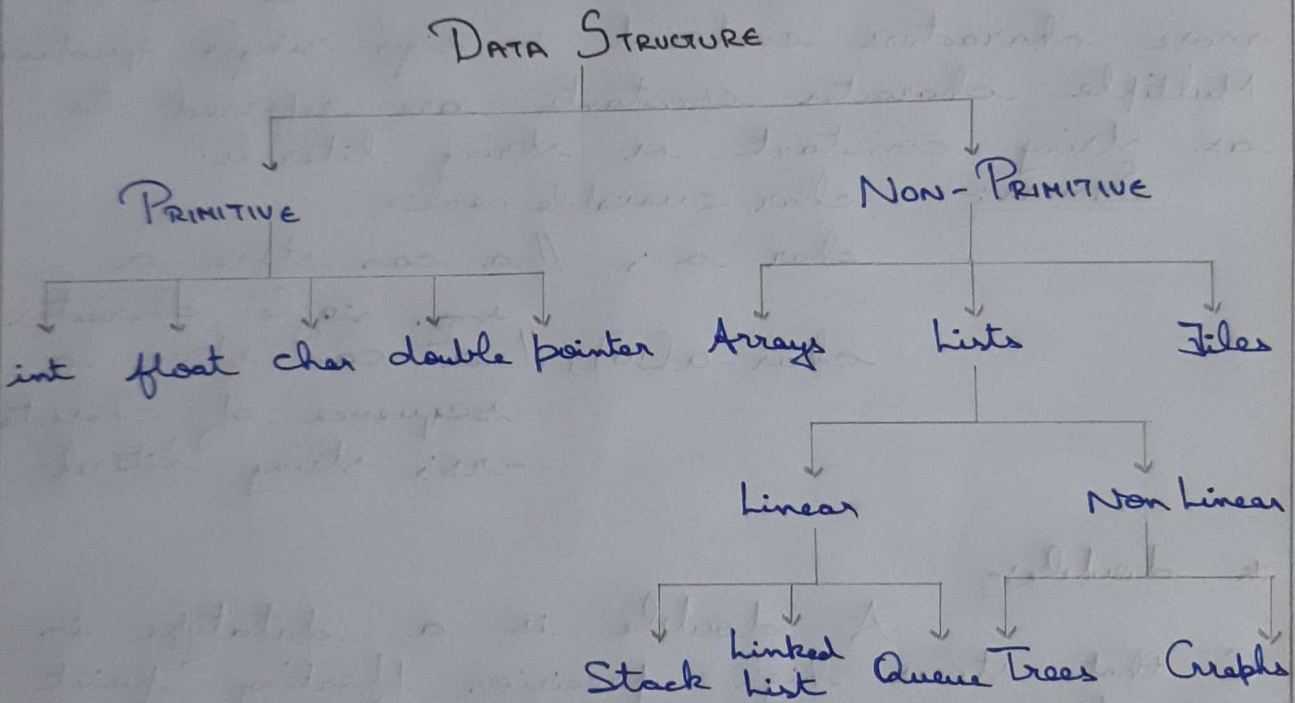
1) Define data structure. Explain the various types of data structures.

DATA STRUCTURE:-

Data structure is defined as a structure used for storing data in an organised and efficient manner. Data structure is broadly classified into 2 categories:-

i) Primitive data structure
ii) Non- primitive data structure



PRIMITIVE DATA STRUCTURE:-

The primitive data structures are the primitive / primary / basic data types. The int, char, float, double and pointer are the primitive data structures that can hold a signal value.

* int :-

An integer (written as int in C) is a whole number (not a fractional number) that can be +ve, -ve or 0.

SYNTAX:- int <variable name>;

int a; // a can store int values like -50, 50 etc.

* float :-

A floating point number usually has a decimal points. Numbers with a fractional part is called as floating (float) numbers.

SYNTAX :- float <variable name>;

float b; // b can store values like 0.5, -1.2 etc.

* char :-

A character constant is one or more characters enclosed in sign single quotes. Multiple character constants are referred to as string constant or string literals.

SYNTAX :- char <variable name>

char a; // a can store a sign single character

char a[10]; // a can store a sequence of characte -rs; string literal.

* double :-

A double is a datatype in C that stores high precision floating point data / numbers in computer memory. It is called as double data type because it can hold the double size of data compared to that of float data type.

SYNTAX :- double <variable name>

double x; // x can store values like 278.55, 5000 etc.

* pointer :-

A variable that is used to store the addresses of another variables

rather than values are called as pointer variables.

SYNTAX:- int * <variable names>;

```
int *p;     // p is a pointer
int a;      variable which points
p = &a;     to the address of
            another variable.
```

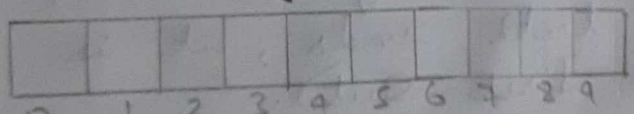
NON PRIMITIVE DATA STRUCTURE:-

Non-primitive data structures are more complicated data structures and are derived from primitive data structures.

