2022

ALL-IN-ONE INTERVIEW QUESTIONS

C INTERVIEW QUESTIONS

@JESWINAUGUSTINE

Disclaimer: These are the personal notes prepared by me while preparing for my future reference. It includes most the common questions asked during the interview and answers have been taken what's found in various articles over the internet. Please do use it for your personal reading only and refrain from sharing it over public domains such as LinkedIn, Facebook, Google Drive etc as it might infringe upon various copyrights. I have tried my best to avoid any errors, still if you find any please let me know by dropping a mail to jeswinaugustine@yahoo.com so that it can be rectified.

WWW.JESWIN.COM

Contents

1. GENERAL QUESTIONS

1) Output of printf("%d")?

There is no variable is provided after %d so compiler will show **garbage** value.

2) What are the return values of printf and scanf?

The **printf** function upon successful return, returns the **number of characters printed** in output device. So, printf("A") will return 1. The **scanf** function returns the number of input items **successfully matched** and assigned, which can be fewer than the format specifiers provided. It can also return zero in case of early matching failure.

3) Printf format identifiers ?

%d %i	Decimal signed integer.
%o	Octal integer.
%x %X	Hex integer.
%u	Unsigned integer.
%c	Character.
%s	String. See below.
%f	double
%e %E	double.
%g %G	double.
%p	pointer.
%n	Number of characters written by this printf.
%%	%. No argument expected.

4) Consider the two structures Struct A and Struct B given below. What will be size of these structures?

```
struct A
{
unsigned char c1 : 3;
unsigned char c2 : 4;
unsigned char c3 : 1;
}a;

struct A
{
unsigned char c1 : 3;
unsigned char : 0;
unsigned char c2 : 4;
unsigned char c3 : 1;
}a;
```

Ans:

The size of the structures will be **1 and 2.** In case of first structure, the members will be assigned a byte as follows -

7	6	5	4	3	2	1	0
c3	c2	c2	c2	c2	c1	c1	c1

But in case of second structure -

8	7	6	5	4	3	2	1	0
c3	c2	c2	c2	c2		c1	c1	c1

The :**0 field (0 width bit field)** forces the **next bit width member** assignment to **start from the next nibble.** By doing so, the c3 variable will be assigned a bit in the next byte, resulting the size of the structure to 2.

5) How to pack a structure?

We can pack any structure using **__attribute__((__packed__))** in gcc. Example -

```
typdef struct A __attribute __((__packed__))
{
    char c;
    int i;
}B;
```

6) Structure Member Alignment, Padding and Data Packing

```
#include <stdio.h>

// Alignment requirements
// (typical 32 bit machine)

// char         1 byte
// short int    2 bytes
// int          4 bytes
// double       8 bytes

// structure A
typedef struct structa_tag
{
  char       c;
  short int  s;
} structa_t;

// structure B
typedef struct structb_tag
{
  short int  s;
  char       c;
```

```
  int       i;
} structb_t;

// structure C
typedef struct structc_tag
{
  char      c;
  double    d;
  int       s;
} structc_t;

// structure D
typedef struct structd_tag
{
  double    d;
  int       s;
  char      c;
} structd_t;

int main()
{
  printf("sizeof(structa_t) = %lu\n", sizeof(structa_t));
  printf("sizeof(structb_t) = %lu\n", sizeof(structb_t));
  printf("sizeof(structc_t) = %lu\n", sizeof(structc_t));
  printf("sizeof(structd_t) = %lu\n", sizeof(structd_t));

  return 0;
}
```

Ans:

```
sizeof(structa_t) = 4
sizeof(structb_t) = 8
sizeof(structc_t) = 24
sizeof(structd_t) = 16
```

Read More on: https://www.geeksforgeeks.org/structure-member-alignment-padding-and-data-packing

7) Can structures be assigned to variables and passed to and from functions?

Yes, they can!

Note: When structures are passed, returned or assigned, the copying is done only at one level (The data pointed to by any pointer fields is **not copied!.**

8) Can we directly compare two structures using the == operator?

No, you cannot!

The only way to compare two structures is to **write your own function** that compares the **structures field by field.**

9) Can we pass constant values to functions which accept structure arguments?

No. Use a **temporary structure** variable.

10) Why do structures get padded? Why does sizeof() return a larger size?

Padding enables the **CPU** to **access the members faster**. If they are **not aligned** (say to word boundaries), then **accessing them might take up more time**. So the padding results in faster access. This is also required to ensure that **alignment properties are preserved when an array of contiguous structures** is allocated. Even if the **structure is not part of an array, the end padding remains**, so that sizeof() can always return a consistent size.

11) Can we determine the offset of a field within a structure and directly access that element?

The offsetof() macro(<stddef.h>) does exactly this.

Eg : **offset_of_a = offsetof(struct mystruct, a)**

12) What are bit fields in structures?

To avoid wastage of memory in structures, a group of bits can be packed together into an integer and it's called a bit field.

struct tag-name {

data-type name1:bit-length;

data-type name2:bit-length;

...

...

data-type nameN:bit-length;

}

13) What is a union? Where does one use unions? What are the limitations of unions?

A union is a special data type available in C that **allows to store different data types in the same memory location**.

Syntax for Declaring a C union

Syntax for declaring a union is same as that of declaring a structure except the keyword struct.

Note: Size of the union is the size of its largest field because sufficient number of bytes must be reserved to store the largest sized field. To access the fields of a union, use dot(.) operator i.e., the variable name followed by dot operator followed by field name
Unions are specially useful where the api which is expecting multiple type of Messages to handle but can receive one Message at a time.

14) How to prevent same header file getting included multiple times?

```
#ifndef _HDRFILE_H_
#define _HDRFILE_H_
#endif
```

15) Which is better #define or enum?

Ans:

- Enum values can be automatically generated by compiler if we let it. But all the define values are to be mentioned specifically.
- Macro is preprocessor, so unlike enum which is a compile time entity, source code has no idea about these macros. So, the enum is better if we use a debugger to debug the code.
- If we use enum values in a switch and the default case is missing, some compiler will give a warning.
- Enum always makes identifiers of type int. But the macro let us choose between different integral types.
- Macro does not specifically maintain scope restriction unlike enum

16) typedef vs #define

#define is a C-directive which is also used to **define the aliases** for various data types similar to **typedef** but with the following differences −

- **typedef** is limited to **giving symbolic** names to types only where as **#define** can be used to define alias for values as well, q., you can define 1 as ONE etc.
- **typedef** interpretation is performed by the **compiler** whereas **#define** statements are processed by the **pre-processor**.

17) What are inline functions?

Inline function is a function which is expanded at the calling block. it is not treated as a separate function by the compiler, but it is up to to the compiler to decide whether an inline function will be expanded or not depending on the size of the function. Ex:

```
inline function_name ()
{
        //function definition
}
```

18) Why is sizeof() an operator and not a function?

sizeof() is a **compile time operator**. To calculate the size of an object, we need the **type information**, which is **available only at compile time**. At the end of the compilation phase, the resulting object code doesn't have (or not required to have) the type information.

Of course, type information can be stored to access it at run-time, but this results in bigger object code and less performance. And most of the time, we don't need it. All the runtime environments that support run time type identification (RTTI) will retain type information even after compilation phase.

On a side note, something like this is illegal...

```
printf("%u\n", sizeof(main));
```

This asks for the size of the main function, which is actually illegal: 6.5.3.4.

The sizeof operator The sizeof operator shall not be applied to an expression that has function type....

19) If I have the name of a function in the form of a string, how can I invoke that function?

Keep a table of names and their function pointers:

```
int myfunc1(), myfunc2();
struct {
        char *name;
        int (*func_ptr)();
} func_table[] = {"myfunc1", myfunc1, "myfunc2", myfunc2,};
```

Search the table for the name, and call via the associated function pointer.

20) What does the error, invalid redeclaration of a function mean?

If there is no declaration in scope then it is assumed to be **declared as returning an int and without any argument type information**. This can lead to discrepancies if the function is later declared or defined. Such functions must be declared before they are called. Also check if there is another **function in some header file with the same name**.

21) Write a C program to check your system endianness?

Big endian and *little endian* are two formats to store multibyte data types into computer's memory.

These two formats are also called ***network byte order*** and ***host byte order*** respectively. In a multibyte data type such as int or long or any other multibyte data type the **right most byte** is called **least significant byte** and **the left most byte** is called **most significant byte**.

In **big endian format** the **most significant byte is stored first**, thus gets stored at the smallest address byte, while in **little endian format** the **least significant byte is stored first**.

```
/*
  Function check_for_endianness() returns 1, if architecture
  is little endian, 0 in case of big endian.
```

```
 */

int check_for_endianness()
{
  unsigned int x = 1;
  char *c = (char*) &x;
  return (int)*c;
}

int main() {

   if(check_for_endianness()) {
      printf("LITTLE ENDIAN \n");
   } else {
      printf("BIG ENDIAN \n");
   }

   return 0;
}
```

22) Print the output of this recursive function C Program?

```
void add(int a, int b)
{
    int c;
    c = a + b;
    add (1,1);
}
```

Ans:

The function will lead to stack overflow. Here, there is no terminating condition and that is why it leads to infinite recursion which results in STACK OVERFLOW. When a function is called,

- It will evaluate actual parameter expressions.
- It will allocate memory for local variables.
- It will store caller's current address of execution.
- Then it executes rest of the function body and reaches end and returns to the caller's address.

23) How should we write a multi-statement macro?

This is the **correct way** to write a multi statement macro.

```
#define MYMACRO(argument1, argument2) do { \
        stament1; \
        stament2; \
        } while(1) /* no trailing semicolon */
```

24) How can I write a macro which takes a variable number of arguments?

One can define the macro with a single, parenthesized "argument" which in the macro expansion becomes the entire argument list, parentheses and all, for a function such as printf():

```
#define DEBUG(args) (printf("DEBUG: "), printf args) if(n != 0) DEBUG(("n is %d\n", n));
```

The caller must always remember to use the extra parentheses. Other possible solutions are to use different macros depending on the number of arguments. C9X will introduce formal support for function-like macros with variable-length argument lists. The notation ... will appear at the end of the macro "prototype" (just as it does for varargs functions).

25) What is the token pasting operator and stringizing operator in C?

Token pasting operator ANSI has introduced a well-defined token-pasting operator, ##, which can be used like this:

```
#define f(g,g2) g##g2
main() {
  int var12=100;
  printf("%d",f(var,12));
}

O/P 100
```

Stringizing operator

```
#define sum(xy) printf(#xy " = %f\n",xy);

main() {
  sum(a+b);
  // As good as printf("a+b = %f\n", a+b);
}
```

26) Define a macro called SQR which squares a number.

```
#define SQR(x) ((x)*(x))
// Use parathesis to give right precedence
```

27) Guess the output of this C program using "##" Operator?

```
#include <stdio.h>
#define A(a,b) a##b
#define B(a) #a
```

```
#define C(a) B(a)

main()
{
 printf("%s\n",C(A(1,2)));
 printf("%s\n",B(A(1,2)));
}

O/P:
12
A(1,2)
```

28) What are #pragmas?

The directive provides a single, well-defined "escape hatch" which can be used for all sorts of (nonportable) implementation-specific controls and extensions: source listing control, structure packing, warning suppression (like lint's old /* NOTREACHED */ comments), etc.
For example
#pragma once

inside a header file is an extension implemented by some preprocessors to help make header files idempotent (to prevent a header file from included twice).

29) What purpose do #if, #else, #elif, #endif, #ifdef, #ifndef serve?

30) What should go in header files? How to prevent a header file being included twice?
31) What's wrong with including header files twice?
32) Is there a limit on the number of characters in the name of a header file?
33) Is it acceptable to declare/define a variable in a C header?

34) What is the difference between if(0 == x) and if(x == 0)?

Nothing!. But, it's a good trick to prevent the common error of writing **if(x = 0)**

35) Should we use goto or not?

You should use goto wherever it suits your needs well. There is nothing wrong in using them. Really. There are cases where each function must have a single exit point. In these cases, it makes much sense to use goto.

```
myfunction() {
if(error_condition1) {
// Do some processing.
goto failure;
}
if(error_condition2) {
// Do some processing.
```

```
goto failure;
}
success:
return(TRUE);
failure:
// Do some cleanup.
return(FALSE);
}
```

Also, a lot of coding problems lend very well to the use of goto. The only argument against goto is that it can make the code a **little un-readable**. But if its commented properly, it works quite fine.

36) Is ++i really faster than i = i + 1?

In C, Any good compiler will and should generate identical code for ++i, i += 1, and i = i + 1. Compilers are meant to optimize code.

The programmer should not be bother about such things. Also, it depends on the processor and compiler you are using. One needs to check the compiler's assembly language output, to see which one of the different approcahes are better, if at all.

But for C++, it can be bit tricky. The ++ operator may or may not be defined as a function. For primitive types (int, double, ...) the operators are built in, so the compiler will probably be able to optimize your code. But in the case of an object that defines the ++ operator things are different.

37) What do lvalue and rvalue mean?

An lvalue is an expression that could appear on the left-hand sign of an assignment (An object that has a location). An rvalue is any expression that has a value (and that can appear on the right-hand sign of an assignment).

The lvalue refers to the left-hand side of an assignment expression. It must always evaluate to a memory location. The rvalue represents the right-hand side of an assignment expression; it may have any meaningful combination of variables and constants.

38) What does the term cast refer to? Why is it used?

