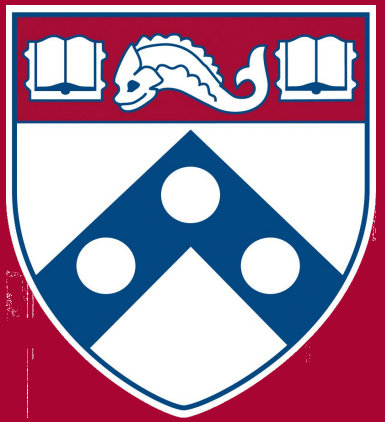
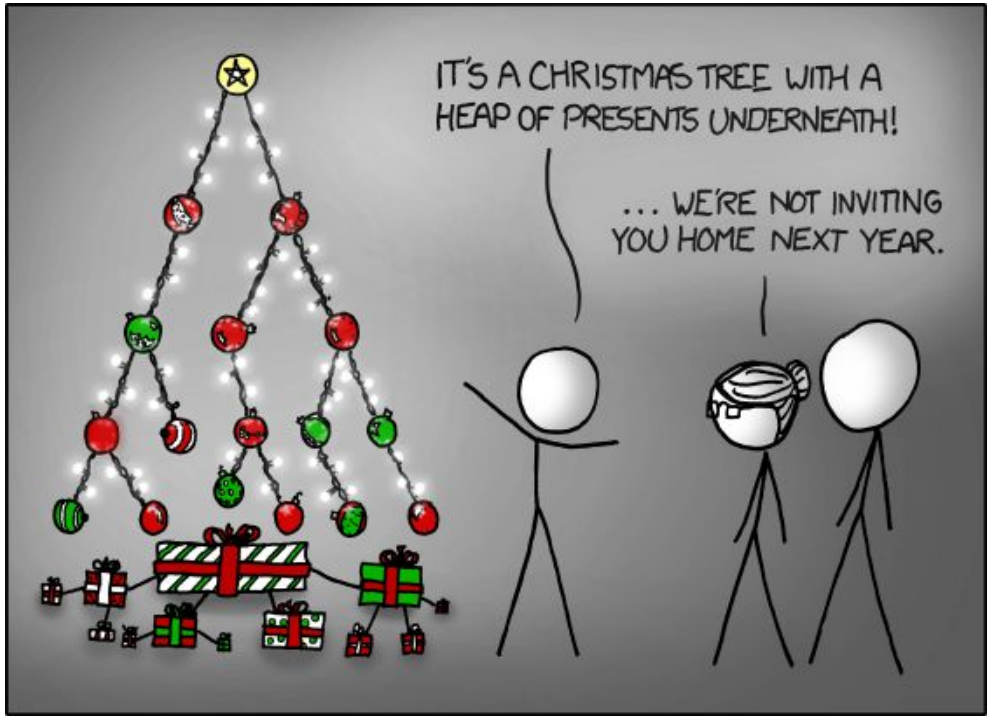
Binary Trees

CIT 5940



Binary Tree



From: sciblogs.co.nz

Introduction

- Benefits
  - Efficient management of large collections
  - Easy implementation
- Applications
  - Priority queues
  - Expression trees
  - Data compression

Definitions

- A binary tree is:
  - A finite set of nodes
  - A root node with two disjoint binary trees called the left and right *subtrees (or children)*
- Each node contains:
  - A value (the data we are storing)
  - A reference to a left child (may be null), and
  - A reference to a right child (may be null)

Definitions

- A binary tree may be empty (contain no nodes)
- Internal node: any node that has at least one non-empty child
- Leaf node: any node that has two empty *children*

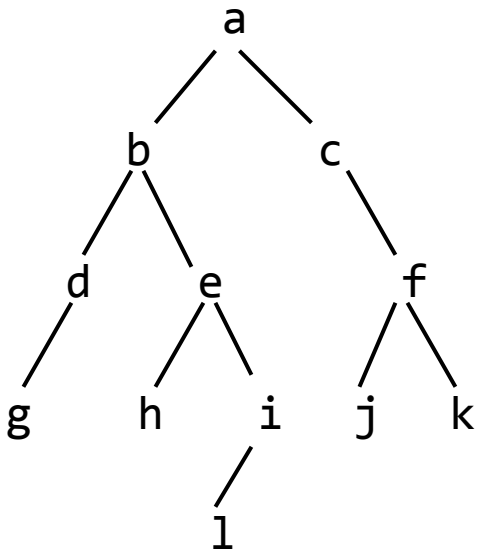
More Definitions

- The *node P* that directly links to a node *A* is the **parent** of *A*. *A* is the **child** of *P*
- A sequence of *nodes* $v_1, v_2, \ldots, v_n$ forms a **path** of length $n-1$ if there exist edges from v_i to v_{i+1} for $1 \leq i < n$
- For a given node *A*, any node on a path from *A* up to the root is an **ancestor** of *A*

