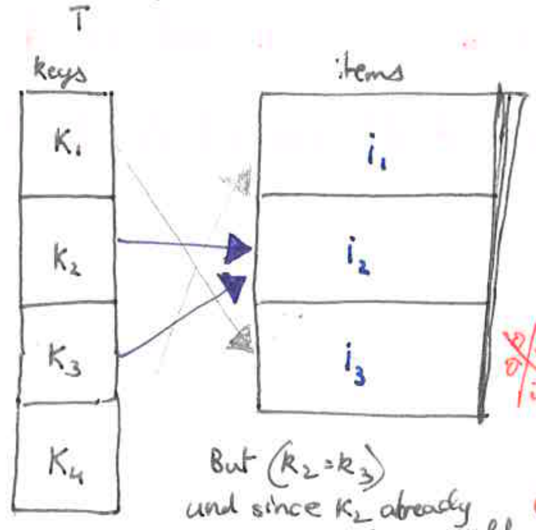
def: Collision Resolution : when a record to be inserted to a hash table map to an already occupied slot in the map, a collision occurs.

1. Chaining:

To manage this, we can use a technique called chaining, which "sets up an array of ~~keys~~ links, to lists of items with the same key."

Hash map

eg

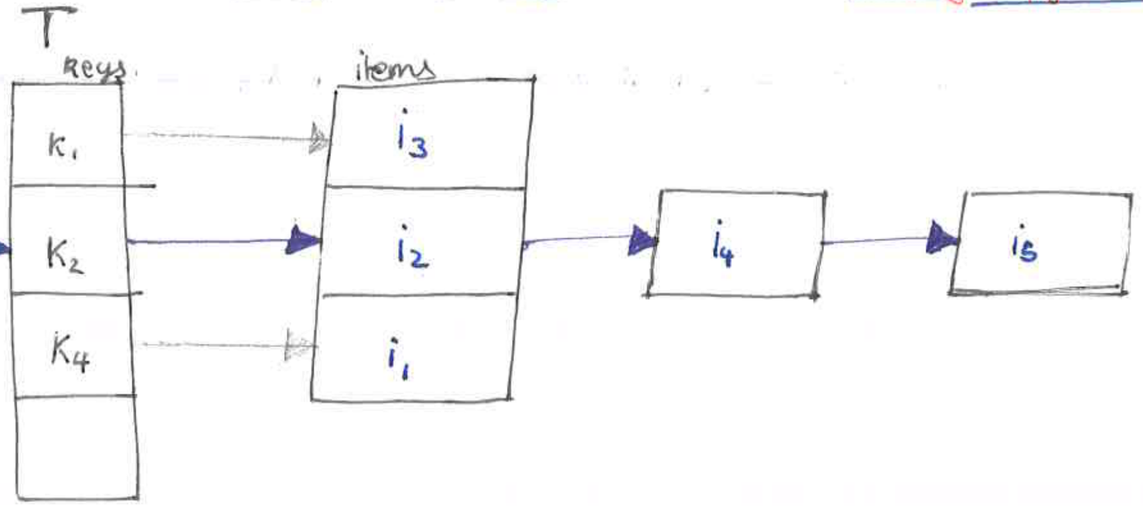


Both K_2 & K_3 map to i_2.
This is a collision.

But ($K_2 = K_3$) and since K_2 already maps to i_2, we would need K_3.

~~0/10~~

Note: not quite. the idea is for items that can be grouped by keys, should be grouped by keys. You simply link the key in question to a linked list of all items that map to the key in question. see eg. pg 8 Lecture 8



this is an illustration of chaining

$\alpha = n/m$
⇒ ↳ load factor of a hash tab

m slots in the hash table
holding n elements

Imp %

2. Open Addressing..

→ All elements are stored in the hash table such that $n \leq m$

→ Insertion systematically probes the table until an empty slot is found.
• i.e. higher chance of the table filling up.

