "tutorialspoint");
    printf("String = %s, Address = %u\n", str, str);

    /* Reallocating memory */
    str = (char *) realloc(str, 25);
    strcat(str, ".com");
    printf("String = %s, Address = %u\n", str, str);
```

```
free(str);

return(0);

}
```

9. Describe unions in c? How is it different from structures? (Jun/Jul 16)

A union is a special data type available in C that allows to store different data types in the same memory location. You can define a union with many members, but only one member can contain a value at any given time. Unions provide an efficient way of using the same memory location for multiple-purpose.

To define a union, you must use the union statement in the same way as you did while defining a structure. The union statement defines a new data type with more than one member for your program. The format of the union statement is as follows –

```
union [union tag] {

    member definition;

    member definition;

    ...

    member definition;

} [one or more union variables];
```

The union tag is optional and each member definition is a normal variable definition, such as `int i`; or `float f`; or any other valid variable definition. At the end of the union's definition, before the final semicolon, you can specify one or more union variables but it is optional. Here is the way you would define a union type named `Data` having three members `i`, `f`, and `str` –

```
union Data {

    int i;

    float f;
```

```
    char str[20];  
  
} data;
```

Now, a variable of Data type can store an integer, a floating-point number, or a string of characters. It means a single variable, i.e., same memory location, can be used to store multiple types of data. You can use any built-in or user defined data types inside a union based on your requirement.

The memory occupied by a union will be large enough to hold the largest member of the union. For example, in the above example, Data type will occupy 20 bytes of memory space because this is the maximum space which can be occupied by a character string. The following example displays the total memory size occupied by the above union –

```
#include <stdio.h>  
  
#include <string.h>  
  
union Data {  
    int i;  
    float f;  
    char str[20];  
};  
  
int main( ) {
```

```
    union Data data;
```

```
    printf( "Memory size occupied by data : %d\n", sizeof(data));
```

```
return 0;  
  
}
```

MODULE 2

Linear Data Structures and their Sequential Storage Representation

1. Define Recursion. What are the various types of recursion? Give two conditions to be followed for successive working of recursive program. Give recursive implementation of binary's search with proper comments. (june/july 2015) (Dec 2014/Jan 2015)(Dec/Jan-16)(Jun/Jul 16)

Soln: Recursive function is a function which contains a call to itself. **Recursive function allows you to divide your complex problem into identical single simple cases which can handle easily. This is also a well-known computer programming technique: divide and conquer.**

Recursion that only contains a single self-reference is known as **single recursion**, while recursion that contains multiple self-references is known as **multiple recursion**. Standard examples of single recursion include list traversal, such as in a linear search, or computing the factorial function, while standard examples of multiple recursion include tree traversal, such as in a depth-first search, or computing the Fibonacci sequence.

Multiple recursion can sometimes be converted to single recursion (and, if desired, thence to iteration). For example, while computing the Fibonacci sequence naively is multiple iteration, as each value requires two previous values, it can be computed by single recursion by passing two successive values as parameters.

direct recursion, in which a function calls itself. Indirect recursion occurs when a function is called not by itself but by another function that it called (either directly or indirectly). For example, if f calls f, that is direct recursion, but if f calls g which calls f, then that is indirect recursion of f. Chains of three or more functions are possible; for example, function 1 calls function 2, function 2 calls function 3, and function 3 calls function 1 again.

Indirect recursion is also called mutual recursion, which is a more symmetric term, though this is simply a different of emphasis, not a different notion. That is, if f calls g and then g calls f, which in turn calls g again, from the point of view of f alone, f is indirectly recursing, while from the point of view of g alone, it is indirectly recursing, while from the point of view of both, f and g are mutually recursing on each other. Similarly a set of three or more functions that call each other can be called a set of mutually recursive functions.

```
#include<stdio.h>
```

```
int main(){
```

```
int a[10],i,n,m,c,l,u;
```

```
printf("Enter the size of an array: ");
```

```
scanf("%d",&n);
```

```
printf("Enter the elements of the array: ");
```

```
for(i=0;i<n;i++){
```

```
    scanf("%d",&a[i]);
```

```
}
```

```
printf("Enter the number to be search: ");
```

```
scanf("%d",&m);
```

```
l=0,u=n-1;
```

```
c=binary(a,n,m,l,u);
```

```
if(c==0)
```

```
    printf("Number is not found.");
```

```
else
```

```
    printf("Number is found.");
```

```
return 0;
```

```
}
```

```
int binary(int a[],int n,int m,int l,int u){
```

```
int mid,c=0;

if(l<=u){
    mid=(l+u)/2;
    if(m==a[mid]){
        c=1;
    }
    else if(m<a[mid]){
        return binary(a,n,m,l,mid-1);
    }
    else
        return binary(a,n,m,mid+1,u);
}

else
    return c;
}
```

**2. Estimate the space complexity of a recursive function for summing a list of numbers.
(june/jul 2014)**

Soln:

```
float rsum(float list[ ], int n)
{
    count++;    /*for if conditional */
    if (n) {
        count++; /* for return and rsum invocation */
        return rsum(list, n-1) + list[n-1];
    }
}
```

```
count++;
return list[0];
}
```

2n+2

| Statement | s/e | Frequency | Total steps |
|-----------------------------------|-----|-----------|-------------|
| float rsum(float list[], int n) | 0 | 0 | 0 |
| { | 0 | 0 | 0 |
| if (n) | 1 | n+1 | n+1 |
| return rsum(list, n-1)+list[n-1]; | 1 | n | n |
| return list[0]; | 1 | 1 | 1 |
| } | 0 | 0 | 0 |
| Total | | | 2n+2 |

```
float rsum(float list[], int n)
{
    if (n) return rsum(list,n-1) + list[n-1];
    return 0;
}
```

Program 1.11: Recursive function for summing a list of numbers

| Type | Name | Number of bytes |
|-----------------------------------|----------------|--------------------------|
| parameter: float | <i>list</i> [] | 2 |
| parameter: integer | <i>n</i> | 2 |
| return address: (used internally) | | 2 (unless a far address) |
| TOTAL per recursive call | | 6 |

Figure 1.1: Space needed for one recursive call of Program 1.11

3. Write an algorithm to convert a valid infix expression to a postfix expression. Also evaluate the following suffix expression for the values: A=1 B=2 C=3.

DATA STRUCTURES AND APPLICATIONS

15CS33

AB+C-BA+C\$- and convert i) $a*(b+c)*d$ ii) $(a+b)*d+e/(f+a*d)+c$

iii) $((a/(b-c+d))*(e-a)*c)$ iv) $a/b-c+d*e-a*c$ iv) $(a*b) +c/d$

v) $((a/b)c)+(d*e)) (a*c)$ to postfix.

(june/jul 2014) (Dec 2014/Jan 2015) (june/jul 2015)(Dec-Jan 16)

Soln:

| Symbols | Stack precedence function F | Input precedence funcyion G |
|------------------|--------------------------------|--------------------------------|
| $+, -$ | 2 | 1 |
| $*, /$ | 4 | 3 |
| $\$$ or \wedge | 5 | 6 |
| operands | 8 | 7 |
| (| 0 | 9 |
|) | - | 0 |
| # | -1 | |

Algorithm to convert from infix to postfix:

- As long as the precedence value of the symbol on top of the stack is greater than the precedence value of the current input symbol, poop an item from the stack and place it in the postfix expression. The code for this statement can be of the form:

