Data Structures

Trees I

CS284

Objectives

- To learn how to use a tree to represent a hierarchical organization of information
- To learn how to use recursion to process trees
- To understand the different ways of traversing a tree
- To understand the difference between binary trees, binary search trees, and heaps
- To learn how to implement binary trees, binary search trees, and heaps using linked data structures and arrays

Trees - Introduction

- All previous data organizations we've learned are linear—each element can have only one predecessor or successor
- Accessing all elements in a linear sequence is $\mathcal{O}(n)$
- Trees are nonlinear and hierarchical
- Tree nodes can have multiple successors (but only one predecessor)
- Trees are recursive data structures because they can be defined recursively

Binary Trees

Definition and Terminology

Tree Expressions

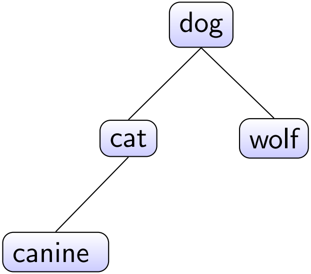
More Examples of Trees

Binary Search Trees

Tree Traversals

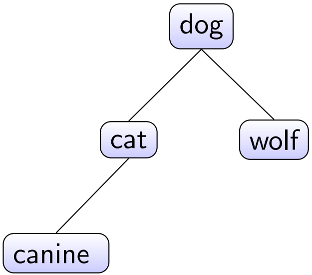
Binary Trees

- We first focus on binary trees
- In a binary tree each element has at most two successors



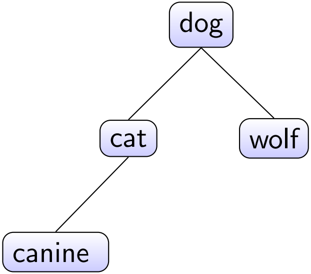
Binary Trees – Terminology

- Node
- Root
- Branches: links between nodes
- Children: successors of a node
- Parent (how many? root?): predecessor of a node
- Siblings: nodes with the same parent

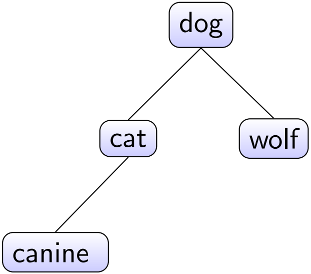


Binary Trees – Terminology (cont.)

- Internal node
- Leaf (= external node)
- Ancestor: generalization of parent-child
- Subtree (of a node): tree whose root is a child of that node



Binary Trees – Terminology (cont.)



Level 1

Level 2

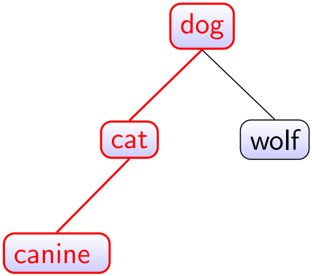
Level 3

In words:

- If node n is the root of tree T, its level is 1
- If node n is not the root of tree T, its level is $1 +$ the level of its parent

Binary Trees – Terminology (cont.)

Height: number of nodes in the longest path the root to a leaf



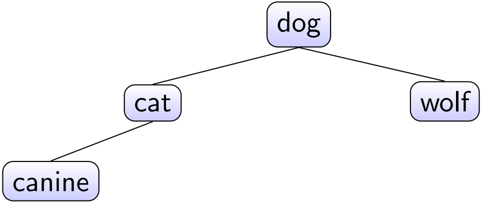
Height is 3 in this example

Tree Expressions

- We can represent trees using tree expressions
- Tree expressions are useful for pencil-and-paper analysis of properties of binary trees
- The set `'a btree` of binary tree expressions over a set `'a` can be defined recursively as follows:
  - `Empty` is an empty binary tree
  - `Node(i,l,r)` is an internal node that has information `i∈'a` and subtrees `l` and `r`

```
type 'a btree = Empty | Node of 'a * 'a btree * 'a btree
```

Tree Expressions




```
Node("dog",
  Node("cat",
       Node("canine",Empty,Empty),
       Empty),
  Node("wolf",Empty,Empty))
```

Revisiting the Height using Tree Expressions

```
let rec height = function
  | Empty -> 0
  | Node(i,lt,rt) -> 1+ max (height lt) (height rt)
```

