1: Tables

Tables

The Table ADT is used when information needs to be stored and acessed via a *key* usually, but not always, a string.

For example:

- Dictionaries
- Symbol Tables
- Associative Arrays (eg in `awk`, `perl`)

Associated with each key is some *data*

The operations required for this ADT are

- Create a new Table
- Insert a key, and associated data, into a Table.
- Delete a key, and associated data, from a Table.
- Retrieve the data associated with a key from a Table.

The Table is essentially a *set* of (key,data) pairs, with the property that, for any k, d, d', if (k, d) and (k, d') are in the set, then $d = d'$.

In other words a Table provides a function $Key \longrightarrow Data$.

Implementing Tables

Tables can be implemented in many ways

- Linked lists of (key,data) pairs
- Tree based representations
  - Ordered binary trees
  - Balanced trees, eg 2-3 trees, Red-Black trees, B-trees, Splay trees

  See lab exercise for simplified version, with no data associated with keys.
- Hash tables

Hash tables

When storing and accessing indexed information, most efficient access mechanism is *direct addressing*.

Simplest example is an array `a`

Can access, or update, any array element `a[i]` in constant time, if we know the index `i`.

If data is stored in other forms of data structure, such as lists, or binary search trees, the access and update cost depends on position of the data in the data structure.

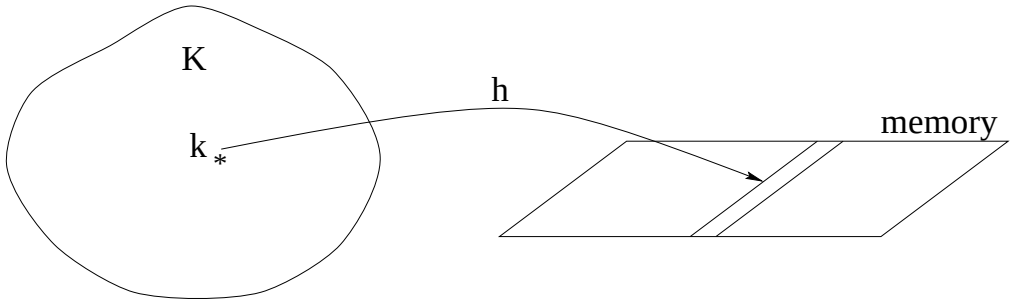
Arrays ok for data indexed by integers

Need similar mechanisms for data with other types of index, e.g. strings of characters

Suppose we are given a set K of keys.

For direct addressing, need a function h, such that, for any $k \in K$

$h(k)$ is the address of the location of data associated with k



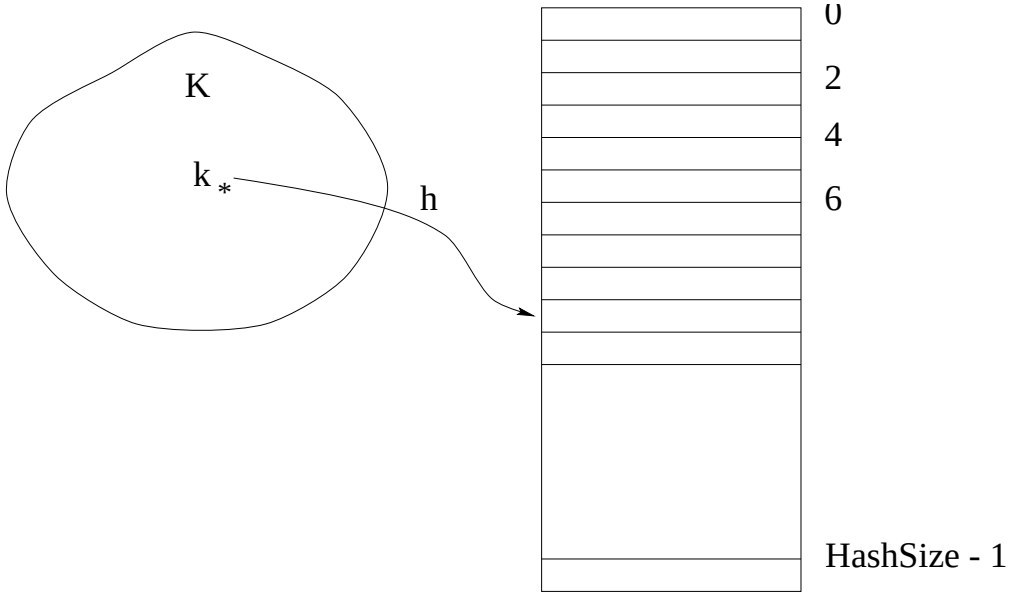
h needs to have the following properties

