Hashing

- Let K be a key space - e.g., set of all alphanumeric strings
- Want to perform following typical operations on keys - insert, lookup and possibly delete
- Let T be a table of m memory locations in which we can store (key,pointer) pairs. T is called a **hash table**.
- Let h be a function that maps keys into table locations; i.e., h: K --> [1,m]. h is called a hash function.
  - time to compute h independent of |K|
  - ideally, then, lookup should take constant time
- For any key, K_j, we store the key (and a pointer to the record containing any information associated with the key) at location $h(K_j)$ in the hash table

Hashing

- Suppose the set of keys to be inserted into the table is known in advance
  - simplest case is when these keys have a "regular" structure
  - example: Keys are all pairs of upper case characters.
    - there are 26^2 keys
    - $h(c_1c_2) = 26*(index(c_1)-1) + index(c_2)$
    - this is the standard lexicographic storage allocation function
    - so sequential memory allocation for arrays is a special (uninteresting) case of hashing

Hashing

- But what if the "known" keys are some random sample of 12 character strings?
  - then it would be very difficult to identify a "perfect" hashing function, h, whose computational time complexity is independent of the size of the set of keys.
    - could arrange the keys in a binary search tree, and use some ordering of the nodes in the tree to define a hashing function, but then it would take log(n) time to compute h.

Hashing

- So, we have to define some general hashing function, independent of the elements of K that we happen to insert into the table
  - notice, though, that if the number of keys we insert is comparable in size to the size of the table, T, then it is VERY unlikely that our hashing function would be 1-1, so that some subsets of keys will hash to the same address in T. When $h(k_1) = h(k_2)$ we have a **collision**
  - alternatively, we can make |T| >> the number of keys we might ever insert - wastes storage and we still might have collisions

Hashing

- Key technical problems:
  - What is a good hash function?
  - When two keys hash to the same table location how do we resolve the collision?
    - Generally, the time to retrieve a key in a hash table is a function of the collision pattern it exhibits with respect to the previously inserted keys
    - And how do memory hierarchies impact this?
  - When we insert a key, is there a way to re-arrange the keys to make subsequent lookups more efficient?
  - What do we do if the table "fills up" (if its **load** factor exceeds some critical value for good performance)?

Simple hashing functions

- A good hashing functions should :

  a) be easy to compute - have constant time complexity

  b) spread the keys out evenly (uniformly) in the hash table - if key k is drawn randomly, probability that h(k) = i should be 1/m, where m = |T|, and independent of i.

- If k is regarded as an integer, then h(k) = kmod m is a good simple method

  a) if keys are alphabetic, then m should not be a power of 2 since kmod m will then be the lower order bits of k, independent of the rest of k

  b) ordinarily choose m to be prime (several reasons, TBD)

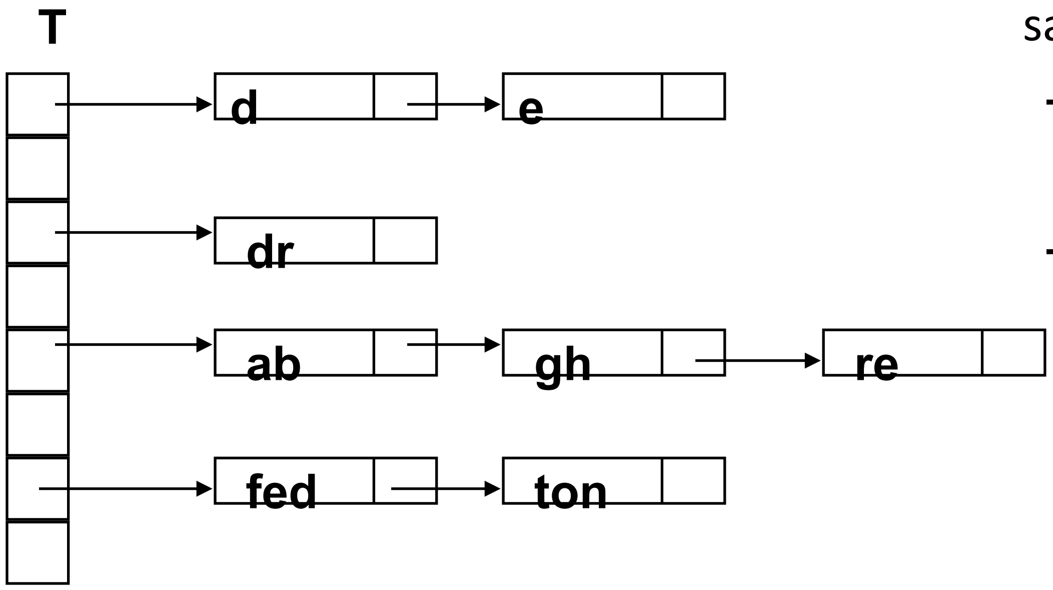
Simple hashing functions

- K mod m works very well when the keys are really uniformly distributed from the space of all possible keys.
- But as the set of keys deviates from uniform, its performance degrades rapidly – get lots of collision even though the table is nearly empty.

Conflict resolution by chaining

- Separate chaining: T[i] does not store a single key , but is a pointer to a list of all keys encountered such that h(k) = i.
- If the number of collisions is small, a simple linked list is sufficient to store all colliding keys
- Data structure is on two levels - hash table that divides entries into m linked lists.
- Lists are referred to as **buckets**.
- During a lookup or insert operation, each access to the data structure is called a **probe**.

Separate chaining



- average number of probes to find a key in the table is (4*2 + 3*3 +4) / 8 = 2.67
- For a hash table with m locations, if we insert n records into the table, then we say the load factor of the table is a = n/m
  - For the table above, a = 1.0
  - For a table with separate chaining, a can be less than or greater than 1. Other methods will only lead to a < 1.
  - Expected cost of search is proportional to load factor

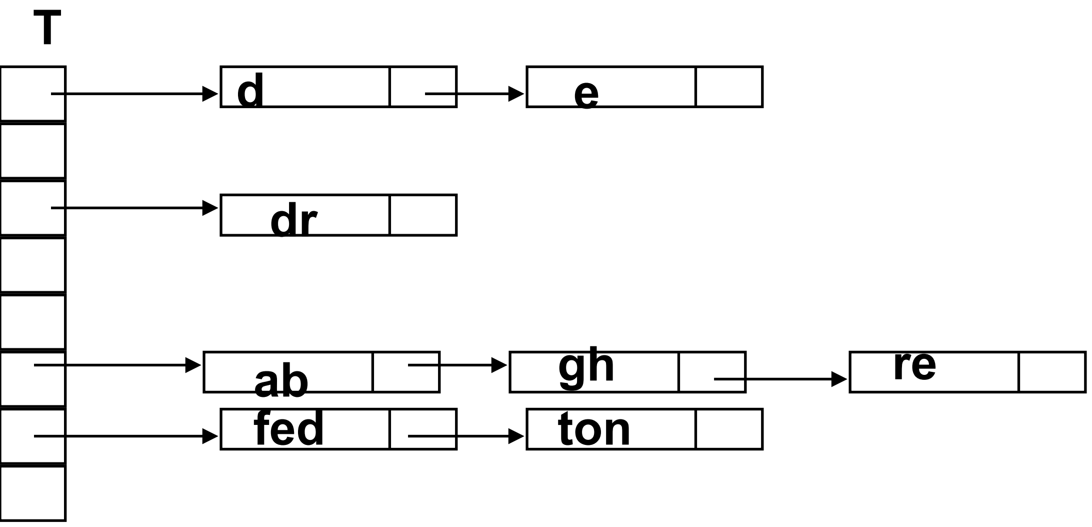
Coalesced chaining

- Seems crazy to use memory for links and keys – maybe just make a bigger hash table to begin with and reduce frequency of collisions!
- But where do we store colliding keys?
- In the table itself!
- Use the empty cells of the hash table to store the elements in the chain
- When a collision is encountered during an insertion, find the next open slot in the table and place the key there, linking it into the chain
  - This slot might be required for a subsequent insertion
  - Just treat this as another collision and perform another sequential search for an open slot

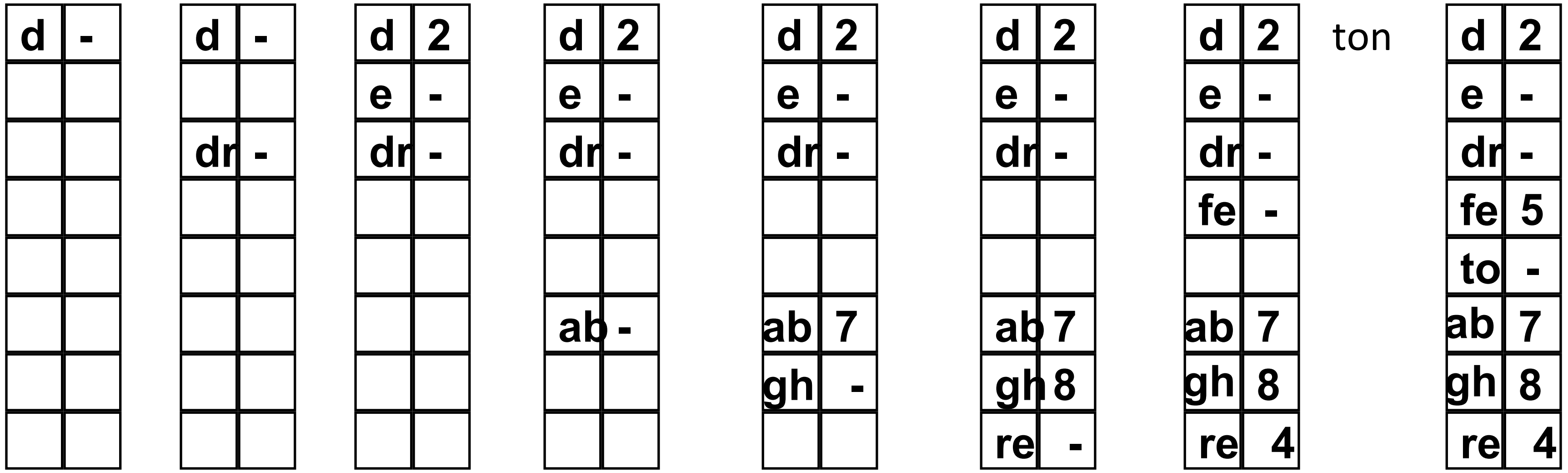
Coalesced Chaining

- Lookup operation is implemented the same as separate hashing, but the implicit chains can contain elements with several different hash values
  - But all elements with a given hash value are in the same chain
  - If an empty slot is encountered, search fails

Example of coalesced chaining



Insertion order:
d
dr
e
ab
gh
re
fed
ton



Open addressing strategies

- Generalization of coalesced chaining – eliminate links!
- For each key, its chain is stored in the hash table in a (key dependent) sequence of positions
  - probe sequence - sequence of table positions belonging to a chain
  - H(k,p) denotes the p'th position tried for key k, where p = 0, 1, ...
- To search for a key, look at successive locations in probe sequence until either the key is found or an empty table position is encountered
  - if key is to be inserted, it can be at that open position

Sequential or linear probing

- H(k,p) = h(k) + pc
- If h(k) = i, then the probe sequence is i, i+c, i+2c, ...
- c has to be relatively prime to m to ensure all locations are on the sequence; let's assume c=1
- In this case, if last position in table is m-1 and probe sequence reaches m-1, next location probed is 0 (wrap around)

Key	Probes to find
a	1
a'	2
c	1
b	3
c'	3
a"	6
g	1
i	1

Insertion sequence: a c g a' b i c' a''

Insertion sequence: a c g a' b i c' a''

Key	Probes to find
a	1
c	1
g	1

Key	Probes to find
a	1
a'	2
c	1
g	1

Key	Probes to find
a	1
a'	2
c	1
b	3
g	1

Insertion sequence: a c g a’ b i c’ a’’

Key	Probes to find
a	1
a'	2
c	1
b	3
g	1

Key	Probes to find
a	1
a'	2
c	1
b	3
g	1
i	1

Key	Probes to find
a	1
a'	2
c	1
b	3
c'	3
g	1
i	1

Key	Probes to find
a	1
a'	2
c	1
b	3
c'	3
g	1
i	1

Key	Probes to find
a	1
a'	2
c	1
b	3
c'	3
a’ ’	6
g	1
i	1

Linear probing

- Easy to implement
- Good behavior when table is not full
- Primary clustering - once a block of a few contiguous locations is formed, it becomes a "target" for subsequent collisions
  - collision makes the cluster become larger
  - the larger the cluster, the bigger a target it is for more collisions
- Simple solution of using the probe sequence i + kj will not work - clusters will just become sequences of the form i, i +kmod m, i+2k mod m, ...

Open addressing with double hashing

- Clusters can be broken up if the probe sequence for a key is chosen independent of its primary position
- Use two hash functions:

  1) h determines the first location in the probe sequence

  2) h' will be used to determine the remainder of the sequence; h' will depend on k.
- H(k,0) = h(k)
  - H(k, p+1) = (H(k,p) + h'(k)) mod m
- In linear probing, h'(k) = 1 for all k

Open addressing with double hashing

- Probe sequence should visit all of the entries in the table
- So, h'(k) must be greater than 0 and relatively prime to m for any k

  - h'(k) and m should have no common divisor

  - if d were a common divisor then

  (m/d * h'(k)) mod m = (m *h'(k)/d)) mod m = 0

    - so m/d'th probe in the sequence would be the same as the first

- Simplest solution is to choose m to be prime

Example:

m = 32

h'(k) = 6

h(k) = 4

probe sequence is:

4, 10, 16, 22, 28, 2, 8, 14, 20, 26, 32, 6, 12, 18, 24, 30, 4, ...

common divisor is 2

32/2 = 16 unique table addresses generated.

Exponential double hashing

- H(k,i) = h(k) + (a**i)h'(k) mod m
  - a is a constant, m is prime, a is a "primitive root" of k
  - If h(k) = k mod m and h'(k) = k mod m-2, then probe sequences will touch all of the entries in the table.
  - The sequences will be more random than the simple double hashing method.
- Practically, this hashing method works much better when the keys deviate from being uniformly distributed (like variable names in a program).

How caches affect hashing

- Modern computers have several levels of memory – L1 cache, L2 cache, memory, disk.
- Performance of any program depends on how often its memory references hit the cache
  - Otherwise you bring some block of memory down the memory hierarchy and push some block up. Takes time.
- Which hashing method would you expect to generate the most cache hits?
  - Intuitively, linear hashing since it probes sequential locations in memory!

How caches affect hashing

- But it DEPENDS on
  - The load factor of the table (percentage of table locations filled), and
  - How much the keys inserted in the table deviate from being uniformly distributed in the key set.
- In fact, if the keys were uniformly distributed (which they never are) then linear hashing is best at almost all load factors.
- But as the key set deviates from uniformity and load factors get larger than 20-30%, double hashing schemes work better, exponential the best.
- Heileman, Gregory L., and Wenbin Luo. "How Caching Affects Hashing." *ALENEX/ANALCO*. 2005.

Test data for hashing

Name	Date of Death	h(K)	h'(K)
J. Adams	7/4/1826	4	7
S. Adams	10/2/03	2	10
J. Bartlett	5/19/95	19	5
C. Braxton	10/10/97	10	10
C. Carro l	11/14/32	14	11
S. Chase	6/19/11	19	6
A. Clark	9/15/94	15	9
G. Clymer	1/23/13	23	1
W. Elery	2/15/20	15	2
W. Floyd	8/4/21	4	8
B. Franklin	4/17/90	17	4
E. Gerry	11/23/14	23	11
B. Gwinnet	5/19/77	19	5
L. Hall	10/19/90	19	10
J. Hancock	10/8/93	8	10
B. Harrison	4/24/91	24	4
J. Hart	5/11/79	11	5

Example - Open addressing with double hashing

1		
2	S. Adams	1
3	E. Gerry	2
4	J. Adams	1
5		
6		
7		
8	J. Hanco	1
9		
10	C. Braxto	1
11	J. Hart	1
12	W. Floyd	2
13		
14	C. Carrol	1
15	A. Clark	1
16		
17	R. Ellery	2
18		
19	J. Bartlet	1
20		
21	B. Frankl	2
22		
23	G. Clymer	1
24	B. Gwinn	2
25	S. Chase	2
26		
27		
28	B. Harris	2
29	L. Hall	2
30		
31		

Brief review

- What is a good hash function?
  - Should spread keys around the table “randomly”
  - Should be efficient to compute (constant time)
- Collisions are inevitable. How do we handle them?
  - Separate chaining – linked lists pointed to by hash table elements. Wasteful – use space for links to build a bigger table to begin with.
  - Implicit collision chains stored in the table
    - Linear probing – same relative collision sequence for every key
    - Double hashing – different collision sequences for different keys

Next

- How can we make lookup more efficient?
  - By ensuring that the keys encountered on a collision chain are weakly ordered – then if we encounter a key on the collision chain that is "larger" than the query, we can terminate search with failure.
    - This will cause unsuccessful lookups to fail early, but will not improve the search cost, on average, for successful lookups
  - Since collision chains overlap, maybe we can move keys around on their collision chains to shorten them.
    - This will reduce the search cost for successful lookups as well as reduce the search time for unsuccessful ones (since the individual collision chains are mostly shorter)

Ordered hashing

- In the example, names were inserted in alphabetical order
- So, all of the chains constructed had their data in alphabetical order
- This would allow us to terminate an unsuccessful search whenever we encounter an element on a chain that follows the search key lexicographically.

Ordered hashing

- In separate chaining we can easily maintain the individual linked lists in alphabetic order
- Can we do this with open addressing?
  - No, but we can come close
- Property to be maintained: during the probe sequence for k, keys encountered before reaching k are smaller than k

Ordered hashing

- Trick: As the probe sequence for key k is followed during insertion, if a key k' is encountered such that k < k', replace k' by k and proceed to insert k' as dictated by its probe sequence.
- Why does this create a hash table with the desired property? Proof by induction:

  1) When the table is empty, the property holds trivially.

