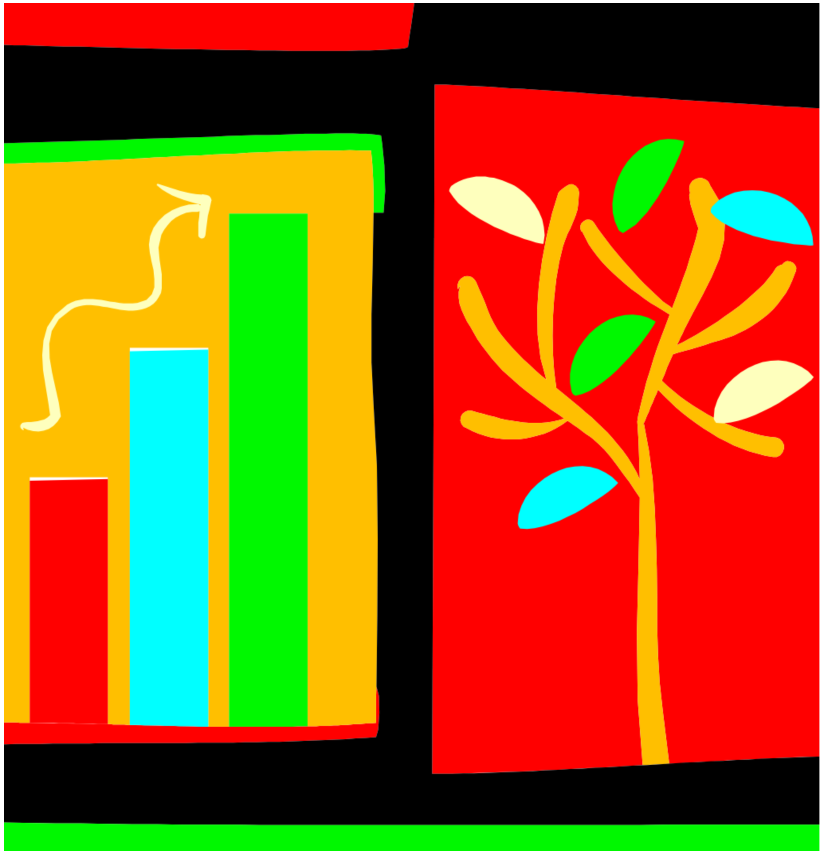
CS/ENGRD 2110
Object-Oriented Programming and Data Structures



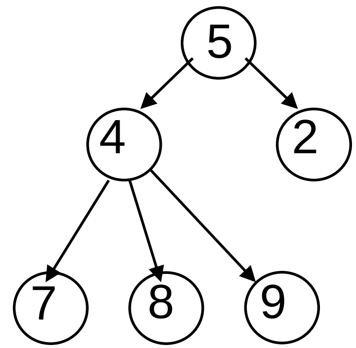
Spring 2012

Thorsten Joachims

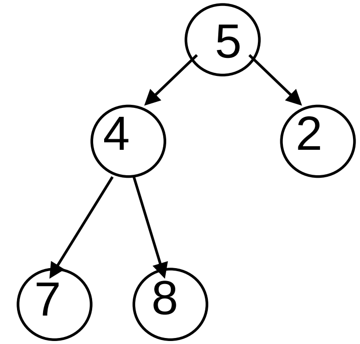
Lecture 9: Trees

Tree Overview

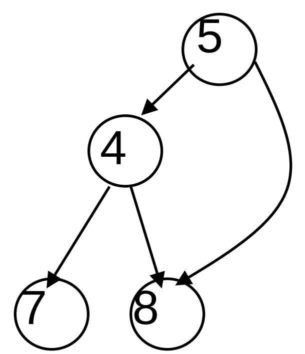
- *Tree*: recursive data structure (similar to list)
  - Each cell may have zero or more *successors* (children)
  - Each cell has exactly one *predecessor* (parent) except the *root*, which has none
  - Cells without children are called *leaves*
  - All cells are reachable from *root*
- *Binary tree*: tree in which each cell can have at most two children: a left child and a right child



