CMPSC250
Lecture 8: Symbol Tables and Basic Search

Prof. John Wenskovitch
02/05/2015
Last Time

• Online data acquisition and priority queues
• Heap implementations, \texttt{sink()} and \texttt{swim()}
• Heapsort and evaluation
Symbol Tables

• **Symbol Table** – A data structure for key-value pairs, associating a key with some value info.
  
  – `put(Key, Value)` – Insert a (key,value) pair into the data structure
  
  – `get(Key)` – Retrieve the value paired with the given key

  – Dictionary (key = word, value = definition)
  – Index (key = term, value = relevant page numbers)
  – Web search (key = search term, value = results)
Symbol Table Rules

• No duplicate keys – each key will be associated with one and only one value (think of it like an array index only holding one item (associative array abstraction))
• No null keys – results in a runtime exception
• No null values – a key without a value should automatically get the value null; no key should specifically be assigned the value null
• Deletion – two styles:
  – Lazy deletion – wipe the value with null
  – Eager deletion – delete the (key,value) pair completely
Ordered Symbol Tables

• Keys can be Comparable objects, so they can be sorted. This lets us perform a number of operations on the symbol table:
  – Min/max – Smallest and largest keys
  – Floor/ceiling – largest key less than or smallest key greater than
  – Rank/selection – number of keys less than a given key, or find the key with a given rank
  – Check key equality
# Ordered Symbol Tables

```java
public class ST<Key extends Comparable<Key>, Value> {

    ST() // create an ordered symbol table
    void put(Key key, Value val) // put key-value pair into the table
        // (remove key from table if value is null)
    Value get(Key key) // value paired with key
        // (null if key is absent)
    void delete(Key key) // remove key (and its value) from table
    boolean contains(Key key) // is there a value paired with key?
    boolean isEmpty() // is the table empty?
    int size() // number of key-value pairs
    Key min() // smallest key
    Key max() // largest key
    Key floor(Key key) // largest key less than or equal to key
    Key ceiling(Key key) // smallest key greater than or equal to key
    int rank(Key key) // number of keys less than key
    Key select(int k) // key of rank k
    void deleteMin() // delete smallest key
    void deleteMax() // delete largest key
    int size(Key lo, Key hi) // number of keys in [lo..hi]
    Iterable<Key> keys(Key lo, Key hi) // keys in [lo..hi], in sorted order
    Iterable<Key> keys() // all keys in the table, in sorted order
}
```
Sequential Search

put(Key key, Value val) {
    for (Node x = first, x != null, x = x.next()) {
        x.val = val;
        return;
    } //for
    first = new Node(key, val, first);
} //put

get(Key key) {
    for (Node x = first, x != null; x = x.next()) {
        if (key.equals(x.key)) {
            return x.val;
        } //if
    } //for
} //get
Sequential Search Visual
Sequential Search Evaluation

• **Theorem:** A search miss and insertion in a sequential search through a symbol table with $N$ key-value pairs requires $N$ compares.

• **Proof:** When searching for a key, we need to test every key in the table against the search key. We must do this full search before each insertion because of our policy of disallowing duplicate keys.

• **Corollary 1:** A search hit (not requiring an insertion) requires $\frac{N}{2}$ compares.

• **Corollary 2:** Inserting $N$ distinct keys into an initially empty symbol table uses $\sim \frac{N^2}{2}$ compares.
Sequential Search Evaluation

![Graph showing sequential search evaluation](image)
Sequential Search Evaluation

• Pros
  – Trivial to implement

• Cons
  – Too slow for very large problems
    • Total number of compares for Leipzig is \( \sim 10^{14} \)
      (roughly \#searches \* \#inserts)
Binary Search

rank(Key key) {
    int lo = 0;
    int hi = N-1;
    while (lo <= hi) {
        int mid = lo + (hi - lo) / 2;
        int cmp = key.compareTo(keys[mid]);
        if (cmp < 0) {
            hi = mid - 1;
        } else if (cmp > 0) {
            lo = mid + 1;
        } else {
            return mid;
        } //if-else
    } //while
    return lo;
} //rank
Binary Search

```java
put(Key key, Value val) {
    int i = rank(key);
    if (i < N && keys[i].compareTo(key) == 0) {
        vals[i] = val;
        return;
    } //if
    for (int j = N; j > i; j--) {
        keys[j] = keys[j-1];
        vals[j] = vals[j-1];
    } //for
    keys[i] = key;
    vals[i] = val;
    N++;
} //put

get(Key key) {
    if (isEmpty()) return null;
    int i = rank(key);
    if (i < N && keys[i].compareTo(key) == 0) {
        return vals[i];
    } else {
        return null;
    } //if-else
} //get
```
## 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>6</td>
</tr>
</tbody>
</table>

**keys[]**

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>C</td>
<td>E</td>
<td>H</td>
<td>L</td>
<td>M</td>
<td>P</td>
<td>R</td>
<td>S</td>
<td>X</td>
</tr>
</tbody>
</table>

- entries in black are `a[lo..hi]`
- entry in red is `a[mid]`
- loop exits with `keys[mid] = P`: return

### Unsuccessful 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>6</td>
</tr>
<tr>
<td>7</td>
<td>6</td>
<td>6</td>
</tr>
</tbody>
</table>

**keys[]**

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>C</td>
<td>E</td>
<td>H</td>
<td>L</td>
<td>M</td>
<td>P</td>
<td>R</td>
<td>S</td>
<td>X</td>
</tr>
</tbody>
</table>

- 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:** Well... same proof.

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

![Graph showing cost vs. operations](image)

- Cost vs. Operations
- Cost values range from 0 to 5737
- Operations range from 0 to 14350
- The graph illustrates the efficiency of binary search with a linear cost increase relative to operations.
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 $\sim 10^{11}$
    • Binary Search reduces the number of compares, but not necessarily the overall running time, because the `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><strong>sequential search</strong></td>
<td>N</td>
<td>N/2</td>
<td>no</td>
</tr>
<tr>
<td><strong>binary search</strong></td>
<td>lg N</td>
<td>lg N</td>
<td>yes</td>
</tr>
</tbody>
</table>

- **sequential search (unordered linked list):**
  - Search: $N$
  - Insert: $N$
  - Search hit: $N/2$
  - Insert: $N$
  - Efficiently support ordered operations: no

- **binary search (ordered array):**
  - Search: $lg N$
  - Insert: $2N$
  - Search hit: $lg N$
  - Insert: $N$
  - Efficiently support ordered operations: yes
Any Questions?