**COMPSCI 330: Design and Analysis of Algorithms** 10/9/2014

Lecture #13

*Lecturer: Debmalya Panigrahi* *Scribe: Samuel Haney*

# 1 Overview

In this lecture, we will discuss a class of data structures called binary search trees.

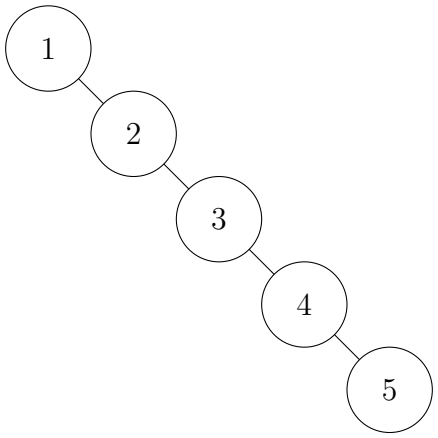
# 2 Motivation

As you have seen, binary search is a valuable tool for locating values in a group. We are given a sorted group of numbers, and we should compare the key against the median and then either go left or right in the group. How should we store this group of numbers to facilitate binary search?

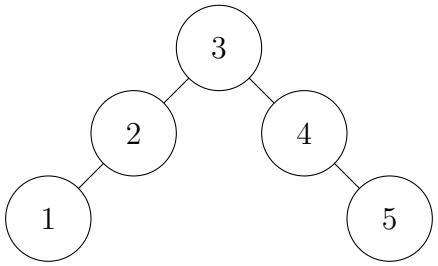
One option is to use an array. Given an array of sorted numbers, we can index into the array to find the median. However, arrays don't work very well if we want to change the list of numbers: insertion into the array takes $O(n)$ time. This problem is solved by using a linked list, but then there is no way to find the median! We construct a structure that always allows us to find the median at each step:

1. In the first step, we need access to the median, $A_{k/2}$, so we add a pointer to it.
2. In the next step, depending on which direction we went, we will need access to $A_{k/4}$ or $A_{3k/4}$. So, we add pointers from $A_{k/2}$ to both of these elements.
3. We continue like this, and obtain a binary tree.

If we can maintain this tree, the runtime of a search will be $O(\log n)$. However, the trick is keeping the tree balanced. If we add an element to the tree, the initial pointer we added no longer points to the median of the group of numbers. In fact, if we add the numbers $1,2,3,4,5$ sequentially, we obtain the following tree:



This is just a linked list! Clearly, the search time will not be $O(\log n)$. Instead, we want the following structure:




A *balanced* tree is one in which subtrees at all levels have (nearly) the same height. Our goal is to develop a method that will maintain a balanced tree over insertions and deletions into the tree.

# 3 AVL Trees

**Definition 1.** *An AVL tree is a tree satisfying the following: for any intermediate nodes, the difference in the heights of the left and right subtrees is at most 1.*

In this section, we show how maintaining this property guarantees $O(\log n)$ runtime. We will not go into depth on how to maintain the AVL property.

**Lemma 1.** *An AVL tree containing n nodes has height $O(\log n)$.*

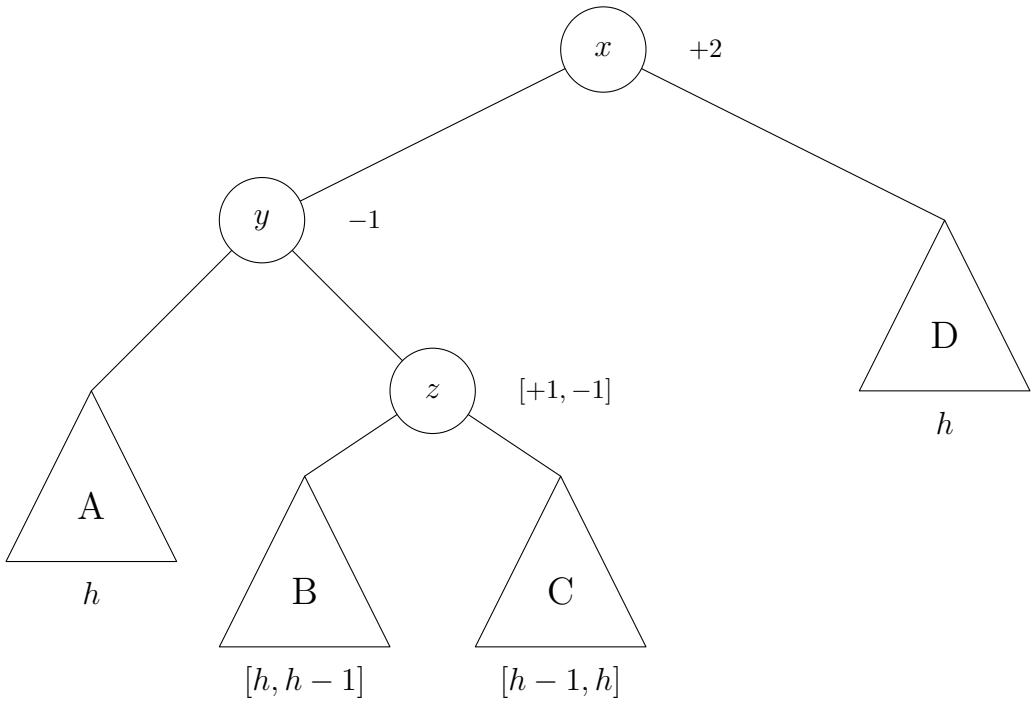
First, note that Lemma 1 is sufficient to maintain an $O(\log n)$ runtime since only one comparison is made in each level of the search. We now give a proof of the lemma:

*Proof.* We prove an equivalent property, that an AVL tree of height $h$ has some $a^h$ nodes where $a$ is a constant (these are equivalent since $n = a^h \implies h = \log_a n$). Assume the height of the tree is $h$ and the heights of the left and right subtrees are $h_l$ and $h_r$. At least one subtree must have height $h-1$. That is, $\max\{h_l, h_r\} = h-1$ (this is not a property of AVL trees, but of all trees). Next, we have $\min\{h_l, h_r\} \geq h-2$ from the balance property of AVL trees. Let $S_h$ be the minimum number of nodes in an AVL tree of height $h$. Given that at most one of the subtrees can have height less than $h-1$, we have

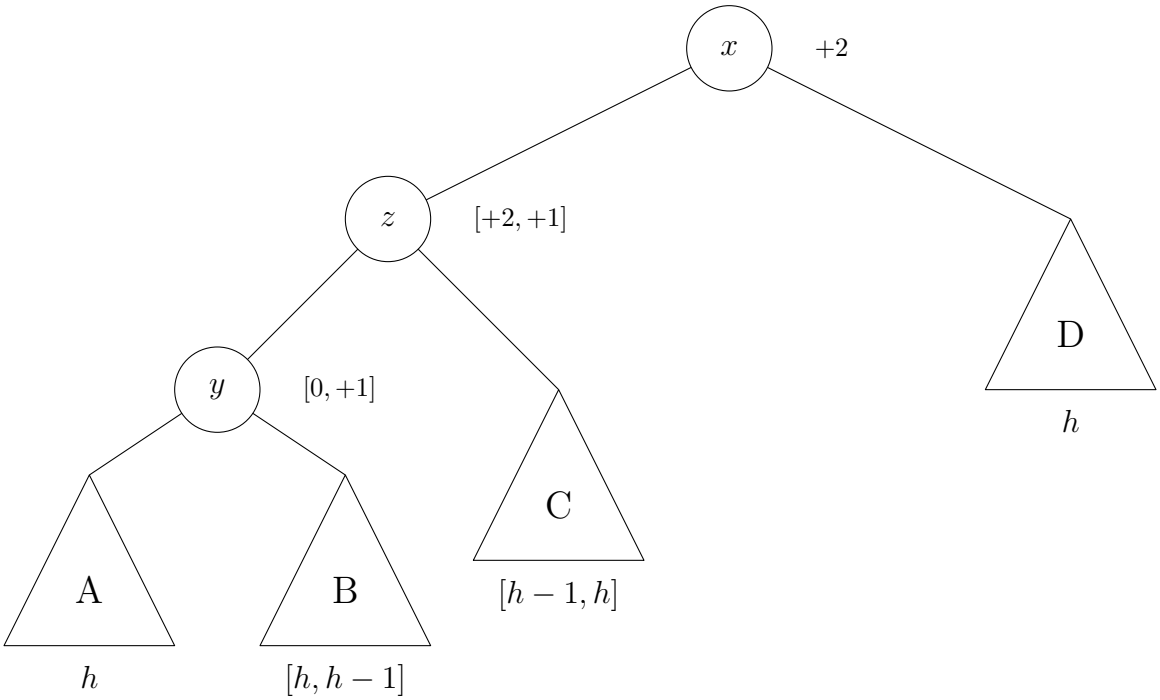
$$S_h = S_{h-1} + S_{h-2} + 1. \tag{1}$$