Casting is a mechanism built into C that allows the programmer to force the conversion of data types. This may be needed because most C functions are very particular about the data types they process. A programmer may wish to override the default way the C compiler promotes data types.

39) What is the difference between a statement and a block?

A statement is a single C expression terminated with a semicolon. A block is a series of statements, the group of which is enclosed in curly-braces.

40) Can comments be nested in C?

NO

41) What is type checking?

The process by which the C compiler ensures that functions and operators use data of the appropriate type(s). This form of check helps ensure the semantic correctness of the program.

42) How will implement Your Own sizeof in C

If only one byte is increased, that means it is character, if the increased value is 4, then it is int or float and so on. So by taking the difference between &x + 1 and &x, we can get the size of x.

Here we will use macro as the datatype is not defined in the function. And one more thing, we are casting using (char*) so, it will tell us that how many character type data can be placed in that place. As the character takes one byte of data.

```
#include <stdio.h>
#define my_sizeof(type) (char *)(&type+1)-(char*)(&type)
main(void) {
   int x = 10;
   char y = 'f';
   double z = 254748.23;
   printf("size of x: %d\n", my_sizeof(x));
   printf("size of y: %d\n", my_sizeof(y));
   printf("size of z: %d\n", my_sizeof(z));
```

43) Why can't you nest structure definitions?

This is a trick question! You can nest structure definitions.

```
struct salary {
    char empname[20];
    struct {
        int dearness;
    } allowance;
} employee;
```

44) What is a forward reference?

It is a reference to a variable or function before it is defined to the compiler. The cardinal rule of structured languages is that everything must be defined before it can be used. There are rare occasions where this is not possible. It is possible (and sometimes necessary) to define two functions in terms of each other. One will obey the cardinal rule while the other will need a forward declaration of the former in order to know of the former's existence.

45) What is the difference between the & and && operators and the | and || operators?

& and | are bitwise AND and OR operators respectively. They are usually used to manipulate the contents of a variable on the bit level. && and || are logical AND and OR operators respectively. They are usually used in conditionals

46) Is C case sensitive (ie: does C differentiate between upper and lower case letters)?

Yes, ofcourse!

47) Can goto be used to jump across functions?

No! This wont work

```
main() {
  int i=1;
  while (i<=5) {
    printf("%d",i);
    if (i>2)
       goto here;
    i++;
  }
}
fun() {
  here: printf("PP");
}
```

48) What is the difference between a deep copy and a shallow copy?

49) Is using exit() the same as using return?

No. The exit() function is used to exit your program and return control to the operating system. The return statement is used to return from a function and return control to the calling function. If you issue a return from the main() function, you are essentially

returning control to the calling function, which is the operating system. In this case, the return statement and exit() function are similar.

50) Differentiate between a linker and linkage?

A linker converts an object code into an executable code by linking together the necessary build in functions. The form and place of declaration where the variable is declared in a program determine the linkage of variable.

51) What is a source code, object code and executable code?

- **Source code** is a program written in a high-level language, such as C.
- **Object code** is a translated version of source code in machine language.
- **Executable code** is a final version of a program, in machine language, that is executed on user's command.

52) What is the difference between structure and union?

- Union allocates the memory equal to the maximum memory required by the member of the union, but structure allocates the memory equal to the total memory required by the members.
- In union, one block is used by all the member of the union but in case of structure, each member has their own memory space.
- Union is best in the environment where memory is less as it shares the memory allocated.
- But structure cannot implement in shared memory.
- As memory is shared, ambiguity is more in union, but less in structure.
- Self-referential union cannot be implemented in any data structure, but self-referential structure can be implemented.

53) How do you use union, provide use case?

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.

Definition	Example

`union [union tag] {`
`  member definition;`
`  member definition;`
`  ...`
`  member definition;`
`} [one or more union variables];`
	`union Data {`
	`  int i;`
	`  float f;`
	`  char str[20];`
	`} data;`

54) What is the purpose of register keyword?

It is used to make the **computation faster**. The register keyword tells the compiler to store the variable onto the **CPU register** if space on the register is available. However, this a very old technique. **Today's processors are smart enough to assign the registers themselves** and hence using the register keywords can actually slowdown the operations if the usage is incorrect.
The keyword register instructs the compiler to persist the variable that is being declared in a CPU registers.

55) What is the difference between #include <filename> and #include “filename”?

For `#include <filename>` the pre-processor searches in an implementation dependent manner, normally in search directories **pre-designated by the compiler/IDE**. This method is normally used to include standard library header files.

For `#include "filename"` the pre-processor searches **first in the same directory** as the file containing the directive, and then follows the search path used for the `#include <filename>` form. This method is normally used to include programmer-defined header files.

56) Tell about Storage Classes in C

There are total of four types of standard storage classes. The table below represents the storage classes in C.

Storage class	Purpose
auto	It is a default storage class.
extern	It is a global variable.
static	It is a local variable which is capable of returning a value even when control is transferred to the function call.
register	It is a variable which is stored inside a Register.

57) Do Global variables start out as zero?
58) When should the register modifier be used?
59) Does C have boolean variable type?
60) What does the typedef keyword do?

61) Mention about volatile keyword and usage.

The volatile keyword is intended to prevent the compiler from applying any optimizations on objects that can change in ways that cannot be determined by the compiler.

A variable should be declared volatile whenever its value could change unexpectedly. In practice, only three types of variables could change:
1. Memory-mapped peripheral registers
2. Global variables modified by an interrupt service routine
3. Global variables accessed by multiple tasks within a multi-threaded application

62) What is Cross compilation?

Cross compilation is a process of compiling and creating executable code for a platform other than the one on which the compiler is running.

Cross compilation is a mechanism of compiling a source code which will be used on different architecture rather than the architecture on which it compile.

63) Difference between Native and Cross compiler

- A native compiler is one that compiles programs for the same architecture or operating system that it is running on. For instance, a compiler running on an x86 processor and creating x86 binaries.
- A cross-compiler is one that compiles binaries for architectures other than its own, such as compiling ARM binaries on a Intel's x86 processor.
- A "cross compiler" executes in one environment and generates code for another. A "native compiler" generates code for its own execution environment.

64) Why we use cross compilation:

Suppose we have a micro controller (eg raspberry pie) which have 900 MHz frequency and less ram size. Now we want a binary or library for some application on it. So that we compile a source code for it. Now as above mention that it has less frequency and ram size so that it take more time for execution. For small code, its ok! but for larger code, its time consuming.

So, we have always an option for cross compilation. Now we have CPU for our desktop which have more frequency and ram size than above mentioned. So, it take less time for execution and make binaries and library for the other architecture.

Eg: Suppose we have x86 arch on which we compile the source code for arm arch. So here we use cross compilation.

2. POINTERS

1) What does *p++ do? Does it increment p or the value pointed by p?

2) What is the difference between const char* p and char const* p?

const char* p :

a) Character pointed by pointer variable p is constant.
b) **This value can't be changed** but we can initialize p with other memory location.

char const* p :

a) pointer p is constant not the character referenced by it.
b) **Address pointed by the pointer can't be changed** but we **can change the value** of the character pointed by p.

3) Print the output of value assignment to a C constant program?

```
#include<stdio.h>
void main()
{
int const * p=5;
printf("%d",++(*p));
}
```

Ans:

This program will give a compiler error: Cannot modify a constant value. Here p is a pointer to a "constant integer". But in the printf function, we tried to change the value of the "constant integer" which is not allowed in C and thus the compilation error.

4) Is 5[array] the same as array[5]?

The C standard defines the [] operator as follows: a[b] == *(a + b)

Therefore a[5] will evaluate to: *(a + 5)

and 5[a] will evaluate to: *(5 + a)

5) How to dynamically allocate a 2D array in C?

a) Using a single pointer:
A simple way is to allocate memory block of size r*c and access elements using simple pointer arithmetic.

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

void main()
{
    int r = 3, c = 4; //Taking number of Rows and Columns
    int *ptr, count = 0, i;

    ptr = (int*)malloc((r * c) * sizeof(int)); //Dynamically Allocating
Memory
    for (i = 0; i < r * c; i++)
    {
        ptr[i] = i + 1; //Giving value to the pointer and simultaneously
printing it.
        printf("%d ", ptr[i]);
        if ((i + 1) % c == 0)
        {
```

```
                printf("\n");
            }
        }
        free(ptr);
    }
```

b) Using an array of pointers

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

int main()
{
    int r = 3, c = 4, i, j, count;

    int* arr[r];
    for (i = 0; i < r; i++)
        arr[i] = (int*)malloc(c * sizeof(int));

    // Note that arr[i][j] is same as *(*(arr+i)+j)
    count = 0;
    for (i = 0; i < r; i++)
        for (j = 0; j < c; j++)
            arr[i][j] = ++count; // Or *(*(arr+i)+j) = ++count

    for (i = 0; i < r; i++)
        for (j = 0; j < c; j++)
            printf("%d ", arr[i][j]);

    /* Code for further processing and free the
      dynamically allocated memory */

    for (int i = 0; i < r; i++)
        free(arr[i]);

    return 0;
}
```

c) Using pointer to a pointer

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

int main()
{
    int r = 3, c = 4, i, j, count;

    int** arr = (int**)malloc(r * sizeof(int*));
    for (i = 0; i < r; i++)
        arr[i] = (int*)malloc(c * sizeof(int));

    // Note that arr[i][j] is same as *(*(arr+i)+j)
    count = 0;
    for (i = 0; i < r; i++)
        for (j = 0; j < c; j++)
```

```
            arr[i][j] = ++count; // OR *(*(arr+i)+j) = ++count

    for (i = 0; i < r; i++)
        for (j = 0; j < c; j++)
            printf("%d ", arr[i][j]);

    /* Code for further processing and free the
       dynamically allocated memory */

    for (int i = 0; i < r; i++)
        free(arr[i]);

    free(arr);

    return 0;
}
```

d) Using double pointer and one malloc() call

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

int main()
{
    int r=3, c=4, len=0;
    int *ptr, **arr;
    int count = 0,i,j;

    len = sizeof(int *) * r + sizeof(int) * c * r;
    arr = (int **)malloc(len);

    // ptr is now pointing to the first element in of 2D array
    ptr = (int *)(arr + r);

    // for loop to point rows pointer to appropriate location in 2D array
    for(i = 0; i < r; i++)
        arr[i] = (ptr + c * i);

    for (i = 0; i < r; i++)
        for (j = 0; j < c; j++)
            arr[i][j] = ++count; // OR *(*(arr+i)+j) = ++count

    for (i = 0; i < r; i++)
        for (j = 0; j < c; j++)
            printf("%d ", arr[i][j]);

    return 0;
}
```

6) What is a function pointer?

```
#include <stdio.h>
// A normal function with an int parameter
// and void return type
```

```
void fun(int a)
{
   printf("Value of a is %d\n", a);
}
int main()
{
   // fun_ptr is a pointer to function fun()
   void (*fun_ptr)(int) = &fun;

   /* The above line is equivalent of following two
      void (*fun_ptr)(int);
      fun_ptr = &fun;
   */
   // Invoking fun() using fun_ptr
   (*fun_ptr)(10);

   return 0;
}
```

7) Declare a function pointer that points to a function which returns a function pointer?

```
#include<stdio.h>
void Hello();

typedef void (*FP)();

FP fun (int);

int main() {

   FP (*main_f)(int) = NULL;
   FP return_f;
   main_f = fun;
   return_f = (*fun)(30);
   (*return_f)();

   return 0;
}

void Hello() {
   printf("hello");
}

FP fun (int a) {
FP p = Hello;
printf("Number %d\n",a);
return p;
}
```

8) Function pointer can be used in place of switch case. Give example?

```
#include <stdio.h>
void add(int a, int b)
{
```

```
    printf("Addition is %d\n", a+b);
}
void subtract(int a, int b)
{
    printf("Subtraction is %d\n", a-b);
}
void multiply(int a, int b)
{
    printf("Multiplication is %d\n", a*b);
}

int main()
{
    // fun_ptr_arr is an array of function pointers
    void (*fun_ptr_arr[])(int, int) = {add, subtract, multiply};
    unsigned int ch, a = 15, b = 10;

    printf("Enter Choice:\n 0 for add\n 1 for subtract\n 2 "
           "for multiply\n");
    scanf("%d", &ch);

    if (ch > 2) return 0;

    (*fun_ptr_arr[ch])(a, b);

    return 0;
}
```

9) How to declare an array of N pointers to functions returning pointers to functions returning pointers to characters?

10) What does the error, invalid redeclaration of a function mean?

11) What is the difference between arrays and pointers?

- Pointers are used to manipulate data using the address. Pointers use * operator to access the data pointed to by them

- Arrays use subscripted variables to access and manipulate data. Array variables can be equivalently written using pointer expression.

12) How can you determine the size of an allocated portion of memory?

WE CANT!

13) What is a void pointer?

A void pointer is used for working with raw memory or for passing a pointer to an unspecified type. Some C code operates on raw memory. When C was first invented, character pointers (char *) were used for that. Then people started getting confused about when a character pointer was a string, when it was a character array, and when it was raw memory.

14) Can math operations be performed on a void pointer?

No.

15) What is a NULL pointer? How is it different from an uninitialized pointer?
16) How is a NULL pointer defined?
17) What is a null pointer assignment error?
18) What is an opaque pointer?
19) Does an array always get converted to a pointer? What is the difference between arr and &arr?
20) How does one declare a pointer to an entire array?

