Trees

Josh Wilcox (jw14g24@soton.ac.uk)

February 11, 2025

Contents

1 Trees

1.1 Defining Trees

- One of the major ways of structuring data
- Used in many number of data structures
  - Binary Trees
  - B-Trees
  - Heaps
  - Tries
  - Suffix Trees
- A tree is an **acyclic undirected graph**
- Trees are often **ordered**
  - One node can be seen as the *root* node
  - Nodes have children nodes directly beneath them
  - All nodes have parents except the root node
  - Nodes with no children are *leaf nodes*
- We can think of a tree as being made up of **subtrees**

1.2 Level of nodes

- It is useful to lavel different levels of the tree
- The level of a node is **its distance from the root**
- The height of the tree is the **number of levels**

2 Binary Trees

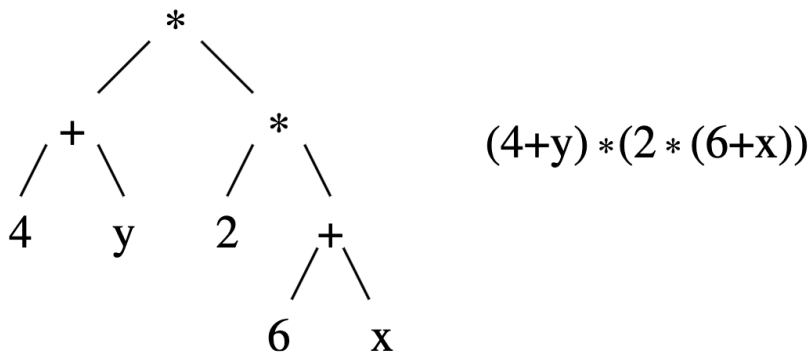
2.1 Definition

- A binary tree is a tree such that each nde has zero, one, or two children
- The greatest number of nodes at level l is 2^l
- The greatest number of nodes of a tree of height h is:

$$\sum_{i=0}^{h-1} 2^i = 2^h - 1$$

2.2 Uses

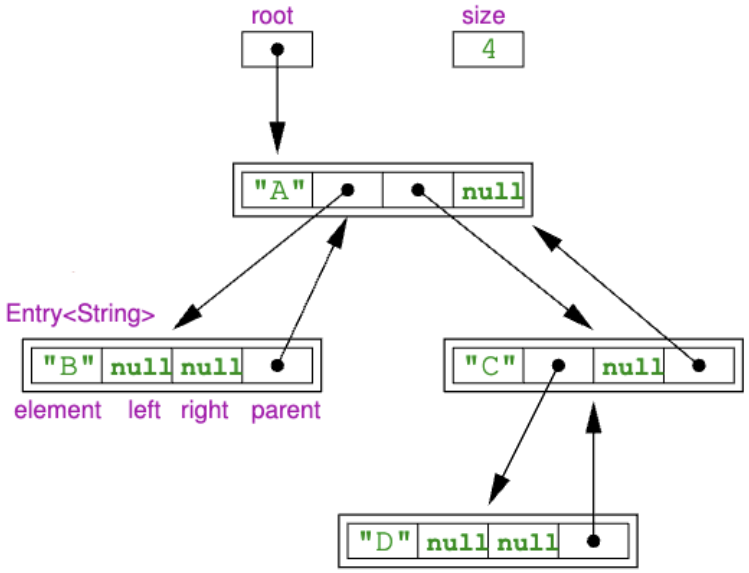
- They can be used as *expression trees*
  - Like how we used trees to represent logic equations in Mathematics I



2.3 Java Implentation

- As they are linked structures, they have a similar setup to linked lists

```
1  public class Node<T>
2  {
3      private T element;
4      private Node<T> left = null;
5      private Node<T> right = null;
6      private Node<T> parent;
7      private Node(T element, Node<T> parent)
8      {
9          this.element = element;
10         this.parent = parent;
11     }
12 }
13
14 public class BinaryTree<E>
15 {
16     private Node<E> root;
17     private int size;
18     ...
19 }
```



Binary Tree

3 Binary Search Trees

3.1 Definition

Defined Recursively:

- Each element in the left subtree is less than the root element
- Each element in the right subtree is greater than the root element
- Both left and right subtrees are binary search trees
- **Base Case** - No Children

3.2 Searching a binary search tree

3.2.1 The Process

- Start at the root
- Compare with element at root
  - If less, go left
  - If more, go right
  - If equal, element found

3.2.2 Speed of Search

- Number of comparisons depends on the level of the node in the tree
- The worst case number of comparisons is the height of the tree
  - This depends on the density of the tree

3.3 Printing out a tree using Java Recursion

```
1 // Recursive Function!
2 public void print(Node<E> e)
```

```
{
    if (e != null)
    {
        print(e.left); // Recursive call (print is not a function)
        System.out.println(e.element);
        print(e.right);
    }
}
```

