CS C341 / IS C361

Data Structures & Algorithms

15-Mar-14

1

Binary Trees

- Traversal(s) and Applications

BINARY TREE – REVIEW

- Definition: A Binary Tree is either
  - an empty Binary Tree OR
  - has a root value and two (sub) Binary Trees.
- Type Definition
  - BinaryTree = EmptyBinaryTree U
    (Element * BinaryTree * BinaryTree)
- Representation (in C)
  - typedef struct _binTree *BinTree;
  - struct _binTree {
    Element val; BinTree left, BinTree right;
  };

Binary Tree – Review [2]

- BinaryTree - Operations
  - BinTree createBinTree()
  - boolean isEmptyBinTree(BinTree)
- Properties:
  - isEmptyBinTree(createBinTree()) == TRUE

Binary Tree – Review [2]

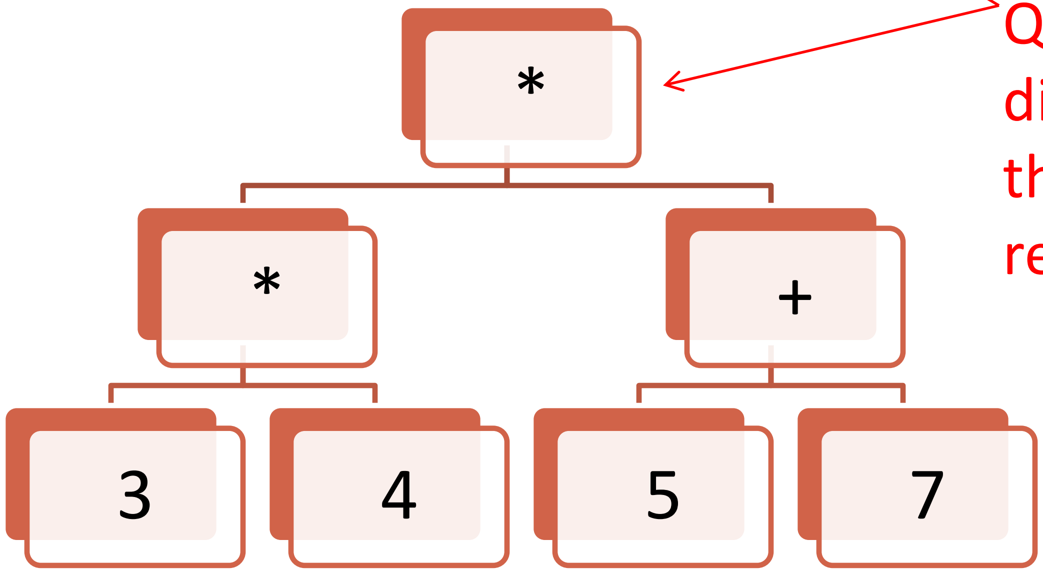
- BinaryTree - Operations
  - BinTree left(BinTree)
  - BinTree right(BinTree)
  - Element rootVal(BinTree)
  - BinTree makeBinTree(Element, BinTree, BinaryTree)
- Properties:
  - makeBinTree(rootVal(bt), left(bt), right(bt)) == bt

Binary Tree - Traversals

- Typical Requirements for a traversal:
  - Enumerating the elements in a collection (represented as a binary tree)
  - Applying some function / procedure on each element in a collection (represented as a binary tree)
- Order of traversal
  - In-Order Traversal:
      - Traverse left, visit Root, Traverse right
    - Application:
      - Enumeration in sorted order in a BST
        - Left – Right vs. Right – Left ??

BINARY TREE - TRAVERSALS

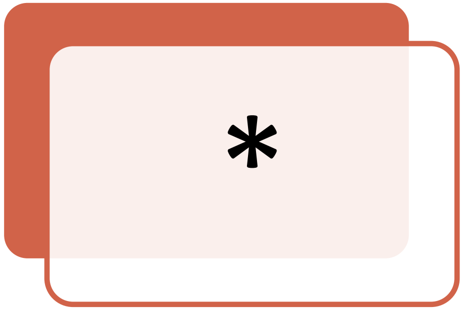
- Consider an expression of the form:
  - (* (* 3 4) (+5 7))
    - Referred to as a "prefix" expression.
- Convert this into an internal representation:



Q: What is the difference between these two forms of representation?

