Intro, BT(~~sear~~ insert, delete, recursive, traversal on BT

Trees

Introduction

→ Tree is natural representation of hierarchical information

hierarchical → nature of hierarchy
arranged in order of rank.

→ Trees are used to represent genelogical information (e.g. - family tree, evolutionary tree)

→ directory structure of a file system of computer.

→ Structure of knock out tournament of any sport.

→ Deway decimal notation, which is used to classify books in library.

Tree is also used to design fast algorithm in computer science because of its efficiency related to simple data structure.

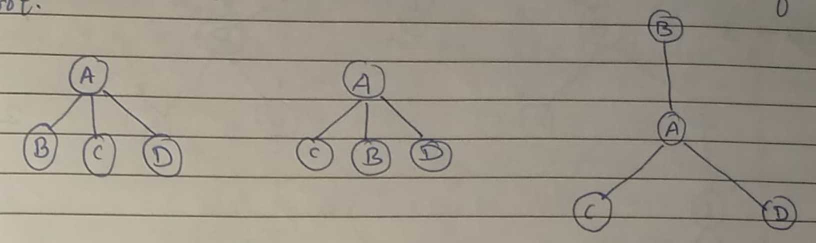
Kinds of tree in Data Structure.

1. Free or unrooted tree:
this is defined as graph such that there exist a unique path between to vertices in the graph.

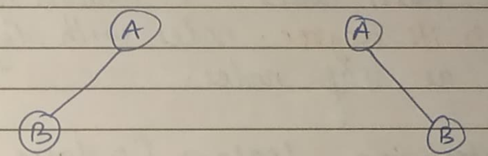
2. Rooted Tree: this is finite set of one or more node such that,

threaded binary tree, height balanced tree AVL

* there is special node called the root.
* the remaining nodes are known as subtree of root.



these three tree shown are distinct if they are viewed as rooted, ordered tree. The first two are identical if viewed as oriented tree. All three are identical if viewed as free tree.



Different binary trees.

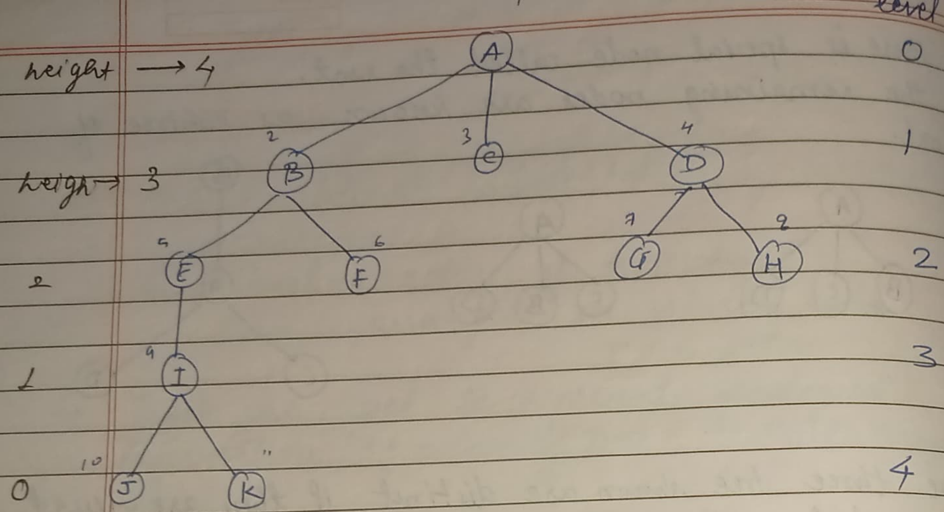


figure 3.3

an example tree.

above tree has 11 nodes. The number of subtree of a node is its degree. nodes with degree 0 are called as leaf nodes.

we can determine. leafs → (0 degree)
J, K, F, C, G, H

A has three degree
B, D, I have two degree.

