Some slides adopted from Princeton CS226 course or Algorithms, 4th Edition
Lecture 31-32: Hash tables

- Hash functions
- Separate chaining
- Linear Probing

Some slides adopted from Algorithms 4th Edition or COS226
## Summary for 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>Sequential search</td>
<td>$n$</td>
<td>$n$</td>
<td>$n$</td>
<td>$n/2$</td>
<td>$n$</td>
<td>$n/2$</td>
</tr>
<tr>
<td>(unordered)</td>
<td>(unordered)</td>
<td>(unordered)</td>
<td>(unordered)</td>
<td>(unordered)</td>
<td>(unordered)</td>
<td>(unordered)</td>
</tr>
<tr>
<td>Binary search</td>
<td>$\log n$</td>
<td>$n$</td>
<td>$n$</td>
<td>$\log n$</td>
<td>$n/2$</td>
<td>$n/2$</td>
</tr>
<tr>
<td>(ordered array)</td>
<td>(ordered)</td>
<td>(ordered)</td>
<td>(ordered)</td>
<td>(ordered)</td>
<td>(ordered)</td>
<td>(ordered)</td>
</tr>
<tr>
<td>BST</td>
<td>$n$</td>
<td>$n$</td>
<td>$n$</td>
<td>$1.39 \log n$</td>
<td>$1.39 \log n$</td>
<td>?</td>
</tr>
<tr>
<td>2-3 search tree</td>
<td>$c \log n$</td>
<td>$c \log n$</td>
<td>$c \log n$</td>
<td>$c \log n$</td>
<td>$c \log n$</td>
<td>$c \log n$</td>
</tr>
<tr>
<td>Red-black BSTs</td>
<td>$2 \log n$</td>
<td>$2 \log n$</td>
<td>$2 \log n$</td>
<td>$1 \log n$</td>
<td>$1 \log n$</td>
<td>$1 \log n$</td>
</tr>
</tbody>
</table>
Basic plan for hashing

- Save items in a key-indexed table (index is a function of the key).

- **Hash function**: Method for computing array index from key.
  - `hash(“A”) = 2`
  - `hash(“B”) = 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.

- **Space-time tradeoff**:
  - If no space limitation: hash function with key as index.
  - If no time limitation: collision resolution with sequential search.
  - If space and time limitation (real world): hashing
Computing hash function

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

- **Requirements**:
  - Computing the hash function efficiently.
  - Every symbol table index is equally likely for each key.

- Although thoroughly researched, still problematic in practical applications.

- **Examples**: Hashing phone numbers or social security numbers.
  - Bad: if we choose the first three digits (area code/geographic region and time).
  - Better: if we choose the last three digits.

- **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(x)` then it should be `x.hashCode()==y.hashCode()`.

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

- **Default implementation**: Memory address of `x`.
  
  - Need to override it 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 \( x.equals(x) \) 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

- `public final class Date {
  private final int month;
  private final int day;
  private final 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 array of objects, use Arrays.deepEquals().
Java implementations of hashCode()
Implementing hash code for arrays

- **31x+y rule.**
  - Initialize hash to 1.
  - Repeatedly multiply hash by 31 and add next integer in array.
- `public class Arrays {
  ...
  public static int hashCode(int[] a) {
    if (a == null)
      int hash = 1;
    for (int i=0; i<a.length; i++) {
      hash = 31*hash + a[i];
    }
    return hash;
  }
}
Implementing hash code for strings

- Treat a string as an array of characters.
  - Initialize hash to 1.
  - public final class String {
    private final char[] s;
    private int hash = 0;
    ...
    public int hashCode() {
      int h = hash;
      if (h != 0) return h;
      for (int i=0; i< length; i++) {
        h = s[i] + (31 * h);
        hash = h;
        return h;
      }
    }
  }

- Not foolproof, e.g., both Aa and BB hash to 2112. Actually, \(2^n\) strings of length \(2n\) hash to the same value!
Java implementations of `equals()` for user-defined types

```java
public final class Date {
    private final int month;
    private final int day;
    private final int year;
    ...
    public boolean 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 of power of 2).

```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 $\sim \sqrt{(\pi m/2)}$ 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 $n = m$, expect most loaded bin has $\sim \ln m / \ln \ln n$ balls.
  - When $n \gg m$, the number of balls in each bin is “likely close” to $n/m$. 
Lecture 31-32: Hash tables

- Hash functions
- Separate chaining
- Linear Probing
Collisions are unavoidable

- **Collision**: Two distinct keys hash to the same index.

- **Birthday problem**: Can’t avoid collisions (unless you have at least quadratic memory).

- **Coupon collector + load balancing**: collisions will be evenly distributed.

- **Challenge**: how to deal with collisions efficiently.

