CMPSC250
Lecture 13: Binary Search Trees

Prof. John Wenskovitch
02/22/2016
Last Time

• Symbol tables – a data structure associating a key with some value info
• Sequential search
• Binary search
Binary Search Visual

successful search for P

<table>
<thead>
<tr>
<th>lo</th>
<th>hi</th>
<th>mid</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>9</td>
<td>4</td>
</tr>
<tr>
<td>5</td>
<td>9</td>
<td>7</td>
</tr>
<tr>
<td>5</td>
<td>6</td>
<td>5</td>
</tr>
<tr>
<td>6</td>
<td>6</td>
<td>6</td>
</tr>
</tbody>
</table>

entries in black are a[lo..hi]

unnecessary search for Q

<table>
<thead>
<tr>
<th>lo</th>
<th>hi</th>
<th>mid</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>9</td>
<td>4</td>
</tr>
<tr>
<td>5</td>
<td>9</td>
<td>7</td>
</tr>
<tr>
<td>5</td>
<td>6</td>
<td>5</td>
</tr>
<tr>
<td>7</td>
<td>6</td>
<td>6</td>
</tr>
</tbody>
</table>

entry in red is a[mid]

loop exits with keys[mid] = P: return

loop exits with lo > hi: return 7
Binary Search Analysis

• **Theorem:** Binary search in an ordered array with \( N \) keys uses no more than \( \log(N) + 1 \) compares for a search, successful or unsuccessful.

• **Proof:**
  
  – Let \( C(N) \) be the number of compares needed to search for a key in a symbol table of size \( N \).
  
  – We know \( C(0) = C(1) = 1 \)
  
  – \( C(N) \leq C(N/2) + 1 \)
  
  – \( C(2^n - 1) \leq C(2^{n-1} - 1) + 1 \)
  
  – \( C(2^n - 1) \leq C(2^{n-2} - 1) + 1 + 1 \)
  
  – \( C(2^n - 1) \leq C(2^0) + 1 + 1 + 1 + \cdots + 1 \)
  
  – \( C(2^n - 1) \leq 1 + n \)
  
  – \( C(N) \leq \log(n) + 1 \)
Binary Search Analysis

• **Theorem**: Inserting a new key into an ordered array of size $N$ uses $\sim 2N$ array accesses in the worst case.

• **Proof**: If we have to insert something at the beginning of the array, we have to move $N$ items one position to the right (one read, one write)

• **Corollary**: Inserting $N$ keys into an initially empty table uses $\sim N^2$ arrays accesses in the worst case.
Binary Search Evaluation

• Pros
  – Still fairly trivial to implement
  – Better than Sequential Search

• Cons
  – Too slow for very, very, very large problems
    • Total number of compares for Leipzig is \(~10^{11}\)
    • Binary Search reduces the number of compares, but not necessarily the overall running time, because the \(\text{put}()\) method is slower.
## Can We Do Better?

<table>
<thead>
<tr>
<th>Algorithm (data structure)</th>
<th>worst-case cost (after N inserts)</th>
<th>average-case cost (after N random inserts)</th>
<th>efficiently support ordered operations?</th>
</tr>
</thead>
<tbody>
<tr>
<td>sequential search</td>
<td>( N )</td>
<td>( N/2 )</td>
<td>no</td>
</tr>
<tr>
<td>unordered linked list</td>
<td>( N )</td>
<td>( N )</td>
<td></td>
</tr>
<tr>
<td>binary search</td>
<td>( \lg N )</td>
<td>( \lg N )</td>
<td>yes</td>
</tr>
<tr>
<td>ordered array</td>
<td>( 2N )</td>
<td>( N )</td>
<td></td>
</tr>
</tbody>
</table>
Goals

- Improve insertion speed beyond the current $N$
- Keep search speed in the $\log(N)$ complexity class
- Keep the implementation as simple as possible

- Improve insertion speed beyond the current $N$
- Keep search speed in the $\log(N)$ complexity class
- Keep the implementation as simple as possible
Binary Tree

- Binary Tree
- Node
- Link
- Root
- Child
- Parent
- Subtree
Binary Search Tree

- **Binary Search Tree** – A binary tree where each node has a Comparable key (and an associated value), and satisfies the restriction that the key in any node is:
  - ...**larger** than the keys in all nodes in that node’s **left** subtree, and
  - ...**smaller** than the keys in all nodes in that node’s **right** subtree.