8.2 tree representation

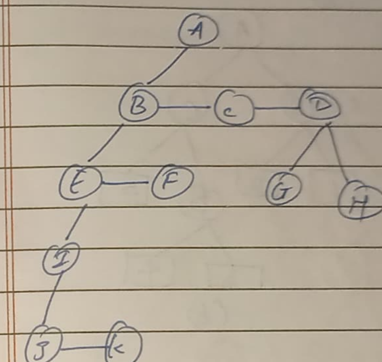
8.2.1 the tree of figure 3.3 can be written in generalized from as (A(B(E(I(J,K)),F, C, D(G,H))). The information in the root node come first followed by a list of subtree.

3.2.2. left Child - Right sibling representation.

3.4a shows the node structure used in this representation. Each node has a pointer to its left most child (if any) and to the siblings on its immediate right (if any)

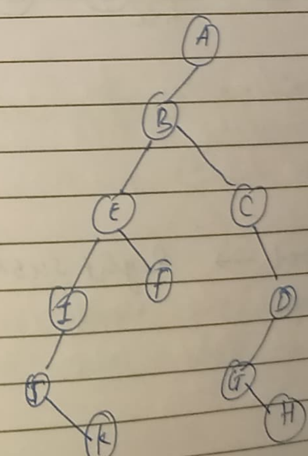
data	
left child	Right sibling

figure 3.3 can represented as 3.4b



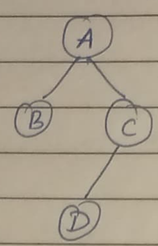
3.4 (b)

3.2.3 Binary tree Representation

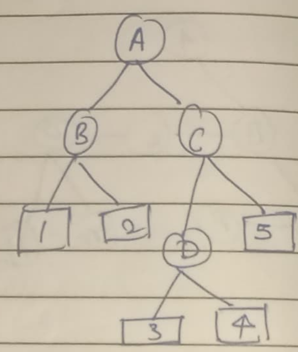


3.3 Binary tree and properties.

Binary tree were defined already. Binary tree is sometime extended by adding external nodes. External nodes are imaginary nodes that are added wherever an empty subtree was present in original tree. The original tree nodes are known as internal nodes. fig 3.5(a) shows a binary tree, and b corresponding extended tree.



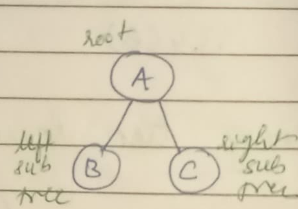
(a)



(b)

3.3.1. Properties :-

tree traversal.
- → inorder
- → preorder
- → post order



inorder traversal

left sub tree → root → Right subtree
B → A → C

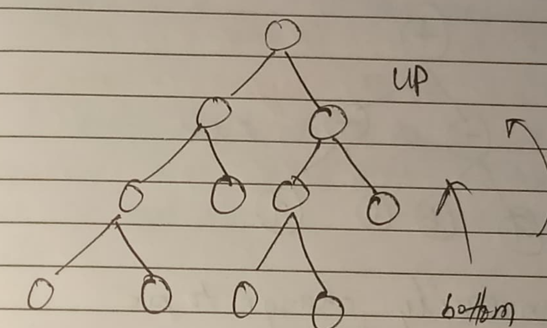
Post order

left sub tree → Right sub tree → Root
B → C – A

Pre order

root → left sub tree → Right Subtree
A → B → C

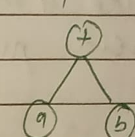
Whenever we have to change from traversal to traversal go bottom to up approach



now when we talk about the expressions

a + b → operator always on roots

operands alaway on leaf

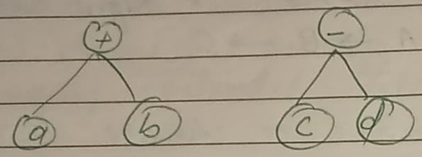


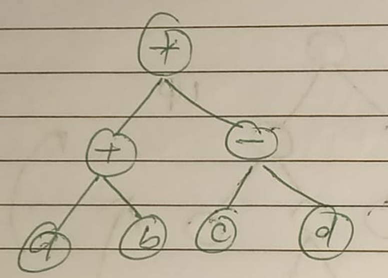
when we need to change order for expression. first create a tree. let see an example