More Definitions

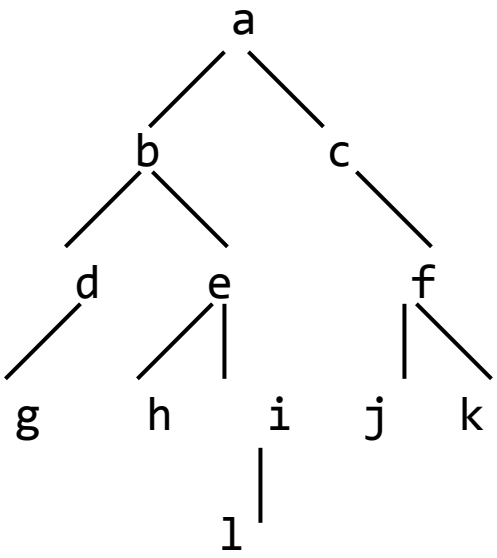
- The **depth** of a node *M* in a tree is the length of the path from the root of the tree to *M*
- a **sibling** of *node A* is any other node with the same *parent* as *A*

Size, Height, Depth and Level



- The **size** of a binary tree is the number of nodes in it
  - This tree has a size of ??
- The **depth** of a node is the length of his path to the root
  - a is at depth zero, e is at depth ???
- The **height** of a tree is the *depth* of the deepest *node*
  - This tree has a height of ???
- All nodes of depth *d* are at **level** *d* in the tree
- The root is at level 0

Size, Height, Depth and Level



- The **size** of a binary tree is the number of nodes in it
  - This tree has a size of 12
- The **depth** of a node is the length of his path to the root
  - a is at depth zero, e is at depth 2
- The **height** of a tree is the *depth* of the deepest *node*
  - This tree has a height of 4
- All nodes of depth *d* are at **level** *d* in the tree
- The root is at level 0

Full BT

- A *binary tree* is **full:**
  - if every node is either a leaf node or else
  - it is an internal node with two non-empty children

Full BT Theorem

- *The number of leaves in a non-empty full binary tree is one more than the number of internal nodes*
-

Proof Sketch of Full BT Theorem

- Base case: Let’s inspect the only non-empty full trees with 0 or 1 internal nodes…

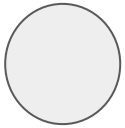
n = 0

n = 1

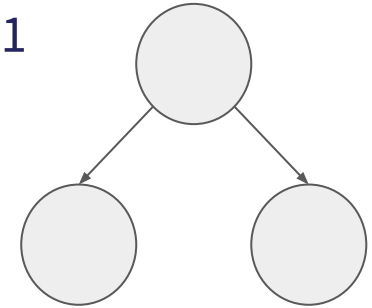
Proof Sketch of Full BT Theorem

- Base case: Let’s inspect the only non-empty full trees with 0 or 1 internal nodes…

n = 0



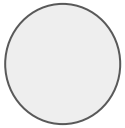
n = 1



Proof Sketch of Full BT Theorem

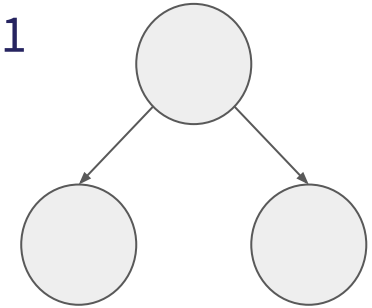
- Base case: Let’s inspect the only non-empty full trees with 0 or 1 internal nodes…

n = 0



one leaf!

n = 1



two leaves!

Proof Sketch of Full BT Theorem

- Induction Hypothesis: Assume that all full BTs with n - 1 internal nodes have n leaves.

Proof Sketch of Full BT Theorem

- Induction Step: Show that a full tree on n internal nodes has n + 1 leaf nodes
- From a maximal depth internal node, remove its children.

Proof Sketch of Full BT Theorem

- Induction Step: Show that a full tree on n internal nodes has n + 1 leaf nodes
- From a maximal depth internal node, remove its children.
- Now:
  - Tree is full, still
  - n - 1 internal nodes

Proof Sketch of Full BT Theorem

- Induction Step: Show that a full tree on n internal nodes has n + 1 leaf nodes
- From a maximal depth internal node, remove its children.
- Now: we have a full tree on n - 1 internal nodes, so we know there are n leaves.

Proof Sketch of Full BT Theorem

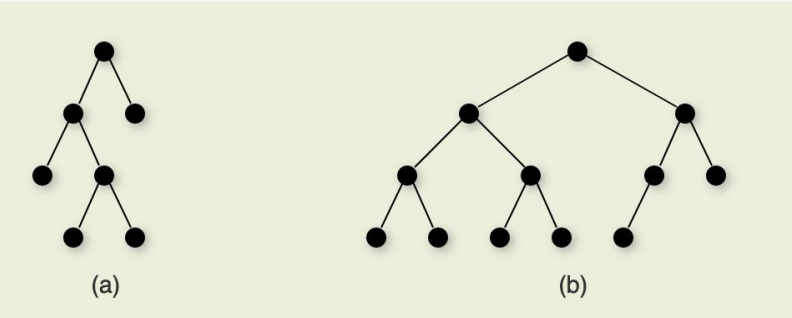
- Induction Step: Show that a full tree on n internal nodes has n + 1 leaf nodes
- From a maximal depth internal node, remove its children.
- Now: we have a full tree on n - 1 internal nodes, so we know there are n leaves.
- Replace the leaf nodes: one leaf is lost but two are gained!
  - The tree is still full
  - The tree now has n internal nodes
  - The tree has n - 1 + 2 → n + 1 leaf nodes! QED

