CSE250 Week 10: Trees (mostly...)

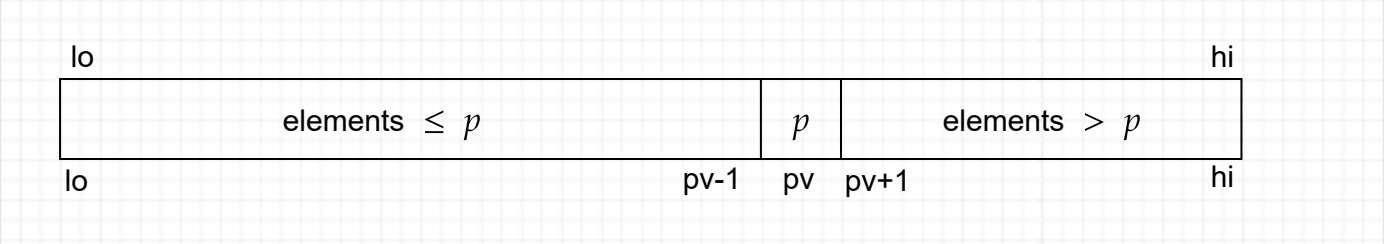
...But first, finishing the copnceptual content of Chapter 15

QuickSort

The text soft-pedals what other texts highlight as two conceptual novelties of this algorithm:

1. Analysis of a deterministic algorithm based on random distributions of data.
2. Randomizing the algorithm itself, to take "bad-luck data" out of the mix.

First, the basic D&C format of the algorithm: It selects an element `p` called the **pivot** from the array and then the **partition** step first rearranges the elements to look like this:



Here `pv` is the array index of the pivot element in the ultimate sorted order, which is what the `partition` step computes. The elements to its right and left are not necessarily sorted already, but they can be sorted in two recursive calls of `quickSort`: this is the D&C part. In the bad-luck case where p happened to be chosen as the least element, we get `pv=0` so `pv-1` does not exist, whereupon we get just one recursive call on indices `pv+1=1` to `hi=n-1`. (The indices here are inclusive on both ends.) In case `p` is unluckily picked as the largest element, things are similar in mirror image. At least progress to termination is guaranteed, but the edge-cases have no "D&C advantage."

Overall, the execution has the following characteristics:

- It sorts an array **in-place** via swaps, without wholesale copying.
- Alternately, it can sort a (linked-)list via cell rearrangements, thus avoiding the appearance of data copying, but this puts you in the simplest case of situation 1.
- It recurses on contiguous sub-ranges of the array (or rearranged pieces of the list), which again helps it work in-place.
- The $\Theta(n)$-time combination step happens *before* the recursive calls. So it is "D&C tail recursive", which means that all the work happens on the way "going down the stack" (text, p484 middle) rather than on the way up as with MergeSort.
- The recursive calls are not guaranteed to be on $\frac{n}{c}$ sized subsets, where $c > 1$ is a fixed

constant, let alone $c = 2$ as with MergeSort. But $c > \frac{3}{2}$ can be argued to hold "most of the time", which is enough to give overall running time $\Theta(n \log n)$ **in most cases**. "Bad luck", however, can yield cases where one of the two subsets steps down only to size $n-1$ or $n-2$ or so, which raises the **absolute worst-case** complexity to "$O(n^2)$".

The first four points make the $\Theta(n \log n)$ time in the (good case of the) fifth point come with a low principal constant in practice. This is what puts the "Quick" into QuickSort. [And is how Sir C.A.R. "Tony" Hoare more than made up for his "billion dollar blunder" of inventing the null reference.]

The `partition` step is prefaced by selecting one element `p` of the array as the **pivot**. We want `p` to be as close to the **median** of the array as possible---but nobody knows how to compute the median in $O(n)$ time to begin with. So we resort to various levels of guessing-and-hoping:

a) Just pick `p` to be the first element in the array (or list).
b) Pick `p` to be the middle element of the array. (Not as friendly to lists.)
c) Pick `p` at random from the array. This is what creates a **randomized algorithm**.
d) Pick `p` to be the middle of three elements: usually the first, last, and middle.

Policy d) is called "Median-of-Three QuickSort" and is close to the most common policy. Policy a) is the quickest when it doesn't run afoul of bad data---weirdly, data that is already almost-sorted counts as a bad case here.