* Array:-

An array is a linear data structure used for storing homogeneous elements in continuous memory location. It is called a linear data structure, since the elements are placed in a linear order ie, one after the other. The elements are stored continously to resolve the address starting from index 0.

Eg:- int age[10]; // variable age can store a total of 10 values in a continuous memory location.



* List:-

Lists are linear data structures consisting of a collection of nodes stored at continuous memory location.

Lists are classified into 2 categories:-

i) Linear lists
ii) Non linear lists.

LINEAR LISTS:-

Linear lists are data structures which store the data elements one after the

other, ie, in a linear manner.
There are 2 types of linear lists. They are as follows:-

⇒ STACK:-

A stack is a linear list in which insertion and deletion are allowed at only one end using a special variable called as top. The insertion operation is called as push() and deletion operation is called as pop(). Since, we are performing push() and pop() at only one end, the stack is called as LIFO (Last In First Out) or FILO (First In Last Out).

⇒ QUEUE:-

Queue is a linear list in which elements can be inserted only at one end ie, last position or rear position called as rear and elements can be deleted only at the other end ie, front or first position called as front. Since, queue is opened at both ends, it is called as FIFO (First In First Out).

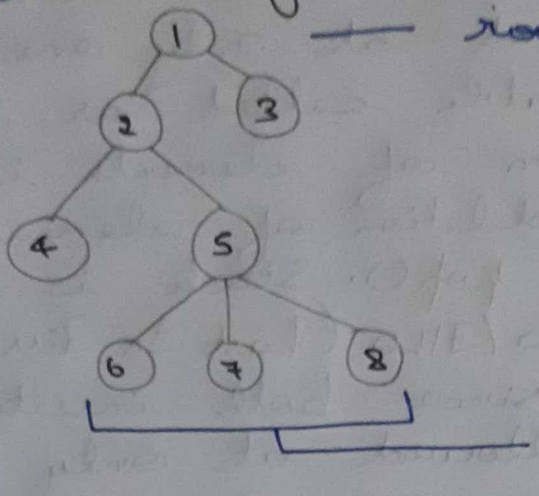
NON LINEAR LISTS:-

A data structure is said to be a non linear data structure if the data elements are stored in a distributed / random manner based on some condition. The 2 types of non linear lists are as follows:-

⇒ TREE:-

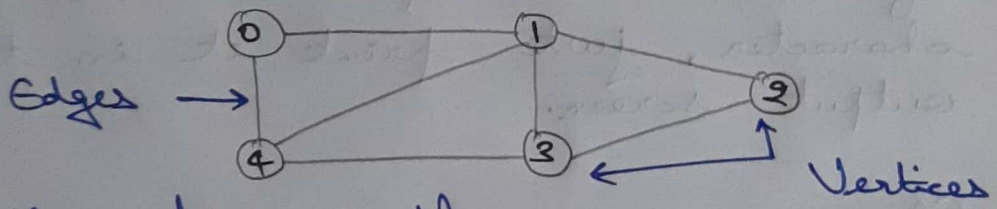
Trees are multi-level data structures with a hierarchical relationship among its elements known as nodes connected

by edges. The topmost node is called as the root node; while the bottom most node is called as the leaf nodes.



b) GRAPHS :-

A graph is a non linear data structure consisting of a finite set of vertices (or nodes) and a set of edges which connect a pair of nodes.



In the above graph,
set of vertices, $V = \{0,1,2,3,4\}$
set of edges, $E = \{01, 23, 34, 04, 14, 13\}$.

* FILES :-

A file is a collection of related information or records stored in a secondary storage such as tapes or disks.

---

3) Explain the algorithm for converting an infix expression to postfix expression. Convert the expression $((2*3)+(4-2))/2$ into postfix and also does the evaluation of the same expression using stack.

STACK:-

A stack is a linear data structure in which insertion and deletion of elements are allowed at only one end using a special variable called as top. The insertion operation of elements is called as push() and deletion of elements in stack is called as pop(). Stack is usually called as LIFO / FILO (Last In First Out / First In Last Out) since both insertion and deletion are allowed at only one end.

ALGORITHM FOR CONVERTING INFIX TO POSTFIX:-

Step:1 – Scan or analyze the infix expression from left to right.

Step:2 – If it is an operand or a character, just print it in the output screen.

Step:3 – Else,

i) If the scanned character is a '('
push it in the stack.

ii) If the precedence order of the scanned (incoming) operator is greater than the precedence order of the operator in the stack, push the incoming operator into the stack.

iii) Else, pop all the operators from the stack which are greater than or equal to in precedence than that of the incoming scanned character. After popping, push the scanned character to the stack.

Step: 4 – If the scanned character is ')', pop the operators in the stack, and print it until a '(' is encountered and discard both '(' and ')' parantheis.

Step-5: Repeat the steps 2 to 4 until the given infix expression is fully scanned.

STEP: 6 – Print the output.

Step: 7 – If still the stack contains any operator, pop it out until the stack is empty.

Step: 8 – End.