```
while(F(s[top]) > G(symbol))
```

```
{
```

```
postfix[j++] = s[top--];
```

```
}
```

- Once the condition in the while loop is failed, if the precedence of the symbol on top of the stack is not equal to the precedence value of the current input symbol, push the current symbol on to the stack. Otherwise, delete an item from the stack but do not place it in the postfix expression. The code for this statement can be of the form :

```
if(F(s[top]) != G(symbol))
```

```
s[++top] = symbol;
```

```
else
```

```
top--;
```

- These 2 steps have to be performed for each input symbol in the infix expression.

AB+C-BA+C\$- A=1 B=2 C=3.

= 1 2 + 3 - 2 1 + 3 \$ -

| sym=string[i] | OP2=s[top--] | OP1=s[top--] | RES | s[++top] | STACK S |
|---------------|--------------|--------------|------------|----------|-------------|
| 1 | | | | 1 | 1 |
| 2 | | | | 2 | 2 1 |
| + | 2 | 1 | 1+2 = 3 | 3 | 3 |
| 3 | | | | 3 | 3 3 |
| - | 3 | 3 | 3-3 = 0 | 0 | 0 |
| 2 | | | | 2 | 2 0 |
| 1 | | | | 1 | 1 2 0 |
| + | 1 | 2 | 2+1 = 3 | 3 | 3 0 |
| 3 | | | | 3 | 3 3 0 |
| \$ | 3 | 3 | 3^3 = 27 | 27 | 27 0 |
| - | 27 | 0 | 0-27 = -27 | -27 | -27 |

Required result = -27

4. What is the advantage of circular queue over ordinary queue? Mention any 2 applications of queues. Write an algorithm CQINSERT for static implementation of circular queue. (june/jul 2014) (Dec/Jan 16)

Soln:

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.

Algorithm CQINSERT for static implementation of circular queue -

Step 1 : **Check for overflow** : This is achieved using the following statements `if(count == queue_size)`

```
{  
    printf("Queue is full\n");  
    return;  
}
```

Step 2: **Insert item** : This is achieved by incrementing `r` by 1 and then inserting as shown below

`= (r+1)%queue_size;`

`q[r] = item;`

Step 3: **Update count** : As we insert an element, update count by 1. This is achieved using the following statement `count++`;

5. Define stack. Implement push & pop functions for stack using arrays. (Dec 2014/Jan 2015) (June/Jul 2014)(Dec/Jan 16)

Soln: A *stack* is an ordered collection of items into which new items may be inserted and from which items may be deleted at one end, called the *top* of the stack. A stack is a dynamic, constantly changing object as the definition of the stack provides for the insertion and deletion of items. It has single end of the stack as top of the stack, where both insertion and deletion of the elements takes place. The last element inserted into the stack is the first element deleted-*last in first out list (LIFO)*. After several insertions and deletions, it is possible to have the same frame again.

Primitive Operations

When an item is added to a stack, it is *pushed* onto the stack. When an item is removed, it is *popped* from the stack.

Given a stack s , and an item i , performing the operation $push(s, i)$ adds an item i to the top of stack s .

`push(s, H);`

`push(s, I);`

`push(s, J);`

Operation $pop(s)$ removes the top element. That is, if $i = pop(s)$, then the removed element is assigned to i .

`pop(s);`

Because of the push operation which adds elements to a stack, a stack is sometimes called a *pushdown list*. Conceptually, there is no upper limit on the number of items that may be kept in a stack. If a stack contains a single item and the stack is popped, the resulting stack contains no items and is called the *empty stack*. Push operation is applicable to any stack. Pop operation cannot be applied to the empty stack. If so, *underflow* happens. A Boolean operation $empty(s)$, returns TRUE if stack is empty. Otherwise FALSE, if stack is not empty.

Representing stacks in C

Before programming a problem solution that uses a stack, we must decide how to represent the stack in a programming language. It is an ordered collection of items. In C, we have ARRAY as an ordered collection of items. But a stack and an array are two different things. The number of

DATA STRUCTURES AND APPLICATIONS

15CS33

elements in an array is fixed. A stack is a dynamic object whose size is constantly changing. So, an array can be declared large enough for the maximum size of the stack. A stack in C is declared as a structure containing two objects:

- An array to hold the elements of the stack.
- An integer to indicate the position of the current stack top within the array.

```
#define STACKSIZE 100
```

```
struct stack {  
  
    int top;  
  
    int items[STACKSIZE];  
  
};
```

The stack *s* may be declared by

```
struct stack s;
```

The stack items may be int, float, char, etc. The empty stack contains no elements and can therefore be indicated by *top* = -1. To initialize a stack *S* to the empty state, we may initially execute

```
s.top = -1.
```

To determine stack empty condition,

```
if (s.top == -1)
```

```
    stack empty;
```

```
else
```

```
    stack is not empty;
```

The empty(s) may be considered as follows:

```
int empty(struct stack *ps)
```

```
{
```

```
    if(ps->top == -1)
```

```
        return(TRUE);
```

```
    else
```

```
return(FALSE);

}
```

Aggregating the set of implementation-dependent trouble spots into small, easily identifiable units is an important method of making a program more understandable and modifiable. This concept is known as **modularization**, in which individual functions are isolated into low-level modules whose properties are easily verifiable. These low-level **modules** can then be used by more complex routines, which do not have to concern themselves with the details of the low-level modules but only with their function. The complex routines may themselves then be viewed as modules by still higher-level routines that use them independently of their internal details.

• Implementing pop operation

If the stack is empty, print a warning message and halt execution. Remove the top element from the stack. Return this element to the calling program

```
int pop(struct stack *ps)
{
    if(empty(ps)){
        printf("%s", "stack underflow");
        exit(1);
    }
    return(ps->items[ps->top--]);
}
```

Differentiate between stack and queue. How are they related to LIFO and FIFO concept? (Jun/Jul 16)

Stack – Represents the collection of elements in Last in First Out order. Operations include testing null stack, finding the top element in the stack, removal of top most element and adding elements on the top of the stack.

Queue - Represents the collection of elements in First In First Out order. Operations include testing null queue, finding the next element, removal of elements and inserting the elements from the queue.

DATA STRUCTURES AND APPLICATIONS

15CS33

Insertion of elements is at the end of the queue
Deletion of elements is from the beginning of the queue.

Write a program to check if a given string is palindrome or not?(Jun-July 16)