[Second half of lecture was a demo of code for MergeSort and QuickSort applied to the Fallows1898.txt dictionary. Some highlights---and lowlights:

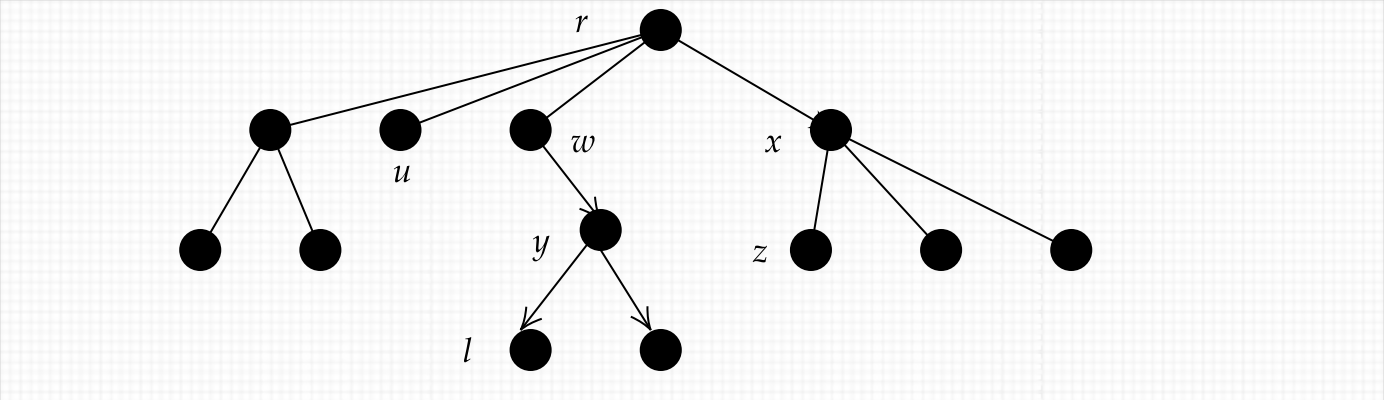
- The non-tail recursive code "`merge1`" for merge causes a StackOverflow, just as the text warns. This is even though the array being sorted (after conversion to a List) has only 6,175 entries. It is not an infinite-recursion error; my lecture actually showed that it was making progress just before it bombed.
- The tail-recursive version "`merge2`"---as noted, basically identical to the text's code---works fine.
- QuickSort is about 4x quicker---when the pivot element is selected randomly.
- When the first element is used as the pivot, its time was similar to MergeSort. This is because Fallows1898.txt is an example of data that is already nearly-sorted. Only some words, notably "Vole", are out of place.
- Picking the middle element should have worked better, but didn't...hmmm...

The last bullet exemplifies why playing with code is a never-ending time sink unto itself. :-]

Trees

Abstractly speaking, a **tree** is a connected undirected graph T with no cycles. Equivalently, T is a connected graph with n nodes and $n-1$ edges, for some (finite) number n.

A **rooted tree** distinguishes one node r as being the **root**. The root r is usually portrayed as being uppermost, i.e., trees "grow down"---like with genealogical trees. Then edges are regarded as radiating away from the root, so that they become directed edges after all. (But, often we implement trees with links going both ways---so don't insist on calling them directed graphs.) Here is a "general tree":