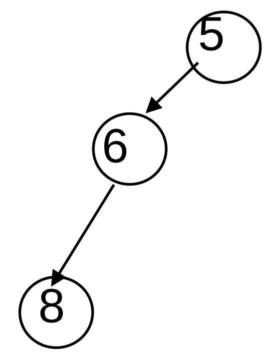
General tree



Binary tree



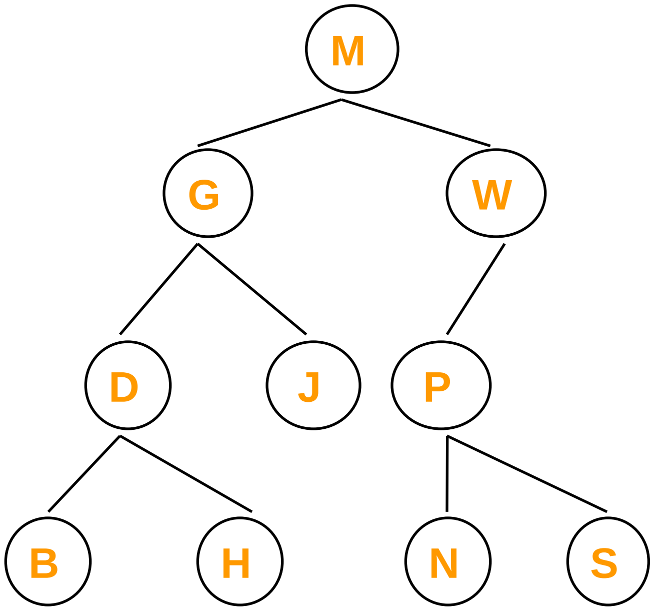
Not a tree



List-like tree

Tree Terminology

- M is the *root* of this tree
- G is the *root* of the *left subtree* of M
- B, H, J, N, and S are *leaves*
- N is the *left child* of P; S is the *right child*
- P is the *parent* of N
- G and W are *siblings*
- M and G are *ancestors* of D
- P, N, and S are *descendants* of W
- Node J is at *depth* 2 (i.e., *depth* = length of path from root = number of edges)
- Node W is at *height* 2 (i.e., *height* = length of longest path to a leaf)
- A collection of several trees is called a ...?



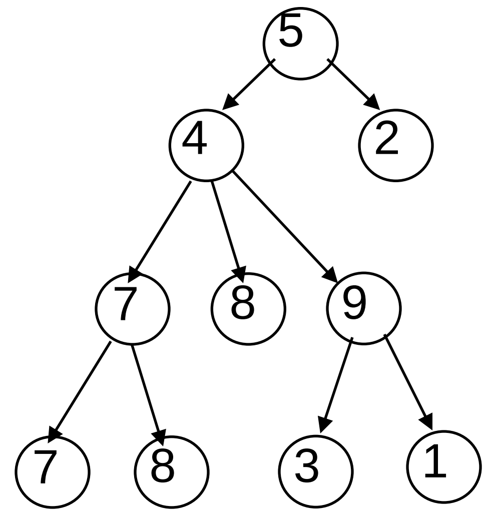
Class for Binary Tree Cells

```
class TreeCell<T> {
   private T datum;
   private TreeCell<T> left, right;

   public TreeCell(T x) {
      datum = x; left = null; right = null;
   }

   public TreeCell(T x, TreeCell<T> lft,
                        TreeCell<T> rgt) {
      datum = x;
      left = lft;
      right = rgt;
   }
   more methods:    getDatum, setDatum, getLeft,
                    setLeft, getRight, setRight
}
```

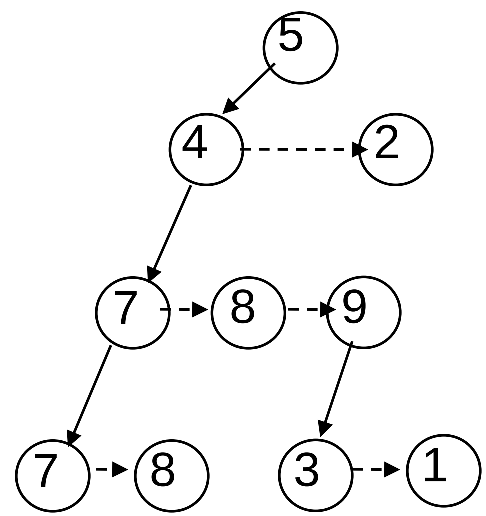
```
... new TreeCell<String>("hello") ...
```

Class for General Trees

```
class GTreeCell {
   private Object datum;
   private GTreeCell left;
   private GTreeCell sibling;

   appropriate getter and
   setter methods
}
```



General tree



Tree represented using GTreeCell

- Parent node points directly only to its leftmost child
- Leftmost child has pointer to next sibling, which points to next sibling, etc.

