COMP 3170 - Analysis of Algorithms & Data Structures

Shahin Kamali

Binary Search Trees
CLRS 12.2, 12.3, 13.2, read problem 13-3
University of Manitoba
Dictionary ADT

**Definition**

A *dictionary* is a collection $S$ of *items*, each of which contains a *key* and some *data*, and is called a *key-value pair* (KVP).

- It is sometimes called an *associative array*, a *map*, or a *symbol table*.
- Keys can be compared and are (typically) unique.
- We often focus on keys; associating data with keys is easy.

**Operations:**

- $\text{search}(x)$: return true iff $x \in S$
- $\text{insert}(x, v)$: $S \leftarrow S \cup \{x\}$
- $\text{delete}(x)$: $S \leftarrow S \setminus \{x\}$
- additional: $\text{join}$, $\text{isEmpty}$, $\text{size}$, etc.
Optional Operations

In addition to the main operations (search, insert, delete), the followings are useful:

- **predecessor**\( (x) \): return the largest \( y \in S \) such that \( y < x \)
- **successor**\( (x) \): return the smallest \( y \in S \) such that \( y > x \)
- **rank**\( (x) \): return the index of \( x \) in the sorted array
- **select**\( (i) \): return the key at index \( i \) in the sorted array → \( i \)’th order statistic
- **isEmpty**\( (x) \): return true if \( S \) is empty
Dictionaries

- Dictionary is a collection of key-value pairs with the support of search, insert, delete (and possibly some other operations).
- There is a total ordering of elements, i.e., keys are comparable.
- Is dictionary an abstract data type or a data structure?
  - It is an abstract data type; we did not discuss implementation.
  - Different data structures can be used to implement dictionaries.
Elementary Implementations

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

- **Unsorted array or linked list**
  - \( \text{search} \ \Theta(n) \)
  - \( \text{insert} \ \Theta(1) \)
  - \( \text{delete} \ \Theta(n) \) (need to search)

- **Sorted array**
  - \( \text{search} \ \Theta(\log n) \)
  - \( \text{insert} \ \Theta(n) \)
  - \( \text{delete} \ \Theta(n) \)
## Data Structures for Dictionaries

<table>
<thead>
<tr>
<th></th>
<th>space</th>
<th>search</th>
<th>insert/delete</th>
<th>predecessor</th>
</tr>
</thead>
<tbody>
<tr>
<td>unsorted array,linked list</td>
<td>$\Theta(n + a)$</td>
<td>$\Theta(n)$</td>
<td>$\Theta(1)/\Theta(n)$</td>
<td>$\Theta(n)$</td>
</tr>
<tr>
<td>sorted array</td>
<td>$\Theta(n + a)$</td>
<td>$\Theta(\log n)$</td>
<td>$\Theta(n)$</td>
<td>$\Theta(\log n)$</td>
</tr>
<tr>
<td>sorted linked-list</td>
<td>$\Theta(n)$</td>
<td>$\Theta(n)$</td>
<td>$\Theta(n)$</td>
<td>$\Theta(n)$</td>
</tr>
<tr>
<td>unbalanced BST</td>
<td>$\Theta(n)$</td>
<td>$\Theta(n)$</td>
<td>$\Theta(n)$</td>
<td>$\Theta(n)$</td>
</tr>
<tr>
<td>balanced BST</td>
<td>$\Theta(n)$</td>
<td>$\Theta(\log n)$</td>
<td>$\Theta(\log n)$</td>
<td>$\Theta(\log n)$</td>
</tr>
<tr>
<td>hash tables</td>
<td>$\Theta(n + a)$</td>
<td>$\Theta(1)^*$</td>
<td>$\Theta(1)^*$</td>
<td>$\Theta(n + a)$</td>
</tr>
<tr>
<td>skip list</td>
<td>$\Theta(n)^*$</td>
<td>$\Theta(\log n)^*$</td>
<td>$\Theta(\log n)^*$</td>
<td>$\Theta(\log n)^*$</td>
</tr>
</tbody>
</table>

- $n$: number of KVPs.
- $a$: the length of array; when we use sorted/unordered arrays, $a \geq n$.
- $^*$: expected time/space
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 $T.left$ is less than the root key.
Every key $k$ in $T.right$ is greater than the root key.
BST Search and Insert

\[ search(k) \] Compare \( k \) to current node, stop if found, else recurse on subtree unless it’s empty

\[ insert(k, v) \] Search for \( k \), then insert \((k, v)\) as new node

Example:
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
  - successor and predecessor have one or zero children (why?)
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:**
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 $T.left$ is less than the root key.
Every key $k$ in $T.right$ is greater than the root key.
BST Search and Insert

