Hashing

It’s not just for breakfast anymore!
Hashing: the facts

• Approach that involves both storing and searching for values (search/sort combination)
• Behavior is linear in the worst case, but strong competitor with binary searching in the average case
• Hashing makes it easy to add and delete elements, an advantage over binary search (since the latter requires sorted array)
Dictionary ADT

• Abstract properties of dictionary:
  – every item has a key
  – to retrieve an item, specify key and retrieval process fetches associated data

• A hash table can be used to provide an array-based dictionary implementation
Setting up the array

• One approach to an array-based dictionary would be to create consecutive keys, storing the records so that each key corresponds to its index -- this is the method used in MS Access, for example.

• An alternative would be to use an existing attribute of the data to be stored as the key value; this approach is more typical of hashing.
Setting up the array

• Use of existing key field presents challenges:
  – Value may be too large for indexing: e.g. social security number
  – No guarantee that individual values will be close enough together for effective indexing: e.g. last 4 digits of social security numbers of students in a class
Solution: hashing

• Instead of direct use of data field, a function is applied to the original value to produce a valid index: this is called the hash function.

• The hash function maps the key to an index that can be used to insert data into the array or to retrieve data based on a given key.

• An array that uses hashing for indexing is called a hash table.
Operations on a hash table

• Inserting an item
  – calculate hash value (index) from item key
  – check index to determine if space is open
    • if open, insert item
    • if not open, collision occurs; search through array for next open slot

• The insertion scheme just described uses open-address hashing
Operations on a hash table

• Retrieving an item
  – calculate hash value based on desired key
  – search array, beginning at calculated index, for desired data
  – search is finished when:
    • item is found; successful search
    • an empty index is encountered; unsuccessful search
Operations on a hash table

• Deleting an item:
  – find index based on hashed key, as with insertion and retrieval
  – mark record at index to indicate the spot is open
Hash tables & Java: a breakfast classic

• Java classes have built-in support for hash tables; all classes inherit the hashcode() method from Object
• The hashcode() method returns a pseudorandom number that can be used as input to a hash method
• One caution: if you define an equals() method for a new class, you should also override the hashcode method – this is because equal objects must have equal hashcodes
public class Table {
    private int manyItems;
    private Object[ ] data;
    // hash table data
    private Object[ ] keys;
    // array of keys parallel to data array
    private boolean[ ] hasBeenUsed;
    // another parallel array used to indicate the status
    // of each index – if not currently in use, but has
    // been used previously, an index should not stop
    // a search operation
Constructor

```java
public Table(int capacity) {
    if (capacity <= 0)
        throw new IllegalArgumentException("Capacity is negative");
    keys = new Object[capacity];
    data = new Object[capacity];
    hasBeenUsed = new boolean[capacity];
}
```
The put method (adds item to table)

```java
class HashTable {
    public Object put(Object key, Object element) {
        int index = findIndex(key);
        Object answer;
        if (index != -1) { // The key is already in the table.
            answer = data[index];
            data[index] = element;
            return answer;
        }
    }
}
```
put method continued

else if (manyItems < data.length) {  // key is not yet in Table.
    index = hash(key);
    while (keys[index] != null)
        index = nextIndex(index);
    keys[index] = key;
    data[index] = element;
    hasBeenUsed[index] = true;
    manyItems++;
    return null;
}

else {  // The table is full.
    throw new IllegalStateException("Table is full.");
}

}
remove method

public Object remove(Object key) {
    int index = findIndex(key);
    Object answer = null;
    if (index != -1) {
        answer = data[index];
        keys[index] = null;
        data[index] = null;
        manyItems--;
    }
    return answer;
}
findIndex method

private int findIndex(Object key) {
    int count = 0;
    int i = hash(key);
    while (count < data.length && hasBeenUsed[i]) {
        if (key.equals(keys[i]))
            return i;
        count++;
        i = nextIndex(i);
    }
    return -1;
}
nextIndex & containsKey methods

private int nextIndex(int i) {
    if (i+1 == data.length)
        return 0;
    else
        return i+1;
}

public boolean containsKey(Object key) {
    return findIndex(key) != -1;
}
hash method

```java
private int hash(Object key) {
    return Math.abs(key.hashCode()) % data.length;
}
```
Java’s HashTable class

- The Java API contains a HashTable class in the java.util package
- Unlike the implementation just presented, the API HashTable grows automatically when it approaches capacity
- This has important performance issues; when the array grows, the entire table must be rehashed
Hash functions

• Choice of hash function can be important factor in reducing the likelihood of collisions

• Division hashing: key % CAPACITY
  – Certain table sizes are more conducive to collision avoidance with this method
  – A 1970 study suggests that a good size is a prime number of the form 4k+3
    – For example, $811 = 202 \times 4 + 3$)
Other hash functions

