Lecture 9: AVL-tree 1
CSC2100 Data Structure

Yufei Tao
CSE department, CUHK

March 17, 2011
In this and the next lectures, we will learn a structure called the **AVL-tree**, which also uses the binary tree as its backbone. This is a powerful structure that supports a large number of operations on an ordered set efficiently.

“**AVL**” was named after the inventors of the structure.
1 Preliminaries
   - Problem
   - Overview

2 AVL-tree
   - Basic concepts
   - Formal description

3 Searching the AVL-tree

4 Analysis
   - Results
   - Height analysis
Problem statement

Problem (Set management by keys)

Let $S$ be a set of $n$ items, each of which has a distinct integer key. We want to support the following operations efficiently:

- **Insert**: Add an item to $S$ with a new key.
- **Delete**: Remove an existing item from $S$ with key $k$.
- **Point search**: Return the item in $S$ with key $k$.
- **Successor**: Given the key $k$ of an item in $S$, return the item in $S$ with the smallest key greater than $k$. 
We are already familiar with insertion, deletion, and point search. Next, let us see some examples about *Successor*.

Let $S = \{10, 35, 40, 53, 60, 79, 81, 88\}$.
- Given $k = 40$, *Successor* returns 53.
- Given $k = 88$, *Successor* returns $\emptyset$. 
Recall that a hash structure with $m$ buckets allows us to support each insertion, deletion and point search in $O(n/m)$ time in expectation.

How about finding successors?
An AVL-tree supports all four operations in $O(\log n)$ time.
Subtrees

Definition (Left and right subtrees)

The *left subtree* of a node $u$ is the binary tree that is rooted at the left child $u_{\text{left}}$ of $u$, and includes all the nodes under $u_{\text{left}}$. If $u$ does not have a left child, its left subtree is empty.

The *right subtree* of $u$ is defined analogously.

Example

- The left subtree of $u_1$ has root $u_2$, and includes all the grey nodes.
- The right subtree of $u_1$ has root $u_3$, and includes all the black nodes.
- The right subtree of $u_2$ is empty.
As before, the height of a binary tree equals the maximum level of the leaf nodes. As a special case, let us define the height of an empty tree to be $-1$.

**Example**

- The left subtree of $u_1$ has height 2.
- The right subtree of $u_3$ has height 0.
- The right subtree of $u_2$ has height $-1$. 
Balance

Definition (Balanced binary tree)

A *balanced binary tree* is a binary tree where every node $u$ satisfies the property that the left and right subtrees of $u$ differ in height by at most 1.

Example

The left tree is balanced, but the right one is not (the black node violates the above property).
**Binary search tree**

**Definition (Binary search tree)**

A *binary search tree* on a set $K$ of $n$ distinct keys is a binary tree with $n$ nodes such that:

- Each node stores a different key in $K$.
- The key of any node $u$ is *greater* than all the keys stored in the left subtree of $u$.
- The key of any node $u$ is *smaller* than all the keys stored in the right subtree of $u$.

Space complexity $O(n)$. 

![Binary search tree example diagram](image)
The AVL-tree

Definition (AVL-tree)

An **AVL-tree** is a balanced binary search tree.

Example

Only the right tree is an AVL-tree (the left tree is not balanced).
Point search

Example

Assume that we want to find the node with key \( k = 60 \).

- Start from the root \( u \) (with key 40).
- Since \( k > 40 \), \( k \) (if it exists) must be in the right subtree of \( u \). Hence, we access the right child \( u' \) (with key 73) of \( u \).
- Since \( k < 73 \), \( k \) (if it exists) must be in the left subtree of \( u' \). Hence, we access the left child \( u'' \) of \( u' \).
- \( u'' \) is exactly what we are looking for.
Algorithm *PointSearch*(k)

/* find the node with key k */

1. \( u = \) the root
2. while \( u \) is not NULL
3. \( \text{if } u.key = k \) return \( u \)
4. \( \text{if } u.key < k \)
5. \( u = \) the right child of \( u \)
6. \( \text{else} \)
7. \( u = \) the left child of \( u \)
8. return \( \emptyset \)
Design an efficient algorithm for the *Successor* operator. We will discuss it in the tutorial.
**Time complexities**

- *PointSearch* accesses at most a single root-to-leaf path. Hence, it takes $O(h)$ time, where $h$ is the height of the AVL-tree.

- In the tutorial, we will see that *Successor* can also be supported in the same amount of time.

The next few slides show that $h = O(\log n)$, where $n$ is the number of nodes.
Height analysis

Height of a balanced binary tree

A balanced binary tree with \( n \) nodes has height \( O(\log n) \).

**Proof.**

Denote the height as \( h \). We will show that a balanced binary tree with height \( h \) must have \( \Omega(2^{h/2}) \) nodes.

Once this is done, it follows that there is a constant \( C > 0 \) such that:

\[
\begin{align*}
  n & \geq C \cdot 2^{h/2} \Rightarrow \\
  2^{h/2} & \leq n/C \Rightarrow \\
  h/2 & \leq \log_2(n/C) \Rightarrow \\
  h & = O(\log n)
\end{align*}
\]
Proof. (cont.)

Let $f(h)$ be the minimum number of nodes in a balanced binary tree with height $h$. It is clear that:

$$f(0) = 1$$
$$f(1) = 2$$
Proof. (cont.)

In general, for \( h \geq 2 \):

\[
    f(h) = 1 + f(h - 1) + f(h - 2)
\]
Height analysis

Height of a balanced binary tree

**Proof. (cont.)**

When $h$ is an even number:

\[
    f(h) = 1 + f(h - 1) + f(h - 2)
\]

\[
    > 2 \cdot f(h - 2)
\]

\[
    > 2^2 \cdot f(h - 4)
\]

...  

\[
    > 2^{h/2} \cdot f(0)
\]

\[
    = 2^{h/2}
\]
Proof. (cont.)

When $h$ an odd number (i.e., $h \geq 3$):

\[
\begin{align*}
f(h) & > f(h - 1) \\
& > 2^{(h - 1)/2} \\
& = 2^{h/2} / \sqrt{2} \\
& = \Omega(2^{h/2})
\end{align*}
\]
Playback of this lecture:

- AVL-tree.
- Space $O(n)$.
- Point search in $O(\log n)$ time.
- Successor search in $O(\log n)$ time.

In the next lecture, we will discuss the insertion and deletion algorithms.