Lecture 17

- ✔ Improving open-addressing hashing
- ✔ Brent's method
- ✔ Ordered hashing

Improving open addressing hashing

- Recall the average case unsuccessful and successful find time costs for common open-addressing schemes (α is load factor N/M)...
- Random hashing, double hashing:

$$U_\alpha = \frac{1}{1-\alpha}$$

$$S_\alpha \approx \frac{1}{\alpha} \ln \frac{1}{1-\alpha}$$

- Linear probing:

$$U_\alpha \approx \frac{1}{2}\left(1 + \frac{1}{(1-\alpha)^2}\right)$$

$$S_\alpha \approx \frac{1}{2}\left(1 + \frac{1}{(1-\alpha)}\right)$$

- It is possible to improve these, and it makes sense to do so in some applications...

Improving successful find time cost

- It is common to have databases in which keys are inserted only once, followed by many successful searches and relatively few unsuccessful searches. Examples:
  - spell checker (insert all words from the dictionary once; then as each word is typed into a document, do a usually-successful find to check its spelling... not found means a misspelled word)
  - compiler symbol table (insert a variable when it is declared; then when a variable is used, do a usually-successful find to check its type... not found means an undeclared variable error)
- *Brent's method* is a variant of double hashing that greatly improves average-case successful find time cost (but increases insert time cost)

Improving unsuccessful find time cost

- It is also common to have databases in which keys are inserted only once, followed by many unsuccessful searches and relatively few unsuccessful searches. Examples:
  - stolen credit card database (insert credit card number when it is reported stolen; then when a customer presents a credit card, do a usually-unsuccessful find to check its status... successful find means a stolen card)
  - noise-word list (insert "noise" words once, at initialization time; then when a word is read in a document, do a usually-unsuccessful find to check if it is a noise word)
- *Ordered hashing* is a variant of double hashing that improves average-case unsuccessful find time cost

Brent's method

- Brent's method for hashing [R. P. Brent, 1973] is a variation on double hashing that improves the average-case time for *successful* searches

- In fact, the average-case successful search time is bounded < 2.5 probes even when the table is full (load factor $\alpha = 1$)!

- The tradeoff is that the insert operation becomes somewhat more expensive, but amortized analysis shows this can be worth it if inserts are rare and successful searches are common

Toward Brent's method: start with double hashing

- Consider a 5-cell table, open addressing, double hashing
- Consider 3 keys: k1, k2, k3
- Suppose the primary and secondary hash functions applied to these keys give these values:

  H(k1) = 0, H2(k1) = 4
  H(k2) = 3, H2(k2) = 1
  H(k3) = 0, H2(k2) = 3

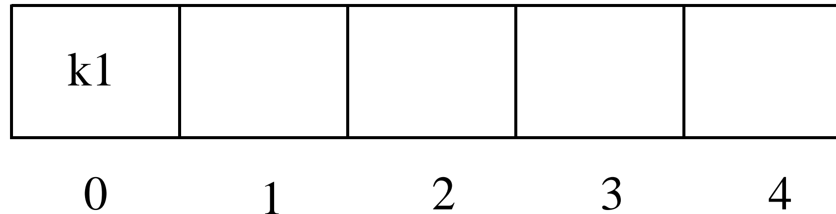
  ... so the probe index sequences for these keys are as follows:

  k1: 0, 4, 3, 2, 1
  k2: 3, 4, 0, 1, 2
  k3: 0, 3, 1, 4, 2

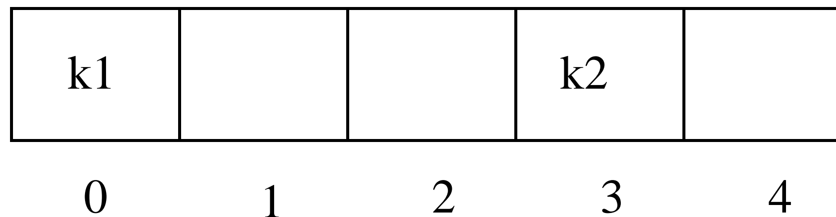
- Now consider the total probe sequence lengths for successful searches in the table as these keys are inserted

Successful search probe path lengths, double hashing

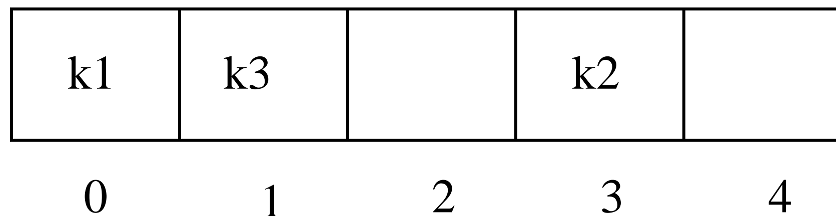
- After inserting k1, the total successful probe path length is 1



