DATE: 15 12 2020
PAGE:

Variables =

Variables name are the placeholders for representing the data.

Data types :-

- A datatype in a programming language is is a set of data with the values having predefined characteristics.

- System/Complier defined data type are called primitted data type.

- Structure in C/C++ and classes in C++/JAVA are means to create our own data type known as user defined data type.

Data Structure :-

Data structure is a way to store and organize the data in a computer so that it can be used at the high efficiency.

Good Write

DATE: ___/___/____
PAGE: _____

Classification

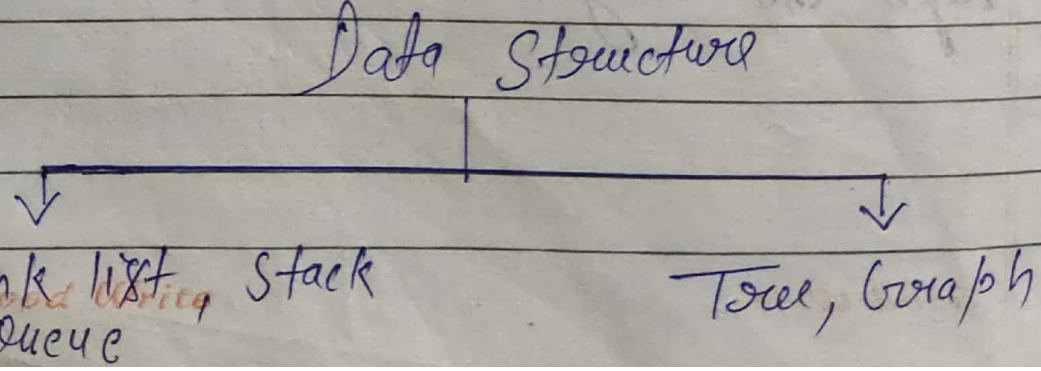
There are two major classification of data structure
i.e linear and Non-linear

— Linear data structure

- List (Link list)
- Stack
- Queue

— NonLinear data structure

- Tree
- Graph



DATE: / /
PAGE:

Astract data type :-

- All primitted data types support all basic operations like addition, subtraction etc.
- The system is providing the implementation for the primitive data types.
- For non premitted data types we also need to define operations.
- Combination of data structure and their operations are known as Abstract data type.
- So data structure is all about creating abstract data type.
- Any peice of information can be handled by defining appropriate data type and set of possible operations.

Good Write.

DATE: __/__/____
PAGE: _____

Algorithm :-

Algorithm is the step by step instruction to solve a given problem.

eg

sum(n): Algorithm to add first n natural numbers. Assume S is a variable initialized to 0.

1. If $n <= 0$ return -1
2. Repeate step 3 and step 4 while $n != 0$
3. $S = S + n$
4. (decreament) $n--$
5. return S

Good Write

DATE: / /
PAGE:

Analysis of algorithm :-

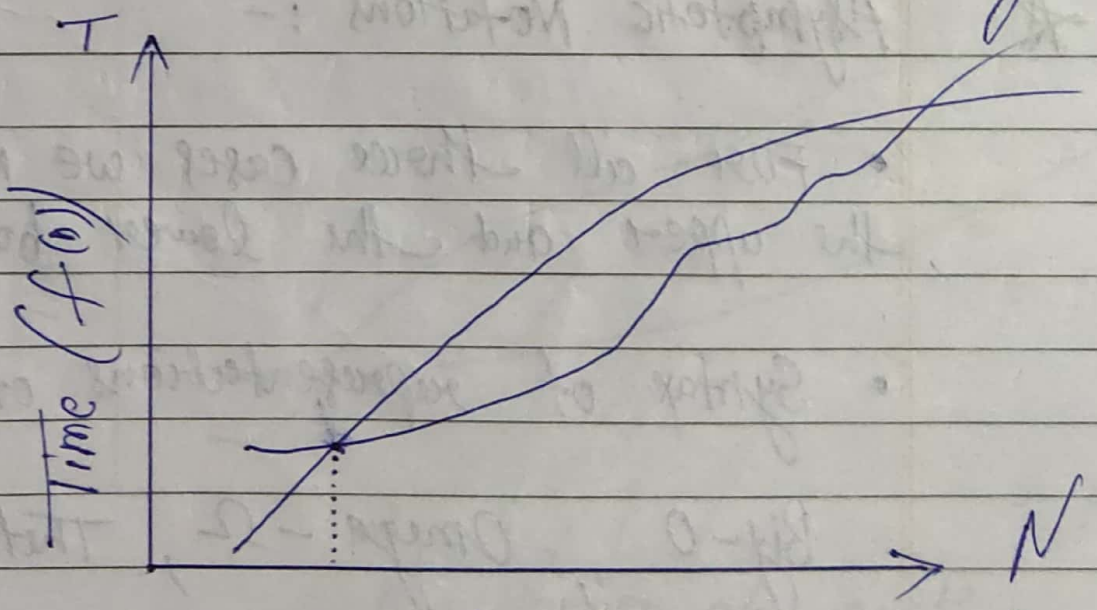
The goal of analysis of algorithm is to compare algorithms mainly in terms of running time, but also in terms of other factors like memory, efforts etc.

Rate of growth :-

- The rate at which the running time increases as a function of input is called rate of growth.

- $f(n) = n^4 + 2n^2 + 100n + 500$

- $f(n) = n^4$, when n is large.



Good Write

DATE : ___/___/_____
PAGE : _____

So we compare the graph of different algorithm to find out which one is suitable.

★ Types of analysis :-

To analyse the given algorithm we need to know on what inputs the algorithm is taking less time and on what inputs the algo is taking huge time. There might be three cases

- Worst case
- Average case
- Best case

★ Asymptotic Notations :-

- For all three cases we need to identify the upper and the lower bounds.
- Syntax of representations of bounds are

Big-O , Omega - Ω , Theta - Θ

Good Write

DATE : __/__/____
PAGE : _____

Big - O Notation :—

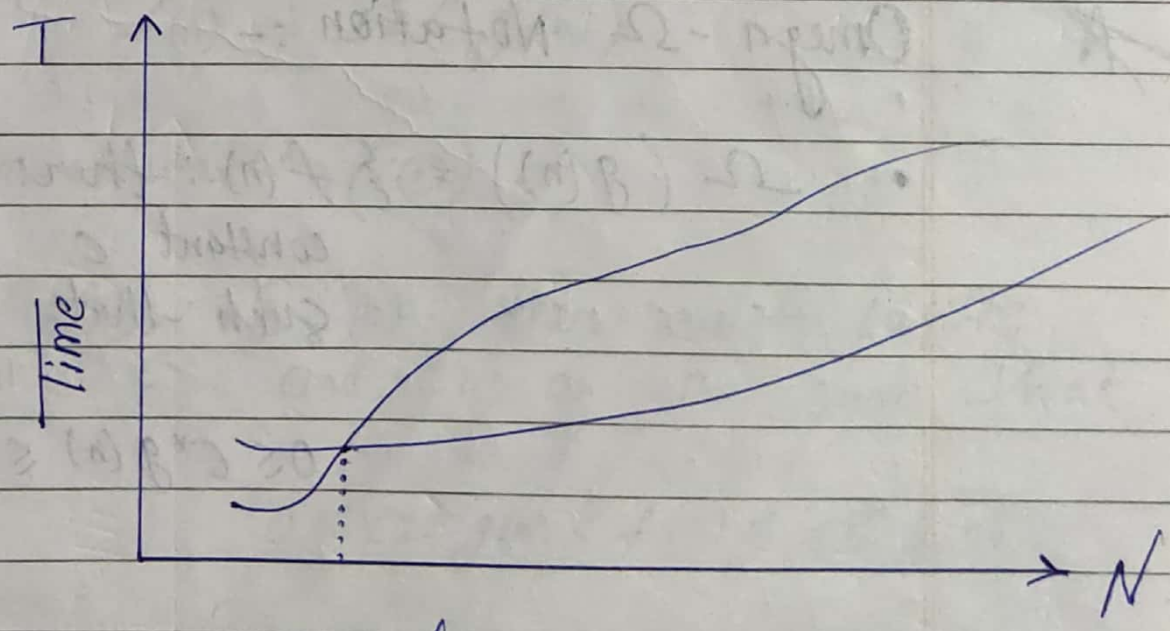
- $O(g(n)) = \{ f(n) :$ there exits positive constraints c and n_0 such that $0 \leq f(n) \leq c * g(n)$ for all $n \geq n_0 \}$

- The statement $f(n) = O(g(n))$ states only that $g(n)$ is an upper bound on the value of $f(n)$ for all n, $n \geq n_0$.

eg,

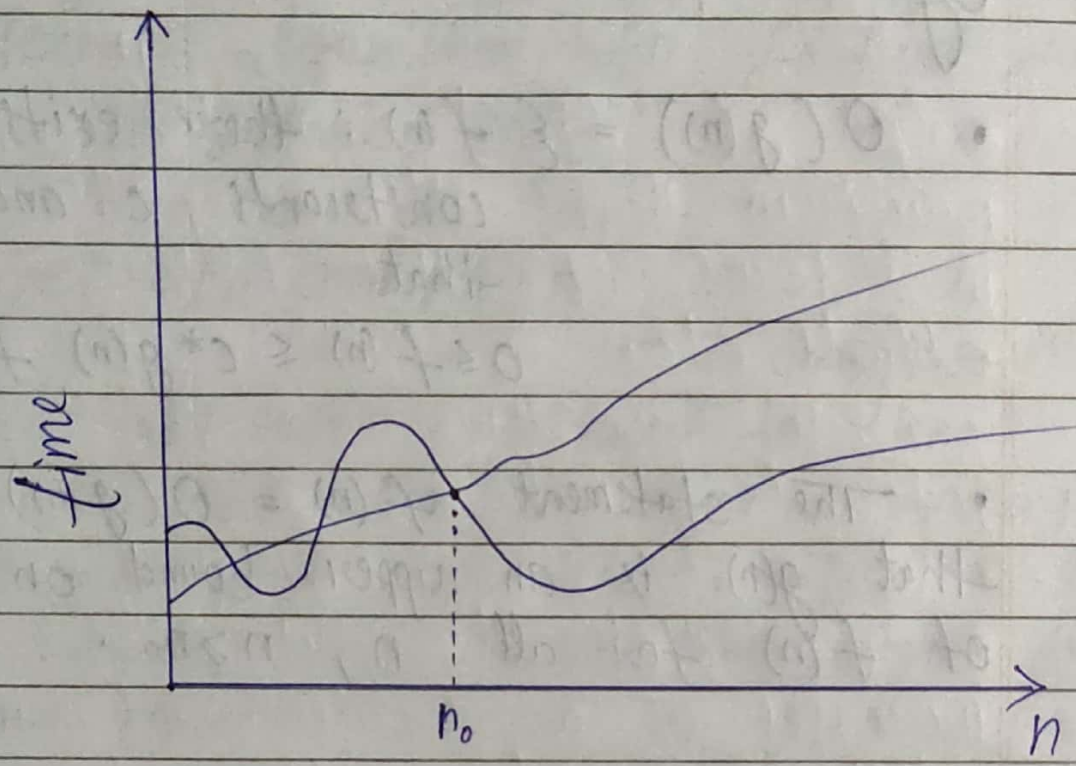
$f(n) = n^2 + 3n + 1$

$2\,g(n) = 2n^2$



N = Number of inputs

Good Write



$$f(n) = O(g(n))$$

Omega - Ω Notation :-

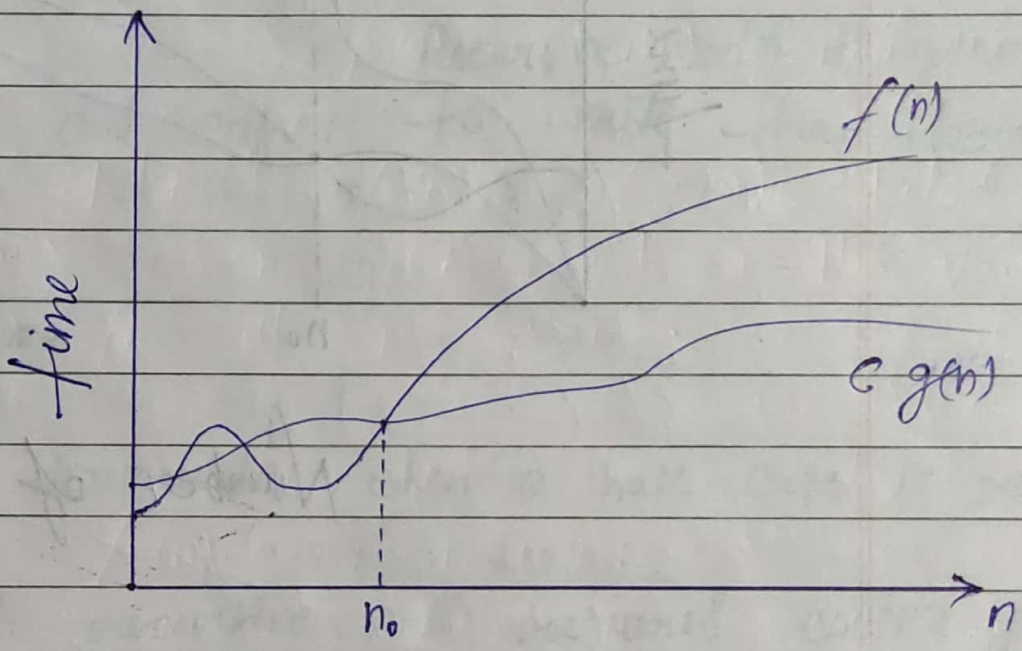
- $\Omega(g(n)) = \{ f(n)$: there exit positive constant c and no such that