21) How to write functions which accept two-dimensional arrays when the width is not known before hand?
22) Is char a[3] = "abc"; legal? What does it mean?
23) If a is an array, is a++ valid?
24) How can we find out the length of an array dynamically in C?

25) Is the cast to malloc() required at all?

26) What is file pointer in C?

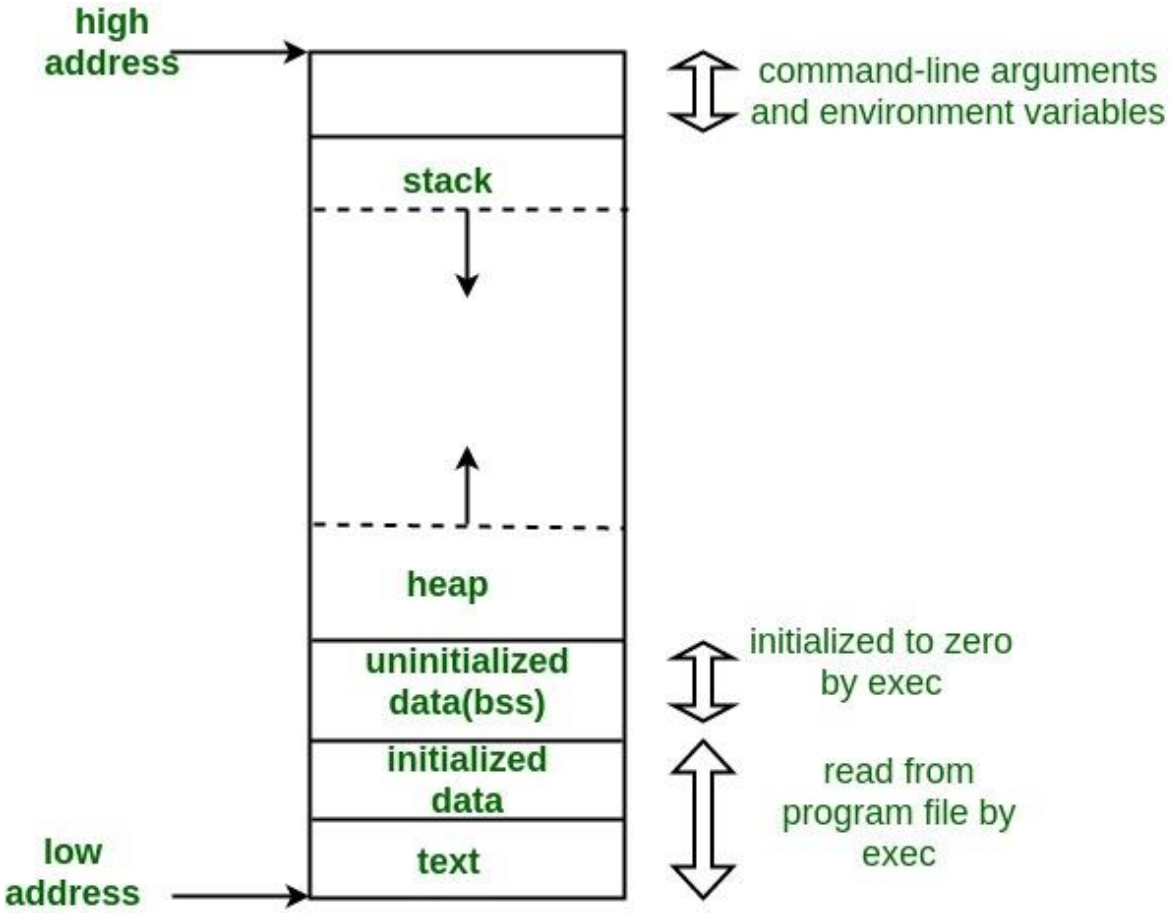
- File pointer is a pointer which is used to handle and keep track on the files being accessed.
- A new data type called FILE is used to declare file pointer. This data type is defined in stdio.h file. File pointer is declared as FILE *fp. Where, 'fb' is a file pointer.
- fopen() function is used to open a file that returns a FILE pointer. Once file is opened, file pointer can be used to perform I/O operations on the file. fclose() function is used to close the file.

What do Segmentation fault, access violation, core dump and Bus error mean?

3. MEMORY ALLOCATION

1) Describe the memory map of a C program.

This is a quite popular question. But I have never understood what exactly is the purpose of asking this question. The memory map of a C program depends heavily on the compiler used. Nevertheless, here is a simple explanation.. During the execution of a C program, the program is loaded into main memory. This area is called permanent storage area. The memory for Global variables and static variables is also allocated in the permanent storage area. All the automatic variables are stored in another area called the stack. The free memory area available between these two is called the heap. This is the memory region available for dynamic memory allocation during the execution of a program.



2) How to free a block of memory previously allocated without using free?

realloc(ptr, 0));

3) What is the point of using malloc(0)?

According to C standard, " If the size of the space requested is zero, the behavior is implementation defined: **either a null pointer is returned**, or **the behavior is as if the size were some nonzero value**, except that the returned pointer shall not be used to access an object". But there is a benefit of this. The pointer return after malloc(0) will be **valid pointer** and can be deallocated using **free()** and it will **not crash** the program.

4) How does free() know the size of memory to be deallocated?

When we use the dynamic memory allocation techniques for memory allocations, then this is done in the actual heap section. It creates one word larger than the requested size. This extra word is used to store the size. This size is used by free() when it wants to clear the memory space.

5) How to deallocate memory without using free() in C?

Solution: Standard library function realloc() can be used to deallocate previously allocated memory. Below is function declaration of "realloc()" from "stdlib.h"

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

If "size" is zero, then call to realloc is equivalent to "free(ptr)". And if "ptr" is NULL and size is non-zero then call to realloc is equivalent to "malloc(size)".

Let us check with simple example.

```
/* code with memory leak */
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
    int *ptr = (int*)malloc(10);

    return 0;
}
```

6) What are the free lists?

Answer: List nodes are created and deleted in a linked list implementation in a way that allows the Link class programmer to provide simple but efficient memory management routines. Instead of making repeated calls to new, the Link class can handle its own free list. A free list holds those list nodes that are not currently being used. When a node is deleted from a linked list, it is placed at the head of the free list. When a new element is to be added to a linked list, the free list is checked to see if a list node is available. If so, the node is taken from the free-list. If the free list is empty, the standard new operator must then be called.

7) What is a dangling pointer?

Generally, dangling pointers arise when the referencing object is deleted or deallocated, without changing the value of the pointers. It creates the problem because the pointer is still pointing the memory that is not available. When the user tries to dereference the daggling pointers than it shows the undefined behavior and can be the cause of the segmentation fault.

In simple words, we can say that dangling pointer is a pointer that not pointing a valid object of the appropriate type and it can be the cause of the undefined behavior.

```
#include<stdio.h>
#include<stdlib.h>
int main()
{
int *piData = NULL;
//creating integer of size 10.
piData = malloc(sizeof(int)* 10);
//make sure piBuffer is valid or not
if (piData == NULL)
{
// allocation failed, exit from the program
fprintf(stderr, "Out of memory!\n");
exit(1);
}
free(piData); //free the allocated memory
*piData = 10; //piData is dangling pointer
printf("%d\n",*piData);
return 0;
}
```

What is a memory leak?
What are brk() and sbrk() used for? How are they different from malloc()?

8) Implement the memcpy () function.

The memcpy function is used to copy a block of data from a source address to a destination address. Below is its prototype.

```
void *memcpy(void * destination, const void * source, size_t num);
```

The idea is to simply typecast given addresses to char *(char takes 1 byte). Then one by one copy data from source to destination. Below is implementation of this idea.

```
void myMemCpy(void *dest, void *src, size_t n)
{
    // Typecast src and dest addresses to (char *)
    char *csrc = (char *)src;
    char *cdest = (char *)dest;

    // Copy contents of src[] to dest[]
    for (int i=0; i<n; i++)
        cdest[i] = csrc[i];
}
```

9) Implement the memmove() () function.

The trick here is to use a temp array instead of directly copying from src to dest. The use of temp array is important to handle cases when source and destination addresses are overlapping.

```
//C++ program to demonstrate implementation of memmove()
#include<stdio.h>
#include<string.h>

// A function to copy block of 'n' bytes from source
// address 'src' to destination address 'dest'.
void myMemMove(void *dest, void *src, size_t n)
{
   // Typecast src and dest addresses to (char *)
   char *csrc = (char *)src;
   char *cdest = (char *)dest;

   // Create a temporary array to hold data of src
   char *temp = new char[n];

   // Copy data from csrc[] to temp[]
   for (int i=0; i<n; i++)
       temp[i] = csrc[i];

   // Copy data from temp[] to cdest[]
   for (int i=0; i<n; i++)
       cdest[i] = temp[i];

   delete [] temp;
}
```

10) What is the difference between memmove and memcpy?

`memmove` can handle overlapping memory, `memcpy` can't.

Consider

```
char[] str = "foo-bar";
memcpy(&str[3],&str[4],4); //might blow up
```

Obviously the source and destination now overlap, we're overwriting "-bar" with "bar". It's undefined behavior using `memcpy` if the source and destination overlap so in this case cases we need `memmove`.

```
memmove(&str[3],&str[4],4); //fine
```

11) What are near, far and huge pointers?

12) When should a far pointer be used?

A far pointer can refer to information outside the 64KB data segment. Typically, such pointers are used with farmalloc() and such, to manage a heap separate from where all the rest of the data lives. If you use a small-data, large-code model, you should explicitly make your function pointers far.

13) Write a sample code to show memory leak?

14) Write a sample code to show stack fault?

15) What is the output of the following program ?

```
#include<stdio.h>
#include<stdlib.h>
int main()
{
        int i, numbers[1];
        numbers[0] = 9;
        free(numbers);
        printf("\nStored integers are ");
        printf("\nnumbers[%d] = %d ", 0, numbers[0]);
        return 0;
}
```

Ans: Undefined behaviour occurs.

16) What is the output of the following program ?

```
#include<stdio.h>
#include<stdlib.h>
int main()
{
        int *j = (int*)malloc(4 * sizeof(int));
        *j = 9;
        free(j);
        printf("%d", *j);
        return 0;
}
```

Ans: Prints a garbage value

17) What is GDB?

GDB stands for GNU Project Debugger and is a powerful debugging tool for C(along with other languages like C++).It helps you to poke around inside your C programs while they are executing and also allows you to see what exactly happens when your program crashes. GDB operates on executable files which are binary files produced by compilation process.

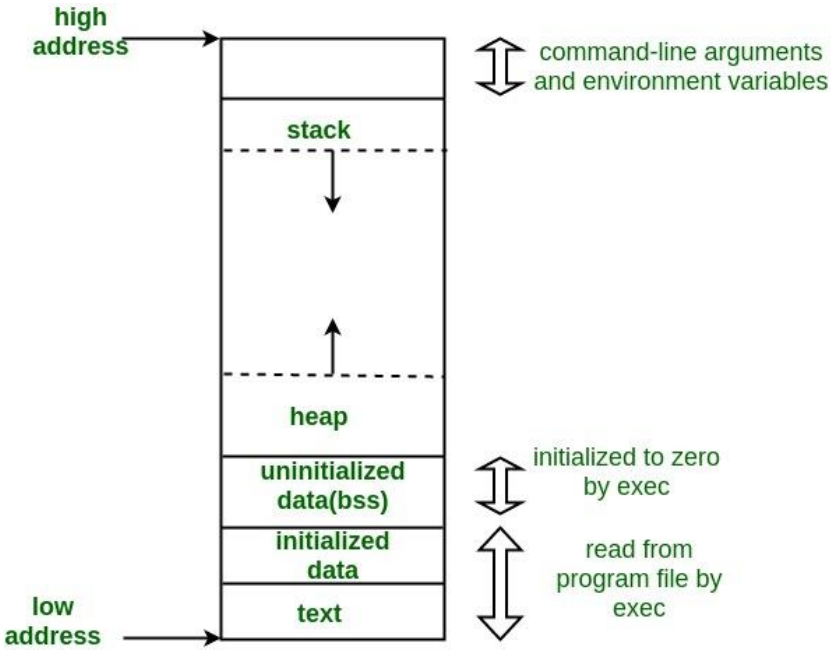
18) How will you debug using GDB?

run or r –> executes the program from start to end.
break or b –> sets breakpoint on a particular line.
disable -> disable a breakpoint.
enable –> enable a disabled breakpoint.
next or n -> executes next line of code, but don't dive into functions.
step –> go to next instruction, diving into the function.

list or l –> displays the code. print or p –> used to display the stored value. quit or q –> exits out of gdb. clear –> to clear all breakpoints. continue –> continue normal execution.

19) Memory layout of C Program

A typical memory representation of a C program consists of the following sections.
1. Text segment
2. Initialized data segment
3. Uninitialized data segment
4. Stack
5. Heap



20) Command to check memory layout?

```
[jeswin@CentOS]$ gcc memory-layout.c -o memory-layout
[jeswin@CentOS]$ size memory-layout
text       data        bss        dec        hex    filename
960        248          8       1216        4c0    memory-layout
```

Must read: https://www.geeksforgeeks.org/memory-layout-of-c-program/

21) Stack vs heap

A stack is a special area of computer's memory which stores temporary variables created by a function. In stack, variables are declared, stored and initialized during runtime. It is a temporary storage memory. When the computing task is complete, the memory of the variable will be automatically erased. The stack section mostly contains methods, local variable, and reference variables.

