Hashing

- Associative Indexing
- Hashing
- Hash Table ADT
- Hash Functions
- Problems with Hashing

❖ Associative Indexing

Regular array indexing is positional (`[0]`, `[1]`, `[2]`, ...):

- can access items by their position in an array
- but generally don't know position for an item with key K
- we need to search for the item in the collection using K
- search can be linear (0(n)) or binary (O($\log_2$n))

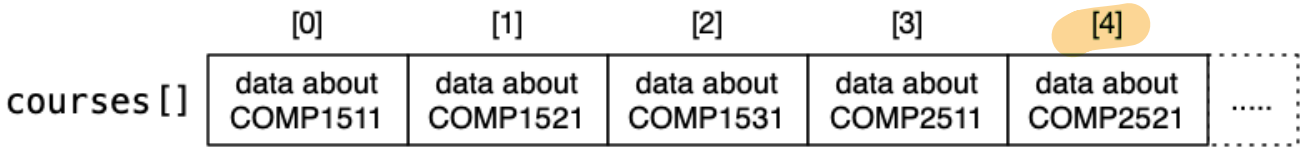
An alternative approach to indexing:

- use the key value as an index ... no searching needed
- access data for item with key K as **`array[K]`**

❖ ... Associative Indexing

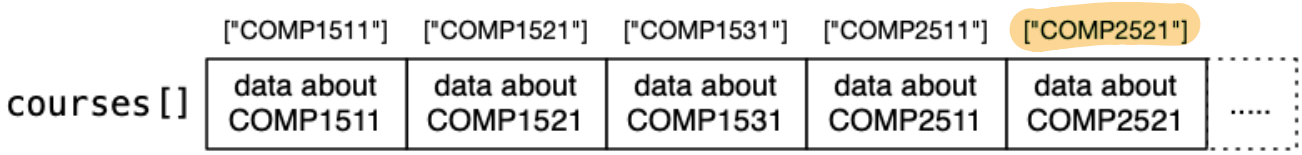
Difference between positional and associative indexing:

Positional (normal) indexing



`courses[4]` *gives access to COMP2521 data*

Associative indexing



`courses["COMP2521"]` *gives access to COMP2521 data*

❖ Hashing

Hashing allows us to get close to associative indexing

Ideally, we would like ...

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

but C doesn't have non-integer index values.

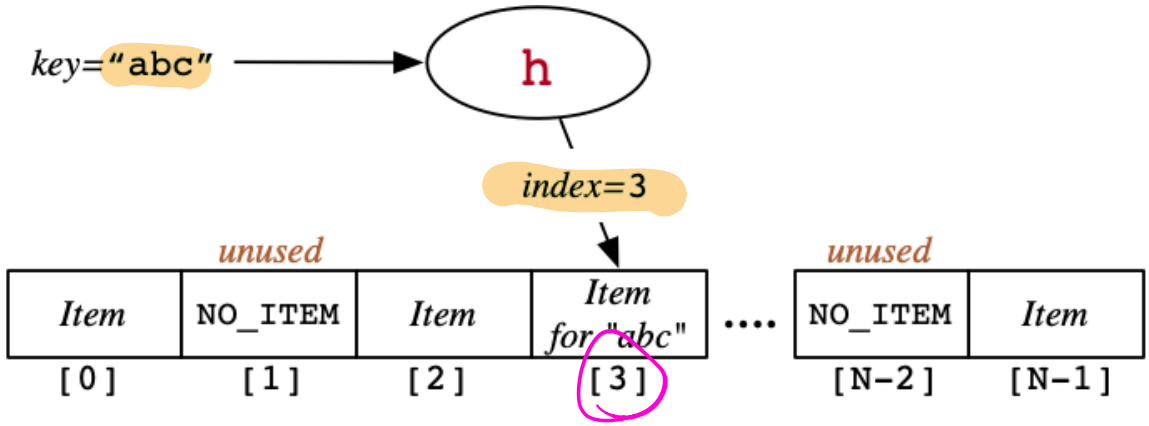
Something almost as good:

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

Hash function `h` converts key to integer and uses that as the index.

❖ ... Hashing

What the **h** (hash) function does



Converts a key value (of any type) into an integer index.

Sounds good ... in practice, not so simple ...

❖ ... Hashing

Reminder: what we'd like ...

