Trees

Computer Science & Engineering 235: Discrete Mathematics

Christopher M. Bourke
cbourke@cse.unl.edu

Trees I

General graphs are great data structures. However, they often have no unifying, underlying mathematical structure from which we can design efficient algorithms.

As an alternative, we can consider classes of graphs that have some underlying structure; e.g. cycles, bipartite, grid, etc.

One of the most widely used classes of graphs are *trees*.

Trees II

Definition

A *tree* is an acyclic graph.

Lemma

An graph is a tree if and only if there is a unique path between any two of its vertices.

Trees III

Trees are used as models in hundreds of applications in computer science, chemistry, geology, botany, etc.

The are useful in modeling any hierarchical structure, etc.

In computer science, trees are useful data structures for searching, indexing, and other applications handling data.

Terminology I

Like any plant, we can *root* a tree; we select (arbitrarily) a node from a tree as the root. The distance of a path from the root to a node defines is ascendency.

- Let v be a node in a tree T.
- A node immediately preceding v is a *parent* of v.
- A node immediately following v is a *child* of v.
- A node preceding/following v is an *ancestor/descendent* of v.
- If $\deg(v) = 1$, then v is a *leaf*.
- If a node is not a leaf, it is an *internal node*.
- A *subtree* with v as its root is the subgraph of v and all of its descendants.

Terminology II

Definition

A rooted tree T is m-ary if every internal node has at most m children. T is a *full m-ary* tree if every node has *exactly* m children.

For $m = 2$, we have a *binary* tree.

Sometimes an *ordering* is defined on a node's children. In this case, we consider *ordered rooted trees*.

In a binary tree, each node has a *left child* and a *right child*, each defining a *left subtree* and *right subtree* respectively.

Properties I

Theorem

A tree $T = (V, E)$ with $|V| = n$ has $n - 1$ edges.

proof: Another easy induction.

Theorem

A full m-ary tree with i internal vertices contains $n = mi + 1$ vertices.

For binary trees with i internal nodes, there are $n = 2i + 1$ vertices.

The number of internal nodes i, leaves l, and n total vertices are all related. In particular, if we know any one of them, then we know the other two.

Properties II

Theorem

Let T be a full m-ary tree.

1. *If T has n vertices, then T has $i = (n - 1)/m$ internal nodes and $l = ((m - 1)n + 1)/m$ leaves.*
2. *If T has i internal vertices, then T has $n = mi + 1$ vertices and $l = (m - 1)i + 1$ leaves.*
3. *If T has l leaves, then T has $n - (ml - 1)/(m - 1)$ vertices and $i = (l - 1)/(m - 1)$ internal nodes.*

Trees I

More Terminology

The *level* of a vertex v in a tree is the length of the unique path from the root to v. The root is at level 0.

The *height* of a tree is the longest path from the root to a leaf. (Alternatively, you may consider *depth*). Therefore, the height is equal to the lowest (highest numbered) level.

Many applications call for *balanced* trees—trees were each subtree is roughly the same height.

A rooted m-ary tree of height h is *balanced* if all leaves are at levels h or $h - 1$.

Theorem

There are at most m^h leaves in an m-ary tree of height h.

Trees II

More Terminology

If the tree is full, this equality is strict. That is,

$$l = m^h$$

In particular, a full binary tree has 2^h leaves.

By one of the previous properties, this also bounds the number of vertices.

Corollary

Let T be an m-ary tree of height h.

If T has l leaves then

$$h \geq \lceil \log_m l \rceil$$

If T is full and balanced then this equality is strict.

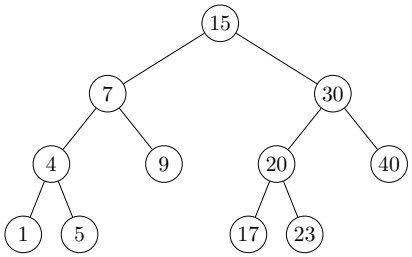
Binary Search Trees I

A *binary search tree* (BST) is a binary tree with each vertex v labeled with a (unique) key such that

- The key of the left-sub-child of v is less than the key value of v.
- The key of the right-sub-child of v is greater than the key value of v.