Applications of Trees

- Most languages (natural and computer) have a recursive, hierarchical structure
- This structure is *implicit* in ordinary textual representation
- Recursive structure can be made *explicit* by representing sentences in the language as trees: Abstract Syntax Trees (ASTs)
- ASTs are easier to optimize, generate code from, etc. than textual representation
- A parser converts textual representations to AST

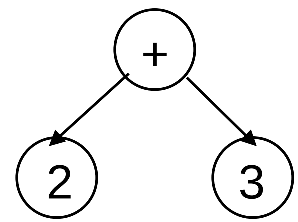
Example

- Expression grammar:
  - E → integer
  - E → (E + E)
- In textual representation
  - Parentheses show hierarchical structure
- In tree representation
  - Hierarchy is explicit in the structure of the tree

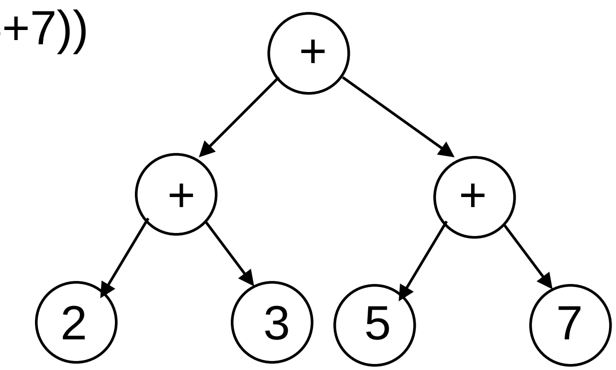
Text | AST Representation

-34

(2 + 3)



((2+3) + (5+7))



Recursion on Trees

- Recursive methods can be written to operate on trees in an obvious way

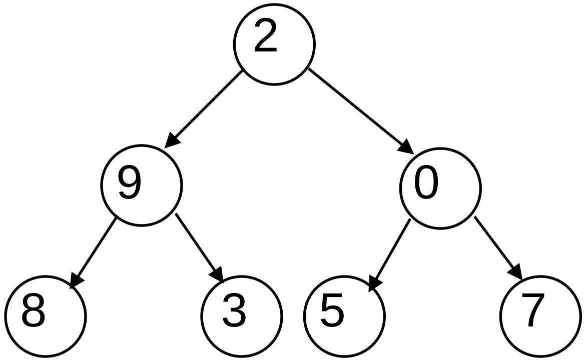
- Base case
  - empty tree
  - leaf node

- Recursive case
  - solve problem on left and right subtrees
  - put solutions together to get solution for full tree

Searching in a Binary Tree

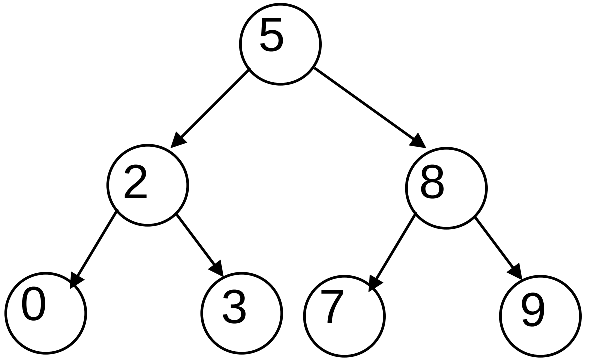
```
public static boolean treeSearch(Object x,
                                  TreeCell node) {
   if (node == null) return false;
   if (node.datum.equals(x)) return true;
   return treeSearch(x, node.left) ||
          treeSearch(x, node.right);
}
```

- Analog of linear search in lists: given tree and an object, find out if object is stored in tree
- Easy to write recursively, harder to write iteratively



Binary Search Tree (BST)

- If the tree data are *ordered* – in any subtree,
  - All *left* descendents of node come *before* node
  - All *right* descendents of node come *after* node
- This makes it *much* faster to search