$$0 \leq c * g(n) \leq f(n) \}$$

Good Write

DATE : ___/___/_____
PAGE : _____

- The statement $f(n) = \Omega(g(n))$ states only that $g(n)$ is only a lower bound on the value of $f(n)$ for all n, $n \geq n_0$



$$f(n) = \Omega(g(n))$$

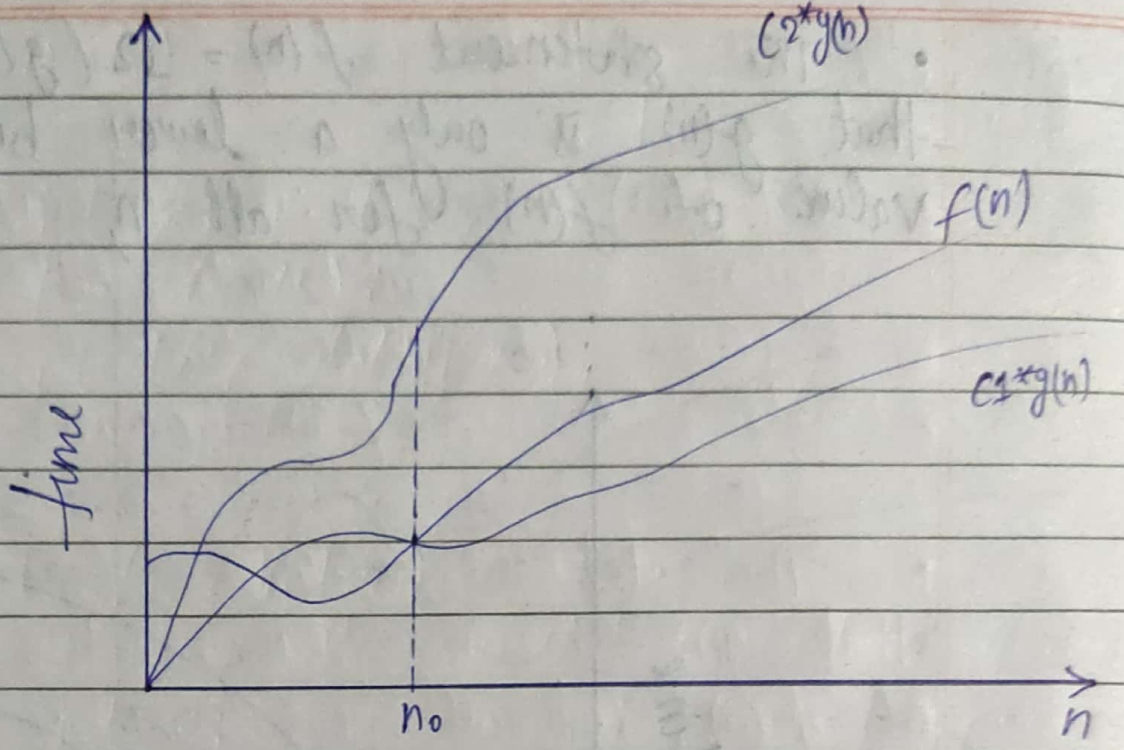
Theta - Notation (Θ) :-

- $\Theta(g(n)) = \{ f(n)$: there exist positive constants c_1 and c_2 and n_0 such that

$$0 \leq c_1 * g(n) \leq f(n) \leq c_2 * g(n)$$

for all $n \geq n_0$

Good Write



Number of inputs

Recursion :—

- Any function which calls itself is called recursive.
- A recursive method solves a problem by calling a copy of itself to work on a smaller problem.
- It is important to ensure that the recursion terminates.

Good Write

DATE: ___/___/____
PAGE: _____

- Each time the function call itself with a slightly simpler version of the original problem.

★ Why recursion :-

Recursive code is generally shorter and easier to write than iterative code.

★ Recursion

- It terminates when a base case is reached.
- Each recursive call requires extra space on the stack frame (i.e memory).
- Solution to some problems are easier to formulate recursively.

Good Write

DATE: __/__/____
PAGE: _____

```
main ()
{
    int k;
    k = fun(5);
    printf("%d", k);
}

int fun(int a)
{
    int s;
    if (a==1)
    return (a)
    s = a + fun(a-1);
    return (s);
}
```

Good Write

DATE: / /
PAGE:

Applications of recursion :—

- Fibonacci Series
- Factorial of a number
- Merge sort, quick sort
- Binary Search
- Tree traversal (DFS, BFS)
- Dynamic Programming
- Divide and conquer algorithm
- Tower of Hanoi
- Backtracking algorithm

Backtracking :—

- Backtracking is the method of exhausted search using divide and conquer.
- Sometimes the best algorithm is to try all possibilities.
- This is always slow.

Good Write

DATE: __/__/____
PAGE: _____

Recursion Problems :-

Here we will discuss few problems based on recursion.

- Tower of Hanoi
- Factorial
- Fibonacci Series
- Greatest common Divisor
- Printing all permutations of a given string.
- Generate all strings of n bits of binary digits.
- Define Power function which could handle negative powers.
- Writing a recursive function to print entered characters in reverse order (TNRN)
- Writing a recursive function to convert decimal to binary.

DATE: 22 12 2020
PAGE:

Tower of Hanoi :-

- The tower of Hanoi is mathematical puzzle.

- The objective of the puzzle is to move the entire stack to another rod.

Rules

- Only one disk may be moved at a time.
- Each move consists of taking the upper disk from one of the rods and sliding it onto another rod, on the top of other disks that may already be present on that rod.
- No disk may be placed on the top of smaller disk.

Good Write

DATE: / /
PAGE:

logic

- Move n-1 top 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 the top n-1 disks from source to auxillary tower can again be thought as fresh problem and can be solved in same manner.

⇒ Interesting facts about tower of Hanoi-

- The Hanoi's tower problem was invented by the french mathematician Edouard Lucas in 1883.
- The number of moves required to correctly move a tower of 64 disks is $2^{64} - 1 = 18,446,744,073,709,551,615$

Good Write

DATE: __/__/____
PAGE: _____

- At a rate of one move per second that is 584, 942, 417, 355 years

Solution

```
void TOH(int n, char BEG, char AUX,
                        char END)

{
    if (n>=1)

    {
        TOH(n-1, BEG, END, AUX);
        printf("%c to %c\n" BEG, END);
        TOH(n-1, AUX, BEG, END);
    }
}
```

DATE : ___/___/____
PAGE : _____

Factorial

```
long factorial (int n)
{
    if (n>0)
    return (n * factorial (n-1));
    else
    return (1);
}
```

Good Write

Program

```
long fact(int n)
{
    if (n>0)
    return (n*fact(n-1));
    else
    return (1);
}
main()
{
    clrscr();
    printf("factorial of 5 is %ld" fact(5));
    getch();
}
```

Good Write

DATE: ___/___/____
PAGE: _____

Fibonacci Series :-

```
int fib (int n)

{
    if (n == 1 || n == 2)
    return 1

    return (fib (n-1) + fib (n-2));
}

main ()

{
    clrscr ();
    printf ("6th term is %d", fib(6));
    getch ();

}
```

Good Write

```
int fib (int n)
{
   if (n==1 :: n==2)
   return (1);
   return ( fib(n-1) + fib(n-2));
}
main ()
{
   int i;
   clrscr();
   for (i=1;i<=10; i++)
   printf ("%d", fib(i));
   getch ();
}
```

Good Write

DATE : ___/___/____
PAGE : _____

Greatest Common Divisor

```
int GCD (int a, int b)
{
  if (a==b)
  return (a);
  if (a%b==0)
  return (b);
  if (b%a==0)
  return (a);
  if (a>b)
  return (GCD(a%b, b));
  else
  return (GCD(a, b%a));
}
main()
{
  clrscr();
  printf("GCD is %d", GCD(105, 91));
  getch();
}
```

Good Write

DATE: / /
PAGE:

Stack :-

- A stack is a list of elements in which an element may be inserted or deleted only at one end, called top of the stack.
- Stack is a specialized data storage structure (Abstract data type).

Array and stack :-

- Unlike arrays, access of elements in a stack is restricted.
- It has two main functions push and pop.
- Insertion in stack is done using push function and removal from a stack is done using pop function.

Good Write

DATE : __/__/____
PAGE : _____

★ Representation :-

- Stack allows access to only the last element inserted hence, an item can be inserted or removed from the stack from one end called the top of the stack.

- It is therefore, also called Last-in-First-out (LIFO) list.

★ Implementation

- Stack can be implemented as

— Array
— Dynamic Array
— Linked List

Good Write

DATE : __/__/____
PAGE : _____

Algorithm PUSH

PUSH (STACK, TOP, MAXSTK, ITEM)

This Procedure pushes an item onto stack.

1. [stack already filled]

   If TOP == MAXSTK, then

   print OVERFLOW and return.

2. Set TOP = TOP + 1

3. Set STACK [TOP] = ITEM.

4. return.

Good Write

DATE : __/__/____
PAGE : _____

Algorithm POP :-

POP (STACK , TOP , ITEM)

This procedure deletes the top element of STACK and assigns it to the variable ITEM.

1. [Stack has an item to be removed ?]

   If TOP == 0 [In C language If TOP == -1]

   print : UNDERFLOW and return.

2. Set ITEM = STACK [TOP]

3. Set TOP = TOP - 1

4. return.

Good Write

DATE: ___/___/____
PAGE: _____

STACK using array :-

```
struct ArrayStack
{
    int top;
    unsigned capacity;
    int* array;
};
```

DATE: 22/01/2021
PAGE:

★ Programming For stack :-

```
Struct ArrayStack
{
    int top;
    int capacity;
    int *array;
}

struct ArrayStack * createStack (int cap)
{
  struct ArrayStack *stack;
  stack = malloc(sizeof(struct Arraystack));
  stack -> capacity = cap;
  stack -> array = malloc(sizeof(int)*stack
                          -> capacity);
  return(stack);
}

int isFull (struct ArrayStack *stack)
{
    if (stack -> top == stack -> capacity-1)
    return(1)
    else
    return(0)
}
```

Good Notes

```
int isEmpty (struct ArrayStack *stack)
{
    if (stack -> top == -1)
    return(1);
    else
     return(0);
}

void push (struct Arraystack * stack, int, item)
{
    if (!isFull(stack))
    {
      stack -> top++;
      stack -> array (stack->top) = item;
    }
}

int pop(struct ArrayStack *stack)
{
    int item;
    if (!isEmpty(stack))
    {
       item = stack -> array[stack -> top];
       stack -> top --;
       return (item);
    }
    return (-1);
}
```

DATE: __/__/____
PAGE: _____

```
main ()
{ int choise, item;
  struct ArrayStack *stack;
  while (1)
   stack = createstack(4);

 while (1)
 {
    clrscr();
    printf("\n1 Push");
    printf("\n2 Pop");
    printf("\n3 Exit");
    printf("\n Enter Your choise \n");
    scanf("%d", &choise);

    switch (choise)
    {
      case 1;
          printf("\n Enter your number");
          scanf("%d", &item);
          push (stack, item);
          break;
```

Good Write

```
case 2;
    item = pop(stack);
    if (item == -1)
    printf(" stack is Empty ");
  else
  printf("\n poped value is %d"
                        , item);
  break;

case 3;
      exit(0)
    }

getch();

}

}
```

Good Write

DATE: __/__/____
PAGE: _____

Printing all permutations of a String-

```
void permutation (char *s, int i, int n)
{
  static int count;
      int j;

  if (i == n)

    {
      count ++;
      printf("(%d) %s \t", count, s);
    }

  else

  for (j = i; j <= n; j++)

      {
        swap((s+i), (s+j));
        permutation (s, i+1, n);
        swap ((s+i), (s+j)); // Backtracking
      }
}
```

Good Write

DATE: ___/___/____
PAGE: _____

```
void swap(char *x, char *y)
{
    char ch;
    ch = *x;
    *x = *y;
    *y = ch;
}
main()
{
    char *str;
    clrscr();
    printf("Enter a string");
    get(str);
    permutation(str, 0, strlen(str)-1);
    getch();
}
```

Good Write

DATE : ___/___/_____
PAGE : _____

A Printing all binary strings —

```
void bin(int n, char A[])
{
    if (n<1)
    printf("\n %.s", A);

    else
    {
        A[n-1] = '0';
        bin(n-1, A);
        A[n-1] = '1';
        bin(n-1, A);
    }
}

main()
{
  void bin(int, char[]);
  char A[11]
  clrscr();
  A[10] = '\0';
  bin(10, A);
  getch()
}
```

Good Write

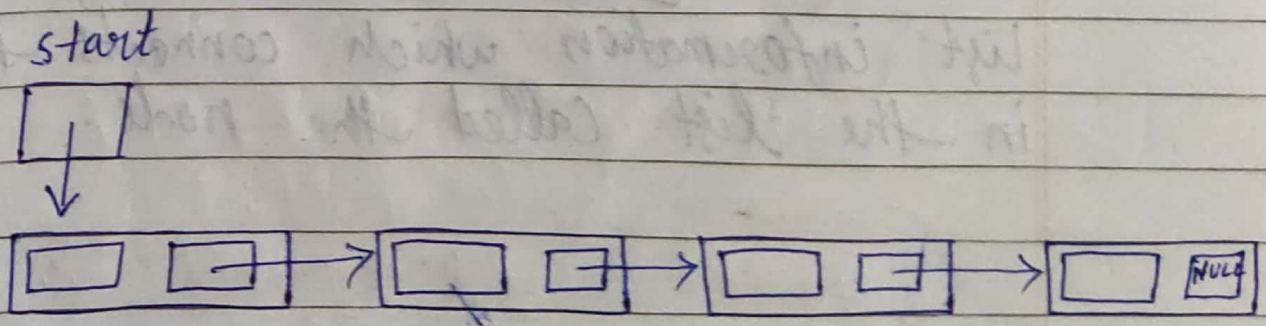
DATE: __/__/____
PAGE: _____

Linked List :—

- Link list is a data structure used for storing collection of data.
- Linked list has the following properties

  — successive elements are connected by pointer.
  — Last element points to NULL.
  — It can grow or shrink the size during execution of program.
  — It can be made just as long as required.
  — It does not waste memory space.

