* Practical no. 1

Aim : Implement Linear Search to find an Item in the list.

Theory :

Linear Search

Linear Search is one of the simplest searching algorithm in which targeted item is sequentially matched with each item in the list.

It is worst searching algorithm with worst case time complexity. It is a force approoch. On the other hand incase of an ordereedlist, instad of searching the list in sequence, A binary search is used which will start by examining the middle term.

Linear Search is a technique ~~the~~ to compare each and every element with the key element to be found. If both them matches, the algorithm returns the element found and ~~th~~ its positions is also found.

2] Sorted Linear search:

Sorting means to arrange the element in increasing or decreasing order.

- Algorithm:

Step1: Create Empty list and assign it to 4 variable

Step2: Accept total no of elements to be inserted into the list from user (say 'n').

Step3: Use for loop using append() method to add the elements in the list.

Step4: Use sort() method to sort the accepted element and a in increasing order the list then print the list.

Step5: Use If statement to give the range in which element is found in given range then display "Element not found".

Step6: Then use else statement. If element is not found in range then satisfy the given condition.

Step7: Use for loop in range from 0 to the total no. of element searched before doing this accept an search no from u using Input statement

```
def linear(arr, x):
    for i in range(len(arr)):
        if arr[i] == x:
            return i
inp = (input("Enter elements in arrange:").split())
array = []
for ind in inp:
    array.append(int(ind))
print("elements in array are:", array)
array.sort()
x1 = int(input("enter element to be searched:"))
x2 = linear(array, x1)
if x2 == x1:
    print("Element found at position", x2)
else:
    print("Element not found")
```

```
>>> Enter element in array: 1 2 3 5
>>> Elements in array: [1 2 3 5]
>>> Enter element to be
```


Step 8 – Again initialize a variable: do are the defind function

Step 9 – Use if conditional statement to check if the variable in step 8 matches with the element you want to find then print the index corresponding to the element if the condition doesn't satisfy than print that element is not found.

Practical No-2

Aim : Binary search Algorithm

Step 1 - Create empty list and assign it to a variable.

Step 2 - Using input method, accept the range of given list

Step 3 - Use for loop, add elements in list using append () method.

Step 4 - Use sort () method to sort the accepted element and assign it in increasing ordered list print the list after sorting.

Step 5 - Use the if loop to give the range in which element is found then ~~display~~ a message "Element not found.

Step 6 - Then use else statement ; if statement is not found in range then satisfy the below condition

Step 7 - Accept an argument and key of the element has to be Searched.


```
a = list (input ("Enter the list of elements:"))
a.sort()
n = len(a)
s = int(input("Enter the number to be searched:"))
if (s > a[n-1] or s < a[0]):
    Print ("Element not found")
else:
    f = 0
    l = n-1
    for i in range (0, n):
        m = int ((f+l)/2)
        if s == a[m]:
            print ("The element is found at:", m)
            break
        else:
            if s < a[m]:
                l = m-1
            else:
                f = m+1
```

```
777 Enter the list of elements : 1524
    Enter the number to be searched : 5
    The element is found at: 1
```

Step 8 – Initialize first to 0 and last element of the list as array is staisfing from 0 hence it is initialized 1 less than the total const

Step 9 – Use for loop and assign the given range

Step 10 – If statement is list and still the element to be searched is not found then find the middle element (m).

Step 11 – Else if the item to searched is still less that the middle term then initialize mid(m) -1 Else initialize f(l) = mid(m)-1

Theory –

Binary search is also known as half interval search logarithmic search binary chop is a search algorithm that finds the position of a target within a sorted array. If your are looking for the number which is at the end of the list then you need to search entire list in linear search which is time consuming This can be avoided by using binar

Practical no. 4.

Aim: Implementation of stacks using Python Lists

Theory: A stack is a linear data structure that can be represented in the real world in the form of a physical stack or a pile. The elements in the stack are added or removed only from one position, i.e the topmost position, Thus, the stack works on the LIFO (Last in first Out) principle as the element that was inserted last will be removed first. A stack can be implemented using array as well as linked list. Stack has three basic operation: push, pop,

Algorithm:

Step 1: Create a class stack with instance variable items.

