Trees

Steven J. Zeil

August 3, 2013

Contents

1 Tree Terminology **2**

2 Tree Traversal **4**
2.1 Recursive Traversals 5

3 Example: Processing Expressions **7**

4 Example: Processing XML **13**

5 Using Trees for Searching **21**
5.1 How Fast Are Binary Search Trees? 24

Most of the data structures we have looked at so far have been devoted to keeping a collection of elements in some linear order.

Trees

Trees are the most common non-linear data structure in computer science.

- Useful in representing things that naturally occur in hierarchies
  - (e.g., many company organization charts are trees)
- and for things that are related in a "is-composed-of" or "contains" manner
  - (e.g., this country is composed of states, each state is composed of counties, each county contains cities, each city contains streets, etc.)

Trees also turn out to be exceedingly useful in implementing fast searching and insertion.

- Properly implemented, a tree can be both searched and inserted into in O(log N) time.

. . . .

1 Tree Terminology

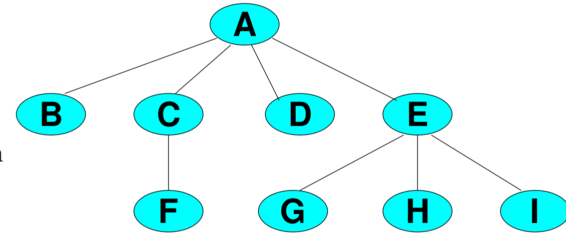
Definition

A *tree* is a collection of nodes.

If nonempty, the collection includes a designated node `r`, the *root*, and zero or more (sub)trees `T1`, `T2`, ..., `Tk`, each of whose roots are connected by an *edge* to `r`.

. . . .

A Tree



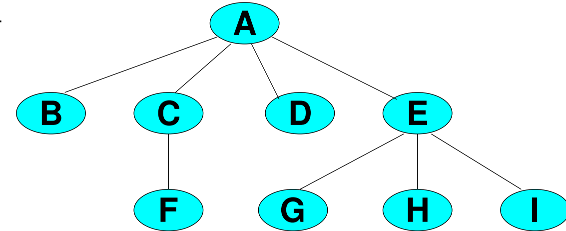
The collection of nodes shown here is a tree.

- We can designate A as the root, and we note that
- the collections of nodes {B}, {C, F}, {D}, and {E,G,H,I}, together with their edges, are all trees whose own roots are connected to A.

. . . .

It's a subtle, but important point to note that in discussing trees, we sometimes focus on the things connected to the root as individual nodes, and other times as entire trees.

Botanical Families

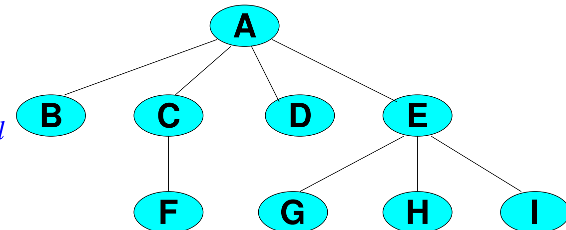


Focusing on a tree as a collection of nodes leads to some other terminology:

- Each node except the root has a *parent*.
- Parent nodes have *children*. Nodes without children are *leaves*.
- Nodes with the same parent are *siblings*.

. . . .

Binary and General Trees



- A tree in which every parent has at most 2 children is a *binary tree*.
- Trees in which parents *may* have more than 2 children are *general trees*.

. . . .

2 Tree Traversal

Tree Traversal

Many algorithms for manipulating trees need to “traverse” the tree, to visit each node in the tree and process the data in that node. In this section, we’ll look at some prototype algorithms for traversing trees.

- All of these use recursion, because trees are easiest to handle that way.

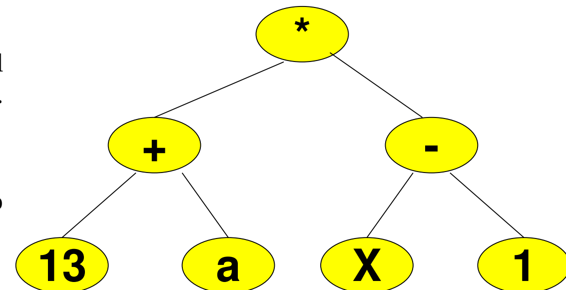
. . . .