The heap is a memory used by programming languages to store global variables. By default, all global variable are stored in heap memory space. It supports Dynamic memory allocation.

Parameter	Stack	Heap
Type of data structures	A stack is a linear data structure.	Heap is a hierarchical data structure.
Access speed	High-speed access	Slower compared to stack
Space management	Space managed efficiently by OS so memory will never become fragmented.	Heap Space not used as efficiently. Memory can become fragmented as blocks of memory first allocated and then freed.
Access	Local variables only	It allows you to access variables globally.
Limit of space size	Limit on stack size dependent on OS.	Does not have a specific limit on memory size.
Resize	Variables cannot be resized	Variables can be resized.
Memory Allocation	Memory is allocated in a contiguous block.	Memory is allocated in any random order.
Allocation and Deallocation	Automatically done by compiler instructions.	It is manually done by the programmer.
Deallocation	Does not require to de-allocate variables.	Explicit de-allocation is needed.
Cost	Less	More
Implementation	A stack can be implemented in 3 ways simple array based, using dynamic memory, and Linked list based.	Heap can be implemented using array and trees.
Main Issue	Shortage of memory	Memory fragmentation
Locality of reference	Automatic compile time instructions.	Adequate
Flexibility	Fixed size	Resizing is possible
Access time	Faster	Slower

22) Why and When gdb says "<value optimized out>" ?

Modern compilers such as gcc are able to perform various optimizations based on syntax and semantic analysis of the code at compile time. The goal of such optimizations is to improve program execution speed and/or reduce binary size, and they typically come with the cost of extra compilation time or reduced debuggability of the program.

The <value optimized out> message in gdb is one symptom of such compiler optimizations. To view the "optimized-out" value of a variable during debugging, you need to turn off gcccompiler optimization, either on a per-variable basis, or program-wide.

Solution One :

If you are interested in a particular variable in gdb, you can delare the variable as "volatile" and recompile the code. This will make the compiler turn off compiler optimization for that variable.

volatile int quantity = 0;

If you declare a variable as volatile, it means that the variable can be modified externally (e.g., by the operating system, a signal handler, or hardware interrupt, etc). This essentially tells the compiler that the variable's value in memory can change at any time. Thus the compiler must not perform any optimization on the variable, which makes the variable accessible to gdb debugger as well.

Solution Two:

Another option to see all <optimized out> variables in gdb is of course disabling gccoptimization altogether.

Look for compilation flags (e.g., in CFLAGS) in your Makefile. You will find something like '-O1', '-O2' or '-O3', which defines various levels of gcc optimization. Remove this flag, or change it to '-O0'.

For example, change:

CFLAGS = -g -O2 -DSF_VISIBILITY -fvisibility=hidden -fno-strict-aliasing -Wall

to:

CFLAGS = -g -O0 -DSF_VISIBILITY -fvisibility=hidden -fno-strict-aliasing -Wall

23) C – Buffer manipulation functions

Functions	Description

memset()	It is used to initialize a specified number of bytes to null or any other value in the buffer
	
	`// ptr ==> Starting address of memory to be filled`
	`// x ==> Value to be filled`
	`// n ==> Number of bytes to be filled starting`
	`// from ptr to be filled`
	`void *memset(void *ptr, int x, size_t n);`

memcpy()	It is used to copy a specified number of bytes from one memory to another `void * memcpy(void *to, const void *from, size_t numBytes);`
memmove()	It is used to copy a specified number of bytes from one memory to another or to overlap on same memory. Difference between memmove and memcpy is, overlap can happen on memmove whereas memcpy should be done in non-destructive way `void * memmove(void *to, const void *from, size_t numBytes);`

memcmp()	It is used to compare specified number of characters from two buffers
memicmp()	It is used to compare specified number of characters from two buffers regardless of the case of the characters
memchr()	It is used to locate the first occurrence of the character in the specified string

4. LINKED LIST

1) How to declare a structure of a single linked list?

```
typedef struct ListNode{
   int data;
   struct ListNode *next;
} Node;
```

2) How to insert a node at the end of a single linked list?

```
void node_insert() {

   printf("************************\n Insert \n************************\n");
   int num=0;
   printf("Enter the number: \n");
   scanf("%d",&num);

   Node *temp = head;

   Node *new_node = (Node*) malloc(sizeof(Node*));
   if(new_node == NULL) {
      printf("New node insertion failed. Try again! \n");
   }
   new_node->data= num;
   new_node->next= NULL;

   if (head == NULL) {
      head=new_node;
      new_node->next=NULL;
      return;
   }
   while (temp->next != NULL){
      temp=temp->next;
   }
```

```
    temp->next=new_node;
}
```

3) How to insert a node at a given position of a single linked list?

```
void insert_at_position(int num,int pos) {
    Node *curr = head,*prev;
    int i=1;

    Node *new_node = (Node*) malloc(sizeof(Node*));
    if(new_node == NULL) {
        printf("New node insertion failed. Try again! \n");
    }
    new_node->data= num;
    new_node->next= NULL;

    if (pos == 1) {
        new_node->next=head;
        head = new_node;
        printf(" Node with %d inserted in %d\n",num,pos );
    } else {
        while( (curr != NULL) && (i<pos) ) {
            i++;
            prev=curr;
            curr=curr->next;
        }
        if (i == pos) {
            prev->next = new_node;
            new_node->next=curr;
            printf(" Node with %d inserted in %d\n",num,pos );
        } else {
            printf("Invalid Insertion \n");
        }
    }
}
```

4) How to delete a node in a single linked list?

```
void node_del (){

    int num=0;
    printf(" Enter the data to be deleted: ");
    scanf("%d",&num);

    Node *curr=head,*prev=NULL;
    if (curr == NULL) {
        printf("Node is empty !\n");
        return;
    }
```

```
    if(curr->data == num ) {
        head = curr->next;
        free(curr);
        return;
    }

    prev=head;
    curr=curr->next;
    while(curr != NULL) {
        if(curr->data == num){
            prev->next = curr->next;
            curr->next = NULL;
            printf("Node with value %d deleted! \n",curr->data);
            free(curr);
            return;
        } else {
            prev=curr;
            curr=curr->next;
        }
    }
    printf("Requested value %d not found ! \n",num);
}
```

5) Implement a stack using linked list?

```
#include <stdio.h>

#define MAX_CAPACITY 1000

typedef struct stack_t {
    int data;
    struct stack_t *next;
} Stack;

Stack *top;
int capacity = MAX_CAPACITY;
int current_capacity = 0;

void display() {

    Stack *temp=top;
    if (temp == NULL) {
        printf("Stack isnt initialized !\n");
        return;
    }
    printf(" Current Capacity is %d \n",current_capacity);
    while (temp!=NULL){
        if(temp == top ) {
            printf("%d <- TOP \n",temp->data);
        } else {
```

```
            printf("%d\n",temp->data);
        }
        temp=temp->next;
    }
}

void push(int num) {
    Stack *new_elem = (Stack*) malloc(sizeof(Stack));
    if(new_elem != NULL) {
        new_elem->data=num;
        new_elem->next=NULL;
    }
    if (current_capacity < MAX_CAPACITY) {
        if(top == NULL) {
            top = new_elem;
        } else {
            new_elem->next = top;
            top = new_elem;
        }
        current_capacity++;
    }
}

void pop() {
    Stack *temp=top;
    if(temp == NULL) {
        printf("Stack is Empty !\n");
        return;
    }
    top=temp->next;
    current_capacity--;
    printf("%d is removed from stack !\n",temp->data);
    free(temp);
}

void peek() {
    if(top == NULL) {
        printf("Stack is Empty !\n");
        return;
    }
    printf(" Top element is %d \n",top->data);
    printf(" Current Capacity is %d \n",current_capacity);
}

int main() {
    int choice=0,num=0;

    while (choice != 5) {
        printf("1. Display\n");
        printf("2. Push\n");
        printf("3. Pop\n");
```

```
        printf("4. Peek\n");
        printf("5. End\n\n\n");

        scanf("%d",&choice);

        switch(choice){
           case 1:
              display();
              break;
           case 2:
              printf("Enter the number: \n");
              scanf("%d",&num);
              push(num);
              break;
           case 3:
              pop();
              break;
           case 4:
              peek();
              break;
           case 5:
              printf("End Program ! \n");
              break;
           default:
              printf("Invalid Choice !\n");
        }
     }
}
```

6) Find nth node from end in the linked list?

Method 1: Brute Force

1) Find the len of the linked list, let say it's 10
2) then iterate again to find the 10-n position

Method 2: Use hash table

Position in the list	Address of the node
1	Address of node 1
2	Address of Node 2

Method 3: using 2 pointers,

1) temp pointer and tempNth pointer, both initialized with head.
2) Increment temp point till nth element, then start increment tempNth element
3) Once temp reaches end, element at tempNth element will be Nth element from the end.

7) How to find if there is a loop in the linked list?

Using 2 pointers with different speed.

```
int CheckLoop(Node *head) {
{
        Node *slowptr = head, *fastprt = head;
        while(slowptr && fastptr && fastpt->next) {
                slowptr=slowptr->next;
                fastptr=fastptr->next->next;
                if(slowptr==fastptr){
                        return 1; /* There is a loop */
                }
        }
        return 0; /* There is no loop */
}
```

8) How to find middle of a linked list?

```
void node_get_middle(){
   Node *slowptr = head, *fastptr = head;
        int i =0, count=1;
        if (fastptr == NULL) {
         printf("Empty List! \n");
          return ;
        }
        while(fastptr->next != NULL) {
          if(i==0) {
           fastptr=fastptr->next;
           i=1;
          } else if (i==1) {
           slowptr=slowptr->next;
          fastptr=fastptr->next;
          i=0;
         count++;
         }
       }
      printf("Middle of the list is %d at %d \n",slowptr->data,count);
}
```

9) Write a C program to return the nth node from the end of a linked list.

Here is a solution which is often called as the solution that uses frames. Suppose one needs to get to the 6th node from the end in this LL. First, just keep on incrementing the first pointer (ptr1) till the number of increments cross n (which is 6 in this case) `STEP 1 : 1(ptr1,ptr2) -> 2 -> 3 -> 4 -> 5 -> 6 -> 7 -> 8 -> 9 -> 10 STEP 2 : 1(ptr2) -> 2 -> 3 -> 4 -> 5 -> 6(ptr1) -> 7 -> 8 -> 9 -> 10` Now, start the second pointer (ptr2) and keep on incrementing it till the first pointer (ptr1) reaches the end of the LL. `STEP 3 : 1 -> 2 -> 3 -> 4(ptr2) -> 5 -> 6 -> 7 -> 8 -> 9 -> 10 (ptr1)` So here you have!, the 6th node from the end pointed to by ptr2!

```
Node * nthNode(Node *head, int n /*pass 0 for last node*/) {

        Node *ptr1,*ptr2;
        int count;
        if(!head) {
                return(NULL);
```

```
        }
        ptr1 = head;
        ptr2 = head;
        count = 0;
        while(count < n) {
                count++;
                if((ptr1=ptr1->next)==NULL) {
                //Length of the linked list less than n. Error.
                    return(NULL);
                }
        }
        while( (ptr1=ptr1->next)!=NULL) {
                ptr2=ptr2->next;
        }
        return(ptr2);

}
```

10) Remove Nth Node From End of List (LeetCode)

```
/**
 * Definition for singly-linked list.
 * struct ListNode {
 *     int val;
 *     struct ListNode *next;
 * };
 */


struct ListNode* removeNthFromEnd(struct ListNode* head, int n){

   int len = 0,i=0;
   struct ListNode *temp = head,*del_node=NULL;
   if(head == NULL)
      return NULL;

   while(temp != NULL){
      len++;
      temp=temp->next;
   }
   //printf("%d",len);
   temp = head;
   if(len-n == 0){
      /* delete the head node */
      del_node = head;
      head = head->next;
      free(del_node);
      return head;
   }
   while(temp != NULL) {
      if(len-i-1 == n) {
         /* delete the next node */
         del_node = temp->next;
         temp->next = temp->next->next;
         free(del_node);
```

```
            return head;
        }
        i++;
        temp=temp->next;
    }
    return NULL;
}
```

11) Reverse a single linked list?

```
void node_rev () {

    Node *curr = head, *new_head=NULL, *next_node=NULL;
    while(curr){
        next_node = curr->next;
        curr->next=new_head;
        new_head=curr;
        curr=next_node;
    }
    head=new_head;
}
```

12) Reverse a single linked list Recusively?