- *(how is this different from our heap implementation?)*
void put(Key key, Value val) {
    root = put(root, key, val);
} //put

Node put(Node x, Key key, Value val) {
    if (x == null) {
        return new Node(key, val, 1);
    } //if
    int cmp = key.compareTo(x.key);
    if (cmp < 0) {
        x.left = put(x.left, key, val);
    } else if (cmp > 0) {
        x.right = put(x.right, key, val);
    } else {
        x.val = val;
    } //if-else
    x.N = size(x.left) + size(x.right) + 1;
    return x;
} //put
Binary Search Tree – Retrieval

Value get(Node x, Key key) {
    if (x == null) {
        return null;
    } //if
    int cmp = key.compareTo(x.key);
    if (cmp < 0) {
        return get(x.left, key);
    } else if (cmp > 0) {
        return get(x.right, key);
    } else {
        return x.val;
    } //if-else
} //get
Binary Search Tree – Insertion Visual

- Inserting L
- Search for L ends at this null link
- Create new node
- Reset links and increment counts on the way up
Binary Search Tree – Retrieval Visual

- Successful search for R
  - Black nodes could match the search key
  - R is less than S so look to the left
  - Gray nodes cannot match the search key
  - R is greater than E so look to the right

- Unsuccessful search for T
  - T is greater than S so look to the right
  - T is less than X so look to the left
  - Link is null so T is not in tree (search miss)

- Found R (search hit) so return value
Binary Search Tree – Analysis

• Is the shape of a BST independent of input?
  – Best case – tree is balanced
  – Average case – tree is sort of balanced, sort of unbalanced
  – Worst case – tree is a linked list
**Theorem:** Search hits in a BST built from N random keys require $\sim 2 \log(N)$ compares in the average case.

**Proof:** Sort of the same proof from Mergesort, based on an extra observation.

**Corollary:** Search misses and insertions also require $\sim 2 \log(N)$ compares in the average case, because they take 1 more compare, which does not change the complexity class.
Binary Search Tree – Analysis

- Let $C(N)$ be the \textbf{internal path length} of a BST for a search operation. Thus, the average cost of each search hit is $1 + \frac{C(N)}{N}$. We know that $C(0) = C(1) = 1$.
- For $N > 1$, $C(N) = (N - 1) + \frac{C(0)+C(N-1)}{N} + \frac{C(1)+C(N-2)}{N} + \ldots + \frac{C(N-1)+C(0)}{N}$.
- Because all heights are equally likely, the average case will be the same as the best case. This lets us simplify: $C(N) = 2\left(C(N - 1)\right) + (N - 1)$.
- We know that this pattern continues: $C(N - 1) = 2\left(C(N - 2)\right) + (N - 1)$.
- Thus, $C(N) = 2\left(2\left(C(N - 2)\right) + (N - 1)\right) + (N - 1)$.
- Simplify: $C(N) = 4\left(C(N - 2)\right) + 2(N - 2) + (N - 1)$.
- We know that this pattern continues: $C(N - 2) = 2\left(C(N - 3)\right) + (N - 3)$.
- Thus, $C(N) = 4\left(C(N - 3) + (N - 3)\right) + 2(N - 2) + (N - 1)$.
- Simplify: $C(N) = 8\left(C(N - 3)\right) + 4(N - 3) + 2(N - 2) + (N - 1)$.
- A pattern emerges: $C(N) = 2^k\left(C(N - k)\right) + \sum_k\left(2^{k-1}\right)(N - k)$.
- Now, consider the case where $k = N$.
- Here, $C(N) = 2^N\left(C(0)\right) + [(2^{N-1})(0) + (2^{N-2})(1) + \ldots + (2^0)(N)]$.
- Simplify: $C(N) = 2^N + (2^N)N$.
- Log: $C(N) = \log(2^N) + \log((2^N)N) = N + N \times \log(N) = O(N \times \log(N))$. ■
Binary Search Tree – Analysis

