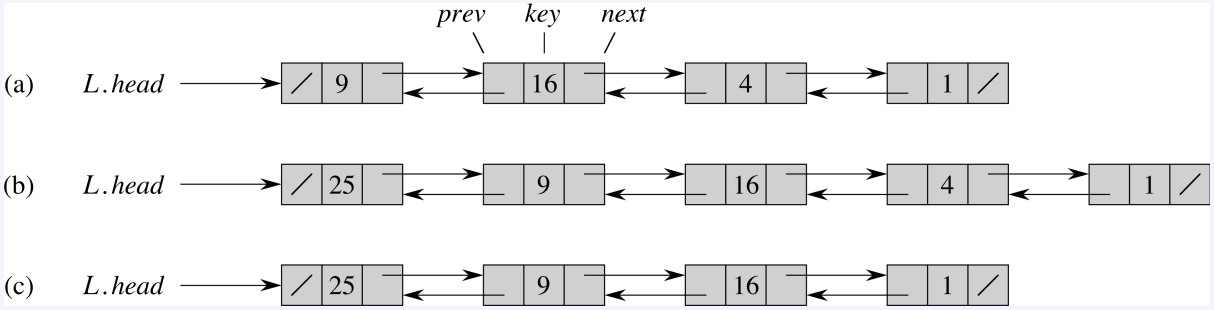
Elementary data structures

Stack : The element deleted is always the most recent one to have entered the data structure (LIFO). Insert and delete at the top

Queue : The element deleted is always the oldest one to have entered the data structure (FIFO). Insert at the end, delete at the beginning

Like heaps, stacks and queues can be implemented using an array

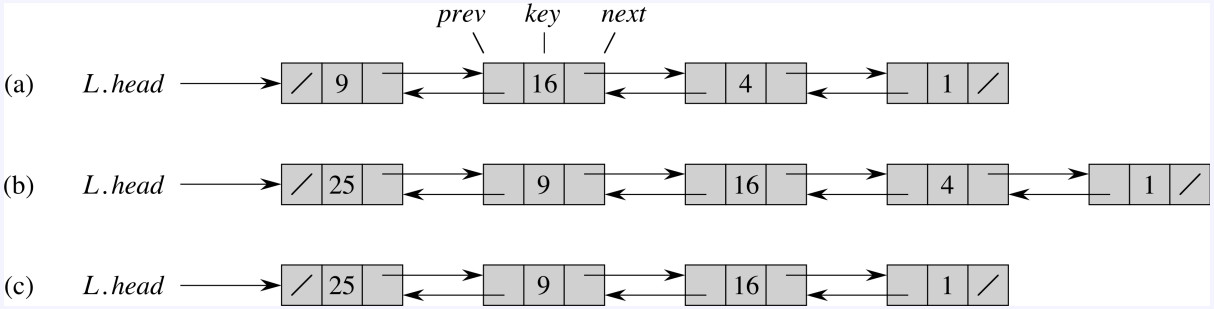
Elementary data structures



Link list : Elements in the set S are organized in a sequence connected by pointers

Doubly linked list : Can be traversed both from the beginning (head) or the end (tail)

Elementary data structures



To search an element in a link list, one needs a key, therefore we assume that each element has a *key* field

LIST-SEARCH(L, k)
  x = L.head
  while (x ≠ NILL and x. key ≠ k)
    x = x.next
  return x

Assumptions

It is common occurrence to assume that elements stored in a data structure have a key field which identify each element

- the key is used to find, delete, add the element in a data structure

Keys of elements are drawn from some set (Universe) U of keys

This set could be a range over the positive integers, physical memory addresses or the strings on an alphabet (like the strings based on the English alphabet that act like keys in the symbol table of a program)

Direct addressing

The key is used to index the entries in the data structure. Suppose :