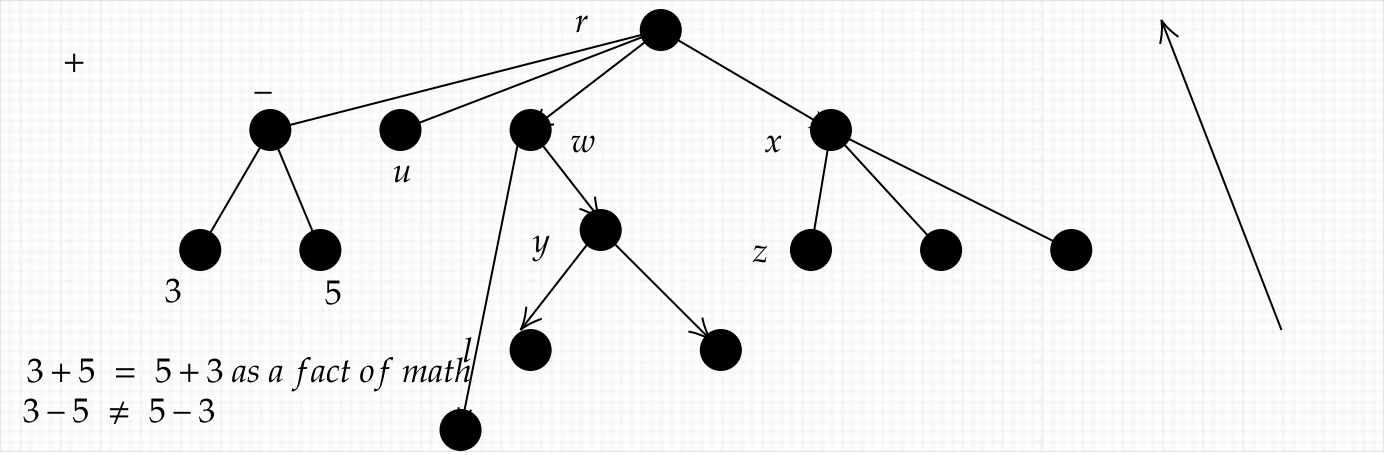
- Every node u other than the root has exactly one edge to a node v that is closer to r. (In the case of u shown, v equals r itself.) Then v is called the **parent** of u.
- A node with no edges going away from r (in the directed view we say: having no out-edges), is a **leaf**. Note that u is a leaf, even though it is closer to r than many other nodes.
- Every non-leaf node, including the root, is called an **interior node**.
- Every interior node w has one or more **children**, meaning nodes connected by edges going away from the root. The node w shown has just one **child** (which happens to be an interior node), while x has three children (which all happen to be leaves). Fellow children of the same node are called **siblings**, and the terms **descendant** and **ancestor** have their familiar meanings from genealogy.
- The **valence** of a node is its number of children. (The word **degree** is often used interchangeably with valence, but technically they are equal only if the tree is regarded as a directed graph.)
- A **binary tree** is one in which every interior node---including the root---has valence 2. A **unary-binary** tree allows nodes of valence 1 too. **Binary search trees** (**BST**s) are in fact unary-binary trees.
- The **depth** of a node u is the number of edges on the path from u to the root. Nodes of equal depth, whether siblings or not, are on the same **level**.
- The **height** of a node u can be reckoned two ways: as the maximum distance to a leaf **below** u,

or (as the text says), the difference between the maximum depth in the tree and the depth of u.

- The **height** h of the whole tree, however, is always the same as the maximum depth of a leaf. The height of the above tree is 3. The height of x is 1 above its own leaves, but 2 overall. You can call u in the above tree a "high leaf".

Now for a notion that (IMHO strangely) does not have a standard name and which the text takes for granted on page 494: A tree is **"sequenced"** if the children of every internal node are listed in sequence. The sequence is usually written **left-to-right**. This specifies a unique **layout** for the tree in two dimensions, as in the above diagram. *The sequencing should not be confused with the different traversal orders defined below.* The sequencing determines those orders, but is not the same as them.

A **path** is a special case of a tree where there is one leaf at the end, the root at the beginning, and every other node has one parent edge toward the root and one edge away from it. This is the graph of a linked list. If you have a singly linked list, then each edge is directed away from the root. If you have a doubly linked list, then it's an undirected path. (If you have a circularly linked list, then the graph is a **cycle**, which is not a tree.)



Implementing Trees

As with lists, trees can be implemented via direct recursion without needing a separate concept of `Node`. Immutable trees are done that way. But we will mainly use standard mutable trees with nodes. Among them there are several choices:

1. Every node in the tree uses the same `Node` class, which has:
   (a) a `list` (or other `Seq`uence) of children; leaves have the empty list.
   (b) optionally, a `parent` link.
   (c) usually, a data item. Sometimes data is only in the leaves, or sometimes internal nodes have a different kind of item (such as an operator) from the leaves.
   (d) Associating data with edges, say in the form of a weight, is rarer, so unlike with general

graphs there isn't usually a separate `Edge` class.