CONVERSION OF INFIX TO POSTFIX EXPRESSION:-

((2*3)+(4-2))/2

SNo	SCANNED	STACK	POSTFIX EXP.	DESCRIPTION
1)	(	(		'(' is an operator so push it to the stack.
2)	(	((		Again, push it to the stack.
3)	2		2	'2' is an operand, so print it
4)	*	((*	2	'*' operator, so push it to the stack.
5)	3	((*	23	'3' operand, print it
6)	)	((*)	23	')' is an operand, so pop all elements in stack until '(' is encountered.
		(	23*	

Sno	Scanned	Stack	Postfix Exp.	Description
7)	+	(+	23*	'+' is an operator, so push it in stack.
8)	(	(+(	23*	'(' push it in stack
9)	4	(+(	23*4	'4' is an operand so print it.
10)	-	(+(-	23*4	'-' is an operator push it in stack.
11)	2	(+(-	23*42	'2' is an operand, so print it.
12)	)	(+(-)	23*42	')', so pop all the elements in stack until '(' is encountered.
		(+	23*42-	
13)	)	(+)	23*42-	')' > pop until '(' is encountered.
			23*42-+	
14)	/	/	23*42-+	'/' is an operator, push it into the empty stack.
14)	2	/	23*42-+2	'2' is an operand, print it.
15)		Empty	23*42-+2/	Pop all the element -s in stack. End.

Postfix form :- 23*42-+2/

EXPLANATION:- The given infix expression is ((2*3)+(4-2))/2.

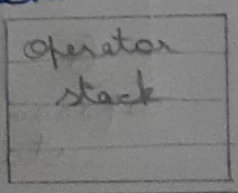
i) The first 2 encountered characters are '(' ; so push it into the stack.

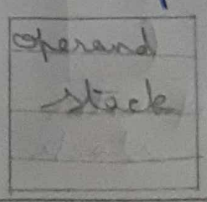
ii) Next, '2' is an operand print it in the output screen followed by '*' which is an operator so push it into the stack

containing 2 '('.
iii) Next in the expression is an operand 3, so print it and followed by ')', pop all the elements in the stack until '(' is encountered.
iv) '+' is an operand, push onto stack. Followe -d by '(' push it into the stack since '(' has lower precedence than '+'. Followed by 4, which is an operand so print it.
v) Now the stack contains elements (+(and the output screen has values 23*4.
vi) Now, scan the next element in the expression which is a '-', so push it in stack.
vii) '2' is an operand, so print it. And followed by 2, there is an ')'. So pop all the elements in stack, until '(' is encount -ered.
viii) Again ')', so pop and print all elements until '(' is encountered.
ix) '/' is an operator, push it in the empty stack.
x) '2' is an operand, print it.
xi) Since, now the expression is fully scanned, pop out all the elements from stack until the stack is empty. And the final postfix expression is:- 23*42-+2/

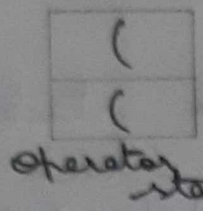
EVALUATION OF THE EXPRESSION:- ((2*3)+(4-2))/2

i) Take 2 stacks (one for operator and other for operand). Push all the operators in operator stack and operands in operand stack.





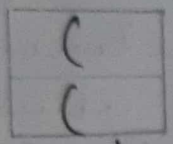
ii) Scan the infix expression from left to right. First 2 characters are '(', so push it in operator stack.



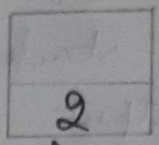
operator stack


operand stack

iii) Next character is 2, → push in operand stack.

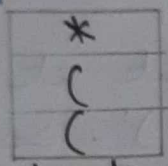


operator

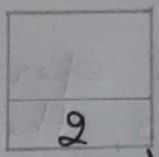


operand

iv) '*' → push in operator stack.

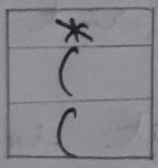


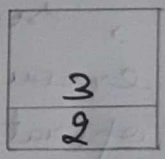
operator



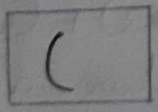
operand

v) '3' → push in operand stack.

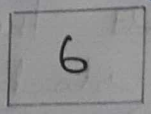




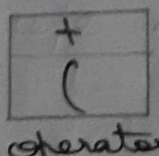
vi) ')' → push in operator stack and pop all operators in stack until '(' is encountered. By popping, we get '*' which is a binary operator, so pop 2 elements from the operand stack; perform the arithmetic evaluation and push the result back into the operand stack to proceed further.



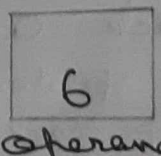
2*3 = 6 →



vii) '+' → operator, so push it in operator stack.



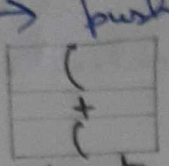
operator stack



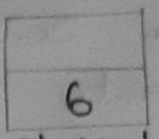
operand stack

A. -

viii) '(' → push it in operator stack.

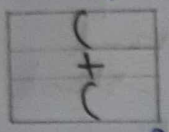


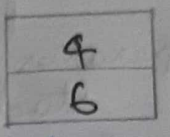
operator stack



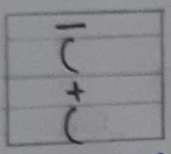
operand stack

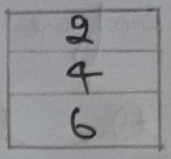
ix) '4' → push in operand.



