26–27: HashTables

David Kauchak  Alexandra Papoutsaki
### Summary for dictionary/symbol table operations

<table>
<thead>
<tr>
<th></th>
<th>Worst case</th>
<th></th>
<th></th>
<th>Average case</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Search</td>
<td>Insert</td>
<td>Delete</td>
<td>Search</td>
<td>Insert</td>
<td>Delete</td>
</tr>
<tr>
<td>BST</td>
<td>$n$</td>
<td>$n$</td>
<td>$n$</td>
<td>$\log n$</td>
<td>$\log n$</td>
<td>$\log n$</td>
</tr>
<tr>
<td>2-3 search</td>
<td>$\log n$</td>
<td>$\log n$</td>
<td>$\log n$</td>
<td>$\log n$</td>
<td>$\log n$</td>
<td>$\log n$</td>
</tr>
</tbody>
</table>
Lecture 26-27: Hash tables

- Hash functions
- Separate chaining
- Linear Probing
Basic plan for implementing dictionaries using hashing

- **Goal**: Build a key-indexed array (table or hash table or hash map) to model dictionaries (or symbol tables) for efficient \( O(1) \) search.

- **Hash function**: Method for computing array index (hash value) from key.
  - \( \text{hash(“Texas”) = 2} \)
  - \( \text{hash(“California”) = 2} \)

- **Issues**:
  - Computing the hash function.
  - Method for checking whether two keys are equal.
  - How to handle collisions when two keys hash to same index.
Computing hash function

- **Ideal scenario**: Take any key and uniformly “scramble” it to produce a symbol table/dictionary index.

- **Requirements**:
  - Consistent - equal keys must produce the same hash value.
  - Efficient - quick computation of hash value.
  - Uniform distribution - every index is equally likely for each key.

- Although thoroughly researched, still problematic in practical applications.

- **Examples**: Dictionary where keys are social security numbers.
  - Bad: if we choose the first three digits (geographical region and time).
  - Better: if we choose the last three digits.
  - Best: use all data.

- **Practical challenge**: Need different approach for each key type.
Hashing in Java

- All Java classes inherit a method `hashCode()`, which returns an integer.

- Requirement: If `x.equals(y)` then it should be `x.hashCode()==y.hashCode()`.

- Ideally: If `!x.equals(y)` then it should be `x.hashCode()!=y.hashCode()`.

- Default implementation: Memory address of `x`.
  - Need to override both `equals()` and `hashCode()` for custom types.
  - Already done for us for `Integer`, `Double`, etc.
Equality test in Java

- **Requirement**: For any objects \( x, y, \) and \( z \).
  - **Reflexive**: \( x.equals(x) \) is true.
  - **Symmetric**: \( x.equals(y) \) iff \( y.equals(x) \).
  - **Transitive**: if \( x.equals(y) \) and \( y.equals(z) \) then \( x.equals(z) \).
  - **Non-null**: if \( x.equals(null) \) is false.

- If you don’t override it, the default implementation checks whether \( x \) and \( y \) refer to the same object in memory.
Java implementations of equals() for user-defined types

```java
date public class Date {
    private int month;
    private int day;
    private int year;
    ...
    public boolean equals(Object y) {
        if (y == this) return true;
        if (y == null) return false;
        if (y.getClass() != this.getClass()) return false;
        Date that = (Date) y;
        return (this.day == that.day &&
                this.month == that.month &&
                this.year == that.year);
    }
}
```
General equality test recipe in Java

- Optimization for reference equality.
  - if (y == this) return true;

- Check against null.
  - if (y == null) return false;

- Check that two objects are of the same type.
  - if (y.getClass() != this.getClass()) return false;

- Cast them.
  - Date that = (Date) y;

- Compare each significant field.
  - return (this.day == that.day && this.month == that.month && this.year == that.year);

  - If a field is a primitive type, use ==.
  - If a field is an object, use equals().
  - If field is an array of primitives, use Arrays.equals().
  - If field is an area of objects, use Arrays.deepEquals().
Java implementations of hashCode()

```java
// public final class Integer {
private final int value;
...
public int hashCode() {
    return (value);
}
}
```

```java
// public final class Boolean {
private final boolean value;
...
public int hashCode() {
    if (value) return 1231;
    else return 1237;
}
}
```
Java implementations of hashCode() for user-defined types

```java
public class Date {
    private int month;
    private int day;
    private int year;

    ... public int hashCode() {
            int hash = 1;
            hash = 31*hash + ((Integer) month).hashCode();
            hash = 31*hash + ((Integer) day).hashCode();
            hash = 31*hash + ((Integer) year).hashCode();
            return hash;
            // could be also written as
            // return Objects.hash(month, day, year);
    }
}
```
General hash code recipe in Java