```
#include <stdio.h>
```

```
#include <string.h>
```

```
int main(){
```

```
    char string1[20];
```

```
    int i, length;
```

```
    int flag = 0;
```

```
    printf("Enter a string:");
```

```
    scanf("%s", string1);
```

```
    length = strlen(string1);
```

```
    for(i=0;i < length ;i++){
```

```
        if(string1[i] != string1[length-i-1]){
```

```
            flag = 1;
```

```
            break;
```

```
        }
```

```
    }
```

```
    if (flag) {
```

```
        printf("%s is not a palindrome", string1);
```

```
}  
  
else {  
    printf("%s is a palindrome", string1);  
}  
  
return 0;  
}
```

MODULE 3

Linear Data Structures and their Linked Storage Representation

1. List out any two applications of linked list and any two advantages of doubly linked list over singly linked list. (june/jul 2014) (Jun-Jul 16)(Dec/Jan 16)

Soln: Applications of linked list:

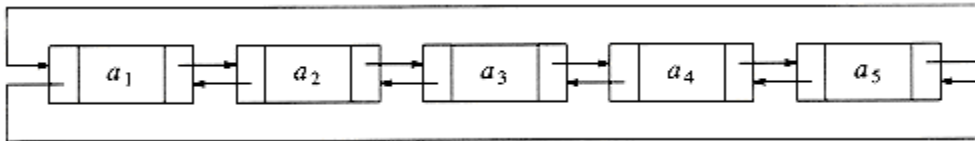
- Arithmetic operations can be easily performed on long positive numbers.
- Manipulation and evaluation of polynomials is easy.
- Useful in symbol table construction(Compiler design).

Advantages of doubly linked list over singly linked list:

| Singly linked list | Doubly linked list |
|--|--|
| 1. Traversing is only in one direction. | 1. Traversing can be done in both directions. |
| 2. While deleting a node, its predecessor is required and can be found only after traversing from the beginning of list. | 2. While deleting a node x, its predecessor can be obtained using llink of node x. No need to traverse the list. |
| 3. programs will be lengthy and need more time to design. | 3. Using circular linked list with header, efficient and small programs can be written and hence design is easier. |

2. Write short note on circular lists. Write a function to insert a node at front and rear end in a circular linked list. Write down sequence of steps to be followed. (june/jul 2015)

Soln: A popular convention is to have the last cell keep a pointer back to the first. This can be done with or without a header (if the header is present, the last cell points to it), and can also be done with doubly linked lists (the first cell's previous pointer points to the last cell). This clearly affects some of the tests, but the structure is popular in some applications. Figure 3.17 shows a double circularly linked list with no header.



Create a new node in memory, set its data to val

♣ Make the node point to itself

♣ if tail is empty, then return this node, it's the only one in the list

♣ If it's not the only node, then it's next is tail ->next

♣ and tail ->next should now point to the new node.

typedef

struct

node {

int

data;

node *next;

} node;

node* AddFront(node* tail, int val)

{

// Create the new node

node *temp = (node*)malloc(sizeof(node));

temp->data = val;

DATA STRUCTURES AND APPLICATIONS

15CS33

```
// Set the new node's next to itself (circular!)
temp->next = temp;

// If the list is empty, return new node
if (tail == NULL) return temp;

// Set our new node's next to the front
temp->next = tail->next;

// Set tail's next to our new node
tail->next = temp;

// Return the end of the list
return tail;
}
```

```
Struct node* AddEnd(struct node* tail, int val)
{
// Create the new node
node *temp = (node*)
malloc ( sizeof(node));
temp->data = val;

// Set the new node's next to itself (circular!)
temp->next = temp;

// If the list is empty, return new node
if (tail == NULL) return temp;

// Set our new node's next to the front
temp->next = tail->next;

// Set tail's next to our new node
```

```

tail->next = temp;

// Return the
new end of the list

return temp;

}

```

3. Write the following functions for singly linked list: i) Reverse the list ii) Concatenate two lists. (june/jul 2014) (june/jul 2015) (Jun/Jul 16)

Soln:

Another useful subroutine is one which concatenates two chains X and Y .

```

procedure CONCATENATE( $X, Y, Z$ )
// $X = (a_1, \dots, a_m), Y = (b_1, \dots, b_n), m, n \geq 0$ , produces a new chain
 $Z = (a_1, \dots, a_m, b_1, \dots, b_n)$ //
 $Z \leftarrow X$ 
if  $X = 0$  then [ $Z \leftarrow Y$ ; return]
if  $Y = 0$  then return
 $p \leftarrow X$ 
while  $LINK(p) \neq 0$  do    //find last node of  $X$ //
 $p \leftarrow LINK(p)$ 
end
 $LINK(p) \leftarrow Y$     //link last node of  $X$  to  $Y$ //
end CONCATENATE

```

This algorithm is also linear in the length of the first list. From an aesthetic point of view it is nicer to write this procedure using the case statement in SPARKS. This would look like:

```

procedure CONCATENATE( $X, Y, Z$ )
case
:  $X = 0$  :  $Z \leftarrow Y$ 
:  $Y = 0$  :  $Z \leftarrow X$ 
: else :  $p \leftarrow X; Z \leftarrow X$ 
while  $LINK(p) \neq 0$  do
 $p \leftarrow LINK(p)$ 
end
 $LINK(p) \leftarrow Y$ 
end
end CONCATENATE

```

Suppose we want to insert a new node at the front of this list. We have to change the LINK field of the node containing x_3 . This requires that we move down the entire length of A until we find the last node. It is more convenient if the name of a circular list points to the last node rather than the first.

4. Write the node structure for linked representation of polynomial. Explain the algorithm to add two polynomial represented using linked lists. (june/jul 2014) (june/july 2013) (Dec/Jan 16)

Soln:

Representation of a Polynomial: A polynomial is an expression that contains more than two terms. A term is made up of coefficient and exponent. An example of polynomial is

$$P(x) = 4x^3 + 6x^2 + 7x + 9$$

A polynomial thus may be represented using arrays or linked lists. Array representation assumes that the exponents of the given expression are arranged from 0 to the highest value (degree), which is represented by the subscript of the array beginning with 0. The coefficients of the respective exponent are placed at an appropriate index in the array. The array representation for the above polynomial expression is given below.

A polynomial may also be represented using a linked list. A structure may be defined such that it contains two parts- one is the coefficient and second is the corresponding exponent. The structure definition may be given as shown below:

```
struct polynomial
{
    int coefficient;
    int exponent;
    struct polynomial *next;
};
```

Thus the above polynomial may be represented using linked list as shown below:

Addition of two Polynomials:

For adding two polynomials using arrays is straightforward method, since both the arrays may be added up element wise beginning from 0 to n-1, resulting in addition of two polynomials. Addition of two polynomials using linked list requires comparing the exponents, and wherever the exponents are found to be same, the coefficients are added up. For terms with different exponents, the complete term is simply added to the result thereby making it a part of addition result. The complete program to add two polynomials is given in subsequent section.

Multiplication of two Polynomials:

Multiplication of two polynomials however requires manipulation of each node such that the exponents are added up and the coefficients are multiplied. After each term of first polynomial is operated upon with each term of the second polynomial, then the result has to be added up by comparing the exponents and adding the coefficients for similar exponents and including terms as such with dissimilar exponents in the result. The 'C' program for polynomial manipulation is given below:

Program for Polynomial representation, addition and multiplication

