Modern Education Society's College of Engineering, Pune

210256: DATA STRUCTURES ALGORITHM LABORATORY (2019 C0URSE)

NAME OF STUDENT: Gauri S. Bharage	CLASS: SE II
SEMESTER/YEAR: IV / 2022-23	ROLL NO: 37
DATE OF PERFORMANCE: 3/5/23	DATE OF SUBMISSION: 17/5/23
EXAMINED BY:	EXPERIMENT NO: 09

TITLE: Implementation of Adelson, Velskii, and Landi (AVL) tree

AIM/PROBLEM STATEMENT: Dictionary stores keywords & its meanings. Provide facility for adding new keywords, deleting keywords, updating values of any entry. Provide facility to display whole data sorted in ascending/ Descending order. Also find how many maximum comparisons may require for any keyword. Use Height balance tree and find the complexity for finding a keyword.

OBJECTIVES:

1. To understand tree data structure.
2. To understand practical implementation and usage of non-linear data structures to solving problems of AVL Tree.

OUTCOMES:

1. Apply and analyze non-linear data structures to solve real world complex problems.

PRE-REQUISITES:

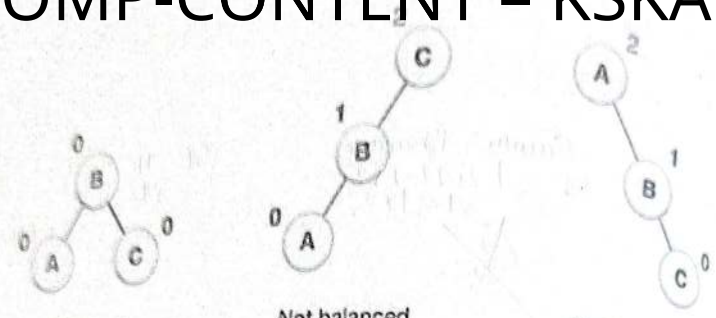
1. Knowledge of C++ programming.
2. Knowledge of AVL tree

THEORY:

It is observed that BST's worst-case performance is closest to linear search algorithms, that is O(n). In real-time data, we cannot predict data pattern and their frequencies. So, a need arises to balance out the existing BST.

Named after their inventor **Adelson, Velski & Landis, AVL trees** are height balancing binary search tree. AVL tree checks the height of the left and the right sub-trees and assures that the difference is not more than 1. This difference is called the **Balance Factor**.

Here we see that the first tree is balanced and the next two trees are not balanced –



In the second tree, the left subtree of **C** has height 2 and the right subtree has height 0, so difference is 2. In the third tree, the right subtree of **A** has height 2 and the left is missing, so it is 0, and the difference is 2 again. AVL tree permits difference (balance factor) to be only 1.

BalanceFactor = height(left-sutree) − height(right-sutree)

If the difference in the height of left and right sub-trees is more than 1, the tree is balanced using some rotation techniques.

AVL Rotations

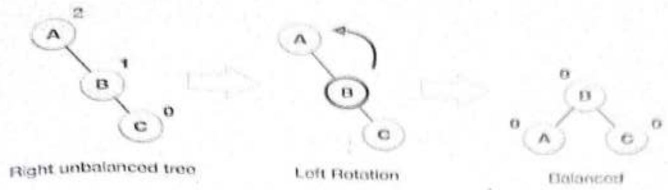
To balance itself, an AVL tree may perform the following four kinds of rotations –

- Left rotation
- Right rotation
- Left-Right rotation
- Right-Left rotation

The first two rotations are single rotations and the next two rotations are double rotations. To have an unbalanced tree, we at least need a tree of height 2. With this simple tree, let's understand them one by one.

Left Rotation

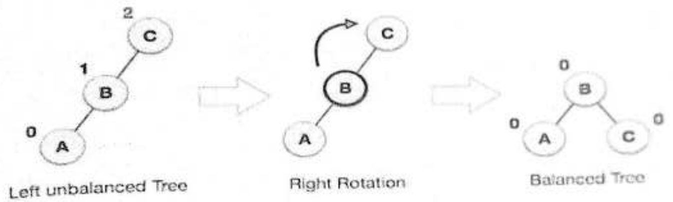
If a tree becomes unbalanced, when a node is inserted into the right subtree of the right subtree, then we perform a single left rotation –



In our example, node **A** has become unbalanced as a node is inserted in the right subtree of A's right subtree. We perform the left rotation by making **A** the left-subtree of B.

Right Rotation

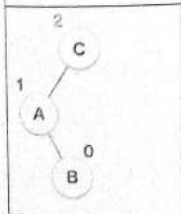
AVL tree may become unbalanced, if a node is inserted in the left subtree of the left subtree. The tree then needs a right rotation.

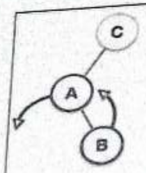
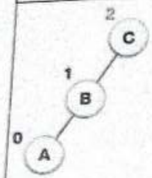
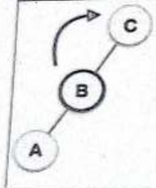
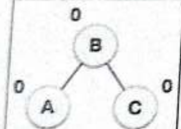


As depicted, the unbalanced node becomes the right child of its left child by performing a right rotation.

Left-Right Rotation

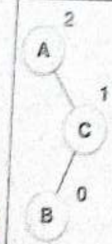
Double rotations are slightly complex version of already explained versions of rotations. To understand them better, we should take note of each action performed while rotation. Let's first check how to perform Left-Right rotation. A left-right rotation is a combination of left rotation followed by right rotation.

State	Action
	A node has been inserted into the right subtree of the left subtree. This makes **C** an unbalanced node. These scenarios cause AVL tree to perform left-right rotation.

	We first perform the left rotation on the left subtree of **C**. This makes **A**, the left subtree of **B**.
	Node **C** is still unbalanced, however now, it is because of the left-subtree of the left-subtree.
	We shall now right-rotate the tree, making **B** the new root node of this subtree. **C** now becomes the right subtree of its own left subtree.
	The tree is now balanced.

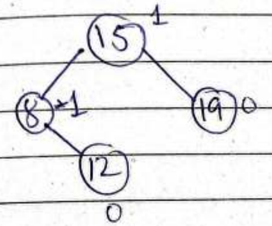
Right-Left Rotation

The second type of double rotation is Right-Left Rotation. It is a combination of right rotation followed by left rotation.