Step 2: Define the init method with self argument and initialize the initial value and then initialize to an empty list

Code:

```
Print ("Anand Maurya")
class stack:
    global tos
    def __init__(self):
        self.l = [0,0,0,0,0,0,0,0]
        Self.tos = -1
    def push(self,data):
        n = len(self.l)
        if self.tos == n-1:
            Print("stack is full")
        else:
            self.tos = self.tos + 1
            self.l[self.tos] = data
    def pop(self):
        if self.tos < 0:
            print("stack empty")
        else:
            k = self.[self.tos]
            print("data =", k)
            self.tos = self.tos - 1
```

Output

Anand Maurya

>>> S.push(10)
>>> S.push(20)
>>> S.push(30)
>>> S.push(40)
>>> S.push(50)
>>> S.push(60)
>>> S.push(70)
>>> S.push(80)
>>> S.push(90)
Stack is full.

>>> S.pop()
data = 80
S.pop()
data: 70
S.pop()
data: 60
S.pop()
data 50
pop()
data 40
pop()
data 30
pop()
data 20

>>> S.pop()
data 10
Stack empty

Step 3: Define methods push and pop under the class stack.

Step 4: Use if statement to give the condition that if length of given list is greater than the range of list then print stack is full.

Step 5: Or Else print statement as insert the element into the stack and initialize the value.

Step 6: Push method used to insert the element but pop method used to delete the element from the stack.

Step 7: If in pop method value is less than 1 than return the stack is empty or else delete the element from stack at topmost position.

Step 8: Assign the element values in push method and print the given value in poped

Step 10: Attach the input and output of above algorithm.

Step 8: first condition checks whether the no. of elements are zero while the second case whether tos is assigned any value. If tos is not assigned any value, then they can be store that stack is empty.

04/01/2020

P&D

Practical - 5

Aim : Implement Quick Sort to sort the given list.

Theory : The quick sort is a recursive algorithm then based on divide & conquer technique

Algorithm:

Step I : Quick sort first selects a value, which is called pivot value. first value element serve as our first pivot value.

Step II : The position process will happen next. It will find the split point and at the same time move others item to appropriate side of list

Step III : Positioning begins by locating two position marker lets call them left mark & right mark at the beginning & end of remaining items in the list. The goal of the platform process is to move item that are on wrong side.

Step IV : We begin by increasing left mask until we locate a value that is greater then the p.v. We then decrement right mark until we find value that is less than the p.v.

Step V : At the point where right mark becomes less than leftmark we stop. This, position of rightmark is now the split point.

```
def quicksort(alist):                                    040
    quicksorthelper(alist,0,len(alist)-1)
def quicksorthelper(alist,first,last):
    if first<last:
        splitpoint = partition(alist,alist,first,last)
        quicksorthelper(alist,splitpoint+1,last)
def partition(alist,first,last):
    pivotvalue = alist[first]
    leftmark = first+1
    rightmark = last
    done = False
    while not done
        while leftmark<=rightmark and alist[leftmark]
                                        <=pivotvalue
        while alist[rightmark]>=pivotvalue
                and rightmark>=leftmark:
        if rightmark<leftmark
            done = True
        else:
            temp = alist[leftmark]
            alist[leftmark]=alist[rightmark]
            alist[first]=alist[rightmark]
            return rightmark.
alist=[42,54,45,67,89,66,55,80,100]
quicksort(alist)
print(alist)
```

040

Output :

[42, 45, 54, 55, 65, 89, 67, 80, 100].

041

Step 6 : The P.V. can be exchanged with content of split point and P.V is now in place.

Step 7 : In addition, all the items to left of split pt. are less than P.V. and all the items to the left to right at split pt. greater than P.V. The list can now be divided at split pt. and quick sort can be recursively on 2 values.

Step 8 : Quicksort function involves recursive functions, quick sort helper.

Step 9. Quick sort helper, begins with same base as ~~merge sort~~.

Step 10 : If length of the list is less than 0 or equal one list is already sorted.

Step 11 : If is the greater than it can be positioned and recursive functions.

Step 12 : The position function implement the process described earlier.

Step 13 : Display and stick the coding and output of above algorithm

Practical – 06

Aim: Implementing a Queue using python list

Theory: Queue is a linear data structure which has two references front and rear.
Implementing a queue using python list is the simplest as the python list provides built in function to perform the specified operations of queue. It is based on principal that a new element is inserted in rear and element of queue is deleted which is at front. It is called FIFO principle.

