HASHING METHODS

Hanan Samet

Computer Science Department and
Center for Automation Research and
Institute for Advanced Computer Studies
University of Maryland
College Park, Maryland 20742
e-mail: hjs@umiacs.umd.edu

HASHING OVERVIEW

- Task: compare the value of a key with a set of key values in a table
- Conventional solutions:
  1. use a comparison on key values (tree-based)
  2. branching process governed by the digits comprising the key value (trie-based)
- Alternative solution is to find a 1-1 mapping (i.e., function) from set of possible key values to a memory address and use table lookup methods to retrieve the record — $O(1)$ process
- Problem: the set of possible key values is much larger than the number of available memory addresses
  1. developing the 1-1 function h is time-consuming as it requires puzzle-solving abilities
     - result is called a perfect hashing function
  2. once h is found, addition of a single key value may render the function meaningless
     - need to develop it anew
  3. can replace h by a program, which may itself be time-consuming to compute
- Result: usually abandon goal of finding 1-1 mapping and use a special method to resolve any ambiguity (i.e., when more than one key value is mapped to the same address — termed a *collision*)

HASHING

- Def: to "mess things up"
- Hashing function $h(k)$ is used to calculate address where to start the search for the record with key value k
- Issues
  1. what kind of a function is $h(k)$?
     - easy and fast to compute
     - minimize the number of collisions
  2. what if $h(k)$ does not yield the desired result?
     - how to handle collisions
- Assume table of size m and $0 \le h(k) < m$
- Example hashing functions:
  1. division techniques
     - often use $h(k) = k \bmod m$
     - choice of m is important
       a. m even
          - bad as $h(k)$ even when k even and odd when k odd
       b. m is a power of the radix of alphanumeric set of character values
          - bad as only least significant characters matter
          - with $m=r^3$, `ABCDEF`, `IJKDEF`, and `KLMDEF` all hash to the same location
       c. usually choose m to be prime
  2. multiplicative techniques
     - entire key value is used
     - examples:
       a. multiply fields and take modulo
       b. add or exclusive-or of fields

SEPARATE CHAINING

- Hash table of size *m*
- One chain (linked list) for each of *m* hash values containing all elements that hash to that location (known as a *collision list*)
- Hash chains are known as *buckets*
- Hash table locations are known as *bucket addresses*
- For *n* key values, average chain size is *n*/*m*
- One chain (linked list) for each of *m* hash values
- Retrieval
  1. use sequential search through chain
  2. speed up unsuccessful search by sorting chain by key value
  3. speed up successful search by self-organizing methods
     - move key value to start of chain each time it is accessed
- Ex:

$h(k)$	NAME	k=KEY	NEXT
0	JIM	49	Λ
1	JOHN	22	Λ
2	RAY	30	Λ
3	SUZY	3	Λ
4			
5			
6	LUCY	41	Λ

JANE	14	Λ

1. add JANE(14)→0
2. add LUCY(41)→6

IN-PLACE CHAINING

- When *m* is large, many of the chains are empty
- Use empty locations in table for the chain
- Must be able to distinguish between free and occupied locations
- Insertion algorithm:
  1. if key value not present, then allocate a free location
  2. link location to chain which was unsuccessfully searched
- Ex:

$h(k)$	NAME	k=KEY	NEXT
0	JIM	49	~~Λ~~ ~~6~~ 5
1	JOHN	22	Λ
2	RAY	30	Λ
3	SUZY	3	Λ
4			
5	~~LUCY~~ JANE	~~41~~ 14	Λ
6	~~JANE~~ LUCY	~~14~~ 41	~~Λ~~ ~~5~~ Λ

  1. add `JANE`(14)→0 which collides with `JIM`(49)→0
  2. add `LUCY`(41)→6 which collides with `JANE`(14)→0 which is stored at 6
     - result in coalescing of chains of `JANE` and `LUCY` making unsuccessful search longer as several chains must be searched
- Can avoid coalescing by moving `JANE` just before adding `LUCY`

IN-PLACE CHAINING INSERTION ALGORITHM

```
location procedure
CHAINING_WITH_COALESCING_INSERTION(k);
begin
  value key k;
  integer i;
  global integer r;
  /* r is the most recently allocated location */
  global hashtable table;
  i←h(k);
  if OCCUPIED(table[i]) then
    begin
      while NOT(NULL(NEXT(table[i])) do
        begin
          if k=KEY(table[i]) then return(i)
          else i←NEXT(table[i]);
        end;
      if k=KEY(table[i]) then return(i);
      while OCCUPIED(table[r]) do r←r−1;
      if r≤0 then return(`OVERFLOW')
      else
        begin
          NEXT(table[i])←r;
          i←r;
        end;
    end;
  MARK(table[i],`OCCUPIED');
  KEY(table[i])←k;
  NEXT(table[i])←NIL;
  return(i);
end;
```