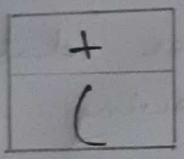


x) - → push in operator stack. '2' → push in operand stack.

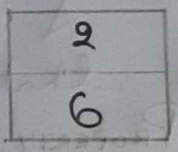




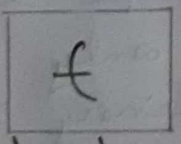
xi) ')' → push in operand stack and pop all elements until '(' is encountered. By popping, we get '-', perform evaluation & push the result back to operand stack.



$4 - 2 = 2 \longrightarrow$

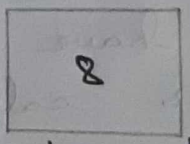


xii) ')' → pop all the elements in operand stack until '(' is encountered. By popping, we get, '+' which is a binary operator, so perform the evaluation and push the result onto the operand stack.



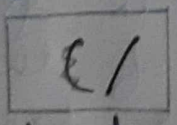
operator stack

$2 + 6 = 8 \longrightarrow$

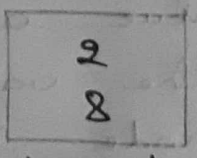


operand stack

xiii) '/' → push in operator stack and '2' in operand stack.



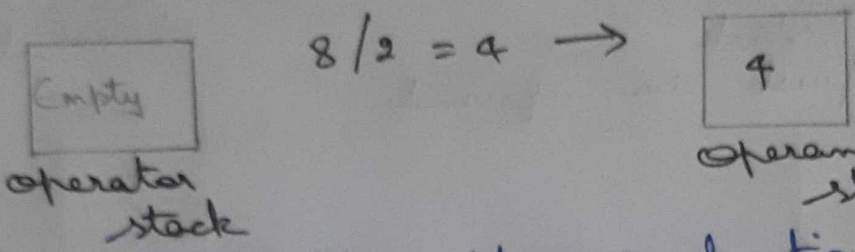
operator



operand

xiv) Now that the expression is fully scanned, empty the operand stack by popping out

the element and performing the arithmetic evaluation and pushing the evaluated result onto the operand stack.



The final result after evaluation of the given infix expression is 4.

3) Explain the heap sort algorithm in detail and demonstrate how it can be used for implementing priority queue.

HEAP TREE:-

A heap tree is a tree data structure that satisfies the following properties:-

1. SHAPE PROPERTY:-

Heap tree is always a complete binary tree which means that all the levels of a tree are fully filled. There should not be a node which has only one child. Every node except leaves should have two children then only a heap is called as complete binary tree.

2. HEAP PROPERTY:-

The property in which the value of the root node is greater than the value of the each of the child node is called as heap property. And every node has to follow the above property to become a heap tree.

ALGORITHM :-

STEP-1 :- Construct a complete binary tree or a heap tree with the given list of elements.

STEP-2 :- Once, the tree is created, heapify the tree ie, place the nodes according to the heap property and then eliminate the root node of the tree by shifting it to the end and then store the heap structure with the remaining elements.

STEP-3 :- Repeat the above step until all the elements are eliminated and (or) in other words; repeat the step until all the elements are sorted.

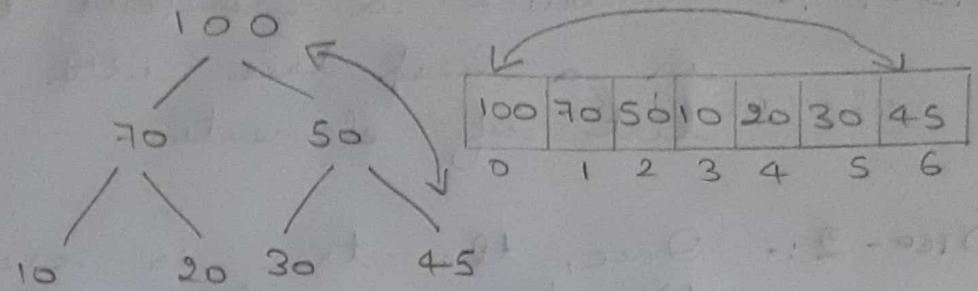
STEP-4 :- Finally, display the eliminated or sorted elements. By default, the elements are displayed in sorted ascending order.

EXAMPLE :-

* Suppose the array that is to be sorted contains the following elements :- 100, 70, 50, 10, 20, 30, 45.

* The first step is to create a heap tree which satisfies both shape property as well as heap property. In the heap tree, the left and right child of

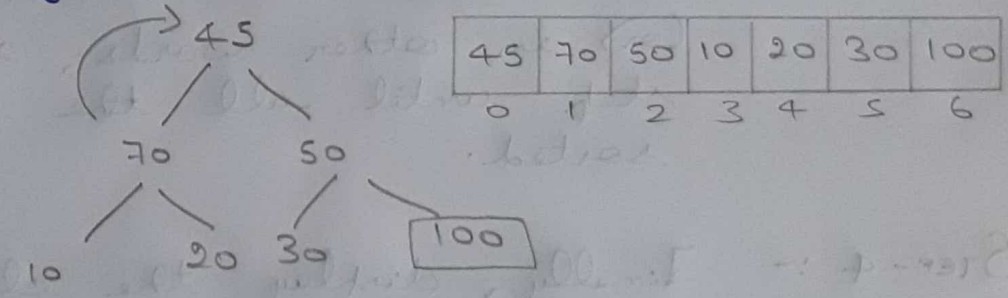
any element would be valued at $(2*i)+1$ and $(2*i)+2^{th}$ index respectively.



