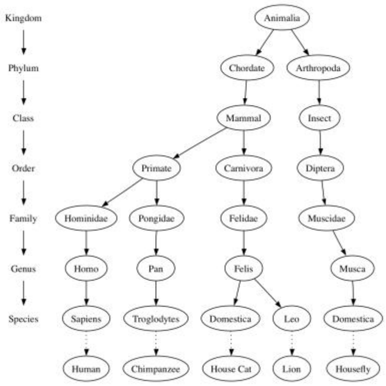
Trees

- Not a linear data structure!
  - Allows for more complex relationships between data elements
- Applications
  - File system
  - Decision Trees
  - Classification systems
  - Database systems
  - Graphics
  - …
- Hierarchical data structure
  - Terminology comes from “trees” and “family trees”
  - Viewed upside down – root is at the top!

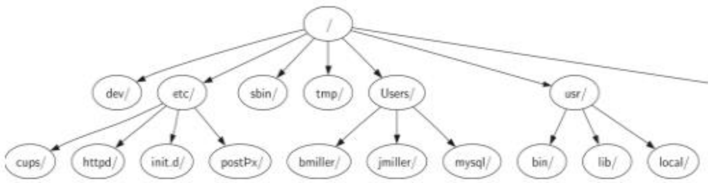
1

Example: Biology Classification Tree



2

Example: File System



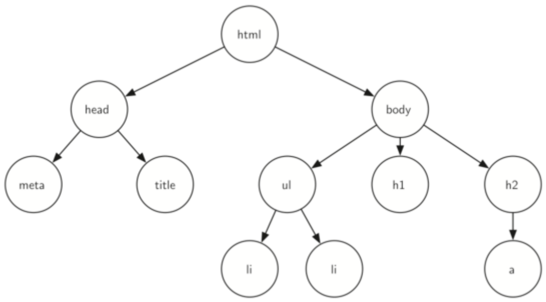
3

Example: HTML source code

```
<html xmlns="http://www.w3.org/1999/xhtml"
    xml:lang="en" lang="en">
<head>
  <meta http-equiv="Content-Type"
      content="text/html; charset=utf-8" />
  <title>simple</title>
</head>
<body>
<h1>A simple web page</h1>
<ul>
  <li>List item one</li>
  <li>List item two</li>
</ul>
<h2><a href="http://www.cs.luther.edu">Luther CS </a><h2>
</body>
</html>
```


4

Example: HTML source code



Rooted Tree

- A rooted tree is a set of nodes and a set of **directed** edges
  - Edge connects two nodes and indicates a relationship between them
  - node is designated as root, it has no incoming edges
  - all other nodes have exactly one incoming edge (from the parent) parent-child
  - unique path from root to each node, path length = #edges
- Leaf: a node without children
- Subtree: set of nodes and edges comprised of a parent and all the descendants of that parent
- Depth of node: path length from root
- Height (level) of a node: length of path to deepest leaf in “subtree”
- Height of a tree = height of the root
- Siblings, ancestors, and descendants

Definition 1

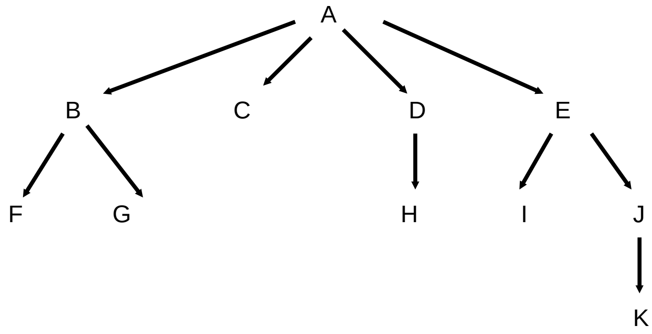
- Definition 1: A rooted tree consists of a set of nodes and a set of edges that connect pairs of nodes. A tree has the following properties:
  - One node of the tree is designated as the root node.
  - Every node n, except the root node, is connected by an edge from exactly one other node p, where p is the parent of n
  - A unique path traverses from the root to each node.
  - If each node in the tree has a maximum of two children, we say that the tree is a binary tree.