To complete the proof, note that $S_h \geq F_h$, where $F_h$ is the $h$th Fibonacci number. As we have seen, $F_h = a^h$ for some $a > 1$. □

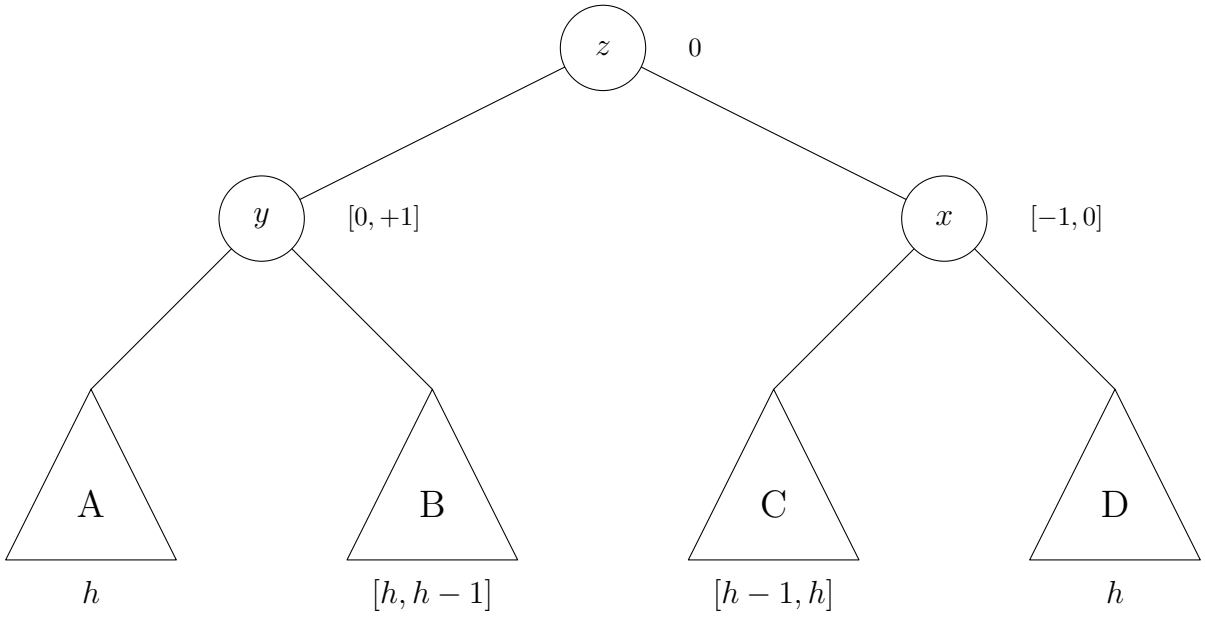
We won't discuss deletions from an AVL tree, but we will briefly discuss insertions. How can we insert into an AVL tree and maintain the AVL property? We use a tree modification called *roations*. The number of rotations per insertion is constant. Consider the following tree, which violates the AVL property:




Here, the number next to each vertex (which we will call the *imbalance value*) denotes the difference in heights of its subtrees. For example, the $+2$ here indicates that the height of $x$'s left subtree is two more than the height of $x$'s right subtree. The value below each subtree is the height of the subtree. We use the notation $[h,k]$ so denote the value of the subtree as either $h$ or $k$. Next, we perform a left rotation which results in the following tree:



This tree is still a binary search tree since we maintain the property that $y < B < z$. The imbalance value of $x$ has remained unchanged. We correct all the imbalance values with a right rotation:




Each imbalance value has magnitude at most 1, so we now have the AVL property.

# 4 Red Black Trees

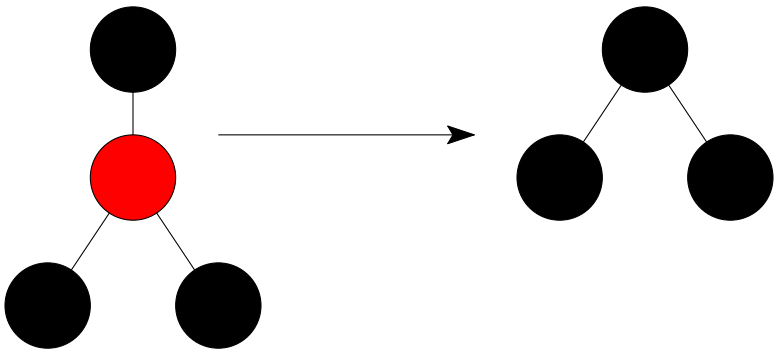
**Definition 2.** *A red black tree is a tree which satisfies the following properties:*

1. *The root is black.*
2. *The leaves are black.*
3. *Both children of a red node are black.*
4. *For any node in the tree, paths to two different leaves have the same number of black nodes.*

We first prove the following lemma, which shows that the search time in a red black tree is $O(\log n)$.

