Trees

- Trees in general
- Binary Trees
- Binary Search Trees
  - As an ADT
  - Potential Problems
- Applications of our new-found way of organizing data

Trees - Definitions

- A tree is a set of nodes connected by directed edges.
- Each node has exactly one in-edge, except one node that is distinguished by having 0 in-edges. This node is the root node.
- For the sake of convenience, the set of nodes is, in some cases, allowed to be empty.
Which of these is a tree?

Which node is the root?
Forests

• A Collection of trees is called a Forest

• Remember that edges are directed: This is a path from A to B, but NOT from B to A:
More Definitions

• A node with no children is called a leaf.

• If there is an edge from node A to node B, then
  • B is a child of A.
  • A is the parent of B.
  • If there is also an edge from A to node C, then B is a sibling of C.

• If there is a path from A to B, then
  • B is a descendant of A.
  • A is an ancestor of B.
  • If the path length is > 0, then a proper descendant or ancestor.

• The length of a path is the number of edges it contains.

• The depth of a tree is the length of the longest path.

• The depth of a node is the length of the path from the root to that node.

• The height of a node is the length of the longest path starting at that node.

• The height of a tree is the height of its root (same as the depth).
Definitions (continued)

In practice, many uses of trees involve nodes with a fixed limit on the number of edges per node (0 through N), although from a theoretical perspective, there are no constraints on the number of edges.

In this case, these trees are often called N-ary trees (unary, binary, etc.)

- Sometimes N is called the branching factor of the tree.
Now some questions...

1. If a tree has N nodes, how many edges does it have? Why? (there are at least two ways to prove this)

   •
   •

2. If an M-ary tree has N nodes, how many "unused" edges (i.e. edges between an existing parent and "non-existent child") does it have?

   •
   •

3. How many nodes can we fit into an M-ary tree of depth D?

   •
Representation of Trees

- M-ary trees: arrays of pointers work well when M is small and if many nodes have > M/2 children (the constant may vary from system to system)

- Linked lists of pointers also work - same sort of trade-off as sparse arrays - trading time for space
Let's zoom in on one of those nodes:

- Each of these linked list cells has a pointer to the next sibling and a child... which are of the same type as this cell. It's a tree!

- Any tree can be represented by a binary tree. Useful!

Q: How many binary nodes do we need to represent a tree containing N nodes?

Don't count the box, do count the circles...
Binary Trees

- For the moment, we’ll focus on **binary trees** - trees whose nodes have 0, 1, or 2 children.

Binary trees are commonly used to represent a relationship between the left and right children of each node, or between the sets of left and right descendants. We’ll explore several such relationships.
Example - "Expression Trees"
for arithmetic expressions

• constants are leaves

• operators are internal nodes whose children are the operands.

• For example: 
\[
( ( 12 + 9 ) / 7 )
\]
Tree Traversals

• A traversal visits every node in the tree once, and usually in a predefined order.

• **Preorder** - process current, go left, go right

• **Postorder** - go left, go right, process current

• **Inorder** - go left, process current, go right

(These ideas can be extended for M-ary trees)
Another way of looking at traversals

Preorder: A B C D E
Inorder: C B D A E
Postorder: C D B E A
Q: Which traversal should we use to print this tree in ordinary mathematical notation?

Q: How can we evaluate this tree, to compute the value of the arithmetic expression it contains? Which traversal(s) should we use?
Binary Search Trees as an ADT

- Operations:
  - \text{value} = \text{FIND}(\text{key})
  - \text{INSERT}(\text{key, value})
  - \text{DELETE}(\text{key})
  - etc., like the other ADTs

- Types:
  - \text{value} - can be of any type; the value is not directly accessed by the ADT operations. The only constraint is that there must be a distinguished value (nil) that \text{FIND} can return to indicate failure.
  - \text{key} - any type that defines transitive less than, equal, and greater than comparisons. (i.e. there is a total order on the set of all possible keys of a specific type.)

- Each of these operations is
  - \(O(\log n)\) "expected"
  - \(O(n)\) "worst case"

We need to do something about the worst case, or performance might be terrible. \textit{We will look at how this can be accomplished next week.}
Binary Search Trees

• So how do we achieve this $O(\log n)$ performance?

• Start with binary trees, but construct them in such a way that the following constraints always hold:
  
  • all left-descendants of node $x$ have a key LESS than $x$’s key
  
  • all right-descendants of node $y$ have a key GREATER than $y$’s key.
  