Definition 2: Recursive

- Definition Two: A tree is either
  - empty or
  - consists of a root and zero or more subtrees, each subtree is a tree The root of each subtree is connected to the root of the parent tree by directed edge from the root to the root of the subtree.

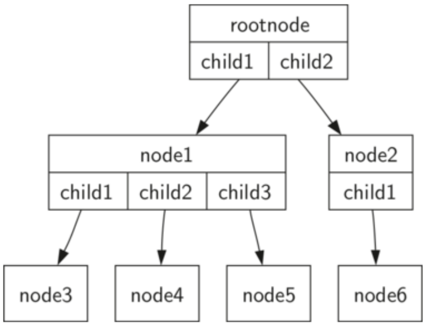
- Notes:
  - A tree can be empty
  - There can be any finite number of children
  - All the edges “point away” from the root

Example of a Rooted Tree



9

Rooted Tree



10

Ordered Rooted Trees

Definition: An *ordered rooted tree* is a rooted tree where the children of each internal vertex are ordered.

- We draw ordered rooted trees so that the children of each internal vertex are shown in order from left to right.

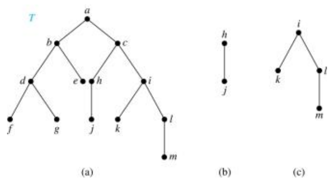
Definition: A *binary tree* is an ordered rooted where where each internal vertex has at most two children. If an internal vertex of a binary tree has two children, the first is called the *left child* and the second the *right child*. The tree rooted at the left child of a vertex is called the *left subtree* of this vertex, and the tree rooted at the right child of a vertex is called the *right subtree* of this vertex.

Example: Consider the binary tree *T*.

(*i*) What are the left and right children of *d*?
(*ii*) What are the left and right subtrees of *c*?

Solution:

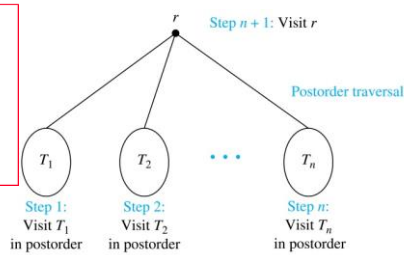
(*i*) The left child of *d* is *f* and the right child is *g*.
(*ii*) The left and right subtrees of *c* are displayed in (b) and (c).

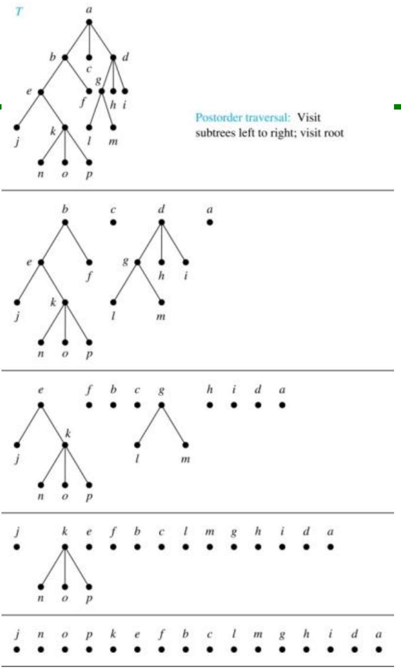


Postorder Traversal in ordered trees

Definition: Let *T* be an ordered rooted tree with root *r*. If *T* consists only of *r*, then *r* is the *postorder traversal* of *T*. Otherwise, suppose that $T_1, T_2, \ldots, T_n$ are the subtrees of *r* from left to right in *T*. The postorder traversal begins by traversing T_1 in postorder, then T_2 in postorder, and so on, after T_n is traversed in postorder, *r* is visited.

```
procedure postordered (T: ordered rooted tree)
r := root of T
for each child c of r from left to right
    T(c) := subtree with c as root
    postorder(T(c))
visit r
```