Example:

```
height(Node("dog",Node("cat",Node("canine",Empty,Empty),Empty)
        Node("wolf",Empty,Empty)))
= 1 + max(height(Node("cat",Node("canine",Empty,Empty),Empty))
             height(Node("wolf",Empty,Empty)))
= 1 + max(1+max(height(Node("canine",Empty,Empty)),height(Empt
             height(Node("wolf",Empty,Empty)))
= 1 + max(1+max(1+max(0,0),0),1+max(0,0))
= 1 + max(2,1)
= 3
```

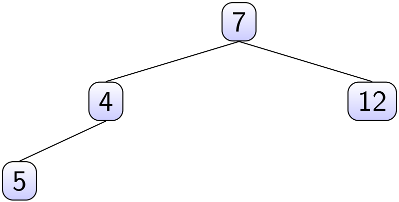
Another Example – The Number of Nodes

```
let rec no_of_nodes = function
  | Empty -> 0
  | Node(i,lt,rt) -> 1+(no_of_nodes lt)+(no_of_nodes rt)
```

Example:

```
no_of_nodes(Node("dog",Node("cat",Node("canine",Empty,Empty),Em
        Node("wolf",Empty,Empty)))
= 1 + no_of_nodes(Node("cat",Node(canine,Empty,Empty),Empty)) +
         no_of_nodes(Node(wolf,Empty,Empty))
= 1 + 1+ no_of_nodes(Node(canine,Empty,Empty)) +
        no_of_nodes(Node(wolf,Empty,Empty)))
= 1 + 1 + 1 + 0 + 1 + 0
= 4
```

Another Example – Sum Tree



Exercise: Write a function `sumT` that adds all the numbers in the tree.
Example:

```
sumT(Node(7,Node(4,Node(5,Empty,Empty)), Empty),
     Node(12,Empty,Empty)))
=  28
```

Another Example – `isEmpty`

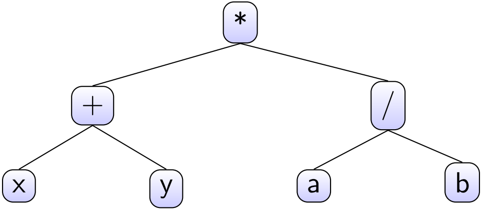
Exercise: Write a function `isEmpty` that returns true if the tree is empty and false otherwise

Example:

```
 isEmpty(Node(7,Node(4,Node(5,Empty,Empty)), Empty),
    Node(12,Empty,Empty)))
=  false
```

Arithmetic Expression Tree

- Each node contains an operator or an operand
- Operands are stored in leaf nodes
- Parentheses are not stored in the tree because the tree structure dictates the order of operand evaluation
- Operators in nodes at higher levels are evaluated after operators in nodes at lower levels

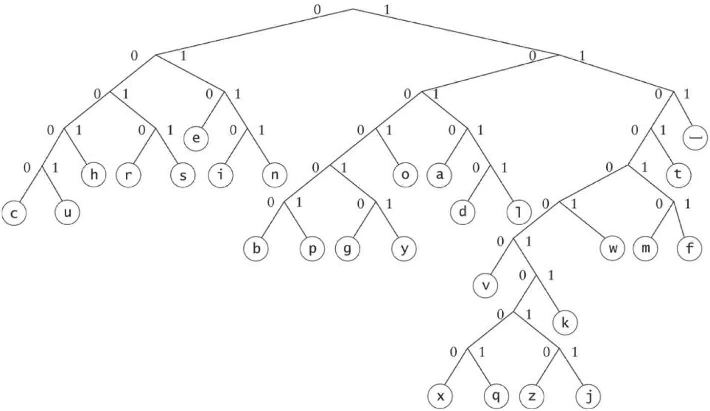


(x + y) * (a / b)

Huffman

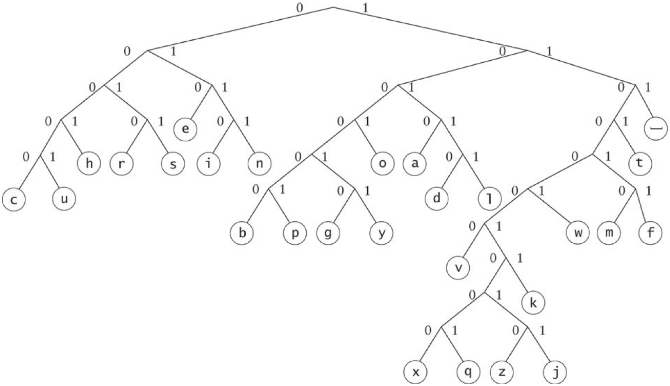
- A Huffman tree represents Huffman codes for characters that might appear in a text file
- As opposed to ASCII or Unicode, Huffman code uses different numbers of bits to encode letters; more common characters use fewer bits
- Many programs that compress files use Huffman codes

Huffman Tree



To form a code, traverse the tree from the root to the chosen character, appending 0 if you turn left, and 1 if you turn right.