This recursive property ensures that all keys in the left-sub *tree* are less than the key value of v and that those in the right-sub tree are greater.

Binary Search Trees II



Binary Search Trees III

Operations that can be performed on a binary search tree include

- Search (access)
- Delete
- Insert

Binary search trees are nice data structures because these operations can usually be performed in $\mathcal{O}(\log n)$ time. Here n is the number of nodes.

BST Searching I

Say we have a BST, T with a key value k and left T_L and right T_R sub-trees. We can search for an item v in T in the obvious manner.

We can compare k to each node and recursively check the proper sub-child.

BST Searching II

Algorithm (BSTSearch)

```
1  IF v = k THEN
2  |   return true
3  END
4  ELSE IF v < k THEN
5  |   return BSTSearch(T_L)
6  END
7  ELSE IF v > k THEN
8  |   return BSTSearch(T_R)
9  END
10 ELSE IF T = φ THEN
11 |   return false
12 END
```

BST Searching III

Such a tree traversal has a worst case run time of $\Theta(h)$ where h is the height of T.

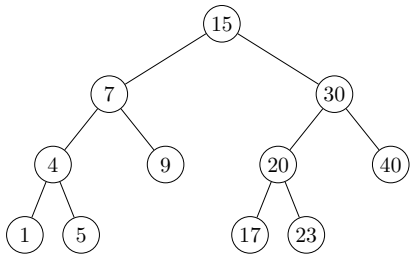
If the tree is *balanced*, then one can expect that $h = \mathcal{O}(\log n)$.

Unfortunately, the worst case can still be linear. When does this happen?

BST Searching IV

Example

Search for 17, 30, and 6 in the previous example.



BST Deleting I

Binary Search Trees are easily implemented via doubly linked lists having *two* "next" pointers (left and right sub-children).

Thus, deletion in a BST is simply a matter of changing some pointers (and cleaning up memory).

To delete a node, we first have to *search* for it.

If the node is a leaf, then we simply delete the pointer to it.

Also, if there is only one child, then we simply *promote* it.

BST Deleting II

Otherwise, we have to "promote" some lower node that *preserves* the binary search tree properties.

Here, we have one of to choices:

- Choose the minimal element among the greater elements (right sub-tree); or
- Choose the maximal element of the lesser elements (left sub-tree).

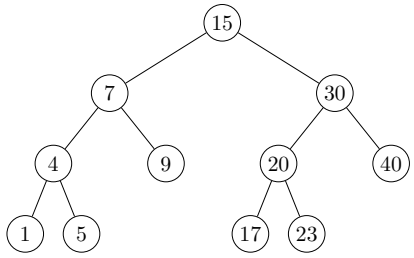
To find such an element, it suffices to traverse to the left (right) sub-tree and then traverse all the way to the right (left).

BST Deleting I

Example I

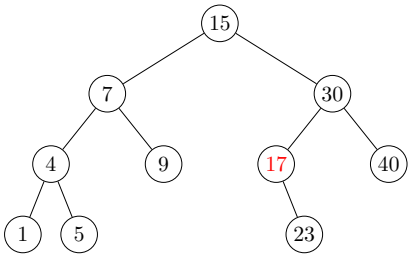
Example

Delete 20 from the previous tree.