Diagram representation —



Good Write

DATE: __/__/____
PAGE: _____

Operations

- There are three main linked list operations :—

Insertion

Deletion

Traversing

Node :—

What we talk about the storing list information, each information set is a node which also plays a role of connection point in the list.

Each information set in list information which connects the point in the list called the node.

Good Write

DATE: ___/___/____
PAGE: _____

→ Defining a node (Data type)

```
struct node
{
    int info;
    struct node *link;
};
```

→ Operations

- Insertion
  - At the end
  - At the beginning
  - After a node
- Deletion
  - Last node
  - First node
  - Particular node
- Traversing

Good Write

DATE: / /
PAGE:

Progeramming For linked list :-

```
#include <stdio.h>

struct node
{
    int info;
    struct node *link;
};

struct node* createNode()
{
    struct node *n; ~~*t;~~
    n = (struct node *) malloc (sizeof
                    (struct node));
    return (n);
}

void insertNode()
{
    struct node * temp, *t;
    temp = createNode();
    printf("Enter a number");
    scanf("%d", &temp -> info);
    temp -> link = NULL;
```

```
if (START == NULL)
    START = temp;

else
{
    t = START;
    while (t -> (link != NULL)
        t = t -> link;
    } t -> link = temp;
}
}

void deleteNode()
{
    struct node *r;
    if (START == NULL)
    printf("List is Empty");

    else
    {
        r = START
    START = START -> link;
    free(r);
    }
}
```

Good Write

DATE: 24/01/2021
PAGE:

```
void viewList()
{
    struct node *t;
    if (START == NULL)
    printf("List is Empty");
    else
    {
      t = START;
      while (t != NULL)
      {printf("%d", t->info);
        t = t->link;
      }
    }
}


int menu()
{
    int ch;
    printf("\n Enter your choice")
    printf("\n 1. Add value to the list")
    printf("\n 2. Delete first value")
    printf("\n 3. Exit")
    printf("\n 3. Viewlist")
    printf("\n 4. Exit")
    Scanf("%d", &ch);
    return (ch);
}
```

Scanned with CamScanner

```
void main ()
{
    clrscr();

    switch (menu())
    {
        case 1;
            insertNode();

        case 2;
            deleteNode();

        case 3;
            viewList();

        case 4;
            exit(0);

        default:
            printf("Invalid Entry");
    }
    getch();

}
```

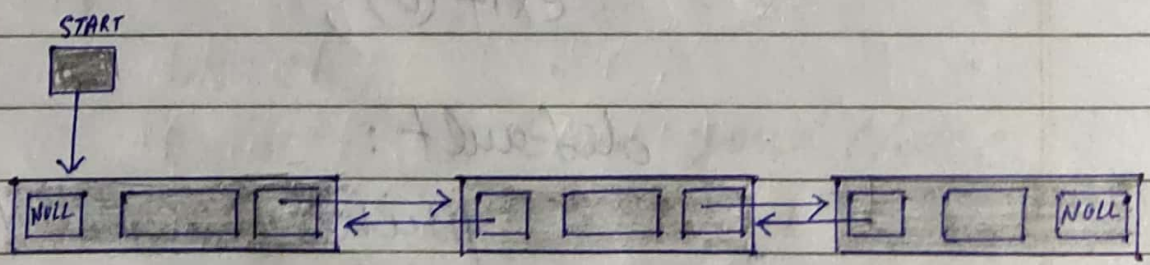
Good Write

Doubly linked list :—

- Doubly linked list contains one extra pointer as compare to the node of singly linked list.
- From any node we can traverse in both the directions.

Node

```
struct node
{
    int info;
    struct node *prev, *next
};
```



Operations

— Insertion
— Deletion
— Traversing

Good Write

DATE: ___/___/_____
PAGE: _____

Programming for doubly linked list :-

```
#include<stdio.h>

struct node *prev, *next;
{
   int info;
   struct node *prev, *next;
};

struct node* start = NULL;

Void insertAsFirstNode()
{
      struct node *n;
      n = (struct node *) malloc(sizeof struct node);
      printf("Enter a number");
      scanf("%d" &n->info);
      n-> prev = NULL;
      n-> next = NULL;
      if (start == NULL)
       start = n;
      else
      {
        start -> prev = n;
        n -> next = start;
        start = n;
      }
```

Good Write

```
void deleteFirstNode()
{
    struct node *r;
    if (start == NULL)
    printf("List is Empty")
    else
    {
        r = start;
        start = start->next;
        start->prev = NULL;
        free(r);
    }
}

void viewList()
{
    struct node *t;
    if (start == NULL)
    printf("List is Empty")
    else
    {
        t = start;
        while(t->next != NULL)
        {
            printf("%d", t->info);
            t = t->next
```

Good Write

DATE: ___/___/_____
PAGE: _____

Polish Notation :-

The method of writing operations of an expression either before their operands or after them is called the polish notation.

- Ways of writing :

1. Infix Notation
2. Prefix Notation
3. Postfix Notation

eg

A+B	Infix
+AB	Prefix
AB+	Postfix

Good Write

⇒ Infix $(A+B)*C$

Convert into prefix and postfix.

$(A+B)*C$

$(+AB)*C$

$*+ABC$ Prefix

$(A+B)*C$

$(AB+)*C$

$AB+C*$ Postfix

⇒ Infix ~~$(A+B)*C$~~ $A+(B*C)$

$A+(*BC)$

$+A*BC$ Prefix

$A+(BC*)$

$ABC*+$ Postfix

Good Write

DATE : ___/___/_____
PAGE : _____

⇒ Infix (A+B)/(C-D)

(+AB)/(-CD)

/+AB-CD Prefix

~~+A~~ (A+B)/(C-D)

(AB+)/(CD-)

AB+CD-/

⇒ A+[(B+C)+(D+E)*F]/G

A+((BC+)+(DE+)*F)/G

A+((BC+)+(DE+F*))/G

A+(BC+DE+F*+)/G

A+(BC+DE+F*+G/)

ABC+DE+F*+G/+

Good Write

DATE: ___/___/____
PAGE: _____

Algorithm to change infix into postfix

- Suppose Q is an arithmetic expression written in infix notation. This algorithm finds the equivalent postfix expression P.

1. Push '(' onto the stack and add ')' to the end of Q.

2. Scan Q from left to right and repeat step 3 to 6 for each element of Q until the stack is empty.

3. If an operand is encountered add it to P.

4. If a left parenthesis is encountered, push it onto stack.

5. If an operator is encountered, then:
   a. Repeatedly pop from stack and add to P each operator which has the same precedence as or higher precedence than ⊗.
   b. Add ⊗ to stack.

Good Write

6. If a right parenthesis is encountered, then,
   a. Repeadetly pop from stack and add to P each operator untill a left paranthesis is encountered.
   b. Remove the left parenthesis.

7. Exit.

Algorithm to evaluate the postfix expression.

- Let P be an arithmetic expression written in postfix notation. We use a stack to hold operands.

1. [Add sentinel] Add a right parenthesis ")" at the end of P.

2. [Loop] Scan P from left to right and repeate step 3 and step 4 for each element of P untill the sentinel ")" is encountered.

3. [Handling of operands] If an operand is encountered, put it on stack.

DATE: __/__/____
PAGE: _____

4. [Handling of operands] If an operator ⊗ is encountered then
   (a) Remove the two top elements of stack, where A is the top element and B is the next top element.
   (b) Evaluate B ⊗ A
   (c) Place the result of B(b) back on stack.
   [End of if structure]
[End of step 2 loop]

5. [Result] Set value equal to the top element on stack.

6. [Finished] Exit.

Good Write

DATE: __/__/____
PAGE: _____

Queue :-

Queue is a line of items waiting to be served in sequential order.

- In queue, the order in which the data arrives is important.

- A queue is an ordered list in which insertions are done at one end (rear) and deletions are done at other end (front)

- Working principle is First In First Out (FIFO).

Storage Structure

— Array / Dynamic Array / Linked List

Operations

— Insertion
— Deletion

Queue is an abstract data type which stores the data linearly.

Good Write

DATE: __/__/____
PAGE: _____

Operations

- createQueue ()
- isEmptyQueue ()
- isFullQueue ()
- queueSize ()
- inQueue ()
- deQueue ()
- deleteQueue ()

Structure

```
struct ArrayQueue
{
    int front, rear;
    int capacity;
    int *array;
};
```

Good Write

DATE: ___/___/_____
PAGE: _____

```
struct ArrayQueue* createQueue(int capacity)
{
    struct ArrayQueue* Q = malloc(sizeof(struct
                                    ArrayQueue));
    if(!Q)
    return(NULL);

    Q -> capacity = capacity;
    Q -> front = Q -> rear = -1;

    Q -> array = malloc(Q -> capacity * sizeof(int));
    if(!Q -> array)
    return (NULL)
    return(Q);
}

int isEmptyQueue(struct ArrayQueue *Q)
{
    return (Q -> front == -1)
}

int isFullQueue(struct ArrayQueue *Q)
{
    return ((Q -> rear + 1) % Q -> capacity == Q -> front);
}
```

Good Write

DATE: __/__/____
PAGE: _____

```
int QueueSize()
{
  return ( Q->capacity - Q->front + Q->rear
                          +1) % Q->capacity );
}

inQueue()

void inQueue (struct ArrayQueue *Q, int data)
{
  if (isFullQueue(Q))
    printf("Overflow");

  else
  {
    Q->rear = (Q->rear+1) % Q->capacity;
    Q->array[Q->rear] = data;

    if (Q->front == -1)
    Q->front = Q->rear;
  }
}
```

Good Write

```
int deQueue (struct ArrayQueue *q)
{
    int data = -1;

    if (isEmptyQueue(q))
    {printf("Underflow");
     return(-1);
    }

    else
    {
        data = q->array[q->front];
        if (q->front == q->rear)
            q->front = q->rear = -1;
        else
            q->front = (q->front+1) % q->capacity;
    }

    return data;

}
```

Good Write

DATE : ___/___/____
PAGE : _____

```
void deleteQueue (struct Arrayqueue *q)
{
  if (q)
  {
    if (q->array)
    {
      free (q->array)
    }
    free (q)
  }
}
```

Good Write

DATE: 26 01 2021
PAGE:

A

Tree :-

Tree is a hierarchical data structure.

- A tree is defined as one or more data item's finite set.
- There is a special node called the root of the tree.
- The remaining nodes are partioned into $n \geq 0$ disjoint subsets, each of which is itself a tree, and the are called sub-trees.

Terminologies ——

Degree :

The number of subtrees of a node is called its degree.

Leaf : A node with degree zero is called leaf.

DATE: __/__/____
PAGE: _____

Tominal Nodes : The leaf nodes are also called terminal nodes.

Degree of Tree : The degree of tree is maximum degree of the nodes in the tree.

Ancestors : The ancestors of a node are all the nodes along the path from the root to that node.

Descendants : The descendants of a node are all the nodes along the path from the node to terminal node

Level number : Each node is assigned a level number. The root node of the tree is assigned a level number 0. Every other node assigned a level number which is 1 more than the level number of its parent.

Generation : Nodes with the same level number are said to belong to the same generation.

Good Write

DATE: __/__/____
PAGE: _____

Height / Depth: The height or depth of a tree is the maximum number of nodes in a branch.

Edge: A line drawn from a node N of T to a successor is called ~~path~~. an edge.

Path: Sequence of consecutive edges is called path.

Branch: Path ending in a leaf is called a branch.