→ Modify the hash function to take the probe number i as the second parameter (depends on both the key and the probe number).

$$h : U \times \underbrace{\{0, 1, \dots m-1\}}_{\text{probe num}} \longrightarrow \underbrace{\{0, 1, \dots m-1\}}_{\text{slot num}}$$

↳ hash fn: ⓗ
↳ determines the sequence of slots examined for a given key.

→ for a given key k;

$\langle h(k,0),\ h(k,1), \dots h(m-1) \rangle$ a permutation of $\langle 0, 1, \dots m-1$

There are two types/methods of open addressing:

- Linear Probing
  (i) if the current location is used, try the next table location.
  $h(k,i) = (h'(k) + i) \bmod m$ ~~(pseudo-code | notes)~~

Pseudo code:

% %

Imp

```
Linear Probing Insert (k)
    if (table is full) return error.
    probe = h(k)
    while( table[probe] is occupied):
        probe = (probe + 1) mod m
    table [probe] = k
```

linear probing (cont'd)

(ii) Uses less memory than chaining.
↳ you don't have to store all the pointers/references if you use linear probing.

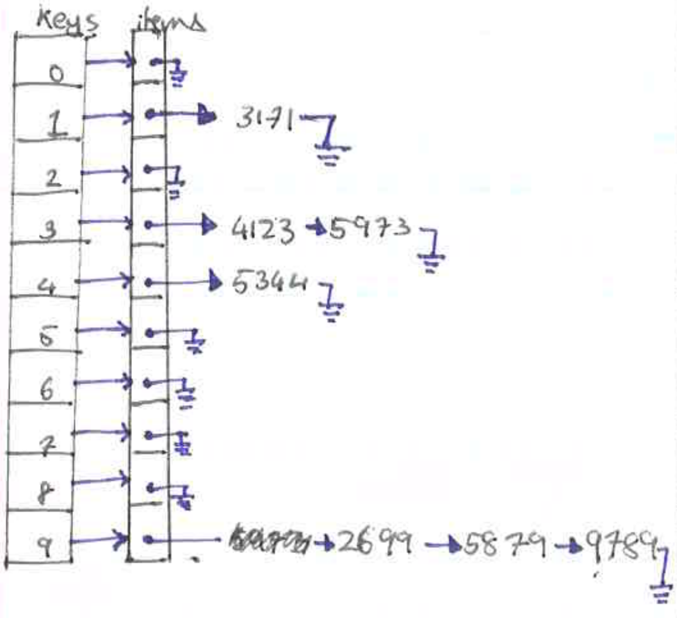
(iii) It is, however, slower than chaining.
↳ it depends to heavily on ~~[illegible]~~ long the table is and has to traverse the whole table at worst-case.

eg. Show the hash tables for the following using both Chaining & linear probing.

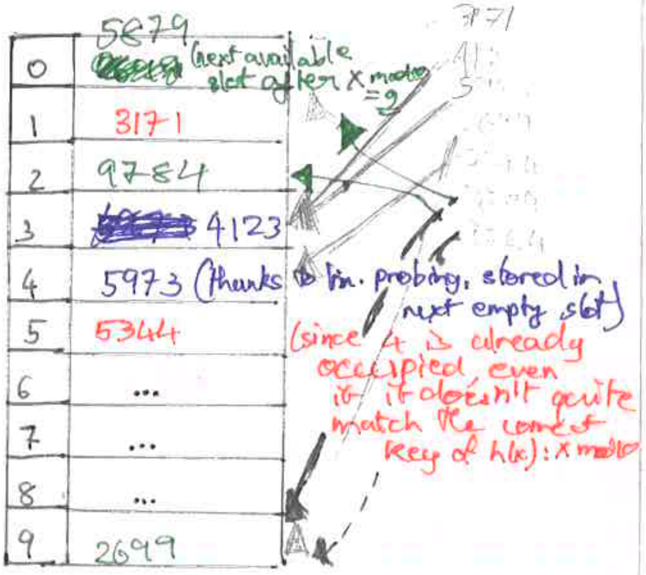
input: {3171, 4123, 5973, 2699, 5344, 5879, 9789}

using the hash function: $h(x) = x \bmod 10$.

