13: Trees
trees: basic concepts
Hierarchies

- Hierarchically organized "stuff" are everywhere
Trees

```
   aa
  /  
bb   cc
 /    /   /
ee   ff   gg
```

nodes
Trees

- **aa**
- **bb**
- **cc**
- **dd**
- **ee**
- **ff**
- **gg**

- **root node**

- **nodes**
Trees

X: a node

X's parent

X's children
Trees: terminology

• In linear data structures, there is an ordering:
  – before (predecessor)
  – after (successor)

• In hierarchical data structures, the ordering is based on the relationship between:
  – parent and children

   \[ Y \text{ is a child of } X \iff X \text{ is a parent of } Y \]
Trees: terminology

• A tree is a collection of nodes
• Each node has:
  – ≥ 0 child nodes
  – 0 or 1 parent nodes

• A node with 0 children is called a leaf node
• A node with 0 parent nodes is called the root node

• A tree has:
  – ≥ 1 leaf nodes
  – exactly one root node
Trees: leaves and root

root node

leaf nodes
Tree: Example

- Arizona
  - Maricopa
    - Tempe
    - Glendale
  - Pima
  - Coconino
    - Flagstaff
  - Tucson
Tree: Example

\[(7+3)*(5-2)\]
General trees: (formal definition)

A general tree T

is either empty or

is a finite set of nodes N such that one node r in N is the root and the remaining N-{r} nodes are partitioned into disjointed subsets, each of which is a general tree.

This is a recursive definition.
General trees:

A general tree is very flexible
  – hard to manage nodes with many children
  – not used often in practice
  – instead, we limit the number of children

General tree  $\rightarrow$ infinite number of children
Binary tree  $\rightarrow$ 2 children (max)
Binary trees

- A tree where each node has at most two children is called a *binary tree*
Binary trees

left subtree

right subtree
Trees: node representation

• A node in a general tree:
  – value(s) at the node
  – references to child nodes:
    o an extensible data structure (e.g., a list, a linked list, or dictionary)
  – (infrequently) reference to parent

• A node in a binary tree:
  – value(s) at the node
  – a reference to the left subtree
  – a reference to the right subtree
  – (infrequently) reference to parent
Binary trees: node representation

class BinaryTree:
    def __init__(self, value):
        self._value = value  # the value at the node
        self._left = None    # left child
        self._right = None   # right child

...
Trees: terminology

More terms

– sibling: a set of nodes that all share the same parent

– degree: the number of children a node has

– edge/path: the line that connects a parent to its child

– level: the number of "edges" from the root to a node

– height: the maximum of the levels of the nodes (or length of the longest path in the tree)
Exercises-1 & 2

Do problems 1 and 2 in the handout.
binary search trees
Examine this binary tree:

What can we say about the values in the nodes to the left of 8?

What can we say about the values in the nodes to the right of 8?
Binary search tree (BST)

A binary search tree is a binary tree where every node satisfies the following:

- The values at these nodes are smaller than $a$.
- The values at these nodes are bigger than $a$.

value at this node
Binary search tree: Searching
Searching a BST

Given a BST $T$ and a value $v$, is there a node in $T$ with value $v$?
Searching a BST

Given a BST $T$ and a value $v$, is there a node in $T$ with value $v$?

Idea: at each node with value $a$:
- if $a == v$: done
- if $v < a$: search left subtree
- if $v > a$: search right subtree
def search(T, v):
    if T == None:
        return False
    if v == T._value:
        return True
    if v < T._value:
        return search(T._left, v)
    else:
        return search(T._right, v)
Searching a BST: Example 1

\( v < T._\text{value}: \)

\( \Rightarrow \) search left subtree
Searching a BST: Example 1

\[v > T._\text{value}: \Rightarrow \text{search right subtree}\]
Searching a BST: Example 1

\[ v \text{ == } T._\text{value}: \Rightarrow \text{return True}\]
Searching a BST: Example 2

\[ v > T._\text{value}: \Rightarrow \text{search right subtree} \]
Searching a BST: Example 2

\[ v < T._\text{value}: \Rightarrow \text{search left subtree} \]
Searching a BST: Example 2

\( v > T\.value: \)  
\( \Rightarrow \) search right subtree
Searching a BST: Example 2

T == None:
⇒ return False

```python
def search(v, T):
    if T == None:
        return False
    if v == T.value:
        return True
    elif v < T.value:
        return search(v, T.left)
    else:
        return search(v, T.right)
```
Exercise-3

Let's create a BST from this sequence: 7, -2, 10, 0, 13, 14, 3. Must maintain the BST property when we insert a new node.