\textit{search}\((k)\) Compare \(k\) to current node, stop if found, else recurse on subtree unless it’s empty

\textit{insert}\((k, v)\) Search for \(k\), then insert \((k, v)\) as new node

Example:
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
  - predecessor is the rightmost node on the left subtree
  - successor is the leftmost node on the right subtree
Binary Search Trees

- How to find max/min elements in a BST?
  - Just find the rightmost/leftmost node in $\Theta(h)$ time
- How can I print all keys in sorted order?
  - Do an in-order traversal of the tree in $\Theta(n)$ time
  - Can we do that in $o(n)$? no! we need to report an output of size $n$

BSTs maintain data in sorted order, which is useful for some queries (an advantage over hash tables which scatter data).
search, insert, delete all have cost $\Theta(h)$, where $h = \text{height of the tree} = \text{max. path length from root to leaf}$

If $n$ items are inserted one-at-a-time, how big is $h$?

- Worst-case: $\Theta(n)$
- Best-case: $\Theta(\log n)$
- Average-case: $\Theta(\log n)$
  (similar analysis to quick-sort1)
**Balanced BSTs**

- Perfectly balanced BSTs: all nodes except for the bottom 2 levels are full (have two children).
  - Too strict for efficient BST balancing.

- Weight balanced: at each internal node $i$, at least $cn_i$ nodes are in its left subtree and $cn_i$ in its right subtree, for some constant $c \in (0, 1/2]$, where $n_i$ denotes the number of descendants for node $i$.

- Height balanced: heights of left and right subtrees of each internal node differ by at most $k$, for some constant $k \geq 1$.
  - For AVL trees, $k = 1$.
  - We will assume $k = 1$ for the remainder of our discussion.

- Height $\Theta(\log n)$ where $n$ is the number of nodes in the tree.

- **All balanced BSTs (with respect to any of above definitions) have height $\Theta(\log n)$**
  - We see the proof for height-balanced BSTs in a minute.
Tree height

**Definition**

The **height** of a node $a$ is the length of the longest path between $a$ and any descendent of $a$

- as opposed to **depth** which is the length of the path between $a$ and the root.
- Height can be defined recursively as follows:

  $$
  \text{height}(a) = \begin{cases} 
  -1, & a = \Phi \\
  1 + \max\{\text{height}(a.\text{left}), \text{height}(a.\text{right})\} & a \neq \Phi
  \end{cases}
  $$

- For a height-balanced BST with $k = 1$, the balancing factor (the difference between the height of the two children) for any node is in $\{-1, 0, 1\}$. 
Bounds for the height of height-balanced BSTs

**Theorem**

For the height \( h(n) \) of a height-balanced BST (with \( k = 1 \)) on sufficiently large \( n \) nodes we have

\[
\log(n) - 1 < h(n) < 1.45 \log(n + 1)
\]

This implies \( h(n) \in \Theta(\log n) \).

Let’s see the proof.
Lower Bound for the height of height-balanced BSTs

- We want to prove $\log(n) - 1 < h(n)$.
- The number of nodes in a binary search tree of height $h$ is at most:
  
  $$n \leq 2^{h+1} - 1 \Rightarrow \log n \leq \log(2^{h+1} - 1) < \log(2^{h+1}) = h + 1$$

  Hence, we have $\log n - 1 < h$. 
Upper Bound for the height of height-balanced BSTs

- We want to show \( h(n) < 1.45 \log(n + 1) \).
  - Let \( s(n) \) denote the minimum number of nodes in a height-balanced BST (with \( k = 1 \))
  - We have \( s(0) = 1 \) \( s(1) = 2 \) \( s(2) = 4 \)

\[
s(h) = \begin{cases} 
1 & h = 0 \\
2 & h = 1 \\
 s(h - 1) + s(h - 2) + 1, & h \geq 2 
\end{cases}
\]

- We can say \( s(h) > F(h) \) where \( F(h) \) is the \( h \)'th Fibonacci number.
  - For large \( n \), we have \( F(h) \approx \frac{1}{\sqrt{5}} \left( \frac{1 + \sqrt{5}}{2} \right)^{h+1} - 1 \)