BST Deleting II

Example I

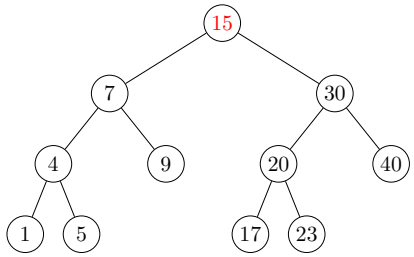


BST Deleting I

Example II

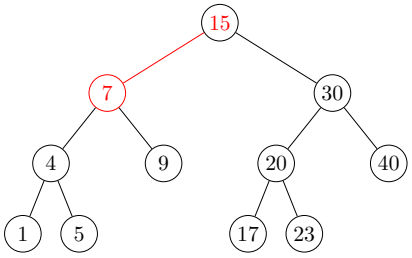
Example

Delete 15 from the previous tree.



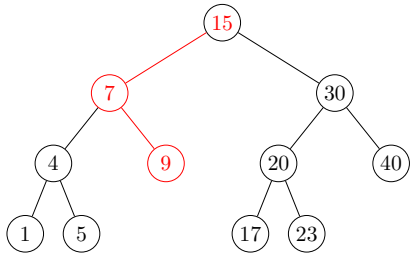
BST Deleting II

Example II



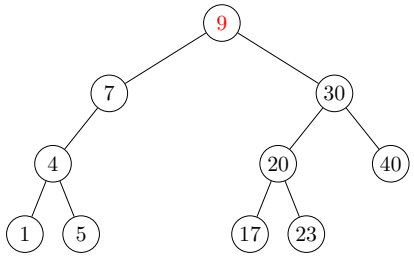
BST Deleting III

Example II



BST Deleting IV

Example II



BST Insertion

For an ordinary binary search tree, we always insert a key k as a leaf. Therefore, insertion is essentially the same as searching.

In particular, we search for a node, v such that

- $k < v$ and the left sub-child of v is null or
- $k > v$ and the right sub-child of v is null.

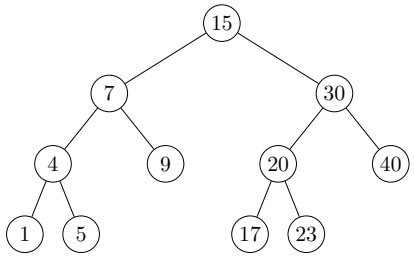
Once we've discovered such a node, insertion is simply a matter of changing some pointers.

BST Inserting I

Example I

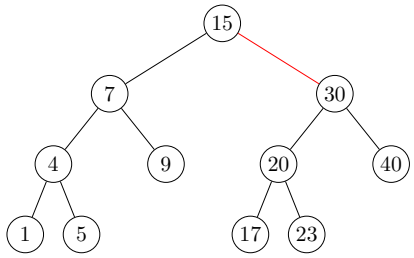
Example

Insert 16 into the previous tree.



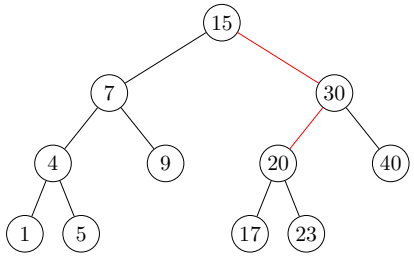
BST Inserting II

Example I



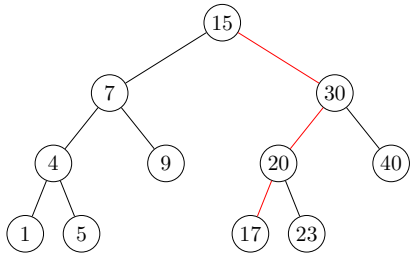
BST Inserting III

Example I



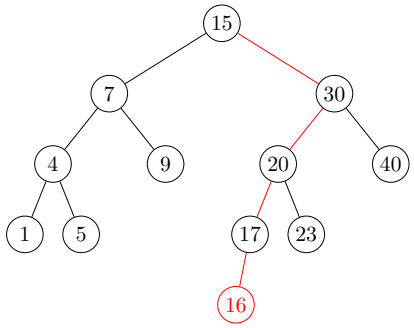
BST Inserting IV

Example I



BST Inserting V

Example I



Coding Theory I

Coding Theory is the study and theory of *codes*—schemes for transmitting data.