$(a+b) * (c-d)$

→ ~~a~~ always take operands at first in bet which will be in same sequence

→ now check operators between them
→ and then see operator which make them combined





now we can easily change to pre and post orders / let see
in pre order we know we have to take root - LST - RST always start from root.
here first root is * and LST + and RST −

but here + is also root for a and b
∴ we have to treat it like a tree and apply same rule so here pre order is

+ a b ——— ①

and if we go RST for root * we get − which also act like tree for c and d

*

so pre order is
− c d ——— ②

now for * ① and ② will act as LST RST

therefore result for pre order expression is
* + a b − c d
⇒ *+ab−cd (same sequence)

same process for post order.

⇒ ab+cd−* (same sequence).

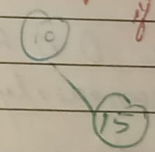
3.4 Binary Search Tree.

a binary search tree has a special kind of sequence that its left subtree is smaller than its root always and right subtree is always greater.

insertion in binary search tree
let we assume that we have array of element
array element

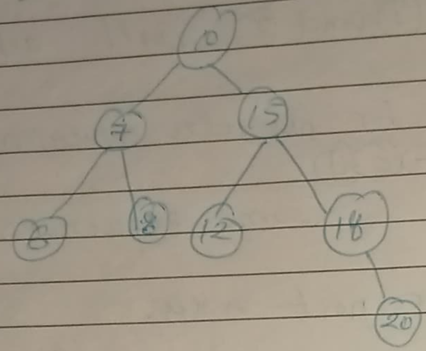
10	15	12	7	8	18	6	20

↓ root ↳ now we compare this from root if smaller than LST if greater than RST



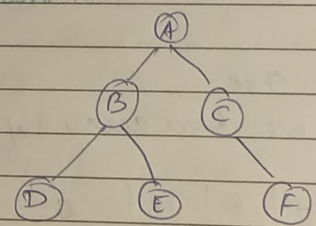
now we will compare 12 from root which is smaller than 12 therefore it will go in RST but there is already 15 therefore it will be compared again with 15 and go RST and LST by determination.

remeber always insert from top to bottom and always compare from root

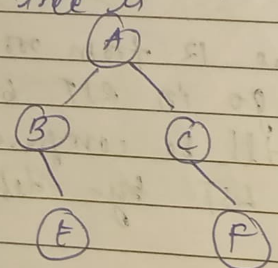


remember the last node (R node) in BST here (20) will always be the largest element and element which is in left will be the smallest. here it is (6)

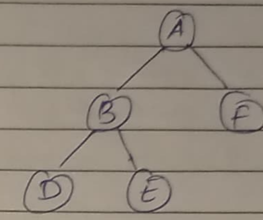
Deletion :-



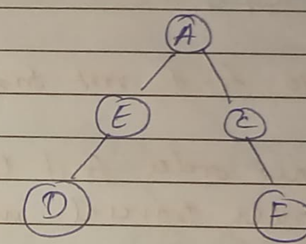
now delete D, since D is having no childs (subtree we can easily) remove it and resultant tree is



now for same tree if we delete c which having a subtree (RST) we have to connect that with root of c show that BST is valid



now if we delete from same Tree on that tree B is having 2 subtrees so we have to re arrange it by that it should be BST

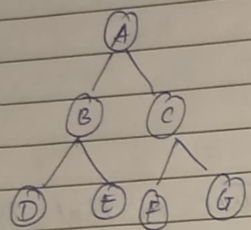


since (E) is large element than D due to E is kept on RST there fore we can swap it generally we use RST as root.

Question → make trees from traversal

for there we need minimum two traversal without it we cant make? BST

let us see a example.