Kinds of Traversals

- A *pre-order* traversal is one in which the data of each node is processed `before` visiting any of its children.
- A *post-order* traversal is one in which the data of each node is processed `after` visiting all of its children.
- An *in-order* traversal is one in which the data of each node is processed after visiting its left child but before visiting its right child.
  - This traversal is specific to binary trees.

. . . .

Example: Expression Trees



Compilers, interpreters, spreadsheets, and other programs that read and evaluate arithmetic expressions often represent those expressions as trees.

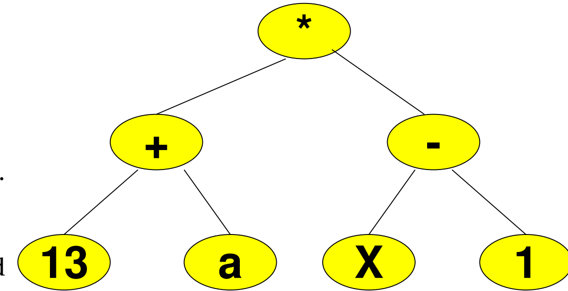
- Constants and variables go in the leaves,
- Each non-leaf node represents the application of some operator to the subtrees representing its operands.

The tree here, for example, shows the product of a sum and of a subtraction.

. . . .

Traversing an Expression Tree

If we were to traverse this tree, printing each node as we visit it, we would get:



Pre-order: * + 13 a - x 1

In-order: 13 + a * x - 1

Compare to ((13+a)*(x-1)), the "natural" way to write this expression.

Post-order: 13 a + x 1 - *

Post-order traversal yields post-fix notation, which in turn is related to algorithms for expression evaluation.

2.1 Recursive Traversals

Recursive Traversals

Let's suppose that we have a binary tree whose nodes are declared as shown here.

```
struct BinaryTreeNode
{
  T data;
  BinaryTreeNode* left;
  BinaryTreeNode* right;
};
```

It's fairly easy to write pre-, in-, and post-order traversal algorithms using recursion.

A Basic Traversal

```
void basicTraverse (BinaryTreeNode* t)
{
  if (t != 0)
    {
     basicTraverse(t->left);
```

```
    basicTraverse(t->right);
  }
}
```

This is the basic structure for a recursive traversal. If this function is called with a null pointer, we do nothing. But if we have a real pointer to some node, we invoke the function recursively on the left and right subtrees. In this manner, we will eventually visit every node in the tree. The problem is, this basic form doesn't *do* anything.

Pre-Order Traversals

But we can convert it into a pre-order traversal by applying the rule:

> process the node *before* visiting its children

```
void preOrder (BinaryTreeNode* t)
{
  if (t != 0)
    {
      foo (t->data);
      preOrder(t->left);
      preOrder(t->right);
    }
}
```

Post-Order Traversals

```
void postOrder (BinaryTreeNode* t)
{
  if (t != 0)
    {
      postOrder(t->left);
      postOrder(t->right);
      foo (t->data);
    }
}
```

We get a post-order traversal by applying the rule:

> process the node *after* visiting its children

In-Order Traversals

And we get an in-order traversal by applying the rule:

> process the node *after* visiting its left descendents and *before* visiting its right descendents.

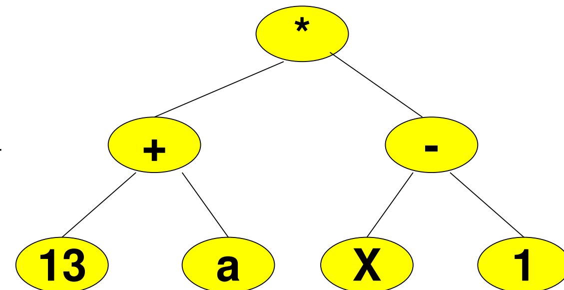
```
void inOrder (BinaryTreeNode* t)
{
  if (t != 0)
    {
      inOrder(t->Left);
      foo (t->Element);
      inOrder(t->Right);
    }
}
```