```
/*Polynomial- Representation, Addition, Multiplication*/
```

```
#include <stdio.h>
```

```
#include <conio.h>
```

```
#include <stdlib.h>
```

```
struct poly
```

```
{
```

```
int coeff;
```

```
int exp;
```

```
struct poly *next;
```

```
}*head1=NULL,*head2=NULL,*head3=NULL,*head4=NULL,*temp,*ptr;
```

```
void create();
```

DATA STRUCTURES AND APPLICATIONS

15CS33

```
void makenode(int,int);

struct poly *insertend(struct poly *);

void display(struct poly *);

struct poly *addtwopoly(struct poly *,struct poly *,struct poly *);

struct poly *subtwopoly(struct poly *,struct poly *,struct poly *);

struct poly *multwopoly(struct poly *,struct poly *,struct poly *);

struct poly *dispose(struct poly *);

int search(struct poly *,int);


void main()

{

int ch,coefficient,exponent;

int listno;

while(1)

{

clrscr();

printf("ntMenu");

printf("nt1. Create First Polynomial.");

printf("nt2. Display First Polynomial.");

printf("nt3. Create Second Polynomial.");

printf("nt4. Display Second Polynomial.");

printf("nt5. Add Two Polynomials.");

printf("nt6. Display Result of Addition.");

printf("nt7. Subtract Two Polynomials.");

printf("nt8. Display Result of Subtraction.");
```

```
printf("nt9. Multiply Two Polynomials.");
printf("nt10. Display Result of Product.");
printf("nt11. Dispose List.");
printf("nt12. Exit");
printf("nntEnter your choice?");
scanf("%d",&ch);
switch(ch)
{
case 1:
printf("nGenerating first polynomial:");
printf("nEnter coefficient?");
scanf("%d",&coefficient);
printf("nEnter exponent?");
scanf("%d",&exponent);
makenode(coefficient,exponent);
head1 = insertend(head1);
break;
case 2:
display(head1);
break;
case 3:
printf("nGenerating second polynomial:");
printf("nEnter coefficient?");
scanf("%d",&coefficient);
printf("nEnter exponent?");
```

DATA STRUCTURES AND APPLICATIONS

15CS33

```
scanf("%d",&exponent);
makenode(coefficient,exponent);
head2 = insertend(head2);
break;
case 4:
display(head2);
break;
case 5:
printf("\nDisposing result list.");
head3=dispose(head3);
head3=addtwopoly(head1,head2,head3);
printf("Addition successfully done!");
break;
case 6:
display(head3);
break;
case 7:
head3=dispose(head3);
head3=subtwopoly(head1,head2,head3);
printf("Subtraction successfully done!");
getch();
break;
case 8:
display(head3);
break;
```

DATA STRUCTURES AND APPLICATIONS

15CS33

case 9:

head3=dispose(head3);

head4=dispose(head4);

head4=multwopoly(head1,head2,head3);

break;

case 10:

display(head4);

break;

case 11:

printf("Enter list number to dispose(1 to 4)?");

scanf("%d",&listno);

if(listno==1)

head1=dispose(head1);

else if(listno==2)

head2=dispose(head2);

else if(listno==3)

head3=dispose(head3);

else if(listno==4)

head4=dispose(head4);

else

printf("Invalid number specified.");

break;

case 12:

exit(0);

default:

DATA STRUCTURES AND APPLICATIONS

15CS33

```
printf("Invalid Choice!");
```

```
break;
```

```
}
```

```
}
```

```
}
```

```
void create()
```

```
{
```

```
ptr=(struct poly *)malloc(sizeof(struct poly));
```

```
if(ptr==NULL)
```

```
{
```

```
printf("Memory Allocation Error!");
```

```
exit(1);
```

```
}
```

```
}
```

```
void makenode(int c,int e)
```

```
{
```

```
create();
```

```
ptr->coeff = c;
```

```
ptr->exp = e;
```

```
ptr->next = NULL;
```

```
}
```

```
struct poly *insertend(struct poly *head)
```

DATA STRUCTURES AND APPLICATIONS

15CS33

```
{  
if(head==NULL)  
head = ptr;  
else  
{  
temp=head;  
while(temp->next != NULL)  
temp = temp->next;  
temp->next = ptr;  
}  
return head;  
}  
  
void display(struct poly *head)  
{  
if(head==NULL)  
printf("List is empty!");  
else  
{  
temp=head;  
while(temp!=NULL)  
{  
printf("(%d,%d)->",temp->coeff,temp->exp);  
temp=temp->next;  
}}
```

DATA STRUCTURES AND APPLICATIONS

15CS33

```
printf("bb ");
}
getch();
}

struct poly *addtwopoly(struct poly *h1,struct poly *h2,struct poly *h3)
{
/*
(5,3)->(6,1) + (7,3)->(9,2) = (12,3)->(6,1)->(9,2)
*/
struct poly *temp1,*temp2,*temp3;
temp1=h1;
temp2=h2;
while(temp1!=NULL || temp2!=NULL)
{
if(temp1->exp==temp2->exp)
{
makenode(temp1->coeff+temp2->coeff,temp1->exp);
h3=insertend(h3);
}
else
{
makenode(temp1->coeff,temp1->exp);
h3=insertend(h3);
makenode(temp2->coeff,temp2->exp);
```

DATA STRUCTURES AND APPLICATIONS

15CS33

```
h3=insertend(h3);  
  
}  
  
temp1=temp1->next;  
temp2=temp2->next;  
  
}  
  
if(temp1==NULL && temp2!=NULL)  
{  
    while(temp2!=NULL)  
    {  
        makenode(temp2->coeff,temp2->exp);  
        h3=insertend(h3);  
        temp2=temp2->next;  
    }  
}  
  
if(temp2==NULL && temp1!=NULL)  
{  
    while(temp1!=NULL)  
    {  
        makenode(temp2->coeff,temp2->exp);  
        h3=insertend(h3);  
        temp1=temp1->next;  
    }  
}  
  
return h3;  
  
}
```

```
struct poly *subtwopoly(struct poly *h1,struct poly *h2,struct poly *h3)
{
/*
(5,3)->(6,1) - (7,3)->(9,2) = (-2,3)+(6,1)-(9,2)
*/
struct poly *temp1,*temp2,*temp3;
temp1=h1;
temp2=h2;
while(temp1!=NULL || temp2!=NULL)
{
if(temp1->exp==temp2->exp)
{
makenode(temp1->coeff-temp2->coeff,temp1->exp);
h3=insertend(h3);
}
else
{
makenode(temp1->coeff,temp1->exp);
h3=insertend(h3);
makenode(-temp2->coeff,temp2->exp);
h3=insertend(h3);
}
temp1=temp1->next;
temp2=temp2->next;
```