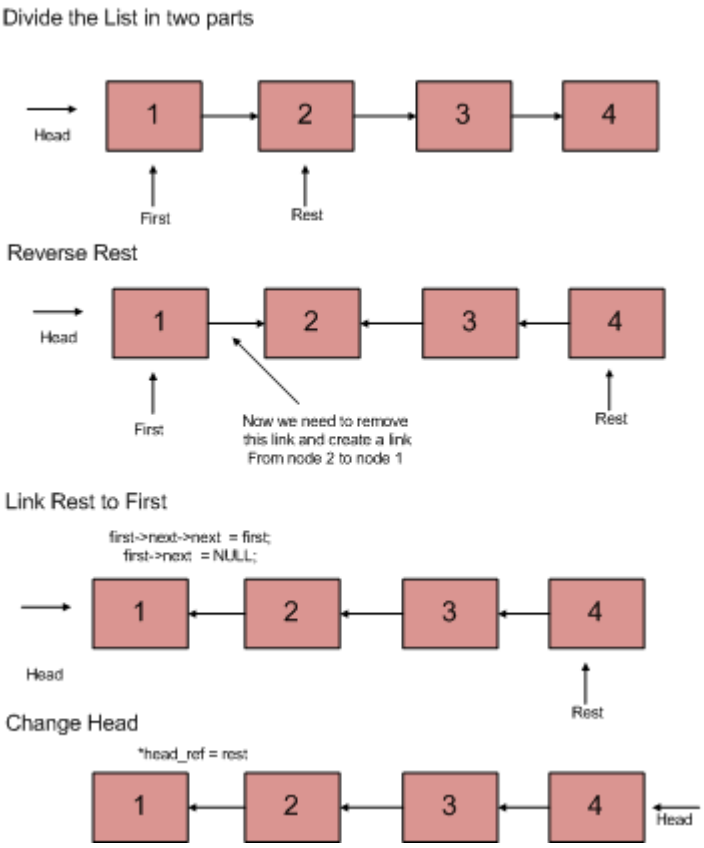
The general recursive algorithm for this is:

1) Divide the list in 2 parts - first node and rest of the list.
2) Recursively call reverse for the rest of the linked list.
3) Link rest to first.
4) Fix head pointer

Here is the code with inline comments:

```
struct node* recursiveReverseLL(struct node* first){

  if(first == NULL) return NULL; // list does not exist.

  if(first->link == NULL) return first; // list with only one node.

  struct node* rest = recursiveReverseLL(first->link); // recursive call on rest.

  first->link->link = first; // make first; link to the last node in the reversed rest.

  first->link = NULL; // since first is the new last, make its link NULL.

  return rest; // rest now points to the head of the reversed list.
}
```

I hope this picture will make things clearer:



(source: geeksforgeeks.org)

13) Display a linked list from the end?

Use Recursive Function:

```
void print_reverse(Node* node){
   /* base case */
   if(node == NULL )
      return;
   /* print the list after head node */
   print_reverse(node->next);
   /* Print  everything */
   printf("%d -> ",node->data);
}
```

14) How to sort linked list?

```
/* Split the nodes of the given list into front and back halves,
   and return the two lists using the reference parameters.
   If the length is odd, the extra node should go in the front list.
   Uses the fast/slow pointer strategy. */

void front_back_split(Node* source,
                Node** frontRef, Node** backRef)
{
   Node* fast;
```

```
    Node* slow;
    slow = source;
    fast = source->next;

    /* Advance 'fast' two nodes, and advance 'slow' one node */
    while (fast != NULL) {
        fast = fast->next;
        if (fast != NULL) {
            slow = slow->next;
            fast = fast->next;
        }
    }

    /* 'slow' is before the midpoint in the list, so split it in two
    at that point. */
    *frontRef = source;
    *backRef = slow->next;
    slow->next = NULL;
}

Node* sorted_merge(Node* a, Node* b)
{
    Node* result = NULL;

    /* Base cases */
    if (a == NULL)
        return (b);
    else if (b == NULL)
        return (a);

    /* Pick either a or b, and recur */
    if (a->data <= b->data) {
        result = a;
        result->next = sorted_merge(a->next, b);
    }
    else {
        result = b;
        result->next = sorted_merge(a, b->next);
    }
    return (result);
}

void merge_sort(Node *headref) {
    Node *curr = headref,*a=NULL,*b=NULL;

    /* Base case -- length 0 or 1 */
    if ((curr == NULL) || (curr->next == NULL)) {
        return;
    }

    /* Split head into 'a' and 'b' sublists */
```

```
   front_back_split(curr, &a, &b);

   /* Recursively sort the sublists */
   merge_sort(a);
   merge_sort(b);

   /* answer = merge the two sorted lists together */
   curr = sorted_merge(a, b);

   head = curr;
}

void node_sort(){
   printf("************************\n Sort \n************************\n");

   /* Merge sort is often preferred for sorting a linked list.
      The slow random-access performance of a linked list makes some other
      algorithms (such as quicksort) perform poorly, and others (such as
      heapsort) completely impossible.

   MergeSort(headRef)
   1) If the head is NULL or there is only one element in the Linked List
      then return.
   2) Else divide the linked list into two halves.
      FrontBackSplit(head, &a, &b); ( a and b are two halves )
   3) Sort the two halves a and b.
      MergeSort(a);
      MergeSort(b);
   4) Merge the sorted a and b (using SortedMerge() discussed here)
      and update the head pointer using headRef.
      *headRef = SortedMerge(a, b);
   */
   merge_sort(head);

   node_traverse();
}
```

15) Segregate Even and Odd numbers in a list?

Given a list :

```
Input  = {12, 34, 45, 9, 8, 90, 3}
Output = {12, 34, 8, 90, 45, 9, 3}
```

```
void node_seg_even_odd() {

    Node *left_head=NULL,*right_head=NULL,*temp=head,*left_ptr=NULL,*right_ptr=NULL;

    /* 1. increment Left pointer when data is odd,
       2. increment right pointer when data is even
       3. finally join right head at the end of left pointer.
    */
    if (temp == NULL) {
```

```
        printf("Empty List !\n");
        return;
    }

    while(temp) {
        if (temp->data%2) {
            /*odd*/
            if (left_head==NULL) {
                left_head=temp;
                left_ptr=temp;
                temp=temp->next;
                left_ptr->next=NULL;
            } else {
                left_ptr->next=temp;
                left_ptr=left_ptr->next;
                temp=temp->next;
                left_ptr->next=NULL;
            }
        } else {
            if (right_head==NULL) {
                right_head=temp;
                right_ptr=temp;
                temp=temp->next;
                right_ptr->next=NULL;
            } else {
                right_ptr->next=temp;
                right_ptr=right_ptr->next;
                temp=temp->next;
                right_ptr->next=NULL;
            }
        }
    }

    if(left_head == NULL && right_head != NULL) {
        head =right_head;
        return;
    }
    if(left_head != NULL && right_head == NULL) {
        head =left_head;
        return;
    }

    temp=left_head;
    while (temp->next != NULL){
            temp=temp->next;
    }
    temp->next=right_head;

    head = left_head;
}
```

16) How to remove duplicates from a sorted linked list ?

```
void node_remove_duplicate() {
        Node *current = head;
        if (current == NULL)
    return; // do nothing if the list is empty

  // Compare current node with next node
  while(current->next!=NULL) {
    if (current->data == current->next->data)
    {
      struct node* nextNext = current->next->next;
      free(current->next);
      current->next = nextNext;
    } else {
      current = current->next; // only advance if no deletion
```

```
        }
    }
}
```

17) Swap nodes of a linked list? (Leetcode)

```
/**
 * Definition for singly-linked list.
 * struct ListNode {
 *     int val;
 *     struct ListNode *next;
 * };
 */

struct ListNode* swapPairs(struct ListNode* head){

    if (head == NULL) return NULL;
    if (head->next == NULL) return head;

    struct ListNode* dummy = (struct ListNode*) malloc(sizeof(struct ListNode));
    dummy->next = head->next;

    struct ListNode* current = head, *first, *second, *temp = dummy;

    while (current != NULL && current->next != NULL ) {

        first = current;
        second = current->next;

        first->next = second->next;
        second->next = first;
        temp->next = second;

        current = first->next;
        temp = first;
    }

    return dummy->next;
}
```

18) Find the intersection of a Y-Shaped Linked List (LeetCode)?

```
/**
 * Definition for singly-linked list.
 * struct ListNode {
 *     int val;
 *     struct ListNode *next;
 * };
```

```
 */

struct ListNode *getIntersectionNode(struct ListNode *headA, struct ListNode *headB) {
   struct ListNode *temp1 = headA, *temp2 = headB;
   int i=0;
   while(temp1 != temp2)
   {

      temp1 = temp1 ? temp1->next : headB;

      temp2 = temp2 ? temp2->next : headA;

   }
   return temp1;

}
```

19) How to declare a structure of a double linked list?

```
typedef struct DLLNODE {
   int data;
   struct DLLNODE *next;
   struct DLLNODE *prev;
} DLLNode;
```

20) How to insert a node in a double linked list?

```
void DLLInsert(int data,int pos) {

   DLLNode *current = head, *new_node;
   int i=1;

   new_node = (DLLNode*) malloc(sizeof(DLLNode));
   if(!new_node) {
      printf("Allocation Error !");
      return;
   }
   new_node->data = data;

   if (pos == 1) {
      new_node->next = head;
      new_node->prev = NULL;
      if(head) {
         head->prev = new_node;
      }
      head = new_node;
      printf("%d inserted in %d!\n",data,pos);
      return;
   }
```

```
   while ( (i<pos-1) && (current->next != NULL)) {
      current = current->next;
      i++;
   }

   if(i != pos-1){
      printf("Invalid pos ! \n");
      free(new_node);
      return;
   }

   new_node->next=current->next;
   new_node->prev=current;
   if(current->next)
      current->next->prev = new_node;
   current->next = new_node;
   printf("%d inserted in %d!\n",data,pos);
   return;

}
```

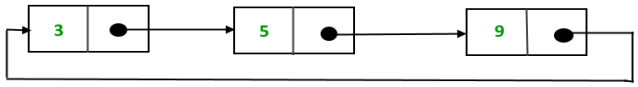
21) How to delete a node in a double linked list?

```
void DLLDelete(int pos) {
   DLLNode *current=head, *new_node;
   int i=1;
   if(head==NULL) {
      printf("List is empty \n");
      return;
   }
   if(pos == 1) {
      head=head->next;
      if(head!=NULL){
         head->prev == NULL;
      }
      free(current);
      return;
   }
   while(i<pos && (current!=NULL)) {
      i++;
      current=current->next;
   }
   if(i!=pos){
      printf("invalid pos !\n");
      return;
   }
   current->prev->next = current->next;
   if(current->next)
      current->next->prev= current->prev;
   printf("%d from %d position \n",current->data,pos);
```

```
    free(current);
    return;
}
```

22) What is a circle linked list?

Circular Linked List is a variation of Linked list in which the first element points to the last element and the last element points to the first element. Both Singly Linked List and Doubly Linked List can be made into a circular linked list.



23) How to count the nodes in a circle linked list?

```
void CLLDisplay(){
    CLLNode *current= head;
    int count=0;
    if (current==NULL) {
        printf("CLL list is empty\n\n");
        return;
    }
    do {
        printf("%d -> ",current->data);
        current=current->next;
        count++;
    }
    while(current != head);
    printf(" | \n");
    printf("%d",count);
    return;
}
```

24) Print the nodes in a circle linked list?

```
void CLLDisplay(){
    CLLNode *current= head;

    if (current==NULL) {
        printf("CLL list is empty\n\n");
        return;
    }
    do {
        printf("%d -> ",current->data);
        current=current->next;
    }
    while(current != head);
    printf(" | \n");
```

```
    return;
}
```

25) Insert a node at the end of a circle linked list?

```
void CLLInsert(int data,int pos) {

   CLLNode *current = head, *new_node;
   int i=1;

   new_node = (CLLNode*) malloc(sizeof(CLLNode));
   if(!new_node) {
      printf("Allocation Error !");
      return;
   }

   new_node->data = data;

   if (pos == 1) {
      new_node->next = current;
      if(current != NULL) {
         while(current->next != head) {
            current=current->next;
         }
         current->next=new_node;
      } else {
         new_node->next=new_node;
      }
      head = new_node;
      printf("%d inserted in %d!\n",data,pos);
      return;
   }

   while ( (i<pos-1) && (current->next != head)) {
      current = current->next;
      i++;
   }

   if(i != pos-1){
      printf("Invalid pos ! \n");
      free(new_node);
      return;
   }

   new_node->next=current->next;
   current->next = new_node;
   printf("%d inserted in %d!\n",data,pos);

   return;
```

```
}
```

26) Delete the first node in a circle linked list? (Needs Revisit)

```
void CLLDelete(int pos) {
   CLLNode *prev, *del_node=head;
   int i=1;
   if(head==NULL) {
      printf("List is empty \n");
      return;
   }

   if (pos == 1) {
      del_node = head;
      prev=head;
      while(prev->next != del_node) {
         prev=prev->next;
      }
      head = del_node->next;
      prev->next=head;
      free(del_node);
      del_node=NULL;
      return;
   }

   while( (i<pos-1) && (del_node->next != head)) {
      i++;
      prev=del_node;
      del_node=del_node->next;
   }

   if(i!=pos-1){
      printf("invalid pos !\n");
      return;
   }
   prev->next = del_node->next;

   printf("%d from %d position \n",del_node->data,pos);
   free(del_node);

   return;
}
```

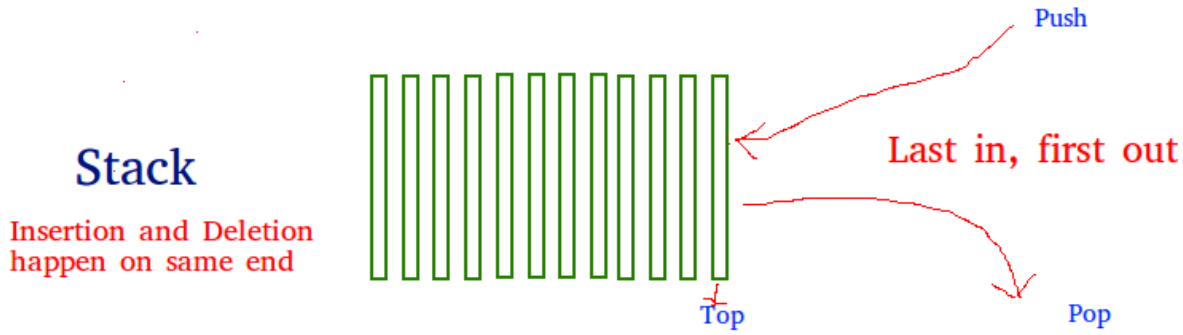
5. STACKS

1) What is a Stack ?