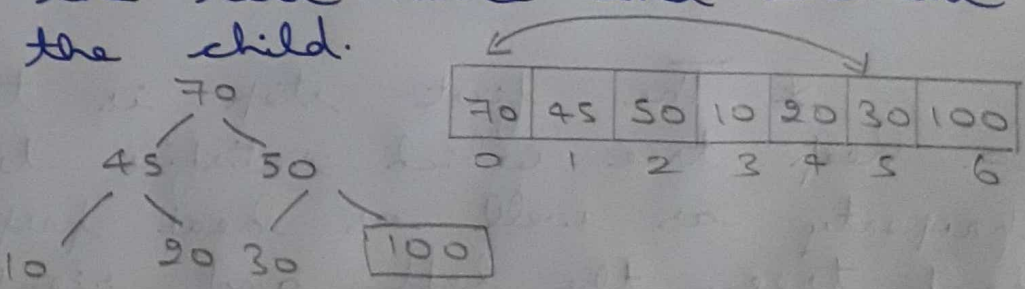
The above diagram, depicts the binary tree version of the given array elements.

STEPS:-

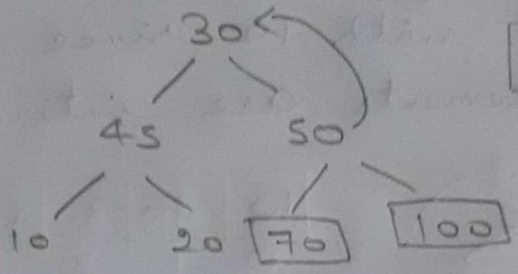
* Since, by default the root node has the greater value than its child node (ie, 100 > 70 and 50), swap 100 to the end of the array (ie, swap 100 with the element 45) to perform the heap sort.



* Element 100 is now sorted, being the greatest element among the given list of elements is now placed at the end of the tree ie, end of the array. Now, heapify the tree. By heapifying, element 70 becomes the root node and element 45 becomes the child.

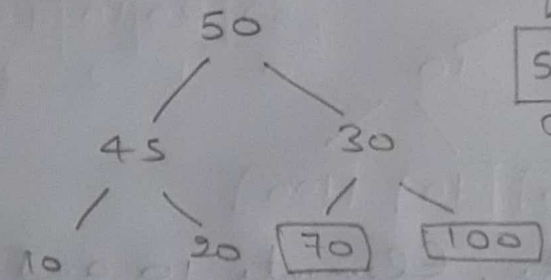


* Now, check whether the root node is greater than the last node of the tree. Since, 70 is not greater than 100, check with the next (ie, second) last node. Since, element 30 is lesser than element 70, place the element 70 at the second last position.

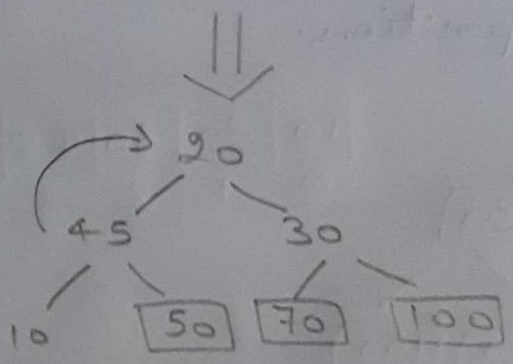


30	45	50	10	20	70	100
0	1	2	3	4	5	6

* Heapify the tree; place element 50 as root node and check with the last and second last node. Since element 50 is lesser than both 100 and 70, place the element at the third last position.