- Sequential Search – 2246 compares
- Binary Search – 484 compares
- Binary Search Tree – 13.9 compares
Binary Search Tree – Analysis

- Sequential Search – 2246 compares
- Binary Search – 484 compares
- Binary Search Tree – 13.9 compares

<table>
<thead>
<tr>
<th></th>
<th>tale.txt</th>
<th>leipzig1M.txt</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>words</td>
<td>distinct</td>
</tr>
<tr>
<td></td>
<td>model</td>
<td>actual</td>
</tr>
<tr>
<td>all words</td>
<td>135,635</td>
<td>10,679</td>
</tr>
<tr>
<td>8+ letters</td>
<td>14,350</td>
<td>5,737</td>
</tr>
<tr>
<td>10+ letters</td>
<td>4,582</td>
<td>2,260</td>
</tr>
</tbody>
</table>
Order-Based Methods

- How can we find the min/max in a BST?

```java
min() {
    return min(root).key;
} //min

min(Node x) {
    if (x.left == null) {
        return x;
    } //if
    return min(x.left);
} //min
```
Order-Based Methods

- How about floor/ceiling?

```java
floor(Key key) {
    Node x = floor(root, key);
    if (x == null) {
        return null;
    } //if
    return x.key;
} //floor

floor(Node x, Key key) {
    if (x == null) {
        return null;
    } //if
    int cmp = key.compareTo(x.key);
    if (cmp == 0) {
        return x;
    } if (cmp < 0) {
        return floor(x.left, key);
    } //if-else
    Node t = floor(x.right, key);
    if (t != null) {
        return t;
    } else {
        return x;
    } //if-else
} //min
```
Order-Based Methods

• Selection?

```java
select(int k) {
    return select(root, k).key;
} //select

select(Node x, int k) {
    if (x == null) {
        return null;
    } //if

    int t = size(x.left);
    if (t > k) {
        return select(x.left, k);
    } else if (t < k) {
        return select(x.right, k - t - 1);
    } else {
        return x;
    } //if-else
} //select
```
Order-Based Methods

• Rank?

```java
rank(Key key) {
    return rank(key, root);
} //rank

rank (Key key, Node x) {
    if (x == null) {
        return 0;
    } //if

    int cmp = key.compareTo(x.key);
    if (cmp < 0) {
        return rank(key, x.left);
    } else if (cmp > 0) {
        return 1+size(x.left)+rank(key, x.right);
    } else {
        return size(x.left);
    } //if-else
} //rank
```
The Tough One – Delete

delete(Key key) {
    root = delete(root, key);
} //delete

delete(Node x, Key key) {
    if (x == null) { return null; }
    int cmp = key.compareTo(x.key);
    if (cmp < 0) { x.left = delete(x.left, key); }
    if (cmp > 0) { x.right = delete(x.right, key); }
    else {
        if (x.right == null) { return x.left; }
        if (x.left == null) { return x.right; }
        Node t = x;
        x = min(t.right);
        x.right = deleteMin(t.right);
        x.left = t.left;
    } //if-else
    x.N = size(x.left) + size(x.right) + 1;
    return x;
} //delete
The Tough One – Delete

defeatMin(Node x) {
    if (x.left == null) {
        return x.right;
    } //if
    x.left = defeatMin(x.left);
    x.N = size(x.left) + size(x.right) + 1;
    return x;
} //delete
Conclusions

<table>
<thead>
<tr>
<th>Algorithm (data structure)</th>
<th>Worst-case cost (after N inserts)</th>
<th>Average-case cost (after N random inserts)</th>
<th>Efficiently support ordered operations?</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>search</td>
<td>insert</td>
<td>search hit</td>
</tr>
<tr>
<td>Sequential search (unordered linked list)</td>
<td>N</td>
<td>N</td>
<td>N/2</td>
</tr>
<tr>
<td>Binary search (ordered array)</td>
<td>lg N</td>
<td>N</td>
<td>lg N</td>
</tr>
<tr>
<td>Binary tree search (BST)</td>
<td>N</td>
<td>N</td>
<td>1.39 lg N</td>
</tr>
</tbody>
</table>
Any Questions?