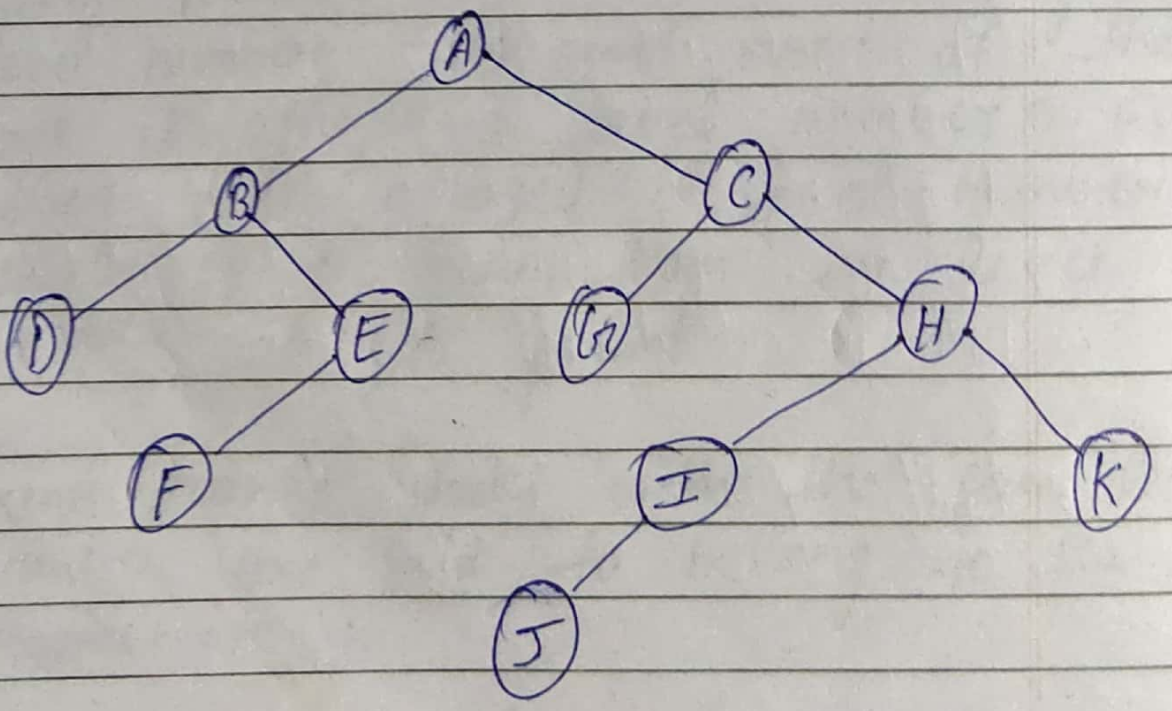
DATE : __/__/____
PAGE : _____

Binary tree :-

A binary tree T is defined as a finite set of elements, called nodes, such that

1. T is empty (called the null or empty tree), or

2. T contains a distinguished node R, called the root of T, and the remaining nodes of T form an ordered pair of disjoint binary trees T_1 and T_2.



Good Write

In the diagram,

- The root of T is the nood A at the top of the diagram

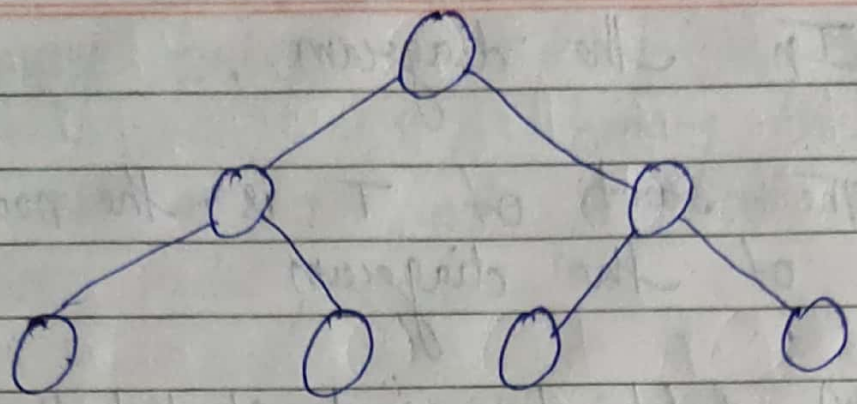
- A left downward slanted line from a nood N indicates a left successor of N, and a right downward slanted line from N indicates a right successor of N.

- The defination of binary tree is recursive since T is defined in terms of the binary subtrees T1 and T2. This means, in particular, that every node N of T contains a left and a right subtree.

Complete binary tree :-

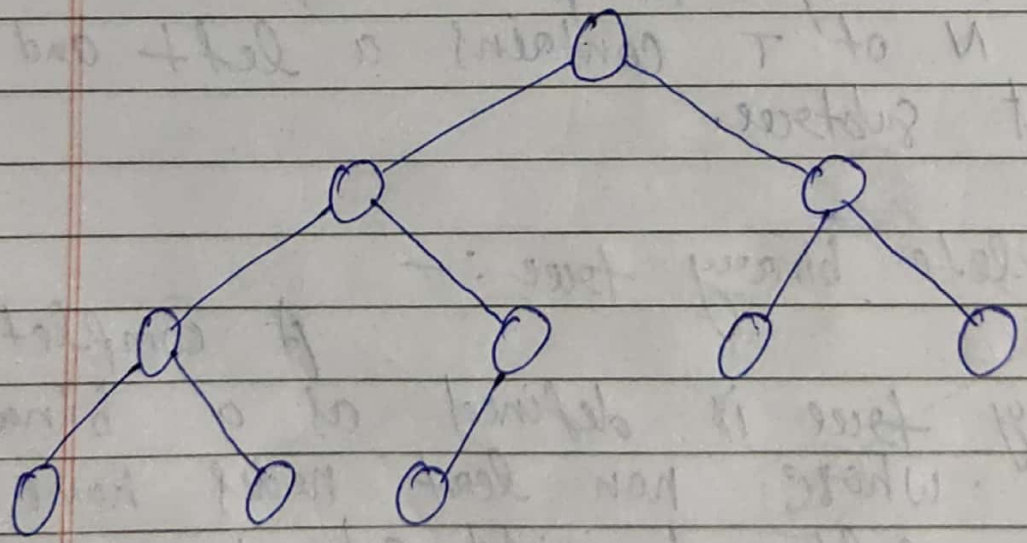
A complete binary tree is defined as a binary tree whose non leaf nodes have no empty left and right sub-tree and all leaves are at the same level.

Good Write



Almost complete binary tree :-

A almost complete binary tree is defined as a binary tree whose non leaf nodes have non empty left and right sub-tree and all leaves are either at the last level or second last level.



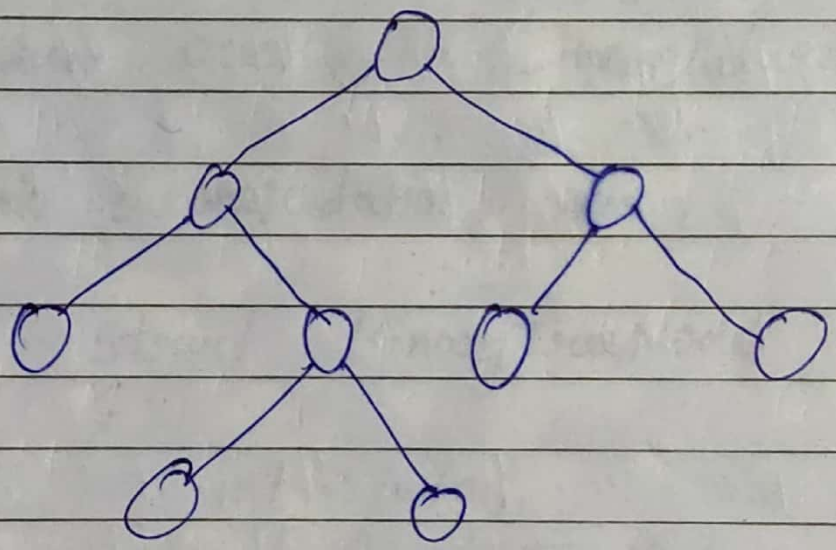
Good Write

DATE : __/__/____
PAGE : _____

Strict binary tree :-

A binary tree called a strict binary tree if every non leaf node in the binary tree has non empty left and right subtree.

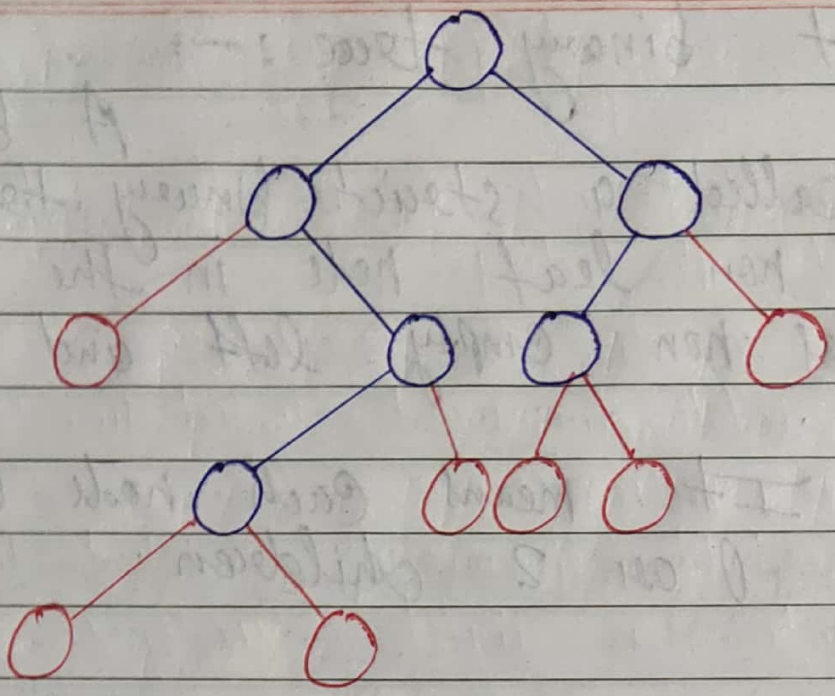
It means each node will have either 0 or 2 children.

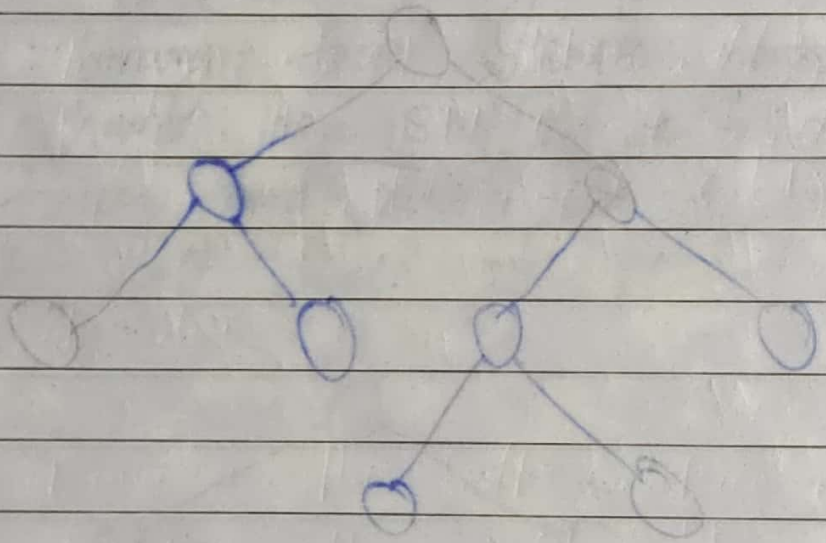


Extended Binary tree :-

An extended binary tree is a tree that has been transformed into a full binary tree. This transformation is achieved by inserting special "External" nodes such that every "internal" node has exactly two children.

DATE : ___/___/_____
PAGE : _____





Good Write

DATE: 28/01/2021
PAGE:

Representation of Binary tree :-

1. Array representation of binary tree
2. Linked list representation of binary tree.

Condition for any representation

- One should have direct access to root
- Any node N of T, one should have direct access to the children of N.

→ Link representation

```
struct BinaryTreeNode
{
    int info;
    struct BinaryTreeNode *left;
    struct BinaryTreeNode *right;
};
```

Good Write

DATE : ___/___/_____
PAGE : _____

Traversing Binary tree :-

- Preorder

— Process the root R
— Traverse left subtree of R in preorder
— Traverse the right subtree of R in "

- Inorder

— Traverse left subtree of R in inorder
— Process the root R
— Traverse the right subtree of R in "

- Postorder

— Traverse the left subtree of R in postorder
— Traverse the right subtree of R in "
— Process the root R.

Good Write

DATE : ___/___/_____
PAGE : _____

Preorder traversal :-

```
void preOrder (struct BinaryTreeNode *root)
{
    if (root)
    {
        printf ("%d", root -> info);
        preOrder (root -> left);
        preOrder (root -> right);
    }
}
```

Good Write

A inorder traversal :-

```
void inOrder (struct BinaryTreeNood *root)
{
    if (root)
    {
        inOrder (root -> left);
        printf ("%d", root -> info);
        inOrder (root -> right);
    }
}
```

Good Write

DATE : __/__/____
PAGE : _____

A Postorder traversal :-

```
void postOrder (struct BinaryTreeNode *root)
{
    if (root)
    {
        postOrder (root -> left);
        postOrder (root -> right);
        printf("%d", root -> info);
    }
}
```

Good Write

DATE: __/__/____
PAGE: _____

Level Order traversal :-

Algorithm

- Visit the root.
- While traversing level L, keep all the elements at level L+1 in queue.
- Go to next level and visit all the nodes at that level.
- Repeat this until all levels are completed.