Stack is a linear data structure which follows a particular order in which the operations are performed. The order may be LIFO(Last In First Out) or FILO(First In Last Out).

Mainly the following three basic operations are performed in the stack:

- **Push:** Adds an item in the stack. If the stack is full, then it is said to be an Overflow condition.
- **Pop:** Removes an item from the stack. The items are popped in the reversed order in which they are pushed. If the stack is empty, then it is said to be an Underflow condition.
- **Peek or Top:** Returns top element of stack.
- **isEmpty:** Returns true if stack is empty, else false.



2) Application of Stacks?

- Balancing of symbols
- Infix to Postfix /Prefix conversion
- Redo-undo features at many places like editors, photoshop.
- Forward and backward feature in web browsers
- Used in many algorithms like Tower of Hanoi, tree traversals, stock span problem, histogram problem.
- Backtracking is one of the algorithms designing technique. Some examples of back tracking are Knight-Tour problem, N-Queen problem, find your way through maze and game like chess or checkers in all this problems we dive into someway if that way is not efficient we come back to the previous state and go into some another path. To get back from current state we need to store the previous state for that purpose we need stack.
- In Graph Algorithms like Topological Sorting and Strongly Connected Components
- In Memory management any modern computer uses stack as the primary-management for a running purpose. Each program that is running in a computer system has its own memory allocations.
- String reversal is also another application of stack. Here one by one each character get inserted into the stack. So the first character of string is on the bottom of the stack and the last element of string is on the top of stack. After Performing the pop operations on stack we get string in reverse order.

3) Write a structure to represent stack in c?

```
// A structure to represent a stack
struct Stack {
```

```
    int top;
    unsigned capacity;
    int* array;
};
```

4) Create a stack

```
// function to create a stack of given capacity. It initializes size of
// stack as 0
struct Stack* createStack(unsigned capacity)
{
    struct Stack* stack = (struct Stack*)malloc(sizeof(struct Stack));
    stack->capacity = capacity;
    stack->top = -1;
    stack->array = (int*)malloc(stack->capacity * sizeof(int));
    return stack;
}
```

5) Create a stack to evaluate value of a postfix expression.

```
// C program to evaluate value of a postfix expression
#include <stdio.h>
#include <string.h>
#include <ctype.h>
#include <stdlib.h>

// Stack type
struct Stack
{
    int top;
    unsigned capacity;
    int* array;
};

// Stack Operations
struct Stack* createStack( unsigned capacity )
{
    struct Stack* stack = (struct Stack*) malloc(sizeof(struct Stack));

    if (!stack) return NULL;

    stack->top = -1;
    stack->capacity = capacity;
    stack->array = (int*) malloc(stack->capacity * sizeof(int));

    if (!stack->array) return NULL;

    return stack;
}

int isEmpty(struct Stack* stack)
{
    return stack->top == -1 ;
}
```

```
char peek(struct Stack* stack)
{
    return stack->array[stack->top];
}

char pop(struct Stack* stack)
{
    if (!isEmpty(stack))
        return stack->array[stack->top--] ;
    return '$';
}

void push(struct Stack* stack, char op)
{
    stack->array[++stack->top] = op;
}


// The main function that returns value of a given postfix expression
int evaluatePostfix(char* exp)
{
    // Create a stack of capacity equal to expression size
    struct Stack* stack = createStack(strlen(exp));
    int i;

    // See if stack was created successfully
    if (!stack) return -1;

    // Scan all characters one by one
    for (i = 0; exp[i]; ++i)
    {
        // If the scanned character is an operand (number here),
        // push it to the stack.
        if (isdigit(exp[i]))
            push(stack, exp[i] - '0');

        // If the scanned character is an operator, pop two
        // elements from stack apply the operator
        else
        {
            int val1 = pop(stack);
            int val2 = pop(stack);
            switch (exp[i])
            {
            case '+': push(stack, val2 + val1); break;
            case '-': push(stack, val2 - val1); break;
            case '*': push(stack, val2 * val1); break;
            case '/': push(stack, val2/val1); break;
            }
        }
    }
    return pop(stack);
}

// Driver program to test above functions
int main()
{
    char exp[] = "231*+9-";
    printf ("postfix evaluation: %d", evaluatePostfix(exp));
```

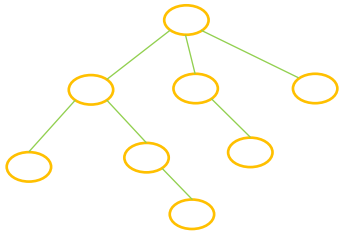
```
    return 0;

}
```

6. QUEUES

7. TREES

1) What are trees in data structure ?

Each nodes points to a number of nodes. Represents hierarchical nature of structure



2) What the properties of Trees ?

Root – The node at the top of the tree is called root. There is only one root per tree and one path from the root node to any node.

Parent – Any node except the root node has one edge upward to a node called parent.

Child – The node below a given node connected by its edge downward is called its child node.

Leaf – The node which does not have any child node is called the leaf node.

Subtree – Subtree represents the descendants of a node.

Path – Path refers to the sequence of nodes along the edges of a tree.

Visiting – Visiting refers to checking the value of a node when control is on the node.

Traversing – Traversing means passing through nodes in a specific order.

Levels – Level of a node represents the generation of a node. If the root node is at level 0, then its next child node is at level 1, its grandchild is at level 2, and so on.

Keys – Key represents a value of a node based on which a search operation is to be carried out for a node.

Height of a Node – the number of edges from the node to the deepest leaf

Depth of a Node – the number of edges from the root to the node.

Degree of a Node – the total number of branches of that node.

Height of a Tree - height of the root node or the depth of the deepest node.

3) What are the types of trees ?

General Tree

Binary Tree

Binary Search Tree

AVL Tree

Red-Black Tree

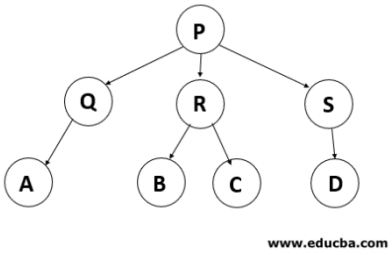
B-tree

4) General Tree

If no constraint is placed on the tree's hierarchy, a tree is called a general tree. Every node may have infinite numbers of children in General Tree. The tree is the super-set of all other trees.

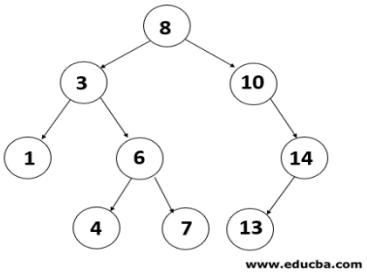
5) Binary Tree

The binary tree is the kind of tree in which most two children can be found for each parent. The kids are known as the left kid and right kid. This is more popular than most other trees. When certain constraints and characteristics are applied in a Binary tree, a number of others such as AVL tree, BST (Binary Search Tree), RBT tree, etc. are also used. When we move forward, we will explain all these styles in deta



6) Binary Search Tree

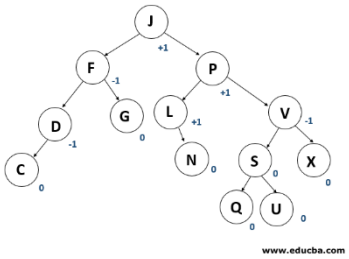
Binary Search Tree (BST) is a binary tree extension with several optional restrictions. The left child value of a node should in BST be less than or equal to the parent value, and the right child value should always be greater than or equal to the parent's value. This Binary Search Tree property makes it ideal for search operations since we can accurately determine at each node whether the value is in the left or right sub-tree. This is why the Search Tree is named.



7) AVL Tree

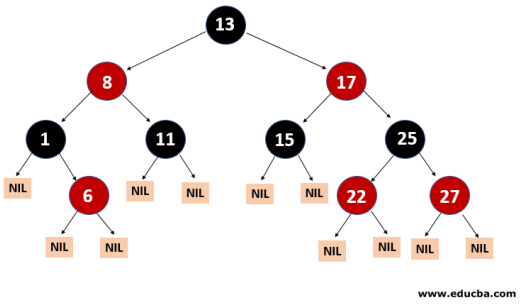
AVL tree is a binary search tree self-balancing. On behalf of the inventors Adelson-Velshi and Landis, the name AVL is given. This was the first tree that balanced dynamically. A balancing factor is allocated for each node in the AVL tree, based on whether the tree is

balanced or not. The height of the node kids is at most 1. AVL vine. In the AVL tree, the correct balance factor is 1, 0 and -1. If the tree has a new node, it will be rotated to ensure that it is balanced. It will then be rotated. Common operations such as viewing, insertion, and removal take O(log n) time in the AVL tree. It is mostly applied when working with Lookups operations.



8) Red-Black Tree

Another kind of auto-balancing tree is red-black. According to the red-black tree's properties, the red-black name is given because the Red-black tree has either red or Black painted on each node. It maintains the balance of the forest. Even though this tree is not fully balanced, the searching operation only takes O (log n) time. When the new nodes are added in Red-Black Tree, nodes will be rotated to maintain the Red-Black Tree's properties.



9) N-ary Tree

The maximum number of children in this type of tree with a node is N. A binary tree is a two-year tree, as at most 2 children in every binary tree node. A complete N-ary tree is a tree where kids of a node either are 0 or N.

10) Advantages of Tree

- The tree reflects the data structural connections.
- The tree is used for hierarchy.
- It offers an efficient search and insertion procedure.
- The trees are flexible. This allows subtrees to be relocated with minimal effort.

11) Operations on a tree

Basic Operations :

- Inserting an element on a tree
- Deleting an element on a tree
- Searching an element on a tree
- Traversing an element on a tree

Auxiliary Operations:

- Finding the size of a tree
- Finding the height of a tree
- Finding the level which has maximum sum
- Finding the least common ancestor LCA

12) Applications of tree data structure

Unlike Array and Linked List, which are linear data structures, tree is hierarchical (or non-linear) data structure.

1. One reason to use trees might be because you want to store information that naturally forms a hierarchy. For example, the file system on a computer:

   file system

   ```
        /   <-- root
     /      \
   ...        home
            /      \
         ugrad      course
          /        /   |   \
        ...      cs101 cs112 cs113
   ```

2. If we organize keys in form of a tree (with some ordering e.g., BST), we can search for a given key in moderate time (quicker than Linked List and slower than arrays). Self-balancing search trees like AVL and Red-Black trees guarantee an upper bound of O(Logn) for search.
3. We can insert/delete keys in moderate time (quicker than Arrays and slower than Unordered Linked Lists). Self-balancing search trees like AVL and Red-Black trees guarantee an upper bound of O(Logn) for insertion/deletion.
4. Like Linked Lists and unlike Arrays, Pointer implementation of trees don't have an upper limit on number of nodes as nodes are linked using pointers.

Other Applications :

1. Store hierarchical data, like folder structure, organization structure, XML/HTML data.
2. Binary Search Tree is a tree that allows fast search, insert, delete on a sorted data. It also allows finding closest item
3. Heap is a tree data structure which is implemented using arrays and used to implement priority queues.
4. B-Tree and B+ Tree : They are used to implement indexing in databases.
5. Syntax Tree: Used in Compilers.
6. K-D Tree: A space partitioning tree used to organize points in K dimensional space.
7. Trie : Used to implement dictionaries with prefix lookup.
8. Suffix Tree : For quick pattern searching in a fixed text.
9. Spanning Trees and shortest path trees are used in routers and bridges respectively in computer networks
10. As a workflow for compositing digital images for visual effects.

13) Structure of Binary tree

```
struct BinaryTreeNode {
   int data;
   struct BinaryTreeNode *Right;
   struct BinaryTreeNode *Left;
};
```

14) Binary Tree Traversal

There are three types of traversals:

- Preorder Traversal
- Postorder Traversal
- Inorder Traversal

Additionally, We have Level Order traversal as well, inspired by BFS in graph

15) Pre-order Traversal

```
void PreOrder( struct BinaryTreeNode *root) { //DLR
   if(root) {
      printf("%d\n",root->data);
      PreOrder(root->left);
      PreOrder(root->right);
   }
}
```

16) In-order Traversal

```
void InOrder( struct BinaryTreeNode *root) { //LDR
   if(root) {
      InOrder(root->left);
      printf("%d\n",root->data);
      InOrder(root->right);
   }
}
```

17) Post-order Traversal

```
void PostOrder( struct BinaryTreeNode *root) { //LRD
   if(root) {
      PostOrder(root->left);
      PostOrder(root->right);
      printf("%d\n",root->data);
   }
}
```

18) Level Order Traversal

19) Write a C program to find the depth or height of a tree.
20) Write a C program to determine the number of elements (or size) in a tree.
21) Write a C program to delete a tree (i.e, free up its nodes)
22) Write C code to determine if two trees are identical

23) Write a C program to find the minimum value in a binary search tree.
24) Write a C program to compute the maximum depth in a tree?
25) Write a C program to create a copy of a tree
26) Write a C program to create a mirror copy of a tree (left nodes become right and right nodes become left)! (Oracle)
27) Write C code to return a pointer to the nth node of an in-order traversal of a BST.