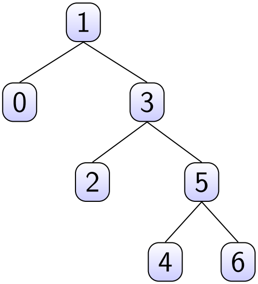
Huffman Tree



Examples: d : 10110 e : 010

Binary Search Tree

- All elements in the left subtree precede those in the right subtree
- A formal definition: A binary tree T is a binary search tree if either of the following is true:
  - $T = Empty$
  - If $T = Node(i, l, r)$, then
    - l and r are binary search trees and
    - i is greater than all values in l and i is less than all values in r



BST Predicate using Tree Expressions

```
let rec is_bst = function
  | Empty -> true
  | Node(i,lt,rt) -> (max_tree lt)<i && (min_tree rt)>i
                               && is_bst(lt) && is_bst(rt)
```

Note

- What is the maximum/minimum of an empty tree?
- Better to avoid computing those when `lt` or `rt` are `Empty`
- Can you modify the above definition accordingly?

Binary Search Tree

- A binary search tree never has to be sorted because its elements always satisfy the required order relations
- When new elements are inserted (or removed) properly, the binary search tree maintains its order
- In contrast, an array must be expanded whenever new elements are added, and compacted when elements are removed—expanding and contracting are both $\mathcal{O}(n)$

BST – Find – Using Tree Expressions

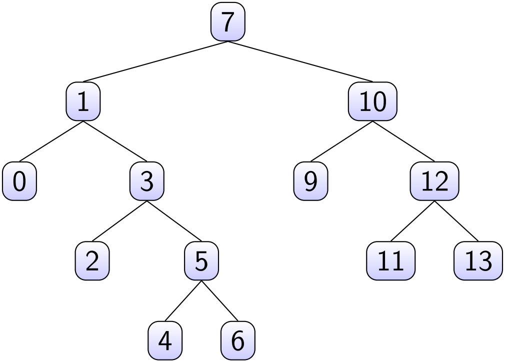
- Search for a target `key`

```
let rec find key = function
  | Empty -> failwith("Not found")
  | Node(i,lt,rt) when key=i -> true
  | Node(i,lt,rt)  ->
     if (key<i)
     then find key lt
     else find key rt
```

- Each probe has the potential to eliminate half the elements in the tree, so searching can be $\mathcal{O}(\log n)$
- In the worst case though, it is $\mathcal{O}(n)$

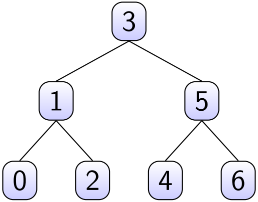
Full, Perfect, and Complete Binary Trees (cont.)

A full binary tree is a binary tree where all nodes have either 2 children or 0 children (the leaf nodes)



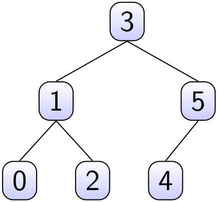
Full, Perfect, and Complete Binary Trees (cont.)

- A perfect binary tree is
  1. a full binary tree of height n
  2. all leaves have the same depth
- Above def. equivalent to requiring that the tree have exactly $2^n - 1$ nodes, n being the height
- In this case, $n = 3$ and $2^n - 1 = 7$



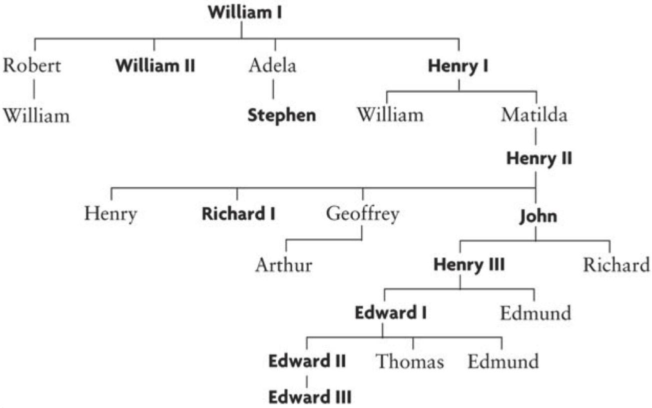
Full, Perfect, and Complete Binary Trees (cont.)