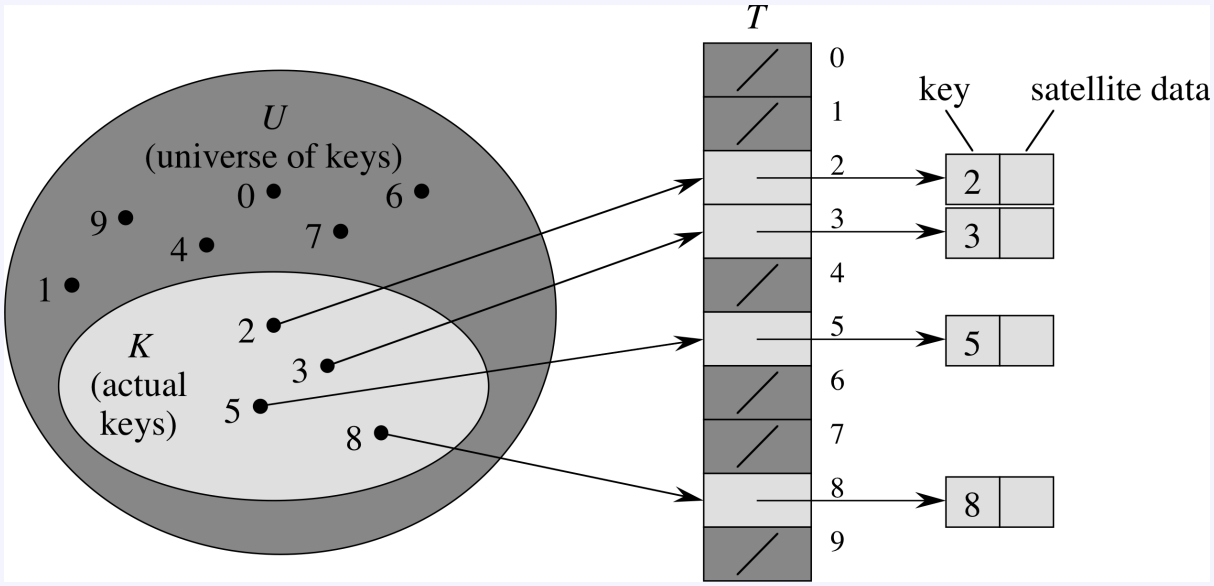
- the set of keys is the range $0..m-1$
- keys are distinct

The idea :

- Set up an array $T[0..m-1]$, the same size as the range of keys
- $T[i] = x$ if $x \in T$ and $key[x] = i$
- $T[i] = NULL$ otherwise

This is called a direct-addressing table

Direct addressing example



Operations take $O(1)$ time

SEARCH(T, k)
return T[k]

INSERT(T, x)
T[key[x]] = x

DELETE(T,x)
T[key[x]] = NILL

Problem with direct addressing

Direct addressing works well when the range m of keys is relatively small

But if the keys are 32-bit integers

- Problem 1 : direct-address table will have 2^{32} entries, more than 4 billion
- Problem 2 : even if memory is not an issue, the time to initialize the elements to NULL may be

Solution : map the universe of keys to a smaller range $0..m-1$

This mapping is performed by a *hash function*

Hash table issues

Solution : map the universe of keys to a smaller range $0..m-1$ called a *hash table*

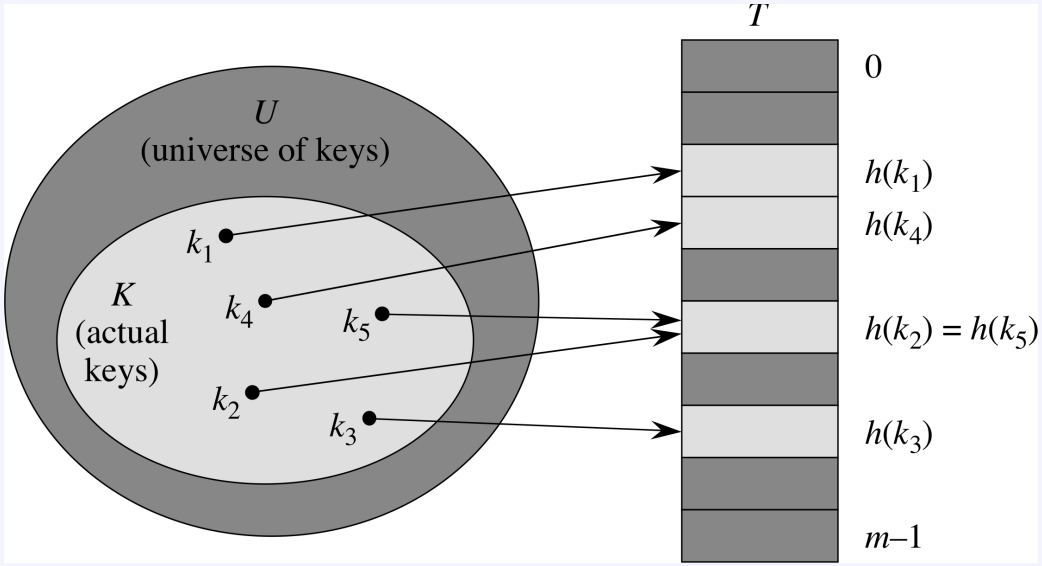
This mapping is performed by a hash function