  2) Table can be extended in one of two ways:

  a) putting a key, which is greater than any of its predecessors (i.e., no interchanges) in a previously empty slot

  b) replace a key in the table by a smaller key - but then any lookup that went through the k’ will go through k also since k < k’.

Ordered hashing

- "Obviously", the final state of the hash table for a given set of keys will be the same, regardless of the order in which the keys are inserted into the table!

h(k) = index of first letter of k in the alphabet
h'(k) = index of second letter of k
Using only first 13 letters of alphabet

ago	1

Insert:
ago
def
ace
dag
abe

Example

0						
	ago	1	ace	1	ace	1
2						
4	def	1	def	1	dag	1
6						
8			ago	2	ago	2
					def	2
10						
12						

Insert:
ago
def
ace
dag
abe

Example

0						
	ace	1	abe	1	abe	1
2						
4	dag	1	ace	2	ace	2
			dag	2	ade	2
6					dag	3
8	ago	2	ago	2	ago	2
	def	2	def	2	def	2
10						
12						

Insert:
ago
def
ace
dag
abe
ade

Ordered hashing

- If we insert a set of keys into a hash table using ordered hashing, then the collision chains satisfy the following property:
  - Any keys encountered on the collision chain for a key, k, will precede k lexicographically.
- The keys on the collision chain will NOT necessarily be sorted (in fact they typically will not be).

	ba							

ab	ba							

ab	ba		bb					

ab	ba		bb	ad				

ab	ba		bb	ad	da			

ab	ba		bb	ad	bd	da		

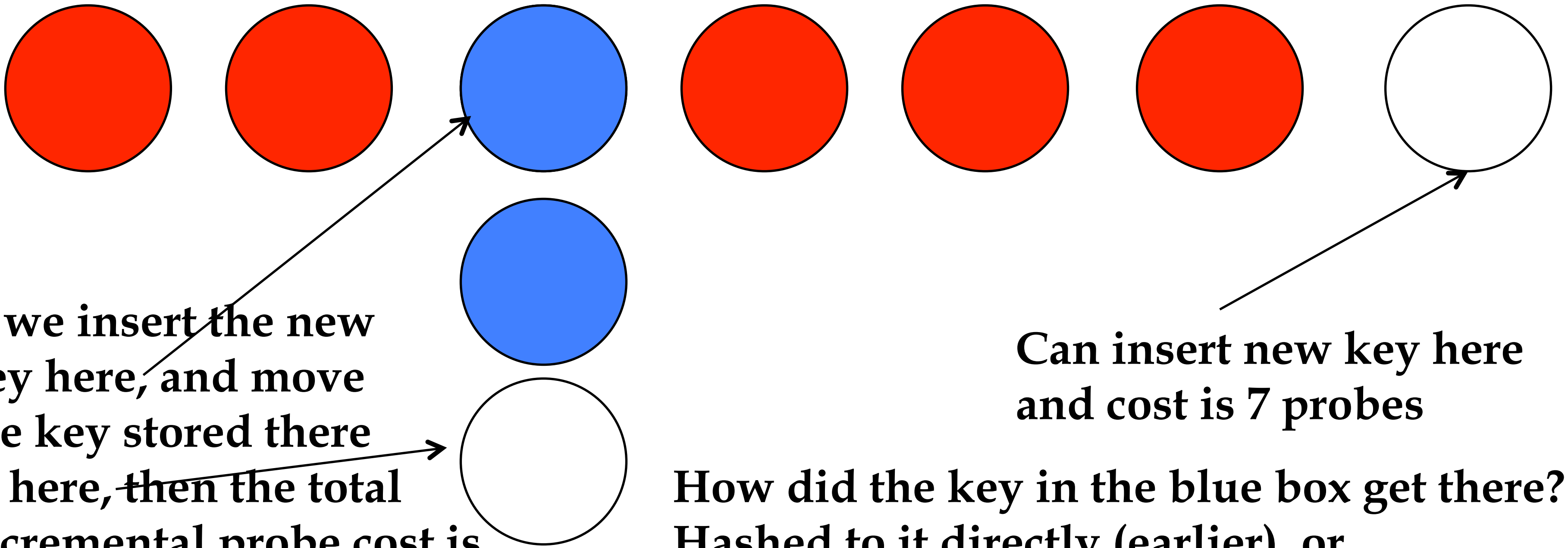
Brent's method

- During insertion process, rearrange keys so that collision lists are shorter, and subsequent searches are faster.
- Good when searches are more common than insertions, because insertions are expensive
- Called self-organizing double hashing
- Let p_0, p_1, ..., p_t be the sequence of table addresses generated by a double hashing function for some key k before encountering an empty table position, p_t.
- If we insert k at p_t, then any subsequent search for k will require t+1 probes

Brent's method

- Now, we could replace any of the keys in locations p_0, p_1, ..., p_{t-1} with k and shorten the search (in terms of number of probes) for k. Suppose we place k in p_r
- But then we have to move the displaced key, k', in p_r somewhere
  - one place to put it is at the end of its chain
  - if the savings in placing k at p_r - (t-r) - is greater than the extra work in subsequent searches for the key stored originally at p_r(the length of the chain starting at p_r), then the switch will, overall, save search time.

Simple example



If we insert the new key here, and move the key stored there to here, then the total incremental probe cost is 5

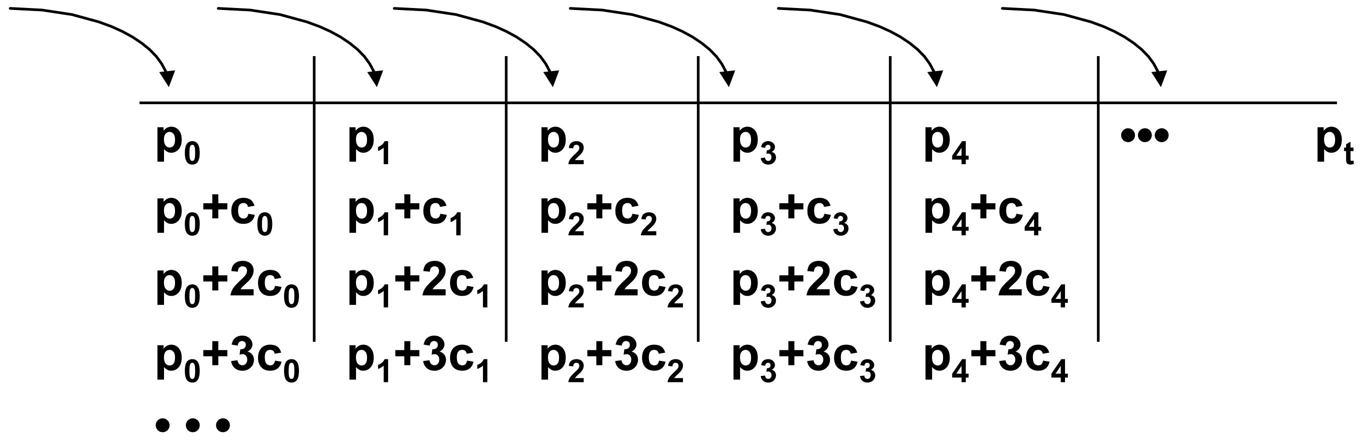
Can insert new key here and cost is 7 probes

How did the key in the blue box get there?
Hashed to it directly (earlier), or
the blue box is on its collision chain

Brent's method

- We will visualize the hash table as being a 2-d grid
- The i'th column will be the collision chain starting at p_i
  - Note that this does **not** mean that the key stored at p_i is the first element in its chain - but the algorithm works in any case, because the **extra** number of probes needed to find the key at p_i is the distance between p_i and the end of the chain passing through p_i.
- The j'th row will contain the address of the j'th element on each collision chain (or will be empty if some collision chain has fewer than j elements)

Brent's method



- The columns are really of different lengths, ending at the first empty location in each collision chain.
- Try to insert k as close to p_0 as possible
- Let L_j be the length of the chain in column j.
- Find the chain that maximizes (t-j) - L_j (savings-additional cost)

Brent's method

p_0	p_1	p_2	p_3	p_4	p_5
1 (5,1)	3 (4,1)	6 (3,1)	10 (2,1)	15 (1,1)	
2 (5,2)	5 (4,2)	9 (3,2)	14 (2,2)		
4 (5,3)	8 (4,3)	13 (3,3)			
7 (5,4)	12 (4,4)				
11 (5,5)					

Gonnet-Munro

- Brent's algorithm only moves keys on the probe sequence of the insertion key to the end of their probe sequences
- Gonnet-Munro will consider moving keys that are on these probe sequences, and on those keys' probe sequences, also.
  - requires some systematic way of searching through these collision chains

Gonnet Munro example

p_0=Tim	p_1=Alan	p_2=Jay	p_3=Katy	p_4=-
Joan	Ruth	-	Kim	-
Ron	-	-	Alex	-
Rita	-	-	Bob	-
-	-	-	-	-

Joan	Rita	Kim	Ruth	Ron	Alex	Bob	Fay	Jill
-	Jay	-	Tim	Kathy	-	-	-	-
-	Kim	-	-	Alan	-	-	-	-
-	Fay	-	-	Jay	-	-	-	-
-	-	-	-	Jill	-	-	-	-
-	-	-	-	-	-	-	-	-

Gonnet Munro

- Suppose we attempt to insert "Rudy" which takes us along the probe sequence p_0, ..., p_4
- Brent's algorithm would replace "Alan" with "Rudy" and would move "Alan" to the end of its probe sequence
  - this saves 3 probes for Rudy and gives up 2 probes for Alan, for a savings of 1

Gonnet Munro

- But notice that we could
  - replace Tim with Rudy - saving 4 probes
  - replace Joan with Tim - costing 1
  - move Joan one position down her probe sequence -costing 1
  - for a total saving of 2, better than Brent

Gonnet Munro search tree

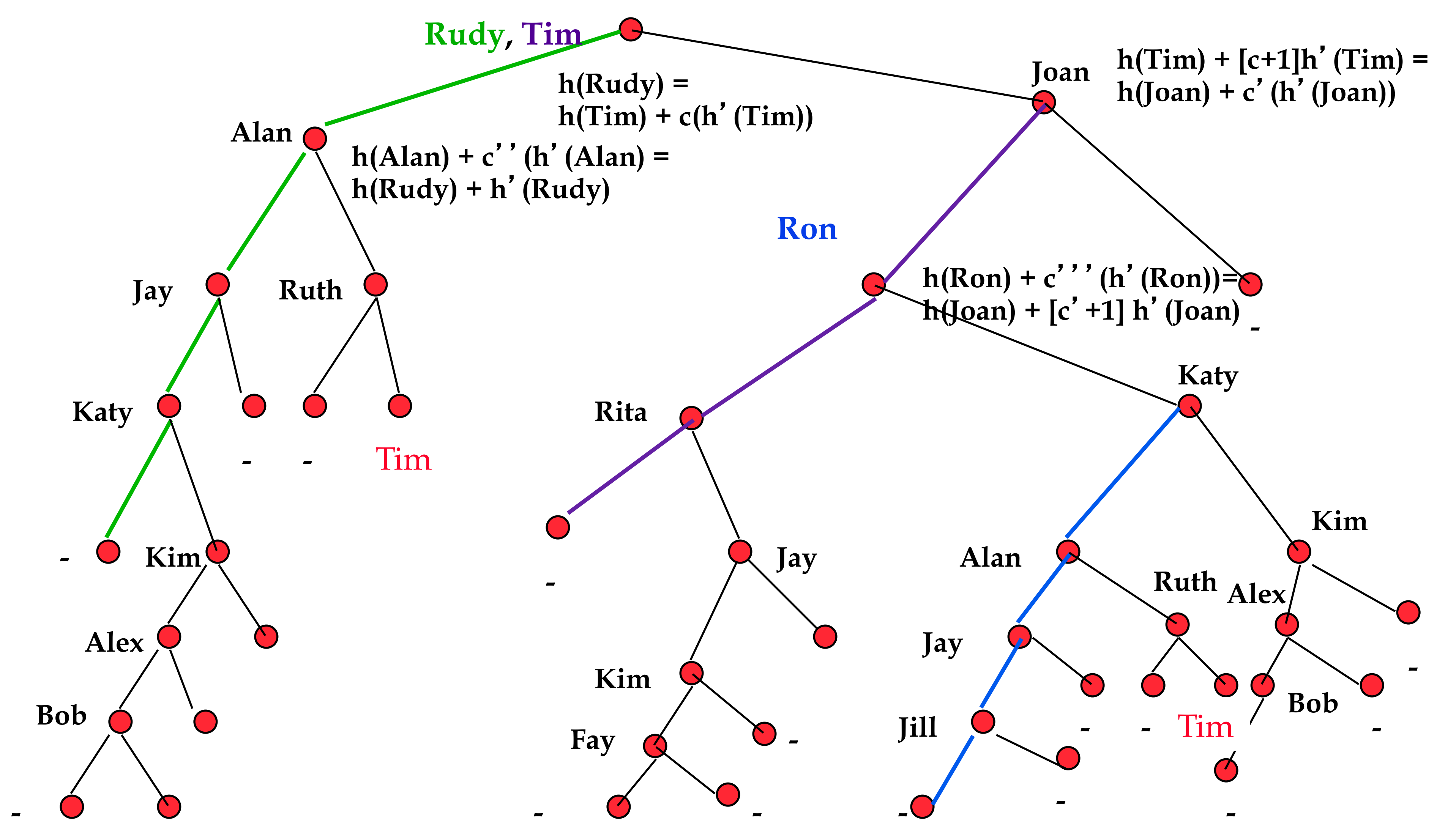
- Search is conducted using a binary tree constructed from the probe sequences
  - left most branch of the tree will be the original probe sequence p_0, ..., p_4
  - right branches will represent probe sequences starting from these positions
  - ...
- Each node in the binary tree will have fields for
  - a Key value
  - the Location (LOC)in the hash table at which this key value is stored
  - The secondary hash function, INC, for Key.

Gonnet Munro search tree

- The root of the tree has
  - LOC = h(k), where k is the key to be inserted
  - INC = h' (k)
  - Key = original key at position LOC

- Given any node, S, in the tree
  - its left son, L_S, corresponds to $LOC_S + INC_S$
    - $LOC_{LS} = LOC_S + INC_S$
    - $INC_{LS} = INC_S$
    - Key_{LS} = key stored at $H[LOC_S + INC_S]$
  - its right son, R_S, is the next location encountered in the probe sequence of Key_S
    - $LOC_{RS} = LOC_S + h'(Key_S)$
    - $INC_{RS} = h'(Key_S)$ - left chain from R_S will be probe sequence for Key_S

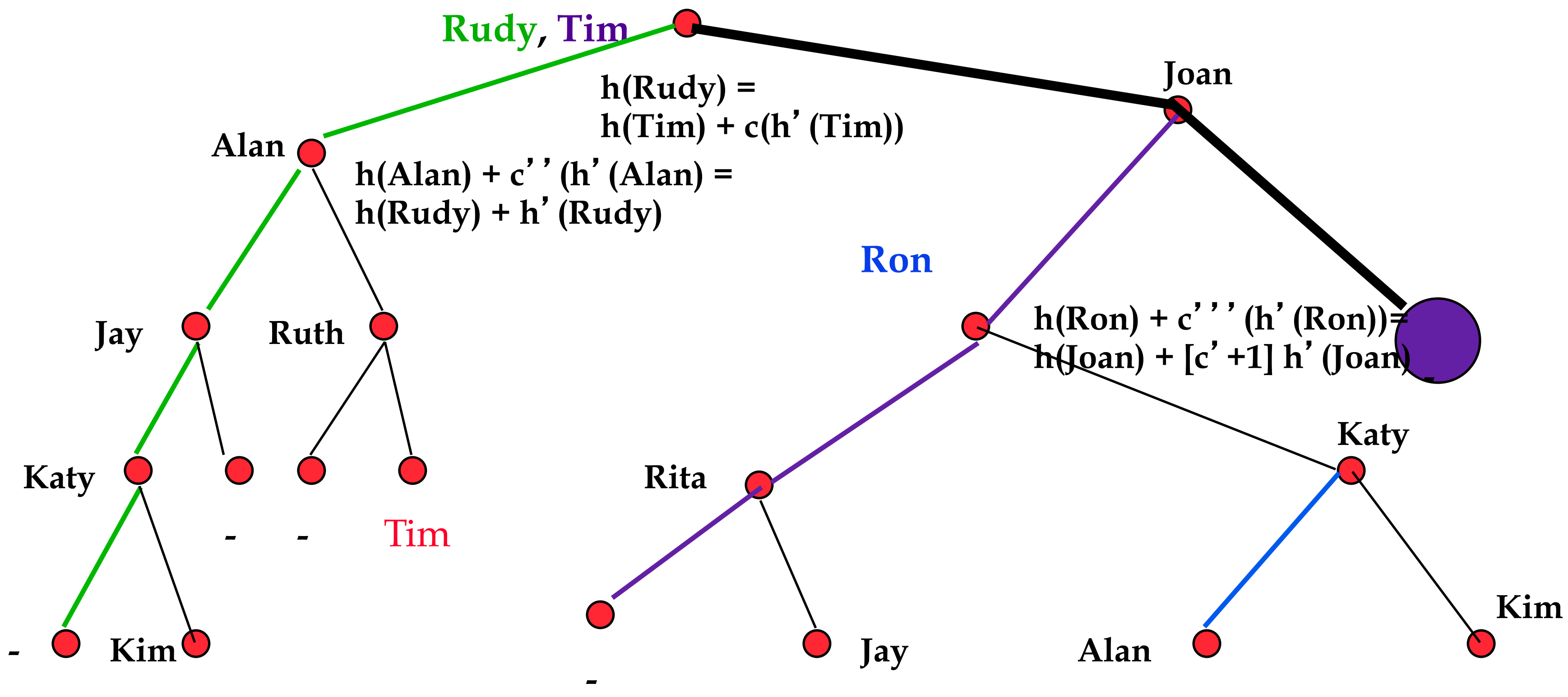
The Tree


Rudy, Tim
Joan
h(Tim) + [c+1]h' (Tim) =
h(Joan) + c' (h' (Joan))
h(Rudy) =
h(Tim) + c(h' (Tim))
Alan
h(Alan) + c'' (h' (Alan) =
h(Rudy) + h' (Rudy)
Ron
Jay
Ruth
h(Ron) + c''' (h' (Ron))=
h(Joan) + [c' +1] h' (Joan)
Katy
Rita
Katy
Tim
Kim
Kim
Jay
Alan
Ruth
Alex
Alex
Jay
Kim
Bob
Jill
Tim
Bob
Fay

