Hash Tables
Too Good to be True?

- Constant time for insertion and searching, regardless of how many data items.
- Search/Insertion
  - arrays – O (n)/O(n)
  - linked lists – O (n)/ O(n) or O(1)
  - trees – O(log(n)) / O(log(n))
  - hash tables – O(1)/O(1)

The Problems

- Must have a good idea ahead of time how many data items there will be.
  - hash tables will not be O(1) if too full
  - moving to a larger hash table is VERY costly
- O(1) yes -- but constant term may be large
- No convenient way to visit the data items in order (any order)
  - must do something else

The Idea of Hashing

- Hash tables are based on arrays.
- The idea is to establish a correspondence between a data key value and some array index.
- By looking at the data, we can quickly find out where to store it (insertion), or where it is stored (search).

Simple Case

- When data keys are well organized.
- Consider a company who stores its employee data in a database.
  - Each employee is assigned an ID.
  - IDs run sequentially from 0.
- Hashing is very simple, just use the ID as the array index

```
Person[] empData = new Person[total];
empData[newRecord.id] = newRecord;
```

BUT – suppose that you do not know the id – how do you get it?

A Dictionary

- Store a 50,000-word dictionary in memory.
- No symbols and all lower case letters.
- What is the index for “cat”?
- Let’s first assign integers to the letters.
  - a: 1, b: 2 … z: 26, blank: 27
- How about adding them up and using the sum?

Multiplying by Powers

- Letters ~ digits – words ~ numbers
- “cat” = 'c' + 'a'*26 + 't' *26*26
- java: Object.hashCode() uses 31 not 26
  - the number rapidly gets too big to be an int
- Generates unique index for each word.
- Generates index for words that do not exist.
  - So lots of wasted space even assuming want to store all english words
- Max word len
  - 3 ==> 17576 indices, 4==>30336176 indices
  - assuming only lower case letters

Hashing

- We need to compress the huge range of numbers into what is reasonable.
- A simple approach is to use the modulus operator.
- The remainder is guaranteed to be in the range of the mod operand.

```
arraySize = numWords * 2;
index = hugeNumber % arraySize;
```

Hashing – “good” hugeNumbers

- Additive
  - Let p be an array of prime numbers
    - ```
      long hash=0;  int j=0
      for (int i=0; i<item.length(); i++) {
            hash = p[j++] * item.getChar(i);
            if (j==p.length) j=0; }
      ```
- Rotative
  - do some “bit shifting” after each letter
    - ```
      int hash = 5381;
      for(int i = 0; i < item.length(); i++)
          hash = ((hash/32) + hash) + item.charAt(i);
      ```
      - Java: hash/32 == hash<<5 but hash<<5 is much faster

Collisions

- The price we pay is that we can no longer guarantee that all words get unique keys.
- If two data items hash to the same index, it is known as a collision.
- Collisions are largely unavoidable.
  - Pick a good hashing function so that collisions are less frequent
  - Work around it when it does happen

Working around Collisions

- Open addressing
  - Recall that we allocate an array twice the size of the number of words
  - If “cats” hashes to 5421, and that location is already occupied, hash “cats” to 5422.
- Separate chaining
  - Have each cell store a linked list of words
  - Every word hashes to an index will be inserted into the list

Linear Probing on Open Addressing

- Search sequentially for next vacant cell.
- 5421 is occupied, try 5422, 5423, and so on.
- Insertion will attempt to insert at hashed index, if occupied, keep incrementing index and insert at first vacant cell.
- Find does the same, except that if a vacant cell is encountered during the probing before a match is found, report failure.

Deletion under Open Addressing

- Deletion usually requires a way to mark a data item as deleted, but not simply vacating the cell.
- Recall the find method will report failure if a vacant cell is encountered before a match. Thus giving up prematurely.
- Instead mark the item off, such as by setting value to –1.
- Often simply not allowed.

find

```
public DataItem find(int key) {
  int hval = hashFunc(key);

  while(hashArray[hval]!=null){
    if(hashArray[hval].getKey()==key){
      return hashArray[hval]; //yes!
    else {
      hashval++;
      if (hashval==arraySize)
         hashval=0;
    }
  }
  return null;
}
```

insert

```
public void insert(DataItem d) {
   int hval = hashFunc(d.getKey());

   while((hashArray[hval]!=null) &&
  (hashArray[hval].getKey()!=-1)) {
      hashval++;
      if (hashval==arraySize)
           hashval=0;
   }
   hashArray[hval] = d;
   return;
}
```

Clustering

- A Clustering is a chain of data items stored out of their hashed locations due to collisions.
- As the table gets full, clustering become larger, which can result in very long probe lengths.
- Performance degenerates seriously when the array is more than 2/3 full, but best kept at 1/2 full (less wastes a lot of space).

Quadratic Probing

- IDEA – To reduce clustering, do not search sequentially.
  - Probe more widely separated cells, that is, at each step, increment the index by the square of the step:
    - instead of: x+1, x+2, x+3…
    - use: x+1, x+4, x+9…
- Secodary Clustering
  - Quadratic probing eliminates the kind of clustering we saw before, known as primary clustering.
  - All keys hashed to a specific index follow the same sequence looking for a vacant cell.
  - Not as serious and it is solvable! How?