- How to compute hash functions.
  - We will look at the multiplication and division methods.
- The hash function could map multiple keys to the same table index.
  - We will look at two techniques to address these "collisions" : chaining and open addressing.

Mapping keys to smaller ranges

Instead of storing an element with key k in index k, use a function h and store the element in index $h(k)$

- The function h is the hash function.
- $h : U \to \{0, 1, \ldots, m-1\}$ so that h[k] is a legal index in T.
- We say that k hashes to index $h(k)$



Collisions

Two or more keys hash to the same index in the hash table

Can happen when there are more possible keys than entries ($|U| > m$).

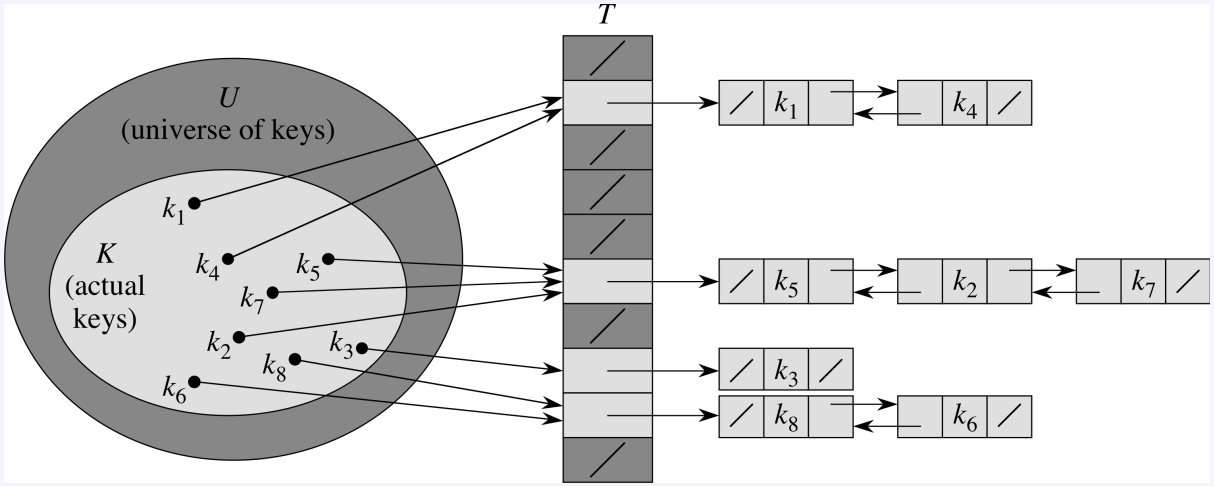
For a given set U of keys with $|U| < m$, may or may not happen.
Definitely happens if $|U| > m$.

Must be prepared to handle collisions in all cases. Use two methods : chaining and open addressing.

Chaining is usually better than open addressing.