- It is easy (and fast) to calculate $h(k)$, given k
- It is 1-1 (i.e. no two keys give the same value of h)
- Uses reasonable amount of memory (i.e. the range of h is not too large)

All these properties are satisfied by array indexing, but in general are too much to ask for.

Hash tables give a way round the problem by relaxing the 1-1 condition.

A hash table is basically an array



A *hash function* is a function

$$h : K \longrightarrow (0 \ldots HashSize - 1)$$

h is used whenever access to the hash table is required – e.g. search,

insertion, deletion

A problem arises when two keys k_1, k_2 have

$$h(k_1) = h(k_2)$$

Where does data associated with these keys go?

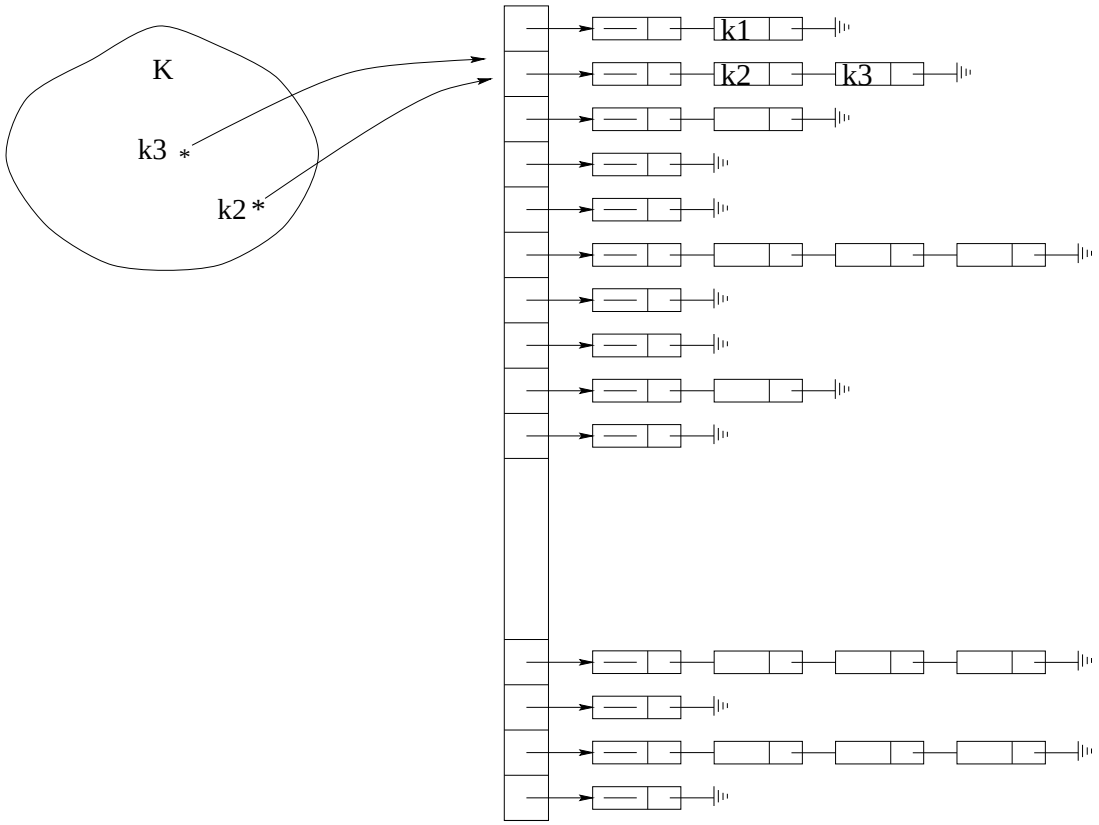
Two mechanisms for resolving this type of conflict

- *Open hashing* (otherwise known as *separate chaining*)
- *Closed hashing* (otherwise known as *open addressing*)

Open hashing

No data is stored in the hash table itself

Hash table just contains pointers to linked lists of data cells



First cells are often dummies, containing no data (as in the above diagram). This simplifies code for insertion and deletion

Have drawn cells containing only a key and a pointer.

