Hashtables

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

find

```
public Object find(String key) {
  int hval = hashfun(key);
  while(hashArray[hval][0]!=null){
    if(((String)hashArray[hval][0]).equals(key))
        return hashArray[hval][1]; //yes!
    else {
        hval++;
        if (hval==hashArray.length)
           hval=0;
      }
  }
  return null;
}
```

insert

```
private void insertLP(String key,
  Object value) {
    int hval = hashfun(key);
    while(hashArray[hval][0]!=null &&
    !((String)hashArray).equals(emkey))
    {
        hval++;
        if (hval==hashArray.length)
           hval=0;
    }
    hashArray[hval][0]=key;
    hashArray[hval][1]=value;
}
```

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

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.

Unique Probing Sequence

- Secondary clustering forms because the probe sequence is always the same: 1, 4, 9, 16 …
- What we want is a different sequence for every key.
- Make the probing dependent on the key, which is unique by design.

Double Hashing

- The idea is to hash the key a second time, using a different hash function, then use the result as a step size.
- Secondary hash function must:
  - not be the same as the primary
  - never output a 0 (there will be no step)
- Function that has worked well:
  - `step = constant-(key%constant)`
  - Array size is prime and constant is smaller than

Double Hashing: `insert`

```
private void insertDH(String key, Object
  value) {
   int hval = hashfun(key);
   int step=hashfun2(key);
   while(hashArray[hval][0]!=null && !
   ((String)hashArray[hval][0]).equals
   (EMKEY)) {
      hval+= step;
      if (hval==hashArray.length)
         hval-=hashArray.length;
    }
   hashArray[hval][0]=key;
   hashArray[hval][1]=value; }
```

Double Hashing -- Why Prime?

- Suppose we have a table size of 15.
- A particular item hashes to initial index of 0, and step size of 5.
  - 0, 5, 10, 0, 5 …
  - Only checks 3 cells!
- Change array size to 13
  - 0, 5, 10, 2, 7, 12, 4, 9, 1, 6, 11, 3, 8
  - All cells are checked.

Separate Chaining

- Recall that linked lists are used to queue up data items that hash to the same index.
- The hash table in fact turns into an array of linked lists.
- Conceptually simpler, also allows for duplicate data items.
- Must include linked list in implementation.

Hash Functions

- Quick computation – avoid too many multiplications and divisions
- Good distribution of hashed items – collision avoidance
  - random keys – mod is good enough
  - non-random keys, i.e. license plate numbers – data dependent
- Use a prime number for mod base

Hashing Efficiency

- If no collision, insertion and search approach O(1).
- If collisions occur, access time depends on probe lengths.
- Average probe length depends on the load factor (how full the table is).

Open Addressing

- Let:
  - L=load factor
  - P = number of probes
- Linear Probing: L = Load factor
  - successful: $P = (1 + 1 / (1-L))/2$
  - unsuccessful: $p = (1 + 1/(1-L)^2) / 2$
- Quadratic and Double Hashing:
  - successful: $P = -\log_2(1-L)/L$
  - unsuccessful: $P = 1 / (1-L)$

Separate Chaining

- Searching
  - successful: P = 1 + L/2
  - unsuccessful: P = 1+L or P=1+L/2
- Insertion:
  - unordered: P=1
  - ordered: P=1+L/2