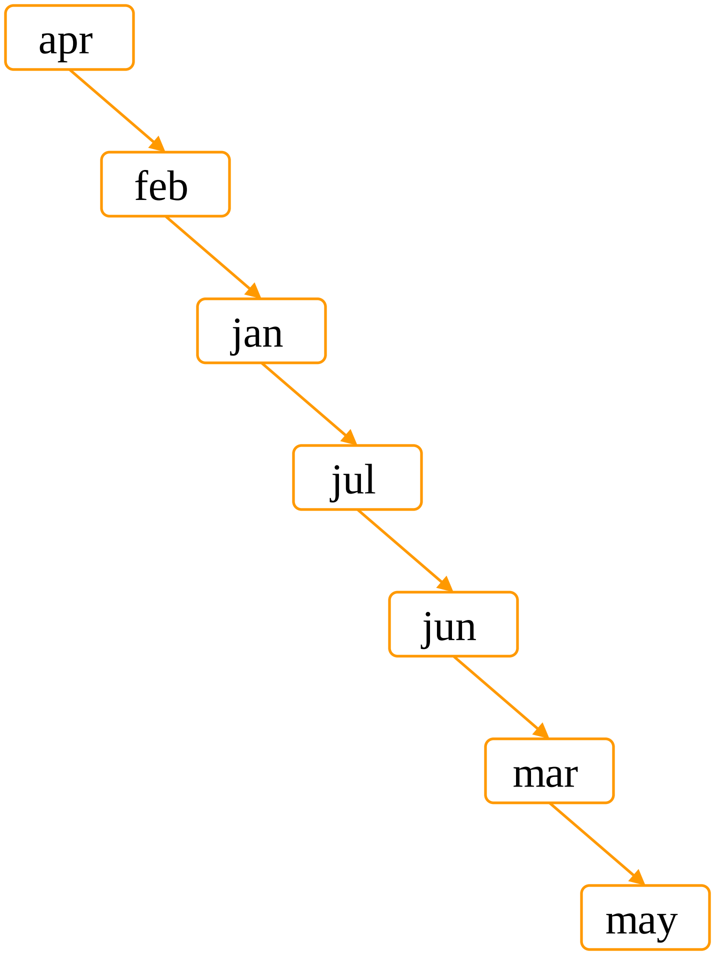

```
public static boolean treeSearch (Object x, TreeCell node) {
   if (node == null) return false;
   if (node.datum.equals(x)) return true;
   if (node.datum.compareTo(x) > 0)
        return treeSearch(x, node.left);
   else
        return treeSearch(x, node.right);
}
```

Building a BST

- To insert a new item
  - Pretend to look for the item
  - Put the new node in the place where you fall off the tree
- This can be done using either recursion or iteration
- Example
  - Tree uses alphabetical order
  - Months appear for insertion in calendar order (i.e. jan, feb, mar, apr, may, jun, jul, …)

What Can Go Wrong?

- A BST makes searches very fast, unless…
  - Nodes are inserted in alphabetical order
  - In this case, we’re basically building a linked list (with some extra wasted space for the left fields that aren’t being used)
  - Maximally high tree → search just as slow as for linked list.
- BST works great if data arrives in random order



Printing Contents of BST

- Because of the ordering rules for a BST, it's easy to print the items in alphabetical order
  - Recursively print everything in the left subtree
  - Print the node
  - Recursively print everything in the right subtree

```
/**
* Show the contents of the BST in
* alphabetical order.
*/
public void show () {
    show(root);
    System.out.println();
}

private static void show(TreeNode node) {
    if (node == null) return;
    show(node.lchild);
    System.out.print(node.datum + " ");
    show(node.rchild);
}
```

Output: `apr feb jan jul jun mar may`

Tree Traversals

- “Walking” over the whole tree is a tree traversal
  - This is done often enough that there are standard names
  - The previous example is an inorder traversal
    - Process left subtree
    - Process node
    - Process right subtree
- Note: we’re using this for printing, but any kind of processing can be done

- There are other standard kinds of traversals
- Preorder traversal
  - Process node
  - Process left subtree
  - Process right subtree
- Postorder traversal
  - Process left subtree
  - Process right subtree
  - Process node