DATE: __/__/____
PAGE: _____

levelOrder ()

```
void levelOrder(struct BinaryTreeNode *root)
{
    struct BinaryTreeNode *temp;
    struct Queue *Q = createQueue();
    if (!root)
    return;
    inQueue(Q, root);
    while (!isEmptyQueue(Q))
    {
        temp = deQueue(Q);
        printf("%d", temp->info);

        if (temp->left)
            inQueue(Q, temp->left);

        if (temp->right)
            inQueue(Q->right);
    }

    deleteQueue(Q);
}
```

Good Write

DATE: 29 01 2021
PAGE:

Binary Search tree :-

- In Binary search tree, all the left subtree elements should be less than root data and all the right subtree elements should be greater than root data. This is called binary search tree property.
- This property should be satisfied at every node in the tree.

Structure to represent BST :-

```
struct BSTNode
{
    int info;
    struct BSTNode *left;
    struct BSTNode *right;
};
```

Good Write

DATE: ___/___/_____
PAGE: _____

Elementary Operations

- Find
- Insertion
- Deletion

```
struct BSTNode* Find (struct BSTNode *root
                                , int data)
{
    if (root == NULL)
        return (NULL);

    if (data < root->data)
        return (Find(root->left, data));

    else if (data > root->data)
        return (Find (root->right, data));

    return (root);

}
```

Good Write

- Insertion

```
Void Insert(struct BSTNode *root, int data)
{
    struct BSTNode *par;
    struct BSTNode *n = malloc(size of
                        (struct BSTNode));

    n -> left = NULL;
    n -> data = data;
    n -> right = NULL;

    if (root == NULL)
    root = n;

    else
    {
        par = root;
        while(par != NULL)
        {
            if (par -> data > data)
            {
                if(par -> left == NULL)
                par -> left = n;
                par = par -> left;
```

Good Write

}

```
else if (par->data < data)
    {
      if (par->right == NULL)
        {
          if (par->ri
```

```
else if (par->data < data)
        {
          if (par->right == NULL)
            par->right = n;
            par = par->right;
        }
    } // end of while
} // else's end
}
```

Good Write

DATE: __/__/____
PAGE: _____

- Deletion

Deletion is simple if we are deleting a leaf node.

there will be 3 cases

(i) If the element to be deleted is a leaf node.

(ii) If the element to be deleted has one child.

(iii) If the element to be deleted has both children.

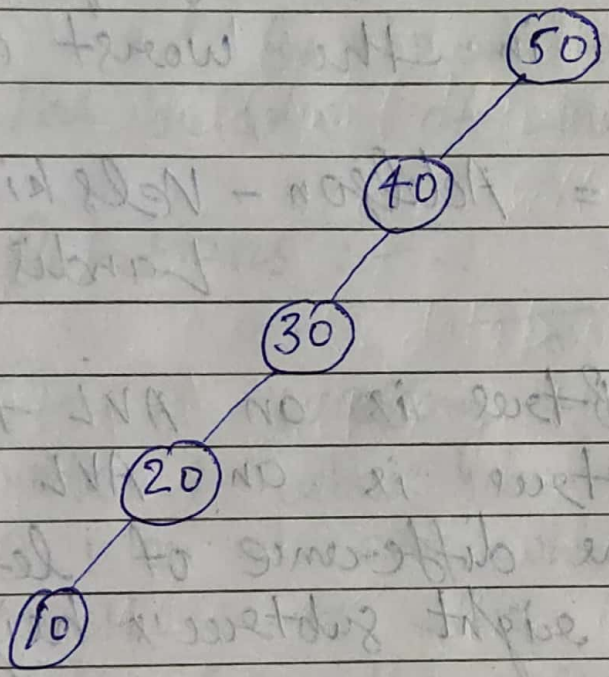
```
struct BSTNode *Delete(struct BSTNode
                       *root, int data)
{
    struct BSTNode *temp;
    if(root == NULL)
      printf("No such element exists");
    else if(data < root->data)
        root->left = Delete(root->left, data);

    else if(data > root->data)
        root->right = Delete(root->right, data);

    else // element found
    {
        if(root->left && root->right)

        { //both children

            temp = FindMax(root->left);
            root->data = temp->data;
            root->left = Delete(root->left, root->data);
        }
```

```
else
{ //one or none child
    temp = root;
    if (root->left == NULL)
        root = root->right;
    if (root->right == NULL)
        root = root->left;
    free(temp);
}
} // end of else
return (root);
} // end of function
```

DATE: 07 01 2021
PAGE:

Problem with Binary search tree :-

The disadvantage of a skewed binary search tree is that the worst case time complexity of a search is O(n).



Time complexity of worst case binary search tree is O(n)

To handle this type of problem we will study AVL Tree.

Good Write

DATE: __/__/____
PAGE: _____

AVL Tree :-

There is a need to maintain the binary search tree to be of balanced height, so that it is possible to obtained for the search option, a time complexity of $O(\log_2 n)$ in the worst case.

AVLTree = Adelson - Velskii and Landis

- An empty B-tree is an AVL tree.
- A binary tree is an AVL tree iff the difference of left subtree and right subtree's height is -1, 0, 1

i.e the Balance factor should be -1 or 0 or 1.

$$BF = |h(T^L) - h(T^R)| \leq 1$$

for AVL Tree, $BF \leq 1$

Good Write

DATE: / /
PAGE:

Insertion in AVL Tree :-

If the insertion of any element in AVL Tree take the balance factor out of the range then to render we use the technique called rotation. By rotation we can restore the balance of the search tree.

Rotations :-

To perform rotations it is necessary to identify a specific node A whose balanced factor is neither 0, 1 or -1 and which is the nearest ancestor to the inserted node on the path from the inserted node from the path from the inserted node to the root.

Types of rotations

(1) LL Rotation :-

Inserted node is in the left subtree of left subtree of node A.

(2) RR rotation :-

Inserted node is in the right subtree of right subtree of node A.

(3) LR rotation :-

Inserted node is in right subtree of left subtree of node A.

(4) RL rotation :-

Inserted node is in left subtree of right subtree of node A.

- Trick to solve

Case 1 : LL or RR rotations

⇒ Make A as child of B

Case 2 : LR or RL rotations

⇒ Make A and B child of C.

Good Write

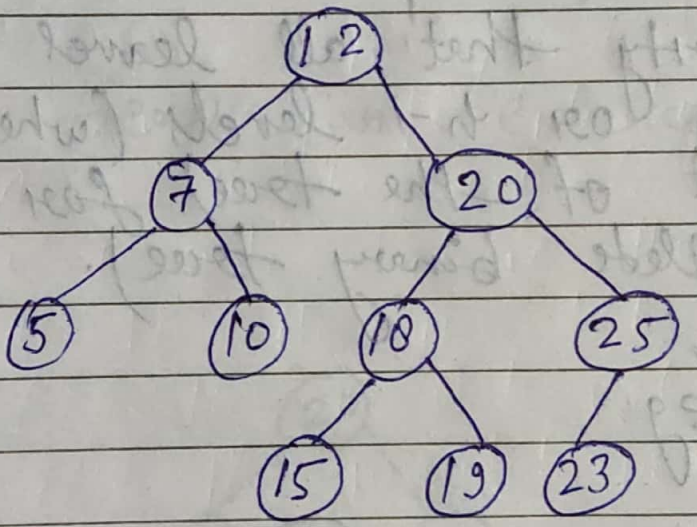
DATE : ___/___/_____
PAGE : _____

Que Generate AVL Tree for the values

5, 7, 19, 12, 10, 15, 18, 20

25, 23.

After having the rotations the final tree is



Good Write

DATE: 08/01/2021
PAGE:

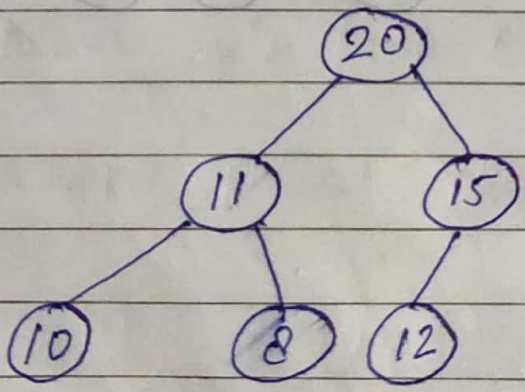
Heap :-

Heap is a tree with some special properties.

- The basic requirement of a heap is that the value of a node must be >= (or <=) to the values of its children.

- The heap has the additional property that all leaves should be at h or h-1 levels (where h is the height of the tree) for some h>0 (complete binary tree).

eg



Good Write

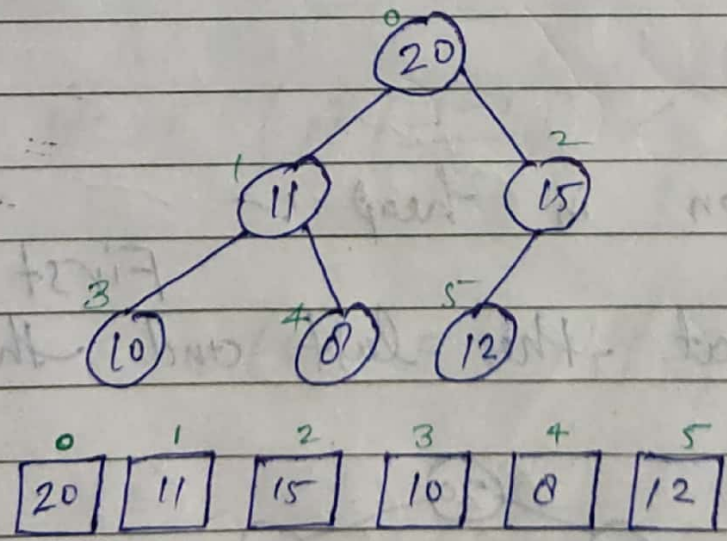
Heap are of two types

Max heap, where the value of parent node is greater than the child

Min heap, Value of child nodes are greater than the parents nodes.

* Representations of heap —

→ Heap can be represented using Array —



Good Write

if index of parent node is i then the index of childs nodes will be —

$$i_1, i_2 = 2i+1, 2i+2$$

if index of child nods are i_1 and i_2 then parents node will be

$$i = \frac{i_1 - 1}{2} \text{ or } \frac{i_2 - 1}{2} \text{ (Integer Value)}$$

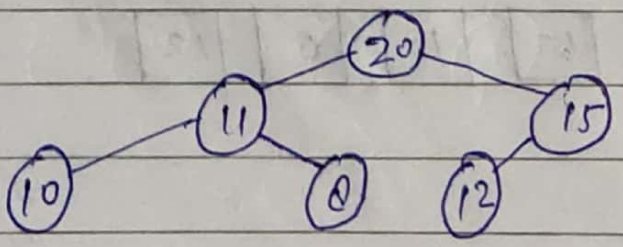
eg, 2.5 = 2

1.5 = 1

and so on

Insertion in heap :-

First add the value at the last and then heapify.



Good Write

DATE: 09/01/2021
PAGE:

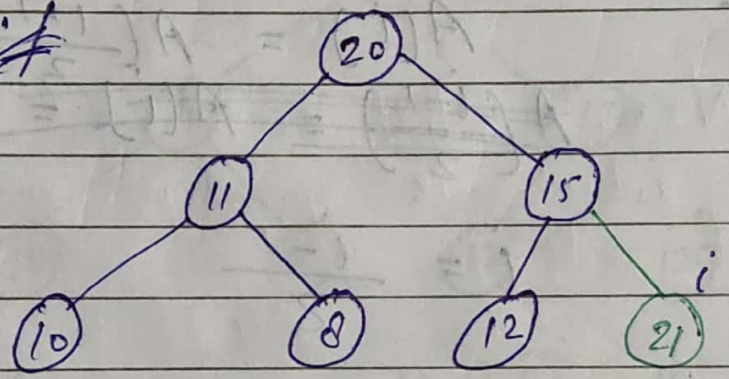
0	1	2	3	4	5	6
20	11	15	10	8	12	

Data = 21

i = 6

A[i] = Data

if



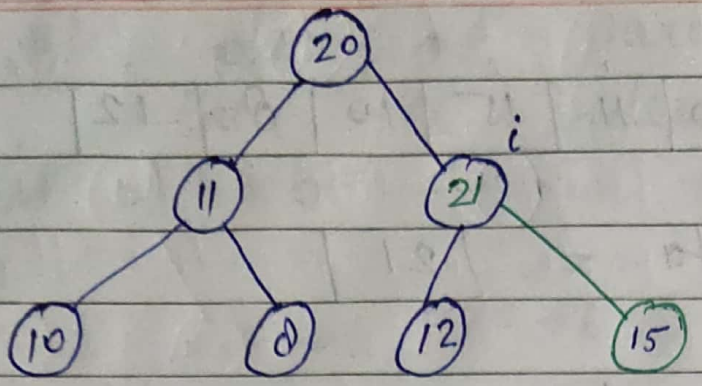
if $\left(A\left[\frac{i-1}{2}\right] < Data\right)$