The cells could, of course contain additional data associated with the key – and usually do

Need following functions to manipulate hash tables

- Initialise a hash table, given its size
  - Create array (of appropriate size) of pointers
  - Create header cells for each array entry
- Find a key k
  - Calculate $h(k)$
  - look along the list with header at $h(k)$ to see if k is there
  - If it is, return pointer to cell containing k, otherwise return a pointer to the cell after which k should be inserted.
- Insert a key k
  - If already there, do nothing
  - Otherwise, insert a new cell, containing k in the list starting at $h(k)$
- Delete a key k

- If not there, do nothing
- Otherwise, delete cell containing k from the list starting at $h(k)$, releasing any memory used by cell.

Data types for open hashing

```
/* The type for keys */
typedef char * Key_Type;

/* Type for size of hash table */
typedef unsigned int Hash_size;

/* Type for index into hash table */
typedef unsigned int Index;
```

```
/* Type for cells containing keys */
/* No auxiliary data in this example */
typedef struct data_cell *cell_ptr;

struct data_cell
{
  Key_Type element;
  cell_ptr next;
};

/* Type for the list of cell pointers */
typedef cell_ptr List;
/* Type for a pointer to an individual cell */
typedef cell_ptr Position;
```

```
/* The underlying hash table stucture */
struct hash_tbl
{
  Hash_size table_size;
  List *the_lists;
/* this will be an array of lists, allocated later */
/* The lists will use headers, allocated later */
};

/* The hash table */
/* - a pointer to the hash table stucture */
typedef struct hash_tbl *Hash_Table;
```

Function headers for open hashing

```
/* A hash function */
Index hash( Key_Type key, Hash_size table_size );

/* Initialise a table of given size */
Hash_Table initialize_table( Hash_size table_size );

/* Find a key in a hash table */
Position find( Key_Type  key, Hash_Table H );

/* Insert a key in a hash table */
void insert( Key_Type  key, Hash_Table H );

/* Delete a key from a hash table */
void delete( Key_Type  key, Hash_Table H );
```

Cost of hashing

Search time for a key k has two components

- Calculating $h(k)$
- Searching the linked list for k

Ignoring, for time being, calculation of $h(k)$, search time depends on

Maximum length of linked lists

= Maximum number of collisions for any k

Aim to choose size of hash table and hash function so that this is as small as possible

Hash functions

If hash table size is H, then need a function

$$h : K \longrightarrow 0 \ldots (H-1)$$

Should have properties

- It is easy to compute
- For given values in K, values of $h(k)$ are uniformly distributed over the range $0 \ldots (H-1)$

  For example, no use having $H = 1000$ and all values of $h(k)$ in the region 200 to 300.

These properties are difficult to achieve, since the keys themselves are often not randomly distributed

Example hash functions

Simple additive function. Adds character values (as ints) and reduces mod the table size.

```
Index hash1( Key_Type key, Hash_size table_size )
{
  unsigned int hash_val=0;

  while( *key != '\0' )
    hash_val += *key++;

  return( hash_val % table_size );
}
```

This function takes no account of the position of characters within the key

e.g. 'cat' and 'act' hash to the same value.