LAMPSON'S IN-PLACE CHAINING

- Avoid extra space for `NEXT` field by not storing entire key value with record
- $k = m \cdot q(k) + h(k)$, $q(k) = \lfloor k/m \rfloor$, $h(k) = k \bmod m$
- Store $q(k)$ in table instead of k
- Can compute k given m, $q(k)$, and $h(k)$,
- Ex: $0 \leq k < 2^{32}$

$q(k)$	$h(k)$
0 … 21	22 … 31

- Since only compare $q(k)$, all elements in same collision list must have the same value of $h(k)$ and thus no coalescing is allowed
- Data structure:
  1. circular collision lists
  2. flag `FIRST` denoting if first element on collision list
  3. pointer `NEXT` to next element in circular list with same $h(k)$ value
- Ex:

$h(k)$	NAME	k=KEY	FIRST	$q(k)$	NEXT
0	JIM	49	T	7	~~0~~ ~~6~~ 5
1	JOHN	22	T	3	1
2	RAY	30	T	4	2
3	SUZY	3	T	0	3
4					
5	JANE	14	F	2	0
6	~~JANE~~ LUCY	~~14~~ 41	~~F~~ T	~~2~~ 5	~~0~~ 6

  1. add `JANE`(14)$\rightarrow$0
  2. add `LUCY`(41)$\rightarrow$6 but 6 contains `JANE`
     - if at least one element of the hash chain starting at 6 exists, then it must be stored there
     - must move `JANE` as it does not belong in 6
- Nice compromise between use of a key value as an index to a table, which is impossible due to large number of possible key values, and storing the entire key value as in a conventional hashing method

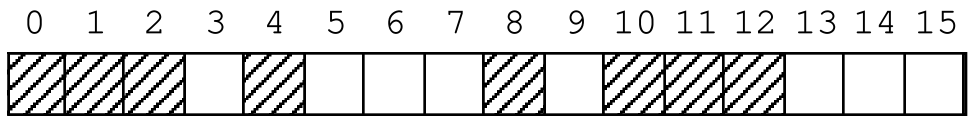
OPEN ADDRESSING

- Like chaining but `NEXT` link field is open or unspecified
- Probe sequence: set of locations comprising collision list of a key
- Goal: cycle through all locations with little or no duplication
- Linear probing: $h(k)$, $h(k)+1$, $h(k)+2$, …, $m-1$, 0, 1, $h(k)-1$
- Insertion Algorithm:
  1. calculate hash address i
  2. if `TABLE`(i) is empty then insert and exit; else $i \leftarrow i+1$ mod m and repeat step 2 until exhausting `TABLE`
- Ex:

h (k)	NAME	*k*=KEY	DELETED
0	JIM	49	N
1	JOHN	22	N
2	~~RAY~~	~~30~~	Y
3	SUZY	3	N
4	JANE	14	N
5			N
6	LUCY	41	N

1. adding `JANE`(14)$\rightarrow$0 yields a collision; cyclic probe sequence causes its insertion in 4
2. adding `LUCY`(41)$\rightarrow$6
3. delete `RAY`(30)$\rightarrow$2
   - problem: if look up `JANE` then don’t find her since a collision exists at location 0, and probe sequence finds location 2 unoccupied
   - solution: add `DELETED` flag to each entry to halt the search during insertion but not during lookup

PILEUP PHENOMENON



- Next key to be inserted goes into one of the vacant locations
- Not all vacant locations are equally probable
- Ex: insert k into location 3 if $0 \leq h(k) \leq 3$
  insert k into location 6 if $h(k)=6$
  3 is four times as likely as 6
- Coalescing in open addressing with linear probing can lead to big growth (e.g., inserting into location 9 makes the list of 8 grow by 4)
- Different from in-place chaining where coalescing causes a list to grow by only one element
- Pileup phenomenon arises whenever consecutive values are likely to occur
- Overcome by a number of techniques:
  1. use additive constant instead of 1
     - should be relatively prime to m so can cycle through table
  2. use a pseudo random number generator to create successive offsets from $h(k)$ (*random probing*)
     - make sure cycle through table
     - *uniform hashing* = when all possible configurations of empty and occupied locations are equally likely = model of hashing for comparing various methods

ANALYSIS OF PERFORMANCE

- n= number of key values in table
- m= maximum size of table
- $\alpha = n/m$ = load factor
- Expected number of probes for successful search:

α	linear probing	random probing (under uniform hashing model)	separate chaining
	$(1-\alpha/2)/(1-\alpha)$	$-(\ln(1-\alpha))/\alpha$	$1+\alpha/2$
0.1	1.06	1.05	1.05
0.5	1.50	1.34	1.25
0.75	2.50	1.83	1.375
0.9	5.50	2.56	1.45

QUADRATIC PROBING

- Alternative to linear probing
- Avoids primary clustering
- $h_i = (h(k)+i^2) \bmod m$
- Locations in the probe sequence can be computed with no multiplication
  1. h_i is location of element i
  2. $h_{i+1} = (h_i + d_i \bmod m)$ where $d_0 = 1$ and $d_{i+1} = d_i + 2$
- Theorem: if m is prime, then quadratic probing will search through at least 50% of the table before seeing a particular location again
- Ex: m=7
  $h_0 = 0$, $h_1 = 1$, $h_2 = 4$
  $h_3 = 9 \bmod 7 = 2$ and $h_4 = 16 \bmod 7 = 2$
- Proof:
  1. let probes i, j probe the same location (assume $i \neq j$)
  2. $i^2 \bmod m = j^2 \bmod m$
  3. $(i^2 - j^2) \bmod m = (i+j) \cdot (i-j) \bmod m = 0 \bmod m$
  4. but i,j are both $< m$ implying $(i-j) \neq c \cdot m$
  5. therefore, $i+j = c \cdot m$ and i or j must be at least $m/2$ since probe sequence starts with i=1, and recycling of values won't occur until at least 50% of table has been searched
- Sequence differs from one obtained from the pseudo random number generator as the pseudo random number generator guarantees that every location will be on a probe sequence

DOUBLE HASHING

- Use an additional hash function $g(k)$ to generate a constant increment for the probe sequence
- Probe sequence for key value k:
  $p_0 = h(k)$
  $p_1 = (h(k) + g(k)) \bmod m$
  $p_2 = (h(k) + 2 \cdot g(k)) \bmod m$
  $p_3 = (h(k) + 3 \cdot g(k)) \bmod m$
  …
  $p_i = (h(k) + i \cdot g(k)) \bmod m$
- $h(k)$ and $g(k)$ should be independent
- $g(k)$ generates values between 1 and $m-1$
- Two different key values will have the same value for h and g with probability $O(1/m^2)$ instead of $O(1/m)$
- Key value k is stored at any one of the locations along its probe sequence
- Key values stored along the probe sequence of k are not necessarily part of k's probe sequence
- Ex: key values s and t can both hash to location u
  key value s: $u = (h(s) + c \cdot g(s)) \bmod m$
  key value t: $u = (h(t) + d \cdot g(t)) \bmod m$

SELF-ORGANIZING DOUBLE HASHING

- Collision lists are long as each location is frequently on the collision lists of many different key values
- Develop techniques for rearranging the elements on the collision lists so that subsequent searches are shorter
- Assume records are retrieved many times once inserted into the table hence it pays to rearrange the collision lists
- Assume trying to insert key value k and probe locations p_0 , p_1 , … p_i , … p_t, where $p_i = (h(k)+i \cdot g(k)) \bmod m$ before finding location p_t empty
- $p_i = (h(k)+i \cdot g(k)) \bmod m$
- Each p_i $(0 \leq i < t)$ is also part of a hash chain consisting of locations: $(p_i + j \cdot g(\mathtt{KEY}(p_i))) \bmod m$ for arbitrary j
- Assume $c_i = g(\mathtt{KEY}(p_i))$, and $p_i = h(\mathtt{KEY}(p_i)) \bmod m$:

p_0	p_1	p_2	p_3	p_4	$\cdots$	p_t
p_0+c_0 • 1	p_1+c_1 • 2	p_2+c_2 • 4	p_3+c_3 • 7	p_4+c_4 • 11		
p_0+2c_0 • 3	p_1+2c_1 • 5	p_2+2c_2 • 8	p_3+2c_3 • 12			
p_0+3c_0 • 6	p_1+3c_1 • 9	p_2+3c_2 • 13				
p_0+4c_0 • 10	p_1+4c_1 • 14					
p_0+5c_0 • 15						

- Actually, $p_i = (h(\mathtt{KEY}(p_i)) + d_i \cdot g(\mathtt{KEY}(p_i))) \bmod m$
- Brent algorithm: try to insert key value k in one of p_i and move the contents of p_i to an empty location along its probe sequence (column) so as to minimize the effective incremental search cost
- Order for testing candidate locations for moving (along diagonals)

