Hashing Algo

Key-indexed arrays had "perfect" search performance O(1)

- but required a dense range of index values
- used a fixed-size array (max size ever needed)
- bigger array ⇒ more useful but wastes more space

Hashing allows us to approximate this performance, but

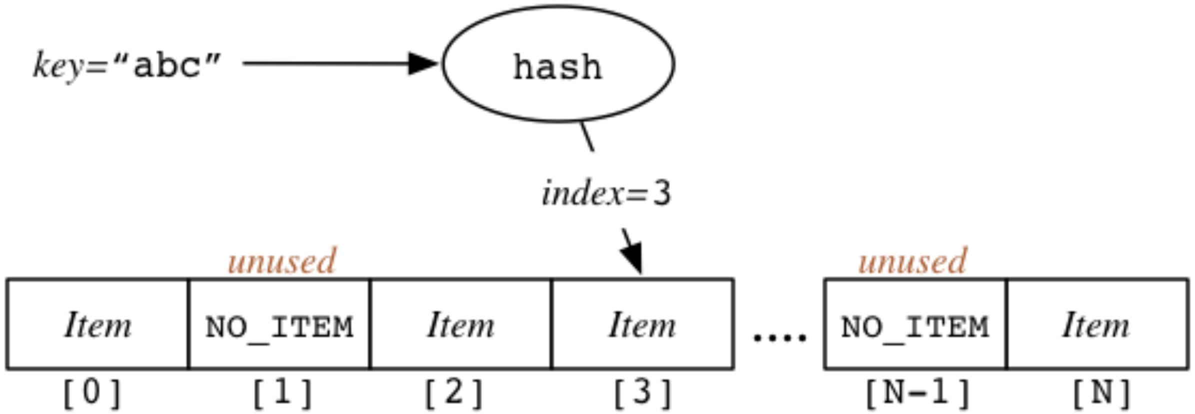
- allows arbitrary types of keys
- map (hash) keys into compact range of index values
- store items in array, accessed by index value

The ideal for key-indexed collections:

```
courses["COMP3311"] = "Database Systems";
printf("%s\n", courses["COMP3311"]);
```

Almost as good:

```
courses[h("COMP3311")] = "Database Systems";
printf("%s\n", courses[h("COMP3311")]);
```



To use arbitrary values as keys, we need three things:

- set of ***Key*** values, each key identifies one Item
- an array (of size N) to store *Items*
- a ***hash function*** $h()$ of type Key→[0..N-1]
  - requirement: if $(x == y)$ then $h(x) == h(y)$
  - requirement: $h(x)$ always returns same value for given x
- a ***collision resolution*** method
  - collision = (x != y && h(x) == h(y))
  - collisions are inevitable when dom(Key) >> N

Notes:

- converts *Key* value to index value [0..N-1]
- deterministic (key value k always maps to same value)
- use *mod* function to map hash value to index value
- spread key values *uniformly* over address range (assumes that keys themselves are *uniformly* distributed)
- Avoid collisions as much as possible, h(k) ≠ h(j) if j ≠ k

- cost of computing hash function must be cheap

```
typedef struct HashTabRep {
   int  N;       // size of array
   Item **items; // array of (Item *)
} HashTabRep;

// create a new hash table
HashTable newHashTable(int N)   // N is the size of the hash table
{
   HashTable new = malloc(sizeof(HashTabRep));
   new->items = malloc(N*sizeof(Item *));
   new->N = N;
   for (int i = 0; i < N; i++)
      { new->items[i] = NULL; }
   return new;
}

// hash function
int hash(Key key, int N)
{
   int val = convert key to int;
   return val % N;
}
```

Examples for Hash Functions

Sum

```
int hash(char *key, int N)
{
    int h = 0; char *c;
    for (c = key; *c != '\0'; c++)
        h = h + *c;
    return h % N;
}
```

A slightly more sophisticated hash function:

```
int hash(char *key, int N)
{
   int h = 0;  char *c;
   int a = 127; // a prime number
   for (c = key; *c != '\0'; c++)
      h = (a * h + *c) % N;
   return h;
}
```