13

Definition: Conceptual diagram

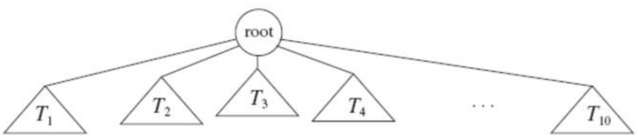


Figure 4.1 Generic tree

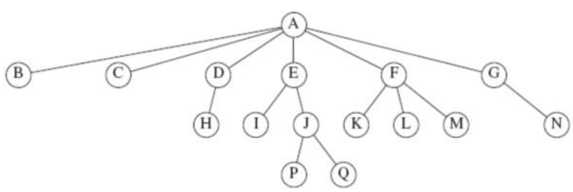


Figure 4.2 A tree

14

Implementation issue for general rooted tree

- Each node has a reference to each child
  - But this means an indeterminate number of references must be kept in the parent
- Solution: Each node contains a reference to first child and next sibling

```
class TreeNode
{
    Object   element;
    TreeNode firstChild;
    TreeNode nextSibling;
}
```

Figure 4.3 Node declarations for trees

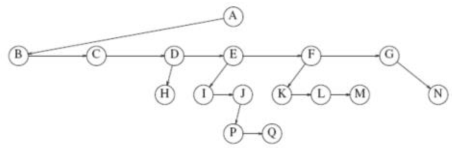


Figure 4.4 First child/next sibling representation of the tree shown in Figure 4.2

Binary trees

- A restricted type of tree that allow efficient implementation and searching
- Maximum of two children – ordered with names left and right!
- Two very important examples
  - Expression trees (general expression tree is not necessarily binary!) Enables efficient storage and conversion of general expressions into efficient code -- compilers
  - Binary search trees allow for efficient searching algorithms
    - » average depth in O(log N) BUT
    - » worst case is O(N)
    - » also provide the basis for thinking about efficient storage and retrieval of large amounts of data

Binary tree implementation in pictures

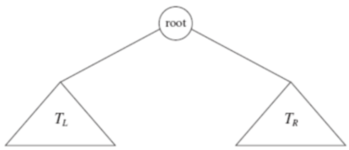


Figure 4.11 Generic binary tree

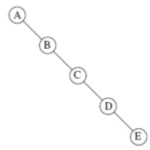


Figure 4.12 Worst-case binary tree

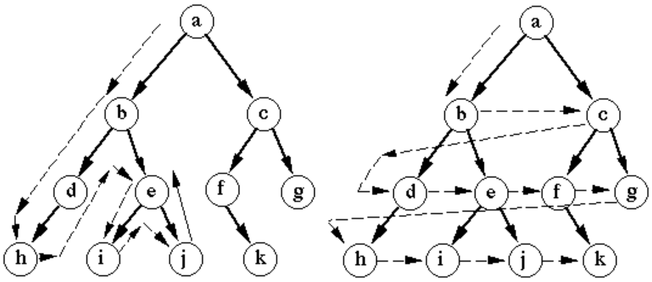
```
class BinaryNode
{
        // Friendly data; accessible by other package routines
    Object     element;      // The data in the node
    BinaryNode left;         // Left child
    BinaryNode right;        // Right child
}
```

Figure 4.13 Binary tree node class

Tree Traversal

- Procedures for systematically visiting every vertex of an ordered tree are called *traversals*.
- Three commonly used tree *traversals* are *preorder traversal*, *inorder traversal*, and *postorder traversal*. These are all examples of Depth First Search
- A fourth traversal Breadth First Search (or Level Order Traversal) is also frequently used