28) Write a C program to delete a node from a Binary Search Tree?
29) Write C code to search for a value in a binary search tree (BST).
30) Write C code to count the number of leaves in a tree
31) Find the closest ancestor of two nodes in a tree. (Cisco)
32) How do you convert a tree into an array?
33) How many different trees can be constructed using n nodes?
34) Implement Breadth First Search (BFS) and Depth First Search (DFS)
35) What is a threaded binary tree?

8. GRAPHS

9. HASHING

10. SORTING

1) **What is heap sort?**
2) **What is the difference between Merge Sort and Quick sort?**
3) **Give pseudocode for the mergesort algorithm**
4) **Implement the bubble sort algorithm. How can it be improved? Write the code for selection sort, quick sort, insertion sort.**
5) **How can I sort things that are too large to bring into memory?** New!

11. STRING

1) **Reverse a string?**

2) **Reverse the words in a sentence in place.**

3) **Write a simple piece of code to split a string at equal intervals**

4) **Given two strings A and B, how would you find out if the characters in B were a subset of the characters in A?**

5) **How does the function strtok() work?**

6) **How does the function strdup(), strncat(), strcmp(), strncmp(), strcpy(), strncpy(), strlen(), strchr(), strchr(), strpbrk(), strspn(), strcspn() work?**

7) **Can you compare two strings like string1==string2? Why do we need strcmp()?**

12. BIT FIDDLING

1) **How to set a particular bit in C?**

Number | = (1<< nth Position)

2) **How to clear a particular bit in C?**

Number &= ~ (1<< nth Position)

3) **How to check if a particular bit is set in C?**

Bit = Number & (1 << nth)

4) **How to toggle a particular bit in C?**

Number ^= (1<< nth Position)

5) **Write an Efficient C Program to Reverse Bits of a Number?**

```
#define CHAR_BITS 8 // size of character
#define INT_BITS ( sizeof(int) * CHAR_BITS)
//bit reversal function
  unsigned int ReverseTheBits(unsigned int num){
  unsigned int iLoop = 0;
  unsigned int tmp = 0; // Assign num to the tmp
  int iNumberLopp = INT_BITS;
  for(; iLoop < iNumberLopp; ++iLoop) {
     if((num & (1 << iLoop))) // check set bits of num {
```

```
        tmp |= 1 << ((INT_BITS - 1) - iLoop); //putting the set bits of num in tmp
      }
    }
    return tmp;
}
```

6) How to swap the two nibbles in a byte ?

```
1) Move all bits of the first set to the rightmost side
   set1 =  (x >> p1) & ((1U << n) - 1)
Here the expression (1U << n) - 1 gives a number that
contains last n bits set and other bits as 0. We do &
with this expression so that bits other than the last
n bits become 0.
2) Move all bits of second set to rightmost side
   set2 =  (x >> p2) & ((1U << n) - 1)
3) XOR the two sets of bits
   xor = (set1 ^ set2)
4) Put the xor bits back to their original positions.
   xor = (xor << p1) | (xor << p2)
5) Finally, XOR the xor with original number so
   that the two sets are swapped.
   result = x ^ xor
```

```
/ C Program to swap bits
// in a given number
#include <stdio.h>

int swapBits(unsigned int x, unsigned int p1, unsigned int p2, unsigned int
n)
{
    /* Move all bits of first set to rightmost side */
    unsigned int set1 = (x >> p1) & ((1U << n) - 1);

    /* Move all bits of second set to rightmost side */
    unsigned int set2 = (x >> p2) & ((1U << n) - 1);

    /* XOR the two sets */
    unsigned int xor = (set1 ^ set2);

    /* Put the xor bits back to their original positions */
    xor = (xor << p1) | (xor << p2);

    /* XOR the 'xor' with the original number so that the
       two sets are swapped */
    unsigned int result = x ^ xor;

    return result;
}

/* Driver program to test above function*/
int main()
```

```
{
    int res = swapBits(28, 0, 3, 2);
    printf("\nResult = %d ", res);
    return 0;
}
```

7) Write a program to get the higher and lower nibble of a byte without using shift operator?

```
#include<stdio.h>
struct full_byte
{
char first : 4;
char second : 4;
};

union A
{
char x;
struct full_byte by;
};

main()
{
char c = 100;
union A a;
a.x = c;
printf("the two nibbles are: %d and %d\n", a.by.first, a.by.second);
}
```

8) How to reverse the odd bits of an integer?

13. OS CONCEPTS

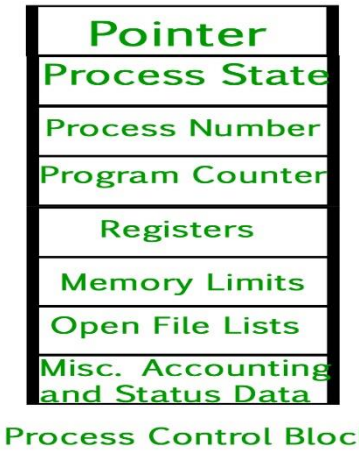
1) What is a process?

An executing program is known as process. There are two types of processes:

- Operating System Processes
- User Processes

2) What is a process table?

A process control block (PCB) contains information about the process, i.e. registers, quantum, priority, etc. The process table is an array of PCB's, that means logically contains a PCB for all of the current processes in the system.



Process Control Block

- **Pointer –** It is a stack pointer which is required to be saved when the process is switched from one state to another to retain the current position of the process.
- **Process state –** It stores the respective state of the process.
- **Process number –** Every process is assigned with a unique id known as process ID or PID which stores the process identifier.
- **Program counter –** It stores the counter which contains the address of the next instruction that is to be executed for the process.
- **Register –** These are the CPU registers which includes: accumulator, base, registers and general purpose registers.
- **Memory limits –** This field contains the information about memory management system used by operating system. This may include the page tables, segment tables etc.
- **Open files list –** This information includes the list of files opened for a process.

3) What are different states of process?

- New Process
- Running Process
- Waiting Process
- Ready Process
- Terminated Process

4) What are the benefits of a multiprocessor system?

A Multiprocessor system is a type of system that includes two or more CPUs. It involves the processing of different computer programs at the same time mostly by a computer system with two or more CPUs that are sharing single memory.

Benefits:

- Such systems are used widely nowadays to improve performance in systems that are running multiple programs concurrently.
- By increasing the number of processors, a greater number of tasks can be completed in unit time.
- One also gets a considerable increase in throughput and is cost-effective also as all processors share the same resources.
- It simply improves the reliability of the computer system.

5) What is the concept of reentrancy?

It is a very useful memory saving technique that is used for multi-programmed time sharing systems. It provides functionality that multiple users can share a single copy of program during the same period.

It has two key aspects:

- The program code cannot modify itself.
- The local data for each user process must be stored separately.

6) What is the difference between process and program?

A program while running or executing is known as a process.

7) What is a Thread?

A thread is a basic unit of CPU utilization. It consists of a thread ID, program counter, register set and a stack.

8) What are the advantages of multithreaded programming?

A list of advantages of multithreaded programming:

- Enhance the responsiveness to the users.
- Resource sharing within the process.
- Economical
- Completely utilize the multiprocessing architecture.

9) Process vs Thread

- Process means a program is in execution, whereas thread means a segment of a process.
- A Process is not Lightweight, whereas Threads are Lightweight.
- A Process takes more time to terminate, and the thread takes less time to terminate.
- Process takes more time for creation, whereas Thread takes less time for creation.
- Process likely takes more time for context switching whereas as Threads takes less time for context switching.
- A Process is mostly isolated, whereas Threads share memory.
- Process does not share data, and Threads share data with each other.

10) Kernel Thread vs User Space Thread

User level thread	Kernel level thread
• User thread are implemented by users.	• Kernel threads are implemented by OS.
• OS doesn't recognize user level threads.	• Kernel threads are recognized by OS.
• Implementation of User threads is easy.	• Implementation of Kernel thread is complicated.
• Context switch time is less.	• Context switch time is more.
• Context switch requires no hardware support.	• Hardware support is needed.
• If one user level thread perform blocking operation then entire process will be blocked.	• If one kernel thread perform blocking operation then another thread can continue execution.
• User level threads are designed as dependent threads.	• Kernel level threads are designed as independent threads.

11) What is semaphore?

Semaphore is a protected variable or abstract data type that is used to lock the resource being used. The value of the semaphore indicates the status of a common resource.

There are two types of semaphore:

- Binary semaphores
- Counting semaphores

12) What is a Binary Semaphore?

Binary semaphore takes only 0 and 1 as value and used to implement mutual exclusion and synchronize concurrent processes.

13) Mutex vs Semaphore: What's the Difference?

BASIS OF COMPARISON	SEMAPHORE	MUTEX
Description	Semaphore is a signaling mechanism and a thread waiting on a semaphore can be signaled by another thread.	The Mutex is a locking mechanism that makes sure only one thread can acquire the mutex at a time and enter the critical section.
Purpose	Semaphore is for processes also.	Mutex is for thread.

Nature	Semaphore is atomic but not singular in nature.	Mutex is typically atomic and singular in nature.
Use	A binary semaphore can be used as a mutex along with providing feature of signaling amongst threads.	A mutex can never be used as a semaphore.
Release Of The Mechanism	Semaphore value can be changed by any process acquiring or releasing the resource.	Mutex object lock is released only by the process that has acquired the lock on it.
What They Are	Semaphore is an integer variable.	Mutex is an Object.
Unlocking	If locked, a semaphore can be acted upon by different threads.	Mutex if locked has to be unlocked by the same thread.
Thread & Process	Only one process can acquire binary semaphore at a time, but multiple processes can simultaneously acquire semaphore in case of counting semaphore.	Only one thread can acquire a mutex at a time.
Working	Semaphore works in kernel space.	Mutex works in user space.
Ownership Concept	The concept of ownership is absent in semaphore.	The thread with mutex has ownership over the resource.
Categorization	Semaphore can be categorized into counting semaphore and binary semaphore.	Mutex does not have further categorization.
The Process Requesting for Resources	If all resources are being used, the process requesting for resource performs wait () operation and block itself till semaphore count become greater than one.	If a mutex object is already locked, the process requesting for resources waits and queued by the system till lock is released.

14) What are monitors? How are they different from semaphores?*

15) Multi-process vs multithread ?

Parameter	Multiprocessing	Multithreading
Basic	Multiprocessing helps you to increase computing power.	Multithreading helps you to create computing threads of a single process to increase computing power.
Execution	It allows you to execute multiple processes concurrently.	Multiple threads of a single process are executed concurrently.
CPU switching	In Multiprocessing, CPU has to switch between multiple programs so that it looks like that multiple programs are running simultaneously.	In multithreading, CPU has to switch between multiple threads to make it appear that all threads are running simultaneously.

Parameter	Multiprocessing	Multithreading
Creation	The creation of a process is slow and resource-specific.	The creation of a thread is economical in time and resource.
Classification	Multiprocessing can be symmetric or asymmetric.	Multithreading is not classified.
Memory	Multiprocessing allocates separate memory and resources for each process or program.	Multithreading threads belonging to the same process share the same memory and resources as that of the process.
Pickling objects	Multithreading avoids pickling.	Multiprocessing relies on pickling objects in memory to send to other processes.
Program	Multiprocessing system allows executing multiple programs and tasks.	Multithreading system executes multiple threads of the same or different processes.
Time taken	Less time is taken for job processing.	A moderate amount of time is taken for job processing.

16) What is IPC?

IPC stands for Inter-Process Communication, and it is a mechanism, in which various processes can communicate with each other with the approval of the OS.

17) Name the Various IPC mechanisms.

- Sockets
- Pipe
- Shared Memory
- Signals
- Message Queues

18) What are sockets?

Sockets are the Inter-process Communication mechanisms that are used to provide point-to-point communication between 2 processes.

Sockets are often utilized in client-server applications because many protocols, such as FTP, SMTP, and POP3 use sockets to implement the connection between server and client.

19) How OS boots, list the operations

1. Once the computer system is turned on, **BIOS** (Basic Input /Output System) performs a series of activities or functionality test on programs stored in ROM, called on **Power-**

on Self Test (POST) that checks to see whether peripherals in system are in perfect order or not.

2. After the **BIOS** is done with pre-boot activities or functionality test, it read bootable sequence from **CMOS** (Common Metal Oxide Semiconductor) and looks for master boot record in first physical sector of the bootable disk as per boot device sequence specified in **CMOS**. For example, if the boot device sequence is –
    1. Floppy Disk
    2. Hard Disk
    3. CDROM
3. After this, master boot record will be searched first in a floppy disk drive. If not found, then hard disk drive will be searched for master boot record. But if the master boot record is not even present on hard disk, then CDROM drive will be searched. If the system is not able to read master boot record from any of these sources, ROM displays the message "**No Boot device found"** and system is halted. On finding master boot record from a particular bootable disk drive, operating system loader, also called Bootstrap loader is loaded from boot sector of that bootable drive· into memory. A bootstrap loader is a special program that is present in boot sector of bootable drive.
4. Bootstrap loader first loads the **IO.SYS** file. After this, **MSDOS.SYS** file is loaded which is core file of DOS operating system.
5. After this, **MSDOS.SYS** file searches to find Command Interpreter in **CONFIG.SYS** file and when it finds, it loads into memory. If no Command Interpreter specified in the **CONFIG.SYS** file, the **COMMAND.COM** file is loaded as default Command Interpreter of DOS operating system.
6. The last file is to be loaded and executed is the **AUTOEXEC.BAT** file that contains a sequence of DOS commands. After this, the prompt is displayed, and we can see drive letter of bootable drive displayed on the computer system, which indicates that operating system has been successfully on the system from that drive.