3.4 Printing out a tree without Java Recursion

```
// Initialize the current node as the root of the tree.
Node<E> e = root;

// Create an empty stack to keep track of nodes for backtracking.
Stack s = new Stack();

// Iterate until there are no more nodes to process via traversal or backtracking.
while(e != null || s.hasElement()){
    // If there is a current node, traverse to its left child.
    if (e != null){
        s.push(e);  // Push the current node onto the stack before moving left.
        e = e.left; // Move to the left child of the current node.
    }
    else{
        // If reaching a null, backtrack to the nearest node that hasn't been fully
        ↪ processed.
        e = s.pop();  // Pop a node from the stack to process it.
        System.out.println(e.element); // Process the current node (e.g., printing
        ↪ its element).
        e = e.right; // Shift to the right child of the node just processed.
    }
}
```

3.5 Using binary trees to implement sets

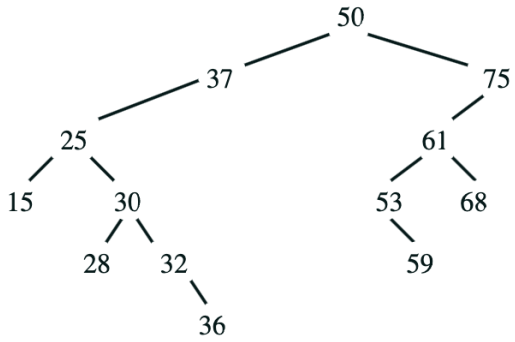
- As there is no order in the sets, we can use a binary search tree to sort them in order which allows for efficiencies
- Allows for rapid search which is a feature we care about
- Binary trees are one of the most efficient ways of implementing
  - Especially if you care about the order of the set at all

4 Tree Iterators

- To iterate through elements, we start from the left-most node, which is the smallest element in the tree.
- The process is as follows:

1. Start at the left most branch (the smallest element)
2. If a right child exists:
   - Move right down once, then move as far left as possible
3. Else, go up to the left as far as possible then move up right once to find the next

- This in-order traversal method ensures that each node is visited in ascending order.



$$\{15\ 25\ 28\ 30\ 32\ 36\ 37\ 50\ 53\ 59\ 61\ 68\ 75\}$$

5 Deletion from Binary Search Trees

5.1 Removing Elements with at most one children in Java

```java
// Case 1: Delete a leaf node (node with no children)
if (e.left == null && e.right == null) {
    // Check whether 'e' is the left child of its parent
    if (e == e.parent.left)
        e.parent.left = null; // Remove the reference from the parent
    else
        e.parent.right = null; // Otherwise remove from the right
}
// Case 2: Delete node with only a left child
else if (e.right == null) {
    // Replace node 'e' with its left child by linking the parent directly
    if (e == e.parent.left)
        e.parent.left = e.left; // Update the parent's left pointer
    else
        e.parent.right = e.left; // Update the parent's right pointer
    e.left.parent = e.parent; // Set the left child's parent to skip 'e'
}
// Case 3: Delete node with only a right child
else if (e.left == null) {
    // Replace node 'e' with its right child by linking the parent directly
    if (e == e.parent.left)
        e.parent.left = e.right; // Update the parent's left pointer
    else
        e.parent.right = e.right; // Update the parent's right pointer
```

```
    e.right.parent = e.parent; // Set the right child's parent to skip 'e'
}
```

5.2 Logic of removing elements with two children

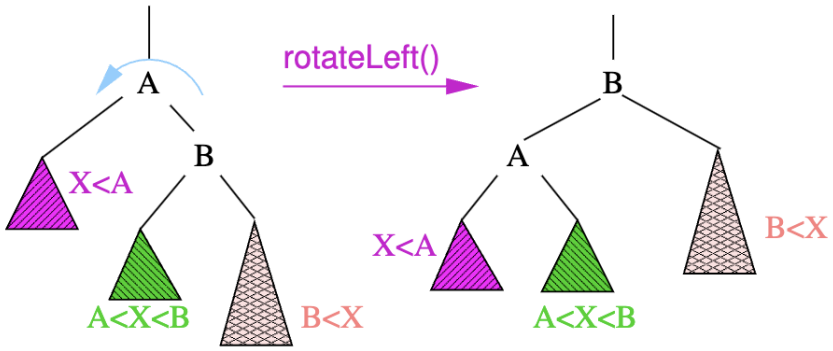
- Replace the element we want to delete with the value that is **next higher**
- Then remove this element from the tree
  - This can become recursive if the successor also has two trees

6 Balancing Trees

6.1 Rotation

6.1.1 Rotation Logic

- We can modify the shape of a tree to make it more balanced by **rotating** elements
- This works by changing the root node