EXAMPLE OF BRENT ALGORITHM

- Attempt to insert `RUDY`

p_0=TIM RUDY	p_1=ALAN RUDY	p_2=JAY RUDY	p_3=KATY RUDY	p_4=?
JOAN	RUTH	ϕ JAY	KIM	ϕ
RON	ϕ ALAN	ϕ	ALEX	ϕ
RITA	ϕ	ϕ	BOB	ϕ
ϕ TIM	ϕ	ϕ	ϕ KATY	ϕ

- First free location in `RUDY`'s probe sequence is p_4 for an increase in cost of 4
- Examine locations in diagonal order for first free location
- Alternatively, find first free location in each hash chain and check if overall search cost is increased by a movement of it
- Movement must result in an increase <4 in the search cost
- Moving `TIM` increases the search cost by 4
- Moving `ALAN` increases its search cost by 2, while increasing that of `RUDY` by 1 for a total of 3
- Moving `JAY` increases its search cost by 1, while increasing that of `RUDY` by 2 for a total of 3
- Moving `KATY` increases its search cost by 4, while increasing that of `RUDY` by 3 for a total of 7
- Move `JAY` as its increase (3) was the least and was encountered first
- Need 2.5 probes on the average for successful search
- Average number of probes for unsuccessful search is not reduced (as high as $(m+1)/2$ when the table is full)

GONNET-MUNRO ALGORITHM

- More general than the Brent algorithm
- Brent algorithm only attempts to move records on the probe sequence of the record being inserted

p_0=TIM	p_1=ALAN	p_2=JAY	p_3=KATY	p_4=?
JOAN	RUTH	ϕ	KIM	ϕ
RON	ϕ	ϕ	ALEX	ϕ
RITA	ϕ	ϕ	BOB	ϕ
ϕ	ϕ	ϕ	ϕ	ϕ

- Gonnet-Munro attempts to move in several stages instead of just one stage
  1. RUDY to TIM, TIM to JOAN, and JOAN to ?
  2. RUDY to ALAN, ALAN to RUTH, and RUTH to?
- Need remaining hash chains

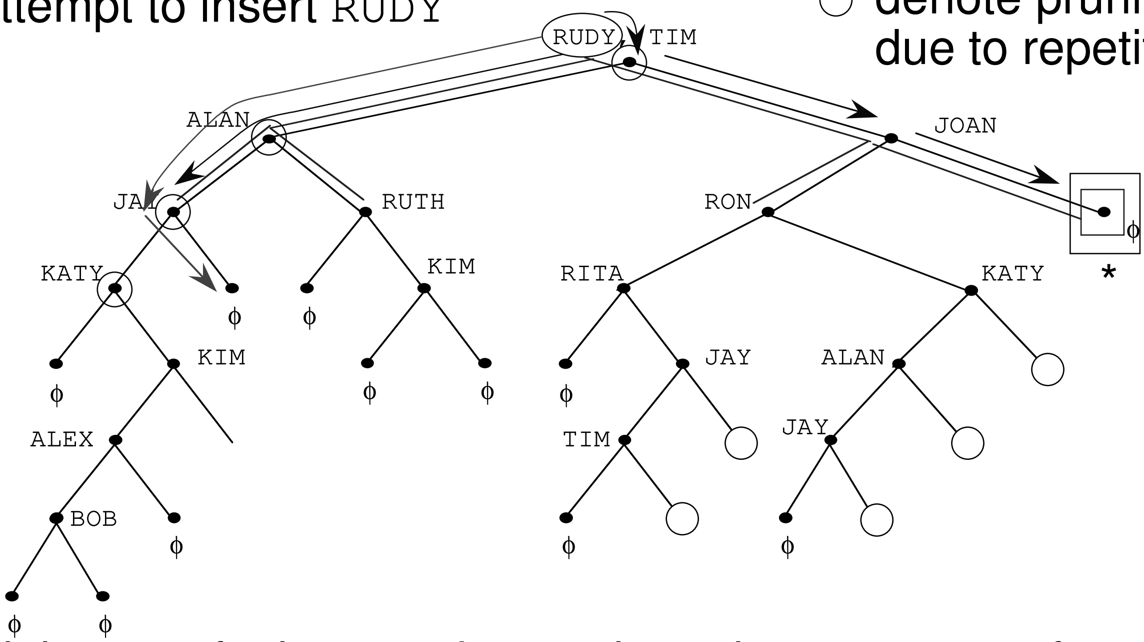
JOAN	RITA	KIM	RUTH	RON	ALEX	BOB
ϕ	JAY	ϕ	KIM	KATY	ϕ	ϕ
ϕ	TIM	ϕ	ϕ	ALAN	ϕ	ϕ
ϕ	ϕ	ϕ	ϕ	JAY	ϕ	ϕ
ϕ	ϕ	ϕ	ϕ	ϕ	ϕ	ϕ