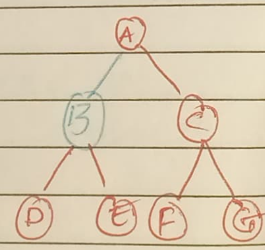
inorder traversal → D B E A F C G
pre order traversal → A B D E C F G

now first step is to find root from both traversal.
we know that in pre order first element is root and in inorder traversal middle element is root

therefore here A is root

now we know that left side of in order is left sub tree.
so there fore we have to find root of that also.
and in pre order traversal left most element is root

therefore here it is B



now post order will be DEB FGCA

AVL Tree

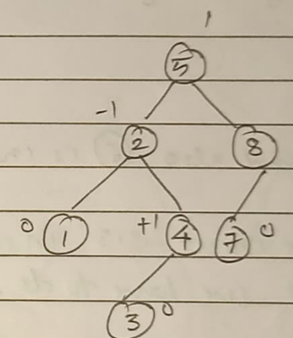
AVL tree were introduced in 1962 who got his name by inventors. Adelson Velski- Landis

The balanced defination in AVL tree is based on height of subtree. the balance information stored at each node should be (-1, 0, +1) it is given by diffrence between height of LST and RST

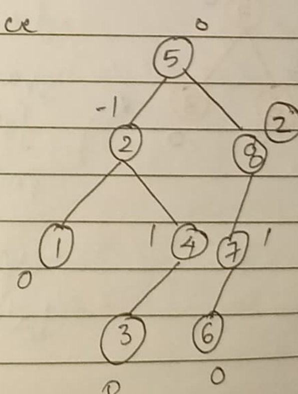
(-1, 0, +1) also known as balance factor.

now rotation to balance a AVL tree.

let us first make a BST



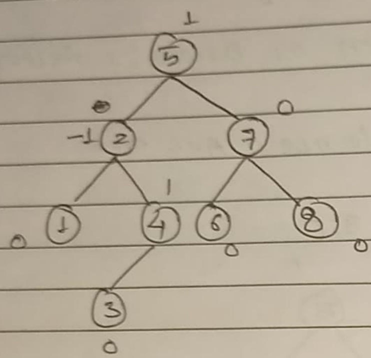
this perfectly balanced AVL tree because Every node is having (+1, 0, -1) balance factor but if we insert 6 then let's see



2 → unstable factor.

so to solve this we have to make rotation if unstable balancing factor having (+)ve sign then its having more branch on left and if its having (-)ve sign then it has more in right.

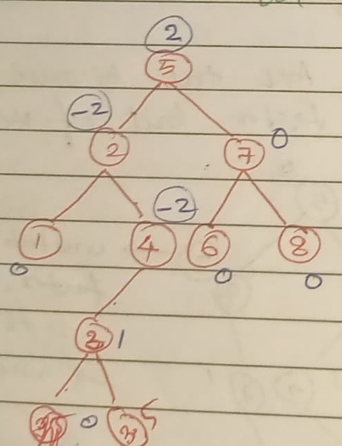
now let's do right rotation w.r.t. 8 still we have BST and let's count its Balancing factor.



now it is balanced and also (6) is inserted.

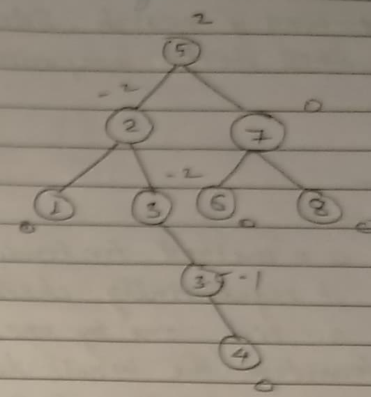
now if we want to insert 3.5 in same tree let's see what we have to do,

there are so many unbalanced factor.

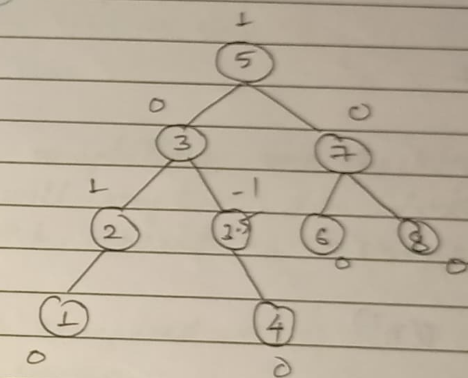


always start balancing from bottom may be up will be balance

now first let's do right rotation w.r.t. (4)



still it is not balanced, now let's do left rotation w.r.t. (2)



now it is balanced. so that's how you do rotation to balance AVL tree.

