Direct Files and Hashing - 2

Today

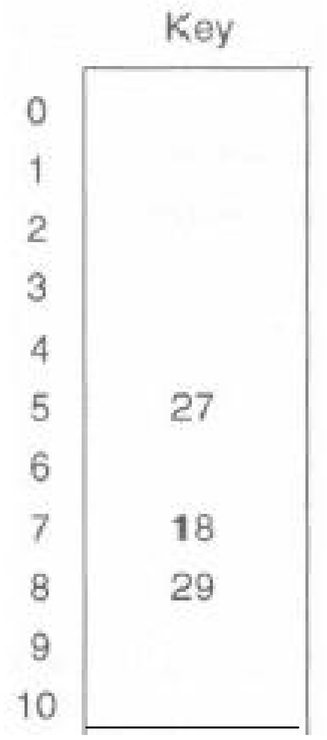
- Direct File Organization
  - Progressive Overflow
  - Buckets
  - Linear Quotient
  - Brent’s Method
  - Binary Tree

Progressive Overflow…

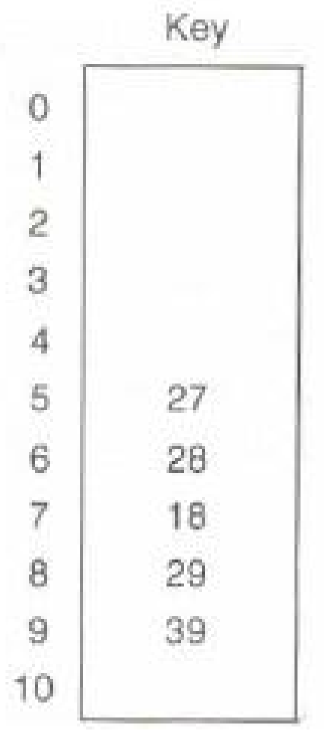
- The primary disadvantage of coalesced hashing: additional storage is needed for the link fields
- In Progressive Overflow (or Linear Probing), when a location is occupied, we look at the *next* location to see if it is empty
- Circular structure; stop when we find an empty slot or until we encounter the home address of the record a second time (full table)
- Poor performance with unsuccessful search, since we continue searching until we encounter an empty space or the initial search address

Example…

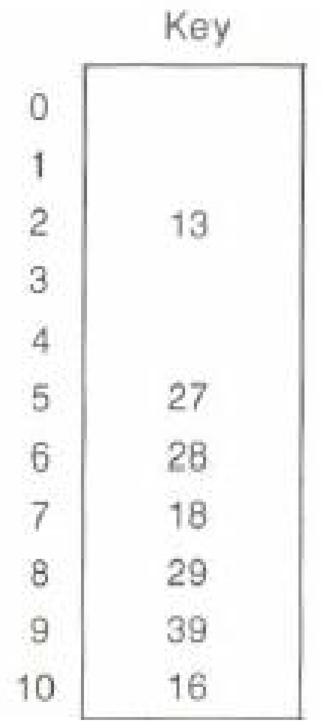
- Keys: 27, 18, 29, 28, 39, 13, and 16
- Hash(key) = key **mod** 11



