Hash functions

- approximates equivalent likelihood that a key will hash to any of the m slots (uniformity)

division method

- use remainder of k when divided by table size (modulo operation)
- best practice to avoid m as power of 2
- good to choose prime somewhere between (i.e. not too close to) powers of two

multiplication method

- multiply key by constant in 0 < constant < 1
- extract bits of fractional component
- multiply by m (number of elements in table) and take the floor
- Knuth suggested (5^1/2 - 1)/2 as the constant
- Sedgewick suggests simplifying to (key * a) % m and using the golden ratio

Prime numbers for hash tables

- Mersenne primes
  - all p = 2^t - 1 are prime for t = 2, 3, 5, 7, 13, 19, 31 and no other t < 31
- most efficient to use a lookup table

Horner's algorithm

Hashing

- hashing is an exception of key-indexed searching
- transform search key into a table address and handle collision resolution for keys that map to the same address
- hashing speeds access at the expense of space
- theoretical optimum search and insert in constant time

Hash Functions

- maps keys to table addresses -> the range of `[0, M - 1]` for M keys

- an ideal hash function makes each output equally likely for each input
- prior to high level languages, a grouping of data could be seen as type variant and hashing was a natural extension of this
- hash functions can often be machine dependent
- basic approaches:
  - floating point numbers in a fixed range can be rounded and/or rescaled by subtracting computing ((n - s) / (t - s)) * M where s is the lower bound for keys and t is the upper bound and M is the number of keys
  - for w-bit integers, they can be converted to floating point numbers and divided by 2 ^ w then multiplied by M
  - the same result can be achieved by multiplying by M and right shifting by w (or vice versa) (only useful for keys evenly distributed in the range)
  - modular hash functions for integer keys -> choose M to be prime (generally closest prime to a power of 2) and computer k % M
  - modular hash for floats -> scale to convert to 0...1, multiply by 2 ^ w and then use a modular hash function
  - for integer keys -> multiply by a constant between 0 and 1 and reduce it modulo M (choose a random constant that does not have a relationship to the M -> a popular choice is the golden ration .618033...
- non-prime M's result in unfortunate mathematical recurrences
- Mersenne primes: (2 ^ t - 1) <= 31 -> 2,3,5,7,13,17,19,31
- fastest to store a table of the largest primes less than 2^n for 8 <= n <= 32
- hash function for string keys (close to Horner's algorithm, NOTE: Horner's algorithm is larger, for computing polynomials):

```
int hash(char *v, int M) {
    int h = 0, a = 127;
    for (; *v != 0; v++) {
        h = (a*h + *v) % M;
    }
    return h;
}
```

- universal hash function for string keys (with pseudorandom coefficient values):

```
int hashU(char *v, int M) {
    int h, a = 31415, b = 27183;
    for (h = 0; *v != 0; v++, a = a*b % (M-1)) {
        h = (a*h + *v) % M;
    }
    return (h < 0) ? (h + M) : h;
}
```

- hash functions that produce random table indices no matter what the keys are are desirable

- using random values for the coefficients and a different random value for each digit in the key leads to universal hashing
- ideal is that the probability of collision is 1 / M
- a symbol table using hashing must be given keys of a type that have a hashing function defined
- universal hashing is often much slower
- important to be aware of the cost of the hash function and it's location (do not place in inner loops)
- often fastest hashing method:

```
inline int hash(Key v, int M) { return v & (M - 1); }
```

Separate Chaining

- important to handle hash collisions
- most basic method is to place a linked list in table addresses and resolve collisions using list search (separate chaining)
- can use a dummy header node or can have the first nodes in the in the lists comprise the table

```
private:
    link* heads;
    int N, M;
public:
    ST(int maxN) {
        N = 0; M = maxN/5;
        heads = new link[M];
        for (int i = 0; i < M; i++) heads[i] = 0;
    }
    Item search(Key v) { return searchR(heads[hash(v, M)], v); }
    void insert(Item item) {
        int i = hash(item.key(), M);
        heads[i] = new node(item, heads[i]); N++;
    }
```