State	Action
	A node has been inserted into the left subtree of the right subtree. This makes **A**, an unbalanced node with balance factor 2.

Theory.

- AVL Tree is height balanced binary search tree.
- It is the one in which the height of the left and right sub-trees of every nodes differ by atmost one.
- For example, consider following tree



The value written besides each node is called balancing factor

- Balancing Factor = Height of left subtree - Height of right subtree
- The balance factor of each node in tree must be +1, 0, -1, then only it is AVL tree.

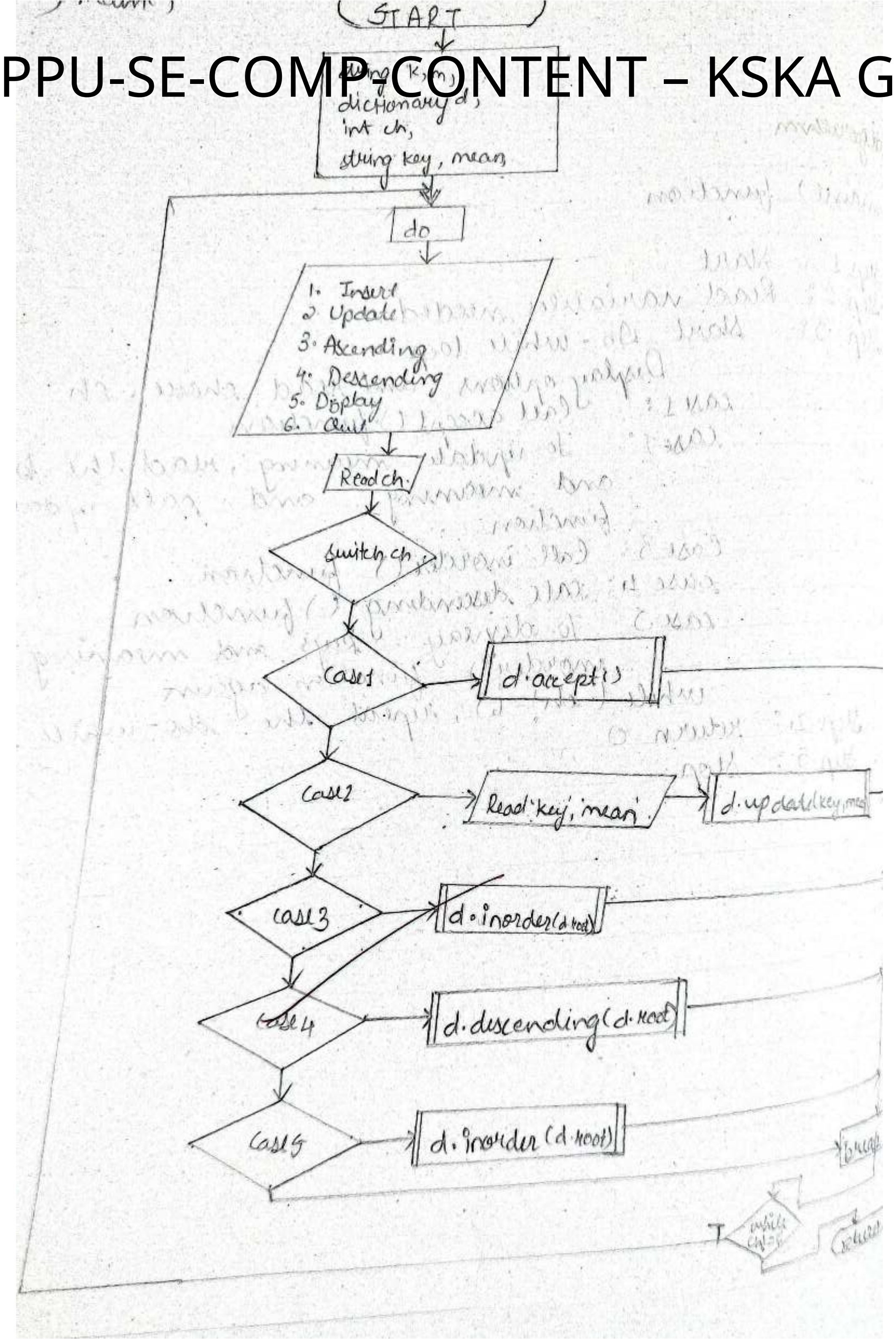
Page No. 14

Date

Algorithm

1) main() function

Step 1 :- Start
Step 2 :- Read variables needed.
Step 3 :- Start Do-while loop.
Display options, and read choice, ch
case 1 :- Call accept() function
case 2 :- To update meaning, read the key and meaning, and call update() function
Case 3 : Call inorder() function.
case 4 : call descending () function
case 5 : To display keys and meaning call inorder() function again
while (ch!=6), repeat the do-while loop.
Step 4 :- return 0.
Step 5 :- Stop.


START
dictionary d,
int ch,
string key, mean
do
1. Insert
2. Update
3. Ascending
4. Descending
5. Display
6. Quit
Read ch.
switch ch
case1
d.accept()
case2
Read key, mean
case3
d.inorder(d.root)
case4
d.descending(d.root)
case5
d.inorder(d.root)
T

2) Class avlnode

Step 1 :- Declare the variables, keyword, meaning, *left, *right, bf

Step 2 :- Define constructor of class and initialize the values.

Step 3 :- Define parameterized constructor and pass values.

Step 4 :- Make friend class as dictionary.