Depth First vs Breadth First Traversals



Depth-first search

Breadth-first search

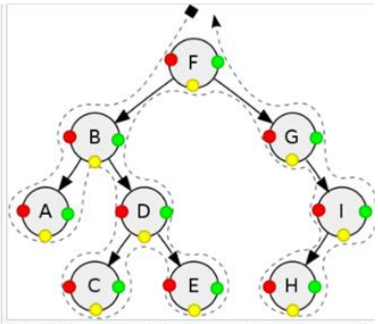
Binary Tree Traversal Implementation

```
def preorder(tree):
    if tree != None:
        visit(tree)
        preorder(tree.getLeftChild())
        preorder(tree.getRightChild())

def inorder(tree):
    if tree != None:
        preorder(tree.getLeftChild())
        visit(tree)
        preorder(tree.getRightChild())

def postorder(tree):
    if tree != None:
        preorder(tree.getLeftChild())
        preorder(tree.getRightChild())
        visit(tree)
```

Traversal Examples



Depth-first traversal of an example tree:
pre-order (red): F, B, A, D, C, E, G, I, H;
in-order (yellow): A, B, C, D, E, F, G, H, I;
post-order (green): A, C, E, D, B, H, I, G, F.

Breadth First Search (Level Order Traversal)

The BFS algorithm starts at the root node and travels through every child node at the current level before moving to the next level.

```
def bfs(self, root=None):
    if root is None:
        return
    queue = [root]
    while len(queue) > 0:
        cur_node = queue.pop(0)
        if cur_node.left is not None:
            queue.append(cur_node.left)
        if cur_node.right is not None:
            queue.append(cur_node.right)
```

Binary Search Trees (BST)

- Binary search trees: all nodes in the left subtree come before the parent, all nodes in the right subtree come after.
- Thus, the objects stored or at least some component, the keys, of the object stored must have an ordering
- To get started we will only consider numbers

Class Node

```
def print_tree(self):

  """   Print tree content inorder      """

  if (self.left != None):
     self.left.print_tree()
  print(self.data)
  if (self.right != None):
     self.right.print_tree()
```

Binary Search Tree Implementation

- Operations on BinarySearchTree object
  - __init__
  - find(self, item)
  - find_min(self)
  - find_max(self)
  - insert(self, item)
  - delete(self, item)
  - Traversals: inorder(self), postorder(self), preorder(self)

Class TreeNode

class TreeNode:

```
def __init____init__(self,key,data=None,left=None,right=None, parent=None):

    self.key = key          # e.g. unique identifier – calpoly id
    self.data = data        # e.g. additional data – current address, …
    self.left = None
    self.right = None
    self.parent = None
```

find/contains

```
def find (self, key):
    p = self.root      # current node
    while p is not None and p.data != key :
        if key < p.data:
            p = p.left
        else:
            p = p.right

    if p.data == key :
        return p           # might want to return data associated with the node or ???
    else:
        return None
```

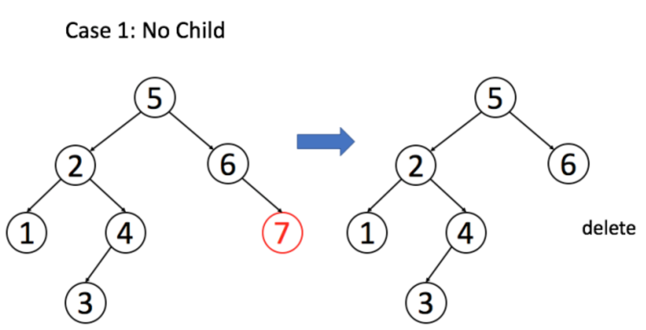

tree.insert(self, newkey)

```
def insert(self, newkey):
    if self.root is None:                # if tree is empty
        self.root = TreeNode(newkey)
        return
    else:
        p = self.root
        if p.key > newkey:
            if p.left is None:
                p.left = TreeNode(newkey)
            else:
                p.left.insert(newkey)
        else:
            if p.right is None:
                p.right = TreeNode(newkey)
            else:
                p.right.insert(newkey)
```

Deleting a node: Three cases

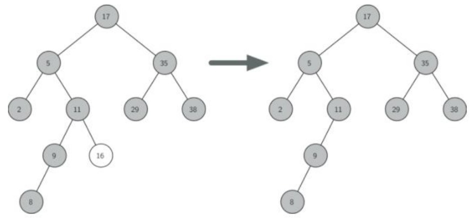
1. The node to be deleted has no children.
2. The node to be deleted has only one child.
3. The node to be deleted has two children.