• Mid-square hash
  – multiply key by itself
  – use some middle digits of the result as hash value

• Multiplicative hash
  – multiply key by a floating-point constant that is less than one
  – use first few digits of fractional part of result as hash value
Insertion and linear probing

• During insertion process, collision may occur

• In case of collision, the insertion function moves forward through the array until a vacant spot is found; the process is known as *linear probing*
The perils of probing

• When many keys hash to the same index, elements start to group in clumps near that index; as the table grows, these clumps get larger

• As the table approaches capacity, these clumps tend to merge into gigantic clusters; hence this process is known as clustering

• Performance of insertion and search functions degrades when clustering occurs
Double hashing

- A common technique used to reduce clustering is double hashing
- In double hashing, a second hash function is used to determine where to seek the next vacancy in the array when a collision occurs
- Rather than using linear probing, double hashing ensures that a few entries are skipped after each collision
Double hashing

• Step 1: hash key and check for collision
• Step 2: if collision occurs, run key through second hash function; check index result spaces beyond original
• Example:
  – 1st hash produces 206 - space at this index is taken, so run 2nd hash
  – 2nd hash produces 9, so check space at index 215 - if not vacant, go to 224, then 233, etc.
Considerations for second hash function

• Value added to index must not exceed valid range of array (0 .. CAPACITY-1)
• Can stay within range by using the following formula to determine next index (where hash2 is the second hash function):
  \[ \text{index} = (\text{index} + \text{hash2(key)}) \mod \text{CAPACITY} \]
Considerations for second hash function

• Every array position must be examined - with double hashing, spots could be skipped, returning to start position before every available location has been probed

• To avoid this, make sure CAPACITY-1 is relatively prime with respect to value returned by hash2 (in other words, 2nd hash value and last array index should have no common factors)
Example values for double hashing

- Both CAPACITY and CAPACITY-2 should be primes -- e.g. 809 and 811
- First hash function:
  \[
  \text{return } (\text{key} \mod \text{CAPACITY});
  \]
- Second hash function:
  \[
  \text{return } (1 + (\text{key} \mod (\text{CAPACITY} - 2)));
  \]
Modifications to dictionary class for double hashing

• Add a hash2 function as private member
• Change next_index to return
  \[(1 + \text{hash2(key)} \mod \text{CAPACITY})\]
Chained hashing

- Open-address hashing uses arrays in which each element contains one entry; when the array is full, can’t add more entries
- Changing array size requires copying and rehashing entire table
- Chained hashing is a more workable alternative
Chained hashing

• Chained hashing combines both static element (array) and dynamic element (list – for example, and ArrayList)
  – each array element is a list which can hold several entries
  – all records that hash to a particular index are placed in the list at that index
  – a chained hash table can hold many more records than a simple hash table
Time analysis of hashing

- In the worst case, every key gets hashed to the same index -- this makes insertion, deletion and searching linear operations (O(N))
- Best case is the same as the linear search algorithm (O(1)), for the same reason
- Neither of these cases is particularly likely, however
Time analysis of hashing

- Average case is relatively complex, especially if deletions are allowed
- Three different formulas have been developed for the average number of elements that must be examined for a successful search -- each corresponds to a different version of hashing (open-address with linear probing, open-address with double hashing, and chained hashing)
Time analysis of hashing

• Each formula depends on the number of elements in the table
• the greater the number of items, the more collisions
• the more collisions, the longer the average search time
Time analysis of hashing

• A load factor ($\alpha$) is used in each formula --
  $\alpha$ is the ratio of the number of occupied table locations to the size of the table:
  
  $\alpha = \text{used} / \text{CAPACITY}$
Time analysis of hashing

• For open-address hashing with linear probing, given the following conditions:
  – hash table is not full
  – no deletions

• Average number of table elements examined for a successful search is:

\[
\frac{1}{2} * \left(1 + \frac{1}{1 - \alpha}\right)
\]
Time analysis of hashing

- Example of open-address hashing with linear probing:
  
  \[
  \text{used} = 525 \\
  \text{CAPACITY} = 713 \\
  \alpha = \frac{525}{713}, \sim .75
  \]

- therefore, average number of searches is:
  \[
  \frac{1}{2} \times (1 + \frac{1}{1 - .75}) = 2.5
  \]

- meaning about 3 table elements must be examined to complete a successful search
Time analysis of hashing

• Average search time for open addressing with double hashing, given:
  – table is not full
  – no deletions

• Formula:

\[-\ln \left(1 - \alpha \right) / \alpha\]

• For same values as previous example, result is slightly less than 2
Time analysis of hashing

• For chained hashing, conditions for average search time are different:
  – each element of a table is the head pointer of a linked list
  – each list may have several items
    • $\alpha$ may be greater than one
  – formula remains valid even with deletions:
    \[ 1 + \frac{\alpha}{2} \]