{

$A(i) = A\left(\frac{i-1}{2}\right) = Data$

$i = \frac{i-1}{2}$

Good Write

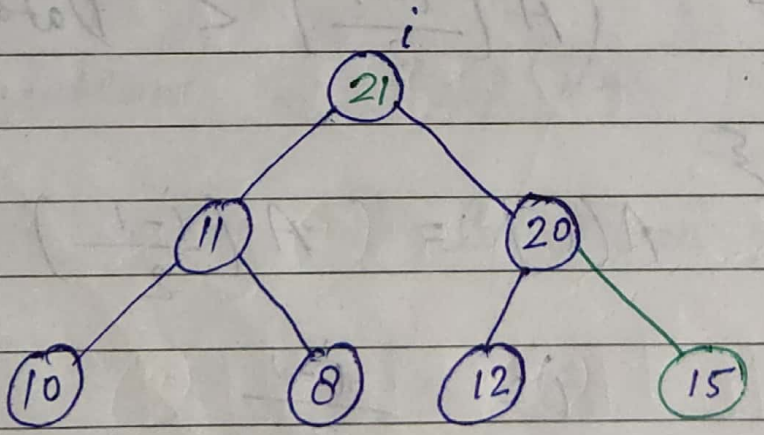


if ($A(\frac{i-1}{2})$ < Data)

{ $A(i) = A(\frac{i-1}{2}) = Data$

~~$A(\frac{i-1}{2}) = A(i) =$~~

$i = \frac{i-1}{2}$



}

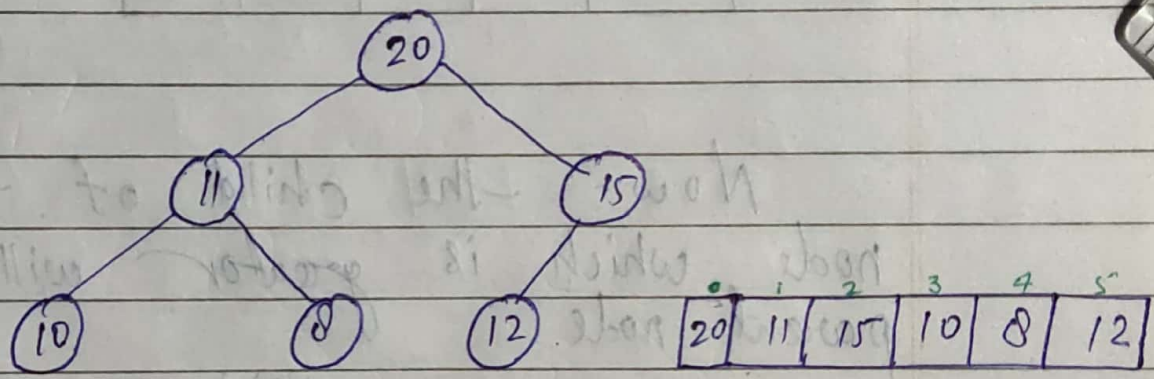
Good Write

DATE : __/__/____
PAGE : _____

Deletion in heap :-

We can delete the top most value.

First we will delete the top most value then we will heapify the heap.



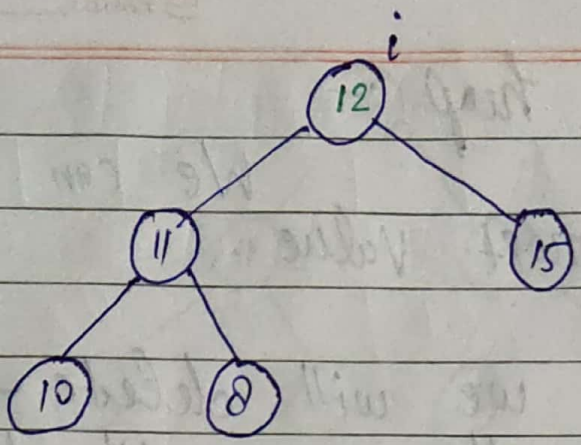
Step 1 = Delete the top most value.

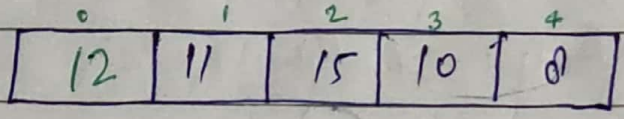
Step 2 = Assign last value to the top most node.

Step 3 = Heapify.

data = A(0)

A(0) = A(5)



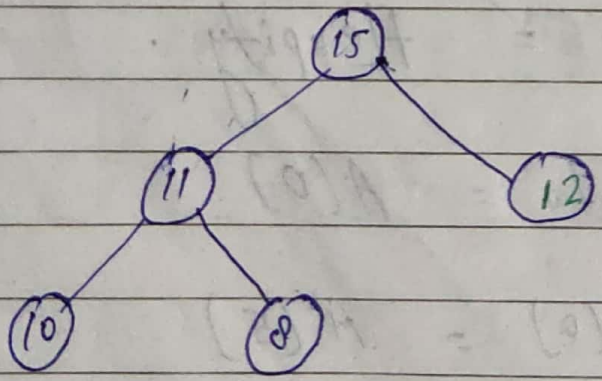


Now the child of top most node which is greator will be the parent node

$$data = A(i)$$

$$A(i) = A(2i+2) = data$$

$$i = 2i + 2$$



Good Write

DATE: __/__/____
PAGE: _____

Threaded Binary tree :-

A threaded binary tree is a tree in which we can do inorder traversal faster and we can do it without stack and any recursion.

There are two types of threaded binary trees

(1) Single threaded

(2) Double threaded

Single threaded : NULL right pointers are made to point to the inorder successor (if successor exist)

Double threaded : Both left and right NULL pointers are made to point to inorder predecessor and inorder successor respectively.

Good Write

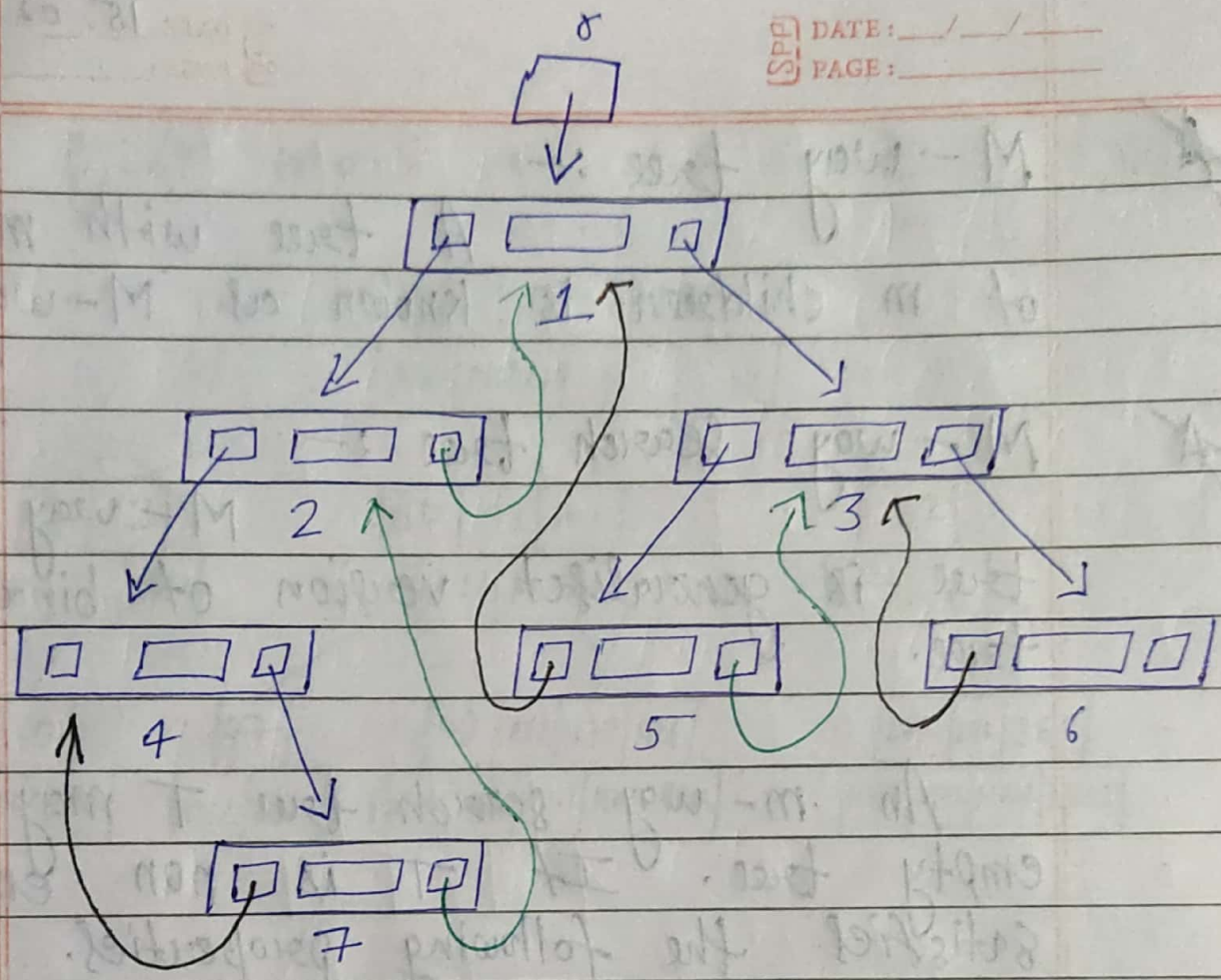
DATE: / /
PAGE:

Structure for threaded BT

```
struct Node
{
    int Key;
    struct Node *left, *right;
    bool isThreaded;
}
```

isThreaded is used to indicate whether the right pointer is a normal right pointer or a pointer to inorder successor.

Good Write



Inorder traversal

= 4 7 2 1 5 3 6

Good Write

DATE: 15 02 2021
PAGE:

M-way tree :-

A tree with maximum of m children is known as M-way tree.

M-way search tree :-

M-way search tree is generalised version of binary search tree.

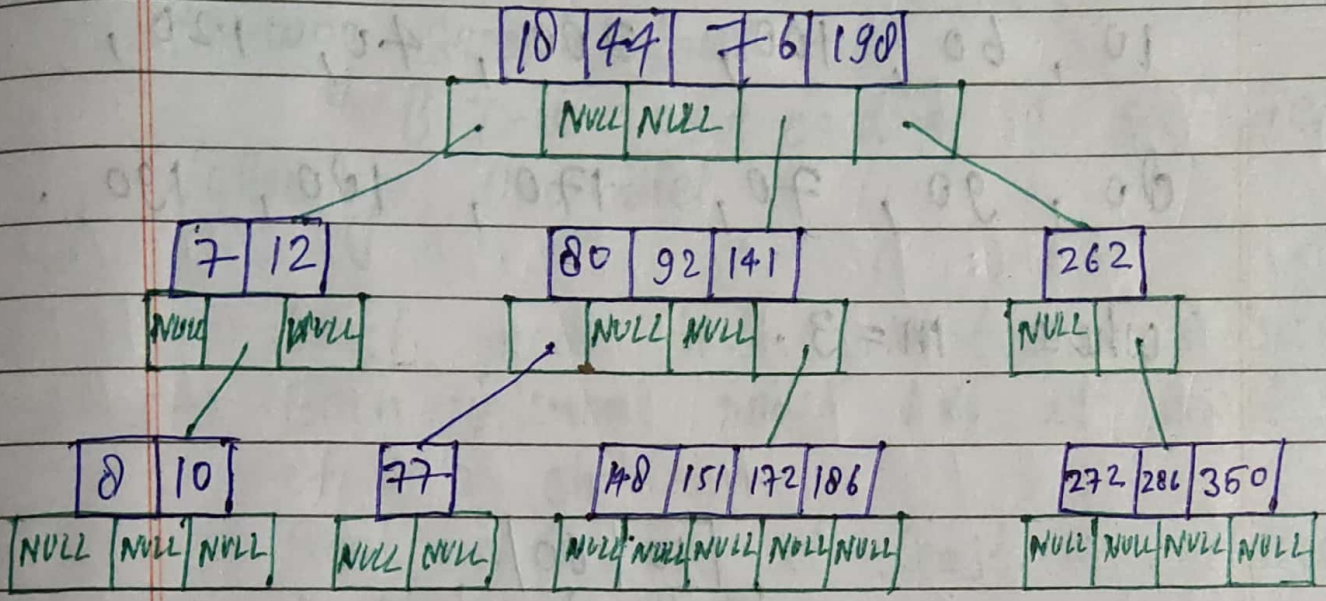
An m-way search tree T may be an empty tree. If T is non empty, it satisfies the following properties.

- Each node has, at most m child nodes.
- If a node has k child nodes where $k \leq m$ then the node can have only (k-1) keys $K_1, K_2, K_3, \ldots K_{k-1}$
- For a node $A_0, (K_1, A_1), (K_2, A_2) \ldots (K_{m-1}, A_{m-1})$ all key values in the subtree pointed to by A_i are less than key K_{i+1}, $0 \leq i \leq m-2$, and all key values in the subtree pointed to by A_{m-1} are greater than K_{m-1}
- Each of subtrees A_i, $0 \leq i \leq m-1$ are also m-way search trees.

Good Write

DATE : ___/___/_____
PAGE : _____

Eg, m = 5



* Operations —

(1) Searching in M-way search tree.

(2) Insertion in M-way search tree.

(3) Deletion in M-way search tree.