- Combine each significant field using the $31x+y$ rule.
- Shortcut 1: use `Objects.hash()` for all fields (except arrays).
- Shortcut 2: use `Arrays.hashCode()` for primitive arrays.
- Shortcut 3: use `Arrays.deepHashCode()` for object arrays.
Modular hashing

- **Hash code**: an `int` between $-2^{31}$ and $2^{31} - 1$

- **Hash function**: an `int` between 0 and $m - 1$, where $m$ is the hash table size (typically a prime number or power of 2).

The class that implements the dictionary of size $m$ should implement a hash function. Examples:

  - ```java
      private int hash (Key key){
          return key.hashCode() % m;
      }
      ```
      
      - Bug! Might map to negative number.

  - ```java
      private int hash (Key key){
          return Math.abs(key.hashCode()) % m;
      }
      ```
      
      - Very unlikely bug. For a hash code of $-2^{31}$, `Math.abs` will return a negative number!

  - ```java
      private int hash (Key key){
          return (key.hashCode() & 0x7fffffff) % m;
      }
      ```
      
      - Correct.
Uniform hashing assumption

- Uniform hashing assumption: Each key is equally likely to hash to an integer between 0 and \( m - 1 \).
- Mathematical model: balls & bins. Toss \( n \) balls uniformly at random into \( m \) bins.
- Bad news: Expect two balls in the same bin after \( \sim \sqrt{\frac{\pi m}{2}} \) tosses.
  - Birthday problem: In a random group of 23 or more people, more likely than not that two people will share the same birthday.
- Good news: load balancing
  - When \( n > > m \), the number of balls in each bin is “likely close” to \( n/m \).
Lecture 26-27: Hash tables

- Hash functions
- Separate chaining
- Linear Probing
Separate Chaining

- Use an array of $m < n$ distinct lists [H.P. Luhn, IBM 1953].
  - **Hash**: Map key to integer $i$ between 0 and $m-1$.
  - **Insert**: Put at front of i-th chain (if not already there).
  - **Search**: Need to only search the i-th chain.
public class SeparateChainingLiteHashST<Key, Value> {

    private int m = 128; // hash table size or number of chains
    private Node[] st = new Node[m]; // array of chains. Node is inner class that holds keys and values of type Object and a reference to the next Node

    public Value get(Key key) {
        int i = hash(key);
        for (Node x = st[i]; x != null; x = x.next;)
            if (key.equals(x.key)) return (Value) x.val;
        return null;
    }

    public void put(Key key, Value val) {
        int i = hash(key);
        for (Node x = st[i]; x != null; x = x.next;)
            if (key.equals(x.key)) {
                x.val = val;
                return;
            }
        st[i] = new Node(key, val, st[i];
    }

    private int hash(Key key) { return (key.hashCode() & 0x7fffffff) % m; }
}
Analysis

- Under uniform hashing assumption, length of each chain is $\sim n/m$.

- **Consequence:** Number of probes (calls to either `equals()` or `hashCode()`) for search/insert is proportional to $n/m$ ($m$ times faster than sequential search).

  - $m$ too large -> too many empty chains.

  - $m$ too small -> chains too long.

  - Typical choice: $m \sim 1/4n$ -> constant time per operation.
Resizing in a separate-chaining hash table

- **Goal**: Average length of chain $n/m = \text{constant lookup}$.
  - Double hash table size when $n/m \geq 8$.
  - Halve hash table size when $n/m \leq 2$.
  - Need to rehash all keys when resizing (hashCode value does not change, but hash value changes).
Parting thoughts about separate-chaining

- **Deletion**: Easy! Hash key, find its chain, search for a node that contains it and remove it.

- **Ordered operations**: not supported. Look into (balanced) BSTs.

- Fastest and most widely used dictionary implementation for applications where key is not important.
Lecture 26-27: Hash tables

- Hash functions
- Separate chaining
- Linear Probing
Linear Probing

- Belongs in the open addressing family.
- Alternate approach to handle collisions when \( m > n \).
- Maintain keys and values in two parallel arrays.
- When a new key collides, find next empty slot and put it there.
- If the array is full, the search would not terminate.
Linear probing

- **Hash**: Map key to integer $i$ between 0 and $m - 1$.

- **Insert**: Put at index $i$ if free. If not, try $i + 1$, $i + 2$, etc.

- **Search**: Search table index $i$. If occupied but no match, try $i + 1$, $i + 2$, etc.
  - If you find a gap then you know that it does not exist.

- Table size $m$ **must** be greater than the number of key-value pairs $n$. 
3.4 Linear Probing Demo
Linear probing
Symbol table with linear probing implementation