- Queue (): Creates a new empty queue.
- Enqueue (): Insert an element at the rear of the queue and similar to that of insertion of linked using tail.
- Dequeue (): Returns element which was at the front. the front is moved to the successive element. An dequeue operation cannot remove element if the queue is empty.

CODE:

```
Class Queue:
    global r
    global f
    def __init__(self):
        self.r=0
        self.f=0
        self.l=[0,0,0,0,0]
    def add(self, data):
        n= len(self.l)
        if self.r<n-1:
            self.l[self.r]=data
            self.r=self.r+1
        else:
            print("Queue is full")
    def remove(self):
        n= len(self.l)
        if self.r<n-1:
            print(self.l[self.f])
            self.f=self.f+1
        else:
            print("Queue is empty")

Q=Queue()
Q.add(30)
Q.add(40)
Q.add(50)
Q.add(60)
Q.add(80)
Q.add(90)
```

Q. remove()
Q. remove()
Q. remove()
Q. remove()
Q. remove()
Q. remove()

Algorithm:

S1: Define a class Queue and assign global variables then define init() method with self argument in init() assign or initalize the init value with help of self argument.

S2: Define empty list and define enqueue() method with 2 arguments, assign length of empty list.

S3: Use if statement that length is equal to queue than Queue is full or else insert element in empty list or display that Queue element added successfully & increment by 1.

S4: Define dequeue() with self argument under this use if statement in that front is equal to length of list then display Queue is empty or else give that front is at 0 and using that, delete element from front side and increment it by 1.

S5: Now, call Queue() function & give element that has to be added in the empty list by using enqueue() and print the list after adding and same for deleting and display the list after deleting the element in list.

Practical - 07

Evaluation of a postfix Expression

Aim : Program on Evaluation of given string by using stack in python Environment i.e. postfix

Theory : The postfix expression is free of any parameters further we look case of the ambiguities of the expression in the program Reading the expression is always from left to right in postfix.

Algorithm :

Define evaluate at function then create an empty sack in python.

Convert the string to a list by using the string method split.

Calculate length of string and point it

Use for loop to assign the range of string then given condition using if statement.

```
def evaluate(s):
    k = s.split()
    n = len(k)
    stack = []
    for i in range(n):
        if k[i].isdigit():
            stack.append(int(k[i]))
        elif k[i] == '+':
            a = stack.pop()
            b = stack.pop()
            stack.append(int(b) + int(a))
        elif k[i] == '*':
            a = stack.pop()
            b = stack.pop()
            stack.append(int(b) * int(a))
        else:
            a = stack.pop()
            b = stack.pop()
            stack.append(int(b)/int(a))
    return stack.pop()
s = "8 69 * + "
r = evaluate(s)
print("The evaluated value is:", r)
print("Rakesh Kumar Anand Maurya")
```

Output:

>>>
The evaluated value is : 6 2

Anand Mawuya

Steps: Scan the token List from left to right - If token is an operator, convert it from a string to an integer and push the value onto the 'p'.

S6: If the token is an operator (*, /, +, -, ^) it will need 2 operands. Pop the 'p' twice. The first pop is 2nd operand and 2nd pop is the first operand.

S7: Perform the Arithmetic operation. Push the result back on the 'p'.

S8: When the input expression has been completely processed, the result is on the stack. Pop the 'p' and return the value.

S9: Print the result of string after evaluation of Postfix

S10: Attach output and input of above algorithm.

Practical – 08

Aim: Implementation of single Linked List by adding nodes from last position

Theory: A Linked list is a linear data structure which stores the elements in a node in a linear fashion but not necessarily contigious. The individual elements of the linked List called a Node. Node comprises of 2 parts ① Data ② Next. Data stores information of element whereas next refers to the next node by pointing towards it. In case of larger List, if we add/remove any element from the list, all the elements of list has to adjust itself every time we add it is very tedious task so Linked List is used to solving this type of task.

Algorithm:

Transversing of linked list means visiting all the nodes in linked list in order to perform some operation on them.