3) Class dictionary

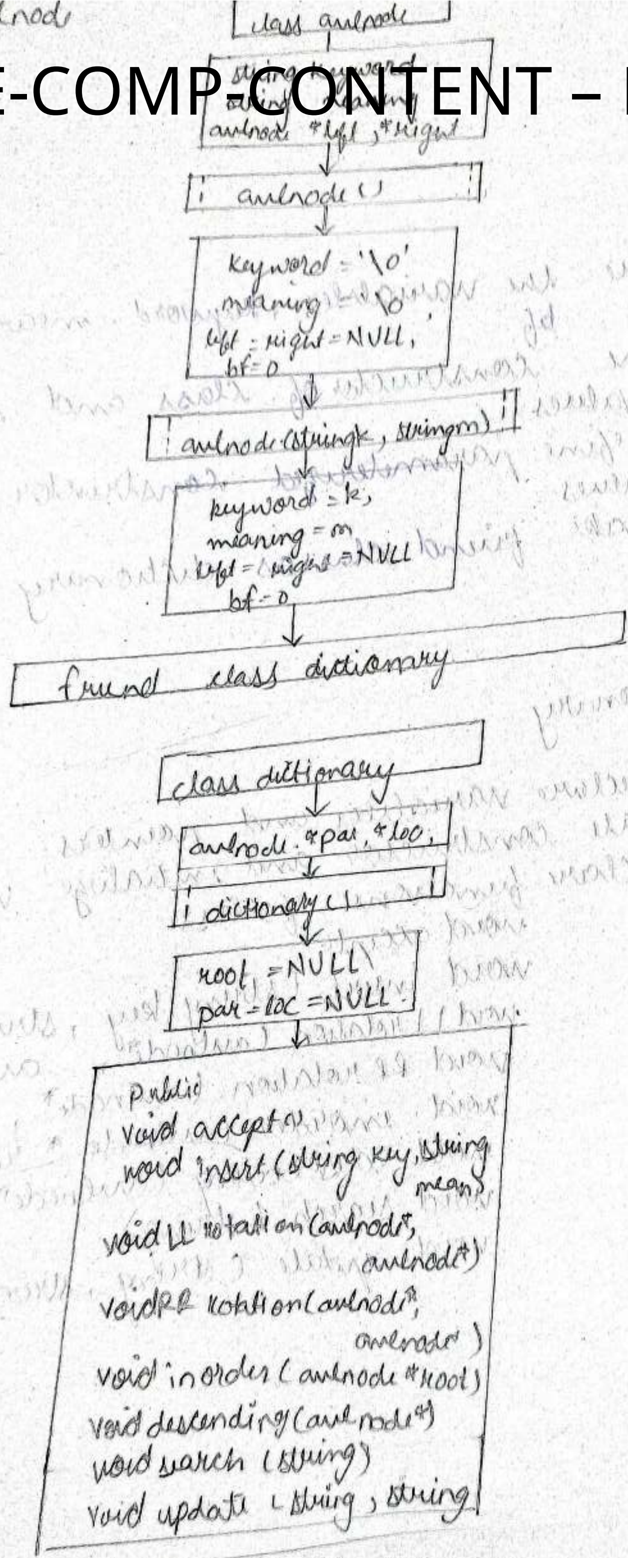
Step 1 :- declare variables and pointers.

Step 2 :- Create constructor and initialize values.

Step 3 :- Declare functions of class:

void accept()
void insert (string key, string mean)
void LLrotation (avlnode*, avlnode*)
void RR rotation (avlnode*, avlnode*)
void inorder (avlnode * root)
void descending (avlnode*);
void search (string)
void update (string, string)

2) class avlnode



class dictionary

4) insert () function

Step 1:- if (entered key is != root)
root = new avlnode (key, mean)
return
else , Step 2

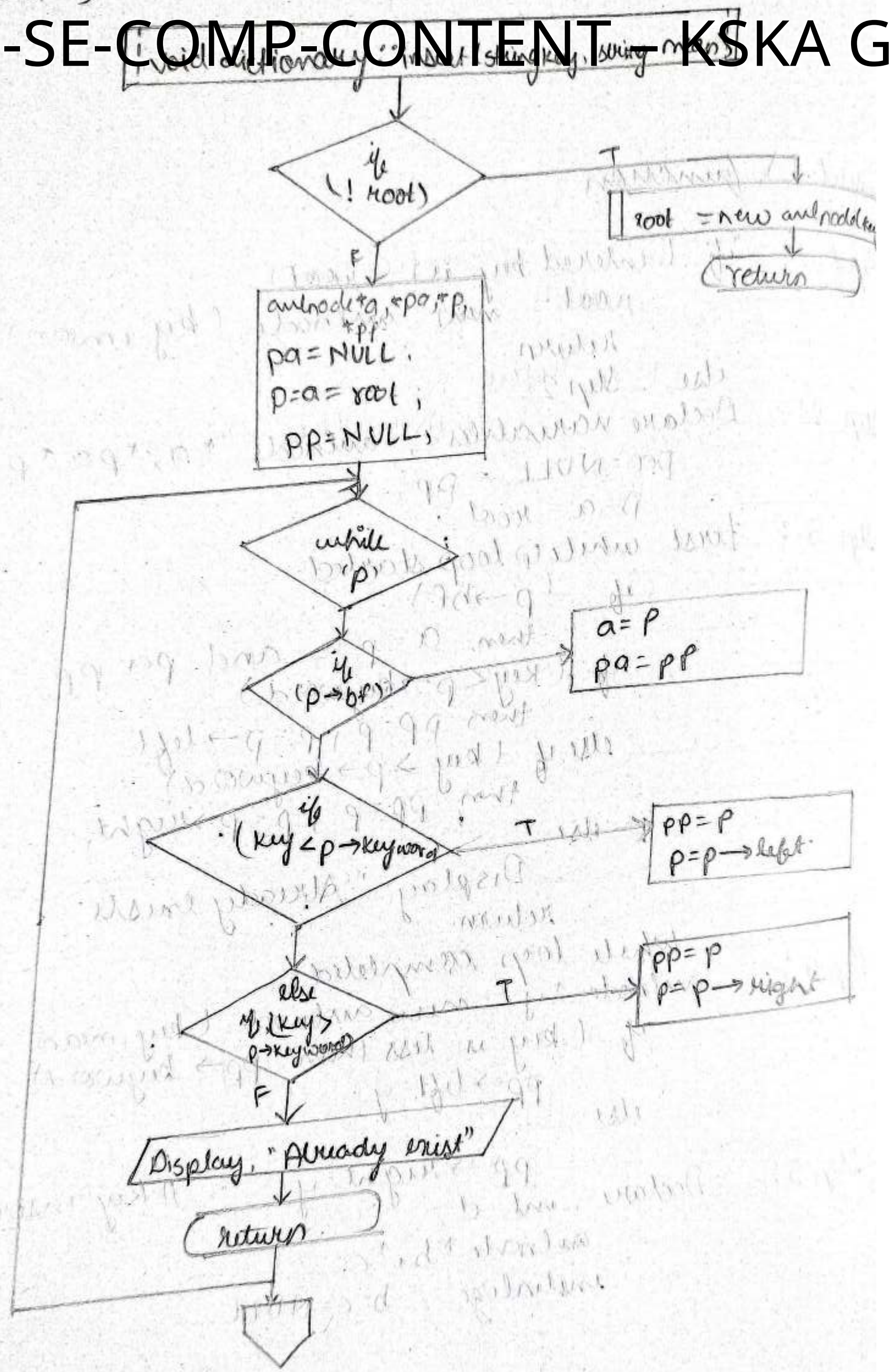
Step 2:- Declare variables , avlnode *a, *pa, *p, *pp
pa = NULL = pp;
p = a = root