This function weights the second and third characters with powers of the size of the alphabet (in this case 27). It ignores all characters after the third.

```
Index hash2( Key_Type key, Hash_size H_SIZE )
{
  int hash_val;

  if ( key[0] == '\0' )
    hash_val = 0;
  else if ( key[1] == '\0' )
    hash_val = key[0] % H_SIZE ;
  else if ( key[2] == '\0' )
    hash_val =  (key[0] + 27*key[1]) % H_SIZE;
  else
    hash_val = (key[0] + 27*key[1] + 729*key[2]) % H_SIZE;
  return(hash_val);
}
```

For a key of length n, this function calculates the polynomial

$$c_0 * 32^{n-1} + c_1 * 32^{n-2} + \ldots c_{n-1} \bmod hashsize$$

```
Index hash3( Key_Type key, Hash_size table_size )
{
  unsigned int hash_val=0;

  while( *key != '\0' )
    hash_val = ( hash_val << 5 ) + *key++;

  return( hash_val % table_size );
}
```

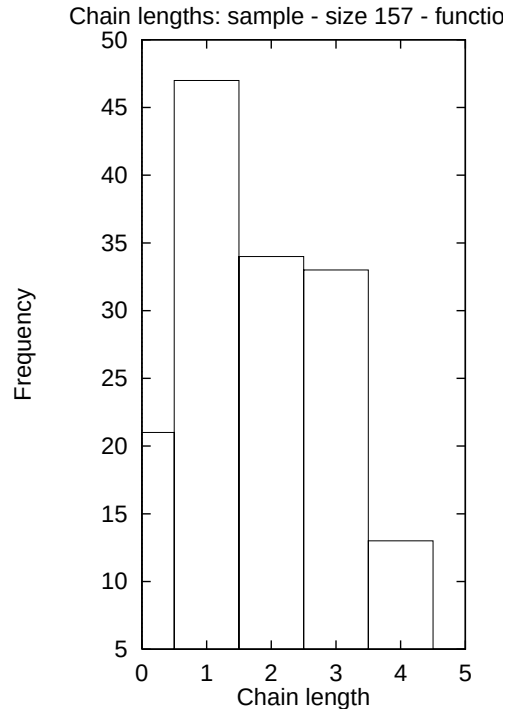
As `hash3` except we avoid possible overflows by reducing mod `table_size` each time.

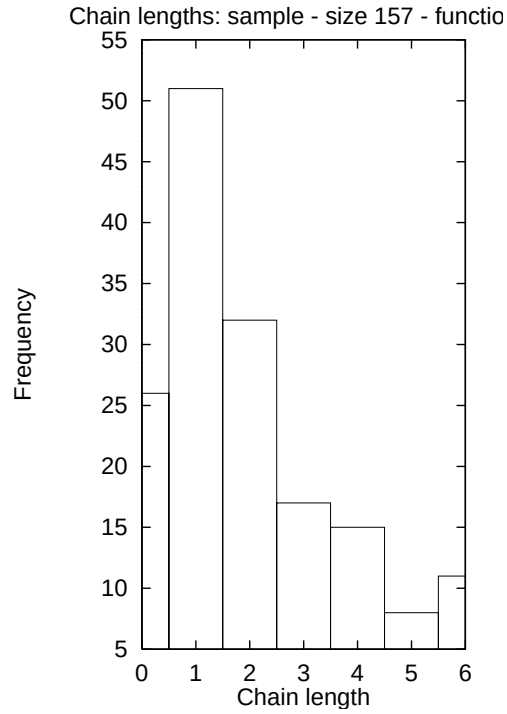
```
Index hash4( Key_Type key, Hash_size table_size )
{
  unsigned int hash_val=0;

  while( *key != '\0' )
    hash_val = ((hash_val << 5) + *key++) % table_size ;

  return(hash_val);
}
```