	Key
0	
1	
2	
3	
4	
5	27
6	
7	18
8	29
9	
10	



	Key
0	
1	
2	
3	
4	
5	27
6	28
7	18
8	29
9	39
10	



	Key
0	
1	
2	13
3	
4	
5	27
6	28
7	18
8	29
9	39
10	16

Progressive Overflow - Evaluation

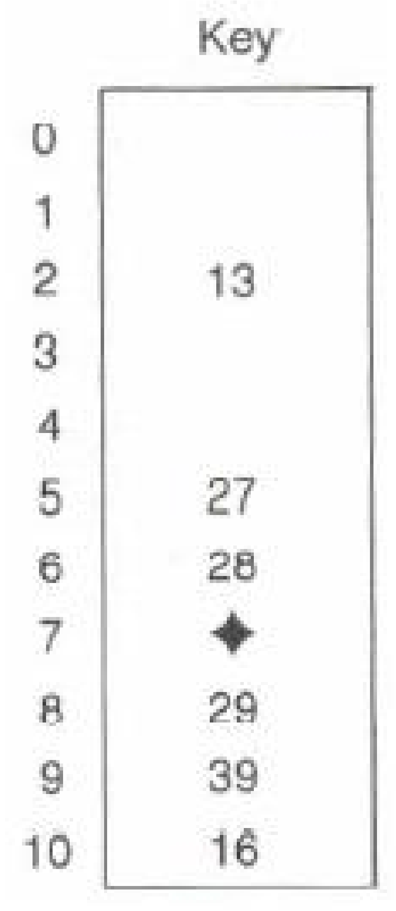
- To retrieve the record with key 16 we must make six probes ! (due to secondary clustering)
- **Secondary Clustering** is the consequence of two or more records following the same sequence of probe addresses
- Primary Clustering occurs when a large number of records have the *same* home address
- Secondary Clustering is caused by records with *different* home addresses

Progressive Overflow – Evaluation 2

- The greatest disadvantage of PO is the large number of retrieval probes associated with it which results to a great extent because of secondary clustering
- Because of the high cost of secondary storage access, and the relatively large number of retrieval probes associated with it, PO is not a practical method for handling collisions
- The average number of probes to retrieve each record in the file one is 2.3 probes, much higher than the 1.4 average probes for coalesced hashing on the equivalent records

Progressive Overflow – Evaluation 3

- In a sense, progressive overflow reduces to a sequential search
- The differences are that progressive overflow uses a variable starting point and it does not need to search the entire file for an unsuccessful search
- To delete a record requires that care to be taken so that we are still able to retrieve the records that remain in the file
- We place an indicator (tombstone) in the location of the deleted record



18 is deleted

Progressive Overflow – Evaluation 4

- To improve insertion performance, a bit string is kept in the primary memory
- A “one” in the bit string would indicate that the corresponding location was occupied
- On insertion, then, instead of making accesses to auxiliary storage, we would only need to check the bit string in primary memory to find the first unoccupied location in the file for inserting the record
- What if we may have duplicate keys?
- What if we need to check if an item had been inserted previously?

Use of Buckets ...

- A **bucket** (a.k.a. page or block) is a storage unit intermediate between a record and a file; it is also the unit in which information is accessed and transferred between storage devices
- **Blocking Factor:** The number of records that may be stored in a bucket
- As the BF increases, the number of auxiliary storage accesses decreases because more colliding records can be stored at one location

Use of Buckets: An Example…

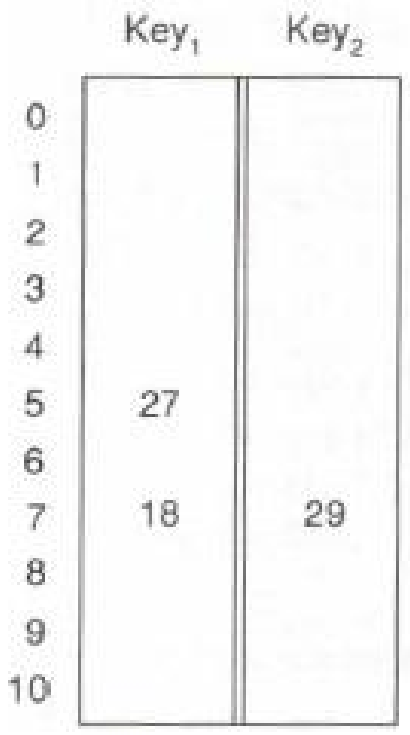
- Keys: 27, 18, 29, 28, 39, 13, and 16
- Hash(key) = key **mod** 11
- Blocking factor is 2

- The average number of probes is 1.0
- The packing factor is 7/22 = 32%
- It was 7/11 = 64% in Progressive Overflow