To use all of value in hash, with suitable "randomization":

```
int hash(char *key, int N)
{
   int h = 0, a = 31415, b = 21783;
   char *c;
   for (c = key; *c != '\0'; c++) {
      a = a*b % (N-1);
      h = (a * h + *c) % N;
   }
   return h;
}
```

A real hash function (from PostgreSQL DBMS):

```
hash_any(unsigned char *k, register int keylen, int N)
{
    register uint32 a, b, c, len;
    // set up internal state
    len = keylen;
    a = b = 0x9e3779b9;
    c = 3923095;
    // handle most of the key, in 12-char chunks
    while (len >= 12) {
        a += (k[0] + (k[1] << 8) + (k[2] << 16) + (k[3] << 24));
        b += (k[4] + (k[5] << 8) + (k[6] << 16) + (k[7] << 24));
        c += (k[8] + (k[9] << 8) + (k[10] << 16) + (k[11] << 24));
        mix(a, b, c);
        k += 12; len -= 12;
    }
    // collect any data from remaining bytes into a,b,c
    mix(a, b, c);
    return c % N;
}

#define mix(a,b,c) \
{ \
  a -= b; a -= c; a ^= (c>>13); \
  b -= c; b -= a; b ^= (a<<8);  \
  c -= a; c -= b; c ^= (b>>13); \
  a -= b; a -= c; a ^= (c>>12); \
  b -= c; b -= a; b ^= (a<<16); \
  c -= a; c -= b; c ^= (b>>5);  \
  a -= b; a -= c; a ^= (c>>3);  \
  b -= c; b -= a; b ^= (a<<10); \
  c -= a; c -= b; c ^= (b>>15); \
```

```
}
```

Collision Resolution

Collision resolution approaches:

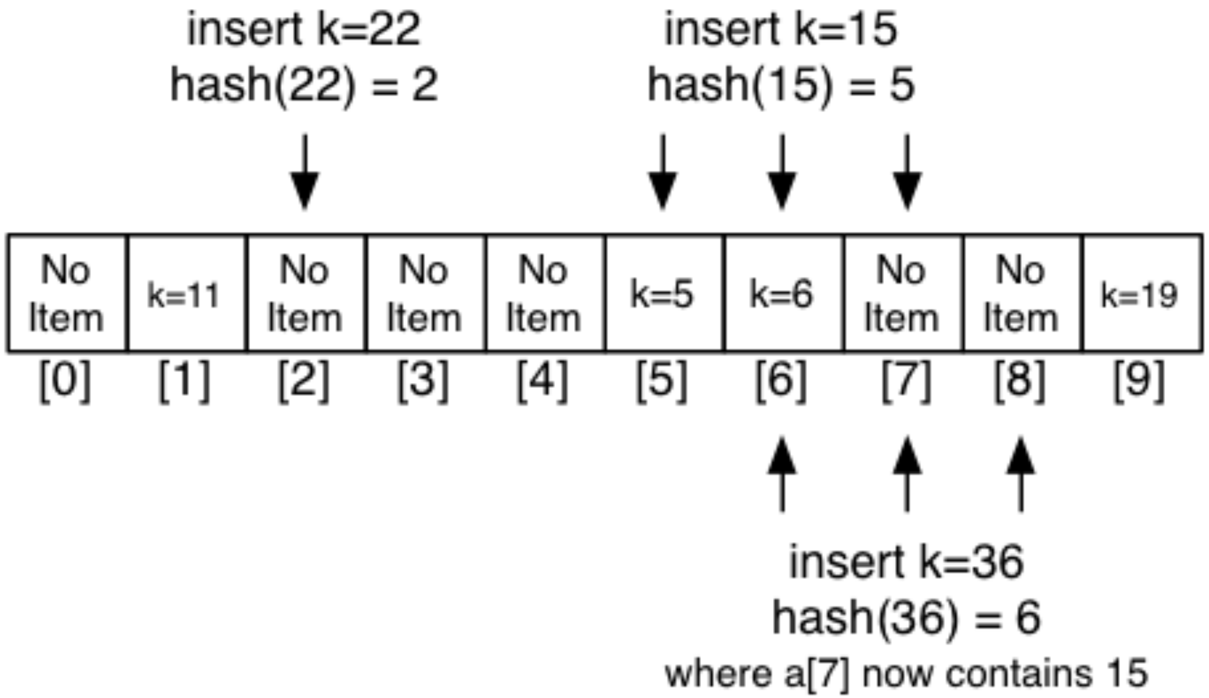
- ***Linear probing***: fast if $\alpha << 1$, complex deletion
- ***Double hashing***: faster than linear probing, esp for $\alpha \cong 1$
- ***Separate chaining***: easy to implement, allows $\alpha > 1$

Only chaining allows α > 1, but performance degrades once α > 1.