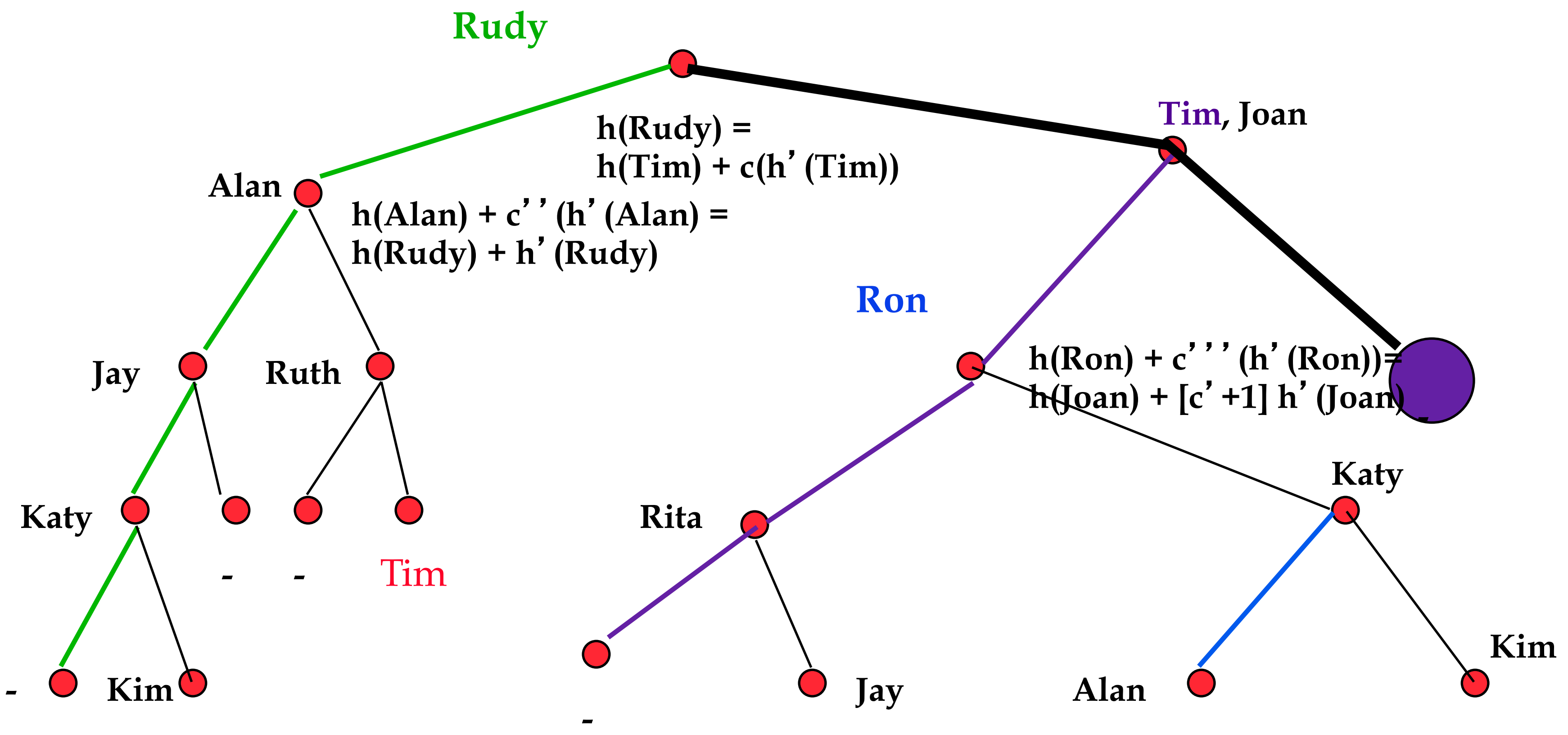
The Algorithm

- Generate probe tree from top to bottom and left to right until encountering the first null position in some hash chain.
- Keys will be moved along the path from the root to this empty position using the following recursive step:
  - Algorithm is at node N holding key K_N trying to insert key K_{NEW}.
  - Move along any left links in path to NULL node until encountering either
    - NULL position - insert K_{NEW} and halt
    - Right pointer to N' - insert K_{NEW} at father of N' and insert original key at father of N' in tree rooted at N'.

Example


Rudy, Tim
Joan
Alan
h(Rudy) =
h(Tim) + c(h' (Tim))
h(Alan) + c' ' (h' (Alan) =
h(Rudy) + h' (Rudy)
Ron
Jay
Ruth
h(Ron) + c' ' ' (h' (Ron))=
h(Joan) + [c' +1] h' (Joan)
Katy
Katy
Rita
-
-
Tim
Kim
-
Kim
Jay
Alan
-

Example

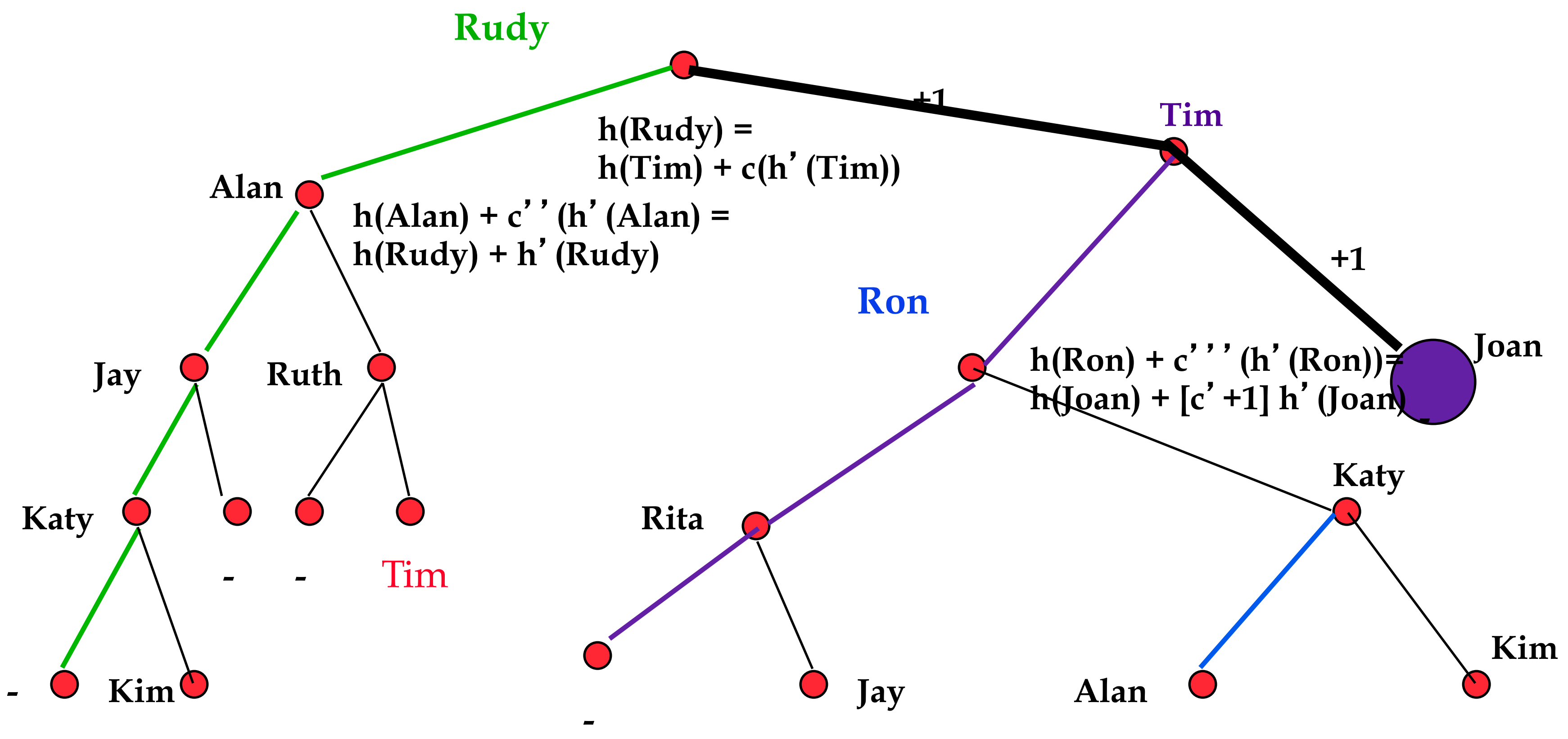


Algorithm is at node N holding key K_N trying to insert key K_{NEW}.
Move along any left links in path to NULL node until encountering either

NULL position - insert K_{NEW} and halt

Right pointer to N' - insert K_{NEW} at father of N' and insert original key at father of N' in tree rooted at N'.

Example

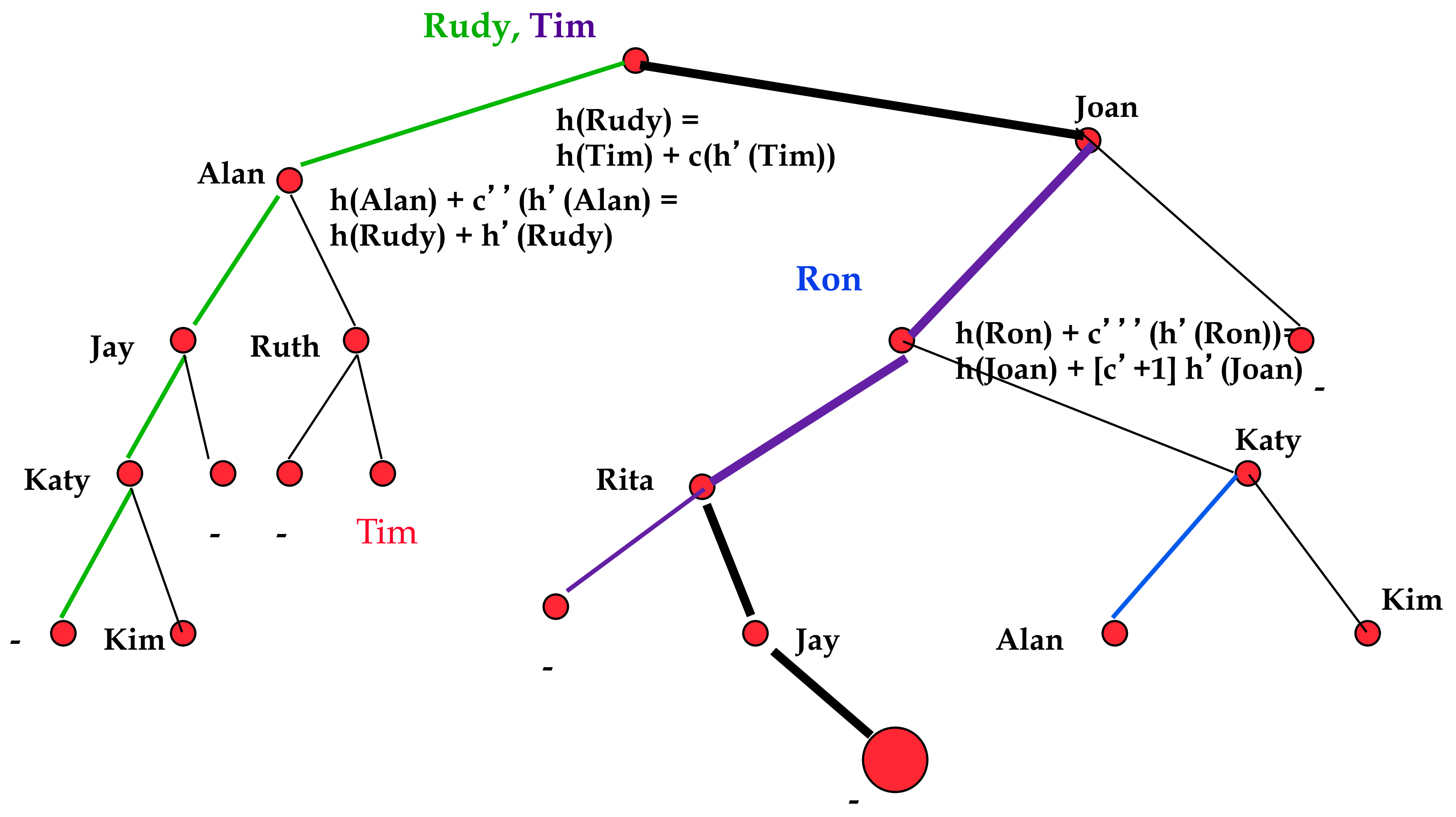


Algorithm is at node N holding key K_N trying to insert key K_{NEW}.
Move along any left links in path to NULL node until encountering either

NULL position - insert K_{NEW} and halt

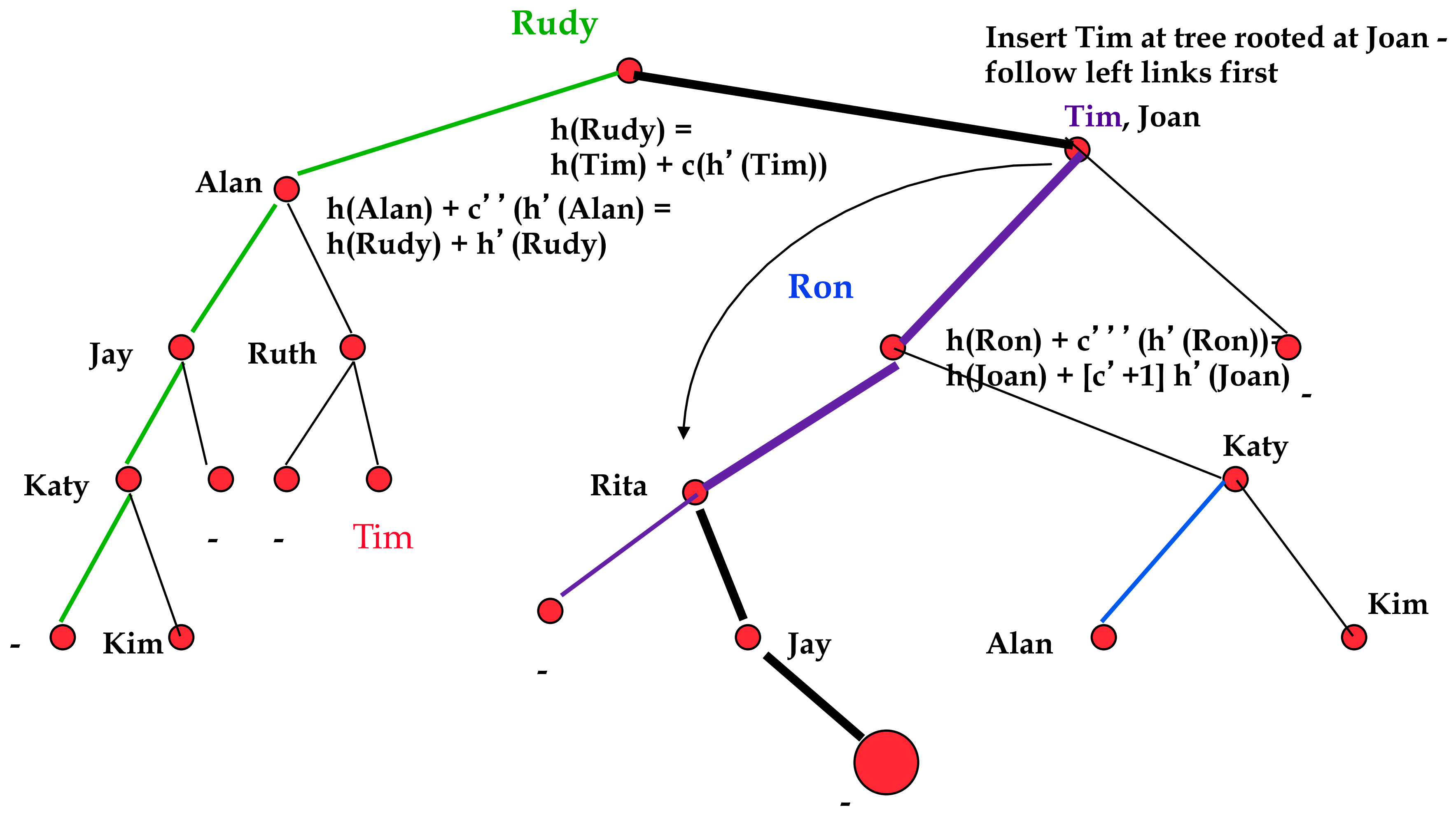
Right pointer to N' - insert K_{NEW} at father of N' and insert original key at father of N' in tree rooted at N'.