**Lemma 2.** *A red black tree of height h contains at least* $2^{h/2}$ *nodes.*

What is the lower bound on the number of black nodes in the tree? We obtain an all black tree by removing each red node and connecting its children to its parent:



Note that this new tree need not be binary.

**Claim 1.** *After removing red nodes, the height of the new tree is at least $h/2$.*

*Proof.* Some root to leaf path in the original tree had length $h$, and at most $h/2$ of the nodes on this path were red. □

In this new tree, paths from any node to a leaf contain only black nodes. Therefore, by property (4) from Definition 2, paths from some node to two different leaves must be the same length. As before, let $S_{h/2}$ be the number of nodes in a tree of height $h/2$ (the height of our tree). Then because our tree is completely balanced, we get

$$S_{h/2} \geq 2S_{h/2-1} + 1. \tag{2}$$

Solving this recurrence gives

$$S_{h/2} \geq 2^{h/2},$$

which completes the proof of Lemma 2. Lemma 2 shows that maintaining a red black tree allows for $O(\log n)$ time searches. We will not show how to maintain the red black properties over insertions and deletions.

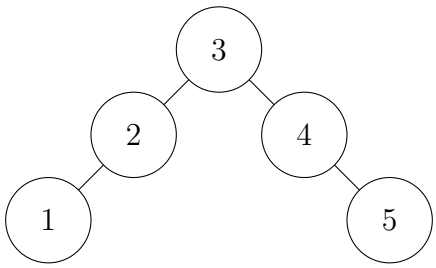
# 5 Optimal BSTs

We now investigate a slightly more general problem. Let $a_1, \ldots, a_n$ be a set of values, and $p_1, \ldots, p_n$ be a set of probabilities where $p_i$ corresponds to $a_i$. $p_i$ gives us the frequency that element $i$ is searched for. The sum of all the probabilities in the set should add up to 1. We want to optimize the average case search time for a key. Formally, we want to minimize

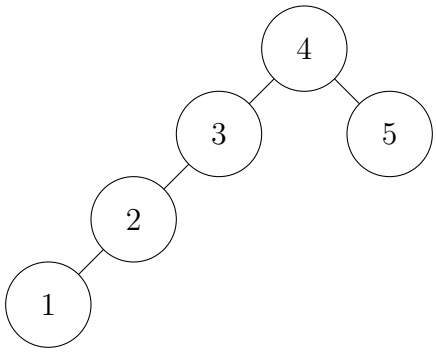
$$\sum_{i=1}^{n} p_i \cdot h(a_i), \tag{3}$$

where $h(a_i)$ is the depth of $a_i$ (we assume the root is at depth 1). Intuitively, elements with higher frequencies should be closer to the top of the tree, since they will be accessed more often.

**Example .** Assume the values $1,2,3,4,5$, have weights $0.05, 0.05, 0.3, 0.2, 0.3$. Without frequencies, the optimal binary search tree would be the one we saw before:




However, with frequencies this is no longer optimal. 3, 4, and 5 have higher frequencies, and should be at the top. The following is the search tree which minimizes Equation 3.




The element with the highest frequency is not necessarily the root of tree. In this case, 3 and 5 have the highest frequencies, and neither is the root.

To compute this optimal BST, we can use dynamic programming. At a high level, we want to make the frequencies similar in each subtree. We give the following recursive algorithm to compute the cost of the minimum BST. This algorithm can be modified to also return the tree itself.

**Algorithm 1** Computing the cost of the optimal BST

```
1: function OPT-BST(a_1, ..., a_n, p_1, ..., p_n)
2:     T_min ← ∞
3:     for i ← 1 to n do
4:         c ← OPT-BST(a_1, ..., a_{i-1}, p_1, ..., p_{i-1})
5:         d ← OPT-BST(a_{i+1}, ..., a_n, p_{i+1}, ..., p_n)
6:         T ← c + d + Σ_{j=1}^{n} p_j
7:         if T < T_min then
8:             T_min ← T
9:     return T_min
```

Each recursive call is to a contiguous sequence of elements in the list. There are $O(n^2)$ such sequences. If we make each of these a subproblem, there are $O(n^2)$ subproblems. The value of each of these sequences takes $O(n)$ time to compute, as can be seen in Algorithm 1. Therefore, a dynamic programming version of Algorithm 1 takes $O(n^3)$ time.