  - hash(“A”) = 2
  - hash(“B”) = 2 ???
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.
Symbol table with separate chaining implementation

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

    private int m = 128;  // hash table size
    private Node[] st = new Node[m];
    // array of linked-list symbol tables. Node is inner class that holds keys and values of type Object

    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) {
        for (i = hash(key);
            st[i] = new Node(key, val, st[i];
```
Analysis

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

- **Consequence:** Number of probes (calls to either `equals()` or `hashCode()`) for search/insert is proportional to $\frac{n}{m}$ ($m$ times faster than sequential search).
  - $m$ too large $\rightarrow$ too many empty chains.
  - $m$ too small $\rightarrow$ chains too long.

- Typical choice: $m \sim \frac{1}{4n}$ $\rightarrow$ 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 (hash code does not change, but hash changes).
Deletion in a separate-chaining hash table

- Find key in chain and remove it along with its associated value.
## Summary for 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>Sequential search</td>
<td>( n )</td>
<td>( n )</td>
<td>( n )</td>
<td>( \frac{n}{2} )</td>
<td>( n )</td>
<td>( \frac{n}{2} )</td>
</tr>
<tr>
<td>(unordered list)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Binary search</td>
<td>( \log n )</td>
<td>( n )</td>
<td>( n )</td>
<td>( \log n )</td>
<td>( \frac{n}{2} )</td>
<td>( \frac{n}{2} )</td>
</tr>
<tr>
<td>(ordered array)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BST</td>
<td>( n )</td>
<td>( n )</td>
<td>( n )</td>
<td>( 1.39 \log n )</td>
<td>( 1.39 \log n )</td>
<td>?</td>
</tr>
<tr>
<td>2-3 search tree</td>
<td>( c \log n )</td>
<td>( c \log n )</td>
<td>( c \log n )</td>
<td>( c \log n )</td>
<td>( c \log n )</td>
<td>( c \log n )</td>
</tr>
<tr>
<td>Red-black BSTs</td>
<td>( 2 \log n )</td>
<td>( 2 \log n )</td>
<td>( 2 \log n )</td>
<td>( 1 \log n )</td>
<td>( 1 \log n )</td>
<td>( 1 \log n )</td>
</tr>
<tr>
<td>Separate chaining</td>
<td>( n )</td>
<td>( n )</td>
<td>( n )</td>
<td>( 3 - 5 )</td>
<td>( 3 - 5 )</td>
<td>( 3 - 5 )</td>
</tr>
</tbody>
</table>
Lecture 31-32: Hash tables

- Hash functions
- Separate chaining
- Linear Probing
Open addressing

- Alternate approach to handle collisions.
- 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

Trace of linear-probing ST implementation for standard indexing client
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

  \[ \frac{1}{2}(1 + \frac{1}{1 - \alpha}) \] for search hits and

  \[ \frac{1}{2}(1 + \frac{1}{(1 - \alpha)^2}) \] 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**: $\alpha = n/m^{1/2}$ -> constant time per operation.
Resizing in a linear probing hash table

- **Goal**: Fullness of array (load factor) \( n/m \leq 1/2 \).
  - Double hash table size when \( n/m \geq 1/2 \).
  - Halve hash table size when \( n/m \leq 1/8 \).
  - Need to rehash all keys when resizing (hash code does not change, but hash changes).
  - Deletion not straightforward.
## Summary for 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>Sequential search</td>
<td>$n$</td>
<td>$n$</td>
<td>$n$</td>
<td>$n/2$</td>
<td>$n$</td>
<td>$n/2$</td>
</tr>
<tr>
<td>(unordered list)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Binary search</td>
<td>$\log n$</td>
<td>$n$</td>
<td>$n$</td>
<td>$\log n$</td>
<td>$n/2$</td>
<td>$n/2$</td>
</tr>
<tr>
<td>(ordered array)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BST</td>
<td>$n$</td>
<td>$n$</td>
<td>$n$</td>
<td>$1.39 \log n$</td>
<td>$1.39 \log n$</td>
<td>?</td>
</tr>
<tr>
<td>2-3 search tree</td>
<td>$c \log n$</td>
<td>$c \log n$</td>
<td>$c \log n$</td>
<td>$c \log n$</td>
<td>$c \log n$</td>
<td>$c \log n$</td>
</tr>
<tr>
<td>Red-black BSTs</td>
<td>$2 \log n$</td>
<td>$2 \log n$</td>
<td>$2 \log n$</td>
<td>$1 \log n$</td>
<td>$1 \log n$</td>
<td>$1 \log n$</td>
</tr>
<tr>
<td>Separate chaining</td>
<td>$n$</td>
<td>$n$</td>
<td>$n$</td>
<td>$3 - 5$</td>
<td>$3 - 5$</td>
<td>$3 - 5$</td>
</tr>
<tr>
<td>Linear probing</td>
<td>$n$</td>
<td>$n$</td>
<td>$n$</td>
<td>$2 - 3$</td>
<td>$2 - 3$</td>
<td>$2 - 3$</td>
</tr>
</tbody>
</table>
Separate chaining vs linear probing

- **Separate chaining:**
  - Performance degrades gracefully as number of keys increases.
  - Clustering less sensitive to poorly-designed hash function.
    - Potentially fewer probes.

- **Linear probing:**
  - Less wasted space.
  - Better cache performance (locality).
Hashing: variations on the theme

- **Two-probe hashing** (separate chaining variant):
  - Hash to two positions, insert key in shorter of the two chains.
  - Reduces expected length of longest chain to $\log \log n$

- **Double hashing** (linear probing variant):
  - Use linear probing, but skip a variable amount, not just 1 each time you have collision.
  - Effectively eliminates clustering.
  - Can allow table to become nearly full.
  - More difficult to implement delete.

- **Cuckoo hashing** (linear probing variant):
  - Hash to two positions, insert key into either position. If occupied, reinsert displayed key into its alternative position and recur.
  - Constant worst case time for search.
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 31-32: Hash tables

- Hash functions
- Separate chaining
- Linear Probing
Readings:

- Textbook: Chapter 3.4 (Pages 458-477)
- Website:
  - [https://algs4.cs.princeton.edu/34hash/](https://algs4.cs.princeton.edu/34hash/)

Practice Problems:

- 3.4.1-3.4.13