Module 4: Dictionaries and Balanced Search Trees

CS 240 - Data Structures and Data Management

Reza Dorrigiv, Daniel Roche

School of Computer Science, University of Waterloo

Winter 2010
Dictionary ADT

A dictionary is a collection of items, each of which contains a key and 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.

Examples: symbol table, license plate database
Elementary Implementations

Common assumptions:

- Dictionary has \( n \) KVPs
- Each KVP uses constant space
  (if not, the “value” could be a pointer)
- Comparing keys takes constant time

Unordered array or linked list

- \textit{search} \( \Theta(n) \)
- \textit{insert} \( \Theta(1) \)
- \textit{delete} \( \Theta(1) \) (after a search)

Ordered array or linked list

- \textit{search} \( \Theta(\log n) \)
- \textit{insert} \( \Theta(n) \)
- \textit{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 $T.left$ is less than the root key. Every key $k$ in $T.right$ is greater than the root key.

![Binary Search Tree Diagram]
BST Search and Insert

\textit{search}(k) \hspace{2pt} \text{Compare } k \text{ to current node, stop if found, else recurse on subtree unless it’s empty}

Example: \textit{search}(24)

\begin{center}
\begin{tikzpicture}
  \node (root) {15}
  \child{node (left) {6}
    \child{node (left_left) {10}
      \child{node (left_left_left) {8}}
      \child{node (left_left_right) {14}}
    }
    \child{node (left_right) {23}
      \child{node (left_right_left) {27}}
      \child{node (left_right_right) {29}}
    }
  }
  \child{node (right) {25}
    \child{node (right_right) {29}}
    \child{node (right_right_right) {50}}
  }
\end{tikzpicture}
\end{center}
**BST Search and Insert**

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

Example: \(\text{search}(24)\)
**BST Search and Insert**

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

Example: *search*(24)
**BST Search and Insert**

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

Example: `search(24)`
**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: *insert*(24, \(\ldots\))
BST Delete

- If node is a leaf, just delete it.
BST Delete

- If node is a leaf, just delete it.
BST Delete

- If node is a leaf, just delete it.
- If node has one child, move child up
 BST Delete

- If node is a leaf, just delete it.
- If node has one child, move child up
BST Delete

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

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

- If node is a leaf, just delete it.
- If node has one child, move child up
- Else, swap with *successor* node and then delete
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:
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: $n - 1 = \Theta(n)$
- Best-case:
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:** $n - 1 = \Theta(n)$
- **Best-case:** $\lg(n + 1) - 1 = \Theta(\log n)$
- **Average-case:**
**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:** $n - 1 = \Theta(n)$
- **Best-case:** $\lg(n + 1) - 1 = \Theta(\log n)$
- **Average-case:** $\Theta(\log n)$
  (just like recursion depth in *quick-sort*1)
AVL Trees

Introduced by Adel’son-Vel’skiǐ 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 $\text{height}(R) - \text{height}(L) \in \{-1, 0, 1\}$:

- $-1$ means the tree is left-heavy
- $0$ means the tree is balanced
- $1$ means the tree is right-heavy
AVL Trees

Introduced by Adel’son-Vel’skiï 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 $\text{height}(R) - \text{height}(L) \in \{-1, 0, 1\}$:

- $−1$ means the tree is left-heavy
- $0$ means the tree is balanced
- $1$ means the tree is right-heavy

Why not just store the actual height?
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 $\text{height}(R) - \text{height}(L) \in \{-1, 0, 1\}$:

- $-1$ means the tree is left-heavy
- $0$ means the tree is balanced
- $1$ means the tree is right-heavy

Why not just store the actual height? It would take $\Theta(n \log \log n)$ space.
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

This is a *right rotation* on node z:

```
   z
  / 
 y   D
 |   
 x   C
|   |   
A   B
```

```
   y
  / 
 x   z
 |   |
 A   B
|   |   
C   D
```

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

This is a *right rotation* on node z:

![Diagram of right rotation]

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

This is a *left rotation* on node $x$:

Again, only two edges need to be moved and two balances updated.
Double Right Rotation