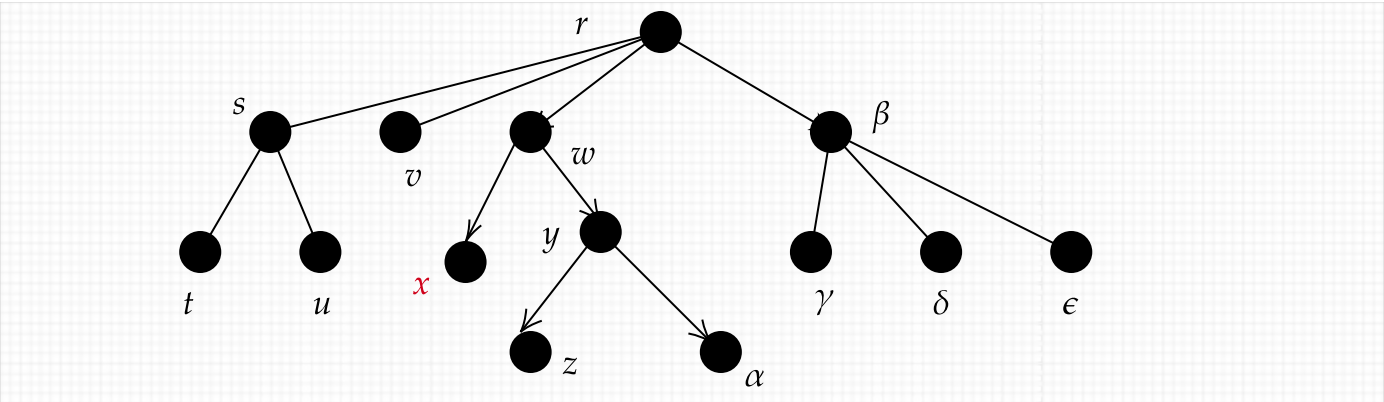
2. Leaves can use a separate `Leaf` class.
3. A binary tree usually specifies the children of a node as `left` and `right` rather than as a list. The text puts null pointers in those fields for leaves, and for one of those fields in the case of a unary node in a unary-binary tree. An alternative is to use a sentinel end node.
4. The "**Circularly Linked Tree**" or "**CLR tree**", used in the famous "MIT White Book" by Cormen, Leiserson, and Rivest (and now Stein) and by the C++ Standard Template Library, makes the root and the sentinel end node be parents of each other, too.

Traversals

1. Level order, which is what BFS does.
2. Preorder
3. Postorder --- not the same as mirror-image of preorder, still L to R.
4. Inorder (well-defined for binary and unary/binary trees).

Visualizations: Geeks for Geeks
https://www.programiz.com/dsa/tree-traversal
http://ceadserv1.nku.edu/longa/classes/mat385_resources/docs/traversal.htm

The labels in our starting diagram are in **preorder**, maybe the most natural default order. The Roman alphabet starting with r for root wraps into the Greek alphabet for the last five nodes.



1. Level order is: $r\ s\ v\ w\ \beta\ t\ u\ x\ y\ \gamma\ \delta\ \epsilon\ z\ \alpha$. Note that this fixes a misleading aspect of the diagram: x should be brought up to "level 2" because it is connected to w in level 1. BFS does not care about the layout of the tree, or whether you wanted to pretend that x was on a level by itself.

The other three orders all use the same "multiple-visitation order", depth-first and left-to-right.(*Caveat:* DFS itself gives the visitation order $r\ s\ v\ w\ \beta\ \gamma\ \delta\ \epsilon\ x\ y\ z\ \alpha\ t\ u$ which is frankly weird; "DFS order" often