```java
public class LinearProbingHashST<Key, Value> {

    private int m = 32768;  // hash table size
    private Value[] Vals = (Value[]) new Object[m];
    private Key[] Vals = (Key[]) new Object[m];

    public Value get(Key key) {
        for (int i = hash(key); keys[i] != null; i = (i+1) % m;)
            if (key.equals(keys[i])) return vals[i];
        return null;
    }

    public void put(Key key, Value val) {
        int i;
        for (int i = hash(key); keys[i] != null; i = (i+1) % m;)
            if (key.equals(keys[i])){
                break;
            }
        keys[i] = key;
        vals[i] = val;
    }

```
Clustering

- **Cluster**: a contiguous block of keys.
- **Observation**: new keys likely to hash in middle of big clusters.
Analysis

- **Proposition**: Under uniform hashing assumption, the average number of probes in a linear-probing hash table of size $m$ that contains $n = \alpha m$ keys is at most

  - $1/2(1 + \frac{1}{1 - \alpha})$ for search hits and
  - $1/2(1 + \frac{1}{(1 - \alpha)^2})$ for search misses and insertions.

- [Knuth 1963]

- **Parameters**:
  
  - $m$ too large -> too many empty array entries.
  
  - $m$ too small -> search time becomes too long.

  Typical choice for **load factor**: $\alpha = n/m \sim 1/2$ -> constant time per operation.
Resizing in a linear probing hash table

- **Goal**: Fullness of array (load factor) \( \frac{n}{m} \leq \frac{1}{2} \).

  - Double hash table size when \( \frac{n}{m} \geq \frac{1}{2} \).

  - Halve hash table size when \( \frac{n}{m} \leq \frac{1}{8} \).

- Need to rehash all keys when resizing (hash code does not change, but hash changes).

- Deletion not straightforward.
## Summary for dictionary/symbol table operations

<table>
<thead>
<tr>
<th></th>
<th>Worst case</th>
<th></th>
<th></th>
<th>Average case</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Search</td>
<td>Insert</td>
<td>Delete</td>
<td>Search</td>
<td>Insert</td>
<td>Delete</td>
</tr>
<tr>
<td>BST</td>
<td>$n$</td>
<td>$n$</td>
<td>$n$</td>
<td>$\log n$</td>
<td>$\log n$</td>
<td>$\log n$</td>
</tr>
<tr>
<td>2-3 search tree</td>
<td>$\log n$</td>
<td>$\log n$</td>
<td>$\log n$</td>
<td>$\log n$</td>
<td>$\log n$</td>
<td>$\log n$</td>
</tr>
<tr>
<td>Separate chaining</td>
<td>$n$</td>
<td>$n$</td>
<td>$n$</td>
<td>$1$</td>
<td>$1$</td>
<td>$1$</td>
</tr>
<tr>
<td>Linear probing</td>
<td>$n$</td>
<td>$n$</td>
<td>$n$</td>
<td>$1$</td>
<td>$1$</td>
<td>$1$</td>
</tr>
</tbody>
</table>
Hash tables vs balanced search trees

- **Hash tables:**
  - Simpler to code.
  - No effective alternative of unordered keys.
  - Faster for simple keys (a few arithmetic operations versus $\log n$ compares).

- **Balanced search trees:**
  - Stronger performance guarantee.
  - Support for ordered symbol table operations.
  - Easier to implement `compareTo()` than `hashCode()`.

- **Java includes both:**
Lecture 26-27: Hash tables

- Hash functions
- Separate chaining
- Linear Probing
Readings:

- Textbook: Chapter 3.4 (Pages 458-477)
- Website:
  - https://algs4.cs.princeton.edu/34hash/
- Visualization:
  - https://visualgo.net/en/hashtable

Practice Problems:

- 3.4.1-3.4.13