```
}  
  
if(temp1==NULL && temp2!=NULL)  
{  
    while(temp2!=NULL)  
    {  
        makenode(temp2->coeff,temp2->exp);  
        h3=insertend(h3);  
        temp2=temp2->next;  
    }  
}  
  
if(temp2==NULL && temp1!=NULL)  
{  
    while(temp1!=NULL)  
    {  
        makenode(-temp2->coeff,temp2->exp);  
        h3=insertend(h3);  
        temp1=temp1->next;  
    }  
}  
  
return h3;  
}
```

5. What is a linked list? Explain the different types of linked list with diagram. Write C program to implement the insert and delete operation on a queue using linked list. (dec 2014/jan 2015)(Dec/Jan 16)

Soln:

```
procedure INSERT__FRONT(A, X)
//insert the node pointed at by X to the front of the circular list
A, where A points to the last node//
if A = 0 then [A X
LINK (X) A]
else [LINK(X) LINK (A)
LINK(A) X]
end INSERT--FRONT
```

To insert X at the rear, one only needs to add the additional statement A X to the **else** clause of *INSERT_-_FRONT*.

MODULE 4

Nonlinear Data Structures

1. Define the tree & the following

i) Binary tree

ii) Complete binary tree

iii) Almost complete binary tree

iv) Binary search tree

v) Depth of a tree

vi) Degree of a binary tree

vii) Level of a binary tree

viii) Sibling

ix) Root node

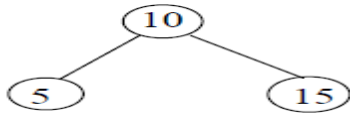
x) Child

xi) Ancestors (Dec 2014/Jan 2015) (June/Jul 2014) (Jun/Jul 16)

Soln:

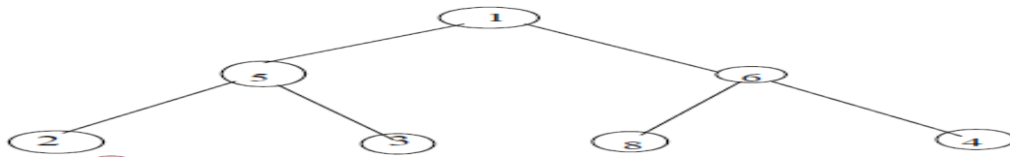
i) Binary tree – A binary tree is a tree which is a collection of zero or more nodes and finite set of directed lines called branches that connect the nodes. A tree can be empty or partitioned into three subgroups, namely :

- * Root – If tree is not empty, the first node is called root node.
- * Left subtree – It is a tree which is connected to the left of root.
- * Right subtree - It is a tree which is connected to the right of root.



ii) Complete binary tree - A strictly binary tree in which the number of nodes at any level i is 2^i , is said to be a complete binary tree.

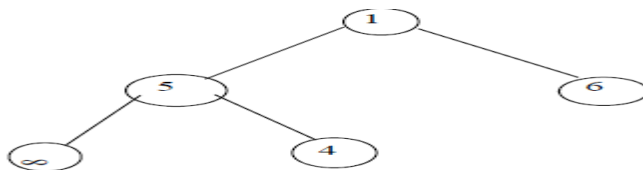
Ex:



iii) Almost complete binary tree - A tree of depth d is an almost complete binary tree if following two properties are satisfied -

1. If the tree is complete upto the level $d-1$, then the total number of nodes at the level $d-1$ should be 2^{d-1} .
2. The total number of nodes at level d may be equal to 2^d . If the total number of nodes at level d is less than 2^d , then the number of nodes at level $d-1$ should be 2^{d-1} and in level d the nodes should be present only from left to right.

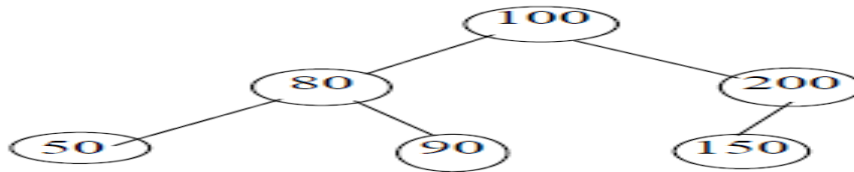
ex:



iv) Binary search tree – A binary search tree is a binary tree in which for each node say x in the tree, elements in the left sub-tree are less than $\text{info}(x)$ and elements in the

right sub-tree are greater than or equal to $\text{info}(x)$. Every node in the tree should satisfy this condition, if left sub-tree or right sub-tree exists.

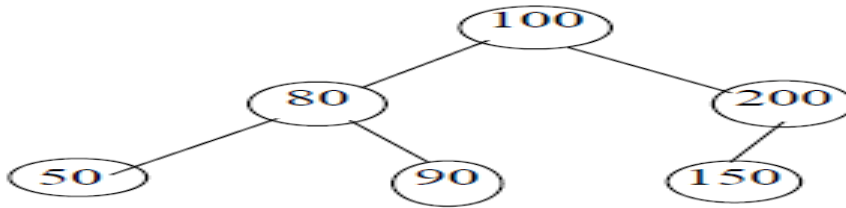
Ex:



v) Depth of a tree – Depth of a tree is defined as the length of the longest path from root to a leaf of the tree. Length of a path is the number of branches encountered

while traversing the path.

Ex:



The depth of the tree = 2

vi) Degree of a binary tree

vii) Level of a binary tree

viii) Sibling

ix) Root node

x) Child

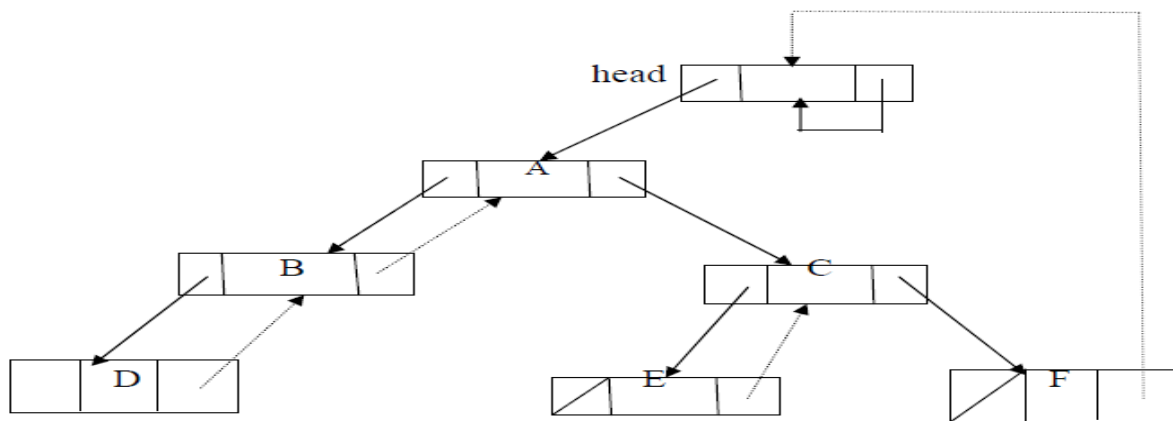
xi) Ancestors

2. What is threaded binary tree? Explain right in and left in threaded binary trees. Advantages of TBT over binary tree. (june/jul 2014) (Dec 2014/Jan 2015)

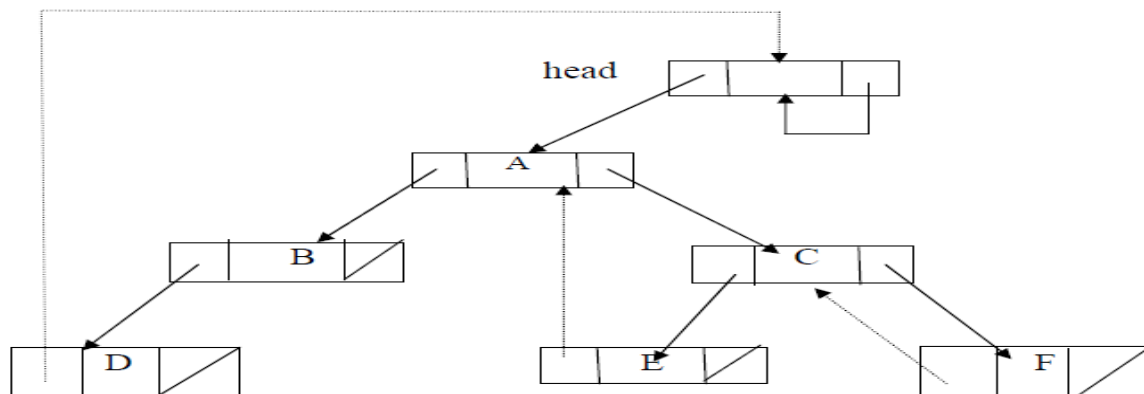
Soln: Threaded binary tree – The link fields in a binary tree which contain NULL values can be replaced by address of some nodes in the tree which facilitate upward movement in the tree. These extra links which contain addresses of some nodes are called threads and the tree is termed as threaded binary tree.

In-threaded binary tree - In a binary tree, if llink of any node contains null and if it is replaced by address of inorder predecessor, then the resulting tree is called left inthreaded binary tree. If rlink of a node is null and if it is replaced by address of inorder successor, the resulting tree is called right inthreaded tree. A inthreaded binary tree is one which is both left and right in-threaded.

EX: Right in-threaded binary tree:



EX: Left in-threaded binary tree:



Advantages of TBT over binary tree:

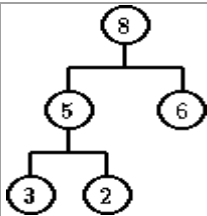
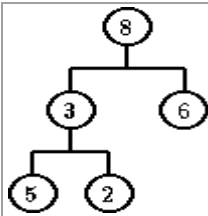
3. What is a heap? Explain the different types of heap? (june/july 2015)

Soln:

A **heap** is a complete tree with an ordering-relation R holding between each node and its descendant.

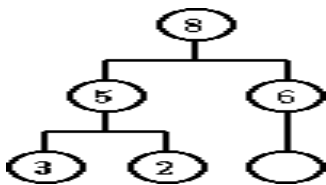
Examples for R: smaller-than, bigger-than

Assumption: In what follows, R is the relation ‘bigger-than’, and the trees have degree 2.

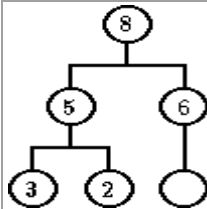
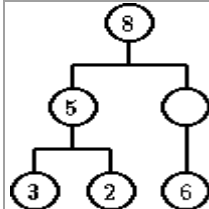
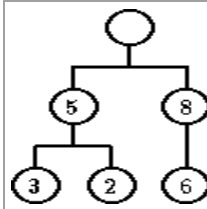
| Heap | Not a heap |
|---|---|
|  |  |

Adding an Element

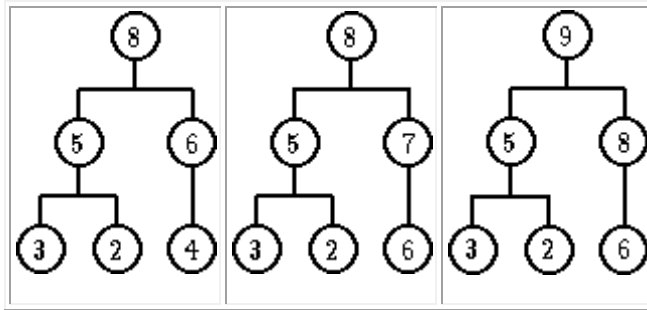
1. Add a node to the tree



2. Move the elements in the path from the root to the new node one position down, if they are smaller than the new element