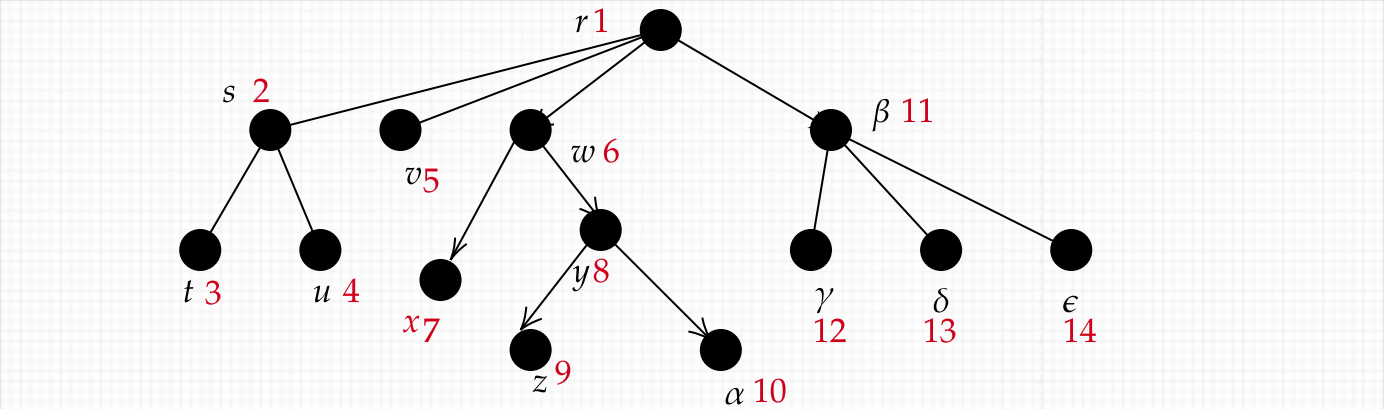
refers to the popping order $r\ \beta\ \epsilon\ \delta\ \gamma\ w\ y\ \alpha\ z\ x\ v\ s\ u\ t$, which we'll see is a mirror image of left-to-right preorder.) The visitation order (**Euler Tour**) of the left-to-right transversal is:

$$r\ s\ t\ s\ u\ s\ r\ v\ r\ w\ x\ w\ y\ z\ y\ \alpha\ y\ w\ r\ \beta\ \gamma\ \beta\ \delta\ \beta\ \epsilon\ \beta\ r_{\text{end}}$$

Each node appears a number of times equal to its degree, which is $1\ +$ its valence. The three major traversal orders are subsequences in which each node appears once. The difference is when each node is "**expanded**" or otherwise acted on. (Note: In the "CL&R Tree", the sentinel end node is the parent of root as well as the `null`-saver of leaves, so it is natural to park there at the end.)

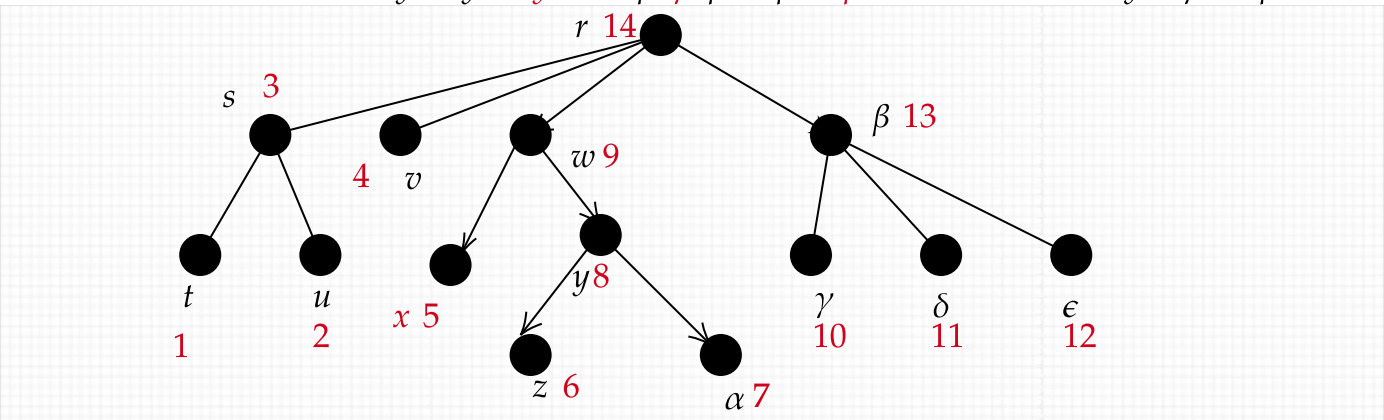
2. **Preorder** acts on the *first* visitation of each node:

$r\ s\ t\ s\ u\ s\ r\ v\ r\ w\ x\ w\ y\ z\ y\ \alpha\ y\ w\ r\ \beta\ \gamma\ \beta\ \delta\ \beta\ \epsilon\ \beta\ r\ =\ r\ s\ t\ u\ v\ w\ x\ y\ z\ \alpha\ \beta\ \gamma\ \delta\ \epsilon$ end



3. **Postorder** always acts on the **last** occurrence of each node:

$r\ s\ t\ s\ u\ s\ r\ v\ r\ w\ x\ w\ y\ z\ y\ \alpha\ y\ w\ r\ \beta\ \gamma\ \beta\ \delta\ \beta\ \epsilon\ \beta\ r\ =\ t\ u\ s\ v\ x\ z\ \alpha\ y\ w\ \gamma\ \delta\ \epsilon\ \beta\ r$ end

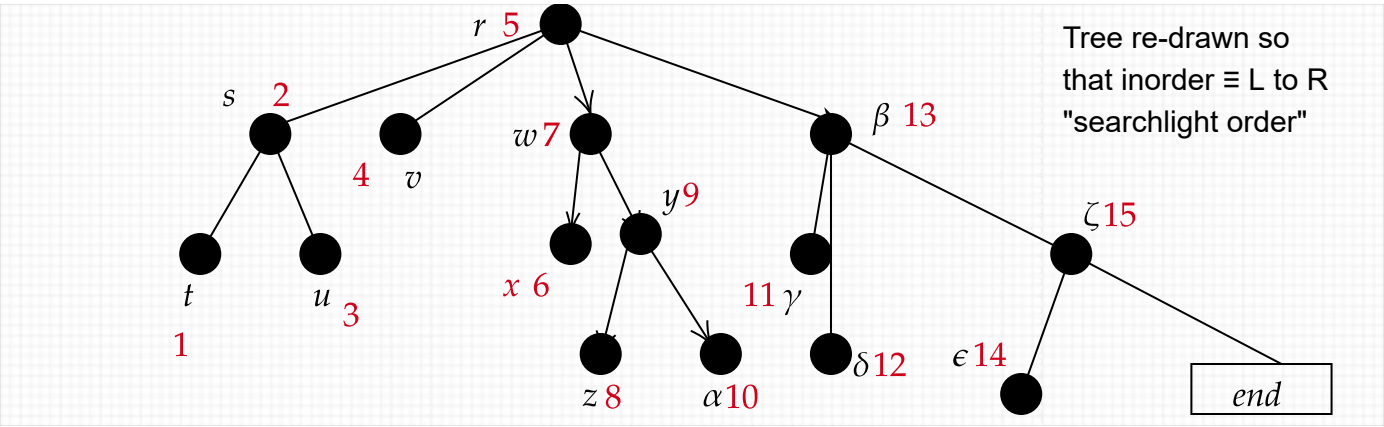


This is the logical order to use when evaluating expression trees. It is also why **postfix notation** (also called **reverse Polish notation** (**RPN**) because this was expounded best by Polish logicians early in

the 20th Century, especially **Jan Łukasiewicz** (note cite of UB's recently-late John Corcoran), but he worked right-to-left) is unambiguous without needing parentheses.

4. **Inorder** acts on one of the **middle** occurrences, whenever the valence is at least 2. In a pure binary tree, every internal node appears exactly three times and so there is a unique middle occurrence. In this example, we have three choices for r and two for β. One of them is:

$r\ s\ t\ s\ u\ s\ r\ v\ \mathbf{r}\ w\ x\ w\ y\ z\ y\ \alpha\ y\ w\ r\ \beta\ \gamma\ \beta\ \delta\ \beta\ \epsilon\ \beta\ r\ = t\ s\ u\ v\ r\ x\ w\ z\ y\ \alpha\ \gamma\ \delta$



In a **unary-binary** tree, we have a further wrinkle depending on whether we distinguish between the child of a unary node (an "**elbow**") bending right or bending left. The other child is `null` in the text, or (better, IMPHO) is the `end` sentinel in the CL&R tree. When the left-child is real, inorder goes there first.

Definition: A **binary search tree** (**BST**) is a unary-binary tree with left-right specified elbow nodes whose keys are sorted by inorder.

Binary Search Tree Visualization