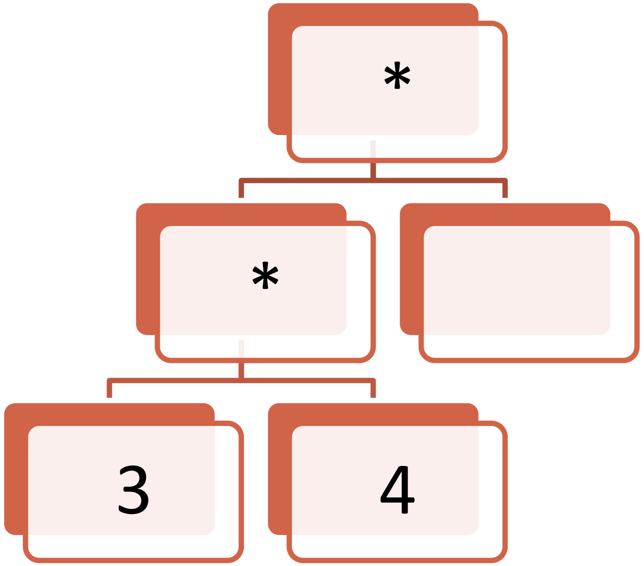
Binary Tree - Traversals

- How do you construct such a representation?
  - Construct the root Node
  - (* (* 3 4) (+5 7))



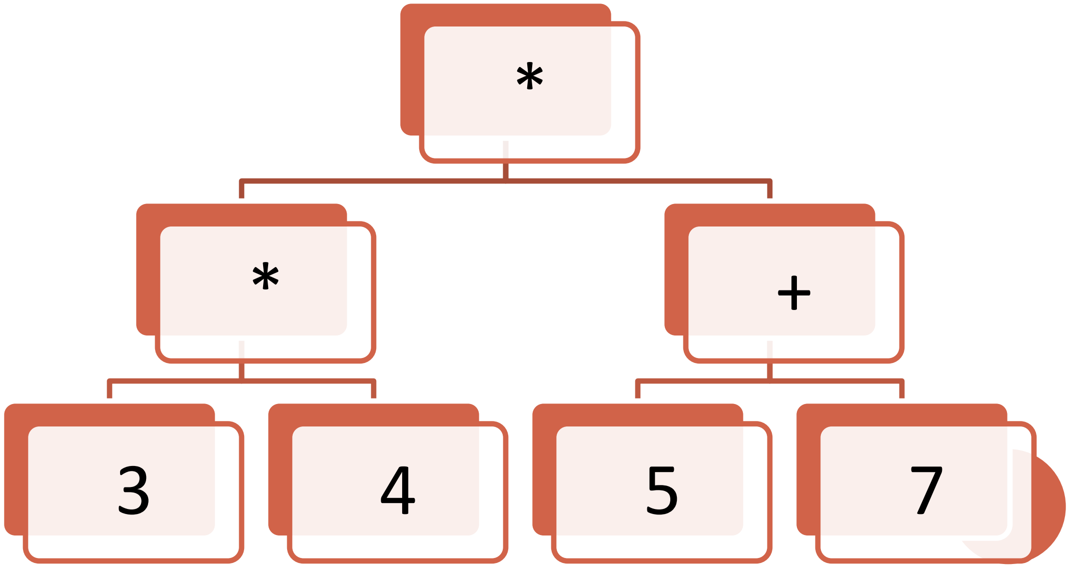
Binary Tree - Traversals

- How do you construct such a representation?
  - Construct the root Node
  - Construct the left sub-tree (i.e. left sub-expression)
  - (* (* 3 4) (+5 7))



BINARY TREE - TRAVERSALS

- How do you construct such a representation?
  - Construct the root Node
  - Construct the left sub-tree
  - Construct the right sub-tree (i.e. right sub-expression)
  - (* (* 3 4) <u>(+5 7))</u>