| new element | 4 | 7 | 9 |
|---------------|---|---|--|
| modified tree |  |  |  |

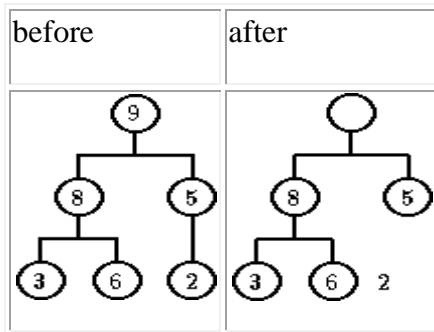
3. Insert the new element to the vacant node



4. A complete tree of n nodes has depth $\lceil \log n \rceil$, hence the time complexity is $O(\log n)$

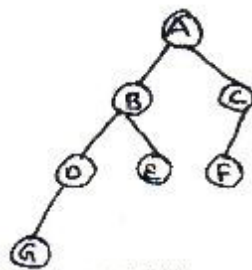
Deleting an Element

1. Delete the value from the root node, and delete the last node while saving its value.



4. Define binary trees. For the given tree find the following : (June 2013) (june/july 2015)

- i. siblings
- ii. leaf nodes
- iii. non – leaf nodes
- iv. ancestors
- v. level of trees



Soln:

The tree elements are called "nodes". The lines connecting elements are called "branches". Nodes without children are called leaf nodes, "end-nodes", or "leaves".

Every finite tree structure has a member that has no superior. This member is called the "root" or root node. The root is the starting node. But the converse is not true: infinite tree structures may or may not have a root node.

The names of relationships between nodes model the kinship terminology of family relations. The gender-neutral names "parent" and "child" have largely displaced the older "father" and "son" terminology, although the term "uncle" is still used for other nodes at the same level as the parent.

- A node's "parent" is a node one step higher in the hierarchy (i.e. closer to the root node) and lying on the same branch.
- "Sibling" ("brother" or "sister") nodes share the same parent node.
- A node's "uncles" are siblings of that node's parent.
- A node that is connected to all lower-level nodes is called an "ancestor". The connected lower-level nodes are "descendants" of the ancestor node.

In the example, "encyclopedia" is the parent of "science" and "culture", its children. "Art" and "craft" are siblings, and children of "culture", which is their parent and thus one of their ancestors. Also, "encyclopedia", as the root of the tree, is the ancestor of "science", "culture", "art" and "craft". Finally, "science", "art" and "craft", as leaves, are ancestors of no other node.

Tree structures can depict all kinds of taxonomic knowledge, such as family trees, the biological evolutionary tree, the evolutionary tree of a language family, the grammatical structure of a language (a key example being $S \rightarrow NP VP$, meaning a sentence is a noun phrase and a verb phrase, with each in turn having other components which have other components), the way web pages are logically ordered in a web site, mathematical trees of integer sets, et cetera.

The Oxford English Dictionary records use of both the terms "tree structure" and "tree-diagram" from 1965 in Noam Chomsky's *Aspects of the Theory of Syntax*.^[1]

In a tree structure there is one and only one path from any point to any other point.

Computer science uses tree structures extensively (*see Tree* (data structure) and telecommunications.)

For a formal definition see set theory, and for a generalization in which children are not necessarily successors, see prefix order.

MODULE 5

Graph

1. Explain the following with an example: i) forest & its traversals. Explain the different method of traversing a tree with following tree

ii) graph iii) winner tree .iv) Selelction trees (june/jul 2015).

Soln:

In computer science, a **graph** is an abstract data type that is meant to implement the undirected graph and directed graph concepts from mathematics.

A graph data structure consists of a finite (and possibly mutable) set of *vertices* or *nodes* or *points*, together with a set of unordered pairs of these vertices for an undirected graph or a set of ordered pairs for a directed graph. These pairs are known as *edges*, *arcs*, or *lines* for an undirected graph and as *arrows*, *directed edges*, *directed arcs*, or *directed lines* for a directed graph. The vertices may be part of the graph structure, or may be external entities represented by integer indices or references.

A graph data structure may also associate to each edge some *edge value*, such as a symbolic label or a numeric attribute (cost, capacity, length, etc.).

2. Describe the binary search tree with an example. Write a iterative & recursive function to search for a key value in a binary search tree. Define ADT of binary search tree. Write BST for the elements { 22,28,20,25,22,15,18,10,14} {14, 5, 6, 2, 18, 20, 16, 18, -1, 21} (june/jul 2014).

Soln:

Searching a binary search tree for a specific key can be programmed recursively or iteratively.

We begin by examining the root node. If the tree is *null*, the key we are searching for does not exist in the tree. Otherwise, if the key equals that of the root, the search is successful and we return the node. If the key is less than that of the root, we search the left subtree. Similarly, if the key is greater than that of the root, we search the right subtree. This process is repeated until the key is found or the remaining subtree is *null*. If the searched key is not found before a *null* subtree is reached, then the key is not present in the tree. This is easily expressed as a recursive algorithm:

```
1 def search_recursively(key, node):
2     if node is None or node.key == key:
3         return node
4     elif key < node.key:
5         return search_recursively(key, node.left)
6     else: # key > node.key
7         return search_recursively(key, node.right)
```

The same algorithm can be implemented iteratively:

```
1 def search_iteratively(key, node):
2     current_node = node
3     while current_node is not None:
4         if key == current_node.key:
5             return current_node
6         elif key < current_node.key:
7             current_node = current_node.left
8         else: # key > current_node.key:
9             current_node = current_node.right
10    return None
```

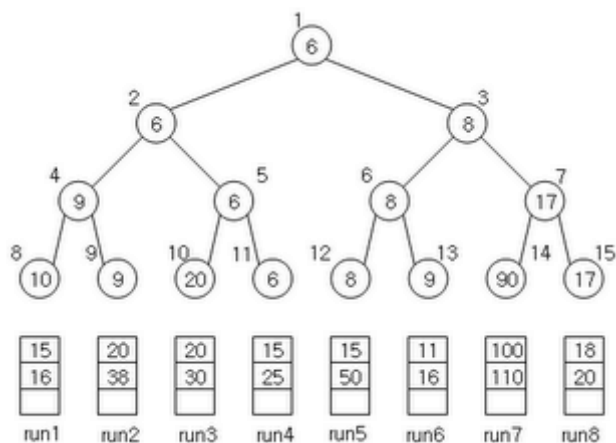
These two examples rely on the order relation being a total order.

If the order relation is only a total preorder a reasonable extension of the functionality is the following: also in case of equality search down to the leaves in a direction specifiable by the user. A binary tree sort equipped with such a comparison function becomes stable.

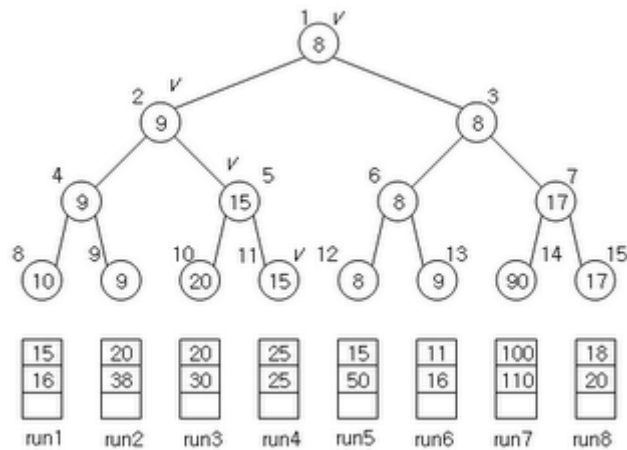
Because in the worst case this algorithm must search from the root of the tree to the leaf farthest from the root, the search operation takes time proportional to the tree's *height* (see tree terminology). On average, binary search trees with n nodes have $O(\log n)$ height.^[a] However, in the worst case, binary search trees can have $O(n)$ height, when the unbalanced tree resembles a linked list (degenerate tree).

3. Explain selection trees, with suitable example. (june/jul 2014)

Soln: Suppose we have k ordered sequences that are to be merged into a single ordered sequence. Each sequence consists of some number of records and is in nondecreasing order of a designated field called the key. An ordered sequence is called a run. Let n be the number of records in the k runs together. The merging task can be accomplished by repeatedly outputting the record with the smallest key. The smallest has to be found from k possibilities and it could be the leading record in any of the k -runs. The most direct way to merge k -runs would be to make $k-1$ comparisons to determine the next record to output. For $k > 2$, we can achieve a reduction in the number of comparisons needed to find the next smallest element by using the idea of a selection tree. A selection tree is a binary tree where each node represents the smaller of its two children.



Selection tree for $k=8$ showing the first three keys in each of the eight runs



4. What is a forest. With suitable example illustrate how you transform a forest into a binary tree. (june/jul 2014)

Soln:

A **forest** is a disjoint union of trees.

The various kinds of data structures referred to as trees in computer science have underlying graphs that are trees in graph theory, although such data structures are generally **rooted trees**. A rooted tree may be directed, called **adirected rooted tree**, either making all its edges point away from the root—in which case it is called an **arborescence**, **branching**, or **out-tree**—, or making all its edges point towards the root—in which case it is called an **anti-arborescence** or **in-tree**. A rooted tree itself has been defined by some authors as a directed graph.

5. What is a winner tree ? Explain with suitable example a winner tree for k=8. (june/jul 2015).

Soln:

is a form of min (max) heap which is a complete binary tree. Every external node represents a player and internal node represents winner. In a tournament tree every internal node contains winner and every leaf node contains one player.

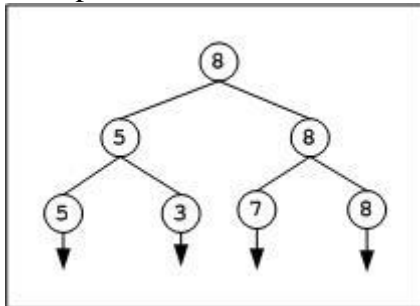
There will be $N - 1$ internal nodes in a binary tree with N leaf (external) nodes. For details see this post (put $n = 2$ in equation given in the post).

It is obvious that to select the best player among N players, $(N - 1)$ players to be eliminated, i.e. we need minimum of $(N - 1)$ games (comparisons). Mathematically we can prove it. In a binary tree $I = E - 1$, where I is number of internal nodes and E is number of external nodes. It means to find maximum or minimum element of an array, we need $N - 1$ (internal nodes) comparisons.

Second Best Player