There are two disciplines in coding theory. The first is where we try to "pad" out a message with minimal redundant information that ensures reliability.

Error correcting codes allow us to detect and/or correct errors in transmission with a minimal blow-up in message size.

Coding Theory II

The second deals with *compressing* data to save space. You should already be familiar with data compression.

- MP3s (uses a form of Huffman coding, but is information lossy)
- jpegs, mpegs, even DVDs
- `pack` (straight Huffman coding)
- `zip`, `gzip` (uses a Ziv-Lempel compression algorithm)

Coding I

Say we have a fixed alphabet, Δ of size n. A *coding* is a mapping of this alphabet to a collection of bit vectors, $\Delta \to \{0,1\}^*$ called *codewords*.

For example, the ASCII coding standard maps 256 unique characters to bit vectors of length 8. This is known as a *fixed length encoding* scheme.

Coding II

For many applications, some characters will occur with higher frequency than others.

- English language ASCII files will use `t,s,r` far more than `z,q`, and may never use $\hat{o}$.
- Data files may contain long strings of 0s or 1s or repeat certain patterns.
- Video and audio may contain redundant information (black fades, "still" scenes).

Coding III

It makes more sense to assign a *shorter* codeword to more frequent characters and longer codewords to less frequent characters.

This *variable-length encoding* scheme reduces the overall *average* length of a codeword used in a file.

In fact, for a given file, if a character doesn't appear at all, we do not even need to consider encoding it.

Coding IV

However, there is an immediate problem that should be apparent.

- With fixed length codes, you implicity know where a codeword begins and ends.
- With variable length codes, you don't.
- Thus, a *delimiter* is necessary.
- Or we can define our coding to be a *prefix free code*.

Prefix Free Codes

A prefix-free coding is one in which no *whole* codeword is the prefix of another (other than itself of course).

Scanning is thus achieved by parsing a codeword until it matches a whole valid codeword.

We then know that the next bit will be the beginning of a new codeword.

Example

- $\{0, 01, 101, 010\}$ is not a prefix free code.
- $\{10, 010, 110, 0110\}$ is a prefix free code.

Building Prefix Free Codes

A simple way of building a prefix free code is to associate codewords with the *leaves* of a binary tree (not necessarily full).

Each edge corresponds to a bit, 0 if it is to the left sub-child and 1 to the right sub-child.

Since no simple path from the root to any leaf can continue to another leaf, then we are guaranteed a prefix free coding.

Using this idea along with a greedy encoding forms the basis of *Huffman Coding*.

Huffman Coding I

We consider here a fixed file f with a fixed alphabet Δ of characters appearing in f.

The relative frequencies are the number of times each $x \in \Delta$ appear in f divided by the total number of characters in f.

Huffman Coding II

Algorithm (HuffmanEncoding)

```
1  FOREACH x ∈ Δ DO
2      Initialize a single node tree T_x.
3      Associate a weight wt(T_x) = freq(x)
4  END
5  REPEAT
6      Create a new root node r
7      Combine the two trees with the smallest weights under r. //a weight of
         a tree is defined as the sum of the weights of its leaves
8
9      Update the weight of the new tree
10 UNTIL Only one tree exists
```

Huffman Trees

The tree constructed in Huffman's algorithm is known as a *Huffman Tree* and it defines a *Huffman Code*.

Example

Construct the Huffman Tree and Huffman Code for a file with the following content.

character	A	B	C	D	E	F	G
frequency	0.10	0.15	0.34	.05	.12	.21	.03
codeword							

Compression Ratio I

For an n character alphabet, a fixed-length coding would require a minimal codeword length of $\log_2 n$.

In particular, the previous example would require 3 bits per codeword.

