Dictionary ADT

A dictionary is a collection of items, each of which contains
- a key
- some data,
and is called a key-value pair (KVP). Keys can be compared and are (typically) unique.

Operations:
- search($k$)
- insert($k, v$)
- delete($k$)
- optional: join, isEmpty, size, etc.

Elementary Implementations

Common assumptions:
- Dictionary has $n$ KVPs
- Each KVP uses constant space
- Comparing keys takes constant time

**Unordered array or linked list**

- search $\Theta(n)$
- insert $\Theta(1)$
- delete $\Theta(n)$ (need to search)

**Ordered array**

- search $\Theta(\log n)$
- insert $\Theta(n)$
- delete $\Theta(n)$
Binary Search Trees (review)

Structure
A BST is either empty or contains a KVP, left child BST, and right child BST.

Ordering
Every key \( k \) in \( T.\text{left} \) is less than the root key.
Every key \( k \) in \( T.\text{right} \) is greater than the root key.

```
  15
 /   \
6    25
 /     \
10     29
  /   \
 8   23   27
    /    \
   14  27  50
```

BST Search and Insert

- \( \text{search}(k) \) Compare \( k \) to current node, stop if found, else recurse on subtree unless it’s empty
- \( \text{insert}(k, v) \) Search for \( k \), then insert \((k, v)\) as new node

Example:

```
  15
 /   \
6    25
 /     \
10     29
  /   \
 8   23   27
    /    \
   14  27  50
```

BST Delete

- If node is a leaf, just delete it.
- If node has one child, move child up
- Else, swap with successor or predecessor node and then delete
Height of a BST

search, insert, delete all have cost $\Theta(h)$, where
$h =$ height of the tree $=$ max. path length from root to leaf

If $n$ items are inserted one-at-a-time, how big is $h$?
- Worst-case:
- Best-case:
- Average-case:

AVL Trees

Introduced by Adel’son-Vel’skiĭ and Landis in 1962, an AVL Tree is a BST with an additional structural property:
The heights of the left and right subtree differ by at most 1.
(The height of an empty tree is defined to be $-1$.)

At each non-empty node, we store $height(R) - height(L) \in \{-1, 0, 1\}$:
- $-1$ means the tree is left-heavy
- $0$ means the tree is balanced
- $1$ means the tree is right-heavy

- We could store the actual height, but storing balances is simpler and more convenient.

AVL insertion

To perform $\text{insert}(T, k, v)$:
- First, insert $(k, v)$ into $T$ using usual BST insertion
- Then, move up the tree from the new leaf, updating balance factors.
- If the balance factor is $-1$, 0, or 1, then keep going.
- If the balance factor is $\pm 2$, then call the fix algorithm to “rebalance” at that node. We are done.
How to “fix” an unbalanced AVL tree

**Goal:** change the *structure* without changing the *order*

Notice that if heights of $A, B, C, D$ differ by at most 1, then the tree is a proper AVL tree.

---

**Right Rotation**

This is a *right rotation* on node $z$:

Note: Only two edges need to be moved, and two balances updated. Useful to fix left-left imbalance.

---

Again . . .
Left Rotation

This is a left rotation on node z:

Again, only two edges need to be moved and two balances updated. Useful to fix right-right imbalance.

Pseudocode for rotations

rotate-right(T)
T: AVL tree
returns rotated AVL tree
1. newroot ← T.left
2. T.left ← newroot.right
3. newroot.right ← T
4. return newroot

rotate-left(T)
T: AVL tree
returns rotated AVL tree
1. newroot ← T.right
2. T.right ← newroot.left
3. newroot.left ← T
4. return newroot
Double Right Rotation

This is a *double right rotation* on node z:

First, a left rotation on the left subtree (y). Second, a right rotation on the whole tree (z).
Useful for left-right imbalance.

Double Left Rotation

This is a *double left rotation* on node z:

Right rotation on right subtree (y), followed by left rotation on the whole tree (z).
Useful for right-left imbalance.
Fixing a slightly-unbalanced AVL tree

Idea: Identify one of the previous 4 situations, apply rotations

\[ \text{fix}(T) \]

\[ T: \text{AVL tree with } T.\text{balance} = \pm 2 \]
returns a balanced AVL tree
1. if \( T.\text{balance} = -2 \) then
2. if \( T.\text{left}.\text{balance} = 1 \) then
3. \( T.\text{left} \leftarrow \text{rotate-left}(T.\text{left}) \)
4. return \text{rotate-right}(T)
5. else if \( T.\text{balance} = 2 \) then
6. if \( T.\text{right}.\text{balance} = -1 \) then
7. \( T.\text{right} \leftarrow \text{rotate-right}(T.\text{right}) \)
8. return \text{rotate-left}(T)

AVL Tree Operations

**search**: Just like in BSTs, costs \( \Theta(\text{height}) \)

**insert**: Shown already, total cost \( \Theta(\text{height}) \)
- \( \text{fix} \) restores the height of the tree it fixes to what it was,
- so \( \text{fix} \) will be called at most once.

**delete**: First search, then swap with successor (as with BSTs), then move up the tree and apply \( \text{fix} \) (as with \( \text{insert} \)).
- \( \text{fix} \) may be called \( \Theta(\text{height}) \) times.
Total cost is \( \Theta(\text{height}) \).

AVL tree examples

**Example**:

```
22
  -1
 10
    1
    4
      1
      6
      0
      16
      0
    14
      1
      13
      0
      18
      -1
    28
      0
    31
      1
      37
      1
      46
      0
      0
```

Haque, Irvine, Smith (SCS, UW) CS240 - Module 4 Spring 2017 19 / 23

Haque, Irvine, Smith (SCS, UW) CS240 - Module 4 Spring 2017 20 / 23

Haque, Irvine, Smith (SCS, UW) CS240 - Module 4 Spring 2017 21 / 23
Height of an AVL tree

Define $N(h)$ to be the least number of nodes in a height-$h$ AVL tree.

One subtree must have height at least $h - 1$, the other at least $h - 2$:

$$N(h) = \begin{cases} 
1 + N(h - 1) + N(h - 2), & h \geq 1 \\
1, & h = 0 \\
0, & h = -1
\end{cases}$$

What sequence does this look like?

AVL Tree Analysis

Easier lower bound on $N(h)$:

$$N(h) > 2N(h - 2) > 4N(h - 4) > 8N(h - 6) > \cdots > 2^iN(h - 2i) \geq 2^{\lceil h/2 \rceil}$$

Since $n > 2^{\lceil h/2 \rceil}$, $h \leq 2 \log n$.

and thus an AVL tree with $n$ nodes has height $O(\log n)$.

Also, $n \leq 2^{h+1} - 1$, so the height is $\Theta(\log n)$.

$\Rightarrow$ search, insert, delete all cost $\Theta(\log n)$. 