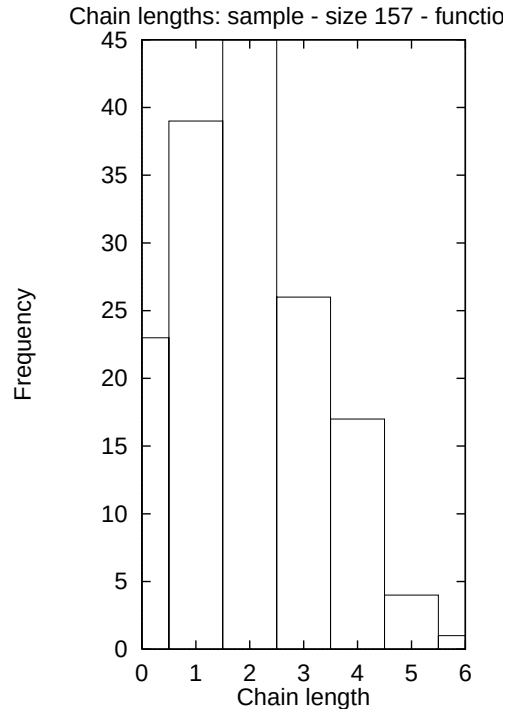
Hash function performance

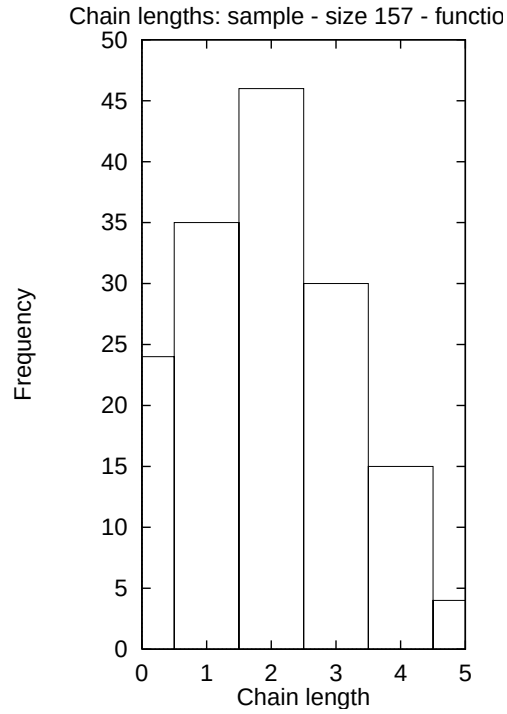
The following slides show the performance of the above hash functions under two sets of conditions

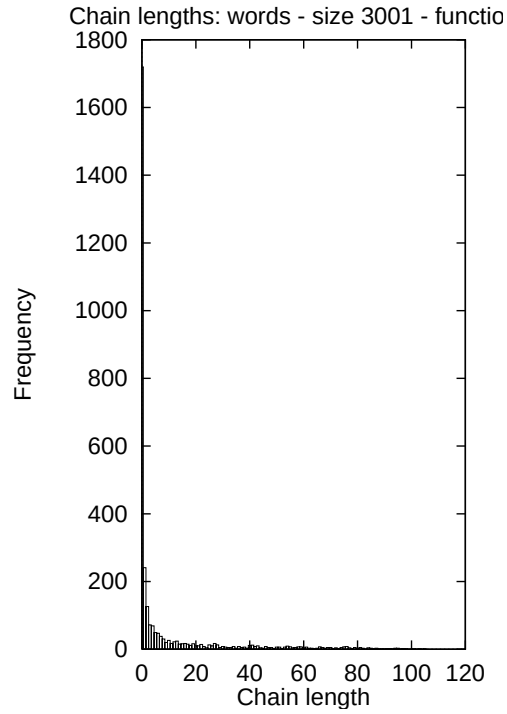
- Data set of 315 words, hash table size 157
- Data set of 25144 words (`/usr/dict/words`), hash table size 3001

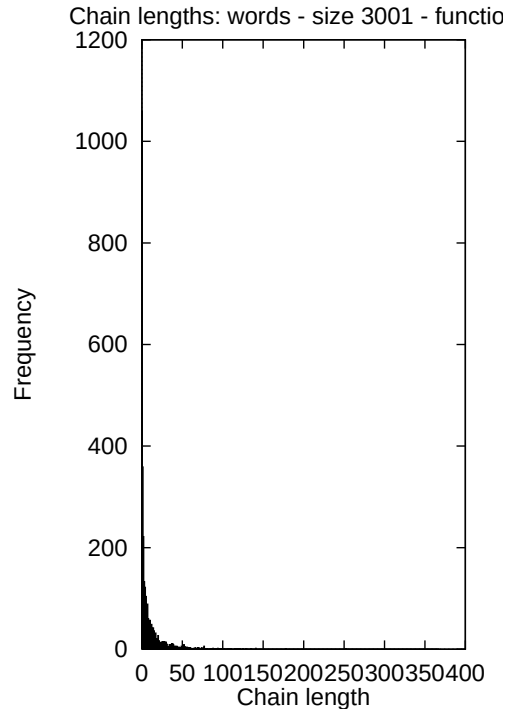

Chain lengths: sample - size 157 - functio
Frequency
Chain length
50
45
40
35
30
25
20
15
10
5
0
1
2
3
4
5