The entire Linked List means can be accessed using first node of list. The first node of list is turn is refered by Read pointer of the

Code:

```
Class node:
    global data
    global next
    def __init__(self,item):
        Self.data = item
        self.next = None
Class linkedlist:
    global s
def __init__(self):
    self.s = None
def add1(self, item):
    newnode = node(item)
    if self.s = None:
        Self.s = newnode
    else:
        head=self.s
        while head.next != None
            head = head.next
        head.next = newnode
def add2(self, item):
    newnode = node(item)
    if self.s == None:
        Self.s = newnode
    else:
        newnode = self.s
        Self.s = newnode
```

```
def display(self):
    head = self.S
    while head.next != None:
    Print(head.data)
    head = head.next
    Print(heads data)
    Start.addL(50)
    Start.addL(60)
    Start.addL(70)
    Start.addL(80)
    Start.addB(40)
    Start.addB(30)
    Start.addB(20)
    Start.display()
    Print("Anand Maurya")
```

Output:

```
20
30
40
50
60
70
80
Anand Maurya
```


53. Thus entire list can be traversed using node which is referred by the head pointer of the linked list.

54. Now that we know that we can traverse the entire Linked List using head pointer, we should only use it to refer the first node of list only.

55. We should not use head pointer to transverse entire linked list because the head pointer is our only reference to 1st node in the Linked List's modifying reference of the head pointer can lead to changes which we cannot revert back.

56. We may lose the reference to the 1st node in our Linked List and hence most of our Linked List so, in order to avoid making some unwanted changes to the 1st node, we will use a temporary node to transverse the entire Linked list.

57. We will use this temporary node as a copy of the node we are currently traversing. Since is are we making currently making a copy of temporary node the datatype of the temporary node is also of the node.

S8. Now that current is referring to the first node, if we write current = current.next it will refer to the next node of the list.

S9. But the 1st node is refered by current. So, we can traverse to 2nd node as current = current.next

S10. Similarly, we can traverse rest of nodes in the linked list using same method by while loop.

S11. Our concern now is to find a terminating condition for the while loop.

S12. The last node in linked list is refered by tail of list. Since the last node of Linked List does not have any next node, the value in the next field of the last node is None.

S13. So we can refer the last node of list as

Self.next = None

S14. We have to now see how to start traversing Linked list & how to identify whether we have reached the last node of list or not.

Attach loading a/p & o/p of above algorithm

```
def sort(arr, l, m, r):
    n1 = m - l + 1
    n2 = r - m
    L = [0] * (n1)
    R = [0] * (n2)
    for i in range(0, n2):
        R[j] = arr[m + 1 + j]

    i = 0
    j = 0
    k = l
    while i < n1 and j < n2:
        if L[i] <= R[j]:
            arr[k] = R[i]
            i += 1
        else:
            arr[k] = R[j]
            j += 1
        k += 1
    while i < n1:
        arr[k] = L[i]
        i += 1
        k += 1
```

Practical -09

Merge Sort

Aim :- Implementation of merge sort by using python

Theory : Merge sort is a divide and conquer algorithm. It divides input array in two halves, calls itself for the two halves and then merges the two sorted halves. The merge (arr, l, m, r) is key process that assumes that arr[l..m] and arr[m+1, r] are sorted and merges the two sorted sub-arrays.

Algorithm:

S1: The unit is divided into and sized in each recursive call until two adjacent elements are obtained.

S2: Now begins the sorting process. the i and j iterates traverse the two halves in each call. the K iterates traverse the whole lists and makes changes along the way.

S3: If the value at j, L[i] is average to the arr[i+j] slot and is encountered. If not then R[j] is chosen

S4: This way the values being assigned the range [l+i] are all sorted.

```
While j<n2:
    arr[k]=R[j]
    j+=1
    k+=1
def mergesort(arr,l,r):
    if l<r:
        m= int((l+(r-1))/2)
        mergesort(arr,l,m)
        mergeSort(arr,m+1,r)
        sort(arr,l,m,r)
arr = [12,23,34,56,78,45,86,98,42]
Print(arr)
    n= len(arr)
    mergesort(arr,0,n-1)
        Print(arr).
```

Output:

>> [12, 23, 34, 56, 78, 45, 86, 98, 42]:
[12, 23, 56, 56, 42, 45, 78, 86, 98]

CODE:

```
set 1 = set()
set 2 = set()
for i in range (8,15):
    set1.add (i)
for i in range (1,12):
    set.2.add (i)
Print ("Set 1: ",set1)
Print ("set 2:", set2)
Print ("\n")
Set 3 = set1 | set 2
Print ("Union of set1 and set2: set3", set3)
set4 = set1 & set2
Print ("Intersection of set1 and set2: set4", set4)
print ("\n")
if set3 > set4:
    Print ("Set3 is superset of set 4")
elif set3 < set4:
    Print ("set 3 is subset of set4")
else:
    Print ("set 3 is not of set4")
if set4 < set3:
    Print ("set4 is subset of set3")
    print ("\n")
```


Sets.