Chaining

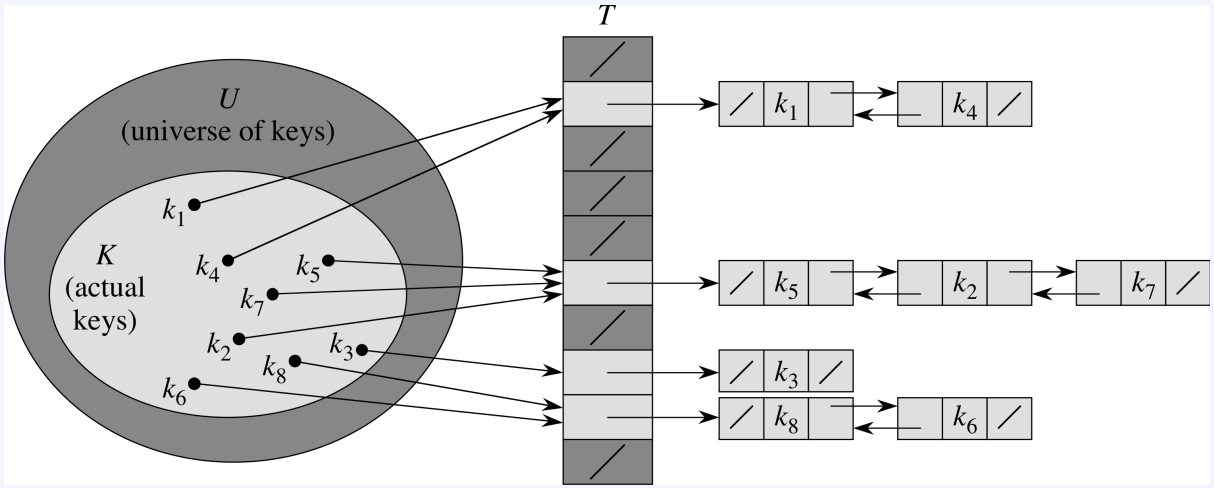
Chaining puts elements that hash to the same index in a linked list :



SEARCH(T, k)
search for an element with key k in list $T[h(k)]$. Time proportional to length of the list of elements in $h(k)$
INSERT(T, x)
insert x at the head of the list $T[h(key[x])]$. Worst-case $O(1)$

Chaining

Chaining puts elements that hash to the same index in a linked list :



DELETE(T,x)

delete x from the list $T[h(key[x])]$. If lists are singly linked, then deletion takes as long as searching, $O(1)$ if doubly linked

Analysis of running time for chaining

load factor : Given n keys and m indexes in the table : the $\alpha = n/m =$ average # keys per index

Worst case : All keys hash in the same index, creating a list of length n. Searching for a key takes $\Theta(n)$ + the time to hash the key. We don't do hashing for the worst case !

Average case : Assume simple uniform hashing : each key in table is equally likely to be hashed to any index

- The average cost of an unsuccessful search for a key is $\Theta(1+\alpha)$
- The average cost of a successful search is $\Theta(1+\alpha/2) = O(1+\alpha)$

Analysis of running time for chaining

If the size m of the hash table is proportional to the number n of elements in the table, say $\frac{1}{2}$, then $n = 2m$, and we have $n \in O(m)$

Consequently, $\alpha = \frac{n}{m} = \frac{O(m)}{m} = O(1)$, in other words, we can make the expected cost of searching constant if we make α constant

Choosing a hash function

A good hash function satisfies (approximately) the assumption of **simple uniform hashing** :

"*given a hash function h, and a hash table of size m, the probability that two non-equal keys a and b will hash to the same slot is* $P(h(a) = h(b)) = \frac{1}{m}$"

In order words, each key is equally likely to hash in any of the m slots of the hash table

Under the assumption of uniform hashing, the load factor α and the average chain length of a hash table of size m with n elements will be $\alpha = \frac{n}{m}$

Hash functions : the division method

$h(k) = k \bmod m$, i.e. hash k into a table with m entries using the index given by the remainder of k divided by m

Example : $k \bmod 20$ and $k = 91$, then $h(k) = 11$.

It's fast, requires just one division operation.

Disadvantage of the division method

$h(k) = k \bmod m$

Disadvantage : Have to avoid certain values of m :

- Powers of 2. If $m = 2^p$ for integer p, then $h(k)$ is just the least significant p bits of k. Examples : $m = 8 = 2^3$
  - h(52) = 4 : h(110100) = 100
  - h(37) = 5 : h(100101) = 101
- The implication is, for example, all keys that end with 100 map to the same index
- Solution : pick table size m = a prime number not too close to a power of 2 (or 10). Example $m = 5$
  - h(52) = 2 : h(110100) = 010
  - h(36) = 1 : h(100100) = 001

Division method : example

More examples with the division method $h(k) = k \bmod m$

Recall : A number in base 10 is
$6789 = (6 \times 10^3) + (7 \times 10^2) + (8 \times 10^1) + (9 \times 10^0) = 6789$

Keys may be strings of characters. Have to be converted into integers using base 128 (7-bit ASCII values). For "CLRS", ASCII values are : C = 67, L = 76, R = 82, S = 83

Interpret CLRS as
$(67 \times 128^3) + (76 \times 128^2) + (82 \times 128^1) + (83 \times 128^0) = 141,764,947$

Hash fct : division method

If $m = 2^p$ for integer p, then $h(k)$ is just the least significant p bits of k.