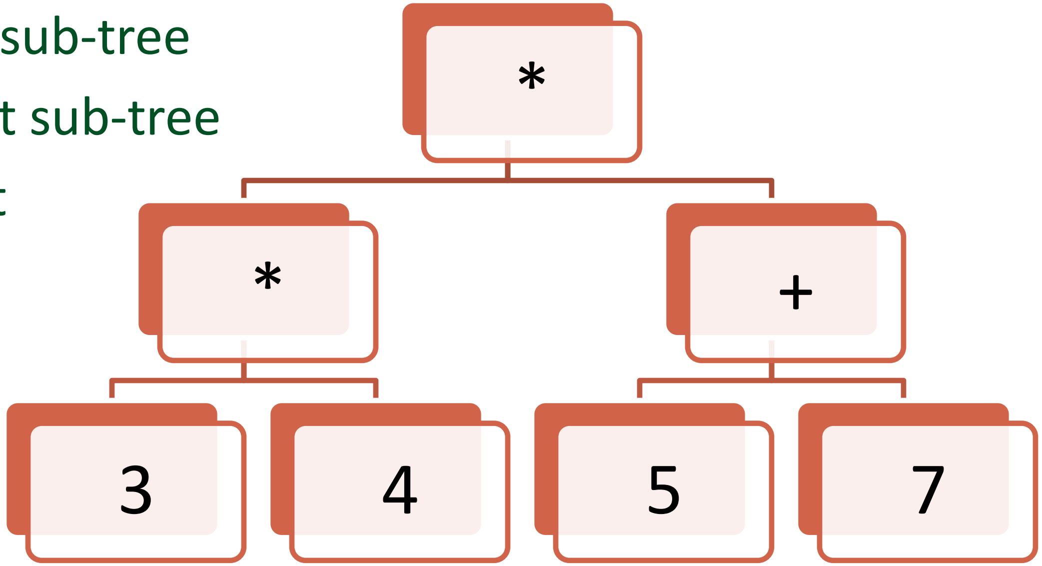


Binary Tree - Traversals

- Pre-Order Traversal:
  - visit Root, Traverse left, Traverse right
- Question:
  - Does left-to-right order matter?
    - e.g. Construction of a binary search tree
- Special case:
  - *find* operation in a BST

Binary Tree - Traversals

- How do you evaluate an expression - given a tree representation?
  - Evaluate the left sub-tree
  - Evaluate the right sub-tree
  - Evaluate the root



- Post-Order Traversal:
  - Traverse left, Traverse right, visit Root

Binary Tree – Application - Encoding

- Encoding Problem:
  - Consider a scenario where strings of symbols are to be encoded:
    - e.g. Machine instructions (*opcodes, addresses*)
    - e.g. Binary representation of HTML/XML documents

```
<BOOKS>
<BOOK YEAR="1999">
<AUTHOR>Abiteboul</AUTHOR>
<AUTHOR>Buneman</AUTHOR>
<TITLE>Data on the Web</TITLE>
<PRICE>40.00</PRICE>
<SHIPPING>10.00</SHIPPING>
</BOOK>
<BOOK YEAR="2002">
…
</BOOK>
</BOOKS>
```

Binary Tree – Application - Encoding

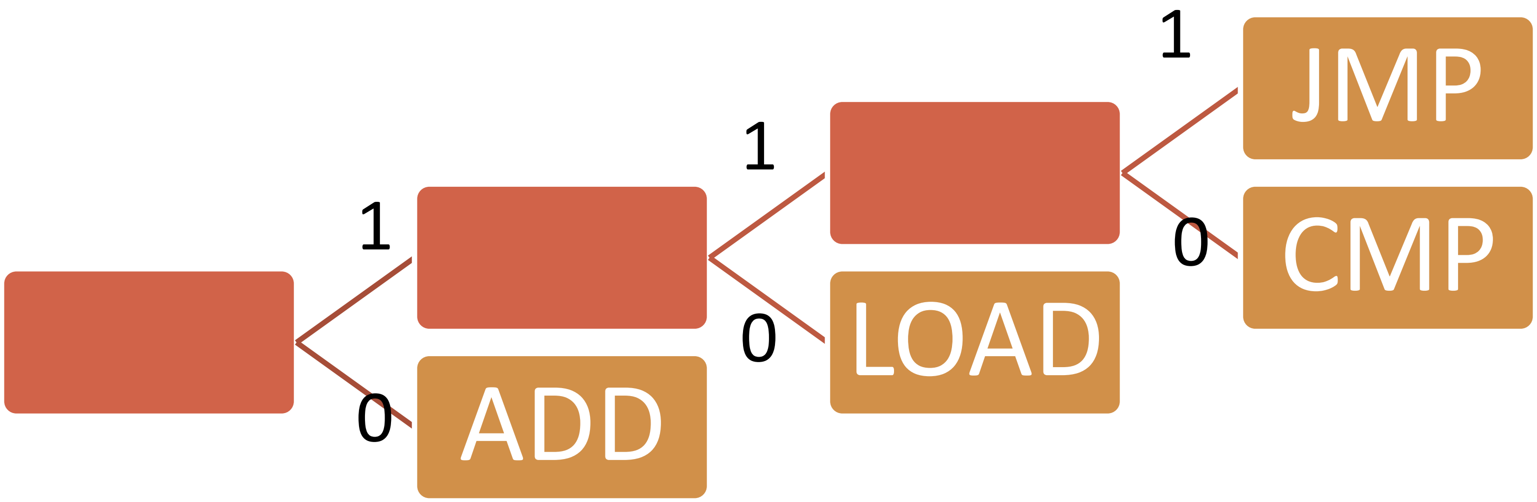
- Encoding Technique
  - If you have N different "symbols" to be encoded,
  - then ┌ logN ┐ bits are required to encode each occurrence of each item
    - *fixed length binary coding*
- Given a string of length M where each item may be any of the N "symbols"
  - Size of the representation is M* ┌ logN ┐
  - Decoding each item (from the encoded form) requires inspecting all the ┌ logN ┐ bits.
- Is it possible to reduce the number of bits required or the work required to decode?