- properties:
  - reduces the number of comparisons for sequential search by a factor of M on average using extra space for M links
  - in a separate chaining hash table with M lists and N keys, the probability that the number of keys in each list is within a small constant factor of N / M is extremely clost to 1
  - NOTE: see the chapter for some formulas for various probabilities
- best to choose M to be small enough to not waste space with empty links (1/5 or 1/10 of expected number of keys)

- can optimize operations depending on importance of search vs insert
- hashing is not good for sort and select but good for search, insert, and remove
- good for symbol table in a compiler

Linear Probing

- if the number of keys can be estimated and contiguous memory can be allocated, then links do not need to be used
- open-addressing hashing: store n items in a table of size M > N, using empty locations to store collisions
- linear probing: when a collision occurs, just find the next open address, wrapping back if the end is reached
- best to keep the table from getting completely full
- linear probing:

```
private:
    Item *st;
    int N, M;
    Item nullItem;
public:
    Symbol_table(int maxN) {
        N = 0; M = 2*maxN;
        st = new Item[M];
        for (int i = 0; i < M; i++) st[i] = nullItem;
    }
    int count() const { return N; }
    void insert(Item item) {
        int i = hash(item.key(), M);
        while (!st[i].null()) i = (i+1) % M;
        st[i] = item; N++;
    }
    Item search(Key v) {
        int i = hash(v, M);
        while (!st[i].null())
        if (v == st[i].key()) return st[i];
        else i = (i+1) % M;
        return nullItem;
    }
```

- uses a table 2 X the expected size
- properties:
  - a = N / M where a is the average number of items in the table is generally larger than 1 (a is called the load factor)
  - when collisions are resolved with linear probing, the average number of probes required to search in a hash table of size M that contains N = aM keys is about 1 / 2 * (1 + (1/(1-a)) for hits and 1 / 2(1 + 1/(1-a)^2) for misses
- the accuracy of the estimates decreases as a approaches 1

- keys in the table are in random order
- sort and select requires other operations so linear probing is not appropriate in these cases
- removal from a table with linear probing:

```
void remove(Item x) {
    int i = hash(x.key(), M), j;
    while (!st[i].null())
        if (x.key() == st[i].key()) break;
            else i = (i+1) % M;
    if (st[i].null()) return;
    st[i] = nullItem; N--;
    for (j = i+1; !st[j].null(); j = (j+1) % M, N--) {
        Item v = st[j]; st[j] = nullItem; insert(v);
    }
}
```

Double Hashing - TODO: more work here

- clustering causes linear probing to run slowly for tables that are nearly full
- double hashing: use a second hash function to set a fixed increment for probing after a collision
- important to choose second hash function with care (e.g. it cannot evaluate to 0)

```
inline int hash_two(Key v) { return (v % 97) + 1; }
```

- double hashing:

```
void insert(Item item) {
    Key v = item.key();
    int i = hash(v, M), k = hashtwo(v, M);
    while (!st[i].null()) i = (i+k) % M;
    st[i] = item; N++;
}

Item search(Key v) {
    int i = hash(v, M), k = hashtwo(v, M);
    while (!st[i].null())
    if (v == st[i].key()) return st[i];
        else i = (i+k) % M;
    return nullItem;
}
```

- properties:
    - when collisions are resolved with double hashing, the average number of probes required to a search a hash table of size M that contains N = aM keys is (1 / a) * (ln (1 / (1-a)) for hist and 1/(1-a) for misses

Dynamic Hash Tables

- increase in keys in a hash table degrades search
- tables fill up and performance decreases drastically towards the upper limit
- trees are more elastic
- doubling a hash table's size is expensive but happens so infrequently its cost is only a constant fraction of building the table
- dynamic hashing works for both linear probing and separate chaining
- keeps size at 1/4 to 1/2 so search cost is less than 3 probes on average
- dynamic hash insertion for linear probing:

```
private:
    void expand() {
        Item *t = st;
        init(M+M);
        for (int i = 0; i < M/2; i++) {
            if (!t[i].null()) insert(t[i]);
        }
        delete t;
    }
public:
    Symbol_table(int maxN) { init(4); }
    void insert(Item item) {
        int i = hash(item.key(), M);
        while (!st[i].null()) i = (i+1) % M;
        st[i] = item;
        if (N++ >= M/2) expand();
    }
```