$128 = 2^7$

So CLRS as $(67 \times 128^3) + (76 \times 128^2) + (82 \times 128^1) + (83 \times 128^0)$
mod $2^7 = 10000111001100101001\mathbf{1010011} = 83 = 1010011$

ABCS as $(65 \times 128^3) + (66 \times 128^2) + (67 \times 128^1) + (83 \times 128^0)$
mod $2^7 = 10000011000010100001\mathbf{1010011} = 83 = 1010011$

Hash fct : division method

If k is a character string interpreted in base 2^p (as in CLRS example), then $m = 2^p - 1$ is a poor choice as well : permuting characters in a string does not change its hash value (Exercise 11.3-3).

For example CLRS $\mod 2^7 - 1 =$ RLCS $\mod 2^7 - 1$. Assume $m = 2^7 - 1 = 127$

CLRS as $(67 \times 128^3) + (76 \times 128^2) + (82 \times 128^1) + (83 \times 128^0)$ $\mod 12^7 = 54$

SRLC as $(83 \times 128^0) + (67 \times 128^3) + (76 \times 128^2) + (82 \times 128^1)$ $\mod 12^7 = 54$

$m = 9$ is a poor choice for integer keys. Try keys 123, 321, 231

Hash fct : multiplication method

1. Choose a constant A in the range $0 < A < 1$
2. Multiply key k by A
3. Extract the fractional part of kA
4. Multiply the fractional part by m
5. Take the floor of the result

$h(k) = \lfloor m(kA \bmod 1) \rfloor$
Examples for $m = 8 = 2^3$ and $A = 0.6180$:

$$
\begin{aligned}
h(52) &= \lfloor 8(52 \times 0.6180 \bmod 1) \rfloor \\
&= \lfloor 8(32.136 \bmod 1) \rfloor \\
&= \lfloor 8(0.136) \rfloor \\
&= \lfloor 1.088 \rfloor \\
&= 1
\end{aligned}
\qquad
\begin{aligned}
h(36) &= \lfloor 8(36 \times 0.6180 \bmod 1) \rfloor \\
&= \lfloor 8(22.248 \bmod 1) \rfloor \\
&= \lfloor 8(0.248) \rfloor \\
&= \lfloor 1.984 \rfloor \\
&= 1
\end{aligned}
$$

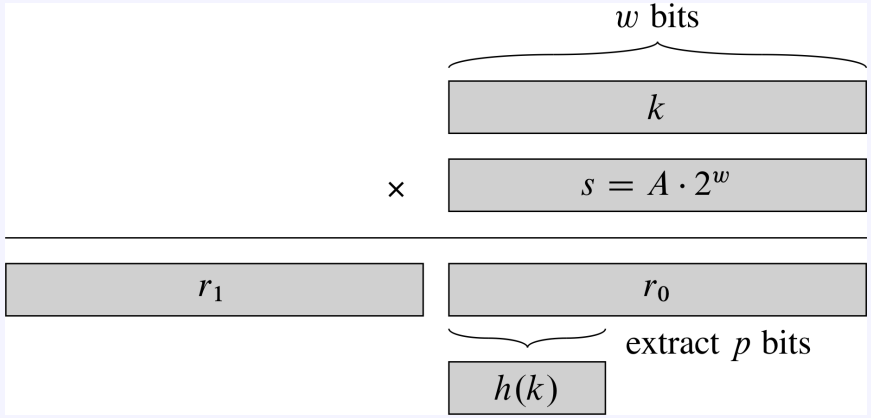
Multiplication method : advantage and disadvantage

- Choose $m = 2^p$ for implementation reasons
- Choose A not too close to 0 or 1
- Knuth : Good choice for $A = (\sqrt{5} - 1)/2$