- Can visualize search for best movement as a binary tree
- Best movement is the closest empty node to the root

EXAMPLE OF GONNET-MUNRO ALGORITHM

- Attempt to insert `RUDY`

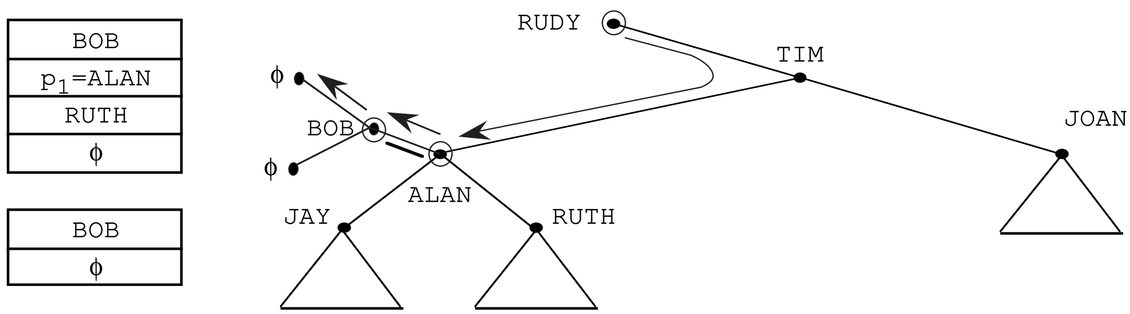
○ denote pruning due to repetition



- Right son of *a* is next element in probe sequence of `KEY`(*a*)
- Left son of *a* is next element in probe sequence of *a* and *a*'s father
- Search generates the tree level by level and chooses the first empty node as the final target for a sequence of relocation steps
- `RUDY` can be relocated to any position in the leftmost part of the tree
- Apply the relocation step as many times as needed to get to desired empty node
- Optimal solution moves `RUDY` to `TIM`, `TIM` to `JOAN`, and `JOAN` to its empty right son
  1. increase in total search cost is 2
  2. better than 3 obtained by Brent algorithm
- Brent algorithm only applies one step and thus must find the empty node in just one iteration
  1. move `RUDY` to `JAY`; `JAY` to empty right son of `JAY`
  2. increase in total search cost is 3

SHORTCOMING OF GONNET-MUNRO ALGORITHM

- Only moves records in forward direction along their hash chains
- Sometimes can reduce the cost by moving backward along the chain
- Ex: suppose `ALAN` is not in the first position along the hash chain starting at *h*(`ALAN`) mod *m* and that `BOB` immediately precedes `ALAN` along the hash chain, although *h*(`ALAN`) ≠ *h*(`BOB`)



- Optimal solution moves `RUDY` to `ALAN`, `ALAN` to `BOB`, and `BOB` to its empty son
  1. increase in total search cost is 1
  2. better than 2 obtained by Gonnet-Munro algorithm
- Requires a ternary tree
  1. need an additional link from *a* to the previous element in the probe sequence of `KEY`(*a*)
     - e.g., `ALAN` to `BOB`
  2. search process interprets previous links as indicating a decrease in cost
  3. if incoming link to *a* is a “previous element” link, then left son of *a* is the prior element in probe sequence of *a* and its father
  4. search process is more complex and empty node at closest level to the root no longer represents the cheapest relocation sequence
  5. optimal solution may require exhaustive search

SUMMARY

- Advantages
  1. separate chaining is superior with respect to the number of probes but need more space
  2. open addressing with linear probing results in more accesses but this is compensated by its simplicity
  3. compares favorably with other search methods as the search time is bounded as the number of records increases (provided the table does not become too full)
- Disadvantages
  1. size of hash table is usually fixed
     - have to worry about rehashing
     - separate chaining with overflow buckets is good
     - use linear hashing or spiral hashing which just split one bucket and rehash its contents instead of rehashing the entire table
  2. after an unsuccessful search we only know that the record is not present
     - we don’t know about the presence or absence of other records with similar key values such as the immediate predecessor or successor
     - contrast with B-trees and other methods based on binary search which maintain the natural order of the key values and permit processing along this order
     - order-preserving hashing methods such as those used to deal with multiattribute data (and spatial data) are an exception
  3. deletion may be cumbersome (e.g., open addressing)
  4. only efficient on the average
     - contrast with B-tree methods which have guaranteed upper bounds on search time, etc.
     - need faith in probability theory!