1 Combinatorics

Problem 1. Give a formula for each of the following quantities:

- The number of leaves in a perfect tree of height \( h \).
- The number of nodes in a perfect tree of height \( h \).
- The number of internal nodes in a perfect tree of height \( h \).
- The number of nodes in a perfect tree with \( l \) leaves (assume \( l \) is valid).

Solution 1.

- \( 2^h \)
- \( 2^{h+1} - 1 \)
- \( 2^{h+1} - 1 - 2^h = 2(2^h) - 2^h - 1 = 2^h - 1 \)
- \( 2^{\log_2 l + 1} - 1 = 2l - 1 \)

Problem 2. Give a formula for each of the following quantities:

- The maximum number of nodes in a tree of height \( h \).
- The minimum number of nodes in a tree of height \( h \).
- The minimum number of nodes in a balanced tree of height \( h \).
- The maximum number of nodes in an unbalanced tree of height \( h \).

Solution 2.

- \( 2^{h+1} - 1 \)
- \( h + 1 \)
- \( 2^{(h-1)+1} - 1 + 1 = 2^h \)
- \( 2^{(h-1)+1} - 1 = 2^h - 1 \)
2 BST Structure

Problem 3. Draw a BST containing the elements 10, 3, 6, 5. If more than one such BST exists, draw a second with 10 as the root.

Solution 3.

Problem 4. Draw a balanced BST containing the set of elements \( S = \{5, 8, 3, 2, 1, 9\} \). For which elements \( x \in S \) does there exist a balanced BST with \( x \) as root containing the elements of \( S \)?

Solution 4. Balanced BSTs containing the elements of \( S \) exist with the root \( x = 3, 5, 8 \).

3 BST Algorithm Examples

Problem 5. Give the sequence of nodes visited when searching for the following elements:

- 2
- 10
- 7
- 14
Solution 5.

- Searching for 2: 22, 7, 3, 1.
- Searching for 7: 22, 7.
- Searching for 12: 22, 7, 13, 16.
- Searching for 30: 22, 7, 13, 16, 18.

**Problem 6.** See Figure 4. Give a sequence of integers such that inserting the elements of the sequence into an empty BST one-at-a-time (in the given order) results in a BST with the shape seen the left portion of Figure 4. Do the same for the shape in the right portion of Figure 4.

Solution 6.

- Left: 3, 2, 1, 5, 4, 6.
- Right: 2, 1, 6, 5, 4.

4 Algorithm Implementation

**Problem 7.** Implement a C++ function that returns the smallest value in a non-empty binary search tree. Assume each Node object has an int instance variable named v storing the value at the node.
Solution 7.

```cpp
int smallest(Node* root)
{
    if (root->left == nullptr)
        return root->v;
    return smallest(root->left);
}
```

**Problem 8.** Implement the same function as in Problem 7 but for general binary trees.

**Solution 8.**

```cpp
int smallest(Node* root)
{
    int ls = root->v;
    int rs = root->v;

    if (root->left != nullptr)
        ls = smallest(root->left);
    if (root->right != nullptr)
        rs = smallest(root->right);

    if (ls <= root->v && ls <= rs)
        return ls;
    if (rs <= root->v && rs <= ls)
        return rs;
    return root->v;
}
```

**Problem 9.** Implement a C++ function that returns the longest string in a binary tree that starts with a given prefix string. If no such string exists, the function should return the empty string.
Solution 9.

string longest(string prefix, Node* root)
{
    if (root == nullptr)
        return "";

    string sl = longest(prefix, root->left);
    string sr = longest(prefix, root->right);

    string r = "";
    if (root->s.find(prefix) == 0)
        r = root;
    if (sl.size() >= sr.size() && sl.size() >= r.size())
        return sl;
    if (sr.size() >= sl.size() && sr.size() >= r.size())
        return sr;
    return r;
}

5 Algorithm Design

Problem 10. Design an $O(h)$-time algorithm that finds the second largest element of a binary search tree of height $h$.

Solution 10. Algorithm:

1. Traverse the rightmost path in the tree, keeping track of the parent of the current node.
2. When the current node $v$ has no right child, check whether $v$ has a left child.
3. If $v$ does not have a left child, return the parent of $v$.
4. Else, traverse the rightmost path in the left subtree of $v$.
5. When the current node $v$ has no right child, return $v$.

Running time: The algorithm runs in $O(h)$ time, since for each depth $0, 1, \ldots, h$, only one node is visited and $\Theta(1)$ is spent.

Correctness: The algorithm relies on two observations. First, that if a node $v$ on the rightmost path has no children, then $v$ is the largest node in the BST and $v$’s parent is the second-largest (since there are no values between $v$ and $v$’s parent). Second, if a node $v$ on the rightmost path has only a left child, then $v$ is the largest node in the BST and the largest node in the left subtree of $v$ is the second-largest (since this is the largest value in the BST other than $v$).
Problem 11. Design a $O(n)$-time algorithm that determines whether a BST $T$ of size $n$ contains two floats $a$ and $b$ such that $a + b = 0$.

Solution 11. Algorithm:

1. Search $T$ for 0. If found, return $\text{true}$.
2. Traverse the nodes of $T$ in reverse order and store all negative numbers seen in a list $L^-$. 
3. Traverse all nodes of $T$ in order and store all positive numbers seen in a list $L^+$. 
4. Negate the elements of $L^-$. 
5. Merge $L^-$ and $L^+$ into a single sorted list $L^{\text{all}}$ of positive numbers. 
6. Scan $L^{\text{all}}$ for duplicates. If any are found, return $\text{true}$. 
7. Return $\text{false}$.

Running time: Each step of the algorithm takes $O(n)$ time, since searching (Step 1), traversal (Steps 2-3), creating lists of $\leq n$ elements (Steps 2-3), modifying every value in a list of $\leq n$ elements (Step 4), merging two lists of $\leq n$ elements each (Step 5), comparing all adjacent pairs of elements in a list of $\leq n$ elements (Step 6), and returning a value each take $O(n)$ time.

Correctness: There are two such elements if and only if a number $x$ and its negation $-x$ are both in $T$. If this is the case, then the negated values of $L^-$ and $L^+$ will both contain $|x|$. Since $|x| = |x|$, any such pairs will appear adjacent in the merged list, and so will be found when scanned in Step 6.