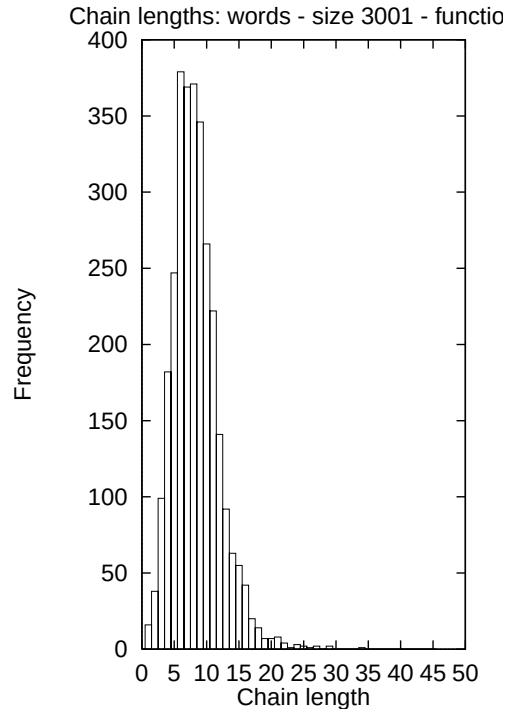

Chain lengths: sample - size 157 - functio
Frequency
55
50
45
40
35
30
25
20
15
10
5
0
1
2
3
4
5
6
Chain length

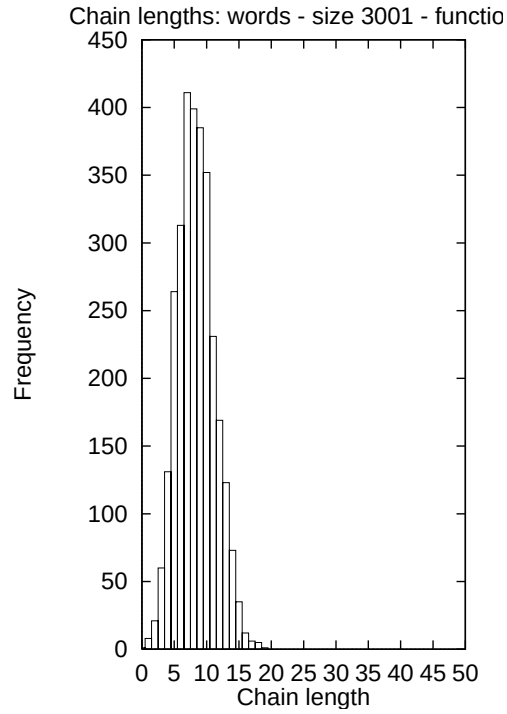

Chain lengths: sample - size 157 - functio
Frequency
Chain length
0
5
10
15
20
25
30
35
40
45
0
1
2
3
4
5
6


Chain lengths: sample - size 157 - functio
Frequency
Chain length
0
5
10
15
20
25
30
35
40
45
50
0
1
2
3
4
5


Chain lengths: words - size 3001 - functio
Frequency
1800
1600
1400
1200
1000
800
600
400
200
0
0 20 40 60 80 100 120
Chain length


Chain lengths: words - size 3001 - functio
1200
1000
800
600
400
200
0
Frequency
0 50 100150200250300350400
Chain length


Chain lengths: words - size 3001 - functio
Frequency
Chain length
0
50
100
150
200
250
300
350
400
0 5 10 15 20 25 30 35 40 45 50


Chain lengths: words - size 3001 - functio
Frequency
Chain length
0
50
100
150
200
250
300
350
400
450
0 5 10 15 20 25 30 35 40 45 50

Closed Hashing (Open addressing)

Open hashing has several disadvantages

- uses pointers, so have time cost of dynamic memory allocation
- needs separate data structure for chains, and code to manage it

Closed hashing has neither of these problems

- all data is stored in the hash table array itself
- when collision occurs, look elsewhere in the array for space

Given key k, hash function $hash$, adopt following procedure to search for k.