Node to delete has 0 children

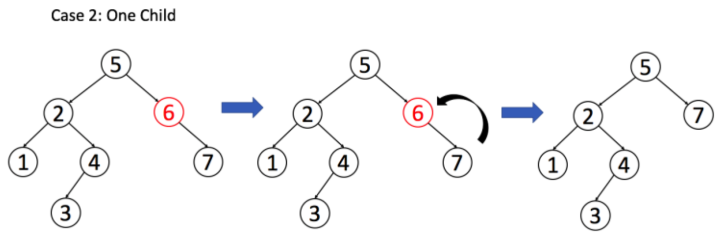


No Children

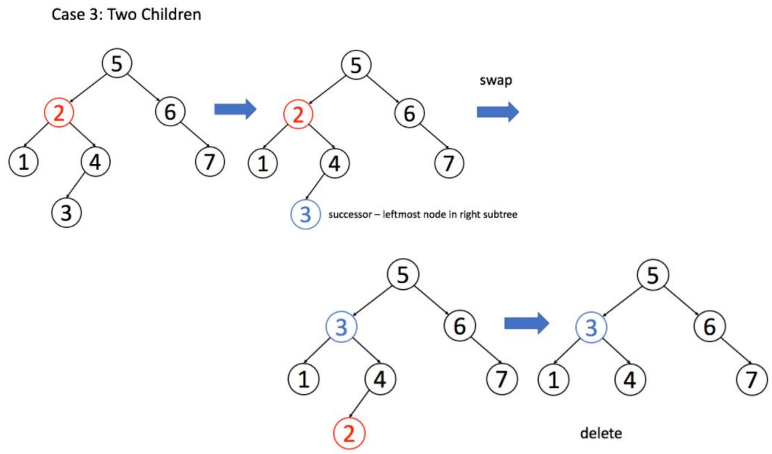
```
if currentNode.isLeaf():
   if currentNode == currentNode.parent.leftChild:
      currentNode.parent.leftChild = None
   else:
      currentNode.parent.rightChild = None
```



Node to delete has 1 child

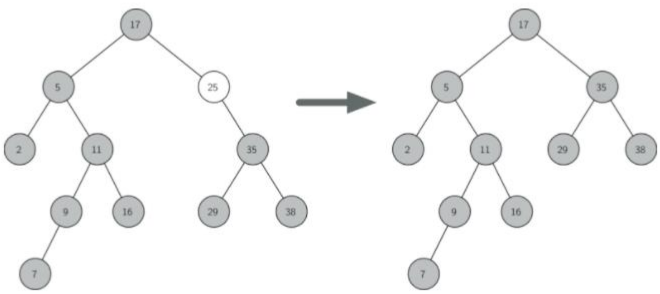


Node to delete has 2 children



33

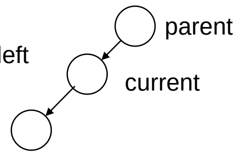
One Child again



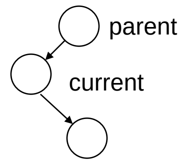
34

One child: Six cases – but symmetry

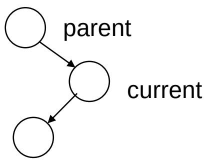
Current has left child, current is left child of parent



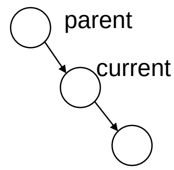
Current has right child, current is left child of parent