Disadvantage : Slower than division method

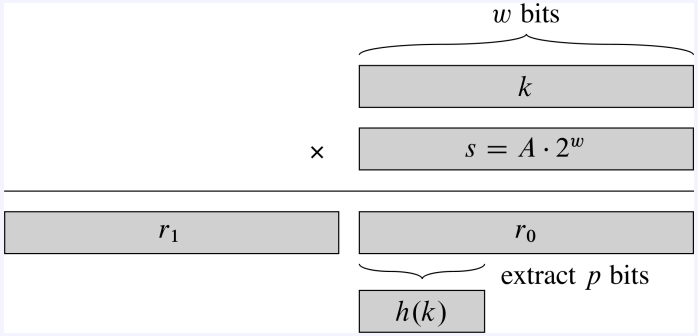
Advantage : Value of m not critical

Multiplication method : implementation



- w bits is the memory word size ; $0 < s < 2^w$
- Multiplying two w-bit words, the result is 2w bits, $r_1 2^w + r_0$
- r_1 is the high-order word of the product and r_0 is the low-order word
- $\lfloor m(kA \bmod 1) \rfloor$ are the p most significant bits of r_0

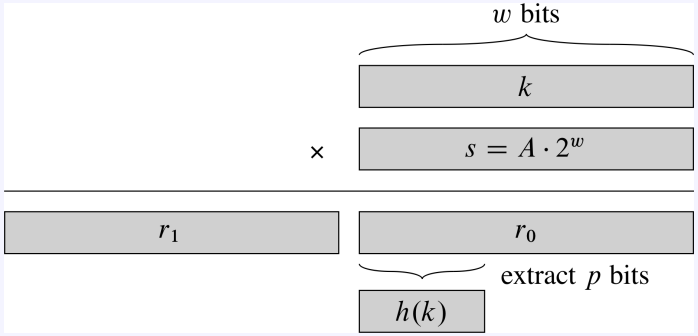
Multiplication method : implementation



Example : $m = 8 = 2^3$ $(p = 3)$; $w = 5$; $k = 21$; $0 < s < 2^5$; $s = 13 \Rightarrow A = 13/32 = 0.40625$

$h(21)$	=	$\lfloor 8(21 \times 0.40625 \bmod 1) \rfloor$	=	$ks = 21 \times 13 = 273$
	=	$\lfloor 8(8.53125 \bmod 1) \rfloor$	=	$8 \times 2^5 + 17$
	=	$\lfloor 8(0.53125) \rfloor$	=	$r_1 = 8, r_0 = 17 = 10001$
	=	$\lfloor 4.25 \rfloor$	=	takes the $p = 3$ most significant bits of r_0
	=	4	=	$100 = 4$

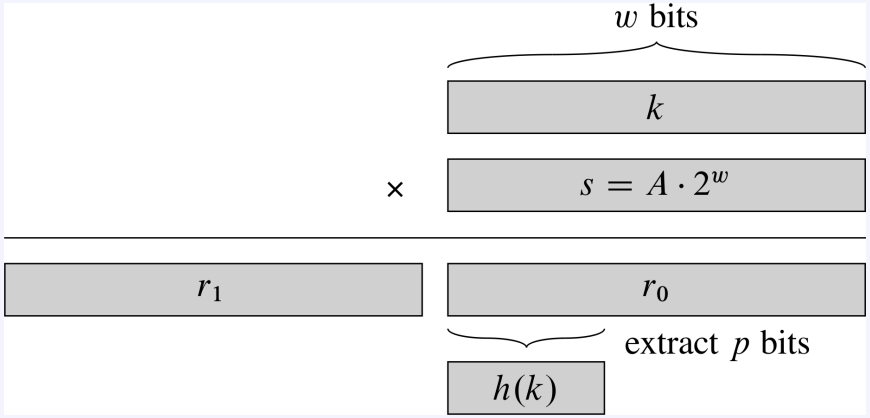
Multiplication method : implementation



Example : $m = 8 = 2^3$ $(p = 3)$; $w = 6$; $k = 52$; $0 < s < 2^6$; $A = 0.625 \Rightarrow s = 0.625 \times 2^6 = 40$

$h(52)$	=	$\lfloor 8(52 \times 0.625 \bmod 1) \rfloor$	=	$ks = 52 \times 40 = 2080$
	=	$\lfloor 8(32.5 \bmod 1) \rfloor$	=	$2080 = 32^6 + 32$
	=	$\lfloor 8(0.5) \rfloor$	=	$r_1 = 32, r_0 = 32 = 100000$
	=	$\lfloor 4 \rfloor$	=	takes the $p = 3$ most significant bits of r_0
	=	4	=	$100 = 4$

Multiplication method : implementation



Example :
$m = 8 = 2^3$; $w = 6$; $k = 52$; $0 < s < 2^6$; $s = 40$; $A = 0.625$

We extract the p most significant bits of r_0 because we have selected the table to be of size $m = 2^p$, therefore we need p bits to generate a number in the range $0..m-1$.