Reading and Writing Trees

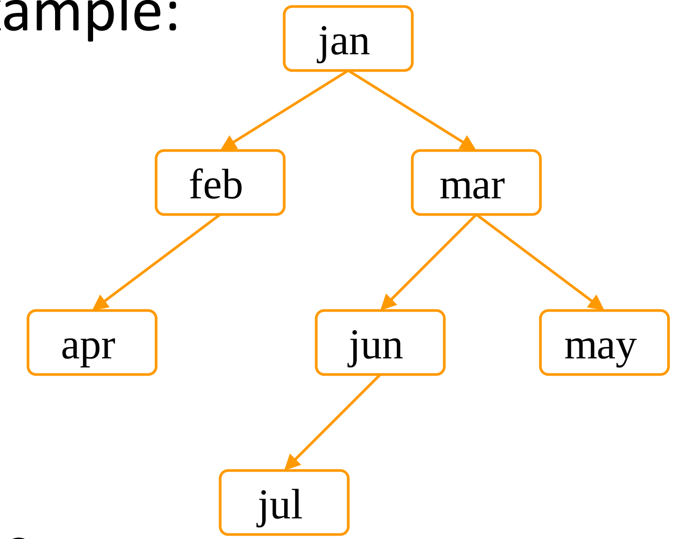
- Write t to file in pre-order:

```
IF t==null THEN
    print null
ELSE
    Print root
    Recurse left subtree
    Recurse right subtree
```

- Read from file in pre-order:

```
next_token = read
IF next_token == null THEN
    return null
ELSE
    root = next_token
    left = Recurse left subtree
    right = Recurse right subtree
    return new TreeCell(root,left,right)
```

- Example:



- File:

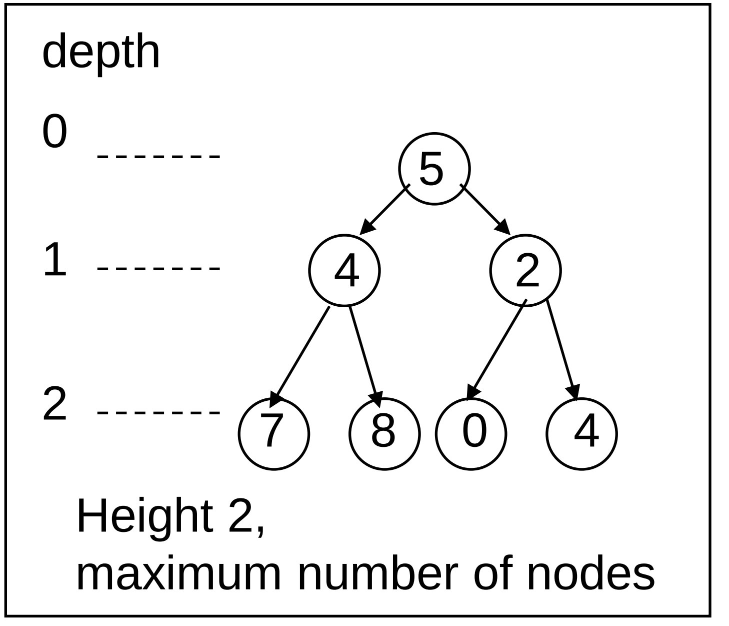
```
jan feb apr null null null
mar jun jul null null null
may null null
```

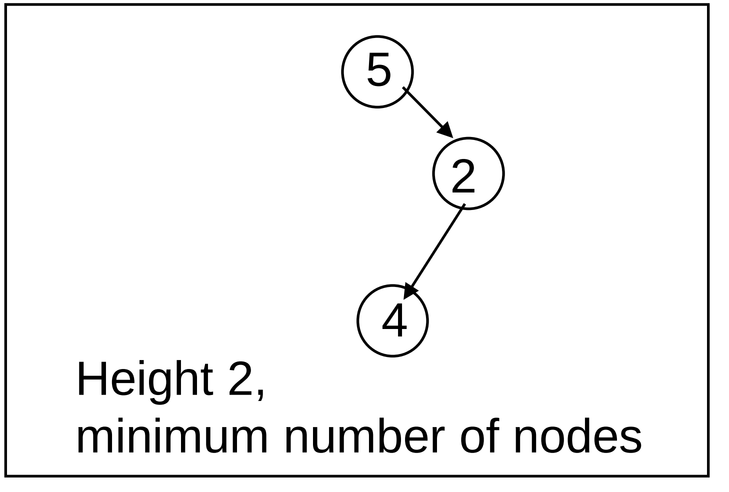
Some Useful Methods

```
//determine if a node is a leaf
public static boolean isLeaf(TreeCell node) {
   return (node != null) && (node.left == null)
                         && (node.right == null);
}

//compute height of tree using postorder traversal
public static int height(TreeCell node) {
   if (node == null) return -1; //empty tree
   if (isLeaf(node)) return 0;
   return 1 + Math.max(height(node.left),
                       height(node.right));
}

//compute number of nodes using postorder traversal
public static int nNodes(TreeCell node) {
   if (node == null) return 0;
   return 1 + nNodes(node.left) + nNodes(node.right);
}
```