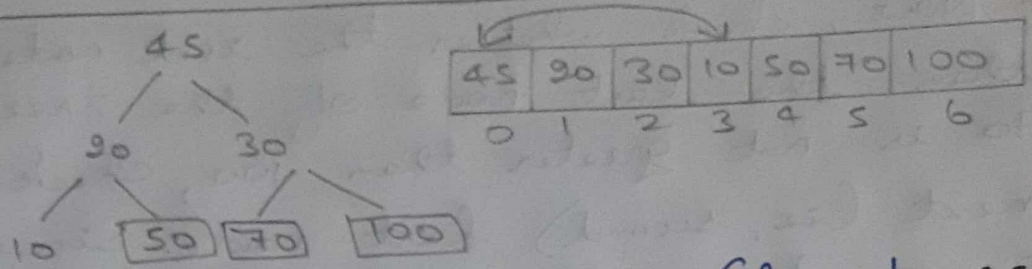


50	45	30	10	20	70	100
0	1	2	3	4	5	6

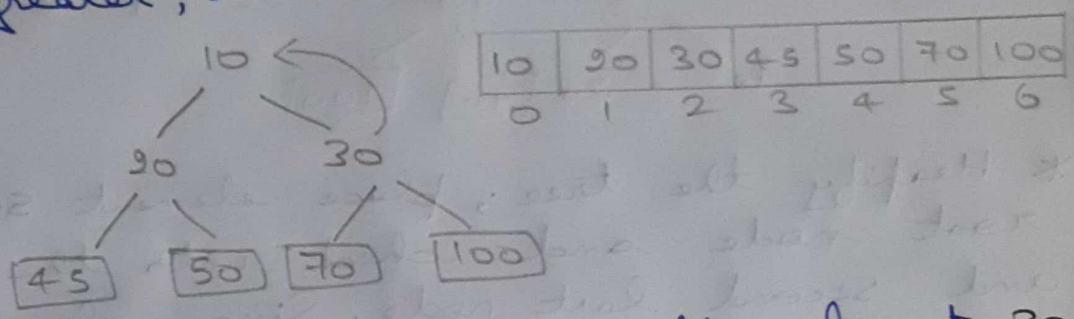


20	45	30	10	50	70	100
0	1	2	3	4	5	6

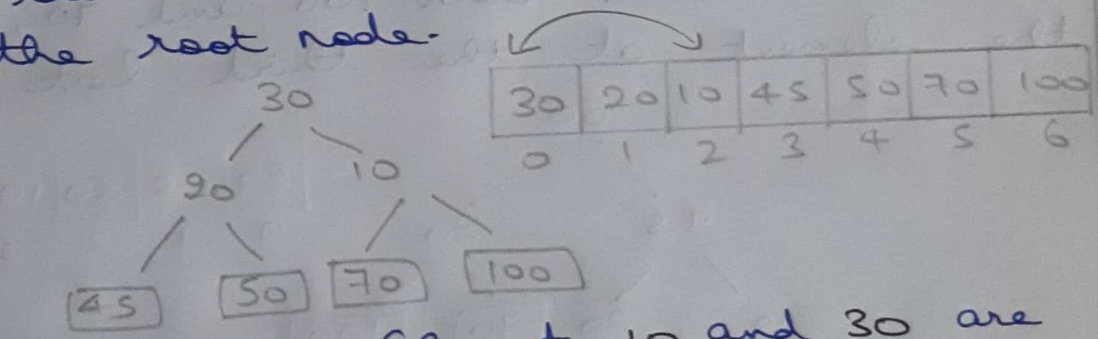
* Heapify the remaining elements. And continue the sorting process of the remaining elements. As a result of heapify, element 45 becomes the root node.



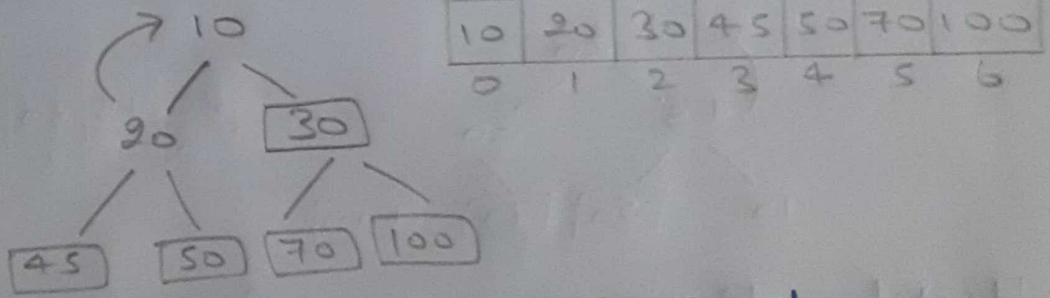
* Now, sort the array elements. Element 45 is checked repeatedly with 100, 70, 50 and 10 respectively. And with 10 since element 45 is greater, elements are interchanged.



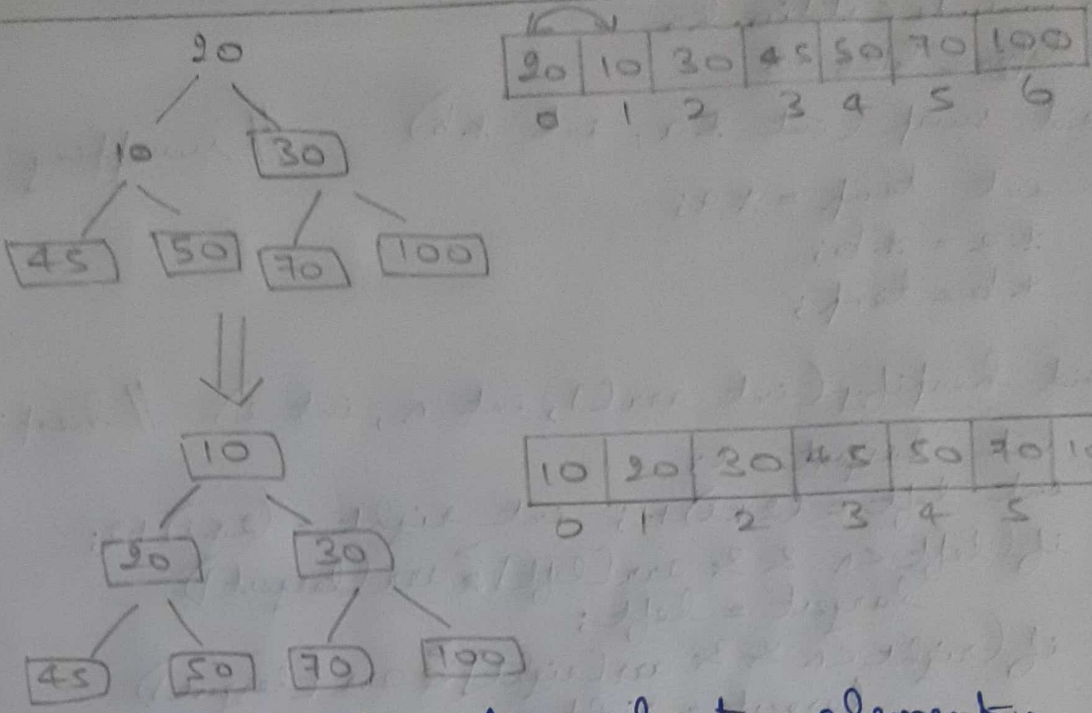
* Heapify the tree. As a result, element 30 becomes the root node.