Binary Tree – Application - Encoding

- Encoding Technique
  - Consider the frequency of occurrence of those symbols:
    - e.g. AUTHOR may occur more often than other symbols in the particular XML database
    - e.g. ADD is the most common instruction in most programs.
  - Encode the most common symbol as the shortest code (1 bit):
    - Say, ADD is encoded as 0
    - Then1 would represent “Any symbol other than ADD”
    - Encode the next most frequent symbol as 10
    - …
  - Variable length coding
    - Specifically known as Prefix codes
    - Size of representation = $\sum$ freq(c) * encLen(c)

BINARY TREE – APPLICATION - ENCODING

- How does decoding work?
  - Say, ADD is encoded as 0, LOAD is encoded as 10, CMP is encoded as 110, and JMP is encoded as 111



  - Each code is a path from the root to a leaf in the tree

Binary Tree – Application - Encoding

- Huffman Coding Technique:
  - Produces optimal prefix code given frequencies of items (to be coded)
- Preconditions : C is an array of symbols;
- for each c in C, c.freq is the frequency of the symbol
- Output: Decoding tree for C

```
HuffmanCode(C) {
    H = buildHeap(C);   // H is C after heapification!
    for j = 1 to |C|-1 {
        x = find(H); H = delete(H);
        y = find(H); H = delete(H);
        H = insert(makeBinTree(x. freq + y.freq, x, y), H);
    }
    return find(H)
}
```

Binary Tree – Application - Encoding

- Huffman's encoding algorithm produces optimal prefix code:
  - Proof omitted.
- Huffman's encoding algorithm uses a "greedy" technique:
  - It makes "a local (i.e. greedy) choice" that results in " a globally optimal" solution.
    - Choice of two lowest frequency items to have the longest code(s).
- Greedy Technique is a design technique to produce efficient algorithms.
  - [Will see more of it later!]

CS C341 / IS C361
Data Structures & Algorithms

(General) Trees

- Traversal(s)
  - Depth First Traversal
  - Breadth First Traversal
- Applications

Trees - Representation

- A *tree* is either
  - an *empty tree* OR
  - a *root* element with one or more *children (i.e. subtrees)*
- N-ary trees
  - Maximum number of children for any node is N
- Arbitrary Branching
  - Number of children is finite but not bounded