Aim: Implementation of sets using python.

Algorithm:

S1: Define two empty sets as set 2 and set1. Now, use if statement providing the range of the above 2 sets.

S2: Now, add() method is used for adding elements according to the given range then print the sets after addition.

S3: Find the union and intersection of above two sets by using (and, &) (or) method print the sets of union and intersection sets.

~~S4: Find the union and intersection of above two sets by using~~

S4: Use if statement do find out the subset and superset of set 3 and set4. Display the above sets.

S5: Display that element in set3 is not in set4 using mathematical operation.

S6: Use its disjoint () to check that anything is common or element is present or not. If not then display that it is mutually exclusive sets.

57. Use clear() to remove or delete the set and print the set after clearing the element present in the set


```
Set 5 = Set 3 - set 4
Print ("Element in set 3 and not in set 4.
        set 5", set5)
Print ("\n")
if set4.isdisjoint(set5):
    Print ("set4 and set5 are mutually
            exclusive \n")
    set5.clear()
    Print ("After applying clear, set5 empty set:")
    Print ("Set 5=", set5)
```

Output:

```
set 1 : {8, 9, 10, 11, 12, 13, 14}
set 2 : {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11}.
Union of set 1 and set 2 : set3 {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14}
Intersection of set 1 and set 2 : set4 {8, 9, 10, 11}
  set 3 is superset of set x
Elements in set 3 and not in set4 : set5 {1, 2, 3, 4, 5, 6, 7, 12, 13, 14}
set 4 and set5 are mutually exclusive
after applying clear, set5 is empty set:
  Set 5 = set()
```

CODE:

```
Class node:
    def __init__(self, value):
        self.left = None
        self.value = value
        self.right = None
Class BST:
    def __init__(self):
        self.root = None
    def add(self, value):
        p = node(value)
        if self.root == None:
            self.root = p
            print("Root is added successfully", p.val)
        else:
            h = self.root
            while True:
                if h.left == None:
                    h.left = p
                    print(p.val, "Node is added to left side successfully")
                    break
                else:
                    h = h.left
            else:
                if h.right == None:
                    h.right = p
                    print(p.val, "Node is added to right side")
```


Practical - 11

Binary Search Tree

Aim: Implementation of Binary search tree using python and Inorder, Preorder, postorder Traversal.

Theory: Binary Tree is a tree which supports maximum of 2 children for any node within the tree. This any particular node can have either 0 1 or 2 children. There is another identify of binary tree that it is ordered such that one child is identified as left child and other as right

(1) Inorder: (i) Traverse left subtree. The left subtree is from right have left and right subtrees
(ii) Visit the root node.
(iii) Traverse the right subtree and repeat it

(2) Preorder: (i) Visit the root node
(ii) Traverse left subtree. The subtree might not have left and right subtree
(iii) Traverse the right subtree and repeat it.

(3) Postorder: (i) Traverse the left subtree. The left subtree might have left and right subtrees.
(ii) Traverse the right subtree
(iii) Visit the root node.

Algorithm:

S1. Define class node and define init() method with 2 arguments. Initialize the value in this method.

S2. Again, define a class BST that is Binary Search Tree with init() methods with self arguments and assign the root to node.

S3. Define add() method for adding the node. Define a variable p=node(value).

S4. Use while loop for checking node ~~is less than or greater~~ the condition that root is more than the else statement for if node is less than the main node then put on a passage that in the left side.

S5. Use while loop for checking node is less than or greater than the main node and break the loop if it is not satisfying.

S6. Use if statement within that else statement checking that node is greater than main root then put it into right side.