Demo

Run the different traversals

3 Example: Processing Expressions

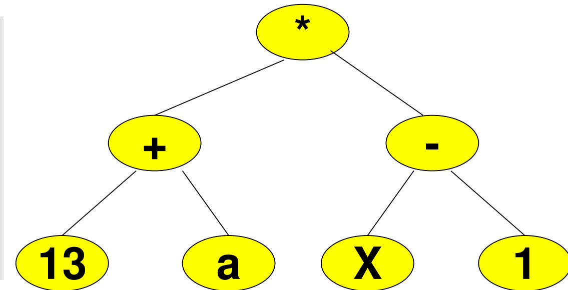
Example: Processing Expressions



Let's look at how we would process expressions like the one shown here. We'll develop a program that can simplify certain basic arithmetic expressions, e.g., converting *(2 - 2) * x + 1 * (y + z)* to *y + z*.

The Data Structure

```
class Expression {
private:
  std::string opOrVarName;
  bool thisIsAConstant;
  bool thisIsAVariable;
  Expression* left;
  Expression* right;
  int value;
   ⋮
```



Note the `left` and `right` pointers, which give this it's essential "tree-ness".

....

The Application

- `simplifier.cpp`, the main program
- `expression.h`
- `expression.cpp`

We'll look at selected parts in more detail.

....

Printing Expressions

```
std::ostream& operator<< (std::ostream& out, const Expression& exp)
{
  if (exp.isAConstant())
    out << exp.getValue();
  else if (exp.isAVariable())
    out << exp.getVariableName();
  else
```

```
    {
      if (!exp.left->isAVariable() &&!exp.left->isAConstant())
        out << "(" << *(exp.left) << ")";
      else
        out << *(exp.left);
      out << exp.getOperator();
      if (!exp.right->isAVariable() &&!exp.right->isAConstant())
        out << "(" << *(exp.right) << ")";
      else
        out << *(exp.right);
    }
  return out;
}
```

. . . .

Distributive Law

```
// Apply the distributive laws a*(b+c) => a*b + a*c
//   and a*(b-c) => a*b - a*c whenever possible
void Expression::distribute()
{
  if ((!thisIsAVariable) && (!thisIsAConstant))
    {
      left->distribute();
      right->distribute();
      if (opOrVarName == "*")
        {
          if (left->opOrVarName == "+" || left->opOrVarName == "-")
            {
              // (a + b) * c
```

```
            Expression* a = left->left;
            Expression* b = left->right;
            Expression* c = right;
            *this = Expression(left->opOrVarName,
                               Expression ("*", *a, *c),
                               Expression ("*", *b, *c));
            distribute();
          }
        else if (right->opOrVarName == "+" || right->opOrVarName == "-")
          {
            // a * (b + c)
            Expression* a = left;
            Expression* b = right->left;
            Expression* c = right->right;
            *this = Expression(right->opOrVarName,
                               Expression ("*", *a, *b),
                               Expression ("*", *a, *c));
            distribute();
          }
      }
  }
}
```

. . . .

Simplifying Expressions

```
// Perform trivial simplifications: 0*a => 0, 1*a => a, a/1 => a,
//    a+0 => a, a-0 => a  c1 op c2 => c3  where a is any expression,
//    the ci are constants, and op is any of * / + -
void Expression::simplify()
```

```
{
  if ((!thisIsAVariable) && (!thisIsAConstant))
    {
      left->simplify();
      right->simplify();

      if (left->isAConstant() && right->isAConstant())
        {
          // Both operands are constants, evaluate the operation
          // and change this expression to a constant for the result.
          if (opOrVarName == "+")
            value = left->value + right->value;
          else if (opOrVarName == "-")
            value = left->value - right->value;
          else if (opOrVarName == "*")
            value = left->value * right->value;
          else if (opOrVarName == "/")
            value = left->value / right->value;
          thisIsAConstant = true;
          opOrVarName = "";
          delete left;
          delete right;
          left = right = NULL;
        }
      else if (left->isAConstant())
        {
          if (opOrVarName == "+" && left->value == 0)
            {
              *this = *right;
            }
          else if (opOrVarName == "*" && left->value == 0)
```

```
        {
          *this = Expression(0);
        }
      else if (opOrVarName == "*" && left->value == 1)
        {
          *this = *right;
        }
    }
  else if (right->isAConstant())
    {
      if (opOrVarName == "+" && right->value == 0)
        {
          *this = *left;
        }
      else if (opOrVarName == "-" && right->value == 0)
        {
          *this = *left;
        }
      else if (opOrVarName == "*" && right->value == 1)
        {
          *this = *left;
        }
      else if (opOrVarName == "*" && right->value == 0)
        {
          *this = Expression(0);
        }
      else if (opOrVarName == "/" && right->value == 0)
        {
          *this = Expression(0);
        }
    }
```

```
    }
}
```