Good Write

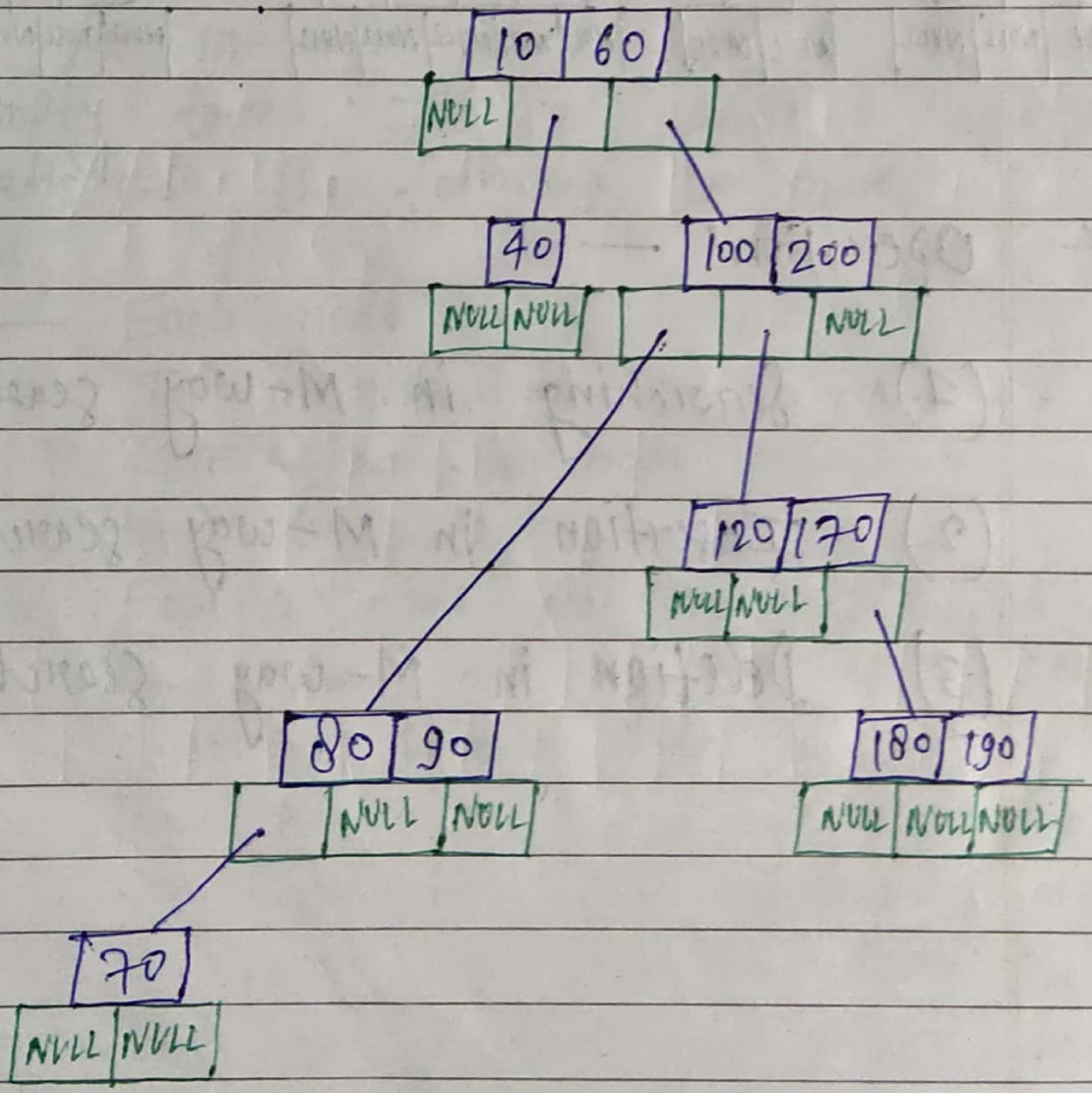
DATE: ___/___/_____
PAGE: _____

Que Generate M-way search tree.

10, 60, 100, 200, 40, 120,

80, 90, 70, 170, 180, 190.

where m = 3.



Good Write

DATE: / /
PAGE:

* B Tree :-

B Tree is a special case of M-way search tree.

B Tree of order M satisfies the following properties

- Each node has atmost m children.
- Each internal nodes has at least ceiling of M/2 children
- Root node has at least 2 children if it is not leaf.
- A non leaf node with K children has K-1 keys.
- All leaf nodes appear in the same level.

* Operations —

(1) Insertion

(2) Searching

(3) Deletion

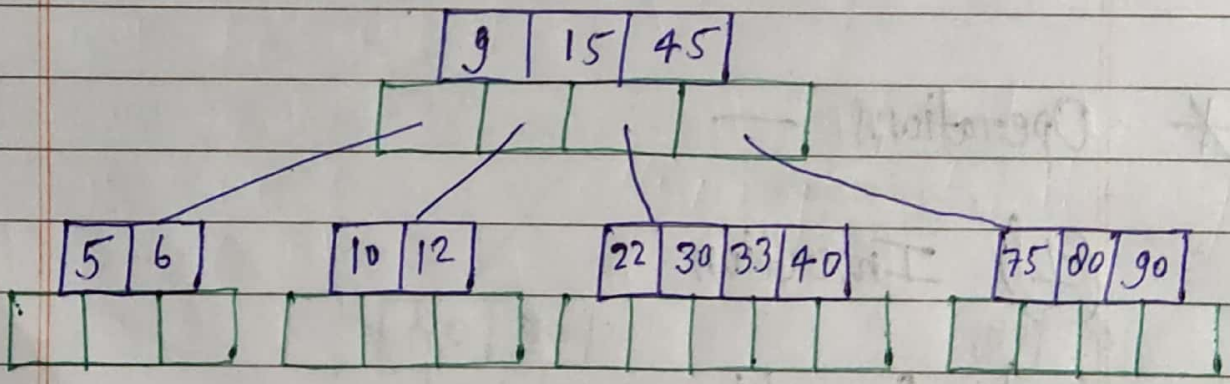
Good Write

Que 30, 45, 75, 10, 80, 40, 90, 6, 15, 22, 33, 9, 5, 12.

Generate BTree for m=5.

Answer Note: We do not open childs for B Tree instead when the node gets heavy it breaks into two parts and the value of middle goes up.

After insertion the final BTree would look like



Good Write

DATE: 16 02 2021
PAGE:

Linear search :-

To search a value in a list on by one.

Linear search is also known as sequential search.

In this search technique, we start at the begining of a list or table and search for the desired record by examining each subsequent record untill either the desired record is found or the list is exhausted.

Algorithm

Algorithm SEQSEARCH (L[], N, ITEM) : The algorithm find an ITEM in the list L of N elements.

Step 1 [Initialize]
Set Flag = 1.

Step 2 [Loop]

Repeat step 3 for K = 0, 1, 2, ..., N-1
(a) Set Flag = 0
(b) Write "search "Success"

Good Write

Step 4 if Flag == 1 then : Write "search unsuccessful".

Step 5 (Finished)

Exit.

Performance

- Number of comparisons made in searching a record in a search table determines the efficiency of the technique.
- On an average the number of comparisons will be $\frac{(n+1)}{2}$.
- The worst case efficiency of this algo. is $O(n)$

Good Write

```
void seq_search (int L(), int N, int ITEM)
{
    int Flag = 1, k;
    for (k = 0; k <= N; k++)

    if (L(k) == ITEM)
    { printf(" Search successful");
      Flag = 0
    }

    if (Flag == 1)

    printf(" Search Unsuccessful")
}

main()
{
    int ITEM
    int A(5) = {11, 22, 33, 44, 55}
    seq_search (A, 4, ITEM)
    printf(" Enter the number for searching
                                    \n");
    scanf("%d", &ITEM);
    seq_search (A, 4, ITEM);
    getch();
}
```

Good Write

DATE: ___/___/____
PAGE: _____

Binary Search :-

To search by checking the middle term in the sorted list is known as binary search

Time Complexity = $O(\log_2 n)$

If the value is high then the middle term, we check the former half and vise versa.

The procedure is continued till the desired key is found or the search interval becomes empty.

Good Write

DATE: __/__/____
PAGE: _____

Algorithm

Algorithm BINSEARCH (L[], n, item): The algorithm find an item in the sorted list L of n elements.

Step 1 [Initialize]

Set $l = 0$, Set $u = N-1$

Step 2 [Loop]

Repeat step 3 and 4 while $u \geq l$

Step 3 [get the mid point]

Set $m = \frac{l+u}{2}$

Step 4 if item == L(m) then:
write " successful, item found"

else if item > L(m) then:
set $l = m+1$

else
set $u = m-1$

[End of Loop]

Step 5 [Finished] Exit.

Good Write

```
void bin_search (int L[], int n, int item);

main ()
{
   int A[] = { 11, 22, 33, 44, 55, 66,
               77, 88, 99, 110 }

   int item;

   printf(" Enter the value for searching");
   scanf("%d", &item);
   void bin_search (A, 10, item);
}

void bin_search (int L[], int n, int item)
{
    int l, u = n-1 ;

    while (l ≤ u)
    {
        m = (l+u)/2 ;

        if (item == L(m))
        {
            printf(" Search successful
                    at index %d, m);
```

Good Write

```
        return;
    }
    else if (item > L(m))
        l = m+1;
    else
        u = m-1;
}
printf("Search Unsuccessful");
}
```

DATE: 18 02 2021
PAGE:

Time complexities of searchings :-

(1) Linked list

$O(n)$

(2) Array (unsorted)

$O(n)$

(3) Array (sorted) [Binary search]

$O(\log_2 n)$

(4) Binary search tree

$O(\log_2 n)$

(5) Hash Table

$O(1)$

Good Write

DATE: __/__/____
PAGE: _____

Hashing :-

Hashing is a technique to convert a range of key values into a range of indexes of an array.

In case when keys are large and can't be directly used as index to store value, we can use technique of hashing.

In hashing, large keys are converted into small ones by using hash functions and then the values are stored in data structure called hash tables.

Hash table :-

Mostly it is an array to store dataset. It is the data structure.

Hash function :-

A hash function is any function that can be used to map dataset of ~~attributes~~ arbitrary size to dataset of fixed size which falls

Good Write

.... Next Page

into the hash table.

The values returned by a hash function are called hash values, hash codes, hash sums, or simply hashes.

Parameters of a good hash function

Easy to compute
Even distribution
~~Minimize~~ Minimize collisions

Perfect hash function, Perfect hashing maps each valid input to a different hash value (no collision)

eqs of hash functions

Ex 1

```
int hash (int x)
{
    return (x%10)
}
```

Good Write

DATE: __/__/____
PAGE: _____

Ex 2

We are given keys in the range 0 to 999, and have a hash table of size 10 (even distribution)

0 - 99	Slot 0
100 - 199	Slot 1
200 - 299	slot 2
300 - 399	slot 3
400 - 499	slot 4
500 - 599	slot 5
600 - 699	slot 6
700 - 799	slot 7
800 - 899	slot 8
900 - 999	slot 9

Ex 2

A good hash function to use with integer key values is the mid-square method.

The mid square method squares the key value, and then takes out the middle r bits of the result, giving a value in the range 0 to $2^r - 1$.

Good Write

DATE: ___/___/____
PAGE: _____

Ex 4

```
int hash (char s[], int m)
{
    int i, sum = 0;
    for (i=0; s[i]; i++)
        sum = sum + s[i]
    return (sum % m)
}
```

Where m is the number of slots

DATE: 19/02/2021
PAGE:

Collision :-

A situation when the resultant hashesh for two or more data elements in the data set, maps the same location in the hash table, is called a hash collision.

In such a situation two or more data elements would qualify to be stored/mapped to the same location in the hash table.

Collision resolution :-

Two types of collision resolutions are there

(1) Open hashing (chaining)

(2) Closed hashing (open addressing)

Open hashing, Collisions are stored outside the table

Closed hashing, Collisions stored into the same table.

Good Write

Closed hashing types

(i) Linear Probing

(ii) Quadratic Probing

(iii) Double hashing

Sorting :-

Sorting is the process of arranging the data in some logical order.

This logical order may be ascending or descending in case of numeric values or dictionary order in case of alphanumeric values.

Types

Internal sorting
External sorting

⇒ Unless untill explicitly stated, sorting means ascending order.

Good Write

DATE: __/__/____
PAGE: _____

Sorting techniques

(1) Bubble sort :-

Bubble sort is very simple and easy to implement sorting technique.

Logic

Compare first two elements and if the left element is greater then the right element, they swap their position.

Comparision proceeds till the end of the array, and for n-1 elements of that array.

Algorithm

Bubble-sort (A, n) : A is array of values and n is the number of elements.

Step1 Repeat for round = 1, 2, 3, ..., n-1

Step2 Repeat for i = 0, 1, 2, ..., n-1 round

Good Write

Step3 if A(i) > A(i+1) then swap A(i) and A(i+1)

Step4 Return

Program

```
int main()
{
    int A[] = {26, 25, 3, 7, 15, 16}
    int i;
    bubble_sort (A, 6)
    printf
    for (i=0; i<=5; i++)
    printf("%d", A(i))
    getch();
}

void bubble_sort (int A[], int n)
{
    int round, i, temp;
    for (round = 1; round <= n-1, round++)
    for (i=0; i<=n-1, i++)
```

Good Write

DATE : __/__/____
PAGE : _____

```
{
    if ( A(i) > A(i+1))
    {
        temp = A(i);
        A(i) = A(i+1);
        A(i+1) = temp;
    }
}
```

Good Write

DATE: __/__/____
PAGE: _____

Modified Bubble Sort :-

In modified bubble sort we can reduce number of comparisions if the list is sorted.