Open addressing

When collisions are handled through chaining, the keys that collide are placed in a link list in the same entry where the collision occurred

Open addressing place the keys that collide in another entry of the hash table by searching an empty slot in hash table using a *probe sequence*

The hash function is modified to include a *probe number* :
$h : U \times \{0, 1, \ldots, m-1\} \rightarrow \{0, 1, \ldots, m-1\}$

The probe sequence $\langle h(k,0), h(k,1), \ldots, h(k,m-1) \rangle$ should be a permutation of $\langle 0, 1, \ldots, m-1 \rangle$ so every position in the table is eventually considered

Open addressing : Insertion

```
HASH-INSERT(T, k)
  i = 0
  repeat
    j = h(k, i)
    if T[j] == NIL
      T[j] = k
      return j
    else i = i+1 // probing
  until i == m
  error "hash table overflow"
```

Open addressing : Searching

```
HASH-SEARCH(T, k)
  i = 0
  repeat
    j = h(k, i)
    if T[j] == k
      return j
    else i = i+1
  until T[j] == NIL or i == m
  return NIL
```

Computing probe sequences

Probe sequences are dealing with collisions. We examine three techniques to generate probe sequences :

- Linear probing
- Quadratic probing
- Double hashing

Linear probing

$h(k,i)$ is based on an ordinary hash function $h' : U \to \{0,1,\ldots,m-1\}$. Thus

$h(k,i) = (h'(k)+i) \mod m$ for $i = 0,1,\ldots,m-1$, thus $h(k,0)$ is the initial hashing. If $h(k,0) \neq NIL$, then there is a collision.

Linear probing resolves collisions by looking at the next entry in the table.

Assume we have the following table T $(h' = k \mod 11)$:

1	2	3	4	5	6	7	8	9	10	11
NIL	44	NIL	29	NIL	52	81	56	NIL	NIL	19

Linear probing

A call to $h(28, 0) = 6$ is a collision. The linear probing strategy will examine entries 7, 8 and finally 9. $h(28, 1) = 7$

1	2	3	4	5	6	7	8	9	10	11
NIL	44	NIL	29	NIL	52	81	56	NIL	NIL	19
						↑				

$h(28, 2) = 8$

1	2	3	4	5	6	7	8	9	10	11
NIL	44	NIL	29	NIL	52	81	56	NIL	NIL	19
							↑			

$h(28, 3) = 9$

1	2	3	4	5	6	7	8	9	10	11
NIL	44	NIL	29	NIL	52	81	56	NIL	NIL	19
								↑		

Quadratic probing

Linear probing suffers from primary clustering : long runs of occupied sequences build up : an empty slot that follows i full slots has probability $\frac{i+1}{m}$ to be filled.

Quadratic probing jumps around in the table according to a quadratic function of the probe number : $h(k,i) = (h'(k) + c_1 i + c_2 i^2) \bmod m$, where $c_1, c_2 \neq 0$ are constants and $i = 0, 1, \ldots, m-1$. Thus, the initial position probed is $T[h'(k)]$.

Issue : Can get a milder form of clustering : secondary clustering : if two distinct keys have the same $h(key)$ value, then they have the same probe sequence.

Quadratic probing

Assuming $c_1 = c_2 = 1$, in this case, the probe sequence will be $h(k,i) = h'(k) + c_1 i + c_2 i^2 \bmod m = h'(k), h'(k)+2, h'(k)+6$, etc.

$h(28,1) = 8$

1	2	3	4	5	6	7	8	9	10	11
NIL	44	NIL	29	NIL	52	81	56	NIL	NIL	19
							↑			

$h(28,2) = 1$

1	2	3	4	5	6	7	8	9	10	11
28	44	NIL	29	NIL	52	81	56	NIL	NIL	19
↑										

Quadratic probing : Issue

Can get a milder form of clustering : secondary clustering : if two distinct keys have the same $h(key)$ value, then they have the same probe sequence.