. . . .

4 Example: Processing XML

XML

XML is a markup language used for exchanging data among a wide variety of programs on the web. It is flexible enough to represent almost any kind of data.

In XML, data is described by structures consisting of nested “elements” and text.

- An element is described as an opening “tag”, the contents of the element, and a closing tag.

. . . .

XML Tags

- Tags are written inside `< >` brackets. Inside the brackets, an opening tag has a tag name followed by zero or more “attributes”. An attribute is a `name="value"` pair, with the value inside quotes. Closing tags have no attributes, and are indicated by placing the tag name immediately after a ’/’ character.
- Finally, if an element has no text and no internal elements, it can be written as `<tag attributes></tag>` or abbreviated as `<tag attributes/>`.

. . . .

An example of XML:

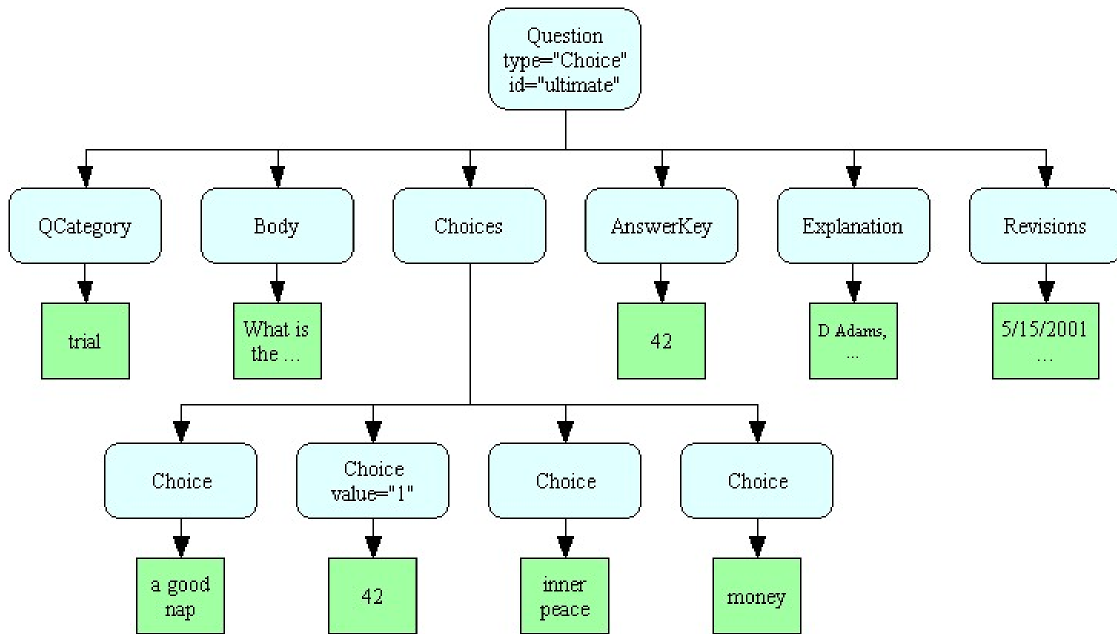
```
<Question type="Choice" id="ultimate">
<QCategory>trial</QCategory>
<Body>What is the answer to the ultimate question of life,
```

```
the universe, and everything?</Body>
<Choices>
<Choice>a good nap</Choice>
<Choice value="1">42</Choice>
<Choice>inner peace</Choice>
<Choice>money</Choice>
</Choices>
<AnswerKey>42</AnswerKey>
<Explanation>D Adams,
The Hitchhiker's Guide to the Galaxy</Explanation>
<Revisions>5/15/2001 1:35:29 PM</Revisions>
</Question>
```

....