Let’s add nodes to the tree up on the Elmo.

Do prob. 3 in the handout.
Constructing a BST

Given a BST $T$ and a value $v$, return the tree $T'$ obtained by inserting $v$ into $T$

- if $T$ is empty: return a node with value $v$
- otherwise:
  - if $v < T._\text{value}$ : insert into $T$'s left subtree
  - if $v == T._\text{value}$ : done
  - if $v > T._\text{value}$ : insert into $T$'s right subtree

Exercise-4
Constructing a BST

```python
def insert(T, v):
    if T == None:
        return Node(v)
    if v < T._value:
        T._left = insert(T._left, v)
    elif v > T._value:
        T._right = insert(T._right, v)
    return T
```
Constructing a BST: Example

Sequence of values: 8 3 11 2 9 5

```python
def insert(T, v):
    if T == None:
        return Node(v)
    if v < T.value:
        T.left = insert(T.left, v)
    elif v > T.value:
        T.right = insert(T.right, v)
    return T
```

T: None
Constructing a BST: Example

Sequence of values: 8 3 11 2 9 5

```python
def insert(T, v):
    if T == None:
        return Node(v)
    if v < T.value:
        T.left = insert(T.left, v)
    elif v > T.value:
        T.right = insert(T.right, v)
    return T
```

T: ```
8
```


Constructing a BST: Example

Sequence of values: 8 3 11 2 9 5

```python
def insert(T, v):
    if T is None:
        return Node(v)
    if v < T.value:
        T.left = insert(T.left, v)
    elif v > T.value:
        T.right = insert(T.right, v)
    return T
```

T:

```
  8
```

Constructing a BST: Example

Sequence of values: 8 3 11 2 9 5

```python
def insert(T, v):
    if T == None:
        return Node(v)
    if v < T.value:
        T.left = insert(T.left, v)
    elif v > T.value:
        T.right = insert(T.right, v)
    return T
```

T:

```
  8
```

40
Constructing a BST: Example

Sequence of values: \(8 \boxed{3} 11 \ 2 \ 9 \ 5\)

def insert(T, v):
    if T == None:
        return Node(v)
    if v < T.value:
        T.left = insert(T.left, v)
    elif v > T.value:
        T.right = insert(T.right, v)
    return T
Constructing a BST: Example

Sequence of values: 8 3 11 2 9 5

def insert(T, v):
    if T == None:
        return Node(v)
    if v < T.value:
        T.left = insert(T.left, v)
    elif v > T.value:
        T.right = insert(T.right, v)
    return T
Constructing a BST: Example

Sequence of values: 8 3 11 2 9 5

```
def insert(T, v):
    if T == None:
        return Node(v)
    if v < T.value:
        T.left = insert(T.left, v)
    elif v > T.value:
        T.right = insert(T.right, v)
    return T
```
Constructing a BST: Example

Sequence of values: 8 3 11 2 9 5

def insert(T, v):
    if T == None:
        return Node(v)
    if v < T.value:
        T.left = insert(T.left, v)
    elif v > T.value:
        T.right = insert(T.right, v)
    return T
Constructing a BST: Example

Sequence of values: 8 3 11 2 9 5

def insert(T, v):
    if T == None:
        return Node(v)
    if v < T.value:
        T.left = insert(T.left, v)
    elif v > T.value:
        T.right = insert(T.right, v)
    return T
Constructing a BST: Example

Sequence of values: 8 3 11 2 9 5

def insert(T, v):
    if T == None:
        return Node(v)
    if v < T.value:
        T.left = insert(T.left, v)
    elif v > T.value:
        T.right = insert(T.right, v)
    return T
Constructing a BST: Example

Sequence of values: 8 3 11 2 9 5