- After also inserting k2, the increase in total successful probe path length is 1, to total is now 1+1 = 2; average probe path length is 2/2 = 1



- Now insert k3. The increase in total successful probe path length is 3; total is now 1+1+3 = 5; average probe path length is 5/3



Brent's method: basic idea

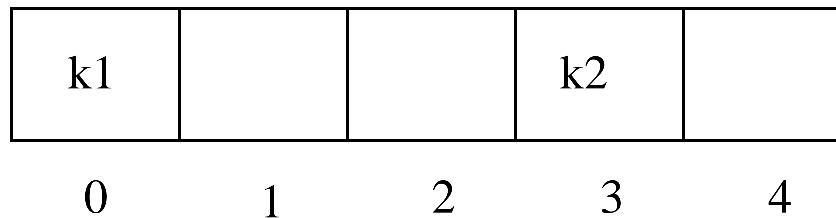
- Brent's idea is to move items out of the probe path of the to-be-inserted key, if it can be determined that this will reduce the overall total successful probe path length

- Consider inserting a new key K:
  - If K's first probe location is empty, just insert K there (this increases overall successful probe path length by 1, which is the best possible)
  - If K's first probe location is full (collision), and second probe location is empty, just insert K there (this increases overall successful probe path length by 2, which also cannot be improved by Brent's method...)
  - But suppose K collides with a key K1 at K's first probe location and with another key K2 at K's second location:
    - Think about moving K1 to K1's next probe location, and putting K in its place!
    - If K1's next location is empty, this will increase K1's successful probe path length by 1, and K's probe sequence will have length 1...
    - This increases the total successful probe path length by 2, which is better than what would happen with regular double hashing in this case (an increase of at least 3)
- This is the basic idea behind Brent's method. Let's look at an example

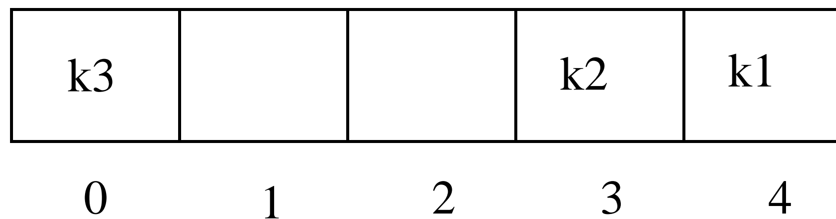
Brent's method: example 1

- Recall the probe sequences
  k1: 0, 4, 3, 2, 1
  k2: 3, 4, 0, 1, 2
  k3: 0, 3, 1, 4, 2

- k1 and k2 have been inserted:



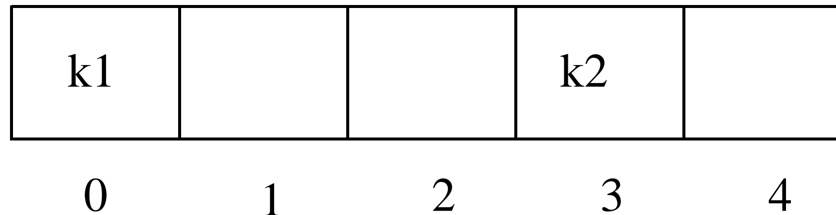
- Now insert k3. We can put it in its first probe location, and move k1 to its next probe location, which is empty:



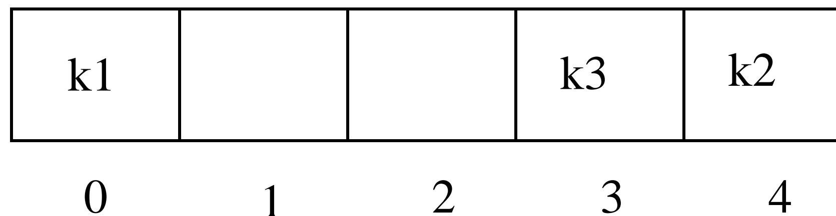
- Total successful search path length is now 2+1+1 = 4; average probe path length is 4/3

Brent's method: example 2

- Same probe sequences...
- k1 and k2 have been inserted:



- Now insert k3. We can put it in its second probe location, and move k2 to its next probe location, which is empty:



- Total successful search path length is now 1+2+2 = 5; average probe path length is 5/3

Brent's method: example 3

- Same probe sequences...
- k1 and k2 have been inserted:

k1			k2	
0	1	2	3	4

- Now insert k3. We can put it in its third probe location, which is empty. (This is the same result as regular double hashing on this sequence of inserts.)

k1			k3	k2
0	1	2	3	4

- Total successful search path length is now 1+2+2 = 5; average probe path length is 5/3
- Of these 3 possibilities, we should pick the one in Example 1, which gives the smallest increase in (and so the best resulting overall total and average) successful find path length. In fact, we wouldn't even have to check the last 2 examples to know this!