Step 3:- First while(p) loop started
if (p→bf)
then a = p and pa= pp
if (key < p→keyword)
then pp = p ; p = p→left
else if (key > p→ keyword)
then pp = p ; p = p→right,
else
Display "Already exists"
return
While loop completed

Step 4:- avlnode *y = new avlnode (key, mean)
if (key is less than pp→ keyword)
pp→left = y
else
pp → right = y // Key inserted

Step 5:- Declare, int d,
avlnode *b, *c
initialize b = c = NULL

4] insert()


void dictionary :: insert(string key, string mean)
if (! root)
T
root = new avlnode(key
return
F
avlnode *a, *pa, *p, *pp
pa = NULL ;
p = a = root ;
pp = NULL ;
while (p)
if (p→bf)
a = p
pa = pp
if (key < p→keyword)
T
pp = p
p = p→left
else if (key > p→keyword)
T
pp = p
p = p→right
F
Display, "Already exist"
return

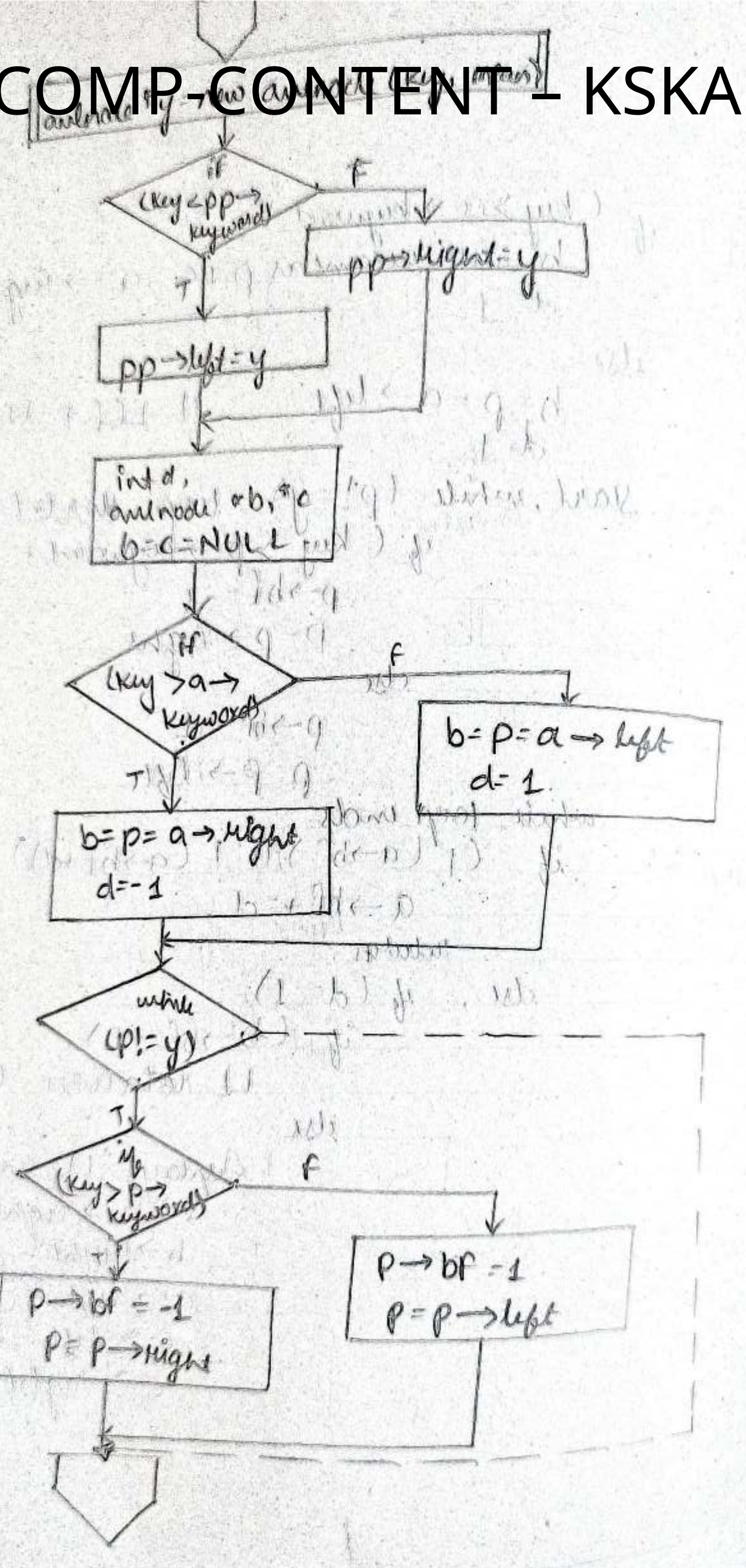
Page No.
Date

Step 6:- if (key > a→ keyword)
    b will remain as p i.e. a → right //Right Heavy
    d=-1
else
    b=p=a→ left . // LEFT HEAVY
    d=1 ;