```python
def insert(T, v):
    if T == None:
        return Node(v)
    if v < T.value:
        T.left = insert(T.left, v)
    elif v > T.value:
        T.right = insert(T.right, v)
    return T
```

![BST Diagram](image-url)
Constructing a BST: Example

Sequence of values: 8 3 11 2 9 5

```python
def insert(T, v):
    if T == None:
        return Node(v)
    if v < T.value:
        T.left = insert(T.left, v)
    elif v > T.value:
        T.right = insert(T.right, v)
    return T
```

```mermaid
graph TD
T[8]
B[2] --> A
D[9] --> C
E[5] --> C
```

1. Insert 2:
   - New node with value 2 is added as left child of node 3.

2. Insert 9:
   - New node with value 9 is added as right child of node 11.

3. Insert 5:
   - New node with value 5 is added as right child of node 11.

Resulting BST:
- Root: 8
- Left child of 8: 3
- Right child of 8: 11
- Left child of 3: 2
- Right child of 3: 9
- Right child of 11: 5
Constructing a BST: Example

Sequence of values: 8 3 11 2 9 5

def insert(T, v):
    if T == None:
        return Node(v)
    if v < T.value:
        T.left = insert(T.left, v)
    elif v > T.value:
        T.right = insert(T.right, v)
    return T
Constructing a BST: Example

Sequence of values: 8 3 11 2 9 5

```python
def insert(T, v):
    if T == None:
        return Node(v)
    if v < T.value:
        T.left = insert(T.left, v)
    elif v > T.value:
        T.right = insert(T.right, v)
    return T
```
Constructing a BST: Example

Sequence of values: 8 3 11 2 9 5

def insert(T, v): (v = 9, T.value = 11)
  if T == None:
    return Node(v)
  if v < T.value:
    T.left = insert(T.left, v)
  elif v > T.value:
    T.right = insert(T.right, v)
  return T
Constructing a BST: Example

Sequence of values: 8 3 11 2 9 5

def insert(T, v):
    if T == None:
        return Node(v)
    if v < T.value:
        T.left = insert(T.left, v)
    elif v > T.value:
        T.right = insert(T.right, v)
    return T
Constructing a BST: Example

Sequence of values: 8 3 11 2 9 5

def insert(T, v):
    if T == None:
        return Node(v)
    if v < T.value:
        T.left = insert(T.left, v)
    elif v > T.value:
        T.right = insert(T.right, v)
    return T
Constructing a BST: Example

Sequence of values: 8 3 11 2 9 5

```python
def insert(T, v):
    if T == None:
        return Node(v)
    if v < T.value:
        T.left = insert(T.left, v)
    elif v > T.value:
        T.right = insert(T.right, v)
    return T
```

```plaintext
def insert(T, v):
    if T == None:
        return Node(v)
    if v < T.value:
        T.left = insert(T.left, v)
    elif v > T.value:
        T.right = insert(T.right, v)
    return T
```
Constructing a BST: Example

Sequence of values: 8 3 11 2 9 5

def insert(T, v):
    if T == None:
        return Node(v)
    if v < T.value:
        T.left = insert(T.left, v)
    elif v > T.value:
        T.right = insert(T.right, v)
    return T
Constructing a BST: Example

Sequence of values: 8 3 11 2 9 5

```python
def insert(T, v):
    if T == None:
        return Node(v)
    if v < T.value:
        T.left = insert(T.left, v)
    elif v > T.value:
        T.right = insert(T.right, v)
    return T
```
Constructing a BST: Example

Sequence of values: 8 3 11 2 9