Current has left child, current is right child of parent



Current has right child, current is right child of parent

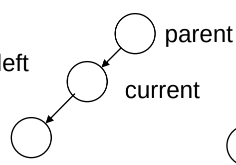


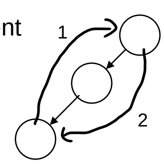
Root to be deleted has left child

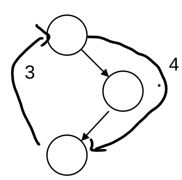
Root to be deleted has right child

One child: details

Current has left child, current is left child of parent








```
if currentNode.hasLeftChild():
        if currentNode.isLeftChild():
1          currentNode.leftChild.parent = currentNode.parent
2          currentNode.parent.leftChild = currentNode.leftChild
        elif currentNode.isRightChild():
3          currentNode.leftChild.parent = currentNode.parent
4          currentNode.parent.rightChild = currentNode.leftChild
        else:
          #  root
```

One Child – half the code

```
else:                                          # this node has one child
  if currentNode.hasLeftChild():                        # has left child
    if currentNode.isLeftChild():
      currentNode.leftChild.parent = currentNode.parent
      currentNode.parent.leftChild = currentNode.leftChild
    elif currentNode.isRightChild():
      currentNode.leftChild.parent = currentNode.parent
      currentNode.parent.rightChild = currentNode.leftChild
    else:                                               # currentNode is root
      currentNode.replaceNodeData(currentNode.leftChild.key,
                    currentNode.leftChild.payload,
                    currentNode.leftChild.leftChild,
                    currentNode.leftChild.rightChild)
  else:                                                 # has a right child
```


find and parent – if no parent field!!

```
def __find_and_parent(self, x):
    '''Search for x, found return None and its parent else none and would-be parent. '''

    q = None         # parent
    p = self.root    # current node
    while p is not None and p.data != x:
      q = p
      if x < p.data:
        p = p.left
      else:
        p = p.right
    return p, q                         #returned as a tuple
```

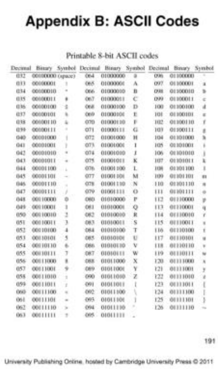
Coding Problem

- *Coding*: assignment of bit strings to alphabet characters
- *Codeword*: bit string assigned to character
- Two types of codes:
  - *fixed-length encoding* (e.g., ASCII)
  - *variable-length encoding* (e,g., Morse code)
- *Prefix-free codes*: no codeword is a prefix of another codeword

Problem: If frequencies of the character occurrences are known, what is the best binary prefix-free code?

Morse and ASCII codes

A •–
B –•••
C –•–•
D –••
E •
F ••–•
G ––•
H ••••
I ••
J •–––
K –•–
L •–••
M ––
N –•
O –––
P •––•
Q ––•–
R •–•
S •••
T –
U ••–
V •••–
W •––
X –••–
Y –•––
Z ––••



Appendix B: ASCII Codes

Printable 8-bit ASCII codes

Decimal	Binary	Symbol	Decimal	Binary	Symbol	Decimal	Binary	Symbol
032	00100000	(space)	064	01000000	@	096	01100000	`
033	00100001	!	065	01000001	A	097	01100001	a
034	00100010	"	066	01000010	B	098	01100010	b
035	00100011	#	067	01000011	C	099	01100011	c
036	00100100	$	068	01000100	D	100	01100100	d
037	00100101	%	069	01000101	E	101	01100101	e
038	00100110	&	070	01000110	F	102	01100110	f
039	00100111	'	071	01000111	G	103	01100111	g
040	00101000	(	072	01001000	H	104	01101000	h
041	00101001	)	073	01001001	I	105	01101001	i
042	00101010	*	074	01001010	J	106	01101010	j
043	00101011	+	075	01001011	K	107	01101011	k
044	00101100	,	076	01001100	L	108	01101100	l
045	00101101	–	077	01001101	M	109	01101101	m
046	00101110	.	078	01001110	N	110	01101110	n
047	00101111	/	079	01001111	O	111	01101111	o
048	00110000	0	080	01010000	P	112	01110000	p
049	00110001	1	081	01010001	Q	113	01110001	q
050	00110010	2	082	01010010	R	114	01110010	r
051	00110011	3	083	01010011	S	115	01110011	s
052	00110100	4	084	01010100	T	116	01110100	t
053	00110101	5	085	01010101	U	117	01110101	u
054	00110110	6	086	01010110	V	118	01110110	v
055	00110111	7	087	01010111	W	119	01110111	w
056	00111000	8	088	01011000	X	120	01111000	x
057	00111001	9	089	01011001	Y	121	01111001	y
058	00111010	:	090	01011010	Z	122	01111010	z
059	00111011	;	091	01011011	[	123	01111011	{
060	00111100	<	092	01011100	\	124	01111100	|
061	00111101	=	093	01011101	]	125	01111101	}
062	00111110	>	094	01011110	^	126	01111110	~
063	00111111	?	095	01011111	_			