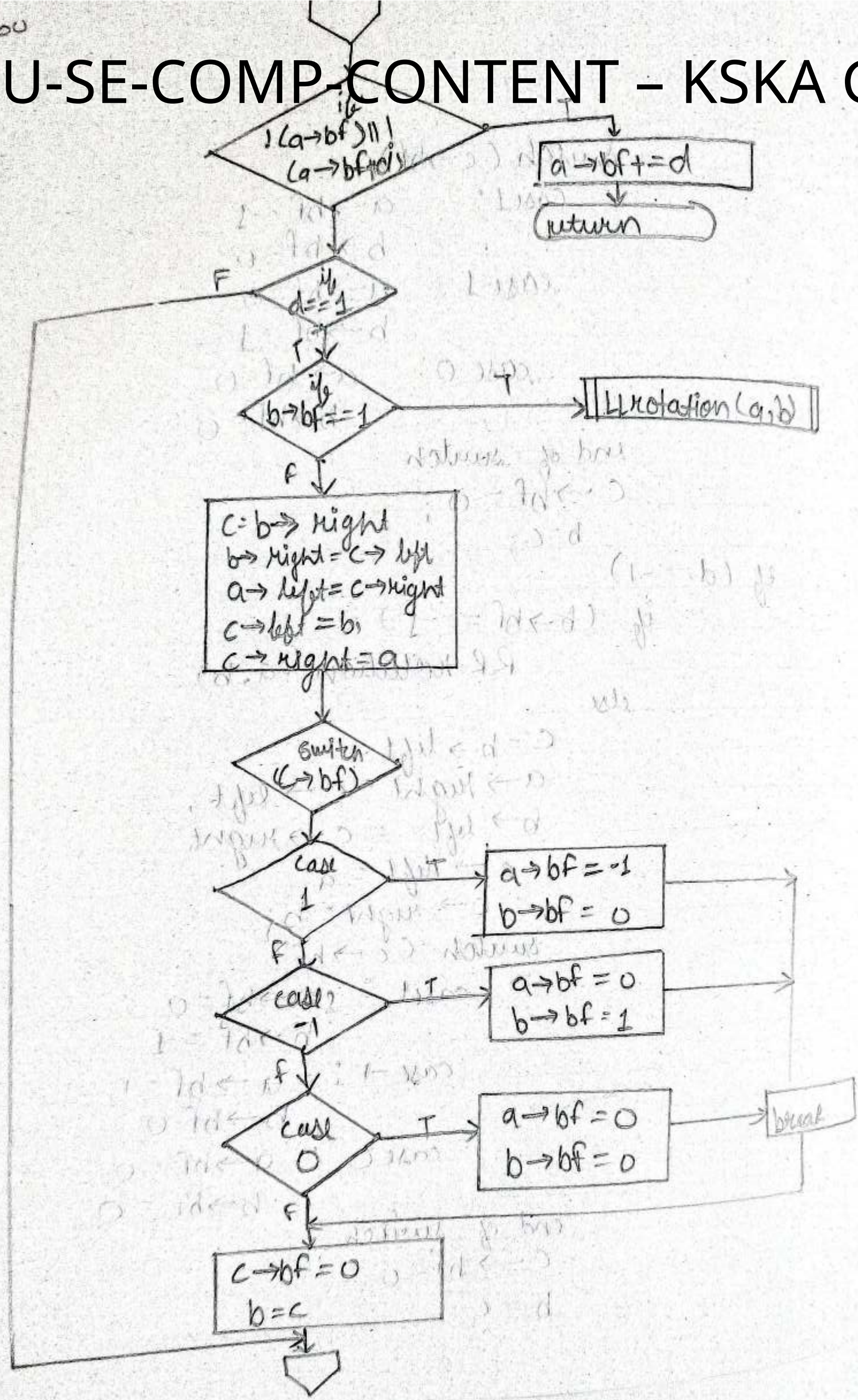
Step 7:- Start, while (p!=y) loop started
    if (key > p→ keyword)
        p→bf=-1
        p=p→ right ;
    else
        p→bf = 1
        p=p→ left
while loop ends.

Step 8:- if (! (a→bf) || ! (a→bf+d))
    a→bf +=d ;
    return
else , if (d==1)
    if (b→bf==1)
        LL rotation (a,b)
    else
        Display "LR rotation"
        c=b→right
        b→right = c→ left .
        a→ left = c→right .
        c→ left = b ;
        c→ right = a ;

198


if (key<pp→keyword)
F
T
pp→right=y
pp→left=y
int d,
avlnode *b,*c
b=c=NULL
if (key>a→keyword)
F
T
b=p=a→left
d=1
b=p=a→right
d=-1
while (p!=y)
T
if (key>p→keyword)
F
T
p→bf=-1
p=p→right
p→bf=1
p=p→left