Instead, the Huffman coding gave us an average codeword length of

$$.10 \cdot 3 + .15 \cdot 3 + .34 \cdot 2 + .05 \cdot 4 + .12 \cdot 3 + .21 \cdot 2 + .03 \cdot 4 = 2.53$$

Compression Ratio II

This means we got a compression ratio of

$$\frac{(3 - 2.53)}{3} = 15.67\%$$

For text files, `pack` (Huffman Coding), claims an average compression ratio of 25-40%.

Depending on the type of data/distribution in any given file, you may expect a lot more or a lot less.

Optimality

It can be shown that Huffman coding actually compresses a file in an optimal way.

That is, no other compression scheme could hope for an (asymptotically) better compression ratio, *ever*.

Note here that we are only considering information loss-less schemes, of course MP3s, jpegs, etc are information *lossy* and can achieve better compression at the expense of losing information.

Worst Case Encoding

There are a couple of situations in which Huffman coding does nothing for us—that is, the compressed file is no more smaller than the original.

- When the probability distribution is uniform: $p(x) = p(y)$ for all $x, y \in \Delta$. What kind of tree will be built?
- When the probability distribution follows a fibonacci sequence (the sum of each of the two smallest probabilities is less than equal to the next highest probability for all probabilities) What kind of tree do we have in this case?

Binary Tree Traversals I

There are three systematic ways to traverse, that is *visit* every node, of a binary tree.

- **Preorder Traversal** – Nodes are visited in root-left-right order.
- **Inorder Traversal** – Nodes are visited in left-root-right order.
- **Postorder Traversal** – Nodes are visited in left-right-root order.

Binary Tree Traversals II

There is a generalization for any m-ary tree:

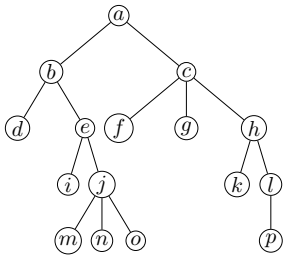
- **Preorder Traversal** – Nodes are visited in (root)-(left child)-(remaining children, left to right) order.
- **Inorder Traversal** – Nodes are visited in (left child)-(root)-(remaining children, left to right) order.
- **Postorder Traversal** – Nodes are visited in (left child)-(remaining children, left to right)-(root) order.

Binary Tree Traversals

Example

Example

(9.3.8) Give the pre-, in-, and postorder traversals of the following tree.



Binary Tree Traversals

Example Continued

The traversals are as follows.

- Preorder

  $a, b, d, e, i, j, m, n, o, c, f, g, h, k, l, p$

- Inorder

  $d, b, i, e, m, j, n, o, a, f, c, g, k, h, p, l$

- Postorder

  $d, i, m, n, o, j, e, b, f, g, k, p, l, h, c, a$

Notations I

Trees can be used to represent mathematical expressions (arithmetic, Boolean, etc).

Preorder, Inorder, and Postorder traversals of such trees correspond to prefix, infix and postfix *notation*.

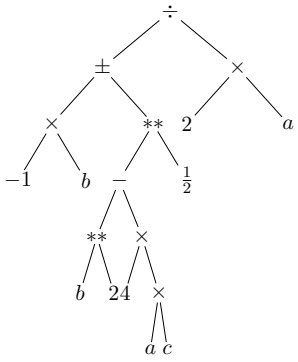
Example

The following tree represents the quadratic formula,

$$\frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

Give the prefix, infix and postfix notation for this formula.

Notations II



Notations III

Infix notation is what we usually use, but may be ambiguous without proper parentheses.

Prefix notation (or *Polish notation*) is unambiguous;

```
(* (+ 0 1) (+ 2 3)) = * + 0 1 + 2 3
```

It is still used in computer languages such as LISP.

Postfix notation is also unambiguous and can be very efficiently realized using a stack data structure. It is still used in many engineering calculators (TI-89). It is also used in the PostScript file format.