Useful Facts about Binary Trees

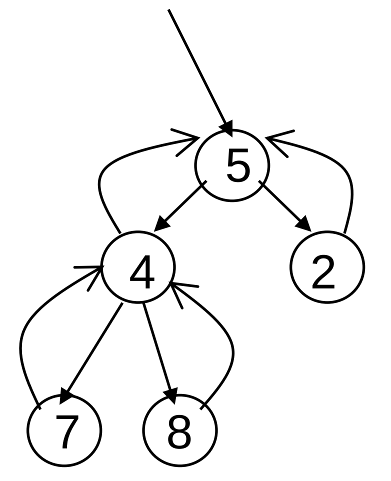
- 2^d = maximum number of nodes at depth d
- If height of tree is h
  - Minimum number of nodes in tree = h + 1
  - Maximum number of nodes in tree = $2^0 + 2^1 + \ldots + 2^h = 2^{h+1} - 1$
- Complete binary tree
  - All levels of tree down to a certain depth are completely filled





Tree with Parent Pointers

- In some applications, it is useful to have trees in which nodes can reference their parents

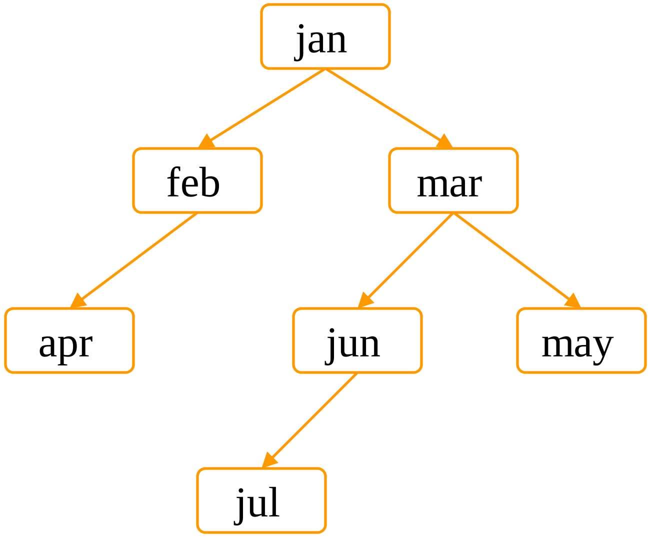


- Analog of doubly-linked lists

Things to Think About

- What if we want to delete data from a BST?

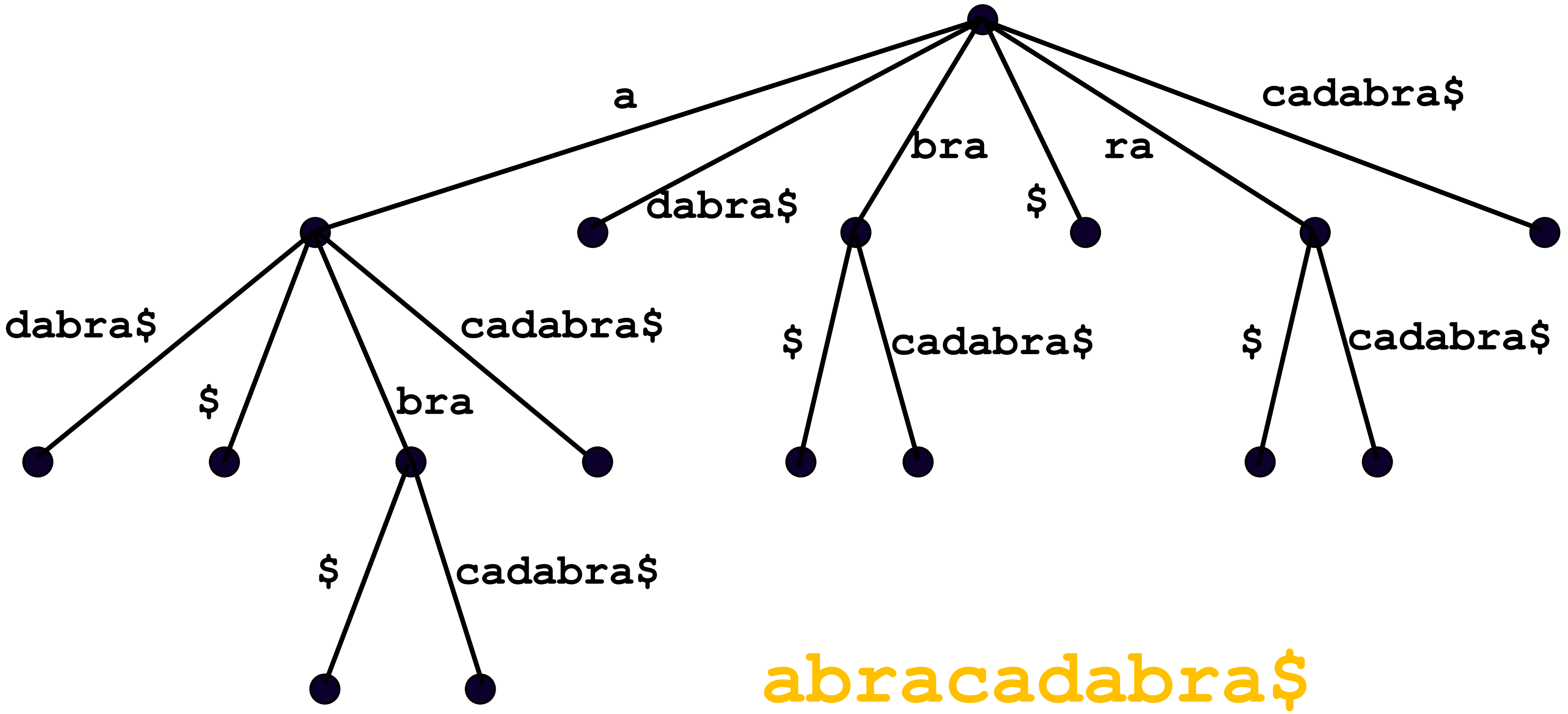
- A BST works great as long as it's *balanced*
  - How can we keep it balanced?