Example

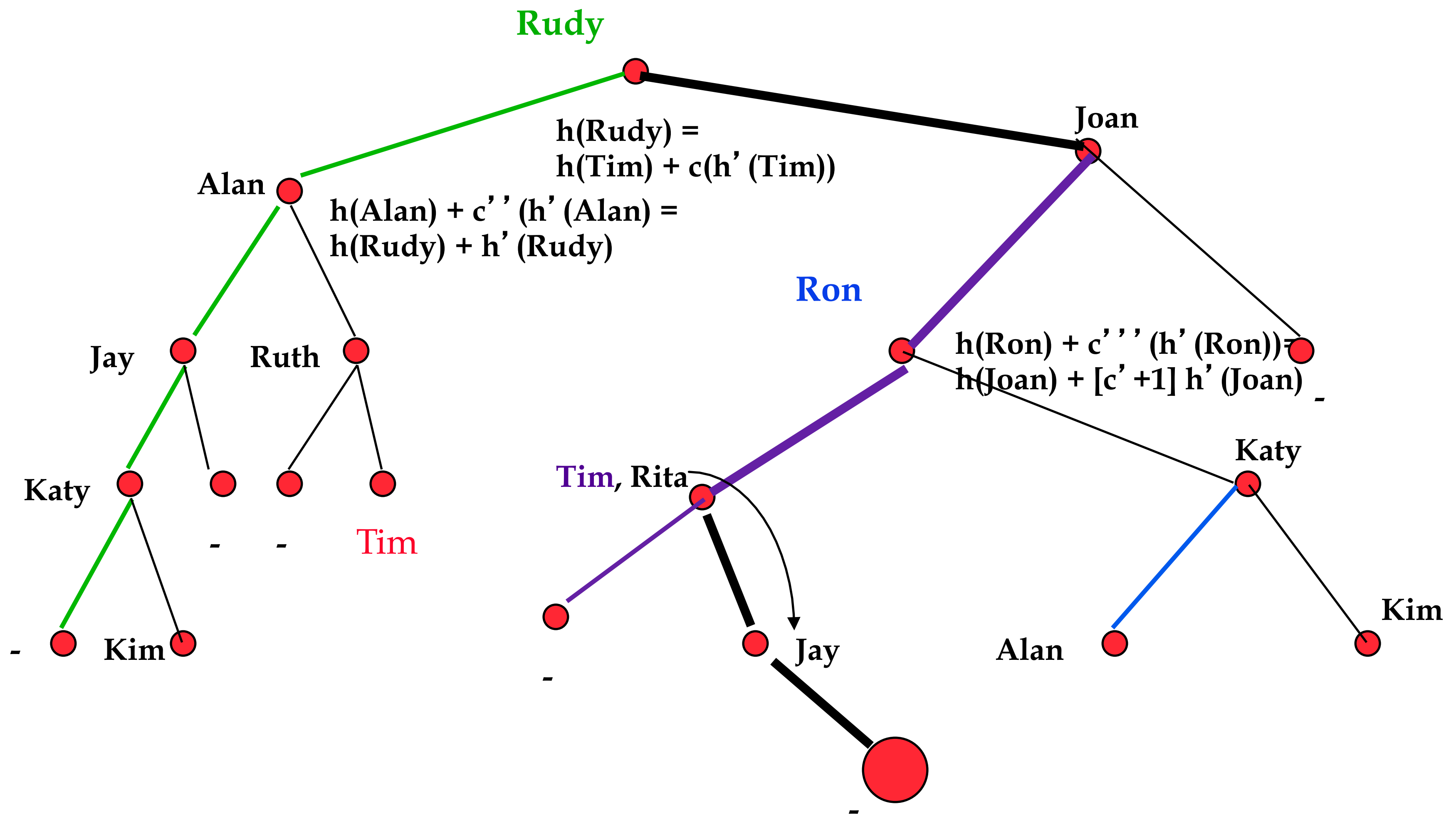


Rudy, Tim
Joan
Alan
h(Rudy) =
h(Tim) + c(h' (Tim))
h(Alan) + c' ' (h' (Alan) =
h(Rudy) + h' (Rudy)
Ron
Jay
Ruth
h(Ron) + c' ' ' (h' (Ron))=
h(Joan) + [c' +1] h' (Joan)
Katy
Katy
Rita
Tim
Kim
Jay
Alan
Kim

Example

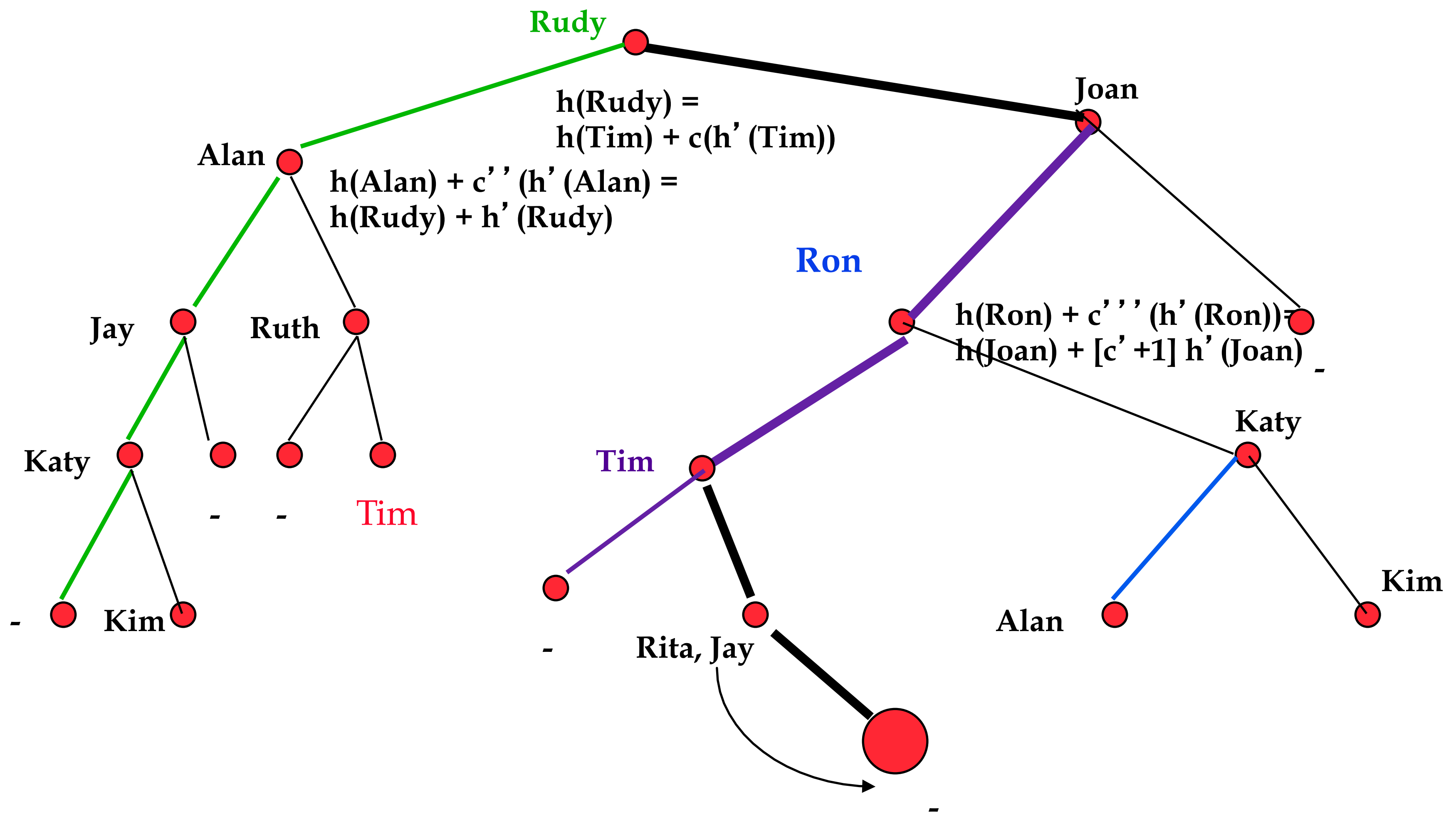

Rudy
Insert Tim at tree rooted at Joan - follow left links first
Tim, Joan
h(Rudy) =
h(Tim) + c(h' (Tim))
Alan
h(Alan) + c' ' (h' (Alan) =
h(Rudy) + h' (Rudy)
Ron
h(Ron) + c' ' ' (h' (Ron))=
h(Joan) + [c' +1] h' (Joan)
Jay
Ruth
Katy
Rita
Katy
Tim
Kim
Jay
Alan
Kim
-
-
-
-
-
-
-

Example

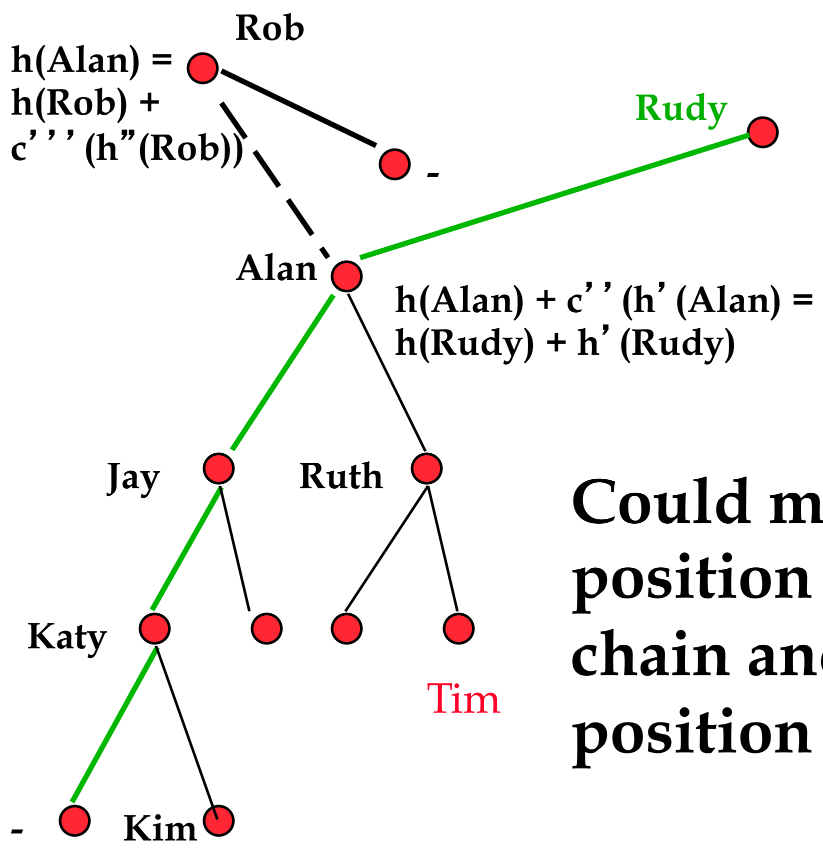


Rudy
h(Rudy) =
h(Tim) + c(h' (Tim))
Joan
Alan
h(Alan) + c'' (h' (Alan) =
h(Rudy) + h' (Rudy)
Ron
Jay
Ruth
h(Ron) + c''' (h' (Ron))=
h(Joan) + [c' +1] h' (Joan)
-
Katy
Katy
Tim, Rita
-
-
Tim
-
Kim
-
Jay
Alan
Kim
-

Example


Rudy
Joan
Alan
h(Rudy) =
h(Tim) + c(h' (Tim))
h(Alan) + c' ' (h' (Alan) =
h(Rudy) + h' (Rudy)
Ron
Jay
Ruth
h(Ron) + c' ' ' (h' (Ron))=
h(Joan) + [c' +1] h' (Joan)
-
Katy
Katy
Tim
-
-
Tim
-
Kim
Kim
Alan
-
Rita, Jay
-

Gonnet Munro not perfect



Rob
h(Alan) = h(Rob) + c''' (h''(Rob))
Rudy
-
Alan
h(Alan) + c'' (h' (Alan) = h(Rudy) + h' (Rudy)
Jay
Ruth
Katy
Tim
-
Kim

Could move Alan back one position on its collision chain and move Rob one position on its chain.

Deletion from hash tables

- Separate chaining - easy, just remove the entry from the linked list
- Open addressing - complicated since chains "overlap" - i.e., position of a key being deleted might be on the probe sequence of another key still in the table.
- Example - sequential probing
  - insert key k in position i
  - insert colliding key k' in position (i+1) mod m
  - delete key k, making table entry i empty
  - search for key k', but find empty slot at position i.

Deletion from hash tables

- Solution: Add a one bit deleted flag to each table entry.
  - search proceeds over any entry with the deleted bit set to 1
  - insertion occurs at the first position that is empty or has deletion bit set to 1.
- Problem: Searches become lengthy after a large number of insertions and deletions have occurred
- Deleted bit can also be used with other collision resolution strategies, but they also suffer from degraded performance over time.
- An expensive solution is to rehash the entire table from scratch

Extendible hashing

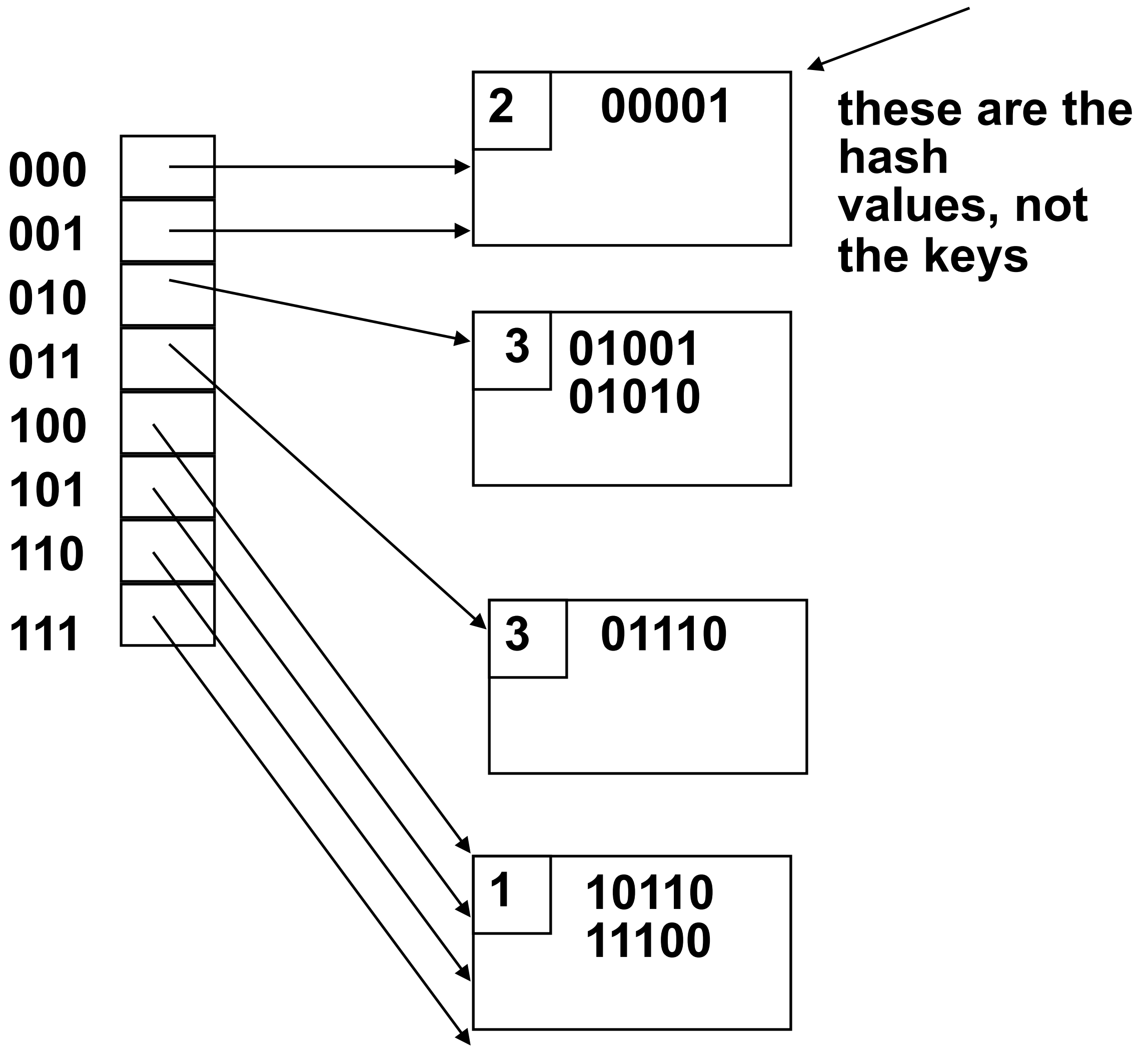
- Main disadvantages of open addressing:
  1) deletions cause performance to be degraded since table is cluttered with "deleted" entries.
  2) size of the table cannot be adjusted if the number of entries is larger than anticipated
- Only solution is to allocate a new table and rehash the contents of the old table
- Extendible hashing
  1) allows table to grow and shrink
  2) Approach will extend to spatial data

Extendible hashing

- Two level data structure: directory (memory) and a set of leaf pages (disk).
- Directory is a table of pointers to leaf pages
- Data records are stored in the leaf pages, which are of fixed size, b.
- Use a hash function that maps keys to bit strings of length L
  - for d <= L, let $H_d(k)$ be the first d bits of H(k). This is called the d-prefix of H(K).
- Example: L = 5, H(k) = 01010, then $H_3(k)$ = 010.