XML and Trees

Although it may not be obvious, XML actually describes a tree structure.



For example, the structure above shows that all the elements are inside a "Question". One of those elements inside the Question is a "Choices" element, and each individual "Choice" occurs inside there.We can diagram this tree structure as shown here.

XML is closely related to HTML

- HTML allows many attributes to be written without placing the value in quotes,
- empty elements can be written as `<tagname>` instead of `<tagname/>`,
- and some non-empty elements can be written without a closing `</tagname>`

For example, the text shown here can be produced by the following XML-legal HTML:

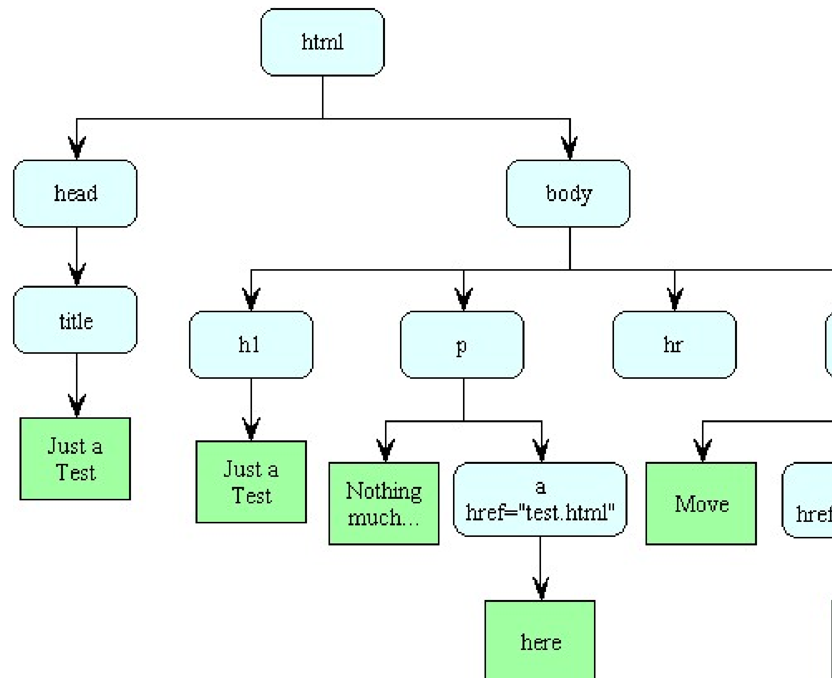
Just a test
Nothing much to see here.

Move along.

```
<html>
  <head>
    <title>Just a Test</title>
  </head>

  <body>
    <h1>Just a test</h1>
<p>Nothing much to see <a href="test.html">here</a>.
</p>
<hr/>
<p>
Move <a href="nextpage.html">along</a>.
</p>
  </body>
</html>
```

The tree structure for this HTML page is:



A program that would read a web page and print a list of all links (`<a>` elements with `href=` attributes).

```
/** Example of tree manipulation using XML documents */

#include <iostream>

using namespace std;

#include <xercesc/parsers/XercesDOMParser.hpp>
#include <xercesc/dom/DOM.hpp>
#include <xercesc/sax/HandlerBase.hpp>
```

```
#include <xercesc/util/XMLString.hpp>
#include <xercesc/util/PlatformUtils.hpp>

using namespace XERCES_CPP_NAMESPACE;

DOMDocument* readXML (const char *xmlFile)
{
  ⋮
}

string getHrefAttribute (DOMNode* linkNode)
{
  DOMElement* linkNodeE = (DOMElement*)linkNode;
  const XMLCh* href = XMLString::transcode("href");
  const XMLCh* attributeValue = linkNodeE->getAttribute(href);
  return string(XMLString::transcode(attributeValue));
}


void processTree (DOMNode *tree)
{
  if (tree != NULL)
    {
      if (tree->getNodeType() == DOMNode::ELEMENT_NODE)
        {
          const XMLCh* elName = tree->getNodeName();
          const XMLCh* aName = XMLString::transcode("a");
          if (XMLString::equals(elName, aName))
            cout << "Link to " << getHrefAttribute(tree) << endl;
        }
      for (DOMNode* child = tree->getFirstChild();
           child != NULL; child = child->getNextSibling())
        processTree(child);
    }
```

```
}


int main(int argc, char **argv)
{
  if (argc != 2)
    {
      cerr << "usage: " << argv[0] << " xmlfile" << endl;
      return 1;
    }

  // Initialize the Xerces XML C++ library
  try {
    XMLPlatformUtils::Initialize();
  }
  catch (const XMLException& toCatch) {
    char* message = XMLString::transcode(toCatch.getMessage());
    cout << "Error during initialization! :\n"
         << message << "\n";
    XMLString::release(&message);
    return 1;
  }

  DOMDocument* doc = readXML(argv[1]);
  if (doc == 0)
    {
      cerr << "Could not read " << argv[1] << endl;
      return 2;
    }
  processTree (doc->getDocumentElement());

  // Cleanup
  doc->release();
  return 0;
```

```
}
```

based upon the Xerces-C++ library.

If this code is compiled and run on the HTML page we have just seen, it would print

```
Link to test.html
Link to nextpage.html
```

Processing XML as a Tree

```
void processTree (DOMNode *tree)
{
  if (tree != NULL)
    {
      if (tree->getNodeType() == DOMNode::ELEMENT_NODE)
        {
          const XMLCh* elName = tree->getNodeName();
          const XMLCh* aName = XMLString::transcode("a");
          if (XMLString::equals(elName, aName))
            cout << "Link to " << getHrefAttribute(tree) << endl;
        }
      for (DOMNode* child = tree->getFirstChild();
           child != NULL; child = child->getNextSibling())
        processTree(child);
    }
}
```

Now, this code features an interface that you have never seen before, and a lot of the details are bound to look mysterious.

Nonetheless, if you look at where the `processTree` function calls itself, you can readily tell that this function works by *pre-order traversal*.

. . . .

5 Using Trees for Searching

Search Trees

For large collections of data, our current data structure allow us to do fast searches or fast insertions, but not *both*.

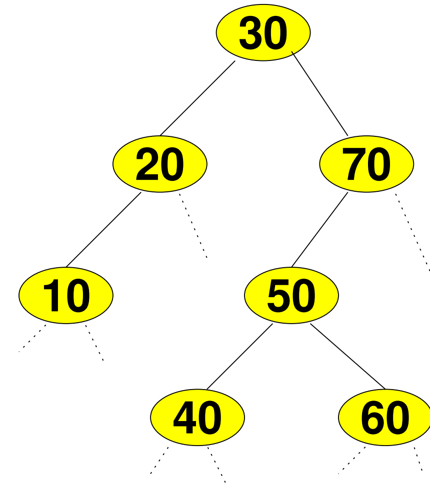
- Arrays and vectors allow fast searching (binary search)
  - But inserting an element requires, on average, moving half the elements.
- Linked lists allow rapid insertions
  - But searches must be sequential
  - Looks at half the elements, on average

Trees offer us a way out of this conflict.

. . . .

Definition: Binary Search Trees

A tree in which every parent has at most 2 children is a *binary tree*.



A binary tree T is a *binary search tree* if for each node n with children T_L and T_R:

- The value in n is greater than the values in every node in T_L.
- The value in n is less than the values in every node in T_R.
- Both T_L and T_R are binary search trees.

. . . .

The Binary Search Tree ADT

Let's look at the basic interface for a binary search tree.

```
template <class T>
class node
{
public:
  node (const T & v,
        node<T> * left,
        node<T> * right)
    : value(v),
      leftChild(left),
      rightChild(right) { }

  T value;
  node<T> * leftChild;
```

```
  node<T> * rightChild;
};
```

- The constructor initializes the data (value) field and the two child pointers.