- A complete binary tree is a perfect binary tree through level $n - 1$ with some extra leaf nodes at level n (the tree height), all toward the left



General Trees

Nodes of a general tree can have any number of subtrees



Tree Traversals

- Often we want to determine the nodes of a tree and their relationship
- We can do this by walking through the tree in a prescribed order and visiting the nodes as they are encountered
- This process is called tree traversal
- Three common kinds of tree traversal
  - Inorder
  - Preorder
  - Postorder

Tree Traversals

- Preorder: visit root node, traverse TL, traverse TR
- Inorder: traverse TL, visit root node, traverse TR
- Postorder: traverse TL, traverse TR, visit root node

Algorithm for Preorder Traversal

1. `if` the tree is empty
2. Return.

   `else`

3. Visit the root.
4. Preorder traverse the left subtree.
5. Preorder traverse the right subtree.

Algorithm for Inorder Traversal

1. `if` the tree is empty
2. Return.

   `else`

3. Inorder traverse the left subtree.
4. Visit the root.
5. Inorder traverse the right subtree.

Algorithm for Postorder Traversal

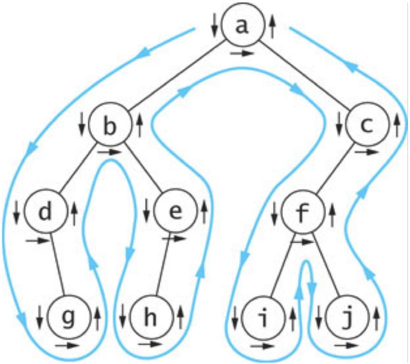
1. `if` the tree is empty
2. Return.

   `else`

3. Postorder traverse the left subtree.
4. Postorder traverse the right subtree.
5. Visit the root.

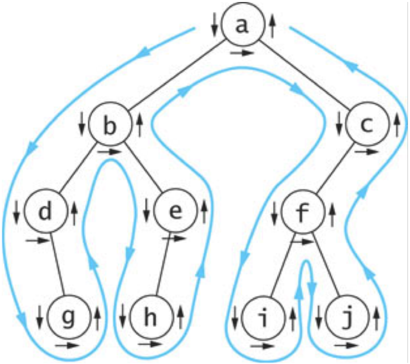
Visualizing Tree Traversals

- You can visualize a tree traversal by imagining a mouse that walks along the edge of the tree
- If the mouse always keeps the tree to the left, it will trace a route known as the Euler tour
- The Euler tour is the path traced in blue in the figure on the right



Visualizing Tree Traversals

- An Euler tour (blue path) is a preorder traversal
- The
  sequence in this example is
  `a b d g e h c f i j`
- The mouse visits each node before traversing its subtrees (shown by the downward pointing arrows)



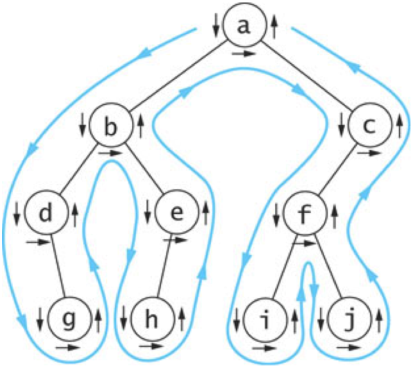
Preorder Traversal using Expression Trees

```
let rec preO = function
    | Empty -> []
    | Node(i,l,r) -> [i] @ (preO l) @ (preO r)
```

- Here `[]` denotes the empty list and `@` denotes list concatenation

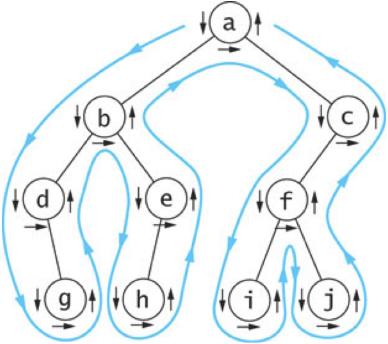
Visualizing Tree Traversals

- If we record a node as the mouse returns from traversing its left subtree (horizontal black arrows in the figure) we get an inorder traversal
- The sequence is
  `d g b h e a i f j c`



Visualizing Tree Traversals

- If we record each node as the mouse last encounters it, we get a postorder traversal (shown by the upward pointing arrows)
- The sequence is
  `g d h e b i j f c a`



Traversals of Binary Search Trees and Expression Trees

- An inorder traversal of a binary search tree results in the nodes being visited in sequence by increasing data value

canine, cat, dog, wolf