Complete BT

- **Complete binary tree** is:
  - A BT where the nodes are filled row (level) by row, with the bottom row filled in left to right

- Complete binary trees have a restricted shape:
  - There is only one tree of *n* nodes for any value of *n*

Full vs Complete BT



(a) Is a Full BT
(b) Is a Complete BT

- The **heap** data structure is an example of a **complete BT**
- Heaps are used in priority queues implementations
- **Huffman coding tree** is an example of **full BT**
- Huffman coding tree is used to implement file compression algorithms

Balanced BT

- A *tree* where the *subtrees* meet some criteria for being balanced
- Balanced criteria:
  - The tree is *height-balanced*
  - The tree has a roughly equal number of *nodes* in each subtree

- Height balanced: a BT in which the *depths* of each *subtree* in the tree are roughly the same
  - **AVL tree** is an example of height-balanced BT

Binary Tree Node Implementations

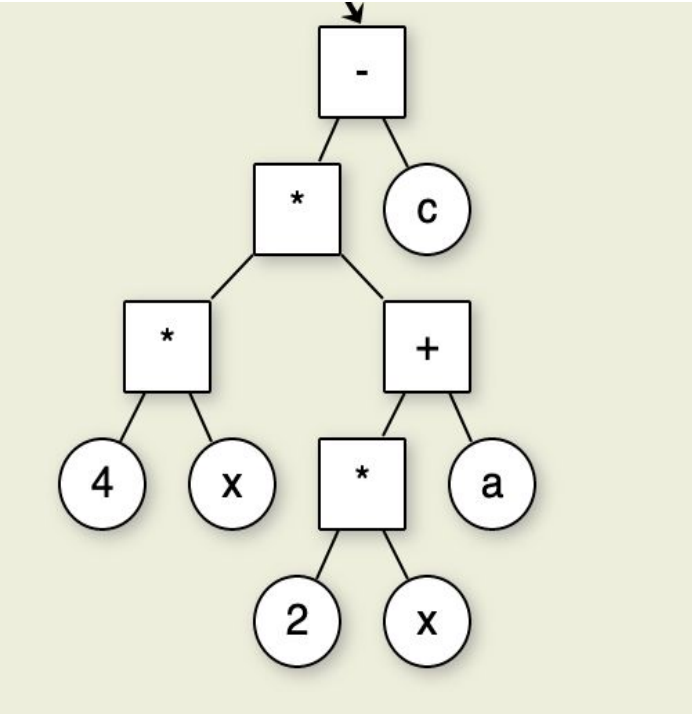
- *Same class definition vs Separate class definitions for leaves and internal nodes?*
  - Only internal nodes have non-empty children
  - Wasteful to store child pointers in leaf nodes
    - Full BT Theorem → 50% nodes are leaves! (Similar holds for complete trees, proof withheld)

- Conclusion: Separate implementations for internal and leaf nodes to reduce overhead. (***Polymorphism***)

Binary Tree Traversals

- **Preorder traversal**: a *traversal* that first *visits* the *root*, then *recursively* visits the left *child*, then recursively visits the right child
- **Postorder traversal**: a *traversal* that first *recursively visits* the left *child*, then recursively visits the right child, and then visits the *root*.
- **Inorder traversal**: a *traversal* that first *recursively visits* the left *child*, then visits the *root*, and then recursively visits the right child.

Expression Tree



- A *tree* structure used to represent a mathematical expression
- *Internal nodes* of the expression tree are operators in the expression, with the subtrees being the sub-expressions that are its operand
- All *leaf nodes* are operands

Activity

- With your neighbor, build (draw) the expression tree from the following prefix (preorder) traversal sequence:

*- - *abc+d*e%gh*

- With your neighbor, build (draw) the expression tree from the following postfix (postorder) traversal sequence:

*ab%cdef-/+**

Coding Example

1. Tree Traversal
2. Expression Tree evaluation using a Stack and a Queue
    - Place nodes in a Queue using postorder traversal
    - Dequeue nodes:
        - If node is a leaf, push onto the Stack
        - Else: (node is operator)
            - pop next two nodes
            - Compute operation and store the result in a new leaf node
            - Push the new leaf node onto the stack
    - Stop when the queue is empty

BT Space Requirements

- **Overhead**: the amount of space necessary to maintain the data structure

1. Every node has two pointers to its children
   - P: space required by a pointer
   - D: space required by a data value
   - Overhead = $\frac{2P}{(2P+D)}$

2. Every node has three pointers (to its children and data record)
   - Overhead = $\frac{3P}{(3P+D)}$

BT Space Requirements

- Full BT:
  - Only internal nodes have child pointers
  - ½ nodes are internal, ½ nodes are leaf
  - Overhead = $\frac{P}{(P+D)}$