Chaining



Linear Probing



0	5879 ~~[illegible]~~
1	3171
2	9784
3	~~[illegible]~~ 4123
4	5973
5	5344
6	...
7	...
8	...
9	2699

(next available slot after x mod 10 = 9)

(Thanks to lin. probing, stored in next empty slot)

(since 4 is already occupied even if it doesn't quite match the correct key of h(x): x mod 10)

- Double Hashing note: (tends to distribute keys more uniformly than linear probing)

$$h(k,i) = (h_1(k) + i \cdot h_2(k)) \bmod m.$$

Pseudo code:

```
DoubleHashingInsert (k)
    if (table is full) return error
    probe  = h1(k)
    offset = h2(k)

    while (table[probe] is occupied)
        probe = (probe + offset) mod m.

    table [probe] = k
```

✱ ✱ IMP

eg Input {3171, 4123, 5973, 2699, 5344, 5879, 9789}

hash fn $h_1(x) = x \bmod 10$

- For double hashing: the second hash fn is

$$h_2(x) = \lfloor x/10 \rfloor \bmod 10$$

Double hashing ↓

0	5973
1	3171
2	
3	4123
4	5344
5	
6	5879
7	9789
8	
9	2699

since slot is unoccupied, 3171 is easily inserted

5973 first collision

③ 5973 mod 10

597 mod 10

⑦

probe + offset results in these slots.

probe = (probe + offset) mod 10

for 5973 = (3+7) mod 10

= 0

double hashing (cont'd)

IMP

Some thinking
equired
before you start
double hashing
during coding

(i) $h_2(K)$ must be relative prime to m to guarantee that the probe sequences is a full permutation of $\langle 0, 1, \dots, m-1 \rangle$

→ let m be prime such ~~an~~ that $1 < h_2(K) < m$

→ Choose $m = 2^d$ and h_2 to always produce an odd number > 1.

Expected number of probes:

1. load factor $\alpha < 1$ for probing (i.e. you need empty slots in the table)
2. ~~Assu~~ Assuming that uniform hashing was used, analysis of probing gives us:

	unsuccessful	successful
chaining	$1 + \alpha$	$1 + \alpha$
probing	$1/(1-\alpha)$	$(1/\alpha) * \ln(1/(1-\alpha)$

HARD TO ~~GU~~ GUARANTEE SIMPLE UNIFORM HASHING..

i) Ensure you avoid any pitfalls, because in practice hashes are very ~~ee~~ useful.

ii) Should compute quickly pls

Hash function: • Division method Pg 11 } Lecture 9
• Multiplication method. Pg 12 } notes

END OF HASHING

TREES

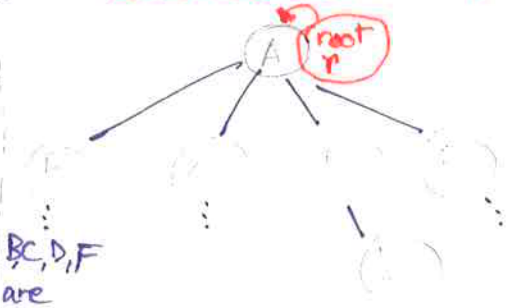
Why did we need trees?

- the linear search time of a linked list (obv $O(n)$) is prohibitive ...
- If we can reduce run time of searching, inserts, delete etc down to $O(\log n)$ we'll obvious save time (:O)

def: A tree is a collection of nodes.

- the collection can be empty
- (recursive definition) If not empty, a tree consists of a "distinguished" node (aka the root node $\underline{r}$) and some non-negative number of non-empty subtrees $\boxed{T_1, T_2, \dots T_k}$ each of whose roots are connected by a directed edge from $\underline{r}$.

Terminology related to trees:



(i) Path: a sequence of nodes $n_1, n_2, \dots n_k$ such that n_i is the parent node of n_{i+1} for $1 \leq i \leq k$.
(i.e. path nodes must be in increasing depth).

- B,C,D,F are "siblings"
- E is a leaf
- A is the "parent node" of B, C, D, & F
- B, C, D & F are "children" of A.