191

University Publishing Online, hosted by Cambridge University Press © 2011

Example

- Let Σ = { lower case letters, five punctuation marks and space}
- How can we encode this – 5 bits since $2^5 = 32$
- Is there a way to reduce the length not of the code for each symbol BUT the average length of a message.
- Use frequency of the occurrence of the symbols
  - e, t, a - most frequent -- q, j, x, z – least frequent
  - Normalize so the sum of frequencies is = 1
  - Use frequencies to compute E(symbol length)
  - E.g. Morse code
- Prefix free codes are codes: for all symbols x and y – codeword(x) is not a prefix of codeword(y)
- Decoding is easy - - why?

Huffman codes – key insights

- Any binary tree with edges labeled with 0's and 1's yields a prefix-free code of characters <u>assigned to its leaves</u>
- Optimal binary tree minimizing the expected (weighted average) length of a codeword can be constructed as follows

.

Example

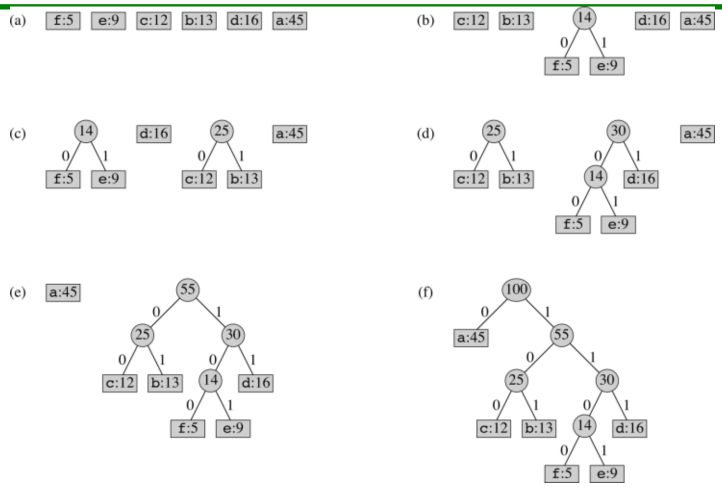
character	A	B	C	D	E
frequency	0.35	0.1	0.2	0.2	0.15
codeword	11	100	00	01	101

average bits per character: 2.25
for fixed-length encoding: 3
compression ratio: (3-2.25)/3*100% = 25%

Huffman's algorithm

- Initialize *n* one-node trees with alphabet characters and the tree weights with their frequencies.
- Repeat the following step *n*-1 times:
  - join two binary trees with smallest weights into one (as left and right subtrees)
  - make the new tree weight equal the sum of the weights of the two subtrees.
- Mark edges leading to left and right subtrees with 0's and 1's, respectively

Constructing a Huffman code tree



Example: Build tree (smallest to left = 0)

character	A	B	C	D	E
frequency	0.32	0.25	0.2	0.18	0.05

Codewords:

H.C. average bits per character:

Fixed-length bits per character:

compression ratio =

Decode: 011110111011011

Assignment 3: Huffman encoding

Extensions and issues

- Image compression
  - Fraction of a bit for a white pixel, higher for black pixel
  - Video/audio – only send changes
- Adaptive encoding
- Many schemes are more effective for particular applications