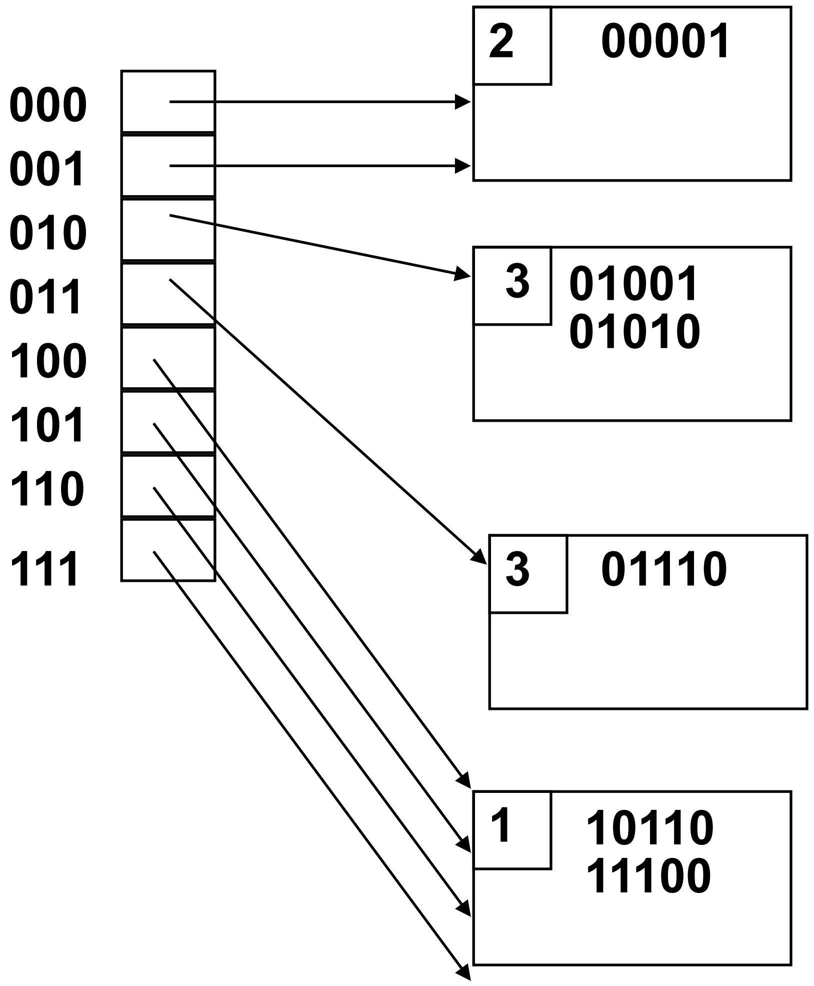
Extendible hashing

- A leaf page consists of all keys whose hash values have a particular prefix. The length of this prefix is called the depth of the page
- Maximum depth of any leaf page is called the depth of the table, D (=3 in the example)



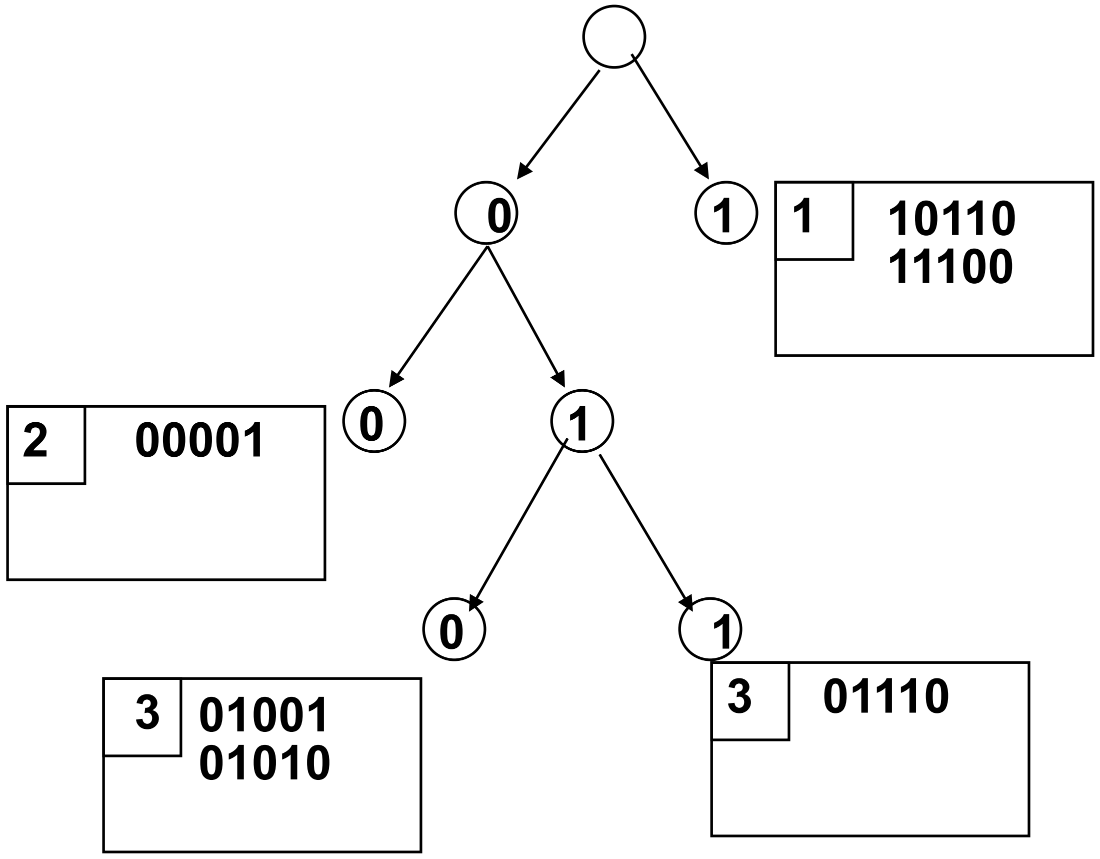
Extendible hashing

- Directory is a table, T, of length 2^D containing pointers to leaf pages.
- To locate page containing key k, compute $H_D(k)$ and follow the pointer in $T[H_D(k)]$.
- If the depth of a leaf page is less than the depth of the table, several pointers will point to it.
- specifically, a leaf page of depth d will be pointed to by 2^{D-d} consecutive entries of T.



Directory is a trie!

- Directory is the top level of a trie, discriminating the first D bits of the hash value of a key.
- Individual leaf pages can be open-addressed hash tables, binary search trees, tries - whatever you want



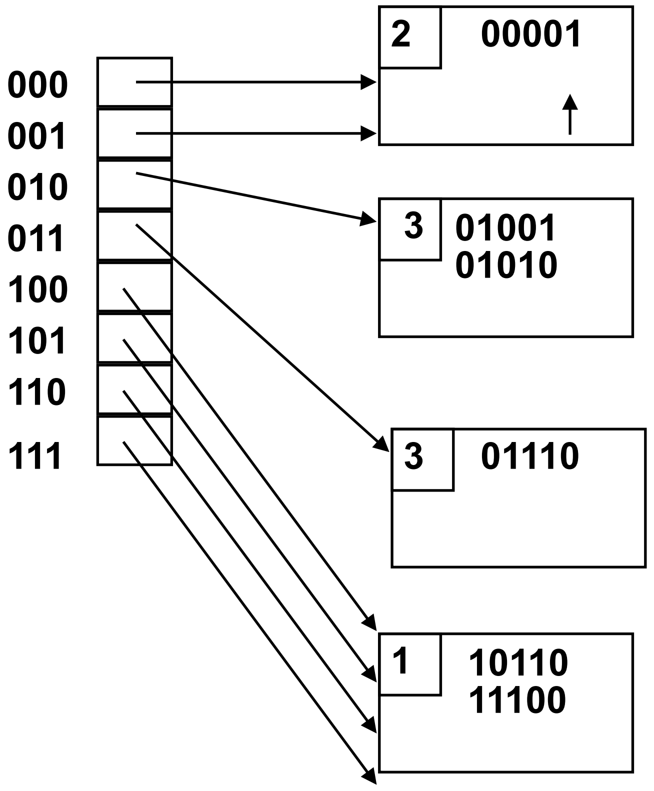
Inserting into a full leaf page - trie splitting

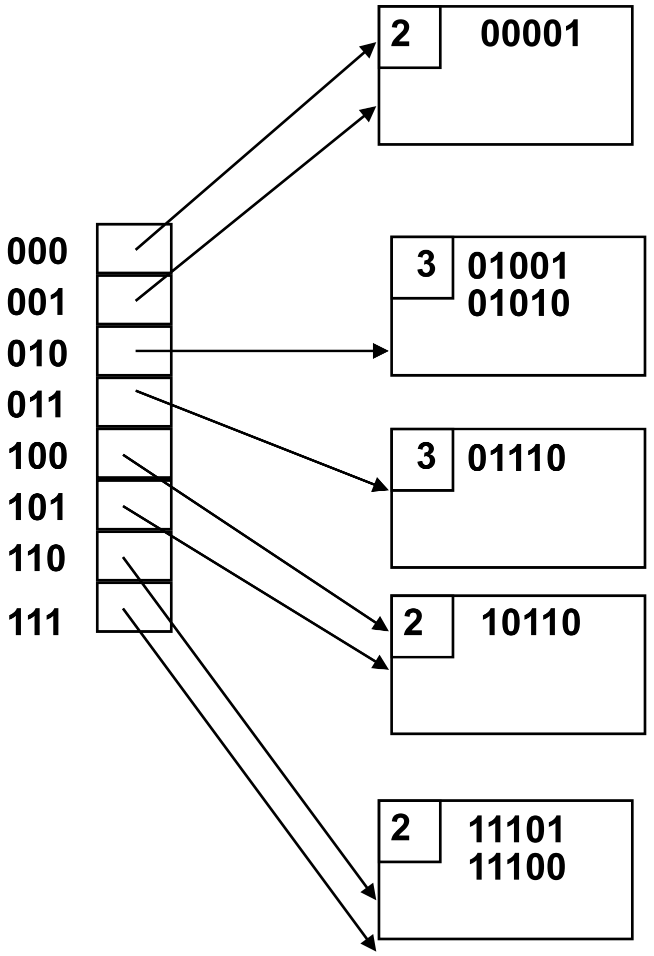
- If we insert a key into a non full page, no problem.
- But if we try to insert into a full page, we must split the page. Split is accomplished by :
    1) increasing the depth of the full page and creating a "buddy" page (extend the trie one level below the overflowed bucket)

    2) distribute the records to the two new pages appropriately (based on their d+1 prefixes)

    3) May have to iterate this process if all keys fall in one of the buddies

Insertion example





Example: Insert key with hash value 11101 into table

Insertion into extendible hash table

- What happens if we split a leaf of maximal depth?

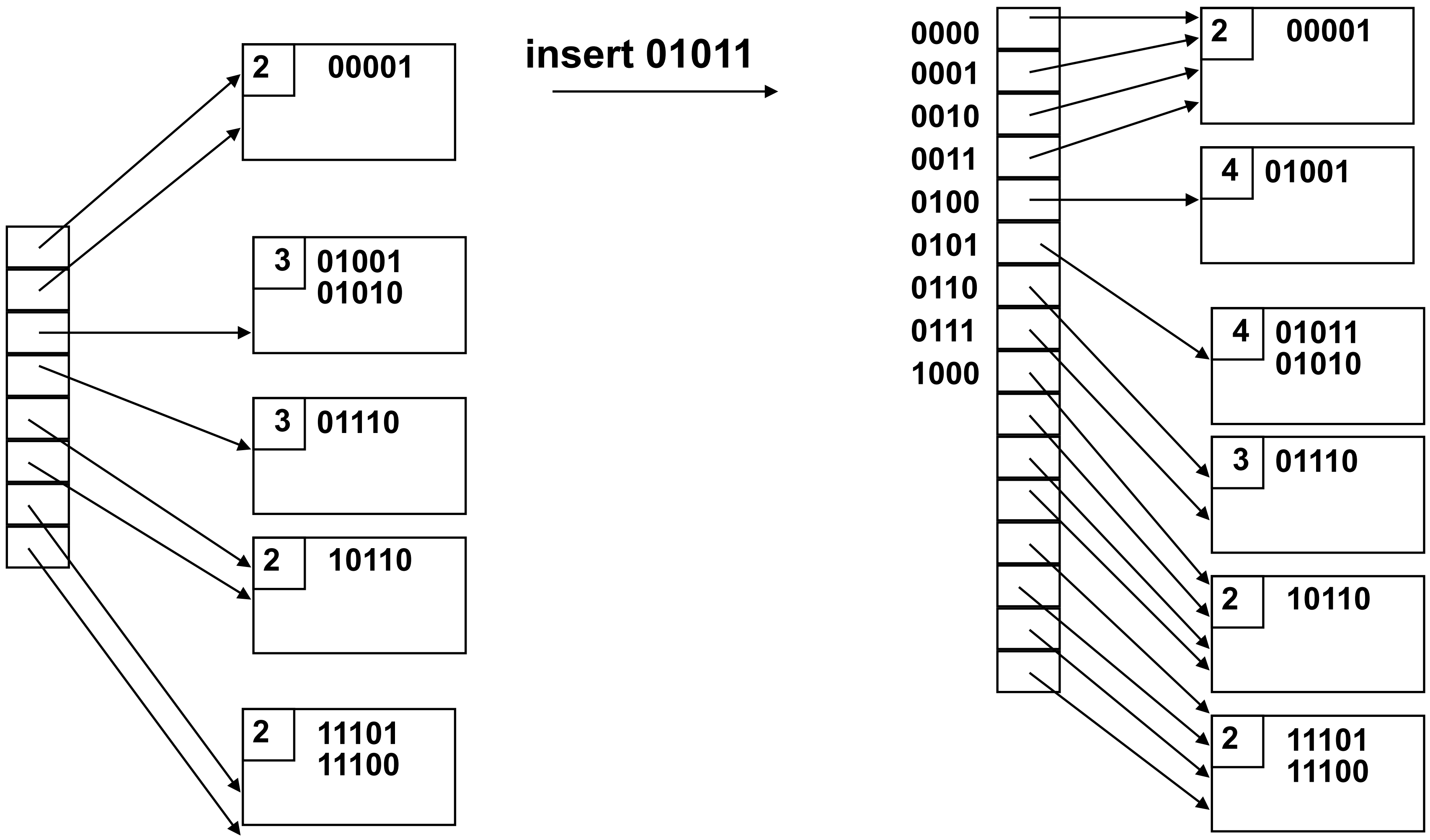
  a) changes the depth of the table itself

  b) must double the size of the directory

  c) generally, entries 2i and 2i+1 of the new directory point to the same leaf page as entry i of the old directory, unless entry i was the one being split

  d) although if the directory is represented as a trie this complexity disappears – all splits are treated the same

Example



2
00001
3
01001
01010
3
01110
2
10110
2
11101
11100
insert 01011
0000
0001
0010
0011
0100
0101
0110
0111
1000
2
00001
4
01001
4
01011
01010
3
01110
2
10110
2
11101
11100

Deletion and extendible hashing

- Extendible hashing can accommodate deletion
- Suppose entries 2i and 2i+1 point to distinct pages of maximal depth
- If deleting an entry from one of them causes their total size to be < b, then pages can be collapsed into a single page
- If they were the only leaf pages of maximal depth, then directory can be halved in size
- Practically, don't always want to collapse two half full pages into one full page - next operation may overflow that page.

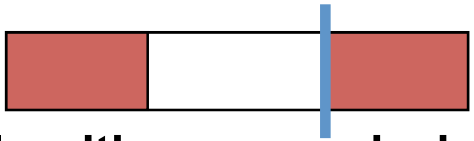
Linear hashing

- Drawback of extendible hashing is that table must double to grow - not a problem for small tables, but a disadvantage for large ones
- Linear hashing allows the table to grow one bucket at a time
  - table is extended when its load factor passes a critical value
  - buckets are split "in sequence" starting at 0 and ending at 2^n.
  - two hashing functions are always "active" - one for split buckets and one for unsplit
  - when the last bucket (the 2^nth one) is split, the table has been doubled in size, and a new cycle of splitting is begun.

Linear hashing

- Suppose that our hash table currently contains m buckets, with $2^n <= m <= 2^{n+1} -1$.
- Two hash functions are “active”
  - $h_n(k)$ = k mod 2^n which is used to access buckets m-2^n through 2^n - 1. These are the buckets which have not been split during this cycle of linear hashing
  - $h_{n+1}(k)$ = kmod^{2n+1} which is used to access buckets 0 through m - 2^n-1 and buckets 2^n through m-1. These are the buckets which have been split, and the buckets introduced as a result of splitting.

Initially, m = 2^n and we have only one hash function



as algorithm proceeds, buckets are split and we have two hash functions, one for the solid area and one for the “empty” area.

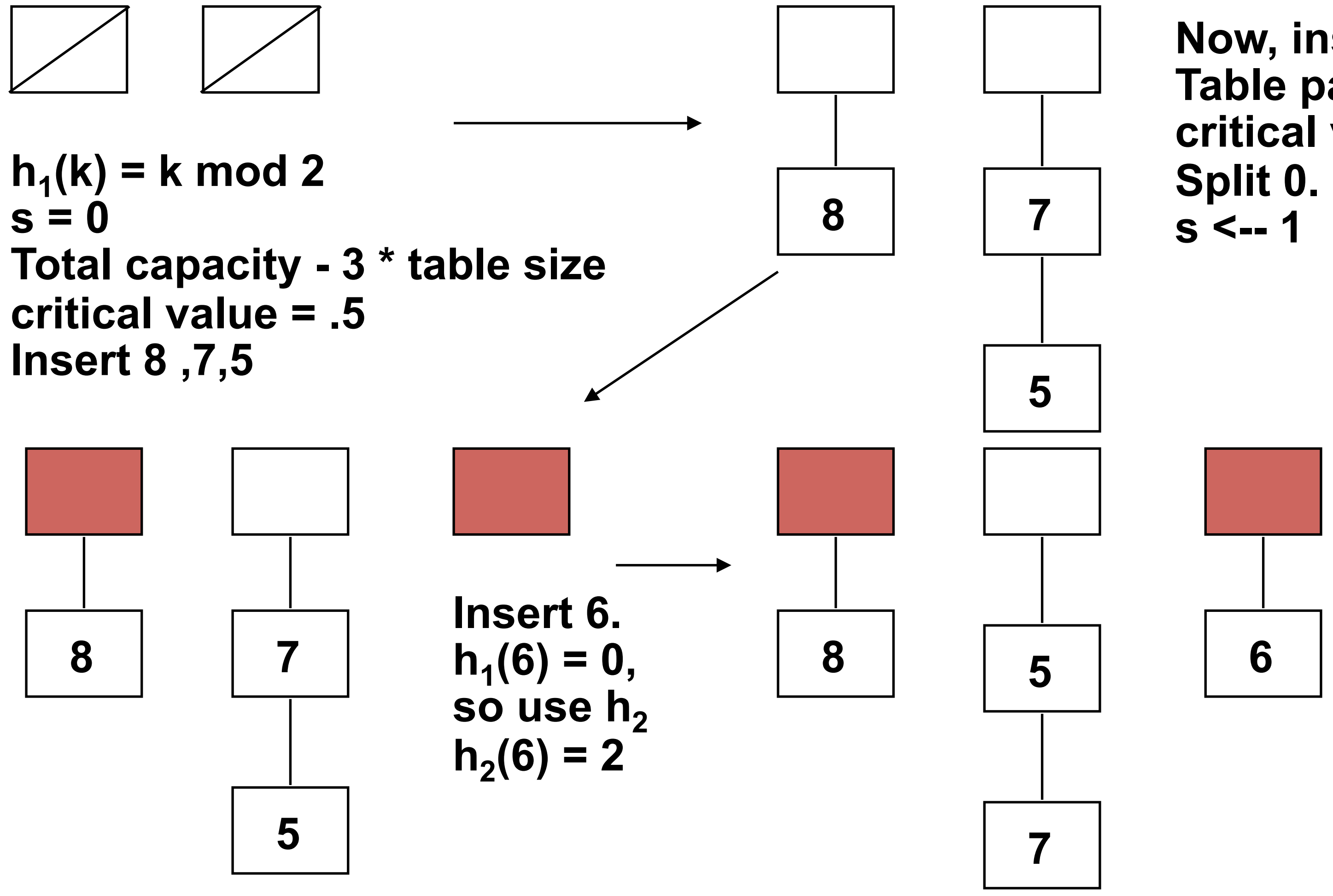
Linear hashing

- Each entry in the hash table stores the records that hash to it using a primary bucket, with some capacity, and a sequence of overflow buckets.
- The load factor for the table is the percentage of its storage being used for records.
- When we insert a record, we increase this factor. If it passes a critical value, then we split one of the buckets, b.
  - It's records are rehashed using h_{n+1}.
  - They are distributed into buckets b and b+ 2^n. This is because the hash function h_{n+1} looks at one more bit than h_n. If that bit is 0, then the record is kept in location b. If that bit is 1, then it will be moved to b + 2^n.