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

First, a left rotation on the left subtree ($x$).
Double Right Rotation

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

![Diagram of the double right rotation]

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

This is a *double left rotation* on node $x$:

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

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

\[
\text{fix}(T) \\
T: \text{AVL tree with } T.balance = \pm 2 \\
1. \quad \text{if } T.balance = -2 \text{ then} \\
2. \quad \text{if } T.left.balance = 1 \text{ then} \\
3. \quad \text{rotate-left}(T.left) \\
4. \quad \text{rotate-right}(T) \\
5. \quad \text{else if } T.balance = 2 \text{ then} \\
6. \quad \text{if } T.right.balance = -1 \text{ then} \\
7. \quad \text{rotate-right}(T.right) \\
8. \quad \text{rotate-left}(T)
\]
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: $\text{insert}(8)$
AVL tree examples

Example: \textit{insert}(8)
AVL tree examples

Example: \textit{insert}(8)
AVL tree examples

Example: $\text{insert}(8)$
AVL tree examples

Example: \textit{insert}(8)
AVL tree examples

Example: \textit{delete}(22)
AVL tree examples

Example: \textit{delete}(22)
AVL tree examples

Example: delete(22)
AVL tree examples

Example: delete(22)
AVL tree examples

Example: \textit{delete}(22)
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?

The Fibonacci sequence!

$$N(h) = F_{h+3} - 1 = \left\lceil \frac{\phi^h}{\sqrt{5}} \right\rceil - 1$$

where $\phi = \frac{1 + \sqrt{5}}{2}$.
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? The Fibonacci sequence!

$$N(h) = F_{h+3} - 1 = \left\lfloor \frac{\varphi^{h+3}}{\sqrt{5}} \right\rfloor - 1,$$

where $\varphi = \frac{1 + \sqrt{5}}{2}$.
Easier lower bound on $N(h)$:

\[ N(h) > 2N(h - 2) > 4N(h - 4) > 8N(h - 6) > \cdots > 2^i N(h - 2i) \geq 2^{\lfloor h/2 \rfloor} \]
AVL Tree Analysis

Easier lower bound on $N(h)$:

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

Since $n > 2^{\lceil h/2 \rceil}$, $h \leq 2 \lg n$, and 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 $\Theta(\log n)$. 

Reza Dorrigiv, Daniel Roche (CS, UW)  
CS240 - Module 4  
Winter 2010  
19 / 29
A 2-3 Tree is like a BST with additional structural properties:

- Every node either contains one KVP and two children, or two KVPs and three children.
- All the leaves are at the same level.
  (A leaf is a node with empty children.)

Searching through a 1-node is just like in a BST.
For a 2-node, we must examine both keys and follow the appropriate path.
Insertion in a 2-3 tree

First, we search to find the leaf where the new key belongs.

If the leaf has only 1 KVP, just add the new one to make a 2-node.

Otherwise, order the three keys as $a < b < c$. Split the leaf into two 1-nodes, containing $a$ and $c$, and (recursively) insert $b$ into the parent along with the new link.
2-3 Tree Insertion

Example: \textit{insert}(19)
2-3 Tree Insertion

Example: \textit{insert}(19)
2-3 Tree Insertion

Example: $insert(19)$
2-3 Tree Insertion

**Example:** \(insert(19)\)
2-3 Tree Insertion

Example: $\text{insert}(41)$
2-3 Tree Insertion

Example: \textit{insert}(41)
2-3 Tree Insertion

Example: \textit{insert}(41)
2-3 Tree Insertion

Example: $\text{insert}(41)$

```
 25 36 43
 18 21       31        41        51
 12 19 24 28 33 39 42 48 56 62
```
2-3 Tree Insertion

Example: \textit{insert}(41)
Deletion from a 2-3 Tree

As with BSTs and AVL trees, we first swap the KVP with its successor, so that we always delete from a leaf.

