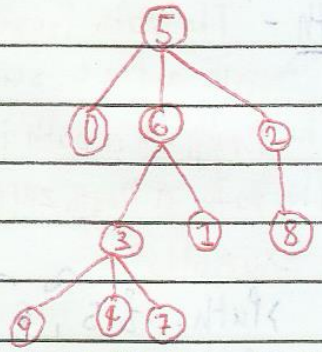
Introduction to Trees

date 04/03/2023

• Trees •

↳ A **tree** is a collection of elements called **nodes**, one of which is distinguished as a **root**, along with the relation ("**Parenthood**") that places a hierarchical structure on the nodes.

• Illustration •



• Exercise •

Based on the Illustration:

1.) What are the nodes of the tree? - 5, 0, 6, 3, 9, 4, 7, 1, 2, 8

2.) What is the root node? - 5

3.) Parenthood:

a.) Parent of 6? - 5

b.) Parent of 7? - 3

c.) Parent of 1? - 6

Note:

- **Recursive** - A func. that calls itself
- **Recursive Def.** - A definition that uses its term to define it

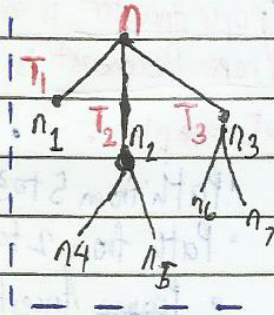
• Recursive Definition of Tree •

1.) A single node by itself is a tree. This node is also the root of the tree

2.) Suppose n is a node and $T_1, T_2, \ldots, T_k$ are trees with roots $n_1, n_2, \ldots, n_k$, respectively.

- We can construct a new tree by making n the parent of nodes $n_1, n_2, \ldots, n_k$.
- In this tree, n is the root $T_1, T_2, \ldots, T_k$ are the Subtrees of the root.
- Nodes $n_1, n_2, \ldots, n_k$ are called the children of node n.

• Illustration •



Introduction to Trees

date _____

* Properties of Trees *

- **Path** - The path from n_1 to n_k is a sequence of nodes $(n_1, n_2, \ldots, n_k)$ in a tree such that n_i is the parent of n_{i+1} for $1 \leq i < k$.

→ The length of a path is one less than the number of nodes in the path. (# of nodes - 1)

→ The path of length zero is the length of the path from every node to itself.

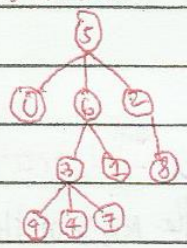
- Examples:
  - > Path: {5, 6, 3} ✓
  - > Path: {6, 5, 2} ✗

// There is a path from 5 to 3 since 5 is a parent of 6 & 6 is a parent of 3

// There is NO path from 6 to 2 since 6 is not a parent of 5. (Paths should start w/ the parent of the end node, etc.)

- > Path: {5, 6, 3, 4} = Length of the Path? → 3
- > Path: {5, 0} = Length of the Path? → 1
- > Path: {2} = Length of the Path? → 0 (Path of length zero)

- Illustration -



• Ancestor and Descendant •

→ If there is a path from node a to node b then a is an ancestor to b and b is a descendant of a. Any node is both an ancestor and a descendant of itself

Note: The root is the only node W/O a proper ancestor

→ Proper Ancestor is an ancestor of a node other than itself.

→ Proper Descendant is a descendant of a node other than itself.

- Examples: *Based on previous Illustration*
  - Path from 5 to 3 - 5 is an ancestor of 3, 3 is a descendant of 5.
  - Path from 2 to 2 - 2 is both an ancestor & a descendant of itself.
  - Proper Ancestors of 3 - 6, 5 / Ancestors of 3 - 3, 6, 5
  - Proper Descendants of 3 - 9, 4, 7 / Descendants of 3 - 3, 9, 4, 7

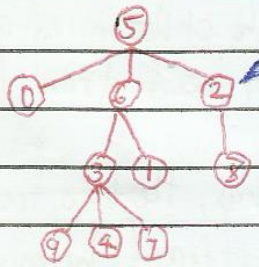
Strandmore

Introduction to Trees

date _____

• Leaf, Null, & Subtree •

Illustration (Again)



> **Leaf** - is a node w/ no proper descendants

> **Null** - A null tree is a tree with no nodes. It is represented by the symbol (Λ).

A sentinel value

> **Subtree** - A subtree of a tree is a node, together with all its descendants.

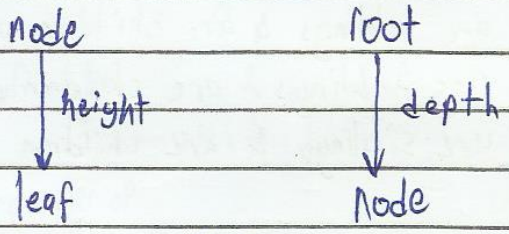
↳ Examples:
- Node 0
- Nodes 6, 3, 9, 4, 7, 1
- Nodes 3, 9, 4, 7

• Height and Depth •

> The height of a node in a tree is the length of the longest path from that node to a leaf.

> The depth of a node is the tree length of the unique path from the root to that node.

Illustrations:



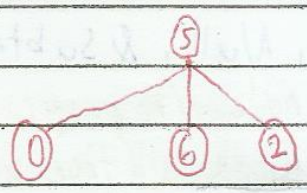
Example: *Based on Illustration*
- Height of 6 is 2
- Depth of 6 is 1

Introduction to Trees

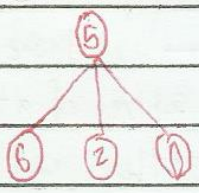
date _____

• Order of Nodes •

→ The children of a node is usually ordered from left-to-right.



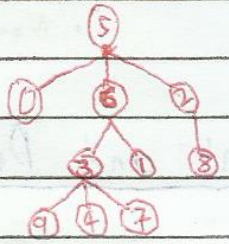
Thus, the two trees on the right are different because the children of node 5 appear in different order.



• Ordering of Siblings •

→ The left-to-right ordering of siblings (children of the same node) can be extended to compare any two nodes that are not related by ancestor-descendant relationship.

Illustration



The relevant rule is that:

if a and b are siblings, and a is to the left of b, then all descendants of a are to the left of all the descendants of b.

*Examples:
- 0, 6, 2 are siblings & are children of 5
  3 & 1 are siblings & are children of node 6
  9, 4, 7 are siblings & are children of node 3

- 6 & 2 are siblings & 6 is to the left of 2, then all the descendants of 6 (including itself) are to the left of the descendants of 2 (including itself.

3 & 1 are siblings, 3 is to the left of 1, then all descendants of 3 are to the left of the descendants of 1.

Sycamore

Introduction of trees

date _____

From Previous Page: What nodes are to the left 1? 0, 3, 9, 4, 7

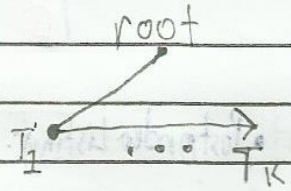
// Why? Since 0 & 6 are siblings and 0 is to the left of 6, then all the descendants of 0 are to the left of all descendants of 6. 1 is a descendant of 6, therefore, 0 is to the left of 1.

★ Systematic Order of Nodes ★

1.) Preorder - root, left, right

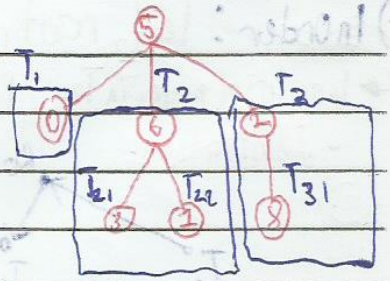
↳ root n of T, preorder of nodes of T_1, preorder of nodes of T_2, ... until preorder of nodes of T_k

>Illustration:



Tree T

>Example:



1.) The root n of T - 5

2.) Preorder of nodes of T_1 - 0

3.) Preorder of nodes of T_2 - 6, 3, 1

4.) Up to Preorder of nodes of T_3 - 2, 8

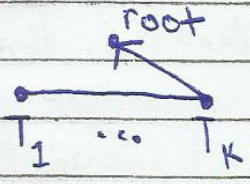
• Preorder Listing: 5, 0, 6, 3, 1, 2, 8

date _____

2.) Postorder: left, right, root

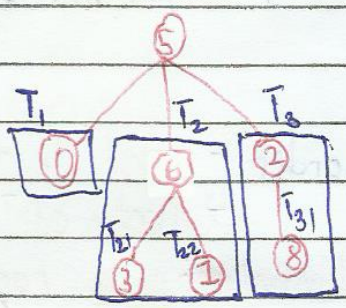
↳ Post order of nodes of T_1, Post order of nodes of T_2, up to postorder of nodes of T_K, root n of T

• Illustration:



• Example:

Tree T



1.) Postorder of nodes of T_1 – 0

2.) Post order of nodes of T_2 – 3, 6, 1

3.) Up to Post order of nodes of T_3 – 8, 2

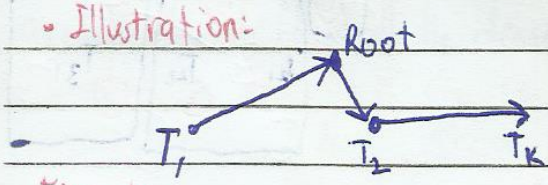
4.) All followed by root n of T – 5

• Postorder Listing: 0, 3, 1, 6, 8, 2, 5

3.) Inorder: left, root, right

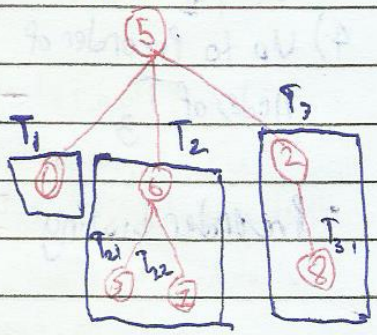
↳ Inorder of nodes in T_1, root n of T, Inoder of nodes in T_2, up to Inorder of nodes in T_K

• Illustration:



• Examples:

Tree T



1.) Inorder of nodes of T_1 – 0

2.) Root n of T – 5

3.) Inorder of nodes of T_2 – 3, 6, 1

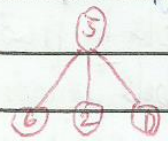
4.) Up to Inorder of nodes of T_3 – 8, 2

• Inorder Listing: 0, 5, 3, 6, 1, 8, 2

Strawbmore

date

• Traversal Summary •



Preorder: 5, 6, 2, 0

Postorder: 6, 2, 0, 5

Inorder: 6, 5, 2, 0

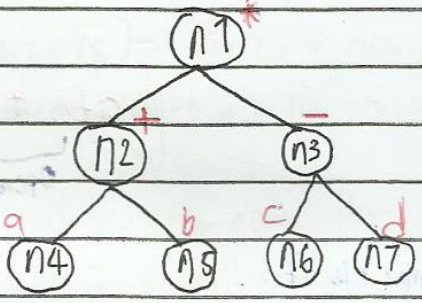
• Labeled and Expression Trees •

• The label of a node is the value 'stored' at the node and not the name of the node.

"An Element is to a List as a Label is to a Tree".

* A Labeled Tree is a tree whose nodes have labels.
* An Expression Tree is a tree where every leaf

Example:



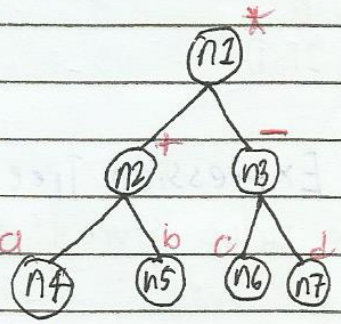
The above expression tree represents the arithmetic expression: $(a+b)*(c-d)$, where $n_1, n_2, \ldots, n_7$ are the names of the nodes, & the operands & operators are the labels.

date _____

• Forms of writing Expression •

1.) Prefix Notation - is the listing of labels in preorder.
2.) Infix Notation - is the listing of labels in inorder.
3.) Postfix Notation - is the listing of labels in postorder

*Example: (using previous expression tree)

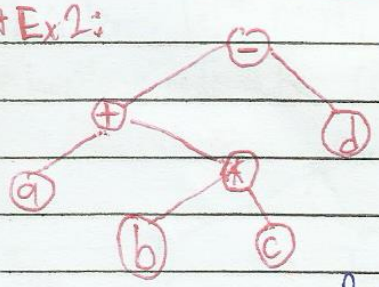


Mathematical Expression:
(a+b)*(c-d)

• Determine the ff:

• Prefix: *+ab-cd // clue for Prefix: operator, operand, operand
• Infix: a+b*c-d
• Postfix: ab+cd-* // clue for Postfix: operand, operand, operator

*Ex 2:



Mathematical Expression:
a+b*c-d

Determine the ff:

• Prefix: -+a*bcd (not sure)
• Infix: a+b*c-d
• Postfix: abc*+d-

operand
operand
operator

Tips for how to draw trees based on given expressions:
1. Look for operator, operand, operand
2. Make subtrees
3. rinse n repeat

ADT Tree Operations

date

1.) PARENT(n, T) - This function returns the parent of node n in tree T. If n is the root, which has no parent, ∧ is returned.

Sentinel value. We can return something else depending on specifications.

2.) LEFTMOST_CHILD(n, T) - The function returns the left most child of node n in tree T and returns ∧ if n is a leaf.

3.) RIGHT_SIBLING(n, T) - This returns the right sibling of node n in the tree T. Right sibling is defined to be the node r having the same parent p as node n & node r lies immediately to the right of node n in the ordering of the children of node p.

4.) LABEL(n, T) - This returns the label of node n in tree T.

5.) CREATEi ($v, T_1, T_2, \ldots, T_i$) - Makes a new root r w/ label v & gives it i children, which are the roots of $T_1, T_2 \ldots, T_i$, in order from left to right.

6.) ROOT(T) - This returns the node that is the root of tree T, or ∧ if T is a null tree.

7.) INITIALIZE(T) - This prepares the tree so that it will be used for the first time.

8.) MAKENULL(T) - This makes tree T be an empty tree.

date _____

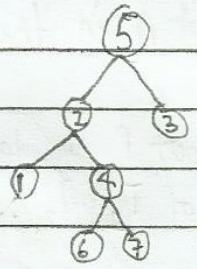
- A Recursive Preorder listing function. (found in book)

// note: This is Pseudocode

```
void PREORDER(node n, Tree T)
{ /*List the labels of the descendants of n in preorder*/
   node C;
   print (LABEL(n, T));
   C = LEFTMOST_CHILD(n,T);
   while (C <> 'Λ') // check if it is not equal to Null
   {
     PREORDER (C,T);
     C = RIGHT_SIBLING(C,T);
   }
}
```

- Simulation (attempt ra ni idk if correct):

Tree:



node n: 2

1. Print 2
2. C gets left most child of 2 (: 1)
3. is not null so it returns to step 1.
4. print 1
5. C gets left most child of 1 (NULL)
6. is Null so C now gets right sibling (4)
7. prints 4, C gets left most child 6
8. not null, prints 6, & gets left most child (Null)
9. is null, C gets right sibling (7).
10. prints 7, gets left most child (Null).
11. is Null, func ends?!?

Final output: 2, 1, 4, 6, 7 ?!?!

★ Implementations of Trees ★

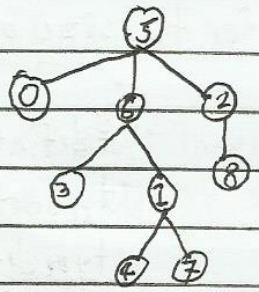
date _____

1.) Parent Pointer Representation // is essentially an array

// The nodes are used as indeces in the array & the parents are placed in the array

Tree T

0	5
1	6
2	5
3	6
4	1
5	-1
6	5
7	1
8	2
9	-2



// 5 is the root, which has no parent, so we use -1 (a sentinel value)

// 9 is a nonexistent node, a different value is required

↳ An Array Representation of Trees.

↳ This representation is the simplest representation of tree T that supports the PARENT operation.

↳ T[x] = y, if node y is the parent of node x;
T[x] = -1, if node x is the root node; and
T[x] = -2, if node x is not a node in the tree.

★ Advantages:

- Parent(n, T) is O(1) (optimum/constant time).

★ Disadvantages:

- Root(T) is O(N) since you have to locate where -1 is.
- Does not facilitate operations that require child_of_information
- Does not specify the order of the children of a node, thus operations LEFTMOST_CHILD and RIGHT_SIBLING are not well defined

Exercise 4

date _____

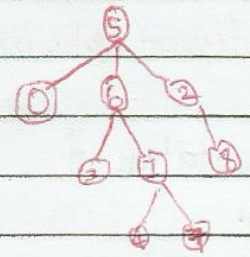
Tree T

0	5
1	6
2	5
3	6
4	1
5	-1
6	5
7	1
8	2
9	-2

1.) Write an appropriate definition of datatypes Tree & node. Include a macro definition for the size of the array

```
#define MAX 10
typedef int Tree[MAX];
typedef int node;
```

Illustration:



2.) Write the code of the function:

node PARENT(node n, Tree T);

• code:

```
Node PARENT(node n, Tree T){
return(T[n]!=-1 && T[n]!=-2)T[n]:-1;
}
```

return -1 if root

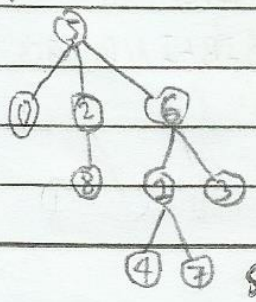
check if node has a parent & is not the root & does exist.

3. Suppose we impose an artificial order of children, i.e., numbering the children in increasing order from left to right. Write the code of the function

node RIGHT_SIBLING(node n, Tree T);

• Artificial order:

// im not sure about this at all



straightmore

Tree T

0	5
1	6
2	5
3	6
4	1
5	-1
6	5
7	1
8	2
9	-2

// still the same in the array

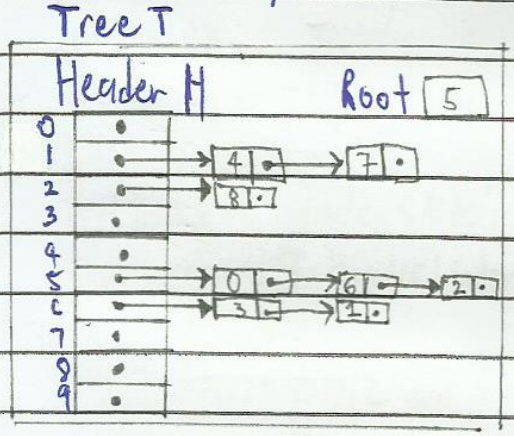
date

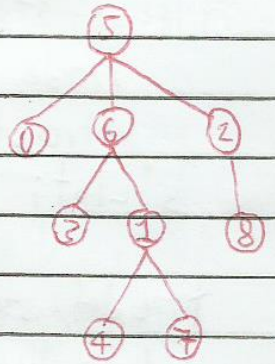
#3 code (Not sure bout dis)

```
node RIGHT_SIBLING(node n, Tree T){
  node r, index, p = T[n];

  for(index = 0; index < MAX; index++){
    if (T[index] == p && index > n){
      r = index;
    }
  }
  return r;
}
```

• Representations of Trees by List of Children •





// Unlike the previous version, where we denote root by -1, this version makes it hard to distinguish the root, so if needed, create a root field.

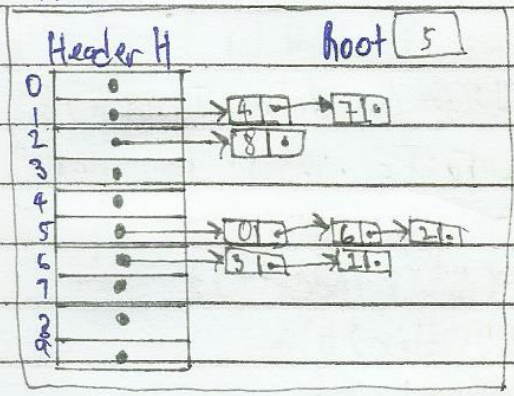
// LEFTMOST_CHILD(node n, Tree T) is O(1)
PARENT(node n, Tree T) is O(N).

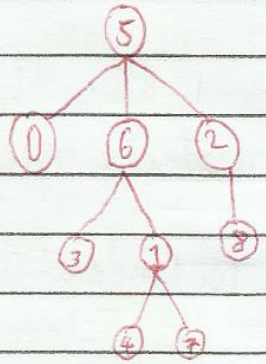
Strathmore

Exercise 5

date _____

Tree T





1.) Write an appropriate definition of the datatypes Tree & node using the representation of trees by List of children. Include macrodefinition for the size of the array.

> Code:

```
#define MAX 10
typedef int node;
typedef struct node {
    node nodeData;
    struct node *link;
} *List;

typedef struct {
    List Header [MAX];
    node Root;
} Tree;
```

2.) Write the code for each of the following:

a.) node LEFTMOST_CHILD (node n, Tree T);

b.) node PARENT (node n, Tree T);

Strathmore

Exercise 5

date

a.) node LEFTMOST_CHILD (node n, Tree T)

//unsure

```
node LEFTMOST_CHILD (node n, Tree T) {
return (T.Header[n] != NULL && T.Header[n]->link != NULL)? T.Header[n]->nodeData
                                                          :-1;
}
```

To check if the node is not empty & is not a leaf

b.) node Parent (node n, Tree T)

//unsure since it is hard to distinguish a parent in
//dis representation

```
node PARENT (node n, Tree T) {
  int index;
  node P = -1;
  List trav;
  for(index=0; index<MAX;) {
   for(trav= T.Header[index]; trav!=NULL && trav->nodeData != n;
                                  trav = trav->link) {}
   if (trav!= NULL && trav->nodeData != T.Root) {
          P= index;
    }
    index++;
  }                                   //note: this can be
  return P;                           improved somehow
}
```

★ Binary Tree ★

date _____

↳ A tree which is either empty ... Or

↳ A tree which every node has either

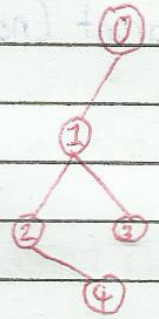
- no child, or
- a left child, or
- a right child, or
- both a left child & a right child.

~• Implementation of Binary Tree •~

TreeT

	LeftChild	RightChild
0	1	-1
1	2	3
2	-1	4
3	-1	-1
4	-1	-1

• Illustration:



// Dis is a structure containing an array of structures
// elements left & right entail the index's/node's children.

• Data struct Def:

```
#define MAX 5
typedef struct {
   int LC, RC;
} Tree[MAX];
```

Strudmore

★ Huffman's Algorithm ★

date _____

↳ One of the techniques in finding optimal prefix length code

↳ Implemented using a forest → a collection of trees.

• Steps •

1.) initially, each character is in a one-node tree by itself

2.) while (# of trees > 1)

a.) select two trees with the smallest probability.

b.) create a bigger tree from the 2 selected trees, and

c.) make the sum of their probability the root of the new tree.

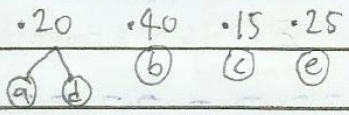
3.) Label the left & right subtrees 0 & 1 respectively.

• Ex:

1.) initial

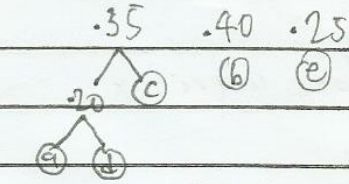
2.) (# trees > 1)? Yes

Merge a & d



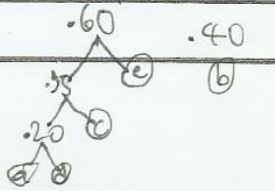
3.) (# trees > 1)? Yes

Merge a, d with c



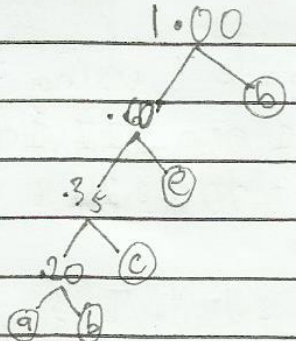
4.) (# trees > 1)? Yes

Merge a, c, d with e

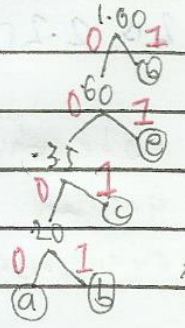


5.) (# trees > 1)? Yes

Merge a, c, d, e with b



6.) (# trees > 1)? NO. Label path left & right w/ 0 & 1



a	.12	0000	4
b	.40	1	1
c	.15	001	3
d	.08	0001	4
e	.25	01	2

Avg. Code length: 1.15

.12	.40	.15	.08	.25
x 4	1	3	4	2
.48	.40	.45	.32	.50

Straightforward

.48
.40
.45
.32
.50
1.15

// writers Note: IT'S 12:00 A.M.

Exercise help

date

. Given the letters & their probability of occurence, using Huffman Algo., determine the Huffman code & the Avg. code length of the Following: A, B, C, D, E, & F.

A → 25% B → 15%
C → 5% D → 15%
E → 20% F → 20%

.25 .15 .05 .15 .20 .20

1.) (A) (B) (C) (D) (E) (F)

2.) (# trees > 1)? Yes

Merge B & C

.20 .25 .15 .20 .20

(C)(B) (A) (D) (E) (F)

3.) (# trees > 1)? Yes

Merge B&C with D

.35 .25 .20 .20

.20 (D) (A) (E) (F)

(C)(B)

4.) (# trees > 1)? Yes

Merge E & F

.35 .40 .25

.20 (D) (E)(F) (A)

(C)(B)

5.) (# trees > 1)? Yes

Merge B, C, D with A

.60 .40

.35 (A) (E)(F)

.20 (D)

(C)(B)

6.) (# trees > 1)? Yes

Merge B, C, D, A with E & F

1.00

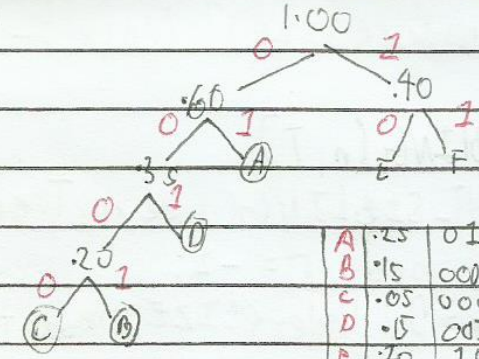
.60 .40

.35 (A) (E)(F)

.20 (D)

(C)(B)

7.) (# trees > 1)? NO. Label



A	.25	01	2
B	.15	001	3
C	.05	0000	4
D	.15	0001	4
E	.20	10	2
F	.20	11	2

// Avg. code length

.25	.15	.05	.15	.20	.20
2	3	4	4	2	2
.50	.45	.20	.60	.40	.40

.50
.45
.20
.60
.40
.40
2.55 → Avg. code length

Straitmore

Summary: Parent Pointer Operations

date _____

Data Struct Def:

```
#define SIZE 10
typedef int node;
typedef int Tree[SIZE
```

- INITIALIZE(T):

```
void INITIALIZE(Tree T){
    int index;
    for(index=0; index<SIZE; index++){
        T[index]=-2;
    }
}
```

- PARENT(n,T):

```
node PARENT(node n, Tree T){
return(n<SIZE && T[n]!=-2)? T[n]:-2;
```

- LEFTMOST_CHILD(n,T): // assume artificial ordering

```
node LEFT MOST_CHILD(node n, Tree T){
  int index;
  for(index=0; index<SIZE && n!=T[index]; index++){}
  return(index<SIZE)? index :-2;
```

- RIGHT_SIBLING(n,T):

```
node RIGHT_SIBLING(node n, Tree T){
  int index; node r=-2;
  for(index=0; index<SIZE; index++){
    if(T[n]==T[index] && index>n){
      r=index;
    }
  }
  return r;
}
```

Summary: Parent Pointer

date

. ROOT (T)

```
node ROOT(Tree T){
  int index;
  for(index=0; index<SIZE && T[index]!=-1; index++){}
  return(index<SIZE)? index: -2;
}
```

// Label & CREATE; t.b.d.

Summary: Representing by List of children

date _____

INITIALIZE(T):

```
void INITIALIZE(Tree *T, int root){
  int index;
  T->Root = root;
  for(index = 0; index<SIZE; index++){
    T->Header[index] = NULL;
  }
}
```

PARENT(n,T):

```
int PARENT(int n, TreeT){
  int index, p=-1;
  LIST trav;
  for(index=0; index<SIZE; index++){
    for(trav=T.Header[index]; trav!=NULL; trav=trav->link){
      if(trav->data==n){
        p=index;
      }
    }
  }
  return p;
}
```

LEFTMOST_CHILD(n,T):

```
int LEFTMOST_CHILD(int n, TreeT){
  int L;
  return L=(n<SIZE && T.Header[n]!=NULL)?T.Header[n]->data:-1;
}
```

Data Struct Def:

```
#define SIZE 10

typedef struct node{
  int data;
  struct node* link;
}*LIST;

typedef struct{
  LIST Header[SIZE];
  int ROOT;
}Tree;
```

Strathmore

Summary: Representing by List of children

date

- RIGHT_SIBLING(int n, T):

```
int RIGHT_SIBLING(int n, Tree T){ //this can be optimized
 int index, r = -1, p = T.Header[n];
  List trav;
for(index=0; index<SIZE&& T.Header[index]!= p; index++){
 for (trav = T.Header[index]; trav != NULL; trav = trav->link){
   if (trav -> link == NULL){
       R = trav -> data;
    }
  }
= return R;
}
```

- ROOT(T):

```
int ROOT(Tree T){
   return T.Root;
```

- Label & CREATE; idk