```python
def insert(T, v):
    if T == None:
        return Node(v)
    if v < T.value:
        T.left = insert(T.left, v)
    elif v > T.value:
        T.right = insert(T.right, v)
    return T
```

Diagram of BST:
Constructing a BST: Example

Sequence of values: 8 3 11 2 9 5
Does Maricopa come before Coconino? If so, why?
Are the leaves more important than the nodes with children?
What does "order" mean here?
tree traversals
Tree traversals

• A traversal of a tree is a systematic way of visiting and processing the nodes of the tree

This usually comes down to the relative order between:
- traversing the subtrees of the node's children; and
- processing the node

"Doing something with the value at the node"
- e.g., printing it out
Tree traversals (n-ary)

There are three widely used traversals:

• **Preorder traversal**
  – process the node first
  – then traverse (and process) its children

• **Inorder traversal**
  – traverse left subtree children
  – then process the node
  – then traverse right subtree

• **Postorder traversal**
  – traverse (and process) the children
  – then process the node

"pre" – visit node first
"in" – visit node in between
"post" – visit node last
Tree traversals (binary)

There are three widely used traversals:

- **Preorder traversal**
  - process the node first
  - then traverse (and process) its children

- **Inorder traversal**
  - traverse left subtree children
  - then process the node
  - then traverse right subtree

- **Postorder traversal**
  - traverse (and process) the children
  - then process the node
# BinaryTree Traversals

## 3 Traversals

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>preorder:</td>
<td>Visit</td>
<td>-----</td>
<td>-----</td>
</tr>
<tr>
<td>inorder:</td>
<td>-----</td>
<td>Visit</td>
<td>-----</td>
</tr>
<tr>
<td>postorder:</td>
<td>-----</td>
<td>-----</td>
<td>Visit</td>
</tr>
</tbody>
</table>
Preorder traversal

Algorithm:
- Visit the node
- Recurse on node's Left subtree
- Recurse on node's Right subtree

Ex:

```
A
/   \
B     D
/   /
C   
```

(where's the base case?)
Trace of preorder traversal

<table>
<thead>
<tr>
<th>Output</th>
<th>Trace</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Call preorder(A)</td>
</tr>
<tr>
<td></td>
<td>Visit A</td>
</tr>
<tr>
<td></td>
<td>Left (call preorder(B))</td>
</tr>
<tr>
<td>B</td>
<td>Visit B</td>
</tr>
<tr>
<td></td>
<td>Left – return immediately</td>
</tr>
<tr>
<td></td>
<td>Right (call preorder(C))</td>
</tr>
<tr>
<td>C</td>
<td>Visit C</td>
</tr>
<tr>
<td></td>
<td>Left – return</td>
</tr>
<tr>
<td></td>
<td>Right – return</td>
</tr>
<tr>
<td>D</td>
<td>Visit D</td>
</tr>
<tr>
<td></td>
<td>Left – return</td>
</tr>
<tr>
<td></td>
<td>Right – return</td>
</tr>
</tbody>
</table>
Preorder traversal (n-ary)

def preorder(T):
    process(T._value)
    for i in range(len(T._children)):
        preorder(T._children[i])

(where's the base case?)
Preorder traversal: Example

Title

Abstract

Sec 1

Sec 2

Sec 3

References

Sec 1.1

Sec 1.2

Sec 2.1

Sec 2.2
Preorder traversal: Example

Title
Abstract
...
Sec 1
...
Sec 1.1
...
Sec 1.2
...
Sec 2
...
Sec 2.1
...
Sec 2.2
...
Sec 3
...
References
Inorder traversal

Algorithm:

- Recurse on node's Left subtree
- Visit node
- Recurse on node's Right subtree

Ex:

B C A D
Inorder traversal (binary trees)

def inorder(T):
    if T == None:
        return
    else:
        inorder(T.left())
        process(T.value)
        inorder(T.right())
Inorder traversal: Example

Print out the values in a BST in sorted order
Postorder traversal

Algorithm:
Recursively visit node's Left subtree
Recursively visit node's Right subtree
Visit node

Ex:

```
C      B      D      A
```

(where's the base case?)
Postorder traversal (n-ary)

def postorder(T):
    for i in range(len(T.children)):
        postorder(T.children[i])  # visit all children first
    process(T.value)
Postorder traversal - Example

Problem: evaluate this expression

\[(x + y \times 3) / (n - 1)\]

suppose that: \(x = 3, y = 2, n = 4\)

Solution:

1) Convert the expression to post-fix notation
   - use an auxiliary stack
   - use the algorithm in the handout

2) Represent the expression as a binary tree
   - use a postorder traversal of the tree
Postorder traversal: Example

Evaluate: \( (x + y \times 3) / (n - 1) \)
suppose that: \( x = 3, y = 2, n = 4 \)
Postorder traversal: Example

Evaluate: \((x + y \times 3) / (n - 1)\)
suppose that: \(x = 3, y = 2, n = 4\)
Trees ↔ __str__

How should we print a tree?
Trees ↔ __str__

Use parentheses to indicate the structure:

( A ( B None None ) (D None None ) )
Trees ↔ traversals

What is the preorder traversal of this tree?
Inorder?
Postorder?
Trees ↔ traversals

• Given a tree, we can figure out its traversals
• Does the converse hold?
  I.e., given a traversal, can we figure out the tree?
  preorder: 1  2  3  4
Trees ↔ traversals

• The two trees below have the same preorder traversal.

Preorder traversal = 1 2 3 4
Exercise

Draw two trees that have the in-order traversal: 1 2 3 4
Solution

• Given just a traversal, we cannot uniquely figure out the tree it came from

Example: Inorder = 1 2 3 4
Trees ↔ traversals

• We cannot derive a *unique* tree from a single traversal

• Why can we get more than one tree shape?
  – Inorder: 3 5 7 9 4
    o Can make a tree with different root nodes with this traversal

• How can we get a unique tree?
  – Idea: provide a second traversal
  – Preorder: 5 3 9 7 4
    o This specifies the root

• Let's draw the tree given the two traversals
  – Can we create an algorithm to do this?
Exercise

Draw the tree given the two traversals below:

preorder:  8  3  2  5  11  9
inorder:  2  3  5  8  9  11
Trees ↔ traversals

Preorder:

\[
\text{Pre} (T_{\text{left}}) \quad \text{Pre} (T_{\text{right}})
\]

Inorder:

\[
\text{In} (T_{\text{left}}) \quad \text{In} (T_{\text{right}})
\]

Same size
Trees ↔ traversals

• Given a preorder \textit{and} an inorder traversal, create the tree

• Given:
  
  – preorder\_list
  
  – inorder\_list

\begin{equation}
\text{node sequences from traversals of a tree}
\end{equation}

Need to do: build a function:

\[
\text{traversals\_to\_tree(}
\text{preorder\_list, inorder\_list)}
\]

that will return the tree for the given traversals.
Trees ↔ traversals

• Given:
  - preorder_list + inorder_list

• Suppose we can figure out:
  - root
  - preorder_left + preorder_right
  - inorder_left + inorder_right
Trees ↔ traversals

• Given:
  - preorder_list + inorder_list

• Suppose we can figure out:
  - root
  - preorder_left + preorder_right
  - inorder_left + inorder_right

• Then:
  traversals_to_tree(preorder_left, inorder_left)
  traversals_to_tree(preorder_right, inorder_right)
Trees ↔ traversals

• Given:
  – preorder_list + inorder_list

• Suppose we can figure out:
  – root
  – preorder_left + preorder_right
  – inorder_left + inorder_right

• Then:

  traversals_to_tree(preorder_left, inorder_left)
  traversals_to_tree(preorder_right, inorder_right)
Trees ↔ traversals

• Given:
  - preorder_list + inorder_list

• Suppose we can figure out:
  - root
  - preorder_left + preorder_right
  - inorder_left + inorder_right

• Then:

  \[
  \text{traversals}_\text{to}_\text{tree}(\text{preorder}_\text{left}, \text{inorder}_\text{left}) \rightarrow T_{\text{left}}
  \]
  \[
  \text{traversals}_\text{to}_\text{tree}(\text{preorder}_\text{right}, \text{inorder}_\text{right}) \rightarrow T_{\text{right}}
  \]
Trees ↔ traversals

Preorder: $a$

Inorder: $a$

$T_{left}$

$T_{right}$
more traversals