- requires allocating size for the new table, rehashing all the keys in the new table, freeing the memory for the old table
- properties:
    - a sequence of t search, insert, and delete symbol-table operations can be executed in time proportional to t and with memory usage always within a constant factor of the number of keys in the table
- this is the best option for a library hashing implementation

Perspective

- choice between linear probing and double hashing depends on the cost of computing the hash and the load factor of the table
- linear probing and double hashing vs. separate chaining comparison is more complicated -> see book for details

- many other hashing methods for special situations:
  - move items around during insertion in double hashing to make successful search more efficient
  - ordered hashing: reduces cost of unsuccessful search - in standard search, the search stops when an empty location is hit or an item with an equal key is hit ordered hashing -> stop when a key greater than or equal to the current key is found requires more work
  - exception dictionary: when the table has a fast search miss and a slow search hit (i.e. certain string searches)

Extendible Hashing - TODO: more notes here

- combines elements of multiway-trie algorithms, sequential access methods, and hashing to a method that generally requires just one or two probes for search and insert
- definition:
  - extendible hash table (of order d): a directory of 2^d references to pages that contain up to M items with keys. The items on each page are identical in their first k bits and the directory contains 2^(d-k) pointers to the page, starting at the location specified by the leading k bits in the keys on the page
- extendible hashing data structures:

```
template <class Item, class Key>
class ST
  {
    private:
      struct node
        { int m; Item b[M]; int k;
          node() { m = 0; k = 0; }
        };
      typedef node *link;
      link* dir;
      Item nullItem;
      int N, d, D;
    public:
      ST(int maxN)
        { N = 0; d = 0; D = 1;
          dir = new link[D];
          dir[0] = new node;
        }
    };
```

- extendible hashing search:

```
private:
  Item search(link h, Key v)
    {
      for (int j = 0; j < h->m; j++)
        if (v == h->b[j].key()) return h->b[j];
      return nullItem;
```

```
    }
public:
  Item search(Key v)
    { return search(dir[bits(v, 0, d)], v); }
```

- extendible hashing insertion:

```
  void split(link h)
    { link t = new node;
      while (h->m == 0 || h->m == M)
        {
          h->m = t->m = 0;
          for (int j = 0; j < M; j++)
            if (bits(h->b[j].key(), h->k, 1) == 0)
                 h->b[h->m++] = h->b[j];
            else t->b[t->m++] = h->b[j];
          t->k = ++(h->k);
        }
      insertDIR(t, t->k);
    }
  void insert(link h, Item x)
    { int j; Key v = x.key();
      for (j = 0; j < h->m; j++)
        if (v < h->b[j].key()) break;
      for (int i = (h->m)++; i > j; i--)
        h->b[i] = h->b[i-1];
      h->b[j] = x;
      if (h->m == M) split(h);
    }
public:
  void insert(Item x) { insert(dir[bits(x.key(), 0, d)], x); }
```

- extendible hashing directory insertion:

```
void insertDIR(link t, int k)
  { int i, m, x = bits(t->b[0].key(), 0, k);
    while (d < k)
      { link *old = dir;
        d += 1; D += D;
        dir = new link[D];
        for (i = 0; i < D; i++) dir[i] = old[i/2];
        if (d < k) dir[bits(x, 0, d)^1] = new node;
      }
    for (m = 1; k < d; k++) m *= 2;
    for (i = 0; i < m; i++) dir[x*m+i] = t;
  }
```

- properties:
    - the extendible hash table built from a set of keys depends on only the values of those keys, and does not depend on the order in which the keys are inserted

- with pages that can hold m items, extendible hashing requires about 1.44 * (N/M) pages for a file of N items on average. The expected number of entries in the directory is about 3.92 * (N ^(1/M))(N/M)