(ii) Length: number of edges on the path

(iii) Depth of a node:
- length of the unique path from the root to that node.
- The depth of a tree = the depth of the tree's deepest leaf.

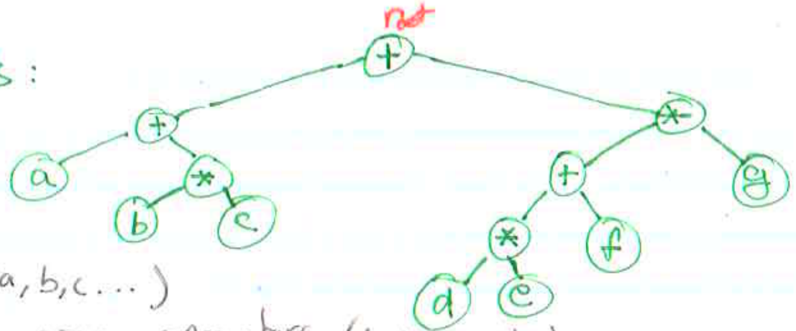
(iii) Height of a node:
- length of the longest path from that node to a leaf.
- all leaves ~~[illegible]~~ are at height 0. (zero).
- The height of a tree = height of its root.

eg: the unix directory's structure is an example of a tree.

BINARY TREES :

def : A tree in which no node can have more than two children is known as a Binary Tree.

eg EXPRESSION TREES :



- The tree's leaves are operands (a, b, c ...)
- Non-leaf nodes are operators (+, * ...etc)
- If an operator was non-binary (i.e. accepted more than two operands, then the tree would not be a binary Tree anymore.)

Traversing trees :

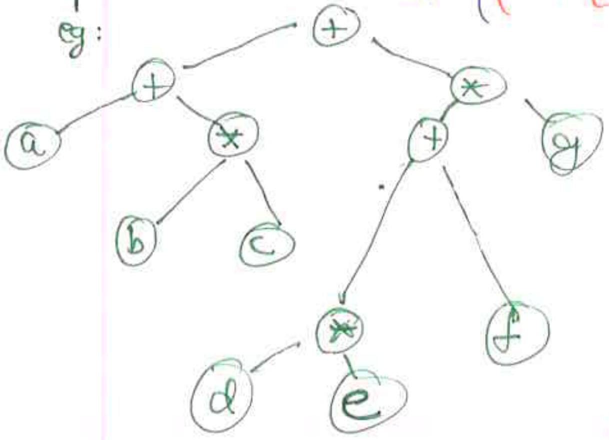
def: Traversal is the process of visiting every node once (and ONLY once?)

→ 3 recursive techniques for traversing trees :
- the left subtree traversed recursively.
- the right " " ".
- the root is visited.

3 orders of traversal:

- Pre-order traversal:
  - → Node , Left , Right $\boxed{\text{IMP}}$
  - → exp: (+ + a * b c * + * d e f g) → aka pre-fix expression
- Post-order traversal:
  - → Left, Right , Node
  - → exp: ~~abc*~~ ([abc*)+] [de*f+g*] +)
    aka post-fix expression
- In-order traversal:
  - → Left, Node, Right
  - → exp: ((a+(b*c))+(((d*e)+f)*g))
    aka infix expression (normal)

eg:



IMP <u>Binary Trees ADT</u>

- BinTree ADT:
  - → Accessor functions:
    - key() : int
    - parent() : BinTree
    - left() : BinTree
    - right() : BinTree
  - → Modifier functions:
    - setKey(K: int)
    - setParent (T: BinTree)
    - setLeft (T: BinTree)
    - setRight (T: BinTree)

end lecture 10

(i) • Dictionary ADT : Search (S, K) Insert (S, x) Delete (S, x)

(i) • Ordered Dictionary ADT : (++ all above methods) min (s) max(s)

returning a pointer to the element in S with the smallest / largest key respectively.