Algorithm

M.Bubble_sort (A, N): A is array of values and N is the number of elements.

Step 1 Repeat step 2, 3, 4 for round = 1, 2, 3, 4, ..., n-1

Step 2 flag = 0

Step 3 Repeat for i = 0, 1, 2, ... n-1 round

if A[i] > A(i+1) the swap A(i) and A(i+1), also set flag = 1

Step 4 if flag == 0 return

Step 5 Return

Good Write

```
int main()
{
    int A[] = { 25, 26, 28, 29, 24 }
    int i;
    M-bubble-sort (A, 5);
    for (i=0; i<=4; i++)
    printf("%d", A[i]);
    getch();
}

void M-bubble-sort (A[], int n)
{
    int round, i, temp, flag;
    for (round=1; round<=n-1; round++)
    {
        flag = 0;
        for (i=0; i<=n-1-round; i++)
            if (A[i] > A[i-1])
            {
                flag = 1;
                temp = A[i];
                A[i] = A[i+1]
                A[i+1] = temp;
            }
```

Good Write

```
if (flag == 0)
{
    printf("Round = %d \n", round);
    return;
}
}
```

(2) Selection Sort :-

Selection sort is the sorting technique in which we select the smallest value in the list.

Working rule

→ Select the smallest value in the list.
→ Swap smallest value with the first value of the list.
→ Again select the smallest value in the list (exclude first value)
→ Swap this value with the second element of the list.
→ Keep doing (n-1) times to place all n values in the sorted order.

Good Write

DATE: 21 02 2021
PAGE:

~~Algorithm~~ Procedure

Procedure Min (A, K, N): An array A is in memory. This procedure finds the location LOC of the smallest element among A(K), A(K+1), A(K+2), ... A(N-1).

Step 1 [Initializes pointers]
Set MIN = A(K) and LOC = K

Step 2 Repeat for J = K+1, K+2, ..., N-1;

if MIN > A(J), then set min = A(J) and LOC = J.
[End of Loop]

Step 3 Return LOC

Good Write

DATE: __/__/____
PAGE: _____

Algorithm

Algorithm Selection (A, N) : This algorithm sorts the array A with N elements

Step 1 [Loop]

Repeat Step 2 and 3 for K = 0, 1, 2, ... N-2 :

Step 3 Call LOC = MIN(A, K, N)

Step 3 [Interchange A(K) and A[LOC].]

Set TEMP = A(K), A(K) = A[LOC] and A[LOC] = TEMP

[END of the Step 1 loop]

Step 4 Exit.

Good Write

```
int min(int A[], int k, int n)
{
    int j, LOC, MIN;
    MIN = A[K]
    LOC = K
    for (j = k+1; j <= n-1; j++)
    {
        if (MIN > A[j])
        {
            MIN = A[j]
            LOC = (j)
        }
    return (LOC);
}

int main ()
{
    int A[] = { 33, 11, 66, 60, 99, 77, 44}
    int LOC, k, temp;
    for (k = 0; k <= 5; k++)
    {
        LOC = min (A, K, 6)
        temp = A(k)
        A(K) = A(LOC)
        A(LOC) = temp;
}
```

Good Write

```
for (k=0 ; k<=7 ; k++)
    printf("%d", A(k));
getch();
}
```

Good Write

DATE: 25/02/2021
PAGE:

Insertion sort :-

Insertion sort is an sorting technique in which we sort the given list.

Algorithm

INSERTION-SORT (A, N): A is an array with N elements.

Step 1 $i = 1$

Step 2 Repeat step 3 to 5 while $i < n$

Step 3 temp = A[i] , $j = i - 1$

Step 4 Repeat while $j >= 0$ and temp < A[j]

$A[j+1] = A[j]$ and $j = j - 1$

Step 5 $A[j+1]$ = temp , $i = i + 1$

Step 6 Exit.

Good Write

```
void insertion_sort (int [], int);

int main ()
{   int i;
    int A[] = {17, 21, 34, 5, 9, 6, 8, 9, 11};
    insertion_sort (A, 9);
    printf
     for (i=0; i<=8; i++)
     printf(" %d", A(i))
     getch();
}

void insertion_sort (int A[], int n)
{
    int i, j, temp;

    for (i=1; i<n; i++)
    {
        temp = A(i);
        for (j = i-1; j>=0 && temp < A[j]
                                ; j--)
            A[j+1] = A[j]

        A[j+1] = temp;
    }
}
```

Good Write

DATE: __/__/____
PAGE: _____

Quick Sort :-

Quick sort is an algorithm of devide and conquer type. This is, the problem of sorting a set is reduced to the problem of sorting two smaller sets.

Working logic

- Procedure Quick
- Algorithm Quick-Sort

- Procedure Quick

Quick (A, n, BEG, END, LOC): Here A is an array with n elements. Parameters BEG and END contain the boundry values of sublist of A to which this procedure applies. LOC keeps track of the position of the first element.

Step 1 Set LEFT = BEG, RIGHT = END and LOC = BEG

Step 2 Scan from Right to left

(a) Repeat while A[LOC] ≤ A[RIGHT] and LOC ≠ RIGHT:
RIGHT = RIGHT - 1

Good Write

(b) if LOC == RIGHT then return

(c) if A[LOC] > A[RIGHT] then swap A[LOC] and A[RIGHT].
also set LOC = RIGHT and Goto Step 3

Step3 Scan from left to right.

(a) Repeat while A[LEFT] ≤ A[LOC] and LEFT ≠ LOC:
LEFT = LEFT + 1

(b) if LOC == LEFT then return

(c) if A[LEFT] > A[LOC] then swap A[LOC] and A[LEFT]
also set LOC = LEFT and Goto Step 2.

DATE: __/__/____
PAGE: _____

• Algorithm

Algorithm Quicksort: This algorithm sorts an array A with N elements.

Step 1 TOP = 1

Step 2 if N > 1 then TOP = TOP + 1, LOWER(TOP) = 0
UPPER(TOP) = N - 1

Step 3 Repeat step 4 to 7 while TOP ≠ -1

Step 4 & POP sublist from stack, set BEG = LOWER(TOP)
END = UPPER(TOP), TOP = TOP - 1

Step 5 Call quick (A, N, BEG, END, LOC)

Step 6 Push the left onto the stack,

if (BEG < LOC - 1) then: TOP = TOP + 1
LOWER(TOP) = BEG, UPPER(TOP) = LOC - 1

Step 7 Push right sublist onto stacks

if (LOC + 1 < END), then: TOP = TOP + 1
LOWER(TOP) = LOC + 1, UPPER(TOP) = EN

Good Write

Step 8 Exit.

DATE: 26/02/2021
PAGE:

```
void quick ( int A[], int N, int BEG, int END,
             int *LOCPTR)
{
  int LEFT, RIGHT, temp;
  LEFT = BEG; RIGHT = END;
  *LOCPTR = BEG;

Step 2;

  while (A[*LOCPTR] <= A[RIGHT]
         && *LOCPTR != RIGHT)
    RIGHT--;

  if (*LOCPTR == RIGHT)
    return

  if (A[*LOCPTR] > A[RIGHT])
  {
    temp = A[*LOCPTR];
    A[RIGHT] A[*LOCPTR] = A[RIGHT];
    A[RIGHT] = temp;
    *LOCPTR = RIGHT;
  }
  goto step 3;
```

Good Write

DATE: __/__/____
PAGE: _____

Step 3;

```
while (A[LEFT] <= A[*LOCPTR]
        && LEFT != *LOCPTR)

    LEFT++;

if (*LOCPTR == LEFT)
  return

if ( A[LEFT] > A[*LOCPTR])
{
    temp = A(LEFT)
    A[LEFT] = A(*LOCPTR)
    A[*LOCPTR) = temp ;
    *LOCPTR = LEFT;
}

goto step2;
}
```

```
void quick_sort (int A[] , int N)
{
    int BEG, END, LOC, TOP=-1;
    int LOWER[10], UPPER[10];
```

Good Write

```
if ( N>1 )
{
    TOP ++ ;
    LOWER (TOP) = 0 ;
    UPPER (TOP) = N-1 ;
}

while ( TOP != -1 )
 {
        BEG = LOWER (TOP) ;
        END = UPPER (TOP) ;
        TOP - - ;
        quick (A, N, BEG, END, &LOC) ;
 }
        if ( BEG < LOC - 1)
 if     {
            TOP ++ ;
            LOWER (TOP] = BEG ;
            UPPER (TOP] = LOC - 1
        }

        if ( LOC + 1 < END)
        {
            TOP++ ;
            LOWER (TOP] = LOC+1 ;
            UPPER (TOP) = END ;
        }
```

DATE: ___/___/____
PAGE: _____

```
    }
}

int main()
{
    int A[] = { 44, 33, 11, 55, 77, 90, 40, 60,
                99, 22, 88, 66 };
    int i;
    quick-sort (A, 12)
    for (i=0; i<=11; i++)
        printf(" %d, A[i]);

    getch()
}
```

Good Write

DATE: 07 03 2021
PAGE:

Heap sort :-

One main application of Heap ADT is sorting, called heap sort.

Logic

- Insert all the elements of an unsorted array into the heap.
- Remove the maximum element (root node element) from the heap and exchange it with the last value of array.
- Heapify.
- Repeat the process untill heap remains with single element.

Algorithm

Algorithm HeapSort (A, N) : An array A with N elements is given. This algorithm sorts the elements of A.

Step 1 [Build a heap H]

h = CreateHeap (N); BuildHeap (h, A, N);

Good Write

DATE: __/__/____
PAGE: _____

Step 2 Initialize

old size = h→ Count;

Step 3 [Sort A by repeatedly deleting the root of H]

for (i = n-1; i > 0; i--)

(a) temp = h→ array [0]
h→ array [0] = h→ array [h→ Count - 1]
h→ array [h→ count - 1] = temp;

(b) h→ count --

(c) percolateDown (h, i)

Step 4 h→ Count = old size;

Step 5 [Finished]
Exit.

Good Write

DATE: __/__/____
PAGE: _____

* Create heap ()

```
struct Heap *CreateHeap (int capacity)
{
  struct Heap *h = (struct Heap*) malloc
                   (sizeof(struct Heap));
  if (h == NULL)
  {
     printf("Memory Error");
     return;
  }

h->count = 0
h->capacity = capacity
h->array = (int*)malloc(sizeof(int)*h->capacity);

if (h->array == NULL)
{
   printf("Memory Error");
   return
}

return (h);
}
```

Good Write

DATE: __/__/____
PAGE: _____

* PercolateDown()

```
void PercolateDown ( struct Heap *h, int i )
{
    int l, r, max, temp;

    l = LeftChild (h, i);
    r = RightChild (h, i);

    if ( l != -1 && h->array(i) > h->array[l])
        max = l

    else
        max = i;

    if ( r != -1 && h->array (r) > array (max])
        max = r;

    if ( max != i)
    {
        temp = h->array (i);
        h->array(i) = h->array (max]
        h->array [max] = temp;
    }
```

Good Write

```
    percolateDown (h, max);
}

int LeftChild (struct Heap *h, int i)
{
    int left = 2*i + 1;
    if (left >= h->count)
        return (-1);
    return (left);
}

int RightChild (struct Heap *h, int i)
{
    int right = 2*i + 1
    if (right >= h->count)
        return (-1)
    return (right);
}
```

Good Write

DATE: __/__/____
PAGE: _____

* Build Heap ()

```
Void BuildHeap (Struct Heap *h, int A[],
                                int n)
{
    int i;
    if (h == NULL)
        return;
    while (n > h -> capacity)
        Resize Heap (h);
    for (i = 0; i < n; i++)
        h -> array [i] = A[i];
    h -> count = n
    for (i = (n-1)/2; i >= 0; i--)
        Percolate Down (h, i);
}
```

Good Write

DATE: __/__/____
PAGE: _____

* Resize Heap ()

```
void ResizeHeap (struct Heap *h)
{
    int i;
    int *arrayold = h->array;
    h->array = (int*) malloc (sizeof
                (int) * h->capacity*2);
    if (h->array = NULL)
    {
        printf (" Memory Error");
        return;
    }
  for (i=0; i<h->capacity ; i++)
        h->array (i) = array old (i);
    h->capacity *= 2;
    free (array old)
}
```

Good Write

DATE: __/__/____
PAGE: _____

* Heap Sort ()

```
Void HeapSort ( int A[], int n)
{
    struct Heap *h = (reate.Heap (n);
    int, old_size, i, temp ;
    Build Heap (h, A, n);
    old_size = h -> count;
    for (i = n-1 ; i > 0 ; i--)
    {
        temp = h -> array (0);
        h -> array (0) = h -> array
                          ( h->count - 1)
        h -> array (0) = temp;
        h -> count--;
        percolateDown (h, i)
    }
    h -> count = old_size;
}
```

Good Write