Brent's algorithm

- Suppose a new key K to be inserted has probe index sequence $p_1, p_2, \ldots, p_{v-1}, p_v$, where location p_v is the first location in the sequence that is empty
- Let k_i be the key in location p_i. Suppose the size of the table is M.
- Brent's method operates as follows:
- If v < 3, just insert K in location p_v as usual; can't do better than that.
- Else, for c = 1,...,v-2 do the following:
  - For d = 1,...,c do the following:
    - Consider moving k_d out of the way. At this stage, we will consider (c-d+1) probes to find an empty location for k_d :

      if location (p_d + (c-d+1) * H2(k_d)) mod M is empty, then move k_d to that location, and put K in location p_d. Done!

      This makes the path length for finding the new key K be d, and increases the path length for finding k_d by c-d+1. So the total increase in successful-find path length is c+1, which is less than v

Analysis of Brent's method

- Brent's algorithm does not always find optimal positions for the keys; it is a simple heuristic that works well on average
  - for one thing, note that when moving a key out of the way, Brent's does not consider also moving keys it collides with...

- Brent's method does not change the average case number of probes for unsuccessful search; it is the same as double hashing:

$$U_\alpha \approx \frac{1}{1-\alpha}$$

- Brent's method does improve the average case number of probes for successful search compared to double hashing (that is the whole point). One can show that this number is bounded, independent of load factor:

$$S_\alpha < 2.49$$

- Of course, insert is now more expensive, $O(N^2)$ worst-case! But this is worth it, if successful searches are much more common than inserts

Ordered hashing

- Ordered hashing [Knuth and Amble, 1974] is a variation on double hashing that improves the average-case time for *unsuccessful* searches
- The average-case time for successful searches is unchanged; in fact, the two become the same
- This technique requires comparable keys
- The tradeoff is that the insert operation becomes more expensive by a constant factor, but this can be worth it

Ordered hashing: basic idea

- In an unsorted linked list with N items, an unsuccessful search take N steps: the list must be searched exhaustively

- However, if the list is sorted in decreasing order, the search could terminate sooner... As soon as you see a list item smaller than the target, you know the target can't be in the list

- This makes unsuccessful search take 1 step in the best case, N/2 in the average case, and N steps worst case: same as successful search

- Ordered hashing sets up probe sequences for any successful search so that the keys belonging to that probe sequence will be in decreasing order (consider an empty table location to have a value smaller than any key)

- Clearly, this will have the desired result... if we can do it!
- One way to do it would be: Know all the keys to be inserted in advance. Sort them. Insert them in decreasing order, using the usual double-hashing insert algorithm
- But we would like an insert algorithm that works "on line", without knowing the keys in advance...

The ordered hashing algorithm

- An "on-line" insert algorithm for ordered hashing is quite simple: "bump" any smaller item along its probe sequence, until finding an empty location...
  ... This never decreases the key value stored in any location in the table, so searches in a later probe sequence will never be stopped too soon

- To insert (or find) a key K in a table of size M, with primary hash function H() and secondary function H2():

  1. Set indx = H(K); offset = H2(K)
  2. While location indx is not empty, and location indx holds a key larger than K:
     set indx = (indx + offset) mod M.
  3. If table location indx is not empty, and holds a key equal K: no need to insert (or successful find). Done!
  3. Else if table location indx is empty, insert key there (or unsuccessful find). Done!
  4. Else collision with a smaller key, say K2. We want to bump it for insert (or unsuccessful find):
     Put K in K2's position,
     set offset = H2(K2), and set K=K2.
     Go to 2.

Ordered hashing: an example

- Insert this sequence of keys in the table: 145, 293, 397, 458, 553
- Use ordered hashing, with H(K) = ten's digit of K, H2(K) = unit's digit of K

0	1	2	3	4	5	6	7	8	9	10

- Now search for 454...

Analysis of ordered hashing

- Ordered hashing does not change the average case number of probes for successful search; it is the same as double hashing:

$$S_\alpha \approx \frac{1}{\alpha} \ln \frac{1}{1-\alpha}$$

- Ordered hashing does improve the average case number of probes for unsuccessful search compared to double hashing (that is the whole point), making it the same as for successful search:

$$U_\alpha \approx \frac{1}{\alpha} \ln \frac{1}{1-\alpha}$$

- Insert still requires finding an empty table location, and so the average case number of probes for insert is the same as for double hashing:

$$I_\alpha \approx \frac{1}{1-\alpha}$$

.... though more memory operations are required, because of key swapping in the table

Next time

- Self-organizing data structures
- Self-organizing lists
- Splay trees
- Spatial data structures
- K-D trees
- The C++ Standard Template Library

Reading: Weiss, Ch. 4 and Ch. 12