Trees - Examples

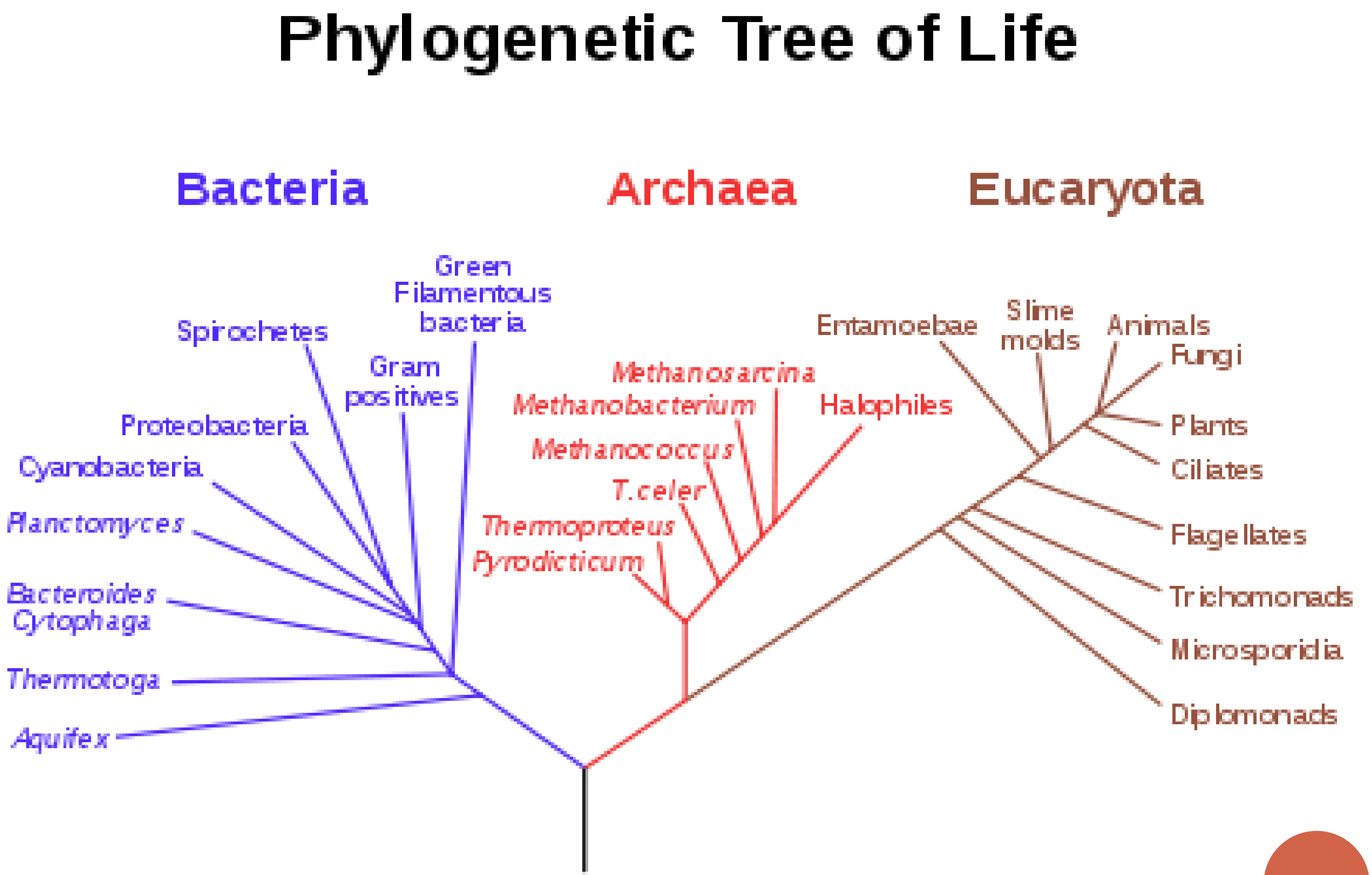
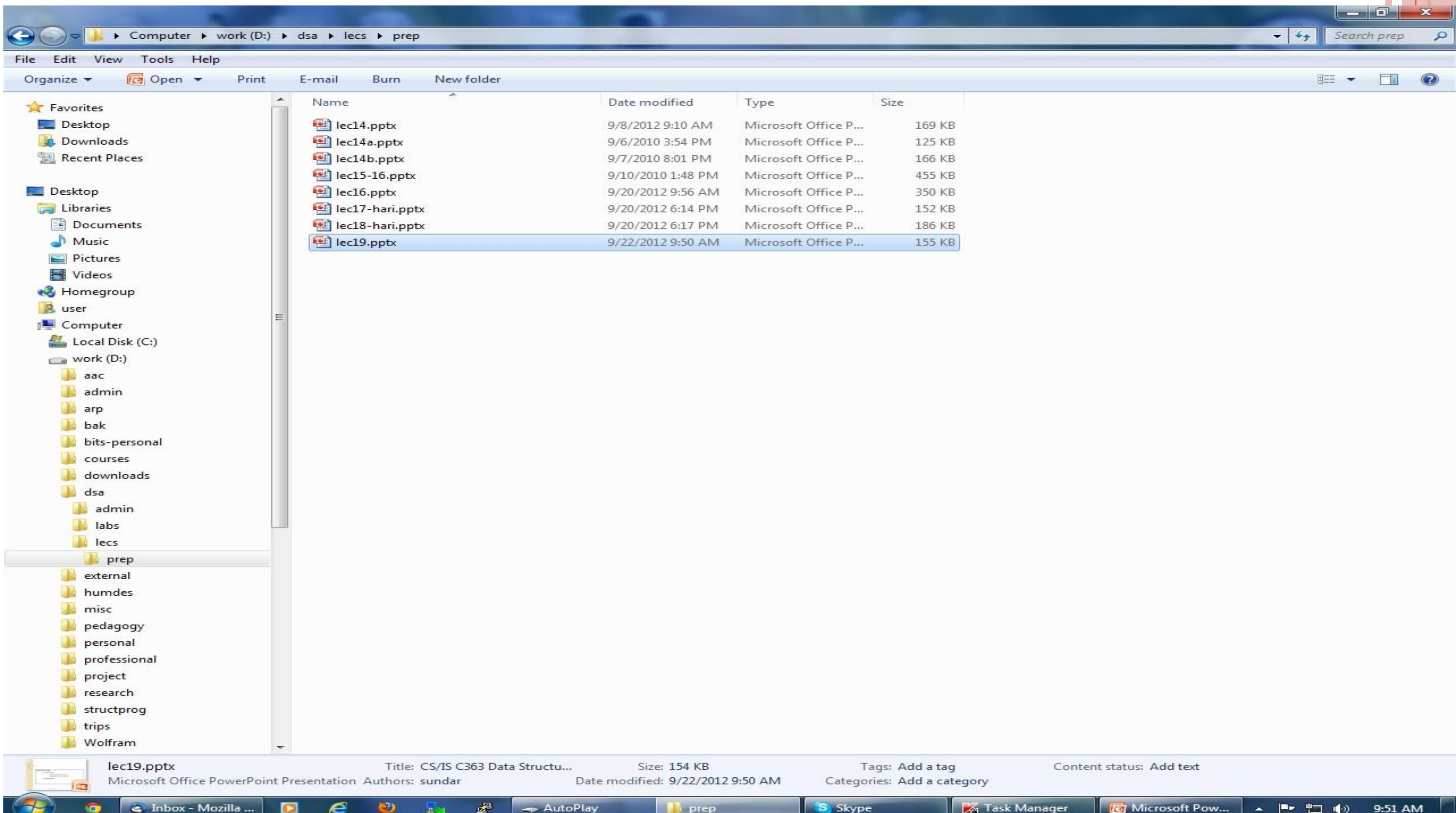