....

Searching a Binary Tree

```
template <class T>
node<T>* find (const T& element,
               const node<T>* t)
{
 if (t == NULL)
   return NULL;
 if (element < t->value)
   return find(element, t->left);
 else if (t->value < element)
   return find(element, t->right);
 else  // t->value == element
   return t;
}
```

We search a tree by comparing the value we're searching for to the "current" node.

- If the value we want is smaller, we look in the left subtree.
- If the value we want is larger, we look in the right subtree.

....

Demo

Run this algorithm.

....

Inserting into Binary Search Trees

```
template <class T>
void insert (const T& element, node<T>*& t)
{
```

```
if (t == NULL)
  t = new node<T>(element, NULL, NULL);
if (element < t->value)
  insert (element, t->left);
else if (t->value < element)
  insert (element, t->right);
else  // t->value == element
  return; // If we want no duplicates
  // insert (element, t->right); // If we permit duplicates
}
```

- Similar to the search function, we move left and right by comparing the data to the current node *t*
- Only difference is what happens when we reach a pointer that is null

....

Demo

Run this algorithm.

....

5.1 How Fast Are Binary Search Trees?

Trees Can Be Fast

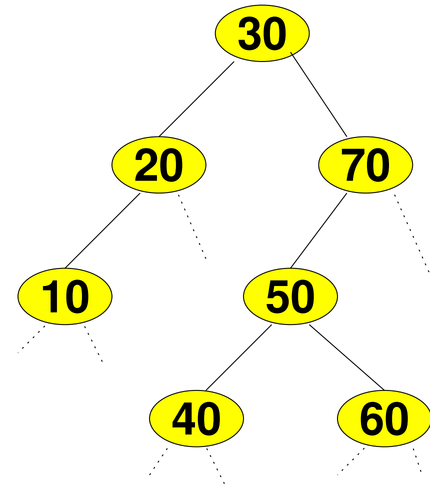
Each step in the BST `insert` and `find` algorithms move one level deeper in the tree.

- The number of recursive calls/loop iterations in all these algorithms is therefore no greater than the height of the tree.
- But how high can a BST be?

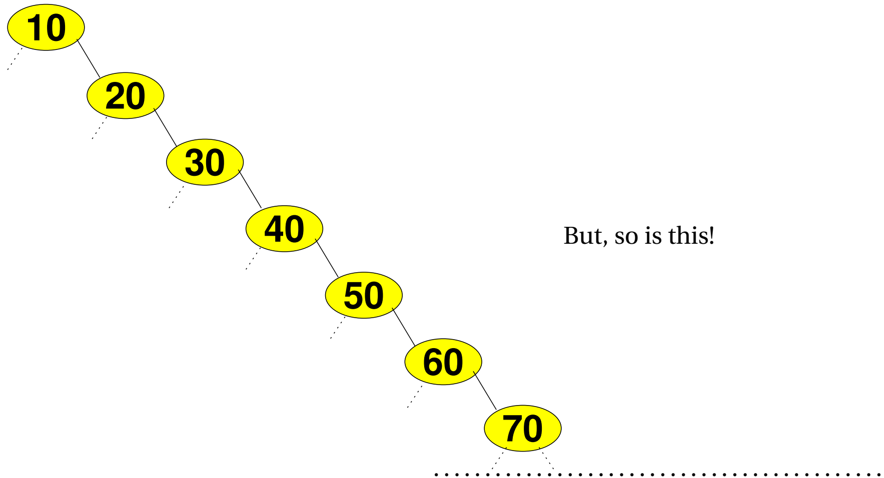
That depends on how well the tree is “balanced”.

....

Shapes of Trees



This is a BST.



But, so is this!

What Determines a Tree's Shape?

The shape of the tree depends upon the order of insertions.

- The worst case is when the data being inserted is already in order (or in reverse order).
  - In that case, the tree *degenerates* into a sorted linked list, as shown earlier.
- The best case is when the tree is *balanced*, meaning that, for each node, the heights of the node's children are nearly the same.
  - Happens (approximately) when data is randomly ordered before being inserted