S7. After this, left subtree and right subtree repeat.

break

else:

h=h.right

```
def Inorder(root):
    if root==None
        return
    else:
        Inorder(root.left)
        Print(root.val)
        Inorder(root.right)
def Preorder(root):
    if root==None:
        return
    else:
        Print(root.val)
        Preorder(root.left)
        Preorder(root.right)
def Postorder(root):
    if root==None:
        return
    else:
        Postorder(root.left)
        Postorder(root.right)
        Print(root.val)

t=BST()
t.add(1)
t.add(2)
t.add(3)
t.add(4)
t.add(5)
```

Output

>>> print ("In Inorder form of tree", Inorder(t.root))

1
2
3
4
5

Inorder form of the None:

>>> print ("preorder form of tree", Preorder(t.root))

1
2
3
4
5

Preorder form of tree None

>>> print ("post order form of tree", Post order(t.root))

3
5
4
2
1

Postorder form of tree None.

This method to arrange the node according to Binary Search Tree.

S8: Define Inorder(), preorder() and postorder() with root argument and use if statement that root is none and return that in all.

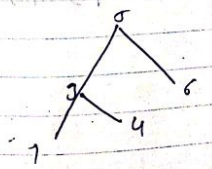
S9. In Inorder, else statment used for giving with condition first left, root and then right node.

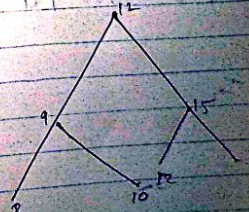
S10. For postorder, In else post, assign left then right and then go for root node.

S11. For preorder, we have to give condition in else that first root, left and then right node.

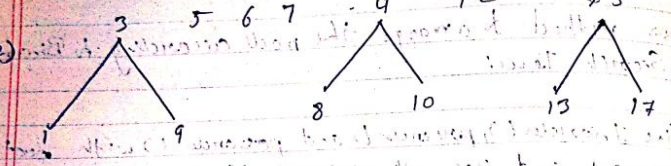
S12. Display the input & output.

(LVR)





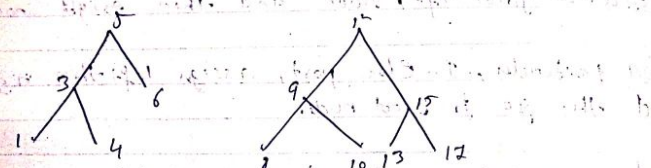
S2) 3 5 6 7 9 12 25



S3. 1 3 4 5 6 7 8 9 10 12 13 15 17

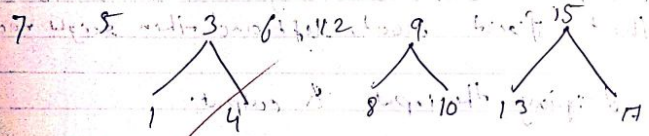
Pre order : (VLR)

S1:

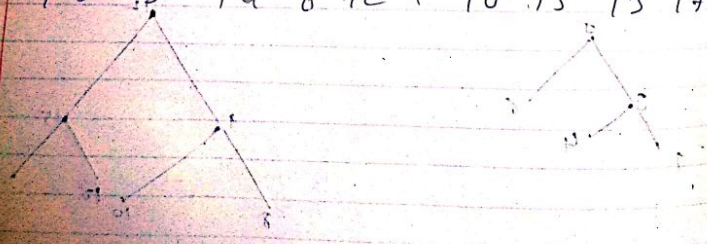


7. 5 3 6 12 9 15

1 4 8 10 13 17

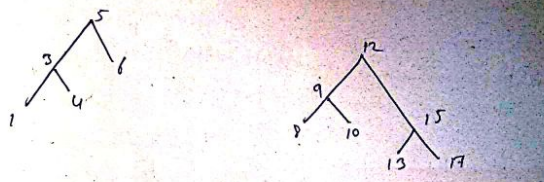


7 5 3 1 4 6 12 9 10 15 13 17.

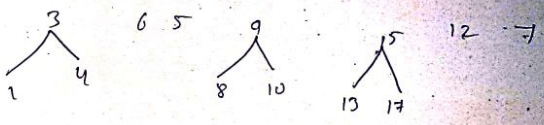


Post order (LRV)

S1:



S2:



S1 1 4 3 6 5 8 10 9 13 17 15 12 7

* Binary search tree.