Closed hashing (open addressing)

A closed hashing can only store the the as many entries as its slots.

Linear Probing



Search cost analysis:

- cost to reach first *Item* is O(1)
- subsequent cost depends how much we need to scan
- affected by ***load*** $\alpha = M/N$ (i.e. how "full" is the table)
- Avg. Cost for successful search = $0.5 * (1 + 1/(1 - \alpha))$
- Avg. Cost for unsuccessful search = $0.5 * (1 + 1/(1 - \alpha)2)$

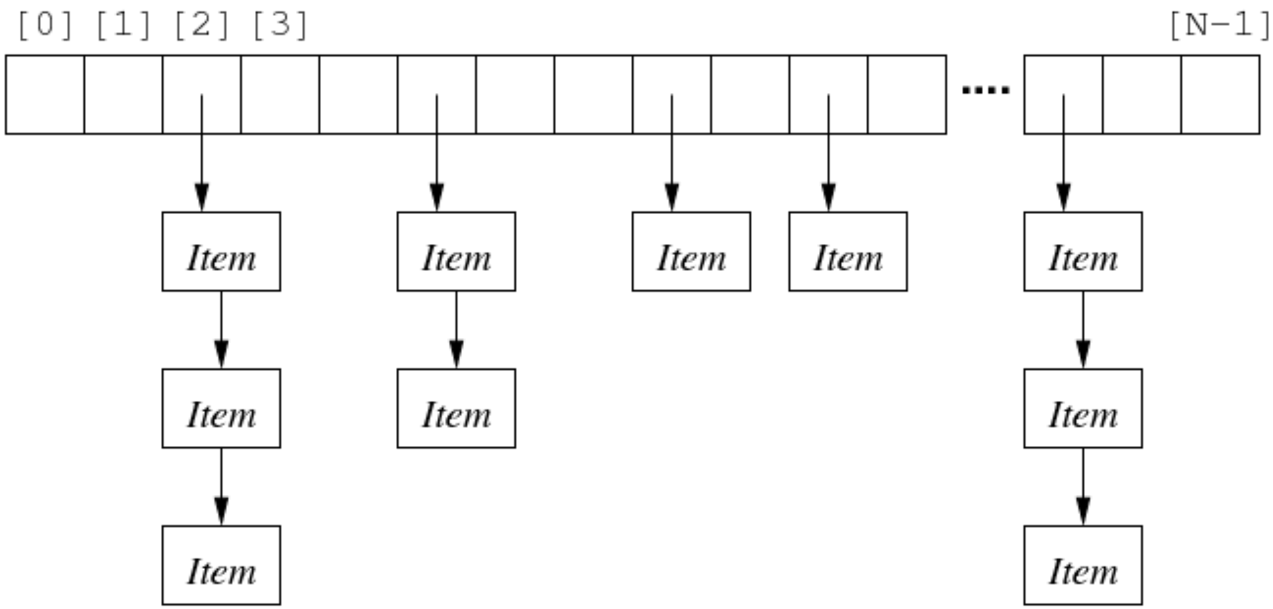
Double hashing

We use a secondary hashing to calculate a new index (first hashing + secondary hashing), when a collision happened.

Can be significantly better than linear probing, especially if table is heavily loaded.

Open hashing (separate chaining)

Store items in a linked list with an ascending order (the linked list is ordered!)



Worst case: all items share one identical key, forming a very long chains.
Cost: to insert k items, we cost $(k-1)+(k-2)+...+2+1=O(k^2)$

Best case: all items all distributed equally, all chains have the same length
Cost: $O(1)$