Successor (S, x) Predecessor (S, x)

↳ obvious, but both functions return pointers to the next element or the previous element respectively for a given ⓧ or NIL if the condition isn't met.

i.e • successor call on the last element returns NIL
 • etc..

(ii) • Unordered list : O(1) : → insert, delete
O(n) : min, max, predecessor, successor, search

(ii) • Ordered list : O(n) : searching, insert, delete
O(1) : [illegible]

IMP ※ Binary Search

• [illegible] for the item being searched [illegible]

• Binary searches run in O(lgn) time.

• Search → O(lg n)
• Insert, Delete → [illegible]
• min, max, [illegible])

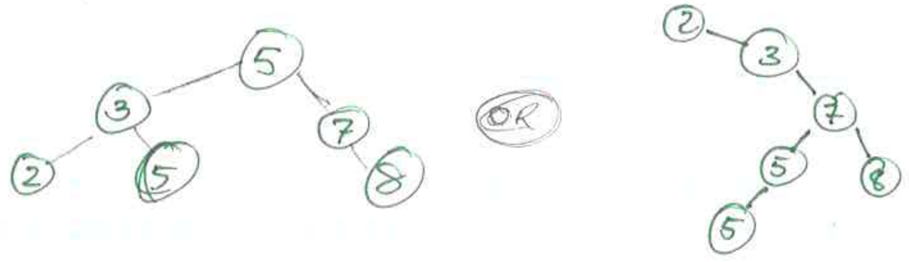
24/June/2019

BINARY SEARCH TREES

IMP %

def: A binary search tree is a binary tree T such that:
- each internal node stores an item (K) of a dictionary
- keys stored at nodes in the left subtree are less than or equal to K.
- keys stored at nodes in the right subtree are greater than or equal to K.

eg. Eg. sequence: 2, 3, 5, 5, 7, 8



Tree T:
- key() → element (at root)
- left() → Tree
- right() → Tree

% Sea~~rching in a~~ BST

- To find an element with key k, in a tree T:
  - compare K with T.key()
  - if $K < T.key()$ search for K recursively in T.left()
  - else, search for k recursively in T.right().

→ can also be iteratively search. [illegible] [illegible] optional [illegible]y.

IMP

BST

⁂ Recursive BST search - pseudocode.

```
BSTSearch (T, k):
    if T == NIL: return NIL
    if K == T.key() : return T.
    if k < T.key() :
        return BSTSearch(T.left(), k)
    else :
        return BSTSearch (T.right(), k)
```

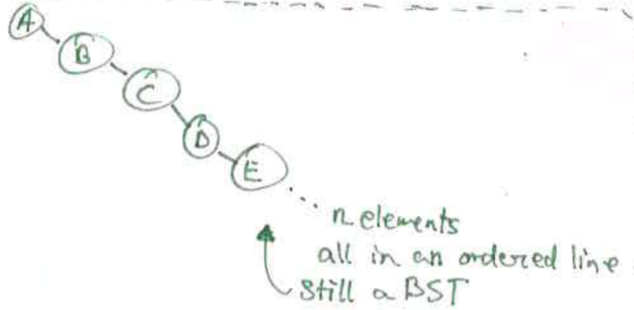
⁂ Iterative BST search - pseudocode

```
BST Search (T, k):
    let x = T                  ([illegible])
    while ((x != NIL) && (k != x.key())):
        if (k < x.key()) :
            x = x.left()       ([illegible]
                                  [illegible] BST node , if k < x.key())
        else :
            x = x.right()
    return x
```

Analysis of BST Searching:

- Worst-case run-time:

eg



If all 'n' elements in a BST are arranged in some permutation of the structure shown on the left, then the worst-case running time is $O(n)$

i.e worst-case BST search = $O(n)$

But in the above example: the height of the tree (h) = the number of elements (n) in the BST.

⇒ In a more "full" BST, it may be noted that worst-case run-time of a BST search is actually $O(h)$ where h = height of the tree

BST Functions:

- BST Successor: Given x; find the node with the smallest key greater than x.key().