- Look at the location s_0 given by $hash(k)$
- If k found at location s_0, return s_0
- Otherwise, if location s_0 is empty, halt search, since k is not in the table
- Otherwise, use second function p to find s_1, given by

$$s_1 = hash(k) + p(1) \pmod{HashSize}$$

- Repeat above with

$$s_i = hash(k) + p(i) \pmod{HashSize}$$

until either find k or an empty cell.

The function p called *collision resolution function* or *probe* function

Simplest function is

$$p(i) = i$$

So $s_i = hash(k) + i$

Start at s_0 and look at consecutive cells until find space.

This is *linear probing*

Hash table insertion – linear probing

This example uses integer keys and the hash function

```
h(k) = k  mod hash_size
```

Data to be inserted is

```
23 3 2 22 45 68 24 47 91
```

Using *linear probing* gives

```
                 insertions --->
          23     3     2    22    45    68    24    47    91
     ==========================================================
       0 ||  23 |  23 |  23 |  23 |  23 |  23 |  23 |  23 |  23 |
       1 ||     |     |     |     |  45 |  45 |  45 |  45 |  45 |
^      2 ||     |     |   2 |   2 |   2 |   2 |   2 |   2 |   2 |
|      3 ||     |   3 |   3 |   3 |   3 |   3 |   3 |   3 |   3 |
|      4 ||     |     |     |     |     |  68 |  68 |  68 |  68 |
H      5 ||     |     |     |     |     |     |  24 |  24 |  24 |
A      6 ||     |     |     |     |     |     |     |  47 |  47 |
S      7 ||     |     |     |     |     |     |     |     |  91 |
H      8 ||     |     |     |     |     |     |     |     |     |
       9 ||     |     |     |     |     |     |     |     |     |
T     10 ||     |     |     |     |     |     |     |     |     |
A     11 ||     |     |     |     |     |     |     |     |     |
B     12 ||     |     |     |     |     |     |     |     |     |
L     13 ||     |     |     |     |     |     |     |     |     |
E     14 ||     |     |     |     |     |     |     |     |     |
      15 ||     |     |     |     |     |     |     |     |     |
|     16 ||     |     |     |     |     |     |     |     |     |
|     17 ||     |     |     |     |     |     |     |     |     |
v     18 ||     |     |     |     |     |     |     |     |     |
      19 ||     |     |     |     |     |     |     |     |     |
      20 ||     |     |     |     |     |     |     |     |     |
      21 ||     |     |     |     |     |     |     |     |     |
      22 ||     |     |     |  22 |  22 |  22 |  22 |  22 |  22 |
     ==========================================================
```

Everything OK until try to insert 45.

$$45 \bmod 23 = 22 \quad \text{full}$$

$$22+1 \bmod 23 = 0 \quad \text{full}$$

$$22+2 \bmod 23 = 1 \quad \text{insert}$$

Note that this produces a block of occupied cells – this effect is known as *clustering*.

Using linear probing means that if a new key is hashed *into the cluster*, need several probes to find space for it, and it then adds to the cluster.

One way to avoid this is to use *quadratic probing*.

$$p(i) = i^2$$

Hash table insertion – quadratic probing

Same data and hash function as before, but using *quadratic probing*

0	23	23	23	23	23	23	23	23	23
1							24	24	24
2			2	2	2	2	2	2	2
3		3	3	3	3	3	3	3	3
4									
5								47	47
6									
7									
8					45	45	45	45	45
9									
10									
11									
12									91
13									
14									
15						68	68	68	68
16									
17									
18									
19									
20									
21									
22				22	22	22	22	22	22

Again everything OK until try to insert 45.

$$45 \bmod 23 = 22 \quad \text{full}$$

$$(22+1^2) \bmod 23 = 0 \quad \text{full}$$

$$(22+2^2) \bmod 23 = 3 \quad \text{full}$$

$$(22+3^2) \bmod 23 = 8 \quad \text{insert}$$

Quadratic probing

Linear probing is *guaranteed* to find an empty space if one exists.