Double hashing

Use two auxiliary hash functions, h_1 and h_2.

h_1 gives the initial probe, and h_2 gives the remaining probes :
$h(k, i) = (h_1(k) + ih_2(k)) \bmod m$

Issue : Must have $h_2(k)$ be relatively prime to m (no factors in common other than 1) in order to guarantee that the probe sequence is a full permutation of $[0, 1, \ldots, m-1]$

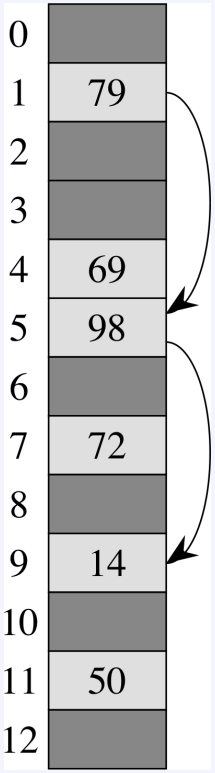
- Could choose m to be a power of 2 and h_2 to always produce an odd number > 1.
- or m prime and h_2 to always produce an integer less than m

Handling collisions : Double hashing

$h(k,i) = (h_1(k) + ih_2(k)) \bmod m$

Example : $m = 13, k = 14, h_1(k) = k \bmod 13$,
$h_2(k) = 1 + (k \bmod 11)$. For $i = 0$

$$\begin{aligned} h(14,0) &= h_1(14) + 0(h_2(14)) \\ &= (14 \bmod 13) + \\ &\quad 0(1 + (14 \bmod 11)) \\ &= 1 + 0(1 + 3) \\ &= 1 \end{aligned}$$

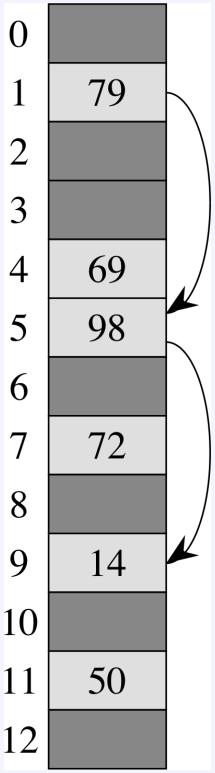


Handling collisions : Double hashing

$h(k,i) = (h_1(k) + ih_2(k)) \bmod m$

Example : $m = 13, k = 14, h_1(k) = k \bmod 13$,
$h_2(k) = 1 + (k \bmod 11)$. For $i = 1$

$$\begin{aligned} h(14,1) &= h_1(14) + 1(h_2(14)) \\ &= (14 \bmod 13) + \\ &\quad 1(1 + (14 \bmod 11)) \\ &= 1 + 1(1 + 3) \\ &= 5 \end{aligned}$$

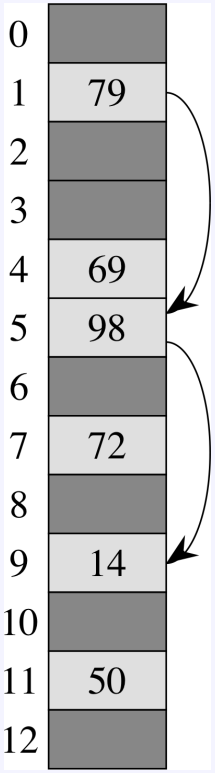


Handling collisions : Double hashing

$h(k,i) = (h_1(k) + ih_2(k)) \bmod m$

Example : $m = 13, k = 14, h_1(k) = k \bmod 13$,
$h_2(k) = 1 + (k \bmod 11)$. For $i = 2$

$$\begin{aligned} h(14,2) &= h_1(14) + 2(h_2(14)) \\ &= (14 \bmod 13) + \\ &\quad 2(1 + (14 \bmod 11)) \\ &= 1 + 2(1 + 3) \\ &= 9 \end{aligned}$$



Open addressing : Deleting

Use a special value DELETED instead of NIL when marking a index as empty during deletion.

- Suppose we want to delete key k at index j.
- And suppose that sometime after inserting key k, we were inserting key k', and during this insertion we had probed index j (which contained key k).
- And suppose we then deleted key k by storing NIL into index j .
- And then we search for key k'.
- During the search, we would probe index j before probing the index into which key k' was eventually stored.
- Thus, the search would be unsuccessful, even though key k' is in the table.