- BINARY TREES -

DATE:

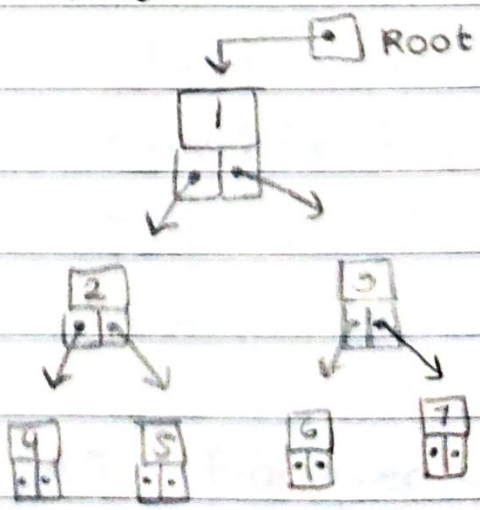
*Binary Tree

- a tree data structure where each node has at most 2 children (left/right child)
- a tree can be:
  - > empty
  - > every node has:
    - no child
    - left child only
    - right child only
    - BOTH
- usually represented by : pointer to root node
- each node has: data
  pointer to L child
  pointer to R child

*Implementation

1. Dynamic Node Representation (Linked)
   - has pointers to L/R child

```
struct node {
    int data;
    struct node * LEFT;
    struct node * RIGHT;
} Tree;
```



ATE:

2. Array Representation (sequential)

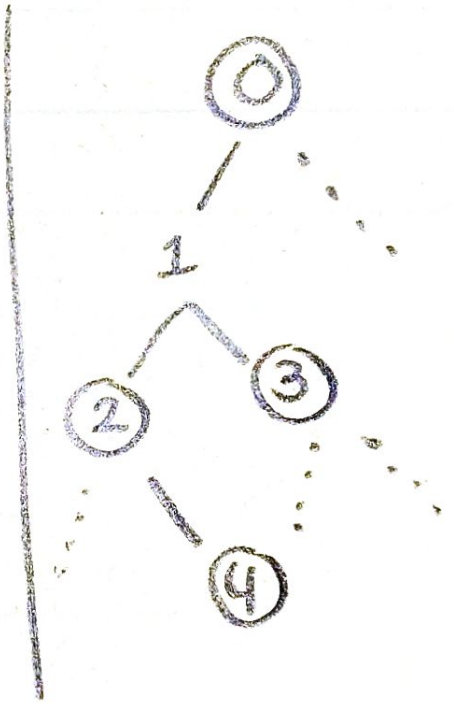
- array indexes: value

elem: left child, -1 if non-existing

right child

Tree T

	L	R
0	1	-1
1	2	3
2	-1	4
3	-1	-1
4	-1	-1



DATE:

* Applications

- codes 1 & 2

	1-bit	2-bit	3-bit
CODES	0	00	000
	1	01	001
		10	011
		11	111
			...

* 0 & 1 = possible values

all possible combinations = code

* Fixed-Length Code

- takes longer
- how many bits involved are known
- needs to determine where code for one symbol ends & begins

* Variable-Length Code

- code length differs
- shorter / uses less no. of bits
- maps different symbols to codewords with variable lengths
- used for data compression
- requires: Prefix Property

* Prefix Property

- no code should be the prefix of another code
- ensure unique code
- distinguishes where one code ends & the next begins

* Average Code Length

Σ (# of bits) * probability of occurence

Probability	Code 2	ACL
.12	000	0.36
.40	11	0.8
.15	01	0.3
.06	001	0.24
.25	10	0.5
		2.20

ACL = 2.20 //

NO:
DATE:

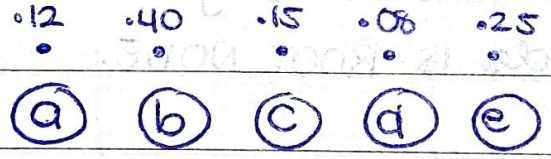
- HUFFMAN's ALGORITHM -

* Huffman's Algorithm

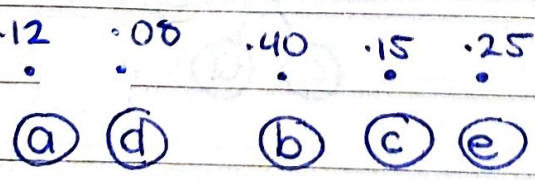
- assigns variable-length codes to input characters
- length of code is based on frequency of corresponding characters
+ builds a Huffman Tree from input characters
+ traverse thru tree & assign codes

* Steps

(1) Create a leaf node for each unique character. Value of frequency field (probability) is used to compare two nodes.



(2) Extract TWO NODES w/ minimum frequency.



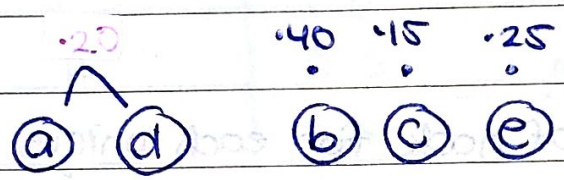
DATE:

(3) Create new internal node with frequency = to SUM of the two nodes frequencies.

Make L child = 1st extracted node

R child = 2nd extracted node

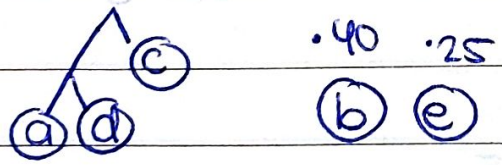
Merge a & d



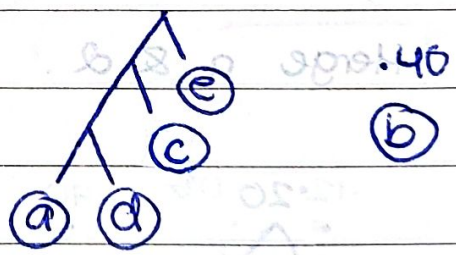
(4) Repeat #2 & #3 until only one node left.

Remaining node is ROOT NODE.

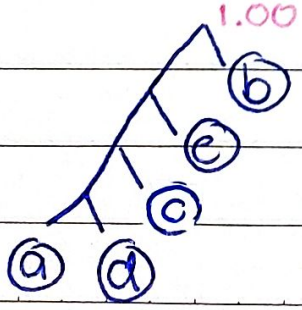
Merge a,d with c



Merge a,d,c with e

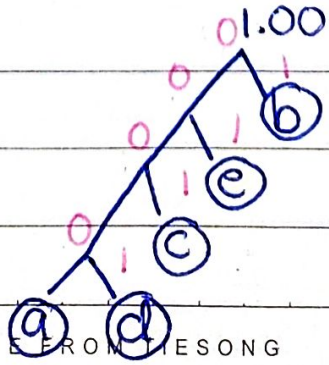


Merge a,d,c,e with b



(5) Label : 0 Left

1 Right



*Traverse thru tree & label / assign code.

DATE:

Character	Code
a	0000
b	1
c	001
d	0001
e	01

* Average Code Length

Probability	Huffman's	ACL
.12	0000	0.48
.40	1	0.40
.15	001	0.45
.08	0001	0.16
.25	01	0.50
		2.15

ACL = 2.15 //

* faster!

NO:
DATE:

- BINARY SEARCH TREE -

*Binary Search Tree

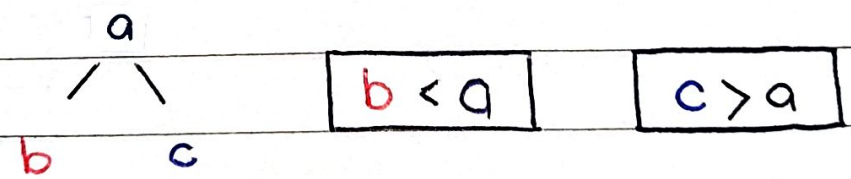
- nodes are labeled with elements of a SET
- so, elements are UNIQUE
- each node has a maximum of 2 CHILDREN (L/R)
- elements are ordered in linear order

< > =

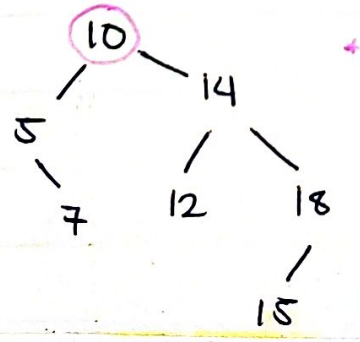
strcmp()

* BST Property

" All elements stored in the left subtree of any node x are all less than the element stored at x, and all elements in the right subtree of x are greater than the element stored at x. "

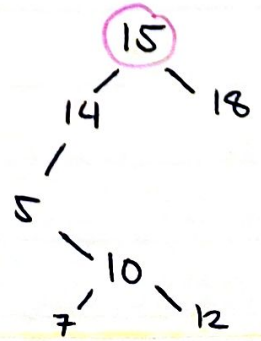


SET A = {10, 5, 14, 12, 7, 18, 15}



* First num is ALWAYS the root.

SET B = {15, 14, 5, 10, 18, 7, 12}



*Running Time

- Member ()

  $O(N)$

  if skewed right,

  just like linked list

- Insert, Delete, Min, Max, Member

  $O(\log_2 N) = x \rightarrow 2^x = N$

  where x is running time

  N is no. of elements

* $O(\log_2 N)$

If tree was COMPLETELY BALANCED:

"Running time would increase by 1 each time the no. of elements doubles."

Running Time #

X	N
1	2
2	4
3	8
4	16
5	32

$2^x = N$

* Avoid $O(N)$ as much as possible!

* Worst case is reaching the leaf level!

*Height Balance

- if the tree's left subtree and right subtree has a maximum height difference is 1.

* Balanced using AVL by performing rotations.

* Implementations

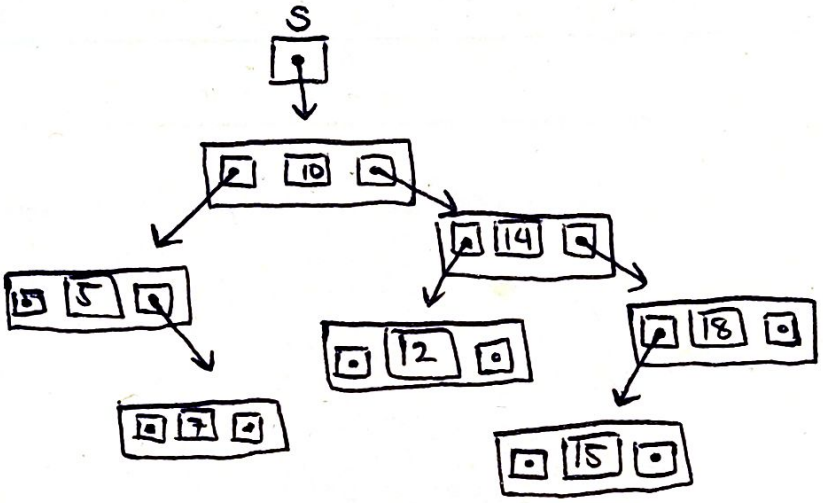
1) Pointer Implementation

- each node will have a pointer to its LEFT & RIGHT child

```
typedef struct cell {
    int elem;
    struct cell *LC;
    struct cell * RC;
} ctype, *BST;

BST S;
```

SET S = {10, 5, 14, 12, 7, 18, 15}



* Operations

1. INSERT(S, key) - given the tree and new node, node is inserted at leaf to (as child) maintain BST property
2. DELETE (S, key) - three possibilities:
   - > Delete node at leaf
   - > Node to delete has 1 child
   - > Node to delete has 2 children

3. MEMBER (S, key) - given tree & key, look for node if existing or not

4. MIN (S) - return minimum key valve found in the tree
- entire tree does not need to be searched

5. MAX (S)

* Traversal Functions

1. INORDER (S) - traverse left subtree
- visit root
- traverse right subtree
- results in ascending order

2. PREORDER (S) - visit root
- traverse left subtree
- traverse right subtree

3. POSTORDER (S) - traverse left subtree
- traverse right subtree
- visit root

NO:

DATE:

- BST OPERATIONS -

// MEMBER () - PN Traversal

1. Start at ROOT.
2. Move to L/R Node depending on element to be searched for:

   L: n < node

   R: n > node

3. Keep traversing element is found OR you reach a leaf node.

> Non-recursive

> Recursive

// INSERT() - PPN Traversal

1. Start at ROOT and look for LEAF Node.
2. Compare new elem n to current node:

   L: n < node

   R: n > node

3. Once leaf is found, insert node either to its left or right, depending on < >

> Non-recursive

> Recursive

// DELETE () - PPN Traversal

Case 1: Node to be deleted is the LEAF.

> simply remove from tree.

Case 2: Node to be deleted has only one child.

> copy the child to the node and delete the node.

Case 3: Node to be deleted has two children.

> Find inorder successor of node.

> copy contents of the inorder successor to the node & delete inorder successor.

REPLACE, not delete

OPTIONS

*immediate predecessor!
- leaf
- LC only

*immediate successor!
- leaf
- RC only

Steps:

1. Start at ROOT.
2. compare new elem to current root :

   L: n < node

   R: n > node

3. Check for possible cases:

   Case 1: delete node

   Case 2: delete L/R node

   Case 3: let root = inorder successor ; recur

> Non-recursive

> Recursive

NO:

DATE:

- RECURSIVE FUNCTIONS -

* Recursive Functions

- a routine / function that calls itself directly or indirectly

* Creating Recursive Functions

- Make sure the function has atleat 1 exit point or base case to ensure that the fn. terminates.
- start the function with the exit point.

* Substitute for For Loops?

Recursive functions creates MULTIPLE ACTIVATION RECORDS for each function call. The process of returning a valve and deallocating the space used by the A.R. This makes the process longer.

> MOST USEFUL: Recursion creates a pointer for EVERY LEVEL of the tree. This allows for traversing downwards AND back upwards.

Applications:

> INORDER, PREORDER, POSTORDER Traversals

> Sudoku

* Loop for one-directional traversing, shorter.
* Recursive for traversing back/upwards.