Consider: game playing

Goal: to write a program to play a 2-person game (e.g., tic-tac-toe, chess, go, ...)

How does this work?
Consider: game playing

Goal: to write a program to play a 2-person game (e.g., tic-tac-toe, chess, go, ...)

Generate successive levels of board positions
• At each level, pick best move for the player at that level
• Work backwards to find the move that will lead to the best position $n$ moves later

*best position for the computer among all positions at this level*
Consider: game playing

• For a nontrivial game (e.g., chess, go) the tree is usually too large to build or explore fully
  • also, usually there are time constraints on play
  • our previous tree traversal algorithms don't work

• Game-playing algorithms typically explore the tree level by level
  • consider the nodes at depth 1, then depth 2, etc.
Level-by-level tree traversal
Level-by-level tree traversal
Level-by-level tree traversal

This order of traversal is called *breadth-first traversal*
Breadth-first tree traversal

Breadth-first traversal order:

1 2 3 4 5 6 7 8 9 10
Breadth-first tree traversal

Data structure: use a queue $q$

Algorithm:

• for each level in the tree:
  – enqueue the nodes at that level
  – while queue not empty:
    o node = $q$.dequeue()
    o process node
Breadth-first tree traversal

Data structure: use a queue $q$

Algorithm:

• for each level in the tree:
  – enqueue the nodes at that level
  – while queue not empty:
    o node = $q$.dequeue()
    o process node
Breadth-first tree traversal

Data structure: use a queue $q$

Algorithm:

• for each level in the tree:
  – enqueue the nodes at that level
  – while queue not empty:
    o node = $q$.dequeue()
    o process node
Breadth-first tree traversal

Data structure: use a queue $q$

Algorithm:

• for each level in the tree:
  - enqueue the nodes at that level
  - while queue not empty:
    o node = $q$.dequeue()
    o process node
Breadth-first tree traversal

Data structure: use a queue $q$

Algorithm:

• for each level in the tree:
  – enqueue the nodes at that level
  – while queue not empty:
    o node = $q$.dequeue()
    o process node
Breadth-first tree traversal

Data structure: use a queue $q$

Algorithm:

• for each level in the tree:
  – enqueue the nodes at that level
  – while queue not empty:
    o node = $q$.dequeue()
    o process node
Breadth-first tree traversal

Data structure: use a queue $q$

Algorithm:
• for each level in the tree:
  – enqueue the nodes at that level
  – while queue not empty:
    o node = $q$.dequeue()
    o process node
Breadth-first tree traversal

Data structure: use a queue $q$

Algorithm:
• for each level in the tree:
  – enqueue the nodes at that level
  – while queue not empty:
    o node = $q$.dequeue()
    o process node
Breadth-first tree traversal

Data structure: use a queue \( q \)

Algorithm:

• for each level in the tree:
  – enqueue the nodes at that level
  – while queue not empty:
    o node = \( q.\text{dequeue}() \)
    o process node
Breadth-first tree traversal

Data structure: use a queue \( q \)

Algorithm:

- for each level in the tree:
  - enqueue the nodes at that level
  - while queue not empty:
    - node = \( q.\text{dequeue}() \)
    - process node
Breadth-first tree traversal

Data structure: use a queue $q$

Algorithm:
• for each level in the tree:
  – enqueue the nodes at that level
  – while queue not empty:
    o node = $q$.dequeue()
    o process node
Breadth-first tree traversal

Data structure: use a queue $q$

Algorithm:

• for each level in the tree:
  – enqueue the nodes at that level
  – while queue not empty:
    o node = $q$.dequeue()
    o process node
Breadth-first tree traversal

Data structure: use a queue $q$

Algorithm:
- for each level in the tree:
  - enqueue the nodes at that level
  - while queue not empty:
    - node = $q$.dequeue()
    - process node