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

In practice, we do something like ...

```
key = "COMP3311";
item = {"COMP3311","Database Systems",...};
courses = HashInsert(courses, key, item);
printf("%s\n", HashGet(courses, "COMP3311"));
```

❖ ... Hashing

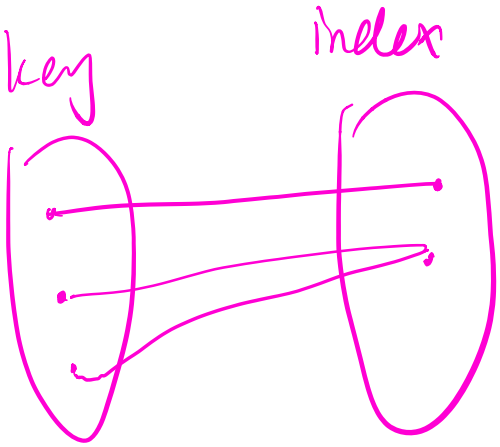
To use arbitrary values as keys, we need ...

- set of **`Key`** values *dom(**`Key`**)*, each key identifies one **`Item`**
- an array (of size *N*) to store **`Item`**s
- a hash function *h()* of type *dom(**`Key`**)* → [0..*N*-1]
  - requirement: if *(x = y)* then *h(x) = h(y)*
  - requirement: *h(x)* always returns same value for given *x*

A problem: array is size *N*, but *dom(**`Key`**)* ≫ *N*

So, we also need a collision resolution method

- collision = (*x* ≠ *y* *but* *h(x) = h(y)*)



• Many keys → one index

❖ Hash Table ADT

Generalised ADT for a collection of **`Item`**s

Interface:

```
typedef struct CollectionRep *Collection;

Collection newCollection();     // make new empty collection
Item *search(Collection, Key); // find item with key
void insert(Collection, Item); // add item into collection
void delete(Collection, Key);  // drop item with key
```

Implementation:

```
typedef struct CollectionRep {
    ... some data structure to hold multiple Items ...
} CollectionRep;
```

❖ ... Hash Table ADT

For hash tables, make one change to "standard" Collection interface:

```
typedef struct HashTabRep *HashTable;
// make new empty table of size N
HashTable newHashTable(int);
// add item into collection
void HashInsert(HashTable, Item);
// find item with key
Item *HashGet(HashTable, Key);
// drop item with key
void HashDelete(HashTable, Key);
// free memory of a HashTable
void dropHashTable(HashTable);
```

i.e. we specify max # elements that can be stored in the collection

❖ ... Hash Table ADT

Example hash table implementation:

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

HashTable newHashTable(int N)
{
   HashTable new = malloc(sizeof(HashTabRep));
   new->items = malloc(N*sizeof(Item *));
   new->N = N;
   new->nitems = 0;
   for (int i = 0; i < N; i++) new->items[i] = NULL;
   return new;
}
```

❖ ... Hash Table ADT

Hash table implementation (cont)