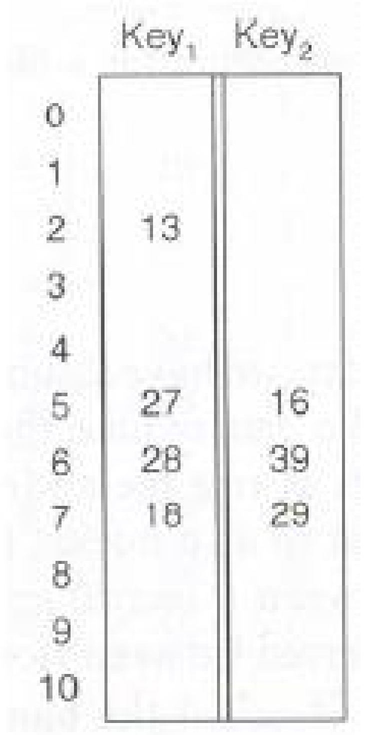
Use of Buckets: An Example…

- Keys: 27, 18, 29, 28, 39, 13, and 16
- Hash(key) = key **mod** 11
- Blocking factor is 2

- The average number of probes is 1.0
- The packing factor is 7/22 = 32%
- It was 7/11 = 64% in Progressive Overflow



	Key_1	Key_2
0		
1		
2		
3		
4		
5	27	
6		
7	18	29
8		
9		
10		



	Key_1	Key_2
0		
1		
2	13	
3		
4		
5	27	16
6	28	39
7	18	29
8		
9		
10		

Use of Buckets: Discussion

- Within a bucket,we need a means of seperating the individual records
- We can achieve this by knowing the record length for fixed-length records, or by placing a special delimiter between variable-length records
- Time to locate a desired record is insignificant when compared with the time required for an access of auxiliary memory