The information explored during best player selection can be used to minimize the number of comparisons in tracing the next best players. For example, we can pick second best player in $(N + \log_2 N - 2)$ comparisons. For details read this comment.

The following diagram displays a tournament tree (*winner tree*) as a max heap. Note that the concept of *loser trees* is different.



The above tree contains 4 leaf nodes that represent players and have 3 levels 0, 1 and 2. Initially 2 games are conducted at level 2, one between 5 and 3 and another one between 7 and 8. In the next move, one more game is conducted between 5 and 8 to conclude the final winner. Overall we need 3 comparisons. For second best player we need to trace the candidates participated with final winner, that leads to 7 as second best.

Median of Sorted Arrays

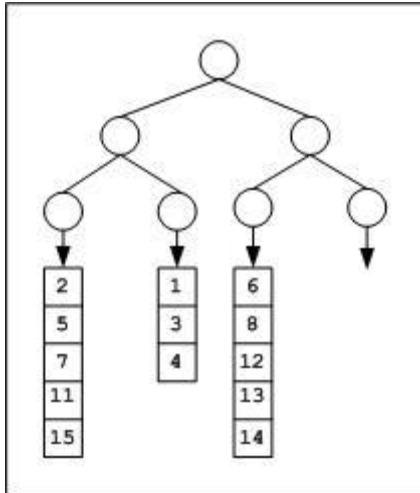
Tournament tree can effectively be used to find median of sorted arrays. Assume, given M sorted arrays of equal size L (for simplicity). We can attach all these sorted arrays to the tournament tree, one array per leaf. We need a tree of height **CEIL** ($\log_2 M$) to have at least M external nodes. Consider an example. Given 3 ($M = 3$) sorted integer arrays of maximum size 5 elements.

{ 2, 5, 7, 11, 15 } ---- Array1

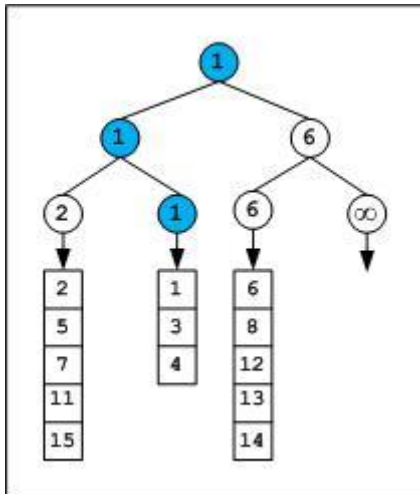
{ 1, 3, 4 } ---- Array2

{ 6, 8, 12, 13, 14 } ---- Array3

What should be the height of tournament tree? We need to construct a tournament tree of height $\log_2 3 = 1.585 = 2$ rounded to next integer. A binary tree of height 2 will have 4 leaves to which we can attach the arrays as shown in the below figure.



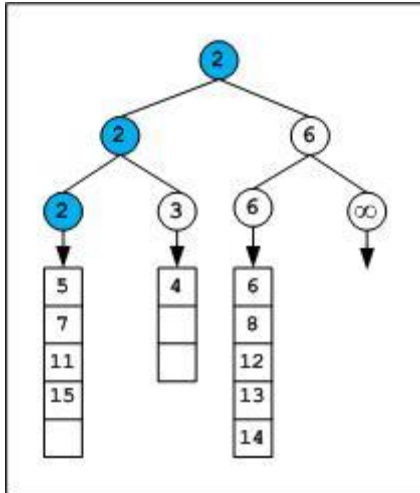
After the first tournament, the tree appears as below,



We can observe that the winner is from Array2. Hence the next element from Array2 will dive-in and games will be played along the winner path of previous tournament.

Note that infinity is used as sentinel element. Based on data being hold in nodes we can select the sentinel character. For example we usually store the pointers in nodes rather than keys, so NULL can serve as sentinel. If any of the array exhausts we will fill the corresponding leaf and upcoming internal nodes with sentinel.

After the second tournament, the tree appears as below,



The next winner is from Array1, so next element of Array1 array which is 5 will dive-in to the next round, and next tournament played along the path of 2.

The tournaments can be continued till we get median element which is $(5+3+5)/2 = 7$ th element. Note that there are even better algorithms for finding median of union of sorted arrays, for details see the related links given below.

In general with M sorted lists of size $L_1, L_2 \dots L_m$ requires time complexity of $O((L_1 + L_2 + \dots + L_m) * \log M)$ to merge all the arrays, and $O(m * \log M)$ time to find median, where m is median position.

Select smallest one million elements from one billion unsorted elements:

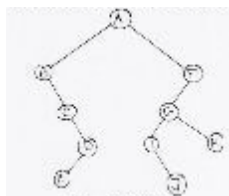
As a simple solution, we can sort the billion numbers and select first one million.

On a limited memory system sorting billion elements and picking the first one million seems to be impractical. We can use tournament tree approach. At any time only elements of tree to be in memory.

Split the large array (perhaps stored on disk) into smaller size arrays of size one million each (or even smaller that can be sorted by the machine). Sort these 1000 small size arrays and store them on disk as individual files. Construct a tournament tree which can have atleast 1000 leaf nodes (tree to be of height 10 since $2^9 < 1000 < 2^{10}$, if the individual file size is even smaller we will need more leaf nodes). Every leaf node will have an engine that picks next element from the sorted file stored on disk. We can play the tournament tree game to extract first one million elements.

Total cost = sorting 1000 lists of one million each + tree construction + tournaments

6. Construct a binary tree having the following sequences.(dec 2014/jan 2015) (june/july 2014)



i) Preorder sequence: ABCDEFGHI

ii) Inorder sequence: BCAEDGHFI

1) pre order: $/+*1\ \$\ 2\ 3\ 4\ 5$

A B D G C E H I F

2) IN ORDER : $1 + 2 * 3 \$ 4 - 5$

D G B A H E I C E