Searching and Sorting

linear search, binary search, comparison of both linear and binary search, Selection sort, insertion sort, Shell short. Comparison of sorting techniques.

Linear Search -

a linear search is a method for finding an element with in a list. it sequentially checks each element with in a list one by one. until the match is not found or the whole list is done

now types of sorting techniques.

1. bubble sort
2. Insertion sort
3. Quick Sort
4. Merge sort
5. Selection sort
6. Shell sort

Bubble Sort $O(n^2)$

Bubble sort is the simplest sorting algorithm that works by swapping the adjacent element repeatedly if they are at wrong place.

E.g. let an array

5	1	4	2	8

First Pass

here algorithm compare first two element and swaps them because 5>1

1	5	4	2	8

now again algorithm will compare 5 and 4 and 5>4, therefore they will be swapped.

1	4	5	2	8

now algorithm will compare 5.2 and it will swap again

1	4	2	5	8

now again algorithm will compare 5 and 8 here it will find that 8>5 therefore it will not be swapped.

1	4	2	5	8

Second Pass.

now after completing and checking whole array. algorithm will compare two (first) elements again. here 4>1, therefore no swapping

but when compared with 2. 2<4 therefore we have to swap.

1	2	4	5	8

now our array is sorted but our algorithm doesnot know if its compeleted. The algorithm needs one whole pass with out any swap to know that it is sorted!

Third Pass

1 2 4 5 8 → 1 2 4 5 8
1 2 4 5 8 → 1 2 4 5 8
1 2 4 5 8 → 1 2 4 5 8
1 2 4 5 8 → 1 2 4 5 8
1 2 4 5 8 → 1 2 4 5 8

Insertion Sort $O(n^2)$

insertion sort is a simple sorting algorithm that work similar to the way you sort playing cards in your hands. This array is virtually divided into a sorted and unsorted part. Values from unsorted part are picked and placed at the correct position in the sorted part.

algorithm

To sort an array of size of n in ascending order.

let us take a example.

12, 11, 13, 5, 6

let us loop for i = 1 (second element of array) to 4 (last element of array.

i = 1, since 11 is smaller than 12, move 12 and insert 11 before 12.

i = 2. 13 will remain at its own position as array before that is smaller

i = 3. 5 will move to the begging and all elements will move one position ahead from 11 to 13.

5, 11, 12, 13, 6.

i = 4. 6 will move to position after 5 and all other element will move again.

Program 1

```
#include <bits/stdc++.h>
using namespace std;

void insertionsort (int arr[], int n)
{
    int i, key, j;
    for (i=1; i<n; i++)
    {
        key = arr[i];
        j = i-1;
        while (j>=0 && arr[j] >key)
        {
            arr [j+1] = arr [j];
            j = j-1;
        }
        arr[j+1] = key;
    }
}
void printarray (int arr[], int n)
{
    int i;
    for (i=0; i<n; i++)
        cout << arr[i] << " " ~~<< endl~~;
    cout << endl;
}
```

```
int main ()
{
    int arr[] = { 12, 11, 13, 5, 6 };
    ~~cin~~ cout<< "Enter your array:" ;
    cin >> arr[i];
    int n = sizeof(arr) / size of(arr[0]);

    insertionsort (arr, n);
    printarray (arr, n);

    return 0;
}
```

an insertion sort starts by considering the two first element of array data. which are data[0] and data[1].

if they are not in order, an interchange take place. then third element. data[2], is considered. If data[2] is less than data[0] and data[1] then this two elements are shifted by one position. data[0] is placed at 1, data[1] at position 2. data[2] at 0.

if data[2] is less than data[1] but not less than data[0]. then only data[1] is moved to position 2 and its place is taken by data[2], If, finally, data[2], If finally, data[2] is not less than both its ~~pre~~ predecessors, it stays in current position.