```
void HashInsert(HashTable ht, Item it) {
   int h = hash(key(it), ht->N);
   // assume table slot empty!?
   ht->items[h] = copy(it);
   ht->nitems++;
}
Item *HashGet(HashTable ht, Key k) {
   int h = hash(k, ht->N);
   Item *itp = ht->items[h];
   if (itp != NULL && equal(key(*itp),k))
      return itp;
   else
      return NULL;
}
```

key() and **copy()** come from **Item** type; **equal()** from **Key** type

❖ ... Hash Table ADT

Hash table implementation (cont)

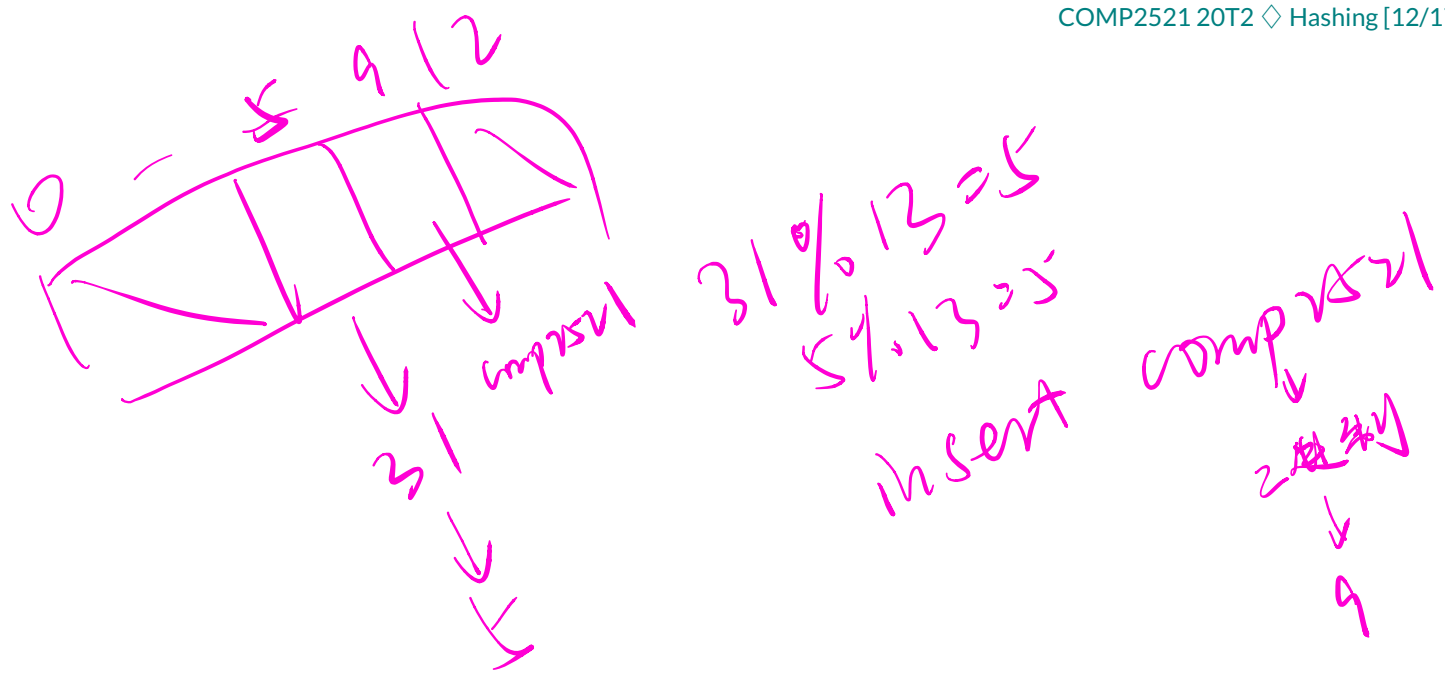
```
void HashDelete(HashTable ht, Key k) {
   int h = hash(k, ht->N);
   Item *itp = ht->items[h];
   if (itp != NULL && equal(key(*itp),k)) {
      free(itp);
      ht->items[h] = NULL;
      ht->nitems--;
   }
}
void dropHashTable(HashTable ht) {
   for (int i = 0; i < ht->N; i++) {
      if (ht->items[i] != NULL) free(ht->items[i]);
   }
   free(ht);
}
```

❖ Hash Functions

Characteristics of hash functions:

- 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)
- as much as possible, $h(k) \neq h(j)$ if $j \neq k$
- cost of computing hash function must be cheap

key = -1
N = 13 hash size
-1 mod 13 always
in 0-12 (index)



❖ ... Hash Functions

Basic mechanism of hash functions:

```
int hash(Key key, int N)
{
    int val = convert key to 32-bit int;
    return val % N;
}
```

index

If keys are **int**s, conversion is easy (identity function)

How to convert keys which are strings? (e.g. "**COMP1927**" or "**John**")

Definitely prefer that *hash*("**cat**",N) ≠ *hash*("**dog**",N)

Prefer that *hash*("**cat**",N) ≠ *hash*("**act**",N) ≠ *hash*("**tac**",N)

❖ ... Hash Functions

Universal hashing uses entire key, with position info:

```
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;
}
```

Has some similarities with RNG. Aim: "spread" hash values over [0..N-1]

❖ ... Hash Functions

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;
}
```

❖ ... Hash Functions

Where **`mix`** is defined as:

```
#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); \
}
```

i.e. scrambles all of the bits from the bytes of the key value

❖ Problems with Hashing

In ideal scenarios, search cost in hash table is *O(1)*.

Problems with hashing:

- hash function relies on size of array (⇒ can't expand)
  - changing size of array effectively changes the hash function
  - if change array size, then need to re-insert all **`Item`**s
- items are stored in (effectively) random order
- if size(KeySpace) ≫ size(IndexSpace), collisions inevitable
  - collision: k ≠ j && hash(k,N) = hash(j,N)
- if **`nitems`** > **`nslots`**, collisions inevitable

31%13 = 5%13
=5
31=k j=5