Not the case for quadratic probing

Consider the following hash table. It isn't full, but an attempt to insert 30, using hash function as before and quadratic probing, will fail.

```
==========
  0 || 10 |
  1 || 21 |
  2 ||    |
  3 ||    |
  4 || 34 |
  5 || 45 |
  6 || 56 |
  7 ||    |
  8 ||    |
  9 || 69 |
==========
```

$$30 \bmod 10 = 0 \quad \text{full}$$
$$(0+1^2) \bmod 10 = 1 \quad \text{full}$$
$$(0+2^2) \bmod 10 = 4 \quad \text{full}$$
$$(0+3^2) \bmod 10 = 9 \quad \text{full}$$
$$(0+4^2) \bmod 10 = 6 \quad \text{full}$$
$$(0+5^2) \bmod 10 = 5 \quad \text{full}$$
$$(0+6^2) \bmod 10 = 6 \quad \text{full}$$
$$(0+7^2) \bmod 10 = 9 \quad \text{full}$$

Never get any values other than $0, 4, 5, 6, 9$, so never hit free space

Can be shown that can guarantee to find a place for a new key k using quadratic probing if following conditions hold

- The table size is prime
- The table is no more than half full

This means that, if quadratic probing used, need table size to be at least twice as big as maximum number of data items

Double hashing

Another way of reducing the clustering problem, without the disadvantages of quadratic probing is *double hashing*.

Choose a second hash function, h_2, and use the collision resolution function

$$p(i) = h_2(k) * i$$

Need to make sure that $h_2(k)$ never takes the value 0 (Why?)

If the table size is prime and p is some prime smaller than the hash table size, the function

$$h_2(k) = p - (k \bmod p)$$

usually works well. (Note that we are assuming integer keys here, but can easily be generalised to string keys)

Using double hashing with the above example, with secondary hash function

$$p - (k \bmod p)$$

with $p = 7$

Can insert 10, 21, 34, 45, 56, 69 with no problems – now try to insert 70 and 19

0	10	10	10
1	21	21	21
2			
3			19
4	34	34	34
5	45	45	45
6	56	56	56
7		70	70
8			
9	69	69	69

$$h_2(70) = 7 - (70 \bmod 7) = 7$$

$$70 \bmod 10 = 0 \quad \text{full}$$

$$(0+7) \bmod 10 = 7 \quad \text{insert}$$

$$h_2(19) = 7 - (19 \bmod 7) = 2$$

$$19 \bmod 10 = 9 \quad \text{full}$$

$$(9+2) \bmod 10 = 1 \quad \text{full}$$

$$(9+2*2) \bmod 10 = 3 \quad \text{insert}$$

Data types for closed hashing

```
/* Type for size of hash table */
typedef unsigned int Hash_size;

/* Type for index into hash table */
typedef unsigned int Index;

/* Type for state of hash table entry */
enum kind_of_entry { legitimate, empty, deleted };

/* Type for hash table entry */
struct hash_entry
{
  Key_Type element;
  enum kind_of_entry state;
};
```

```
typedef struct hash_entry cell;

/* The underlying hash table stucture */
struct hash_tbl
{
  unsigned int table_size;
  cell *the_cells; /* an array of hash_entry cells,
                    * allocated during initialisation */
};

/* The hash table */
/* - a pointer to the hash table stucture */
typedef struct hash_tbl *Hash_Table;
```

Rehashing

Two major disadvantages to closed hashing

- Need to know upper bound on amount of data before building the hash table
- As the hash table fills up, more clashes occur and insertion becomes more expensive. At some stage insertion will fail (at best when the table is full).

Rather than make the hash table extremely large in first place, can get round both problems by *rehashing*.

This involves building a new hash table double the size of original, inserting all data from the original hash table into the new table, and freeing memory resources taken by old table.

This means that we are not limited by size of data expected in advance,

Can rehash if run out of space.

Rehash can take place either

- When becomes full – probably a bit late
- When table loading reaches some predetermined factor (70% often used)

Rehashing should be triggered automatically by an insert, without user intervention.

Easy to implement