    Refer : http://www.c-jump.com/CIS24/Slides/Booting/Booting.html

20) Types of Booting

1. **Cold Booting/Switch Booting –** When the user starts computer by pressing power switch on system unit, the operating system is loaded from disk to main memory this type of booting is called **Cold Booting.** This booting takes more time than Hot or Warm Booting.
2. **Hot or Warm Booting –** Hot booting is done when computer system comes to no response state/hang state. Computer does not respond to commands supplied by user. There are many reasons for this state, only solution is to reboot computer by using the **Reset button** on cabinet or by pressing a combination of **ALT + CTRL + DEL** keys from keyboard.

21) What is a socket?

A socket is used to make connection between two applications. Endpoints of the connection are called socket.

22) What is a real-time system?

Real-time system is used in the case when rigid-time requirements have been placed on the operation of a processor. It contains a well defined and fixed time constraints.

23) What is the use of paging in operating system?

Paging is used to solve the external fragmentation problem in operating system. This technique ensures that the data you need is available as quickly as possible.

24) What is the concept of demand paging?

Demand paging specifies that if an area of memory is not currently being used, it is swapped to disk to make room for an application's need.

25) What is virtual memory?

Virtual memory is a very useful memory management technique which enables processes to execute outside of memory. This technique is especially used when an executing program cannot fit in the physical memory.

26) What is thrashing?

Thrashing is a phenomenon in virtual memory scheme when the processor spends most of its time in swapping pages, rather than executing instructions.

27) When does trashing occur?

Thrashing specifies an instance of high paging activity. This happens when it is spending more time paging instead of executing.

28) What is deadlock? Explain.

Deadlock is a specific situation or condition where two processes are waiting for each other to complete so that they can start. But this situation causes hang for both of them.

29) What are the four necessary and sufficient conditions behind the deadlock?

These are the 4 conditions:

1) **Mutual Exclusion Condition**: It specifies that the resources involved are non-sharable.

2) **Hold and Wait Condition**: It specifies that there must be a process that is holding a resource already allocated to it while waiting for additional resource that are currently being held by other processes.

3) **No-Preemptive Condition**: Resources cannot be taken away while they are being used by processes.

4) **Circular Wait Condition**: It is an explanation of the second condition. It specifies that the processes in the system form a circular list or a chain where each process in the chain is waiting for a resource held by next process in the chain.

30) What is Banker's algorithm?

Banker's algorithm is used to avoid deadlock. It is the one of deadlock-avoidance method. It is named as Banker's algorithm on the banking system where bank never allocates available cash in such a manner that it can no longer satisfy the requirements of all its customers.

31) What is fragmentation?

Fragmentation is a phenomenon of memory wastage. It reduces the capacity and performance because space is used inefficiently.

32) What is spooling?

Spooling is a process in which data is temporarily gathered to be used and executed by a device, program or the system. It is associated with printing. When different applications send output to the printer at the same time, spooling keeps these all jobs into a disk file and queues them accordingly to the printer.

33) What is Belady's Anomaly?

Belady's Anomaly is also called FIFO anomaly. Usually, on increasing the number of frames allocated to a process virtual memory, the process execution is faster, because fewer page faults occur. Sometimes, the reverse happens, i.e., the execution time increases even when more frames are allocated to the process. This is Belady's Anomaly. This is true for certain page reference patterns.

34) What are the different scheduling algorithms

First-Come, First-Served (FCFS) Scheduling.
Shortest-Job-Next (SJN) Scheduling.
Priority Scheduling.
Shortest Remaining Time.
Round Robin(RR) Scheduling.
Multiple-Level Queues Scheduling.

35) What are zombie processes?

Formally, these processes are known as **defunct** processes. If a child process is still in the process table even after the parent process has been executed, this scenario could cause a zombie process.

Even the kill commands do not have any effect on these processes. If the **Wait System Call** read the exit status of the process, then the zombie process would remove from the process table.

36) What is the reader-writer lock?

Answer: Reader-writer lock is used to prevent data integrity. This lock allows concurrent access to read operation, which means multiple threads can read data simultaneously.

But it does not allow concurrent write, and if one thread wants to modify data via writing, then all the other threads will be blocked from reading or writing data.

37) How do you find out if a machine is 32 bit or 64 bit?

One common answer is that all compilers keep the size of integer the same as the size of the register on a perticular architecture. Thus, to know whether the machine is 32 bit or 64 bit, just see the size of integer on it.

38) What do the system calls fork(), vfork(), exec(), wait(), waitpid() do?
39) Whats the difference between fork() and vfork()?
40) Whats the difference between fork() and exec()?
41) How does freopen() work? *
42) How are signals handled? *
43) What is meant by context switching in an OS?*
44) What are short-, long- and medium-term scheduling?
45) What is the Translation Lookaside Buffer (TLB)?
46) What is cycle stealing?
47) What is a reentrant program?
48) When is a system in safe state?
49) What is busy waiting?
50) What is pages replacement? What are local and global page replacements?*
51) What is meant by latency, transfer and seek time with respect to disk I/O?
52) In the context of memory management, what are placement and replacement algorithms?
53) What is paging? What are demand- and pre-paging?*
54) What is mounting?
55) What do you mean by dispatch latency?
56) What is multi-processing? What is multi-tasking? What is multi-threading? What is multi-programming?*
57) What is compaction?
58) What is memory-mapped I/O? How is it different frim I/O mapped I/O?
59) List out some reasons for process termination.

14. THREADS

1) Suppose you have thread T1, T2, and T3. How will you ensure that thread T2 will run after T1 and thread T3 after T2?

2) What is a Thread?

A Thread is a concurrent unit of execution. We can say that it is part of the process which can easily run concurrently with other parts of the process.

3) What is multithreading?

Multithreading is a process of executing more than one thread simultaneously. The main advantage is:

- Threads share the same address space
- Thread remains lightweight
- Cost of communication between threads is low.

4) What are the states associated with the thread?

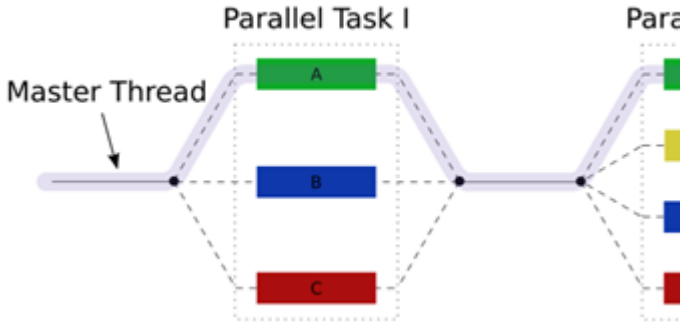
- Ready
- Running
- Waiting
- Dead state

5) What are the thread states?

Following are the different thread states:

New: A thread which is just instantiated is in the new state. When a start() method is invoked, the thread becomes the ready state. Then it is moved to the runnable state by the thread scheduler.

- **Runnable:** A thread which is ready to run
- **Running:** A thread which is executing is in running state.
- **Blocked:** A blocked thread is waiting for a monitor lock is in this state. This thing can also happen when a thread performs an I/O operation and moves to the next state.
- **Waiting:** It is a thread that is waiting for another thread to do the specific action.
- **Timed_waiting:** It is a thread that is waiting for another thread to perform.
- **Terminated:** A thread that has exited is in this state.



6) What are the major differences between Thread and Process?

The thread is a subset of process. The process can contain multiple threads. The process can run on on different memory space, but all threads share the same memory space.

7) What is deadlock?

Deadlock is a situation when a thread is waiting for an object lock, that is acquired by another thread and second thread also waiting for an object lock that is acquired by the first thread. As both threads are waiting for each other to release this condition is called deadlock.

8) Explain the differences between User-level and Kernel level thread?

• User-level threads are faster than kernel-level threads from the creation and managing perspective.

• User-level threads are generic, whereas the kernel-level threads are more specific to the concerned operating system.

• In the case of the user level, the multithreading process can't be implemented on multiprocessing, whereas kernel level can themselves be multithreaded.

9) What is a race condition? How will you find and solve race condition?

10) What do you mean by thread starvation?

In the situation when a thread does not have sufficient CPU for its execution Thread starvation happens.

However, it may happen in the following scenarios

- Low priority threads will get less CPU compared to high priority threads. Lower priority thread can starve away waiting to get more CPU space to perform calculations.
- The thread may be waiting indefinitely for a lock on object's monitor but notify() may repeatedly be awakening some other threads. In that case, also thread starves away.

10) What is the meaning of busy spin in multi-threading?

- Busy spin is a technique that concurrent programmers employ to make a thread wait on certain condition. This is quite different from traditional methods like wait() and sleep()which all involves relinquishing CPU control. This method does not require abandon CPU, instead it the just runs the empty loop.

11) What is context-switching in multi-threading?

- It is the process of storing and restoring of CPU state. This helps to resume thread execution from the same point at a later point in time. It is one of the essential features for multitasking operating system and support for the multi-threaded environment.

12) What are some common problems you have faced in multi-threading environment? How did you resolve it?

15. FILE OPERATIONS

1) How do stat(), fstat(), vstat() work? How to check whether a file exists?
2) How can I insert or delete a line (or record) in the middle of a file?
3) How can I recover the file name using its file descriptor?
4) How can I delete a file? How do I copy files? How can I read a directory in a C program?
5) How does one use fread() and fwrite()? Can we read/write structures to/from files?
6) Whats the use of fopen()?
7) fclose() ?
8) fprintf()?
9) getc() ?
10) putc() ?
11) getw() ?
12) putw() ?
13) fscanf()?
14) feof() ?
15) ftell() ?
16) fseek() ?
17) rewind() ?
18) fread() ?
19) fwrite() ?
20) fgets() ?
21) fputs() ?
22) freopen() ?
23) fflush() ?
24) ungetc()?
25) How to check if a file is a binary file or an ascii file? New!

16. Compiling and Linking

1) **How to list all the predefined identifiers?**
2) **How the compiler make difference between C and C++?**
3) **What are the general steps in compilation?**
4) **What are the different types of linkages?**
5) **What do you mean by scope and duration?**
6) **What are makefiles? Why are they used?***

17. NETWORKING

1) Write a program to convert an IP address to int?

2) There is a packet with Multiple lengths of 4 bytes in it. [Len] Data[Len2] Data[Len3]Data.

Write a code where function takes the *pkt and returns same pkt where Len is total length and data is all together. Length header is 4 Bytes

Eg:

*pkt = 10 [1-10] 20[0-20] 30[0-30],

Out *pkt = 60[0-60] packet.

18. GENERIC PROGRAMS

1) **Write a code to print the number that appears only once in the array?**

Input Array {12,1,12,3,12,1,1,1,2,3,3}

2) **Print items in the list that has #odd w/o any additional memory space {3,3,4,3,1,1} ==> 3,3,4,3**

3) **Remove multiple spaces and tabs from the string and count the number of them**

4) **Write C code to implement the Binary Search algorithm.**

5) **Two sum: find the number of pairs with sum=K, do not use the same numbers/pairs twice**

6) Code for Tower of Hanoi (Juniper)

Algorithm:

- Move the top of n-1 disks from Source to Auxiliary Tower
- Move the nth disk from Source to Destination tower
- Move the n-1 disks from Auxiliary tower to Destination tower
- Transferring n-1 disks is taken as a new problem and can be solved recursively.

7) Find the first repeating letter in the string (Citrix)

8) How do you calculate the maximum subarray of a list of numbers? (Juniper)

9) Solve the Rat In A Maze problem using backtracking.

10) What is the 8 queens problem? Write a C program to solve it.

11) Write a program to merge two sorted linked list in sorted order, so that if an integer is in both the arrays, it gets added into the final array only once. (VMware)

12) Write a program to merge two unsorted linked list in sorted order, so that if an integer is in both the arrays, it gets added into the final array only once. (Extreme Networks)

13) Build Lowest Number by Removing n digits from a given number (Citrix)

Given string num representing a non-negative integer `num`, and an integer `k`, return *the smallest possible integer after removing* `k` *digits from* `num`.

Example 1:

```
Input: num = "1432219", k = 3
Output: "1219"
Explanation: Remove the three digits 4, 3, and 2 to form the new number 1219
which is the smallest.
```

Example 2:

```
Input: num = "10200", k = 1
Output: "200"
Explanation: Remove the leading 1 and the number is 200. Note that the output
must not contain leading zeroes.
```

Example 3:

```
Input: num = "10", k = 2
Output: "0"
Explanation: Remove all the digits from the number and it is left with nothing
which is 0.
```

Constraints:

- `1 <= k <= num.length <= 10^5`
- `num` consists of only digits.
- `num` does not have any leading zeros except for the zero itself.

14) K largest(smallest) number in an array – (use max/min heap) (Oracle)
15) Find the two non-repeating elements in an array of repeating elements (Juniper)
16) Longest Palindromic Substring (Samsung)
17) Smallest subarray with sum greater than a given value (Juniper)
18) Reverse words in a string (Capgemini)
19) Write your own STRSTR function

2

20) How to generate prime numbers? How to generate the next prime after a given prime?