Say we’re deleting KVP $x$ from a node $V$:

- If $X$ is a 2-node, just delete $x$.
- Elseif $X$ has a 2-node sibling $U$, perform a transfer:
  Put the “intermediate” KVP in the parent between $V$ and $U$ into $V$, and replace it with the adjacent KVP from $U$.
- Otherwise, we merge $V$ and a 1-node sibling $U$:
  Remove $V$ and (recursively) delete the “intermediate” KVP from the parent, adding it to $U$. 
2-3 Tree Deletion

Example: \textit{delete}(43)
2-3 Tree Deletion

Example: $\text{delete}(43)$
2-3 Tree Deletion

Example: $\textit{delete}(43)$
2-3 Tree Deletion

Example: delete(19)
2-3 Tree Deletion

Example: delete(19)
Example: delete(19)
2-3 Tree Deletion

Example: delete(42)
2-3 Tree Deletion

Example: $\text{delete}(42)$
2-3 Tree Deletion

Example: $\text{delete}(42)$
2-3 Tree Deletion

Example: \textit{delete}(42)
2-3 Tree Deletion

**Example:** \(\text{delete}(42)\)
B-Trees

The 2-3 Tree is a specific type of B-tree:

A *B-tree of minsize* $d$ is a search tree satisfying:
- Each node contains at most $2d$ KVPs.  
  Each non-root node contains at least $d$ KVPs.  
- All the leaves are at the same level.

Some people call this a B-tree of order $(2d + 1)$, or a $(d + 1, 2d + 1)$-tree.  
A 2-3 tree has $d = 1$.

*search, insert, delete* work just like for 2-3 trees.
Height of a B-tree

What is the least number of KVPs in a height-\(h\) B-tree?

<table>
<thead>
<tr>
<th>Level</th>
<th>Nodes</th>
<th>Node size</th>
<th>KVPs</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>(d)</td>
<td>2(d)</td>
</tr>
<tr>
<td>2</td>
<td>2((d+1))</td>
<td>(d)</td>
<td>2(d)((d+1))</td>
</tr>
<tr>
<td>3</td>
<td>2((d+1)^2)</td>
<td>(d)</td>
<td>2(d)((d+1)^2)</td>
</tr>
<tr>
<td>(\ldots)</td>
<td>(\ldots)</td>
<td>(\ldots)</td>
<td>(\ldots)</td>
</tr>
<tr>
<td>(h)</td>
<td>2((d+1)^{h-1})</td>
<td>(d)</td>
<td>2(d)((d+1)^{h-1})</td>
</tr>
</tbody>
</table>

\[
\text{Total: } 1 + \sum_{i=0}^{h-1} 2d(d+1)^i = 2(d+1)^h - 1
\]

Therefore height of tree with \(n\) nodes is \(\Theta((\log n)/((\log d))).\)
Analysis of B-tree operations

Assume each node stores its KVPs and child-pointers in a dictionary that supports $O(\log d)$ search, insert, and delete.

Then search, insert, and delete work just like for 2-3 trees, and each require $\Theta(\text{height})$ node operations.

Total cost is $O \left( \frac{\log n}{\log d} \cdot (\log d) \right) = O(\log n)$. 
Dictionaries in external memory

Tree-based data structures have poor memory locality: If an operation accesses $m$ nodes, then it must access $m$ spaced-out memory locations.

**Observation**: Accessing a single location in external memory (e.g. hard disk) automatically loads a whole block (or “page”).

In an AVL tree or 2-3 tree, $\Theta(\log n)$ pages are loaded in the worst case.

If $d$ is small enough so a $2^d$-node fits into a single page, then a B-tree of minsize $d$ only loads $\Theta((\log n)/(\log d))$ pages.

This can result in a huge savings: memory access is often the largest time cost in a computation.
B-tree variations

**Max size** $2d + 1$: Permitting one additional KVP in each node allows *insert* and *delete* to avoid *backtracking* via *pre-emptive splitting* and *pre-emptive merging*.

**Red-black trees**: Identical to a B-tree with minsize 1 and maxsize 3, but each 2-node or 3-node is represented by 2 or 3 binary nodes, and each node holds a “color” value of red or black.

**B⁺-trees**: All KVPs are stored at the leaves (interior nodes just have keys), and the leaves are linked sequentially.