* Sort the elements. Elements 10 and 30 are interchanged their positions.



* Heapify the tree and element 20 becomes the root node. And after that, as a result of sorting, element 10 and 20 interchange their positions.



* As we traverse to the last element, the elements in the array is fully sorted using the heap and thus called a heap sort.

PRIORITY QUEUE:-

A priority queue is a special special type of queue in which each element is associated with a priority value. And elements are served on the basis of their priority. The element with the highest value is considered the highest priority element and the highest priority element(s) are served or processed first. Also, the values are removed on the basis of priority in priority queue.

In short, the process of heapifying (ie, the process of making the highest value element from the given list of elements as root node and sorting / processing the same element is infact a way of implementing the priority queue.

PROGRAM:-

```
#include<stdio.h>
#define SIZE 5
void swap(int *a, int *b)          // swapping the values
{
    int temp = *a;
    *a = *b;
    *b = temp;
}
void heapify(int arr[], int n, int i)   // heapify the elements
{
    int largest = i;
    int left = (2*i)+1; int right = (2*i)+2;
    if(left<n && arr[left] > arr[largest])
        largest = left;
    if(right<n && arr[right] > arr[largest])
        largest = right;
    if(largest!=i)
    {
        swap(&arr[i], &arr[largest]);
        heapify(arr, n, largest);
    }
}
void heapsort(int arr[], int n)     // sorting the elements after heapify process.
{
    for(int i = n/2-1; i>=0; i--)
        heapify(arr, n, i);
    for(int i=n-1; i>=0; i--)
    {
        swap(&arr[0], &arr[i]);
        heapify(arr, i, 0);
    }
}
void print(int arr[], int n)        // printing the sorted array elements
{
    for(int i=0; i<n; i++)
        printf("%d ", arr[i]);
    printf("\n");
}
int main()                          // Driver function.
{
    int arr[SIZE] = {11, 2, 10, 50, 25};
    int n = sizeof(arr)/sizeof(arr[0]);
    heapsort(arr, n);
    printf("Heap Sorted array is \n");
    print(arr, n);
    return 0;
}
```

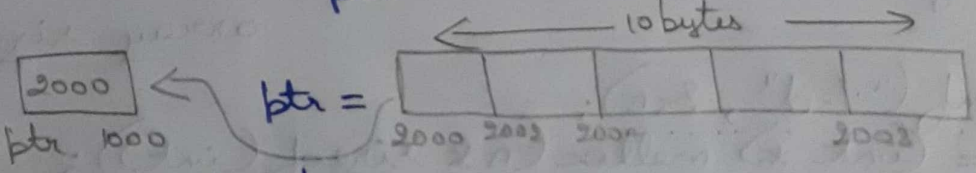
4) Explain dynamic memory allocation functions in detail.

DYNAMIC MEMORY ALLOCATION :-

Dynamic memory allocation is a procedure in which the size or memory of a data structure (like array) is allocated / changed during runtime. There are 4 functions as follows :-

i) malloc() :-

* malloc() (or) memory allocation method in C is used to dynamically allocate a single large block of memory with the specific size.
* It doesn't initialise memory at execution time so that it initializes each block with the default garbage value initially.
* SYNTAX :- (datatype *) malloc (sizeof (datatype)).
* Eg :- int *ptr; = (int *) malloc (5 * sizeof (int));



* ptr is a pointer variable which holds the address of the first byte in the allocated memory. A large 10 bytes of memory is dynamically allocated to the pointer variable.
* If the space is insufficient, the allocation fails and returns a null pointer.

ii) free() :-

i) free() method in C is used to dynamically deallocate the memory.
ii) The memory allocated by malloc() or calloc() is not deallocated on their own. Hence, the free() method is used, whenever, the

memory is allocated dynamically.

* It is the responsibility of the programmer to make use of free() method function to release the memory since one of the statements in the programe is dynamically allocating the memory.