2 cases:

CASE 1: • if right subtree of x is non-empty.

def case 1: • Successor: it is the leftmost node in the right subtree

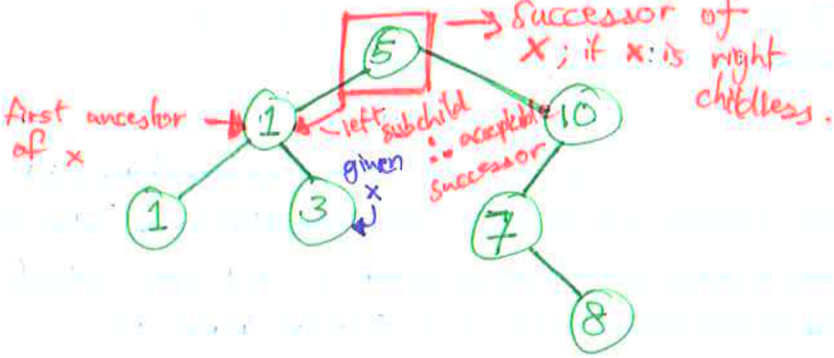
• Can also be found by returning the treeMinimum(x.right())

CASE 2: • if the right subtree of x is empty.

def case 2: • Successor: it is the lowest "ancestor" of x, whose left child is also an ancestor of x.

↳ WHAT THE FUCK DOES THAT EVEN MEAN ?? (see example image) case 2: Successor (cont'd)

eg Observe the following BST. Given (x = 3) when BSTSuccessor (T, x) is called.



ans By the definitions of a node's successor, we can see that x has no child to the right. Therefore, we employ definition 2 of the successor of a node.

i.e The lowest ancestor (of x) that is an ancestor to another ancestor of x

- the lowest "grand-ancestor" of x (if its grand-child x belong to its left subtree), is its successor if x has no right child.

Thus:
- first ancestor of x = 1 (i.e: x's parent node).
- the sucessor of x, is the lowest ancestor of x's parent = 5 (in the example above)

BST Successor Pseudocode

case 1: if x has a non-empty right subtree

→ if implemented; simply return the lowest key in x.right() for case 1

check case 1:

if not case 1, address: case 2

```
BSTSuccessor ( x ) :
    if (x.right() != NIL):   return TreeMinimum (x.right())
    assign y = x.parent()
    while ((y != NIL) && (x == y.~~right~~left())):
        assign x = y
        assign y = y.parent()
    return y.
```

x.parent() → ~~get~~ search for x's parent / aka first ancestor.

(y != NIL) → ensure the parent exists //

(x == y.left()) → ensure x ~~is~~ it's parents left — CHECK WITH SOMEONE!!

once x's parent has been found, we are now attempts to find x.parent()'s ancestor & confirm that x.parent() is the "grand-ancestor"'s left subtree..

Thus the grand-ancestor is x's successor..

Does (x) have to be the right child to its parent??

CHECK THE CODE; IT DOESN'T SEEM CORRECT!

End lecture 11

24/June/2019

25/June/2019

- BST Insertion:

    - takes an element of a tree z (whose left and right children are NIL) and insert it into T.

    - find place in T where z belongs (similar to searching for z.key())

    - i.e. compare z.key() with the root of the tree and find out where z can fit in the tree according to BST properties.

run-time : O(h)   h → height of tree

BST Insertion Pseudocode

```
- Tree Insert (T, z):

•     assign y = NIL
•     assign x = T

•     ~~if z.key() < x.key():~~

•     while (x != NIL):
          • assign y = x

          • if (z.key() < x.key()):
              • assign x = x.left()
          • else:
              • assign x = x.right()
```

```
    z.setParent(y)
    if (y != NIL):
        if (z.key() < y.key()):
            y.setLeft(z)
        else:
            y.setRight(z)
    else:
        assign T = z.
// end
```

Sorting a Binary Search Tree:

```
TreeSort(A):
    assign T = NIL
    for (int i = 1; i <= n; i++):
        TreeInsert(T, BinTree(A[i]))
// sorted BST.
```

↳ Aatish, you called it something else. check against notes.