We have \( n > \frac{1}{\sqrt{5}} \left( \frac{1 + \sqrt{5}}{2} \right)^{h+1} - 1 \rightarrow \sqrt{5}(n + 1) \geq \left( \frac{1 + \sqrt{5}}{2} \right)^{h+1} \rightarrow \log(\sqrt{5}(n + 1)) \geq (h + 1) \log(\frac{1+\sqrt{5}}{2}) \rightarrow h < \frac{\log \sqrt{5} + \log(n+1)}{\log(1+\sqrt{5})-1} - 1 \)

\[
= \frac{1}{\log(1+\sqrt{5})-1} \log(n + 1) + \frac{\log \sqrt{5}}{\log(1+\sqrt{5})-1} - 1 < 1.45 \log(n + 1)
\]
Bounds for the height of height-balanced BSTs

**Theorem**

For the height $h(n)$ of a height-balanced BST (with $k = 1$) on sufficiently large $n$ nodes we have $\log(n) - 1 < h(n) < 1.45 \log(n + 1)$

- This implies $h(n) \in \Theta(\log n)$.
- So, it is desirable to maintain a height-balanced binary search tree (they are asymptotically the best possible BSTs).
BST Single Rotation

- Height of a height-balanced BST on $n$ nodes is $\Theta(\log n)$
- A self-balancing BST maintains the height-balanced property after an insertion/deletion via tree rotation

Every rotation swaps parent-child relationship between two nodes (here between 2 and 4)

Tree rotation preserves the BST key ordering property.

Each rotation requires updating a few pointers in $O(1)$ time.

original height: $\max(\text{height}(a) + 2; \text{height}(b) + 2; \text{height}(c) + 1)$
new height: $\max(\text{height}(a) + 1; \text{height}(b) + 2; \text{height}(c) + 2)$
AVL Trees

- Introduced by Adel’son-Vel’skii and Landis in 1962
- An AVL Tree is a height-balanced BST
  - 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 $\text{height}(R) - \text{height}(L) \in \{-1, 0, 1\}$:
  - $-1$ 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 $\text{fix}$ algorithm to “rebalance” at that node.
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

- When the followings hold, we apply a right rotation on node $z$
  - The balance factor at $z$ is -2.
  - The balance factor of $y$ is 0 or -1.

**Note:** Only two edges need to be moved, and two balances updated.
Left Rotation

When the followings hold, we apply a **left rotation** on node $z$

- The balance factor at $z$ is 2.
- The balance factor of $y$ is 0 or 1.

Again, only two edges need to be moved and two balances updated.
Pseudocode for rotations

\[\text{rotate-right}(T)\]
\[
T: \text{AVL tree} \\
\text{returns rotated AVL tree} \\
1. newroot ← T.left \\
2. \(T.left \leftarrow \text{newroot.right}\) \\
3. \(\text{newroot.right} \leftarrow T\) \\
4. \textbf{return} newroot
\]

\[\text{rotate-left}(T)\]
\[
T: \text{AVL tree} \\
\text{returns rotated AVL tree} \\
1. newroot ← T.right \\
2. \(T.right \leftarrow \text{newroot.left}\) \\
3. \(\text{newroot.left} \leftarrow T\) \\
4. \textbf{return} newroot\]
Double Right Rotation

- When the followings hold, we apply a **double right rotation** on z
  - The balance factor at z is -2 & the balance factor of y is 1.

First, a left rotation on the left subtree (y).
Second, a right rotation on the whole tree (z).
Double Left Rotation

This is a *double left rotation* on node $z$; apply when balance of $z$ is 2 and balance of $y$ is -1.

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

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

```plaintext
fix(T)
T: AVL tree with T.balance = ±2
returns a balanced AVL tree
1. if T.balance = −2 then
2. if T.left.balance = 1 then
3. T.left ← rotate-left(T.left)
4. return rotate-right(T)
5. else if T.balance = 2 then
6. if T.right.balance = −1 then
7. T.right ← rotate-right(T.right)
8. return rotate-left(T)
```
AVL Tree Operations

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

**insert**: Shown already, total cost $\Theta(\text{height})$

*fix* will be called *at most once*.

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

Total cost is $\Theta(\text{height})$. 
AVL tree examples

Example:

```
        22
         /-1-
        /     /
10  1   31 1
  /                  /
4  1   14 1   28 0
 /   /                 /
6  0   13 0   18 -1
      /                  /
     16 0               46 0
```
AVL tree analysis

- Since AVL-trees are height-balanced, their height is $\Theta(\log n)$ (previous class)
- Search can be done as before (no need for rebalancing)
- $\text{Insert}(x)$ takes $\Theta(\log n)$ and involves at most one fix.
- $\text{Delete}(x)$ takes $\Theta(\log n)$ and involves at most $\Theta(\log n)$ fixes.