Each element data[i] is inserted into its proper location j such that 0≤ j ≤ i, and all element greater than data[i]

an outline of insertion sort algorithm as follows.

```
insertionsort (data[], n)
   for i=0 to n-1
        move all element data[j] greater than data[i]
by one position;
        place data[i] in its proper position;
```

an implementation of insertionsort

```
template <class T>
void insertionsort ( T data[], int n)
{
    for (int i = 1, j ; i<n ; i++)
    {
        T tmp = data[i];
        for ( j=i; j>0 && tmp < data[j-1]; j--)
        data[j] = data [j-1];
        data[j] = tmp;
    }
}
```

Comb Sort.   o (n logn)

Comb sort is mainly an improvement over bubble sort. bubble sort always compares adjacent values. So all inversion are removed one by one. Comb sort improves on bubble sort using gap of size more than 1. The gap start with large number values and shrink by a factor of 1.3 in every iteration. until it reaches the value 1. Thus comb sort removes more than one inversion counts with one swap and perform better than bubble sort. Although, it works better than bubble sort on average, worst case remains $O(n^2)$. Below is the implementation.

```cpp
#include <iostream>
using namespace std;
int getNextgap (int gap)
{
    gap = (gap * 10)/13;
    if (gap < 1)
        return 1;
    return gap;
}
void combsort (int a[], int n)
{
    int gap = n;
    bool swapped = true;
    while (gap != 1 || swapped == true)
    {
        gap = getNextGap (gap);
        swapped = false;
        for (int i=0; i<n-gap; i++)
        {
            if (a[i] > a[i + gap])
            {
                swap (a[i], a[i+gap]);
                swapped = true;
            }
        }
    }
}
int main ()
{
    int a[] = { 8, 4, 1, 56, 3, -44, 23, -6, 28, 0};
    int n = sizeof(a) / sizeof (a[0]);
    combSort (a, n);
    cout << "Sorted array : " << endl;
    for ( int i=0 ; i<n ; i++)
    {
        cout << a[i] << "  ";
        ~~return 0;~~
    }
    return 0;
}
```

output → Sorted array:
-44 -6 0 1 3 4 8 23 28 56

{ (cxxdroid) }

Shell sort:

Efficient sorting algorithm

order of n^2 ~~log n~~ limit for a sorting method is too much large and must be broken to improve efficiency.

Shell sort is an algorithm that sorts the element far apart from each other and successively reduced the interval between the elements. This is generalised version of ~~the~~ insertion sort.

Optimal sequence

$\frac{N}{2}, \frac{N}{4}, \dots 1$ } original sequence

$1, 4, 13 \dots \frac{3^K - 1}{2}$ } knuth's increment

How shellsort works

suppose we have to sort the following array
{9, 8, 3, 7, 5, 6, 4, 1}

we are using shells original sequence as interval in our algorithm.

in the first loop if the array is N=8, the element lying at interval $\frac{N}{2} = 4$ and swapped if they are not ordered.

a. 0^{th} element will be compared to 4^{th}

b. if the 0^{th} element is greater than the 4^{th} one then, then 4th element will first stored in temp variable and the element stored in 0^{th} position ~~in temp will be~~ will be stored in 4^{th}

0^{th}				4^{th}			
9	8	3	7	5	6	4	1
0	1	2	3	4	5	6	7

temp

5

9	8	3	7	9	6	4	1
5	8	3	7	9	6	4	1

this process goes on for all elem

5	8	3	7	9	6	4	1
5	6	3	7	9	8	4	1
5	6	3	7	9	8	4	1
5	6	3	1	9	8	4	7

In the second loop $\frac{N}{4} = \frac{8}{4} = 2$, again lying at this are sorted

5	6	3	1	9	8	4	7
3	6	5	1	9	8	4	7
3	1	5	6	9	8	4	7

now same process goes on for every elements

3	1	5	6	9	8	4	7
3	1	5	6	9	8	4	7
3	1	4	6	5	8	9	7
3	1	4	6	5	8	9	7

5. finally at the interval is $\frac{N}{8} = \frac{8}{8} = 1$

3	1	4	6	5	8	9	7
1	3	4	6	5	8	9	7
1	3	4	5	6	8	9	7
1	3	4	5	6	7	8	9

shell sort algorithm

shellSort (array, size)
for interval i <- size/2n down to 1 for each interval "i" in array sort all the element at interval "i" end shellsort.