Linear hashing

- Algorithm maintains a pointer s, which points to the next bucket to be split.
- s starts at 0; when it reaches 2^n, we have split all of the buckets during the current cycle. At this point we increment n by 1, and reset s to 0.
- Notice that the bucket split is not, generally, the bucket into which the last record was inserted! But, as records are inserted, we will eventually split all buckets.
- The hash table does not have to occupy contiguous memory - we can insert one level of indirection with a directory, allowing the table to be kept in fragmented memory or on disk.

Example

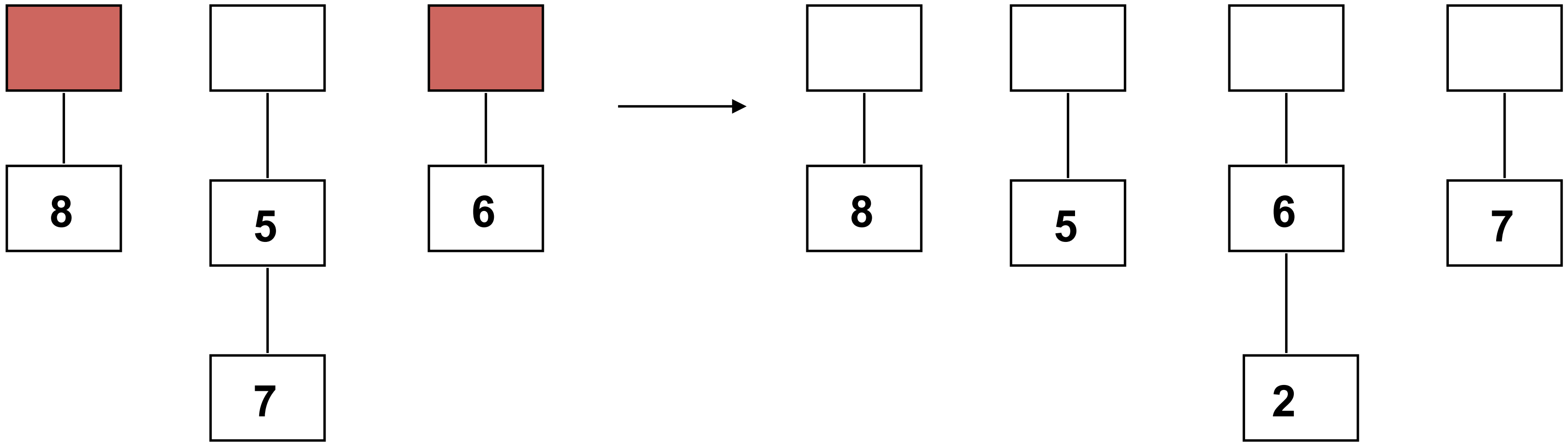


$h_1(k)$ = k mod 2
s = 0
Total capacity - 3 * table size
critical value = .5
Insert 8 ,7,5

Now, insert 5.
Table passes critical value.
Split 0.
s <-- 1

Insert 6.
$h_1(6) = 0$,
so use h_2
$h_2(6) = 2$

Example



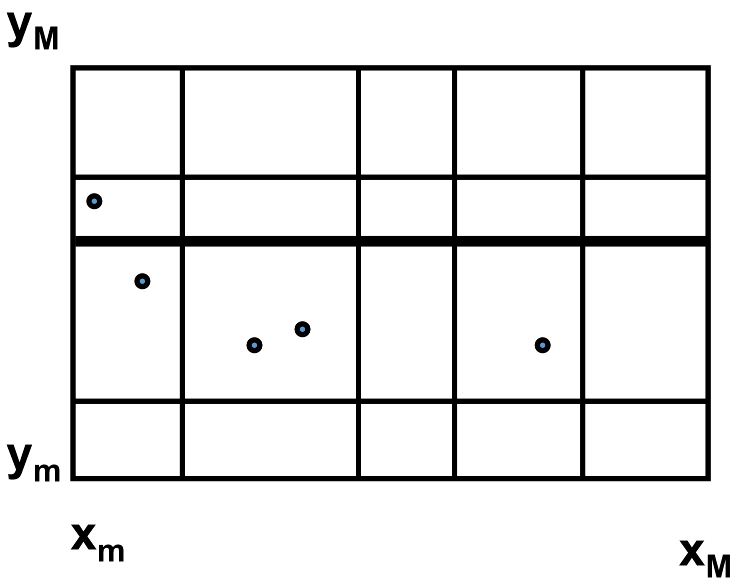
- Insert 2
- Table passes capacity
- Bucket 1 is split, and cycle is reset (s <-- 0)

Grid files - extendible hashing in 2D

- Method for storing large numbers of points on external storage
- Data space is bounded by $[x_m, x_M]$ and $[y_m, y_M]$ and is called the **grid space**.
  - at any time it is partitioned into a set of rectangular blocks by a collection of grid lines
- Data points, themselves, are stored in buckets on disks.
  - grid file contains one entry for each block in the current partition of the data space,
  - each entry in the grid file is a pointer to the bucket on disk where the data points contained within that partition are stored.
  - buckets have a fixed capacity - overflowing requires splitting

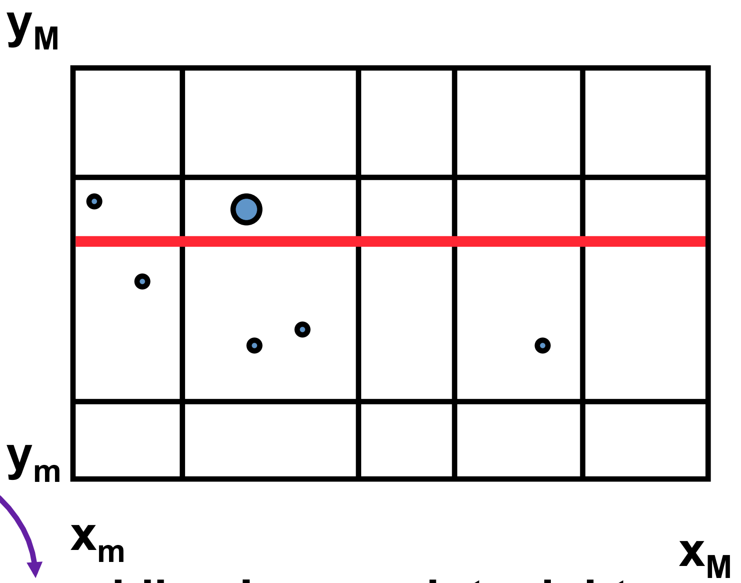
Grid files

- If we have m horizontal and m vertical grid lines, then we have defined m^2 grid blocks.
  - so, with 2000 grid lines (1000 vertical and 1000 horizontal) we define 1,000,000 grid blocks - each a pointer to where we will find the address of the bucket (disk address) where the data points are actually stored.



Grid files

- So, need two accesses to get to the points themselves; the grid file is organized this way because **many grid blocks can point to the same bucket.**
- method assumes that the grid lines themselves can be stored in memory. If we can store the grid block directory in memory as well, then only one disk access is required.



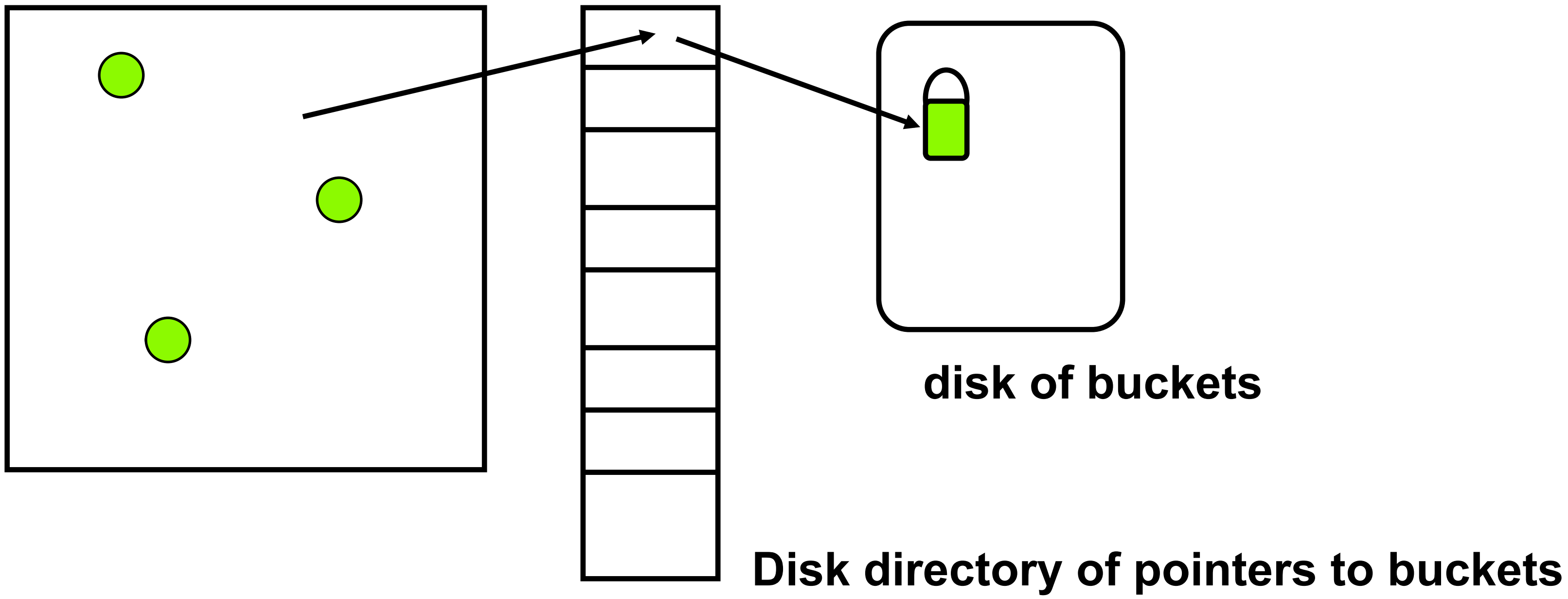
• adding large point might require inserting the bold horizontal line
• don't want to split all of the buckets that this line crosses, requires re-assigning points
• so, will have a many to one mapping from grid cells to buckets

Grid files

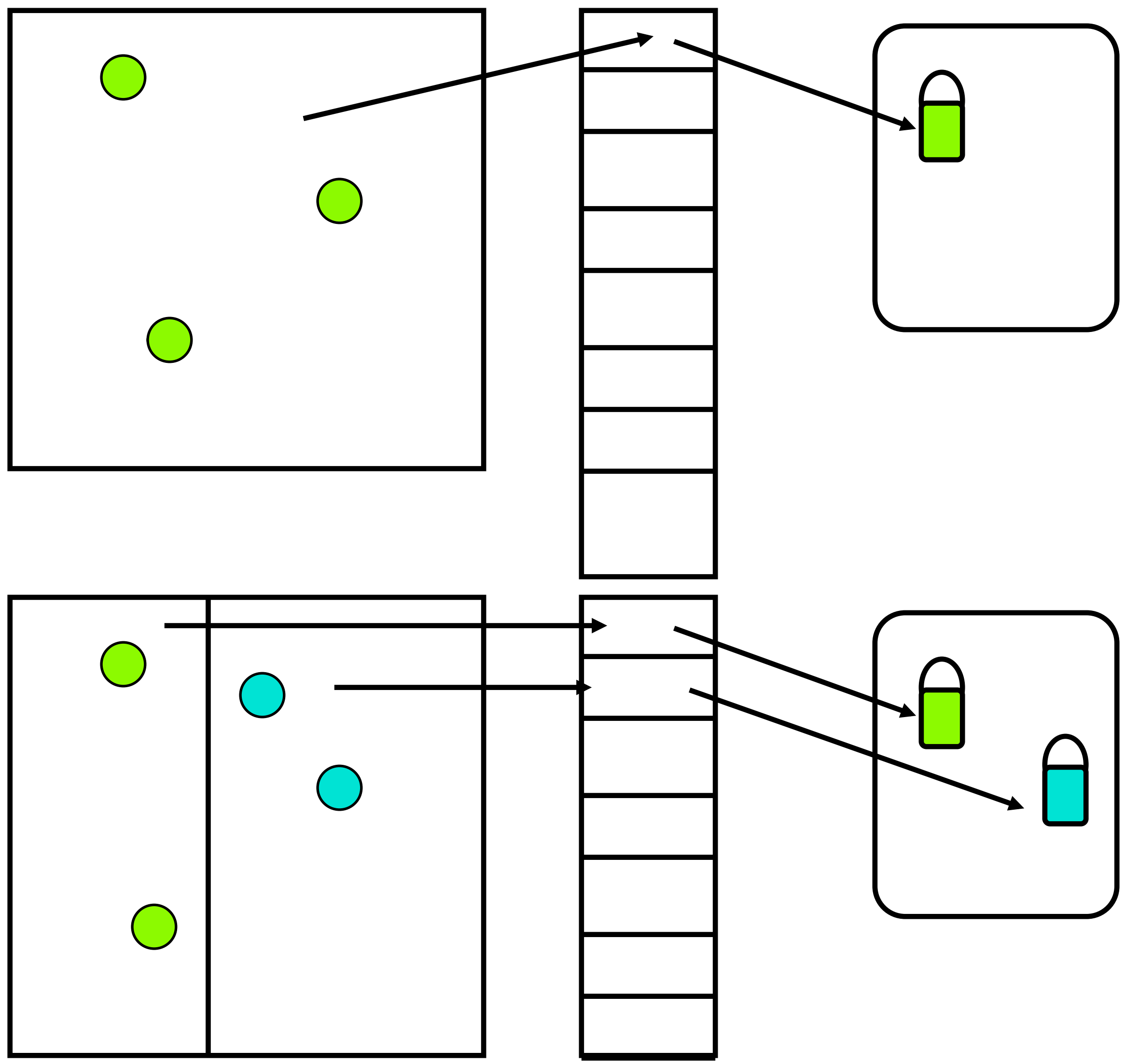
- Allow any rectangular set of grid blocks to point to the same bucket
  - makes it easy to split and merge buckets as points are added to or deleted from the set.
- Determining the grid block in which a point (x,y) lies
  - Maintain two 1-d arrays storing the x and y values of the grid lines.
  - Simple binary search of each array will yield the grid block (x’,y’) coordinates of (x,y)
  - Since directory size is known, can use coordinates of (x’, y’) to directly compute the location on disk in which this grid block is stored - say, using lexicographic ordering.