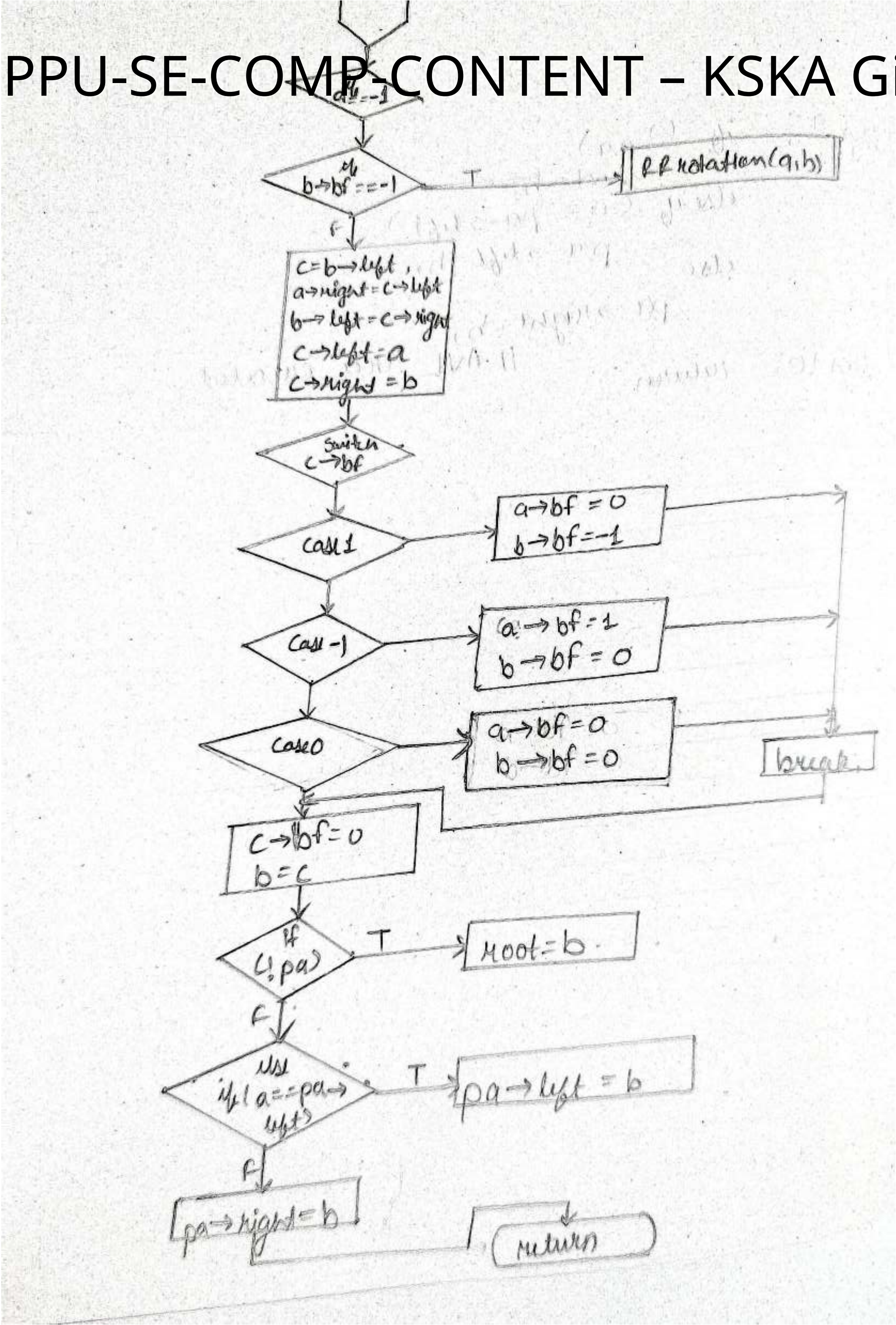
```
switch (c->bf)
    case1:      a->bf =-1
                b-> bf=0
    case-1:     a->bf=0
                b->bf =1
    case 0:     a-> bf=0
                b->bf=0
  end of switch
  c->bf = 0;
    b=c;
if (d==-1)
      if (b->bf ==-1)
            RR rotation (a,b)
      else
            c=b->left
            a-> right = c-> left;
            b-> left  = c->right;
              c-> left = a;
               c-> right = b;
            switch (c->bf)
                case1 :   a->bf=0
                          b->bf =-1
                case -1 :   a->bf =1
                            b->bf=0
                case 0 :    a->bf = 0
                            b->bf = 0
        end of switch
        c-> bf= 0
        b = c.
```


if
!(a→bf)|| !(a→bf+d)
T
a→bf+=d
return
F
if d==1
T
if b→bf==-1
T
LLrotation(a,b)
F
c=b→right
b→right=c→left
a→left=c→right
c→left=b;
c→right=a
switch (c→bf)
case 1
T
a→bf=-1
b→bf=0
F
case -1
T
a→bf=0
b→bf=1
F
case 0
T
a→bf=0
b→bf=0
break
F
c→bf=0
b=c

Page No.
Date

Step 9 :- if (!pa)
root = b;
else if (a == pa → left)
pa → left = b;
else
pa → right = b;
// AVL tree created.

Step 10 :- return.


T
F
RR rotation(a,b)
c=b→left ,
a→right = c→left
b→left = c→right
c→left = a
c→right = b
switch c→bf
case 1
a→bf = 0
b→bf = -1
case -1
a→bf = 1
b→bf = 0
case 0
a→bf = 0
b→bf = 0
break.
c→bf = 0
b = c
if (!pa)
root = b.
else if (a==pa→left)
pa→left = b
pa→right = b
return

Page No.
Date

descending() function

Step 1 : If (root)
call descending (root → right)
Display, root → keyword, root → meaning
call descending (root → left

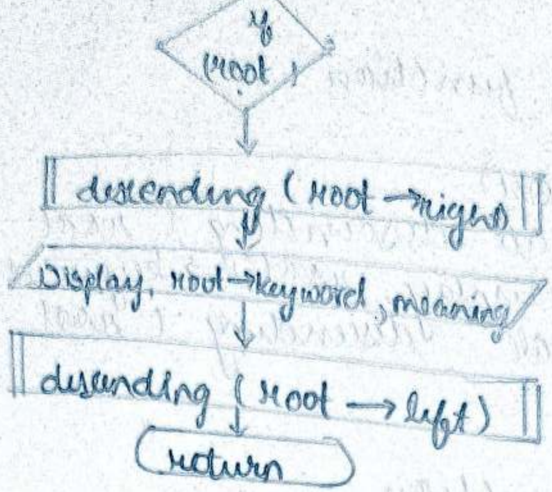
Step 2:-return

accept() function

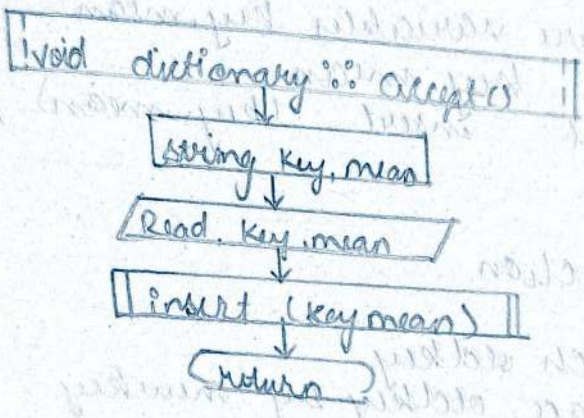
Step 1 :- Declare variables key, mean.
Step 2 :- Read key, mean.
Step 3 :- call insert (key, mean) function