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

- What about other queries (e.g., get-max(), get-min(), rank(), select())?
- One great thing about AVL trees is that they can be easily augmented to support these queries in a good time (this is the main advantage of the trees over say Hash tables).
Augmented Data Structures

- In practice, it often happens that you want an abstract data type to support additional queries.
  - To implement this, we need to **augment** the underlying data structure.
  - Augmentation often involves storing additional data which facilitates the query.

- Consider AVL tree which supports search, insert, delete in $\Theta(\log n)$ time.

  - What if your ‘boss’ asks you to **additionally** support minimum, maximum, rank, and select?
  - Without augmentation, minimum and maximum take $\Theta(\log n)$ while rank and select require linear time (in-order traversal to retrieve the sorted list of keys).
  - What if your angry boss wants them to be faster?
Augmenting AVL trees

- We can augment AVL trees to support minimum/maximum in $\Theta(1)$.
- Just add a pointer to the leftmost/rightmost leaf of the tree.
- After updating the tree by an insert/deleted, make sure that the pointer still points to the smallest/largest element.
Augmenting AVL trees

- After an insertion, update the minimum pointer
  - If the newly inserted key is less than minimum, update the the minimum pointer to point to it (similar for maximum pointer).
  - It takes an additional time of $\Theta(1)$ (the insertion time is still $\Theta(\log n)$)

- Similar update for max pointer
Augmenting AVL trees

- After a deletion, update the minimum pointer
  - Check if the minimum element was deleted. If so, update the minimum pointer to the successor of the deleted element
  - Finding the successor takes additional time of $\Theta(1)$
    - Let $x$ be the min element before deletion; there is nothing on the left of $x$.
    - The right subtree of $x$ has zero or one node (otherwise $x$ is unbalanced).
    - If there is an item $y$ on the right of $x$, then it is the successor of $x$
    - If $y$ is a leaf, then its parent is the successor

- Similar update for max pointer
Augmenting AVL trees

**Theorem**

We can augment AVL trees by adding only two pointers ($\Theta(1)$) extra space to support minimum/maximum queries in $\Theta(1)$ and without changing time complexity of other queries (insertion, deletion, and search).
Augmenting AVL trees

- Can we augment AVL trees to support rank/select operations in $O(\log n)$ time?
  - $\text{rank}(x)$ reports the index of key $x$ in the sorted array of keys
  - $\text{select}(i)$ returns the key with index $i$ in the sorted array of keys

- Idea 1: Store the rank of each node at that node.
  - $O(\log n)$ rank and select are guaranteed (why?)
  - Is it a good augment data structure? No because inserting an item (e.g., key 1 here) might require updating all stored ranks
    Insertion/deletion take $\Theta(n)$. Failed!
Augmenting AVL trees

- Idea 2: At each node, store the size (no. of nodes) of the subtree rooted at that node
  - The size of a node is the sum of the sizes of its two subtrees plus 1.
  - The size of an empty subtree is 0.
- The rank of a node $x$ in its own subtree is the size of its left subtree.
  - E.g., rank of root 12 is 6

```
2  1  1

/    \\    /  \/
3  8  11  17
  /   \    /  \
 2    1   1
```

```
12  11
  /   \    /  \
 7    15 4
  /  \      /  \
3    9    13 24
  /  \  /  \  /  \/
 2  8  11  17
```

Augmenting AVL trees

Selection on an AVL tree augmented with size data is similar to quickselect, where the root acts as a pivot.

Select(i): compare i with the rank of the root r (size of left subarray).

- If equal, return the root r
- if $i < \text{rank(root)}$, recursively find the same index i in the left subtree
- if $i > \text{rank(root)}$, recursively find index $i - \text{rank(root)} - 1$ in the right subtree

E.g., select(x,5) $\rightarrow$ select(y,5) $\rightarrow$ select(z,2) $\rightarrow$ select(w,0) $\rightarrow$ w (11) is returned
Augmenting AVL trees

select \((node, i)\)

- If \(node = \emptyset\) then return error \((i\) exceeds number of nodes). [Could have checked this at the root: if \(i \geq node.size\)]
- If \(i = node.left.size\) then return \(node.key\).
- If \(i < node.left.size\) then return \(select(node.left, i)\).
- If \(i > node.left.size\) then return \(select(node.right, i - node.left.size - 1)\).