Suffix Trees

- Given a string `s`, a suffix tree for `s` is a tree such that
  - each edge has a unique label, which is a non-null substring of `s`
  - any two edges out of the same node have labels beginning with different characters
  - the labels along any path from the root to a leaf concatenate together to give a suffix of `s`
  - all suffixes are represented by some path
  - the leaf of the path is labeled with the index of the first character of the suffix in `s`
- Suffix trees can be constructed in linear time

Suffix Trees

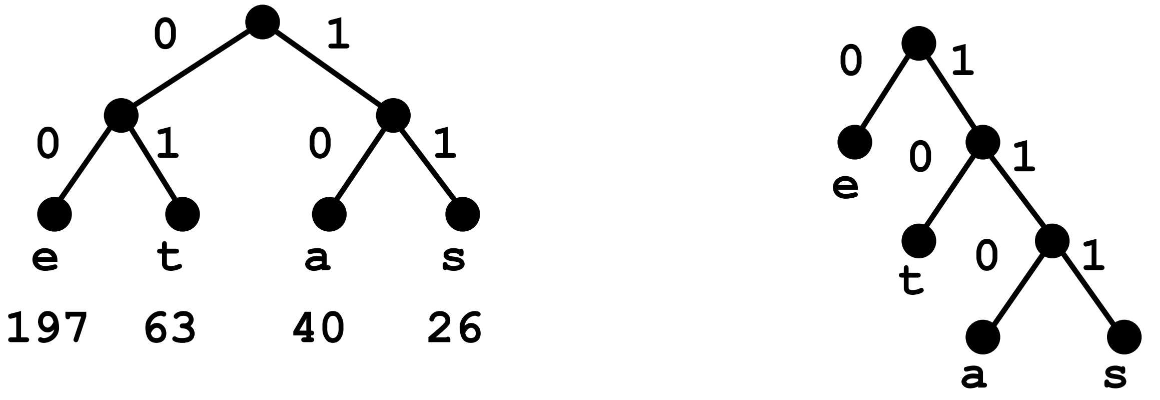


abracadabra$

Suffix Trees

- Useful in string matching algorithms (e.g., longest common substring of 2 strings)
- Most algorithms linear time
- Used in genomics (human genome is ~4GB)

Huffman Trees



Fixed length encoding

197*2 + 63*2 + 40*2 + 26*2 = 652 bits

Huffman encoding

197*1 + 63*2 + 40*3 + 26*3 = 521 bits

Huffman Compression of “Ulysses”