Updating grid files

- Initially, there is only one grid block for the entire data space
- Assume that bucket capacity is three points



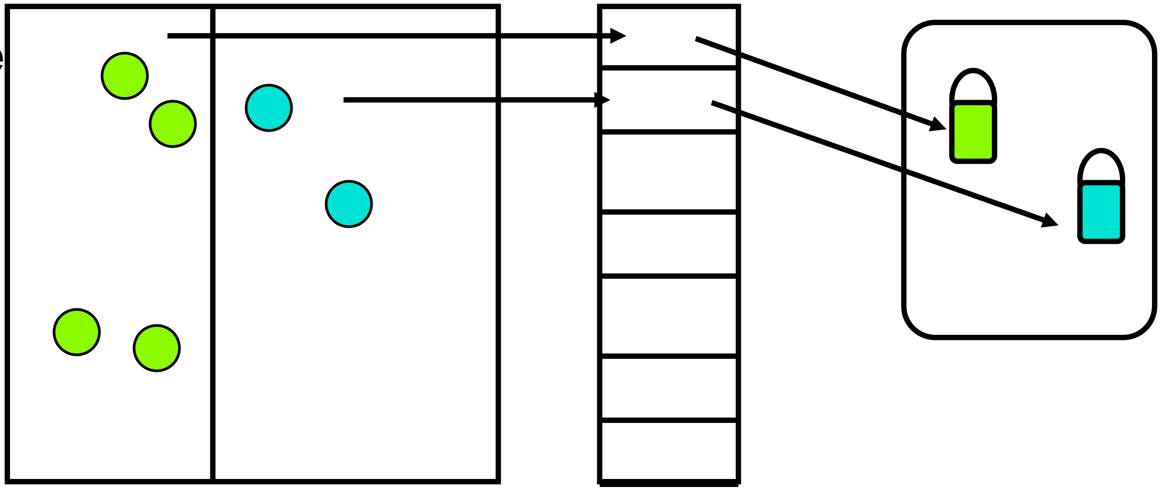
Updating grid files



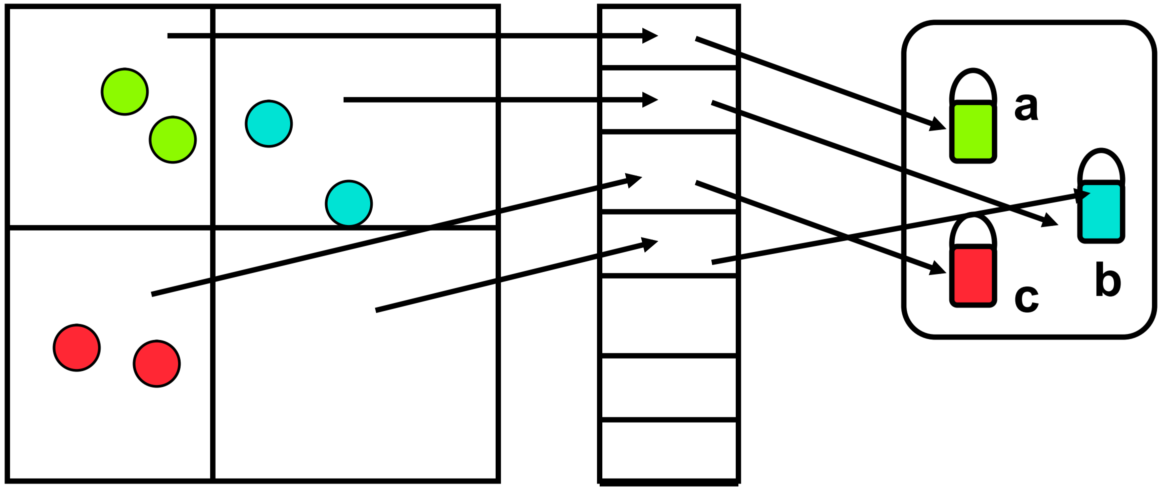
Updating grid files

- What is involved in this update?
  - Must read and update two directory entries on disk, which may or may not be on the same disk page
  - Must update first bucket, and create second bucket of points - another three disk accesses (R/W original bucket, W new bucket)

add 2 more points



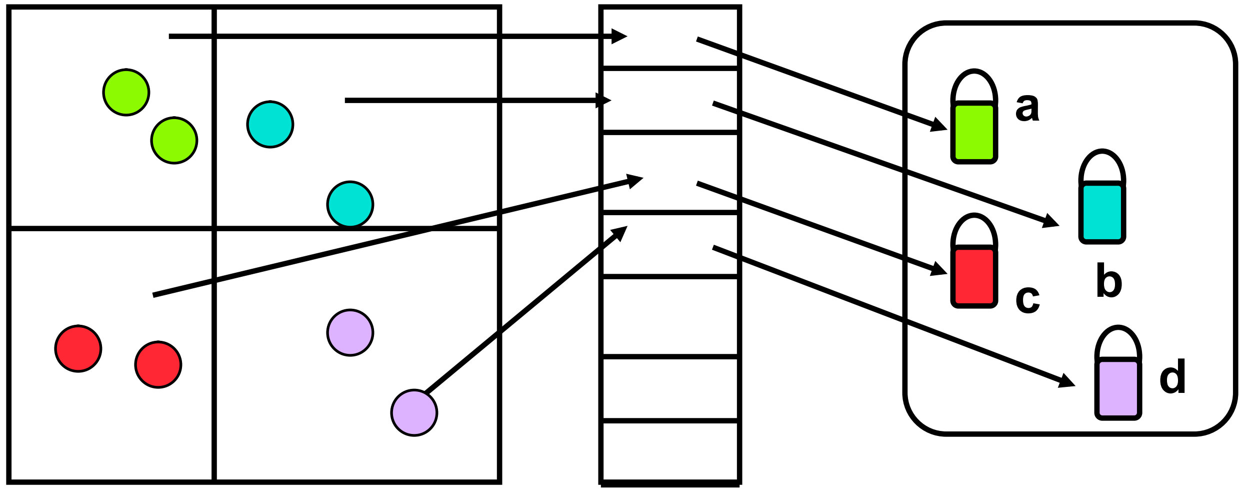
Updating grid files



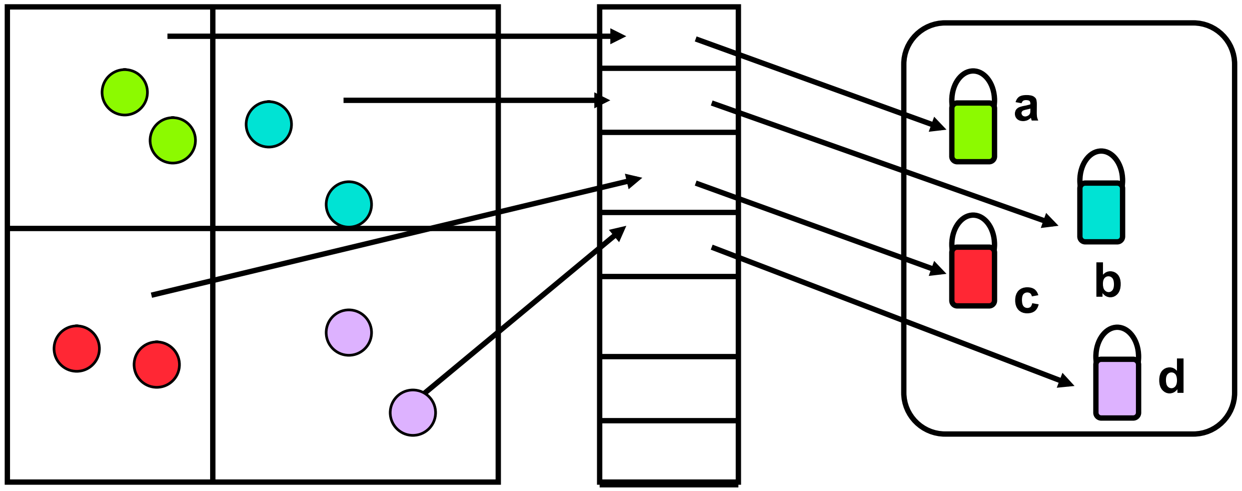
- Bucket a is split into buckets a and c
- Grid block (2,2) points to bucket b, even though it contains no points from that block - but every block must point to some bucket.

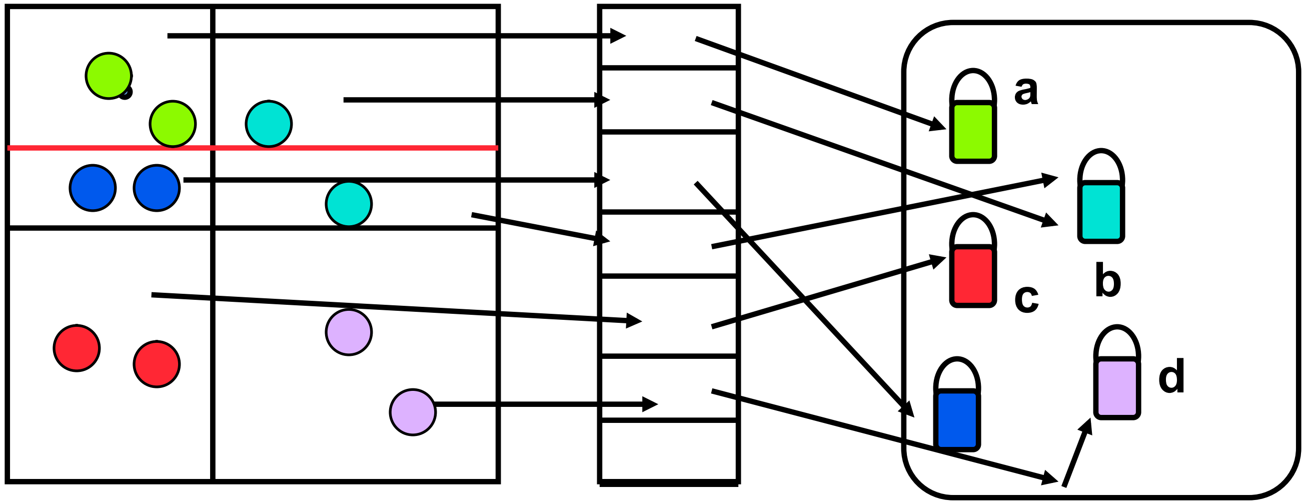
Updating grid files

- Now suppose that we add two points to grid block (2,2)
  - This cause the associated bucket, b, to overflow
  - But we do not have to add more grid lines, because we had the many to one mapping of grid blocks to buckets



Updating grid files





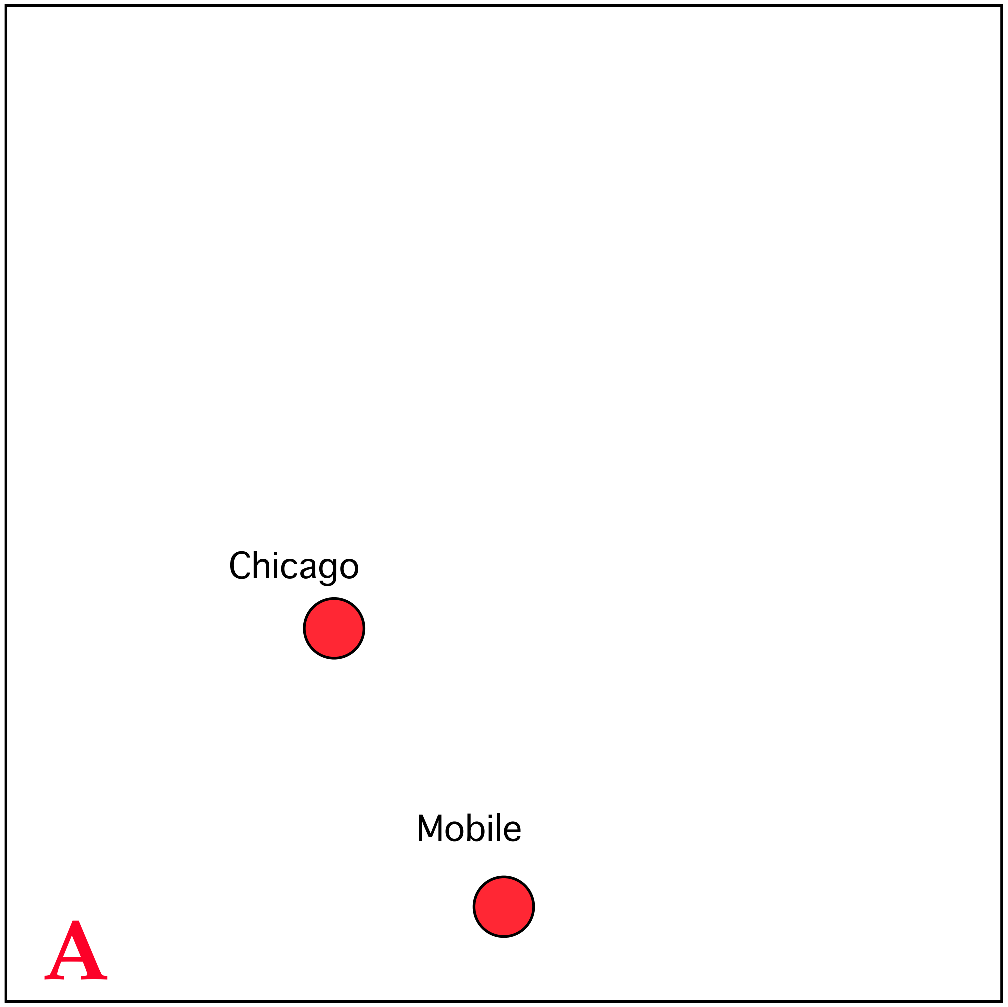
Updating grid files

- Generally, once we have a large number of grid lines, insertions will only lead to bucket splittings and not to the insertion of new grid lines.
- This is important for efficiency, because when we introduce new grid lines we have to rewrite large parts of the directory and pointers to it.

Extendible hashing in 2-D - EXCELL

- All grid blocks of the same size
- Grid refinement for grid files splits one interval (either x or y) into two intervals
- For EXCELL, refinement splits all intervals in two along fixed partition lines and doubles the directory like extendible hashing.
- Since all grid blocks are the same size, the linear scales used by the grid file are no longer needed. Can use the Morton code of the grid block to index into set of pointers to data buckets.

Example

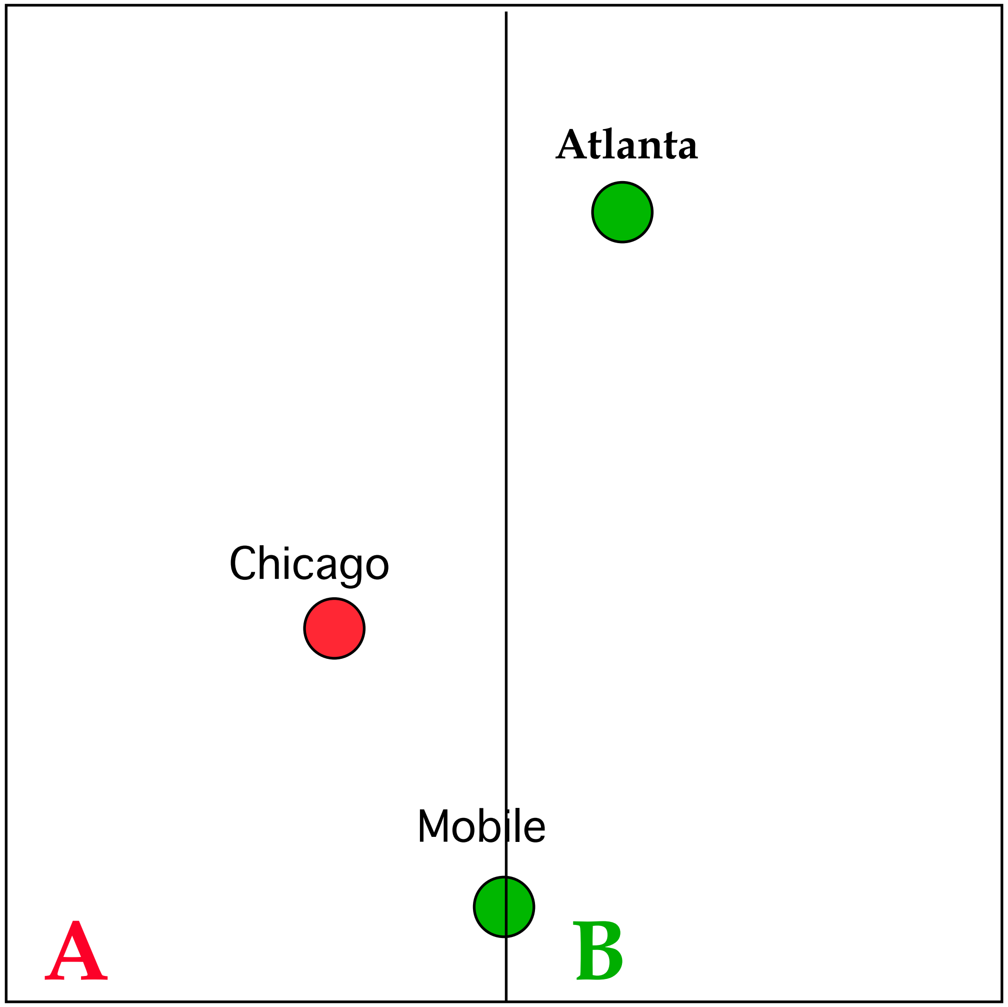


Bucket capacity is 2 records

Grid directory implemented as an array, initially containing one element corresponding to the entire space

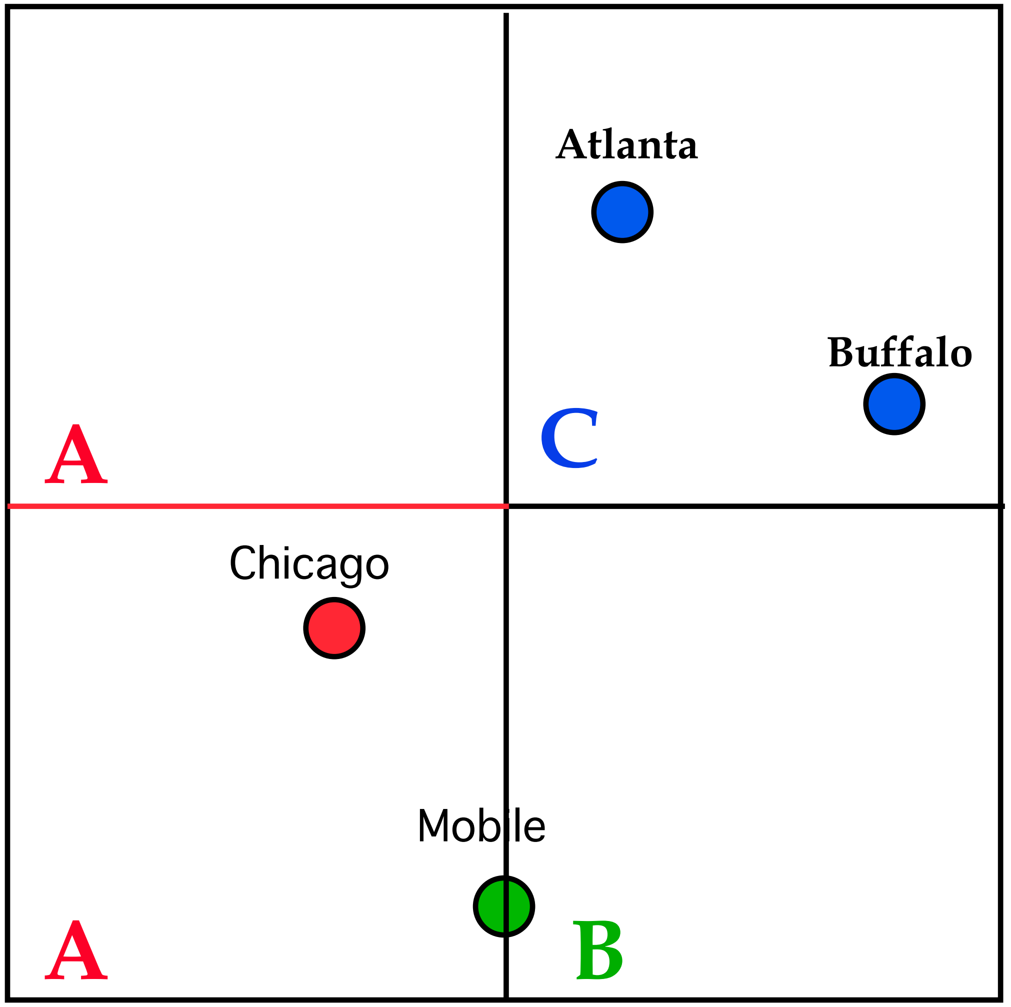
Inserting Chicago and Mobile fills up bucket A