7) update() function

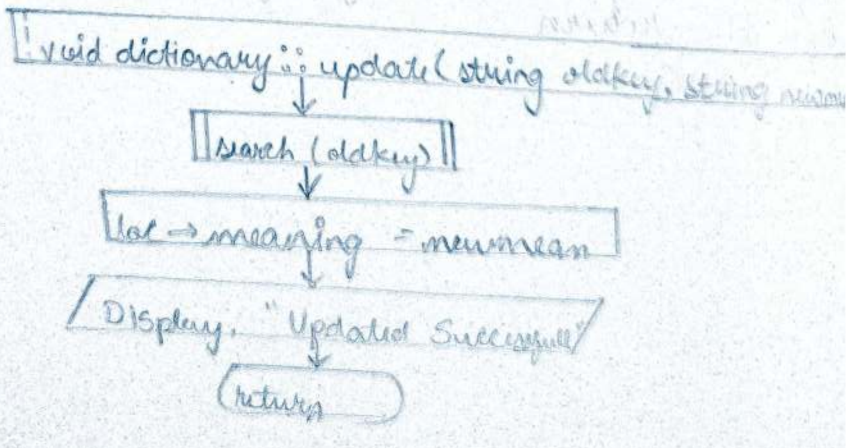
Step 1: search oldkey
Step 2: Replace oldkey by newkey.
Step 3: return



accept()

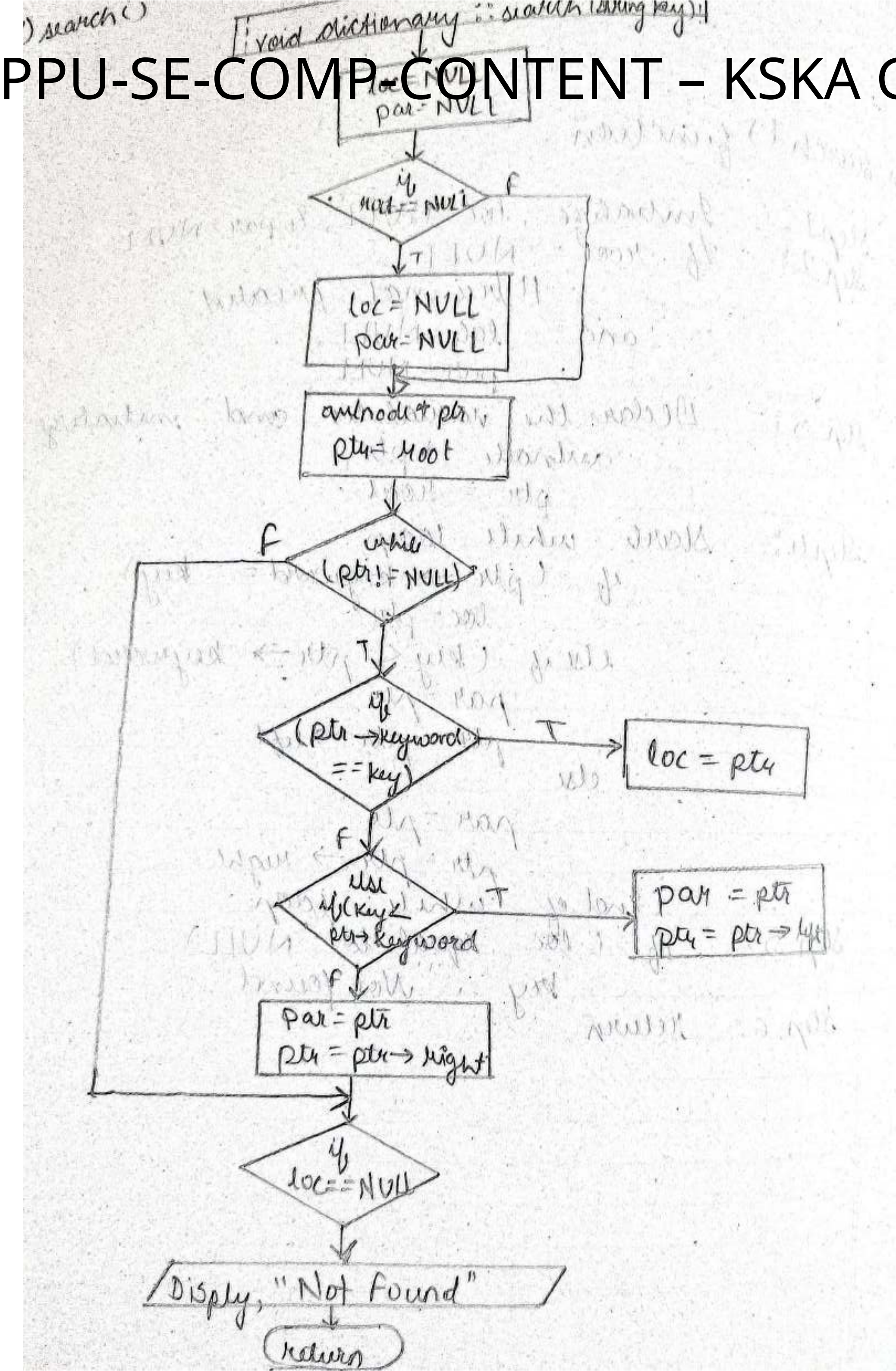


update()



c) search() function

Step 1 :- Initialize , loc = NULL, & par = NULL
Step 2 :- If root == NULL
//Tree not created
and, loc = NULL,
par = NULL,
Step 3 :- Declare the variables and initialize pointers
avlnode * ptr;
ptr = root ;
Step 4 :- Start while loop
if (ptr → keyword = = key)
loc = ptr
else if (key < ptr → keyword)
par = ptr,
ptr = ptr → left
else
par = ptr
ptr = ptr → right
(end of while loop
Step 5 :- if (loc equal to NULL)
key , "Not found"
Step 6 :- return


void dictionary :: search (string key)
loc = NULL
par = NULL
if root == NULL
T
F
loc = NULL
par = NULL
avlnode * ptr
ptr = root
while (ptr != NULL)
F
T
if (ptr → keyword == key)
T
loc = ptr
F
else if (key < ptr → keyword
T
par = ptr
ptr = ptr → left
F
par = ptr
ptr = ptr → right
if loc == NULL
Disply, "Not Found"
return

9) LL rotation () function

Step 1: Transfer the node values as:
a→left = b→right
b→right = a
a→bf = b→bf = 0