6.1.2 Java Rotation

```
// This function performs a left rotation at node 'e' in a binary tree.
// It rearranges the subtree such that 'e's right child becomes the new root
// of that subtree, and 'e' becomes the left child of its former right child.
// This rotation handles cases where the node to be rotated has 0, 1, or 2
↪  children.
void rotateLeft(Node<T> e){
    // Identify the right child, which will become the new root of this subtree.
    Node<T> r = e.right;

    // Set e's right child to r's left subtree.
    // This step works even if r has 0 or 1 child:
    // - If r has no left child, e.right becomes null.
    // - If r has a left child, that subtree is preserved.
    e.right = r.left;

    // If r's left child exists, update its parent pointer to 'e'.
    // This ensures the subtree of r's left child remains linked correctly.
    if (r.left != null)
        r.left.parent = e;
```

```
19
20      // Link r's parent to be the same as e's parent.
21      r.parent = e.parent;
22      // If 'e' is the root (has no parent), update the root pointer to 'r'.
23      if (e.parent == null)
24          root = r;
25      // Otherwise, update e's parent's child reference from e to r.
26      else if (e.parent.left == e)
27          e.parent.left = r;
28      else
29          e.parent.right = r;
30
31      // Place 'e' as the left child of 'r'.
32      // This repositions 'e' in the tree, completing the rotation.
33      r.left = e;
34      // Update 'e's parent pointer to reflect its new parent, 'r'.
35      e.parent = r;
36  }
```

6.2 Double Rotation

- If the unbalance tree is on the inside, we need a double rotation.
- This is equivalent to performing a left rotation first, followed by a right rotation.

7 AVL Trees

An AVL tree is a binary search tree where which

- The heights of the left and right subtree differ by most 1
- The left and right subtrees are each AVL trees
- This guarantees the worst case AVL tree has **Logarithmic Depth**

7.1 Minimum Number of Nodes for height h

- Let $m(h)$ be the minimum number of nodes in an AVL tree of height h
- The tree has to be made up of a root and two subtrees
  - One of the subtrees has height $h-1$
  - In the worst case, the other tree has height $h-2$ as they can only differ by 1

$$m(h) = m(h-1) + m(h-2) + 1$$

  - $m(1) = 1, m(2) = 2$

7.1.1 Proof of Logarithmic Depth

$$m(h) = m(h-1) + m(h-2) + 1$$

$m(1) = 1, m(2) = 2$ Prove by induction $m(h) \geq (\frac{3}{2})^{h-1} \quad \forall h \geq 1$

Basis Step: Show $m(h) \geq (\frac{3}{2})^{h-1} \quad \forall 1 \leq k \leq 2$

$$m(1) = 1 \leq (\frac{3}{2})^0, \qquad m(2) = 2 \leq (\frac{3}{2})^1$$

Assumption Step: Assume $m(h) \geq (\frac{3}{2})^{h-1}$ for $h = k \quad k \geq 4$ and $h = k - 1$

Inductive Step: Use the assumption to show $m(h) \geq (\frac{3}{2})^{h-1}$ for $h = k + 1$

$$m(k+1) = m(k) + m(k-1) + 1$$

We know that $m(k) \geq m(k-1)$

$$m(k+1) \geq (\frac{3}{2})^{k-1} + (\frac{3}{2})^{k-2} + 1$$

$$m(k+1) \geq (\frac{3}{2})^{k-1}(1 + \frac{3}{2} + (\frac{3}{2})^{1-k}) \geq (\frac{3}{2})^{k-1} \cdot \frac{5}{2}$$

$$m(k+1) \geq (\frac{3}{2})^{k-1} \cdot \frac{10}{4} \geq (\frac{3}{2})^{k-1} \cdot \frac{9}{4} \geq (\frac{3}{2})^{k+1}$$

If we assume that $m(h) \geq (\frac{3}{2})^{h-1}$ for $h = k \quad k \geq 4$ and $h = k - 1$, we can show that $m(h) \geq (\frac{3}{2})^{h-1}$ for $h = k + 1$. As we have shown that $m(h) \geq (\frac{3}{2})^{h-1}$ for $h = 1$ and $h = 2$, we have shown that $m(h) \geq (\frac{3}{2})^{h-1} \quad \forall h \geq 1$ by mathematical induction

Let n be the number of elements of the AVL tree Use this proof to show that h is $\mathcal{O}(log(n))$

$$n \geq m(h) \; \forall h$$

$$\therefore n \geq (\frac{3}{2})^{h-1} \; \forall h$$

$$log(n) \geq log((\frac{3}{2})^{h-1})$$

$$log(n) \geq (h-1)log(\frac{3}{2})$$

$$h \leq \frac{log(n)}{log(\frac{3}{2})} + 1$$

h is $\mathcal{O}(log(n))$

8 Implementing AVL Trees

- To implement an AVL tree, we include additional information at reach node indicating the **balance** of the **subtrees** where:

  `balanceFactor = (height of left subtree) - (height of right subtree)`

- To add an element:
  - Find the location where it is to be inserted and insert
  - Iterate up through the parents, readjusting the balanceFactor
  - If the balance factor exceeds ± 1 then re=balance the tree then stop
  - Else if the balance factor goes to zero then stop