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:

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

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

- \textit{predecessor}(x): return the largest \( y \in S \) such that \( y < x \)
- \textit{successor}(x): return the smallest \( y \in S \) such that \( y > x \)
- \textit{rank}(x): return the index of \( x \) in the sorted array
- \textit{select}(i): return the key at index \( i \) in the sorted array \(\rightarrow i^{th} \text{ order statistic} \)
- \textit{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**

  \[
  \begin{align*}
  \text{search} & \quad \Theta(n) \\
  \text{insert} & \quad \Theta(1) \\
  \text{delete} & \quad \Theta(n) \text{ (need to search)}
  \end{align*}
  \]

- **Sorted array**

  \[
  \begin{align*}
  \text{search} & \quad \Theta(\log n) \\
  \text{insert} & \quad \Theta(n) \\
  \text{delete} & \quad \Theta(n)
  \end{align*}
  \]
## Data Structures for Dictionaries

<table>
<thead>
<tr>
<th>Data Structure</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.

```
       15
     /   \
    6     25
   /     /
  10    23    29
 /     /     /
8     14   27  50
```
BST Search and Insert

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

\textit{insert}(k, v) \text{ Search for } k, \text{ then insert } (k, v) \text{ 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 $= \text{max. path length from root to leaf}$

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**

*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 success\text{\textit{or}}\text{\textit{or}} precedent\text{\textit{o}}\text{\textit{or}}\text{\textit{or}} node and then delete
  - precedent\textit{is the rightmost node on the left subtree}
  - successor\textit{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).
Height of a BST

*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-sort*)
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:

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

- 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.
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 $s(0) = 1 \quad s(1) = 2 \quad s(2) = 4$

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

- 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 $n > \frac{1}{\sqrt{5}} \left(\frac{1+\sqrt{5}}{2}\right)^{h+1} - 1 \rightarrow \sqrt{5}(n + 1) \geq \left(\frac{1+\sqrt{5}}{2}\right)^{h+1} \rightarrow \\
\log(\sqrt{5}(n + 1)) \geq (h + 1) \log\left(\frac{1+\sqrt{5}}{2}\right) \rightarrow h < \frac{\log \sqrt{5} + \log(n+1)}{\log(1+\sqrt{5})-1} - 1 \rightarrow \\
= \frac{1}{\log(1+\sqrt{5})-1} \log(n + 1) + \frac{\log \sqrt{5}}{\log(1+\sqrt{5})-1} - 1 < 1.45 \log(n + 1)$
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’skiǐ 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 $height(R) - height(L) \in \{-1, 0, 1\}$:
  - $-1$ 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 $-1, 0, \text{ or } 1$, 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.
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

\begin{center}
\begin{tabular}{|l|}
\hline
\textbf{rotate-right}(T) \\
\hline
\textit{T: AVL tree} \\
\hline
\textbf{returns rotated AVL tree} \\
\hline
1. \hspace{0.5cm} \textit{newroot} $\leftarrow$ \textit{T}.left \\
2. \hspace{0.5cm} \textit{T}.left $\leftarrow$ \textit{newroot}.right \\
3. \hspace{0.5cm} \textit{newroot}.right $\leftarrow$ \textit{T} \\
4. \hspace{0.5cm} \textbf{return} \hspace{0.5cm} \textit{newroot} \\
\hline
\end{tabular}
\end{center}

\begin{center}
\begin{tabular}{|l|}
\hline
\textbf{rotate-left}(T) \\
\hline
\textit{T: AVL tree} \\
\hline
\textbf{returns rotated AVL tree} \\
\hline
1. \hspace{0.5cm} \textit{newroot} $\leftarrow$ \textit{T}.right \\
2. \hspace{0.5cm} \textit{T}.right $\leftarrow$ \textit{newroot}.left \\
3. \hspace{0.5cm} \textit{newroot}.left $\leftarrow$ \textit{T} \\
4. \hspace{0.5cm} \textbf{return} \hspace{0.5cm} \textit{newroot} \\
\hline
\end{tabular}
\end{center}
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.

```
  z
 /   \
 y---- x
 |   |
 A-- D
```

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

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

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

$fix$ 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 $\Theta(height)$ times.

Total cost is $\Theta(height)$.  


AVL tree examples

Example:

```
22
  /\-1
10 / 31
  /\ /\ |
 4 / 14 / 28 / 37
  /\ /\ /\ /\ |
 6 / 13 / 18 / 46
```

The AVL tree is balanced around the root node 22.
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 Data Structures

- First, figure out what additional information should be stored?
- Second, figure out how, using the additional information, answer new queries (e.g., min and rank in AVL trees) efficiently?
- Third, figure out how to update existing operations (e.g., insertion and deletion) to keep the stored information updated.
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, first, re-arrange the tree if required (to keep it AVL). Keep a pointer to the newly inserted element.
  - After the insertion, if the newly inserted key is less than minimum, update 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

![Augmented AVL Tree Diagram]
Augmenting AVL trees

- For deleting node \( x \), check if \( x \) is the minimum element. If so, first update the minimum pointer to the successor of \( x \).
- Finding the successor of minimum takes additional time of \( \Theta(1) \)
  - Let \( x \) be the min element before deletion; we know 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.
- After updating the pointer, delete as in regular AVL trees.
- Similar update for max pointer

![Diagram of AVL tree with minimum and maximum pointers highlighted.](image)
Augmenting AVL trees

Theorem

We can augment AVL trees by adding only two pointers (Θ(1)) extra space to support minimum/maximum queries in Θ(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?