Figure from Wikipedia

TREES - EXAMPLES

- Examples
  - File System
- A file system can be represented as a tree:
  - Each directory (or folder in Windows) is a non-terminal node
  - Each non-directory file is a terminal node

TREES - EXAMPLES



TREES – OPERATIONS AND REPRESENTATION

- Operations
  - TREE createTree(int maxChildren)
    - // Use an invalid argument for arbitrary branching
  - TREE isEmptyTree(TREE)
  - Element rootVal(TREE)
  - Iterator getChildren(TREE)
    - // Define an iterator to access children.
  - TREE makeTree(Element rootVal, TREE *children)
- Representation
  - Each node is represented as a record: <*Value, Children*>
    - List of *Children* is usually an array or a linked list.
      - When do you use a linked list?

TREES - REPRESENTATION

- N-ary trees

```
typedef struct  _node  *TREE;
struct  _node {
    Element val;
    TREE  children[N];        // N is a constant
    int numCh;                // actual number of children
};
```

Alternative: Allocate dynamically!

- Arbitrary Branching

```
typedef struct  _node  *TREE;
struct  _node {
   Element val;
   TREE  *children;
};   // children is head of a linked list
```

TREES – TRAVERSALS - DFT

- Depth First Traversal
  - Traverse one path (from root to leaf) completely before starting on another path
- Algorithm:

```
dfsTree(Tree t)
{
    if t is empty return;
    visit root;            // do what you have to!
    for each c in children of t { dfsTree(c); }
}
```

TREES – TRAVERSALS – DFT [2]

- Depth First Traversal – Recursive Implementation

```
dfsTree(Tree t)
{
        if (isEmptyTree(t)) return;
        visit(rootVal(t));              // do what you have to!
        Iterator Ch = getChildren(t);
         while (hasMoreElements(Ch)) {
               dfsTree(getNextElement(Ch));
         }
}
```

- Recursive call is inside a loop – how to eliminate this?