```
10  9  8  7  6  5
```

node None

[Diagram of a tree with nodes 1 to 10, showing the queue with the order 10, 9, 8, 7, 6, 5]
Breadth-first tree traversal

Data structure: use a queue $q$

Algorithm:

• for each level in the tree:
  – enqueue the nodes at that level
  – while queue not empty:
    o node = $q$.dequeue()
    o process node
Breadth-first tree traversal

Data structure: use a queue $q$

Algorithm:

- for each level in the tree:
  - enqueue the nodes at that level
  - while queue not empty:
    - node = $q$.dequeue()
    - process node

... etc. ...
Breadth-first tree traversal

Data structure: use a queue $q$

Algorithm:
• Create a queue $q$
• Put the root in $q$
• While $q$ not empty
  o node = $q$.dequeue()
  o process node
  o enqueue its children
Breadth-first vs. Depth-first

• Stacks and queues are closely related structures
• What if we use a stack in our tree traversal?
  – the deeper levels of the tree are explored first
  – this is referred to as depth-first traversal
BST / Complexity
Binary Search tree: complexity

Searching: $O(\log n)$, where $n$ is the number of elements in the tree

Note: this tree is balanced

What if the tree is not balanced?
BST / Complexity

• Unbalanced BST
  – How many comparisons does it take to find 2?
  – Worst case, complexity can be $O(n)$
  – Skewed trees can result from sorted input

• Balanced trees
  – AVL Trees: trees are kept balanced on insertion, deletion, etc.
Trees: summary

• An n-ary tree represents a hierarchy

• A binary tree represents a sequence

• They show up in all kinds of contexts
  – including many in computer science

• Various kinds of tree traversals reflect different ways of processing the information and structure of trees

• Recursion is often the simplest way to process trees
Encoding characters

- ASCII uses fixed length encodings:

<table>
<thead>
<tr>
<th>char</th>
<th>ASCII</th>
<th>binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>g</td>
<td>103</td>
<td>1100111</td>
</tr>
<tr>
<td>o</td>
<td>111</td>
<td>1101111</td>
</tr>
<tr>
<td>p</td>
<td>112</td>
<td>1110000</td>
</tr>
<tr>
<td>h</td>
<td>104</td>
<td>1101000</td>
</tr>
<tr>
<td>e</td>
<td>101</td>
<td>1100101</td>
</tr>
<tr>
<td>r</td>
<td>114</td>
<td>1110010</td>
</tr>
<tr>
<td>s</td>
<td>115</td>
<td>1110011</td>
</tr>
</tbody>
</table>

What is this word?
11010001101111111001011100111100101

binary: 1101000 1101111 1110010 1110011 1100101 (group by 7's)
ASCII: 104 111 114 115 101
Encoding characters

• ASCII
  – The common alphanumeric characters use 7 bits
    o 128 possibilities ($2^7$)
  – The encoding for horse is:
    1101000 1101111 1110010 1110011 1100101
    h o r s e
  – Takes 35 bits to encode 'horse'
  – Question: if we only used the characters in the previous table, could we come up with a different encoding scheme?
  – Need to represent 7 characters and a space: total of 8
Encoding characters

• 8 characters total: needs only 3 bits
• 3-bit encoding

<table>
<thead>
<tr>
<th>char</th>
<th>code</th>
<th>binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>g</td>
<td>0</td>
<td>000</td>
</tr>
<tr>
<td>o</td>
<td>1</td>
<td>001</td>
</tr>
<tr>
<td>p</td>
<td>2</td>
<td>010</td>
</tr>
<tr>
<td>h</td>
<td>3</td>
<td>011</td>
</tr>
<tr>
<td>e</td>
<td>4</td>
<td>100</td>
</tr>
<tr>
<td>r</td>
<td>5</td>
<td>101</td>
</tr>
<tr>
<td>s</td>
<td>6</td>
<td>110</td>
</tr>
<tr>
<td>space</td>
<td>7</td>
<td>111</td>
</tr>
</tbody>
</table>

011 001 101 110 100
3 1 5 6 4

takes 15 bits
Huffman Coding

• The idea behind Huffman coding:
  – Use fewer bits (not 7) for more frequently occurring characters
  – (for example, vowels occurs in text frequently)
  – Variable number of bits for characters

• But, if representation of a character is variable, how would you "decode"? How many bits go with each character?
  – 110100011011111100101110011100101
Huffman Coding

• Use a tree
  – the ASCII characters are the leaves
  – left edges are 0
  – right edges are 1
  – root-to-leaf paths provide the bit sequence used to encode the characters
  – concatenate the path to get the bit sequence
Huffman Coding

Source: Astrachan

path: right-right-left-left-left-left-right  using the 0/1 convention
Huffman Coding

The structure of the tree can be used to determine the coding of any leaf by using the 0/1 edge convention. A different tree gives a different coding.

The tree below gives the coding on the right.

Source: Astrachan
Huffman Coding

The structure of the tree can be used to determine the coding of any leaf by using the 0/1 edge convention. A different tree gives a different coding.

The tree below gives the coding on the right.

Source: Astrachan

What would the encoding be for hope?
Huffman Coding

Prefix codes/Huffman codes

prefix property: no bit-sequence encoding of a character is the prefix of any other bit-sequence encoding.

When all characters are stored in leaves and every non-leaf node has two children, the coding produced by the 0/1 convention has the prefix property

invented by Huffman 1952