COMP20003
Algorithms and Data Structures
Hash Tables

Kris Ehinger
Department of Computing and Information Systems
University of Melbourne
Semester 2

So far…

- Dictionary search has been based on key comparisons.
  - Linked list (sorted and unsorted)
  - Array (sorted and unsorted)
  - Binary Search Tree
  - Balanced Trees

This section

- A way of going directly to the desired item
- Hash tables:
  - Search usually takes only 1 (or few) operations.
    - (on average)
    - (if managed well)
    - (but very bad worst case)
- Probabilistic data structure

Textbook

- Skiena: Chapter 3, Section 3.7

Direct storage and search (a hypothetical fast method)

Task: Store in dictionary items with keys within the range 0 to **RANGE-1**.

- Data structure: Array

```
itemtype*    A[RANGE];
```

- Operations:

```
void initialize( itemtype* A) {
      for(i=0;i<RANGE;i++)
         A[i] = NULL;
   }
void insert( itemtype* item ) { A[item->key] = item; }
itemtype* search( int key )   { return A[key]; }
```


Direct storage and search (a hypothetical fast method)

Create table and initialize:

```
#define RANGE 10000
#define EXAMPLEKEY 8179
struct item{
  int key;
  char *info;
} item;
item  *A[RANGE];
item  *newitem;

for( i=0;i < RANGE; i++ ) A[i] = NULL;
```


Direct storage and search (a hypothetical fast method)

- Insert item with key=**EXAMPLEKEY**:

```
newitem = (item *)malloc……
newitem->key = EXAMPLEKEY;
newitem->info = …malloc….strcpy…
A[EXAMPLEKEY] = newitem;
```

- Search for item with key=**EXAMPLEKEY**:

```
return A[EXAMPLEKEY];
```


Thought experiment: Build an array to store employee data

- Keys = names
- How to map names to array positions?
- Names known in advance:
  - Alphabetize, assign first name = 0, second name = 1, etc.
- Names not known in advance?

Thought experiment: Build an array to store employee data

AA – 0	AAA – 676	AAAA – 18252
AB – 1	AAB – 677	AAAB – 18253
AC – 2	AAC – 678	AAAC – 18254
⋮	⋮	⋮
AZ – 25	AZZ – 1351	
BA – 26	BAA – 1352	
BB – 27	BAB – 1353	
⋮	⋮	
ZY – 674	ZZY – 18250	
ZZ – 675	ZZZ – 18251	

- Bad solution: needs a lot of space, most of which will never be used

Thought experiment: Build an array to store employee data

AA	AAKER	AAMOLD
AAB	AAKHUS	AAMOT
AABERG	AAKRE	AAMOTH
AABY	AALAND	AANDERUD
AADLAND	AALBERS	AANENSON
AAFEDT	AALDERINK	AANERUD
AAGAARD	AALFS	AANESTAD
AAGARD	AALGAARD	AANONSEN
AAGESEN	AALTO	AARDAL …

US surname dataset: https://data.world/fivethirtyeight/most-common-name

- Better solution? Still needs a lot of space.
- No way to handle a name not in the list.

Thought experiment: Build an array to store employee data

Any name → number from 0-ARRAYSIZE

- Best solution: find a function that can map any name to an array position

Limitations of array approach?

- Key range is often not known
- `RANGE` might be too large for a practical array size
- We have assumed that keys are unique

How to make it practical

- Solution: circular array
  - Squash the keys to fit into an array:
    - A[100]
    - Store key in A[key%100]
- Issue: Collisions
  - If key1 = 200 and key2 = 400, both map to A[0]
  - Collisions are *always* possible, so *must* have a plan
  - Solution: Patterns
    - Use complicated mapping of keys to disrupt patterns

Patterns Excercise

Key = Input % modulo

Input	Modulo 8	Modulo 7
0		
4		
8		
12		
16		
20		
24		
28		

Use the idea: make it practical

Input	Modulo 8	Modulo 7
0	0	0
4	4	4
8	0	1
12	4	5
16	0	2
20	4	6
24	0	3
28	4	0

Use more complicated mapping → **prime numbers to disrupt patterns**.