Step 2: return

10) RR rotation () function

Step 1: Transfer the values as
a→right = b→left
b→left = a
a→bf = b→bf = 0

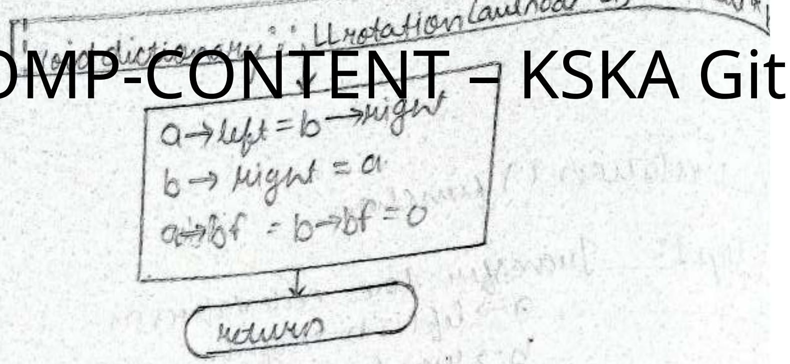
Step 2: return

11) inorder () function

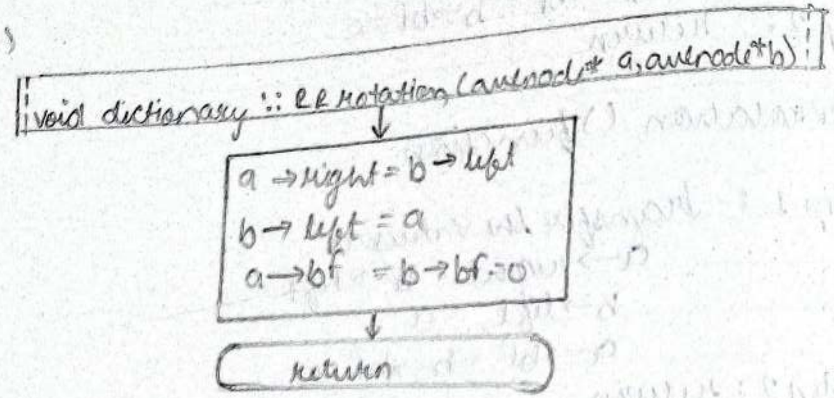
Step 1: if (root)
call inorder (root→left)
Display keyword and meaning
inorder (root → right)

Step 2: return

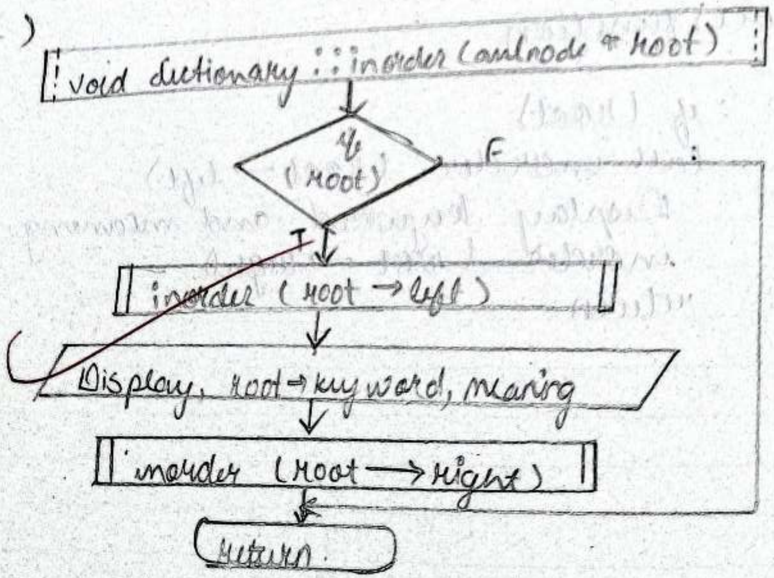
9) LLrotation( ) 208



10) RRrotation( )



11) inorder( )



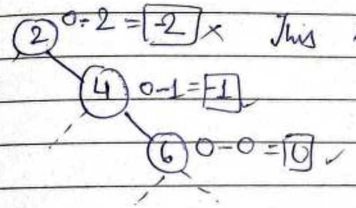
Questions

1) What is an AVL tree? Explain with the help of example. What are the applications of AVL tree.

AVL Tree is invented by GM Adelson - Velsky and EM Landis. AVL tree can be defined as height balanced binary search tree in which each node is associated with a balance factor which is calculated by subtracting the height of its right sub-tree from that of its left sub-tree.

→ [Height of left subtree - Height of right subtree] ≤ 1 = Balance factor

eg:-



This is not avl tree as balance factor = -2

Balance factor should be -1, 0, 1 only

Applications of AVL tree.

It is applied in corporate areas and storyline games.

Software that needs optimized search.

It is used to index huge records in a database and also to efficiently search in that

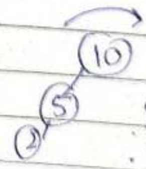
Highly applicable in sets and dictionaries.

Q) What is the difference between OBST, Huffman's tree and AVL tree

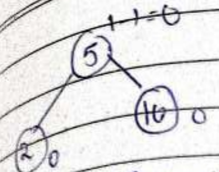
OBST	Huffman	AVL
- stands for Optimal Binary Search Tree.	- Used for Huffman Coding	- It is type of self balancing BST.
- It is a BST that minimizes average search time for given set of keys	- It is variable length prefix coding algorithm	- It maintains a balance factor for each node i.e difference bet^n left and right node
- It is constructed based on frequency or probability of accessing each key	- Built based on frequency of occurance of characters in the i/p data, with frequently occuring characters having shorter codes	- It automatically performs rotations and adjustment to ensure that balance factor remains within range.

Explain single rotation and Double rotation with example.

Single Rotation :- If

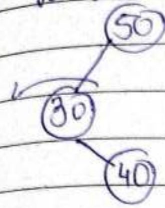


It is not AVL
∴ rotate

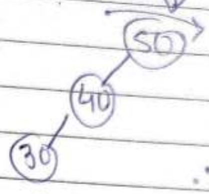


After single rotation it formed AVL tree

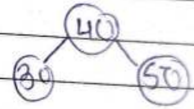
3) Double Rotation



is the tree which is not AVL
∴ we will make 1st rotation



Still not a AVL tree
∴ 2nd rotation



(This is perfect AVL tree).

4) Write down time and space complexity of AVL tree.

a. The space complexity of AVL Tree is O(n) in average and worst case.

- The time complexity of AVL Tree is O(log n)

Conclusion

Hence successfully implemented and understood AVL tree data structure.