complexity

- worst case $O(n^2)$
- best case $O(n \log n)$
- average case $O(n^{1.25})$

Program.

```
#include <iostream>
using namespace std;        //shell sort
void shellsort (int array[], int n){
    //rearange elements at each n/2, n/4, n/8....
interval.
    for (int interval = n/2 ; interval>0; interval /= 2)
        for (int i = interval; i<n ; i+=1){
            int temp = array[i];
            int j;
            for (j=i; j>= interval && array[j - interval]
            temp ; j -= interval) {
            array[j]= array[ j- interval];
            }
            array[j]= temp;
        }
    }
}
```

```
// print an array

void printArray (int array[], int size){
    int i;
    for(i=0; i<size; i++)
        cout<<array[i] <<" ";
        cout<<endl;
}
// driver code
int main (){
    int data[] = {9, 9, 3, 7, 5, 6, 4, 1};
    int size = sizeof(data) / sizeof (data[0]);
    shellsort (data, size);
    cout<<"Sorted array : "<<endl
    printArray (data, size);
}
```

repeatedly swap two adjacent element if they are in wrong order.

wrong order = L > R

(1) 12, 45, 23, 51, 19, 8

12, 23, 45, 51, 19, 8

12, 23, 45, 19, 51, 8

12, 23, 45, 19, 8, 51 → this max element reached in right place

unsorted | sort

$\boxed{5}$ iteration for $\boxed{6}$ element

(2) 12, 23, 45, 19, 8

12, 23, 19, 45, 8

12, 23, 19, 8, 45, 51

no comparison

(3) 12, 23, 19, 8, 45, 51

12, 19, 23, 8, 45, 51

12, 19, 8, 23, 45, 51

(4) 12, 19, 8, 23, 45, 51

12, 8, 19

(5) $\boxed{8, 12, 19, 23, 45, 51}$ sorted

$\boxed{n-1}$ iteration for n elements

1st iteration — n-1

2 — n-2

3 — n-3

4 — n-4

5 — n-5

$\boxed{i^{th}}$ — $\boxed{n-i}$.

```
#include <iostream>
using namespace std;
int main(){
 int n;
 cin>>n;
 int arr[n];
 for(int i=0;i<n;i++){
        cin>>arr[i];
        }
  int counter = 1;
  while(counter < n){
       for(int i=0; i<n-counter; i++){
            if(arr[i]<arr[i+1]){
                int temp=arr[i];
                arr[i]=arr[i+1];
                arr[i+1]=temp;
            }
       }
       counter++;
  }
  for(int i=0; i<n; i++){
       cout<<arr[i]<<" ";
  }
  return 0;
}
```

(4) insertion sort

insert an element from unsorted array to its correct position in sorted array.

where left element is smaller and right element is greater

12, 45, 23, 51, 19, 8

assume sorted

checked with sorted

+1

12, 45, 23, 51, 19, 8

12, 23, 45, 51, 19, 8

~~sorted~~ (assume) ✓

+1 +1 +1

12, 23, 45, 51, 19, 8

12, 19, 23, 45, 51, 8

sorted $\boxed{8, 12, 19, 23, 45, 51}$

code:-

```
#include <iostream>
using namespace std;
int main(){
  int n;
  cin>>n;
  int arr[n];
  for(int i=0; i<n; i++){
     cin>>arr[i];
  }
```


because we assumed that 1st element is sorted

```
  for (int i=1; i<n; i++){
     int current = arr[i];
     int j=i-1;
     while (arr[j] > current && j>=0){
         arr[j+1] = arr[j];
         j--;
     }
     arr[j+1] = current;
  }

  for(int i=0; i<n; i++){
     cout<<arr[i]<<" ";
  }
  return 0;
}
```