Trees – Traversals – DFT [3]

- Depth First Traversal – (Naïve) Iterative Implementation

```
dfsTree(Tree t)
{    Stack st = createStack();
     BEGIN:  if (isEmptyTree(t)) return;
     visit(rootVal(t));            // do what you have to!
     Iterator Ch = children(t);
     while (hasMoreElements(Ch)) {
             st = push(getNextElement(Ch), st);
      }
      if (!isEmptyStack(st)) {
           t = top(st); st = pop(st);   goto BEGIN;
      }
}
```

Can we avoid pushing all children on stack?

Trees – Traversals – DFT [4]

- Depth First Traversal – Iterative Implementation

```
dfsTree(Tree t)
{   if (isEmptyTree(t))  return;
    Stack st = createStack();

    st = push(getChildren(t),  st);
     while (!isEmptyStack(st)) {
          tCh = top(st);
            if (hasMoreElements(tCh)) {
              t = getNextElement(tCh);

              st = push(getChildren(t), st);
          } else {  st = pop(st);
          }
      }
}
```

Where should the visits occur?

Store the iterator for a node's children instead of storing all children.

The Iterator is a pointer to a node in a linked list or an (integer) index and starting address of an array.

Trees – Traversals – DFT [5]

- Depth First Traversal – Iterative Implementation

```
dfsTree(Tree t)
{   if (isEmptyTree(t))  return;
    Stack st = createStack();
    visit(rootVal(t));
    st = push(getChildren(t),  st);
     while (!isEmptyStack(st)) {
           tCh = top(st);
             if (hasMoreElements(tCh)) {
                t = getNextElement(tCh);
                visit(rootVal(t));
                st = push(getChildren(t), st);
           } else {  st = pop(st);
           }
       }
}
```

Visit a node and then push its iterator on stack.

When all children of a node are visited pop the corresponding iterator!

Tree Traversals – BFT

- Breadth First Traversal (a.k.a. Level Order Traversal)
  - Traverse one level (of depth) completely before starting on a lower level
- Algorithm:

```
bfsTree(Tree t) {
  if (!isEmpty(t))  {  visit(rootVal(t));   bfsLevel("children of t"); }
}

bfsLevel(Set remSet)  {
    copy remSet into cSet
    for each c in cSet {
        visit(c);
        remSet = remSet  - { c }  U  getChildren(c);
     }
    bfsLevel(newRemSet);
}
```

Representation for Set?

No ordered queries

First-in-First out – Why?

A FIFO Queue is a natural choice

This is a tail call

TREE TRAVERSALS – BFT [2]

```
bfsTree(Tree t) {
  if (!isEmpty(t)) {
    Queue q = createQ();  q = addQ(t, q); bfsLevel(q);   }
}
 bfsLevel(Queue q) {
  while (!isEmpty(q)) {
      t = getQ(q);  q = deleteQ(q);
      visit(t);
      Iterator ch = getChildren(t);
     while (hasMoreElements(ch))
                 q = addQ(getNextElement(ch), q);
   }
}
```

Faster addition?
Queue of Iterators:
  may increase
  some work for
  getQ

TREES - TRAVERSALS

- Time Complexity of DFT and BFT
  - O(n)
- Space Complexity
  - DFT: Size of stack
    - Height of the tree
  - BFT: Size of queue
    - Maximum # nodes in two consecutive levels
      - Exercise: Make it more precise!
- Under what conditions does the size of the queue (in BFT) get larger than the size of the stack (in DFT)?
  - Give a comparative characterization so that one can choose DFT or BFT
    - for a particular application and/or a given (class of) tree(s)!

Trees – Applications – File Systems

- Implement the following using traversal:
  - (Unix command) find
  - (Unix Command) cp – R
    - Look up the man pages to understand the commands.
    - Look up man pages (for *dirent / readdir*)
    - Choice of traversal
      - DFT or BFT?? Why??
  - (Windows Explorer) Navigation
    - Expand "on click"
  - Game Playing (e.g. Chess)
    - Find the next steps (given current board position)
    - Best First Traversal : Find the best next steps and expand them further
- Exercise:
  - Identify other day-to-day (computational) examples!