* SYNTAX:- free (dynamically allocated memory variable);
free(ptr);

ILLUSTRATION PROGRAM:-

```
# include <stdio.h>
# include <stdlib.h>   // dynamic memory allocation functions are defined under this header file.

int main()
{
    int *ptr;
    int i, n;
    printf("\n Enter the no. of elements in array size :");
    scanf("%d", &n);
    ptr = (int *) malloc (n * sizeof (int));
                // dynamically allocates memory locations, each of size int.
    if (ptr == NULL){
        printf("Memory not allocated");
        exit(0);
    }
    else
    {
        printf("\n Memory allocated successfully");
        for(i=0; i<n; i++)
        {
            *ptr[i] = i+1;
        }
```

```
printf("\n The elements of the array are:");
for (i=0; i<n; i++)
{
    printf("%d ", *ptr(i));
}
}
free(ptr); // deallocating the memory.
printf("\n Memory allocated by malloc() is freed");

    return 0;
}
```

SAMPLE OUTPUT:-

Enter the no. of elements in array : 5
Memory allocated successfully.
The elements of the array are : 1 2 3 4 5
Memory allocated by malloc() is freed.

iii) calloc():-

* calloc() or contingous allocation method in C is used to dynamically allocate the specified number of blocks of memory of the specified type.

* It is similar to malloc() but has 2 differences. They are as follows:-
  i) It initializes each block with a default value of 0.
  ii) It has 2 arguments / parameters compared to malloc().

* SYNTAX:- ptr = (datatype *) calloc(no. of memory locations, sizeof(datatype));

ptr(int *) calloc(5, sizeof(int));

The above statement allocates contingous space in memory with 5 memory locations each of size 2 bytes.

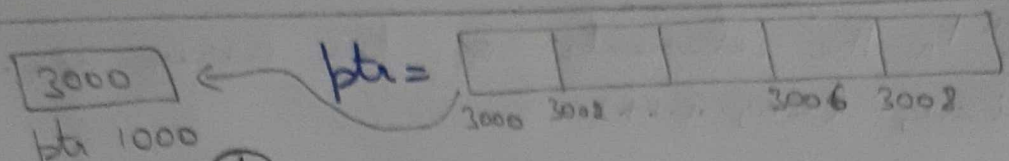


Illustration Program:-

```
#include <stdio.h>
#include <stdlib.h>
int main()
{
    int *p, i;
    p = (int *) calloc (3, sizeof (int)); // memory allocated dynamically
    if (p == NULL)
    {
        printf ("\n Memory not allocated");
        exit (0);
    }
    else
    {
        *(p+0) = 10;
        *(p+1) = 20;
        *(p+2) = 30;
        printf ("\n Memory allocated using calloc");
        for (i=0; i<3; i++)
        {
            printf ("%d ", *(p+i));
        }
    }
    free (p);  // deallocating the dynamically
                  allocated memory.
    return 0;
}
```

Sample Output:-

Memory allocated using calloc

10 20 30

iv) realloc():-

* realloc() or re-allocation method in C is used to dynamically change the memory allocation of a previously allocated memory. In short, realloc() is used to change the memory size allocated using malloc() and calloc().

* Reallocation of memory maintains the already present value and new blocks will be initialized with default garbage value.

* SYNTAX:- ptr = (int *) realloc (ptr, new size);

```
int *ptr;
ptr = (int *) malloc (5 * sizeof(int));
```
↳ This statement allocates 10 bytes of memory.
```
ptr = (int *) realloc (ptr, 10 * sizeof(int));
```
↳ This statement reallocates the memory size to 20 bytes (i.e, 10 bytes is added to the ptr. variable ptr).

* Similarly, we can also decrease the allocated memory size. And the excess memory space after decreasing is automatically removed by the compiler.

ILLUSTRATION PROGRAM:-

```
# include <stdio.h>
# include <stdlib.h>
int main()
{
    int *ptr, n, i;
    n = 5;
    printf("The array size : %d", n);
```

```
ptr = (int *) calloc (n, sizeof (int));
if (ptr == NULL)
{
  printf ("\n Memory not allocated");
  exit (0);
}
else
{
  printf ("\n Memory allocated successfully
                    using calloc");
  for (i=0; i<n; i++)
  {
    ptr[i] = i+1;
  }
  printf ("\n The elements are :");
  for (i=0; i<n; i++)
  {
    printf ("%d ", ptr[i]);
  }
  n = 10;   // changing the array size.
  printf ("\n New array size : %d ", n);
  ptr = realloc (ptr, n * sizeof (int));
          // reallocating the memory size
                of ptr variable with
                the new array size 'n'.
  printf ("\n Memory reallocated using
                    realloc");
  for (i=5; i<n; i++)
  {
    *ptr[i] = i+1;
  }
  printf ("The elements are :");
  for (i=0; i<n; i++)
  {
    printf ("%d ", *ptr[i]);
  }
```

```
    }
    free(ptr); // deallocating the dynamically
                  allocated memory.
    return(0);
}
```

SAMPLE OUTPUT:-

```
The array size : 5
Memory allocated successfully using calloc
The elements are: 1 2 3 4 5

New array size : 10
Memory reallocated using realloc
The elements are : 1 2 3 4 5 6 7 8 9 10
```