BST Deletion: · delete node x from a tree T

3 cases when considering deletion:

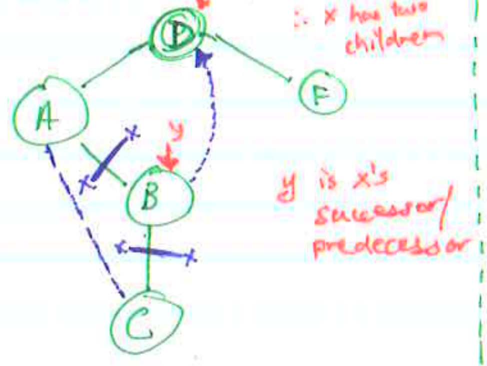
case 1 → if x has no children
case 2 → if x has one child
case 3 → if x has both children

- Case 1: simply remove x, since it has no children and will not really affect other tree elements apart from its parent

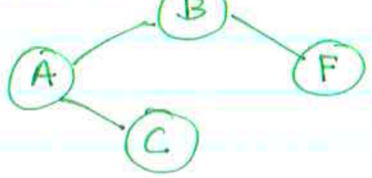
- Case 2: if x has one child, then before deleting x, we make x.parent() point to x's child instead of x itself.

- case 3: if x has 2 children, then:
  - find x's successor (or predecessor) y.
  - remove y (y can ~~hav~~ at most have only one child) (that way, y's parent can then point to y's child instead of y)
  - replace x with y

eg



After deletion:



BST delete Pseudocode:

- Tree Delete (T, z):

```
if ((z.left() == NIL) || (z.right() == NIL)):
    assign y = z
else:
    assign y = tree successor (z)

if (y.left() != NIL):
    assign x = y.left()
else:
    assign x = y.right()

if (x != NIL):
    x.setParent(y.parent())

if (y.parent() == NIL):
    assign T = x

else if (y == y.parent().left()):
    y.parent().setLeft(x)

else:
    y.parent().setRight(x)

if (y != z):
    z.setKey(y.key())

return y
```

Y —— X

25/June/2019

IMP %

BALANCED BINARY SEARCH TREE

→ each node has two children (except the leaves)

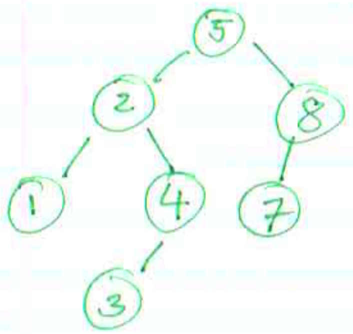
→ the height of the left & right subtrees are equal ??

→ All balanced binary BSTs have to be complete binary trees.

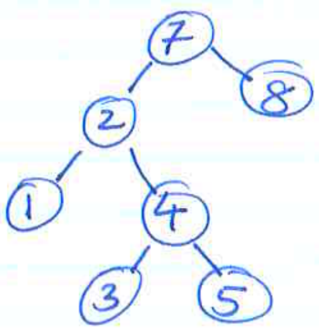
AVL TREE: aka [Adelson - Velskii and Landis Binary Search Tree]

def:
- An AVL tree is a BST in which; for every node in the tree, the height of the left & right subtrees differ by at-most 1.

eg. AVL - Tree



Non-AVL tree



AVL trees can only have "certain" minimum, ~~numbers~~ tree heights based on how many nodes their tree contains.

for example:

if n = 1 , the tree's height = 0

if n = 2 , the tree's height = 1

if n = 3 , the tree's height = 1

similarly n = 4 , height = 2
=> if, n = 5 , height = 2
=> if, n = 6 , height = 2

if n = 7, height = 3 etc, etc. See example tree's below:

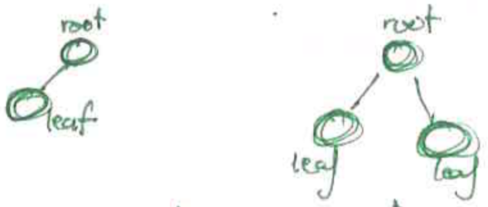
---

n = 1 tree:

height = 0

---

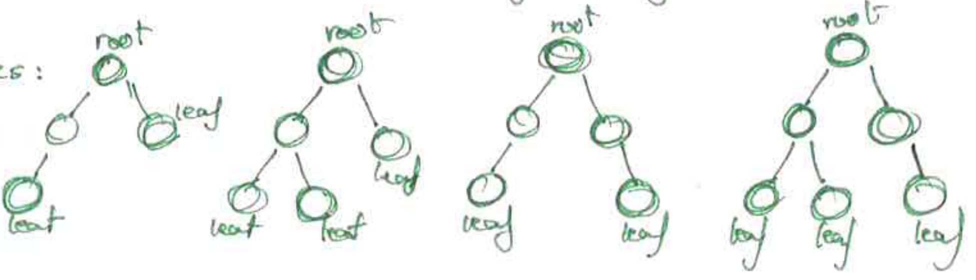
n = 2 trees:
n = 3,



height = 1

---

n=4 trees:
n=5
n=6



etc, etc.

$h = O(\log N)$

=> most operations on an AVL tree will take $O(\log N)$ time.

- When inserting a node to an AVL tree:
  → insert a node as you would for a regular BST

  → if the AVL property gets violated, ~~remove~~ relocate the newly added node to restore the AVL property.

  → Rebalance the tree at the deepest node on the subtree to guarantee that the entire tree satisfies the AVL condition as expected.

IMP

IF α is the ~~node~~ node/tree that must be rebalanced:

Case 1: an insertion of a node into the left subtree of the left subchild of α.

Case 2: an insertion of a node into the right subtree of the left subchild of α

Case 3: an insertion of a node into the left subtree of the right subchild of α

Case 4: an insertion of a node into the right subtree of the rig
subchild of α.

def: ROTATIONS

- the rebalance of an AVL tree can be done by "rotation"

how & when to rotate: 1)• Insertion occurs on the "outside" i.e (the left-most / right-most) nodes and is fixed by "single rotation" of the tree.

2)• Insertion occurs on the "inside" nodes and is fixed by "double-rotation" of the tree.

INSERTION ALGORITHM

→ Add new node like in a regular BST

→ Trace the path from *newly inserted leaf to the root*

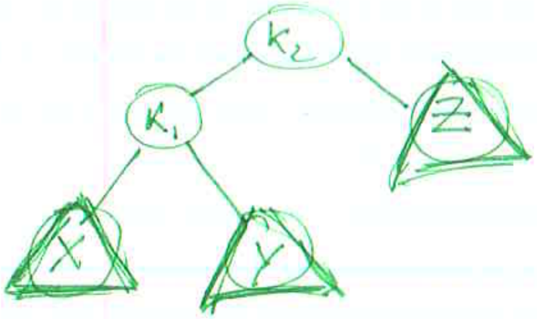
- At EACH encountered node, x; check if heights of left(x) & right(x) differs at most by 1.
  - → if yes, proceed to parent(x) ~~[illegible]~~
  - → if not, restructure by performing either a single or double rotation.

AVL properties: • left subtree $\overset{\text{right}}{=} \pm 1$
•

Single rotation for case 1:

~~AVL prop.~~

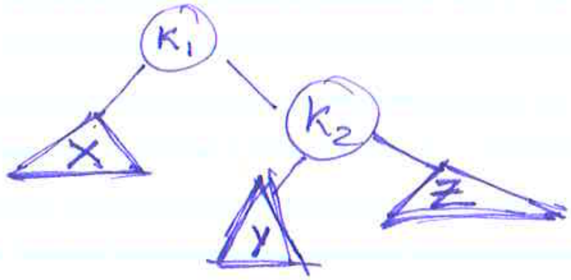
eg violation:



if ~~x~~ subtree X gets an insertion, then at node ⓀK2, height of left(K2) ≠ height of right(K2)

i.e. violates AVL property.

Solution: single rotation



eg Insert 3, 2, 1, 4, 5, 6 to an AVL tree, sequentially: