data_tructures(&algorithms, lecture04)

Doan Trung Tung, PhD – University of Greenwich (Vietnam)

Plan

01

Binary Search

Recursive Binary Search, Iterative Binary Search, analysis

02

Tree

Tree terminologies, construct a tree, tree traversal

03

Binary Search Tree

Construct a BST, search on BST, delete notes

04

AVL Tree

Balance tree, rotation operations

Binary Search

Recursive Binary Search, Iterative Binary Search, analysis

Normal Search

- Key idea: going from left to right (or vice versa), looking for x and return found position.
- Complexity: O(n)

```c
int normal_search(int *a, const int n, const int x)
{
    for (int i = 0; i < n; i++)
    {
        if (x == a[i]) return i;
    }
    return -1;
}
```

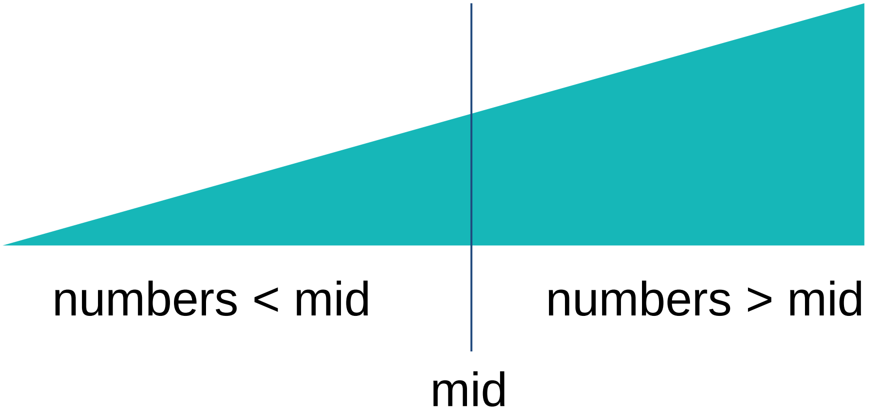
Recursive Search

- Base case: start = n => not found
- Recursive case: check at start, else recursively check at start + 1

```
int recursive_search(int *a, const int n,
                     const int x, const int start)
{
    if (start == n) return -1;
    if (a[start] == x) return start;
    return recursive_search(a, n, x, start + 1);
}
```

Binary Search

- <u>Key idea</u>: for sorted collection, check at the middle and if not found, check only left or right half because the collection is sorted



Binary Search

- Recursive Implementation: $T(n) = T(n/2) + c$
- Complexity: $O(\log n)$

```
int binary_search(int *a, const int lo, const int hi, const int x)
{
    if (lo > hi) return -1;

    int mid = (lo + hi) / 2;

    if (a[mid] == x)      return mid;
    else if (a[mid] < x) return binary_search(a, mid + 1, hi, x);
    else                 return binary_search(a, lo, mid - 1, x);

}
```

c

T(n/2)

Binary Search

- Recursive Implementation: $T(n) = c + c + .. + c$
- $= c\log(n)$

```
int binary_search_iter(int *a, const int n, const int x)
{
    int lo = 0, hi = n - 1;
    while (lo <= hi)
    {
        int mid = (lo + hi) / 2;
        if (a[mid] == x) return mid;
        else if (a[mid] < x) lo = mid + 1;
        else hi = mid - 1;
    }
    return -1;
}
```

c

Tree

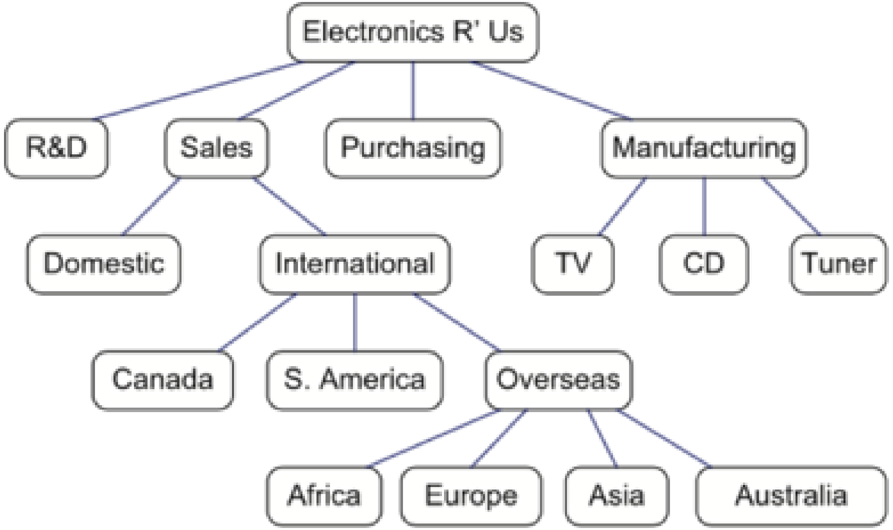
Tree terminologies, construct a tree, tree traversal

What Is A Tree

- A tree is an abstract model of a hierarchical structure
- A tree consists of nodes with parent-child relation
- Applications:
  - Organization charts
  - File systems
  - Programming environments
- Every node except one (a root) has a unique parent

Formal Definition

- Empty structure is an empty tree
- Non-empty tree consists of a root and its children, where these children are also trees



Tree Terminologies

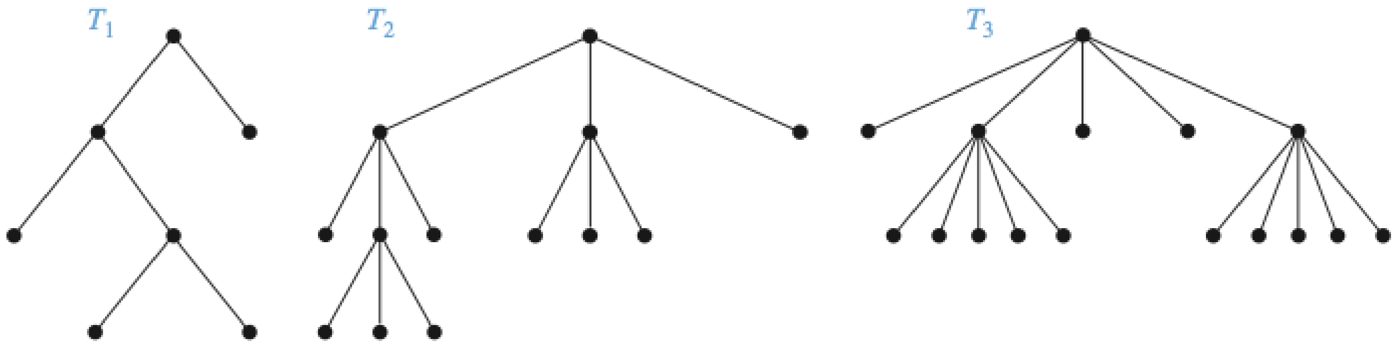
- Root: unique node without a parent
- Internal node: node with at least one
- Leaf: node without children
- Ancestors: parent, grandparent, great-grand-parent, …
- Descendants: child, grandchild, great-grandchild, etc.
- Level: a root is at level 1 (sometimes 0). A father is at level *i* then its children are at level *i+1*

Tree Terminologies

- Height: maximum level in a tree
- Empty tree is a legitimate tree of height 0 (by definition)
- A single node is a tree of height 1
- Degree (order): number of its children
- Each node has to be reachable from the root through a unique sequence of arcs, called path.
- The number of arcs in a path is called the length of the path.

m-ary Trees

- Definition: A rooted tree is called an m-ary tree if each vertex has at most m children.
- The tree is called a full m-ary tree if every internal vertex has exactly m children.
- An m-ary tree with m = 2 is called a binary tree.



Properties of Trees

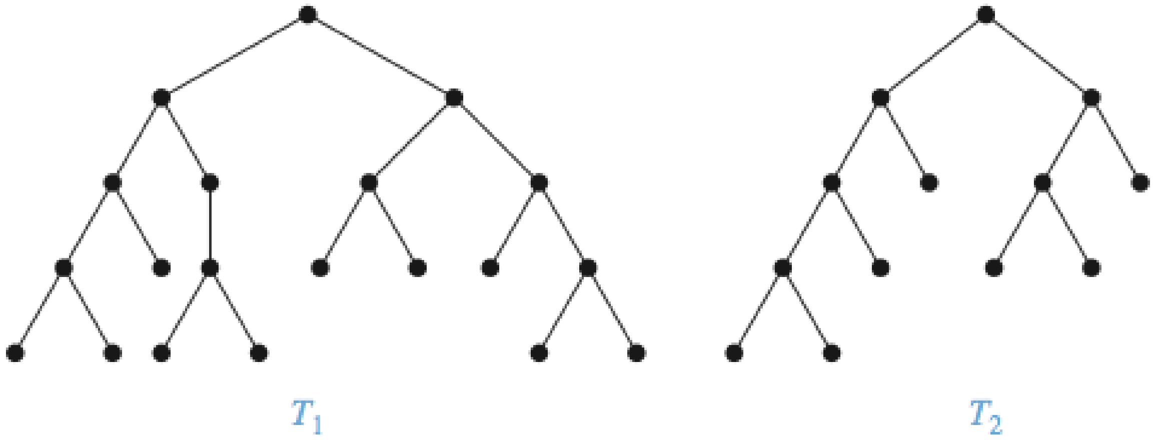
- A tree with n vertices has $n - 1$ edges
- If a full m-ary tree with n vertices, i internal vertices and l leaves then only one of n, i and l can determine the others.
  - If n vertices, then $i = (n - 1) / m$ and $l = [(m - 1)n + 1]/m$
  - If i internal vertices, then $n = mi + 1$ and $l = (m - 1)i + 1$
  - If l leaves then $n = (ml - 1)/(m - 1)$ and $i = (l - 1) / (m - 1)$

Example: MLM Company

- A MLM company starts by one person. Then each person has to find 4 people to sell products and join the network (which are called referrals). Some people can do this, but others cannot. If there are 100 people who cannot find any referrals, then how many people who have referrals and how many people in the company?

Balanced Trees

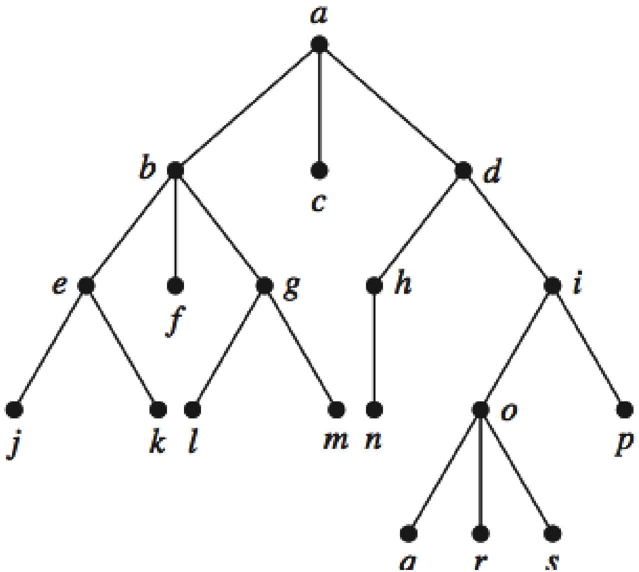
❖ An m-ary tree of height h is called a balanced tree if each leave has level h or h−1.



which one is balanced?

Exercises

- Which vertex is the root?
- Which vertices are internal?
- Which vertices are leaves?
- Which vertices are children of j ?
- Which vertex is the parent of h?
- Which vertices are siblings of o?
- Which vertices are ancestors of m
- Which vertices are descendants of b?
- Is T a m-ary tree for some positive integer m?
- What is the level of each vertex of T?

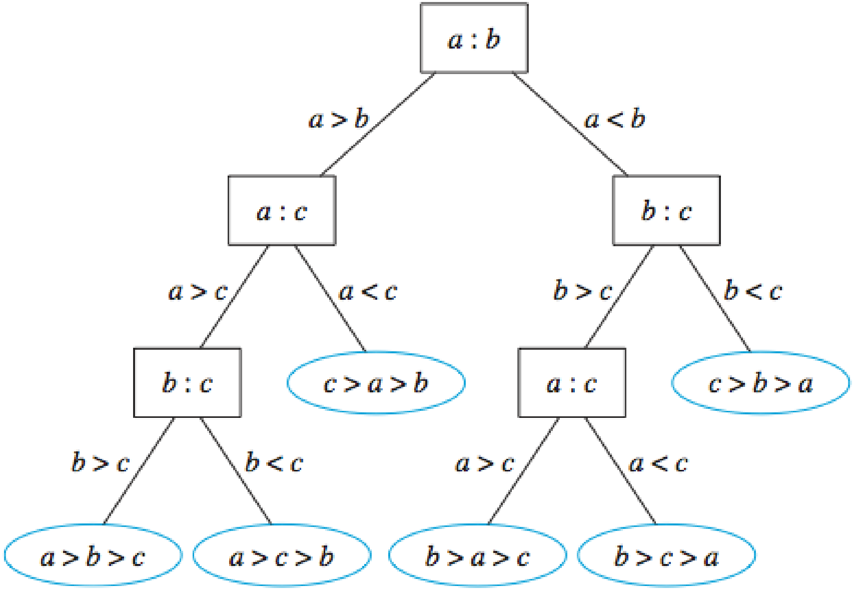


Excercises

- How many vertices does a full 5-ary tree with 100 internal vertices have?
- How many leaves does a full 3-ary tree with 100 vertices have?

Tree Application: Decision Tree

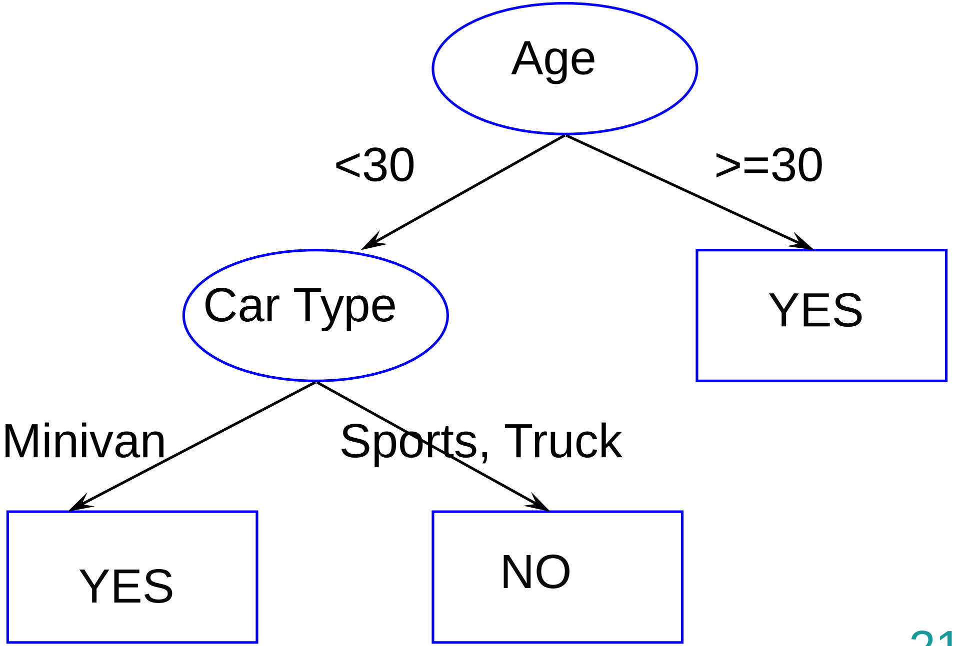
- A rooted tree in which each internal vertex corresponds to a decision that need to choose and each edge incident with it is a possible solution.



Decision Tree Example

Age	Car	Class
20	M	Yes
30	M	Yes
25	T	No
30	S	Yes
40	S	Yes
20	T	No
30	M	Yes
25	M	Yes
40	M	Yes
20	S	No

Bob is 43, will Bob buy a Sport car?
Mike is 29, will Mike buy a Sport car?



Tree Application: Huffman Coding

- Fixed length Coding
  - If we use bit strings of length n to represent 26 letters, what is the minimum of n?

A → 00001
B → 00010
C → 00011
D → 00100
E → 00101
F → 00110
G → 00111

000110000110100 ←→ CAT

  - What is the disadvantage of using fixed length bit strings?

Huffman Coding

- Variable length coding
  - a ~ 0
  - b ~ 1
  - c ~ 01
  - d ~ 10
- What does this mean: 0110?

Huffman Coding

- Prefix Coding
  - Code of any character is not prefix of any other character's code

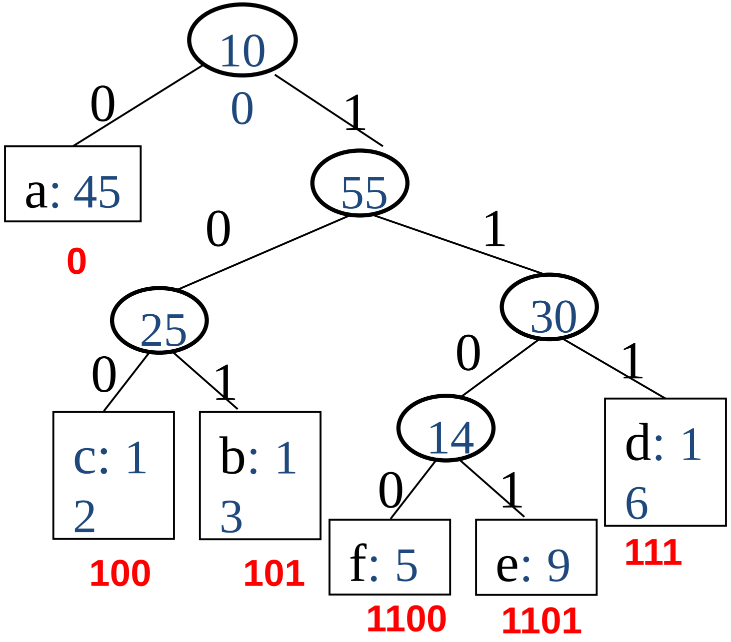
E	0
T	11
N	100
I	1010
S	1011

What does this mean?

11010010010101011

Huffman Coding

- Huffman Coding Tree: A binary tree represent prefix codes with their frequencies

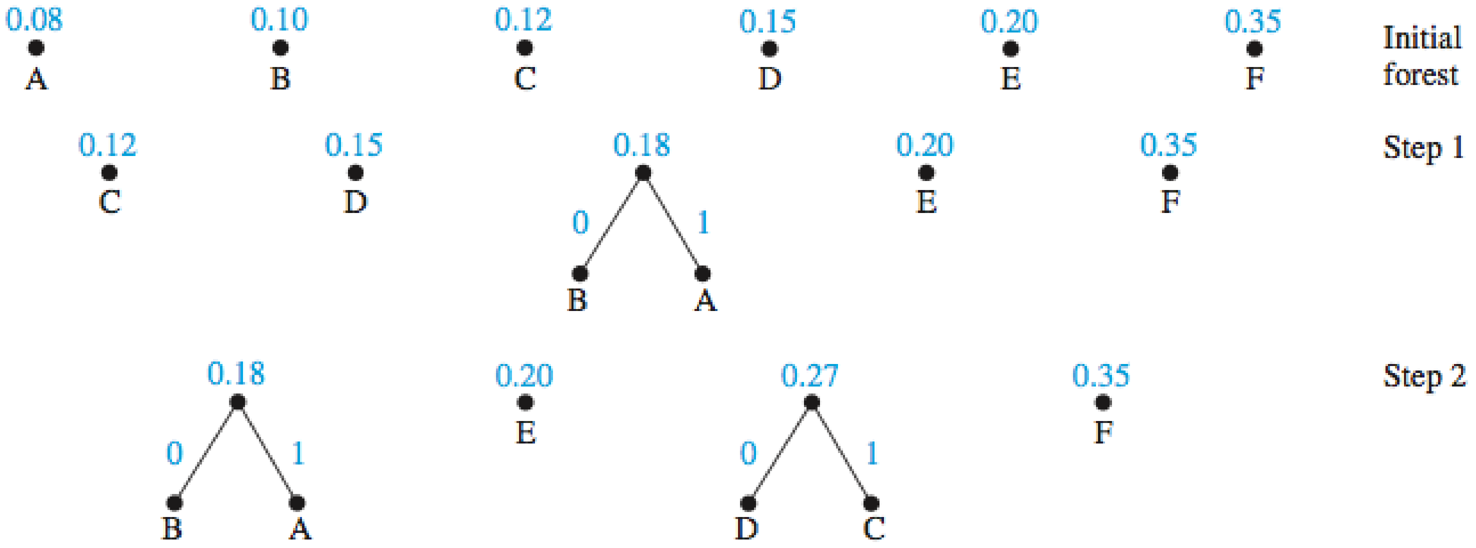


Huffman Coding

- Build a binary tree to create prefix codes
  - Give a message that need to be coded
  - Write out all distinct letters of the message together with their frequencies. Consider each letter as a binary tree with only one vertex. Call the frequency corresponding to each tree its weight.
  - At each step, combine two trees whose sum of weights is a minimum into a single tree by adding a new root, and assign this sum of weights as the weight of the new tree.
  - The algorithm is finished if a single tree is constructed.

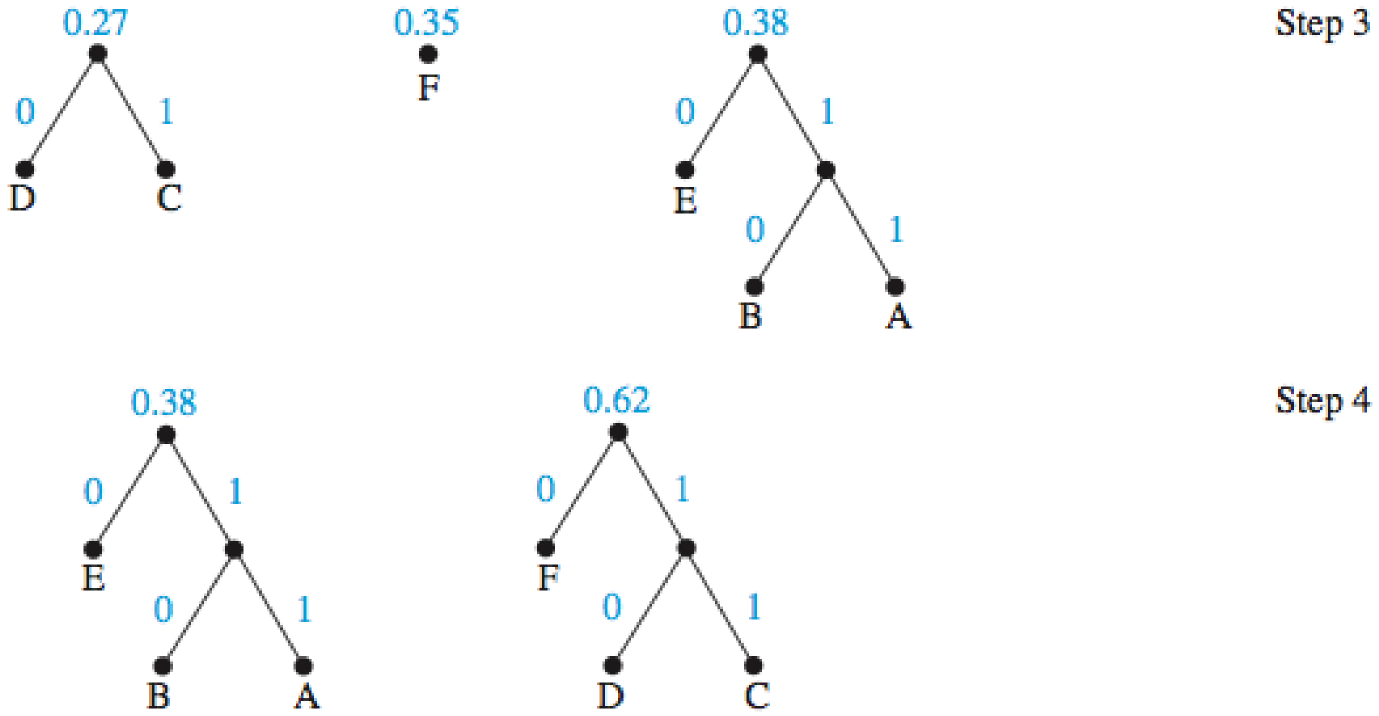
Huffman Coding

❖ Example



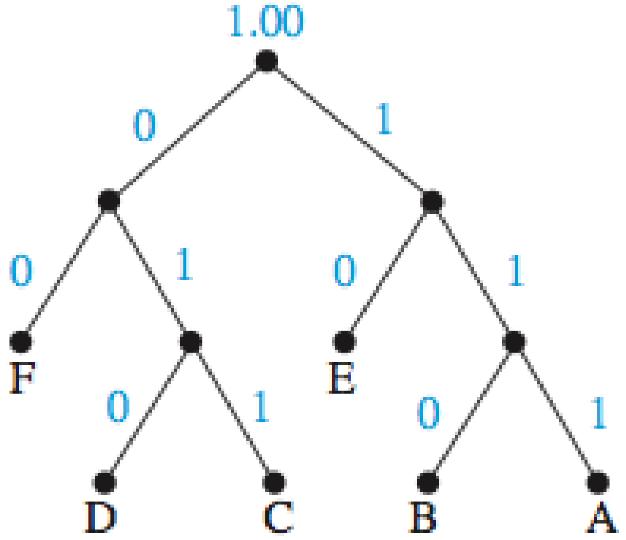
Huffman Coding

- Example



Huffman Coding

- Example



Step 5

A = 111
B = 110
C = 011
D = 010
E = 10
F = 00

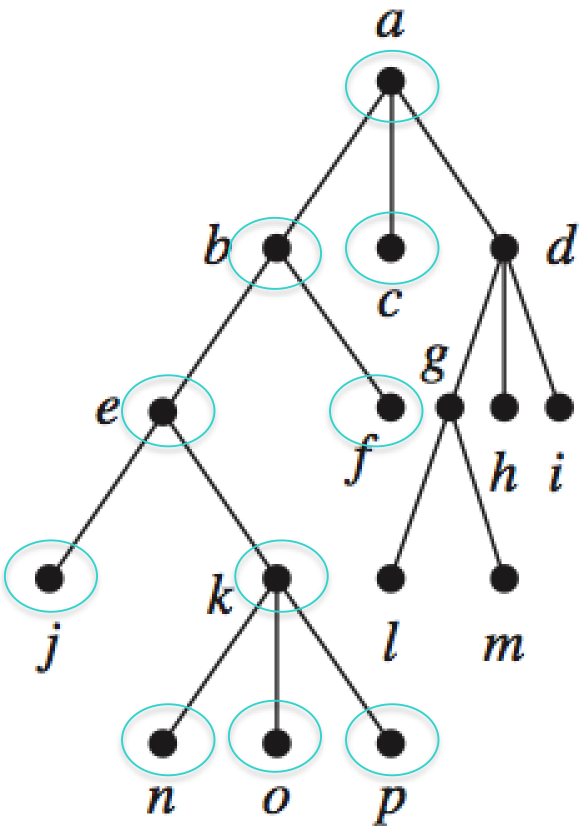
What does this mean?
0111110010

Average number of bits:
$3 \cdot 0.08 + 3 \cdot 0.10 + 3 \cdot 0.12 + 3 \cdot 0.15 + 2 \cdot 0.20 + 2 \cdot 0.35 = 2.45$

Huffman Coding

- Exercise
  - Which of these codes are prefix codes?
    a) a:11,e:00,t:10,s:01
    b) a:0,e:1,t:01,s:001
    c) a:101,e:11,t:001,s:011,n:010
    d) a: 010, e: 11, t: 011, s: 1011, n: 1001, i: 10101
  - Use Huffman coding to encode these symbols with given frequencies: A: 0.10, B: 0.25, C: 0.05, D: 0.15, E: 0.30, F: 0.07, G: 0.08. What is the average number of bits required to encode a symbol?

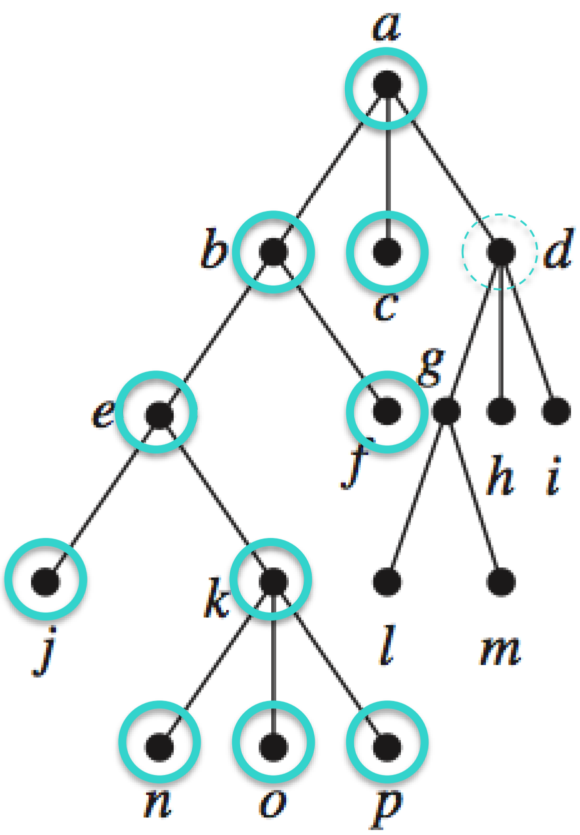
Tree Traversal: Pre-order



- Visit the root
- Traverse each of T_1, T_2, ...,T_n in preorder.

a b e j k n o p f c d g l m h i

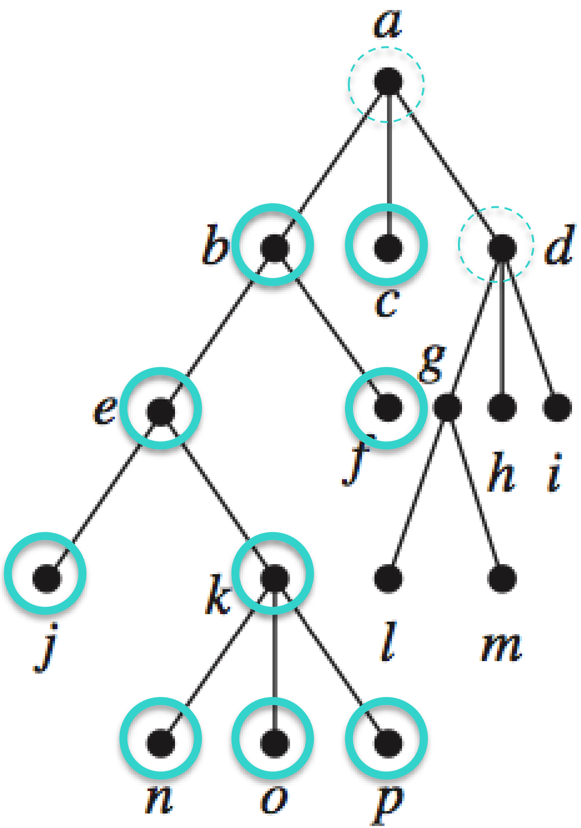
Tree Traversal: In-order



- Traverse left child in In-order.
- Visit the root.
- Traverse each of right children in In-order

j e n k o p b f a c l g m d h i

Tree Traversal: Post-order



- Traverse each of $T_1, T_2, \ldots, T_n$ in Post-order.
- Visit the root.

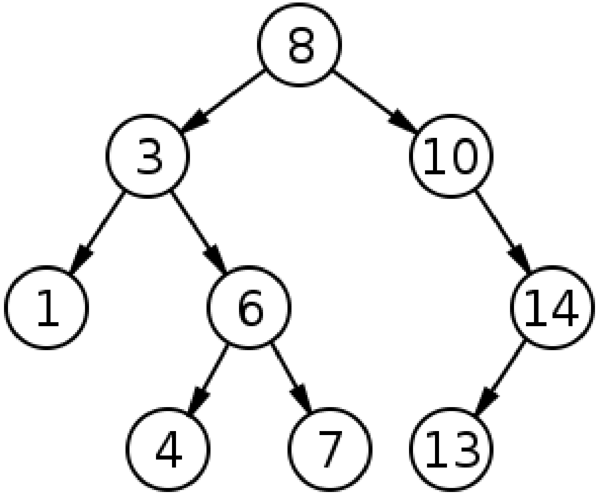
j n o p k e f b c l m g h i d a

Binary Search Tree

Construct a BST, search on BST, delete notes

Binary Search Tree

- A Binary Search Tree is a binary tree with the following properties:
  - Each child of a vertex is designated as a right or left child, no vertex has more than one right child or left child
  - Each vertex is labeled with a key which is both larger than the keys of all vertices in its left subtree and smaller than the keys of all vertices in its right subtree

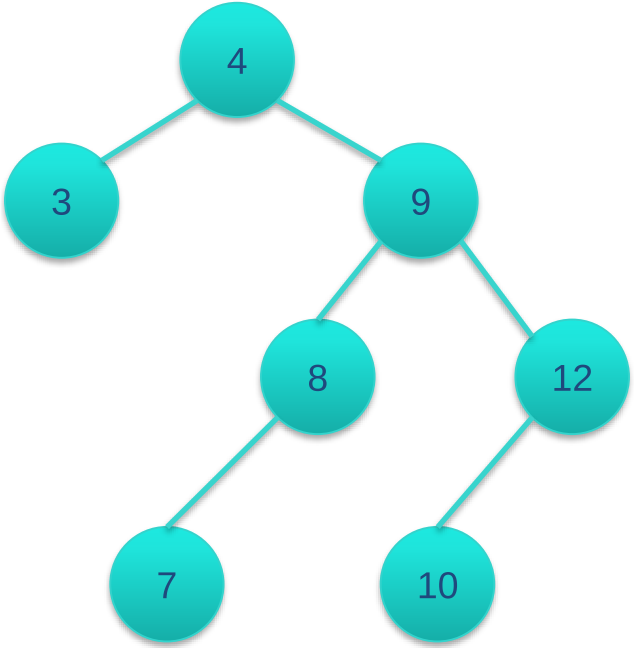


Binary Search Tree

- Start with a tree containing just one vertex, namely, the root.
- The first item in the list is assigned as the key of the root.
- To add a new item, first compare it with the keys starting at the root and moving to the left if the item is less than the key of the respective vertex if this vertex has a left child. Do the same for the right child.
- When the item is less than the respective vertex and this vertex has no left child, then a new vertex with this item as its key is inserted as a new left child. Do the same for the right child

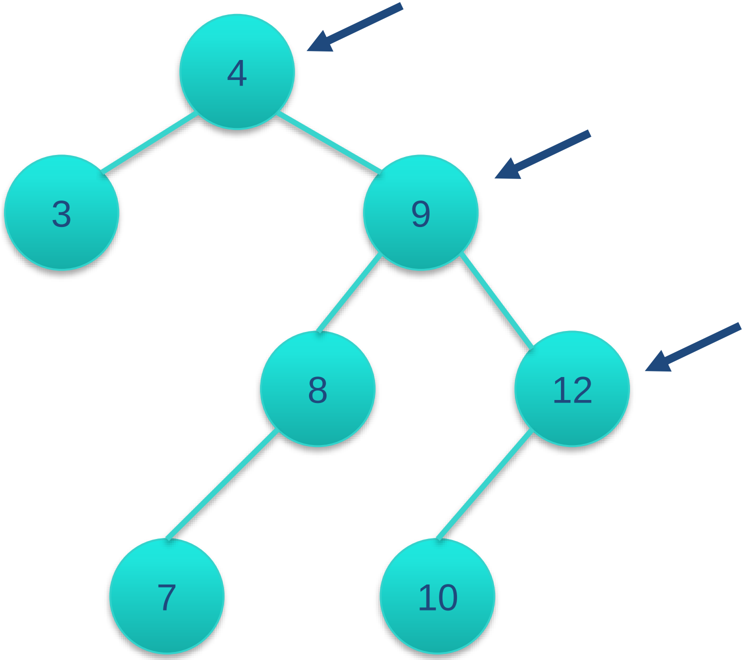
Binary Search Tree

- Construct a BST from a list {4, 9, 12, 8, 3, 10, 7}



Binary Search Tree

- Search an element in BST: O(logn)
  - Search number 13?



m-ary Tree Implementation & Traversal

- Define tree structure

```
#define MAX_CHILDREN 3

typedef struct str_tree_node * child;

struct str_tree_node
{
    char data;
    child children[MAX_CHILDREN];
};

typedef struct str_tree_node tree_node;
```

m-ary Tree Implementation & Traversal

❖ Define tree operations

```
tree_node* create_node(const char data);
void clear_tree(tree_node **root);
void in_order(tree_node * const root);
void pre_order(tree_node * const root);
void post_order(tree_node * const root);
```

m-ary Tree Implementation & Traversal

- In-order traversal

```
void in_order(tree_node * const root)
{
    if (root == NULL) return;

    in_order(root->children[0]);
    printf("%c ", root->data);
    for (int i = 1; i < MAX_CHILDREN; i++)
        in_order(root->children[i]);
}
```

Binary Search Tree Implementation

- Declare BST structure

```c
typedef struct str_node node;

struct str_node
{
    int key;
    node *left;
    node *right;
};
```

Binary Search Tree Implementation

❖ Declare BST operations

```
node* create_node(const int key);
void insert_node(node **root, const int key);
void insert_child(node **child, node* n);
node* create_bst(int *a, const int n);
node* search_bst(node *root, const int key);
void pre_order(node *root);
```

Binary Search Tree Implementation

- Define BST operations

```
void insert_node(node **root, const int key)
{
    node *n = create_node(key);

    if (*root == NULL) *root = n;
    else
    {
        if ((*root)->key == key) return;
        if ((*root)->key <  key) insert_child(&(*root)->right, n);
        else                     insert_child(&(*root)->left,  n);
    }

}
```

Binary Search Tree Implementation

❖ Define BST operations

```
node* create_bst(int *a, const int n)
{
    node *root = NULL;

    for (int i = 0; i < n; i++)
    {
        insert_node(&root, a[i]);
    }
    return root;
}
```