Example



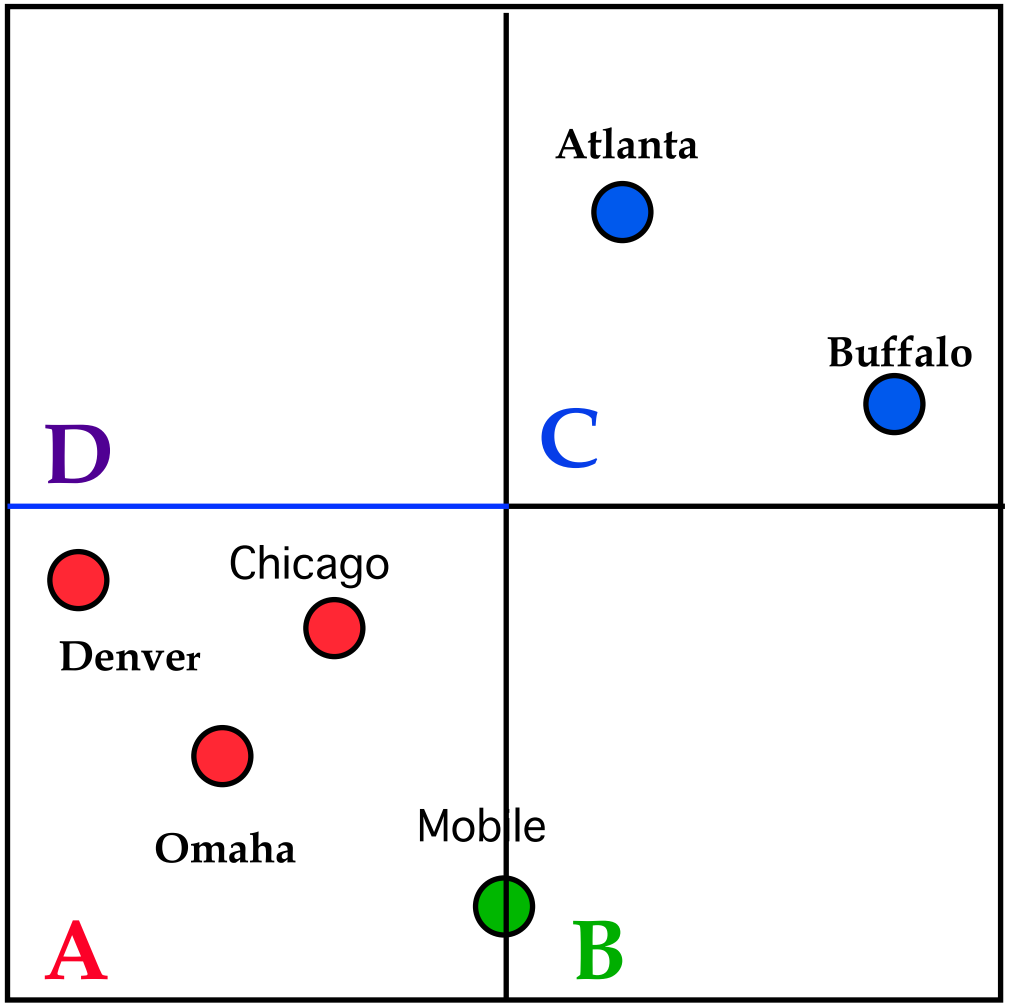
- Inserting Toronto causes bucket A to overflow
- Must double the directory to include 2 elements
- Split bucket A and move Toronto and Mobile to bucket B

Example



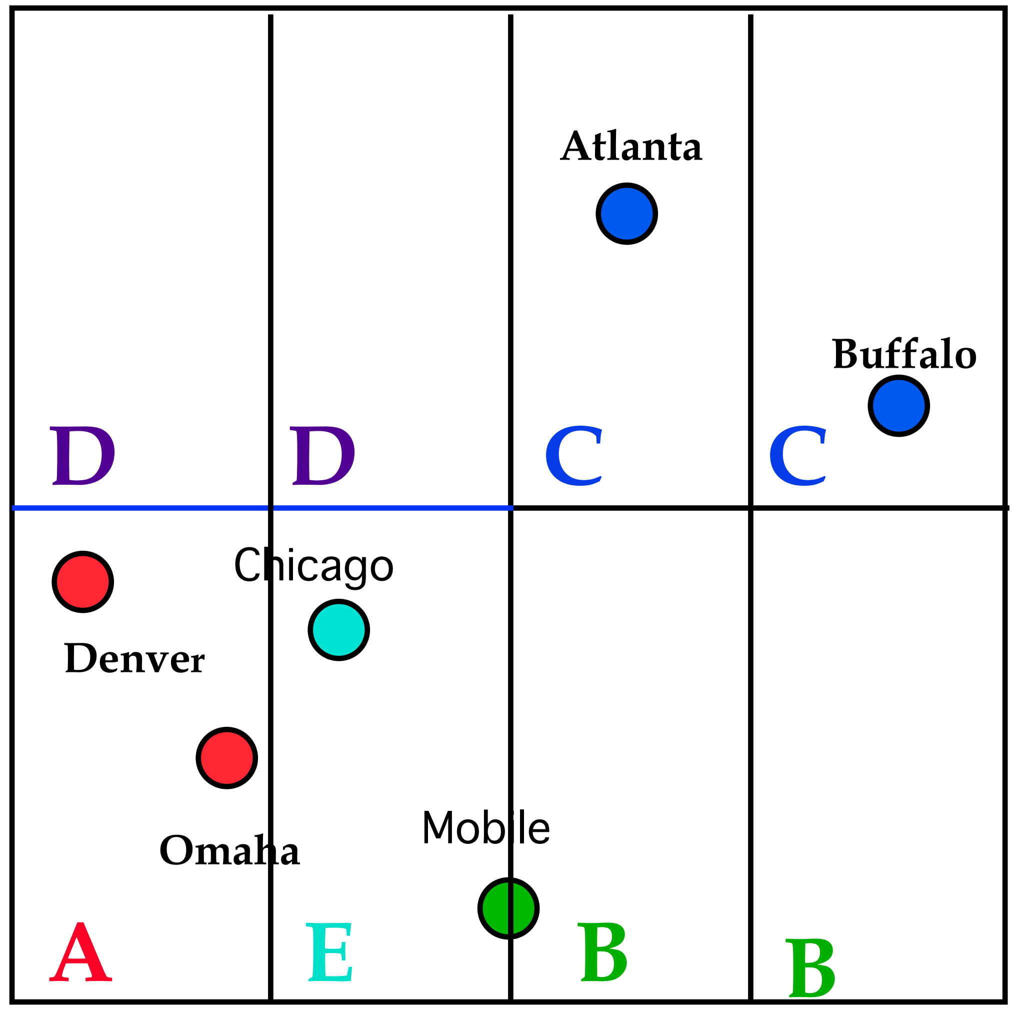
- Inserting Buffalo causes bucket B to overflow
- Double the directory by splitting along y

Example



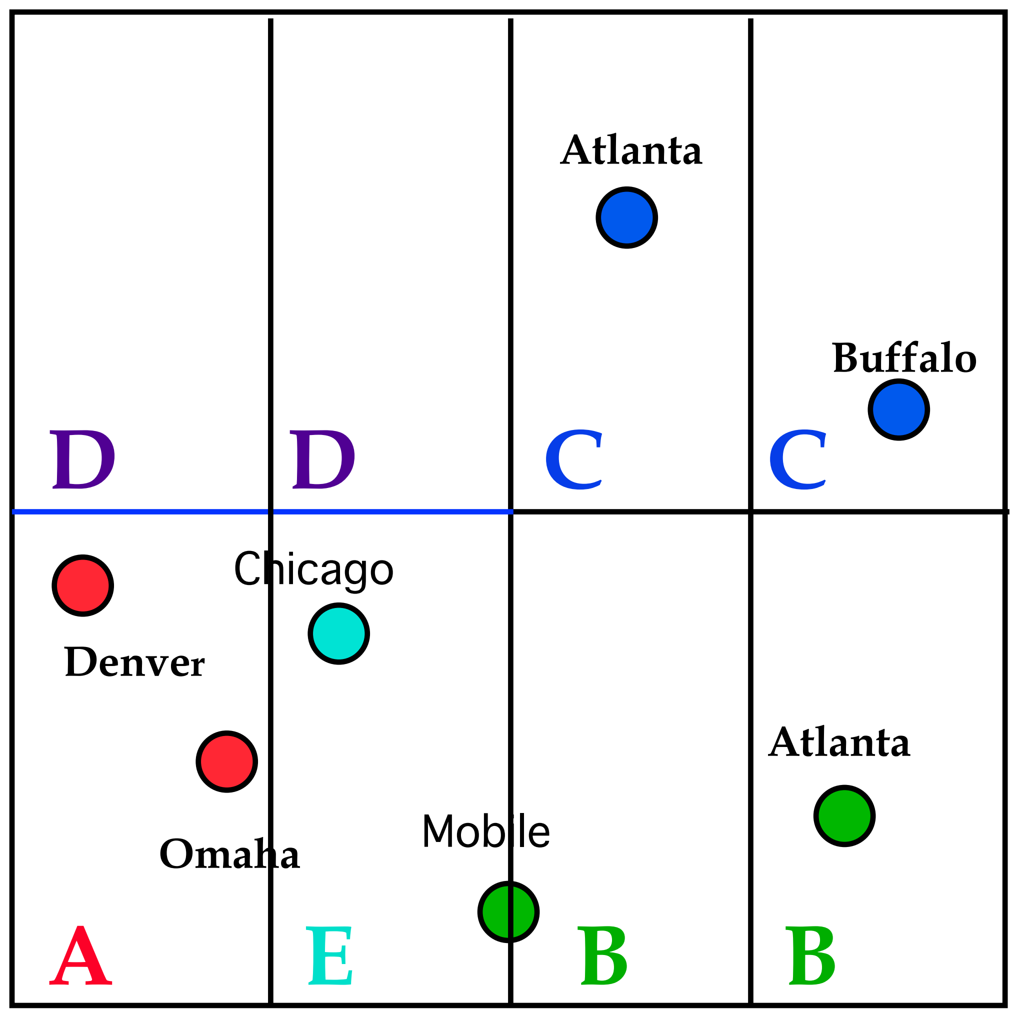
- Denver can be placed into bucket A
- Omaha also belongs in bucket A
  - bucket A can be split in two without changing the size of the directory - creating D
  - However, all of the points are in A!

Example



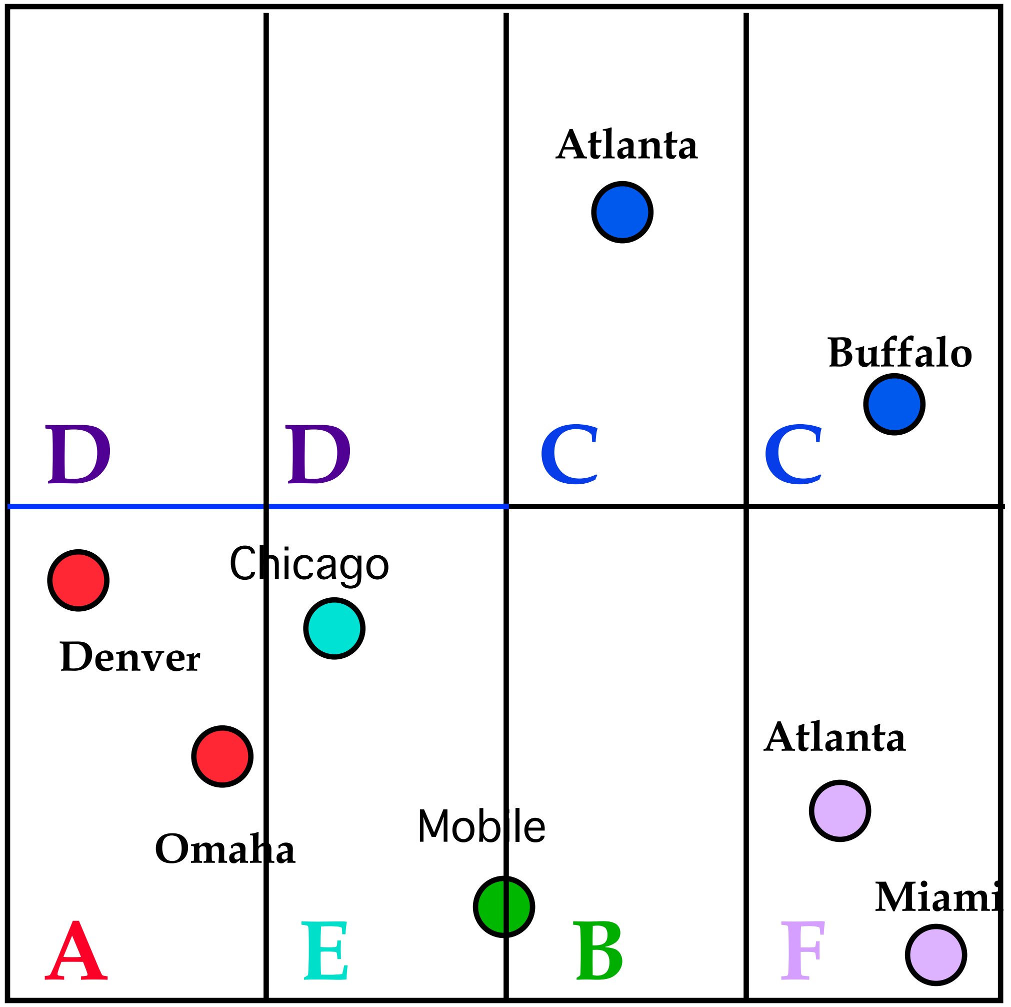
- Split along the x attribute again doubling the directory

Example



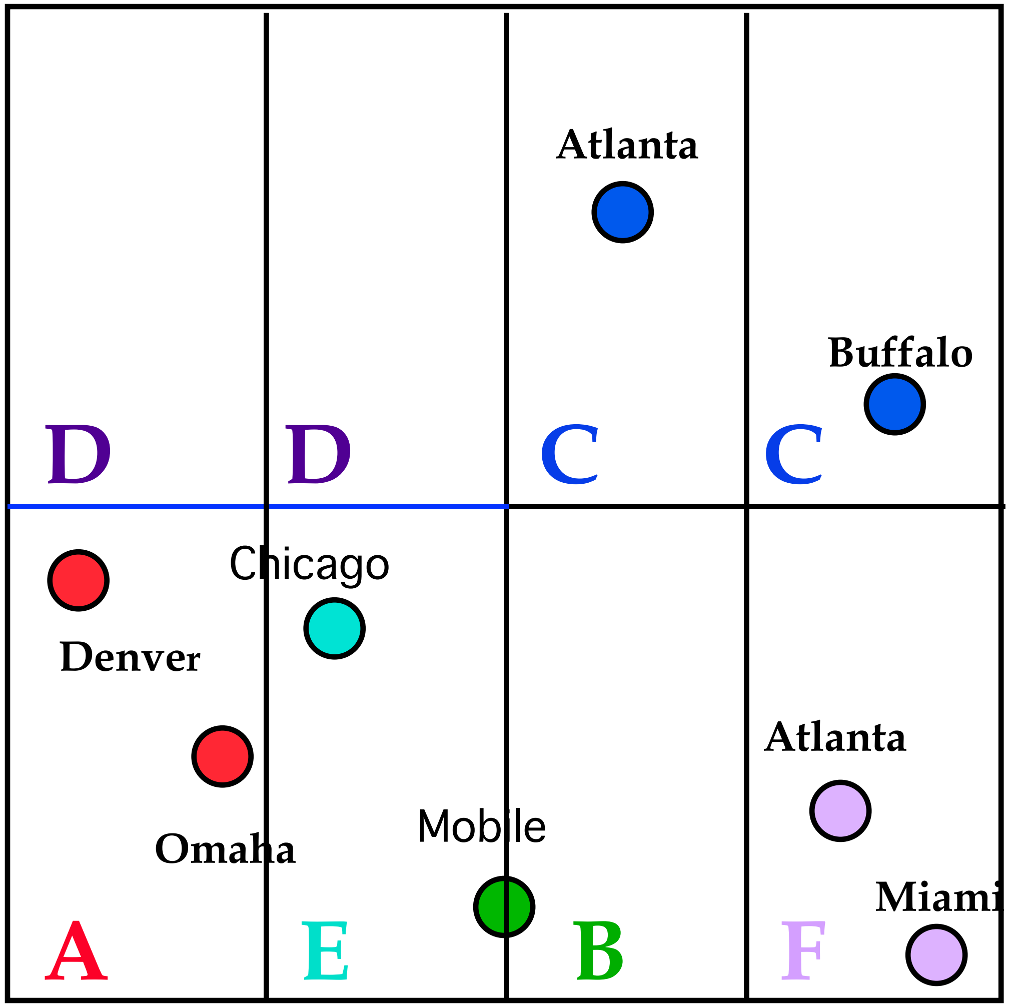
- Insert Atlanta into bucket B - OK

Example



- But insertion of Miami into bucket B cause B to overflow
- Bucket B is split into B and F, but the directory does not have to be changed.

Example



1,1	A
1,2	D
2,1	E
2,2	D
3,1	B
3,2	C
4,1	F
4,2	C