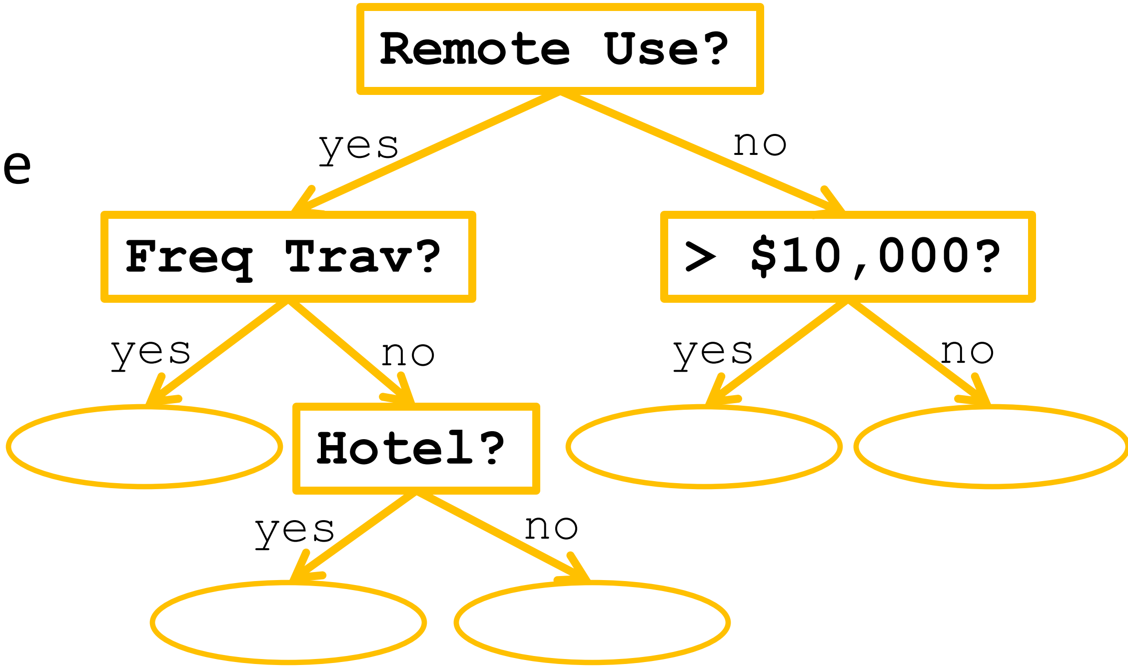
Char	#occ	ascii	bits and	Huffman code
' '	242125	00100000	3	110
'e'	139496	01100101	3	000
't'	95660	01110100	4	1010
'a'	89651	01100001	4	1000
'o'	88884	01101111	4	0111
'n'	78465	01101110	4	0101
'i'	76505	01101001	4	0100
's'	73186	01110011	4	0011
'h'	68625	01101000	5	11111
'r'	68320	01110010	5	11110
'l'	52657	01101100	5	10111
'u'	32942	01110101	6	111011
'g'	26201	01100111	6	101101
'f'	25248	01100110	6	101100
'.'	21361	00101110	6	011010
'p'	20661	01110000	6	011001
...				
'7'	68	00110111	15	111010101001111
'/'	58	00101111	15	111010101001110
'X'	19	01011000	16	0110000000100011
'&'	3	00100110	18	011000000010001010
'%'	3	00100101	19	0110000000100010111
'+'	2	00101011	19	0110000000100010110

```
original size   11904320
compressed size  6822151
42.7% compression
```

Decision Trees

- Classification:
  - Attributes (e.g. is CC used more than 200 miles from home?)
  - Values (e.g. yes/no)
  - Follow branch of tree based on value of attribute.
  - Leaves provide decision.

- Example:
  - Should credit card transaction be denied?



BSP Trees

- BSP = Binary Space Partition
  - Used to render 3D images composed of polygons (see demo)
  - Each node n has one polygon p as data
  - Left subtree of n contains all polygons on one side of p
  - Right subtree of n contains all polygons on the other side of p
- Paint image from back to front. Order of traversal determines occlusion!

Tree Summary

- A *tree* is a recursive data structure
  - Each cell has 0 or more successors (*children*)
  - Each cell except the *root* has at exactly one predecessor (*parent*)
  - All cells are reachable from the *root*
  - A cell with no children is called a *leaf*
- Special case: *binary tree*
  - Binary tree cells have a left and a right child
  - Either or both children can be null
- Trees are useful for exposing the recursive structure of natural language and computer programs