- $rank(x)$ reports the index of key $x$ in the sorted array of keys
- $select(i)$ 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.
Augmenting AVL trees

- We want to 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
- At each node, store the size (no. of nodes) of the subtree rooted at that node.
Selection in Augmented 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 \(i < \text{rank}(\text{root})\), recursively find the same index \(i\) in the left subtree
  - if \(i > \text{rank}(\text{root})\), recursively find index \(i - \text{rank}(\text{root}) - 1\) in the right subtree

E.g., select\(5,12\) \(\rightarrow\) select\(5,7\) \(\rightarrow\) select\(2,9\) \(\rightarrow\) select\(0,11\) \(\rightarrow\) 11

is returned
Augmenting AVL trees

- To find \( \text{rank}(x) \) on an AVL tree augmented, search for \( k \).
- On the path from the root to \( x \), sum up sizes of all left sub trees
  - When searching for \( x \), when you recurs on the right subtree, add up the size of the left subtree plus one (for the current node).
  - When the node was found, add up the size of its left subtree to the computed rank.

\[
\begin{align*}
\text{rank}(16,20) & \xrightarrow{\text{left}} \text{rank}(16,12) \quad \text{res} \quad \text{res} \quad \text{res} \quad \text{res} \\
& \xrightarrow{\text{right}} \text{rank}(16,17) \xrightarrow{\text{left}} \\
\text{rank}(16,14) & \xrightarrow{\text{right}} \text{rank}(16,16) \quad \text{res} \quad \text{res} \\
& \xrightarrow{\text{right}} \text{rank}(25,28) \xrightarrow{\text{left}} \text{rank}(25,25) \quad \text{res} \\
& \text{res} \quad \text{res} \quad \text{res} \\
& \quad \text{res} \quad \text{res} \quad \text{res} \quad \text{res} \\
\end{align*}
\]
Augmenting AVL trees

\[ \text{rank}(\text{searchKey}) \]

- \text{return rank}(\text{searchKey}, \text{root})

\[ \text{rank}(\text{searchKey}, \text{node}) \]

- If \( \text{node} = \emptyset \) then return \(-\infty\) (node doesn’t exist).
- If \( \text{searchKey} = \text{node.key} \) then return \( \text{node.left.size} \).
- If \( \text{searchKey} < \text{node.key} \) then return \( \text{rank}(\text{searchKey}, \text{node.left}) \).
- If \( \text{searchKey} > \text{node.key} \) then return \( 1 + \text{node.left.size} + \text{rank}(\text{searchKey}, \text{node.right}) \).
Updating Augmented AVL trees

- After an **insertion**, the sizes of all ancestors of the new node should be incremented; do it before fixing the tree.

- After a **deletion**, the sizes of all ancestors of the deleted node should be decremented; do it before fixing the tree.

- The 2 nodes involved in each **single rotation** must have their sizes updated. (recall that double rotation involves two single rotations)
  - Only sizes of A and B should be updated. It can be done in constant time!

\[
\begin{align*}
\alpha + \beta + \gamma + 2 & \quad \alpha + \beta + 1 \\
\alpha + \beta \quad \gamma & \quad \alpha \\
\alpha & \quad \beta \quad \gamma
\end{align*}
\]
Updating Augmenting AVL trees

- `insert(2)`: first insert the new node and update sizes of ancestors.
- After the insertion, node 3 is unbalanced, since it is left-heavy and its left child (1) is right heavy, first apply a left rotation; update the sizes of the two involved node (1 and 2).
- Now 3 is left-heavy and its left child (2) is not right-heavy; apply a single rotation between them and update their sizes.
Augmenting AVL trees

**Theorem**

*It is possible to augment an AVL tree by storing the sizes of each subtree so that select and rank operations can be supported in $\Theta(\log n)$ time. The time complexity of other operations (search, insert, and delete) remain unchanged.*

- In fact, we can merge such AVL tree with a doubly linked list to support predecessor and successor operations.
Augmented Data Structures Summary

- **Steps to Augmenting a Data Structure**
  - Specify an ADT (including additional operations to support).
  - Choose an underlying data structure.
  - Determine the additional data to be maintained.
  - Develop algorithms for new operations.
  - Verify that the additional data can be maintained efficiently during updates.