  • (We will disallow duplicate keys, for the time being)

• Because of this structure, when we compare a key $k$ to the key in node $n$ we can immediately determine:
  
  • whether node $n$ contains $k$
  
  • whether node $n$’s left descendants CANNOT contain $k$
  
  • whether node $n$’s right descendants CANNOT contain $k$

If all goes well, each comparison eliminates half the remaining possibilities.
• We get $O(\log n)$ performance for $\text{FIND}$ and $\text{INSERT}$ results when the tree is balanced - each node has roughly equal numbers of left and right descendants:

Each comparison eliminates more than half of the remaining keys from the search!

But:

If the tree is not “balanced”, then each comparison eliminates only one-- just as bad as a linked list! $O(n)$

Q: So how can we ensure that we can $\text{FIND}$ is $O(\log n)$?
Binary Search Tree Operations

**VALUE = FIND (KEY)**

- Begin at the root node T.
- Loop:
  - If T is nil, return nil
  - If T->key == key, return T
  - If T->key > key then T = T->left
  - Otherwise, T = T->right

**INSERT (KEY, VALUE)**

- Search for the key, in the same manner as FIND, but keep track of the father node F of T as well. If FIND returns non-nil, change value of the returned node to Value. If FIND returns nil, add a new node as the left or right child of F.
- Note that if a new node is added, it is always a leaf node.
- Also note the special case of inserting into an empty tree.
Deletion from a Binary Search Tree

DELETE (KEY)

DELETE is substantially more complicated than FIND or INSERT. The first step is to find the node to remove and its father, similar to what we did in INSERT. What we do next depends on the characteristics of the node that we found:

- If nil (i.e. we couldn’t find the node we wanted to remove), then do nothing.

- If the node is a leaf node, remove it from the tree by removing the pointer from its father.

- If the node has one child, replace the pointer from the father to the node with a pointer from the father to the child. (How can we be sure that this will always work?)

- If the node has two children:
  - Find the inorder successor (the leftmost right descendant) of the node.
  - Move the value and key from that node to the current node.
  - Remove the inorder successor. (Luckily, the inorder successor must be a leaf or have only one child, so removing it is easy.)
The Tennis Ladder Problem

• Given $2^n$ players,
  • How many matches are needed to find the best player?
  • The second best player?
  • The best and worst players?
Tries

We will return to the subject of binary search trees next week, but first we will take some time to study an interesting tree-based data structure with some useful properties.

• A very different data structure - looks like a tree, but behaves quite differently!

• Supports
  • \texttt{v = FIND} (key)
  • \texttt{INSERT} (key, value)
  • \texttt{DELETE} (key)

• All three operations can be performed in $O(L)$ time where $L$ is the length of the key.

• The time required by these operations $O(L)$, regardless of the number of keys in the trie! (if $L$ is bounded, they’re all $O(1)$!)

How is this accomplished?

- Keys must be strings.

  A string is a *finite* sequence of **characters** drawn from a *finite alphabet*.

  An alphabet is a set, along with an operator that can test whether two elements in the set are identical. (there is often an ordering placed on the elements of the set, i.e. alphabetical order, but we will not assume one)

- Each character in a key is represented by an edge in a tree.

- Nodes that represent the end of a key are marked with an **END-OF-KEY** marker.

  ![Diagram](image)

  - To search for a key, start at the root and follow the edge corresponding to each character.
    - If no such edge, fail.
    - If search ends at **END-OF-KEY** node, succeed.
What words (keys) are represented by this trie?

- Note - it is very tempting to "collapse" all of the "at"s together, but this won't work. A more sophisticated representation can allow this, but not a trie. We will study such representations later...
Representing Tries:

de la Briandais Trees and Patricias

• Tries can be represented by M-ary trees (where M is the size of the alphabet). This can potentially waste a gigantic amount of memory, however, as we have seen:

\[
M + (M - 1)(N - 1) \text{ unused edges}
\]

• How can we do better?

  • Tries implemented as lists of lists - like the linked list representation we saw earlier. These are called *de la Briandais* trees.

  • If many of the nodes in a particular path have only one child, we can “compress” the path to a single edge by allowing each edge to represent a string of characters, instead of a single character. This is the approach used by the *patricia* data structure, which is described in the course book.
Algorithms

• Print all of the keys in a trie, in order.
  • What is the big-O of such an algorithm? What is the big-Omega? (how can we even tackle such a problem?)

• Find all the keys in a trie that differ by one character from a given string.