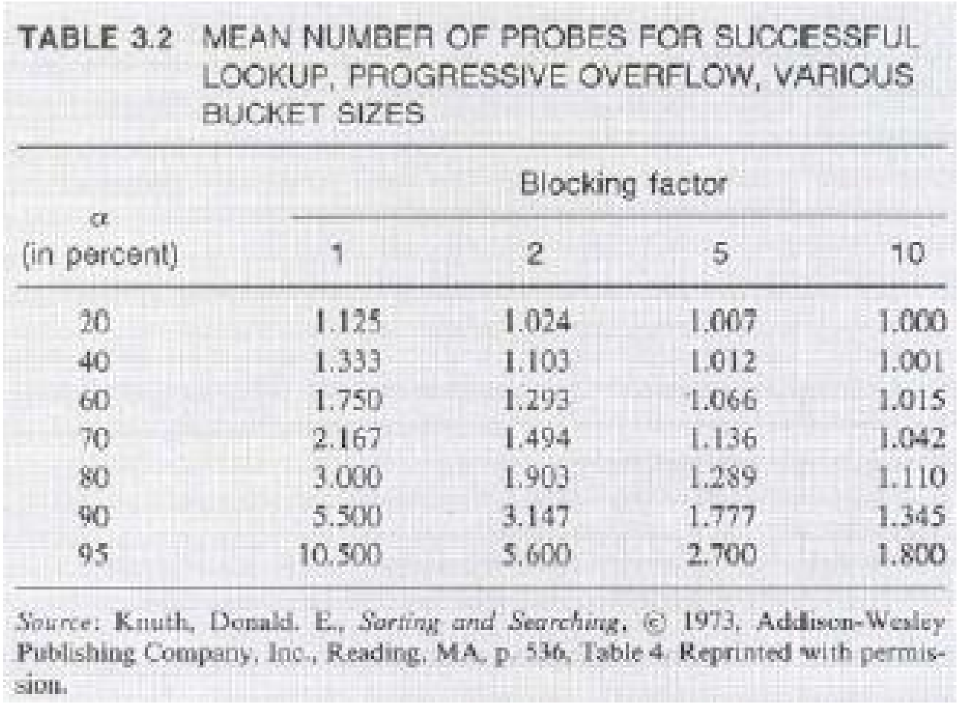


TABLE 3.2 MEAN NUMBER OF PROBES FOR SUCCESSFUL LOOKUP, PROGRESSIVE OVERFLOW, VARIOUS BUCKET SIZES

α (in percent)	Blocking factor 1	2	5	10
20	1.125	1.024	1.007	1.000
40	1.333	1.103	1.012	1.001
60	1.750	1.293	1.066	1.015
70	2.167	1.494	1.136	1.042
80	3.000	1.903	1.289	1.110
90	5.500	3.147	1.777	1.345
95	10.500	5.600	2.700	1.800

Source: Knuth, Donald. E., *Sorting and Searching*, © 1973, Addison-Wesley Publishing Company, Inc., Reading, MA, p. 536, Table 4. Reprinted with permission.

Linear Quotient…

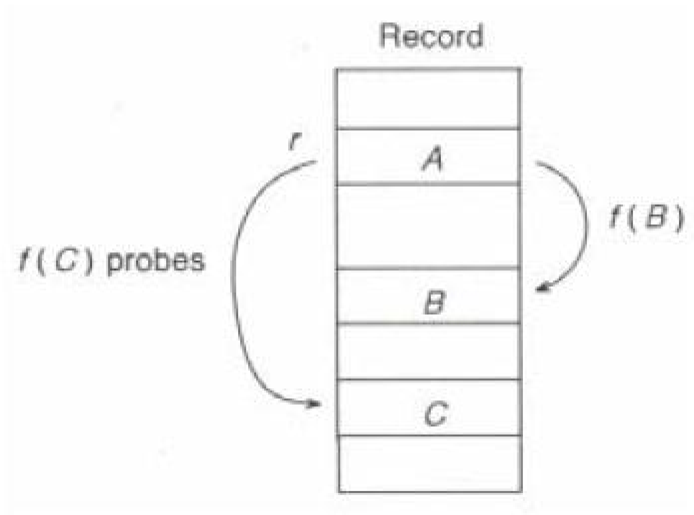
- Very similar to Progressive Overflow Algorithm
- LQ uses a *variable* increment instead of a constant increment of one in PO
- The purpose is to reduce the secondary clustering and consequently the average number of probes
- Also known as a form of “**double hashing**”
  - Hash once to get the home address (H_1)
  - Hash the second time to get the increment (H_2)
- H_2=quotient (Key/P) **mod** P or
- H_2’=(Key **mod** (P-2)) + 1

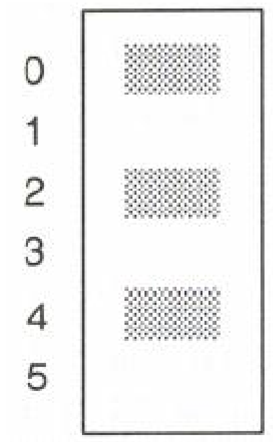
Linear Quotient Algorithm…

- Hash the key of the record to be inserted to obtain the home address for storing the record
- If the home address is empty, insert the record at that location, else
    - Determine the increment by obtaining the quotient of the key divided by the table size. If the result is zero, set the increment to one.
    - Initialize the count of locations searched to one.
    - While the number of locations searched is less than the table size
        - Compute the next search address by adding the increment to the current address and then moding the result by the table size
        - If the address in unoccupied, then insert the record and terminate with a successful insertion
        - If the record occupying the location has the same key as the record being inserted, terminate with a “duplicate record” message
        - Add one to the count of the locations searched
    - Terminate with a “full table” message

Linear Quotient…2

- Unlike coalesced hashing, synonyms are not usually on the same probe chain with linear quotient
- LQ requires a prime number table size
- Otherwise, the searching could cycle through a subset of the table several times and then indicate a full table when space is available
- Example: Try to insert a record with a home address of 0



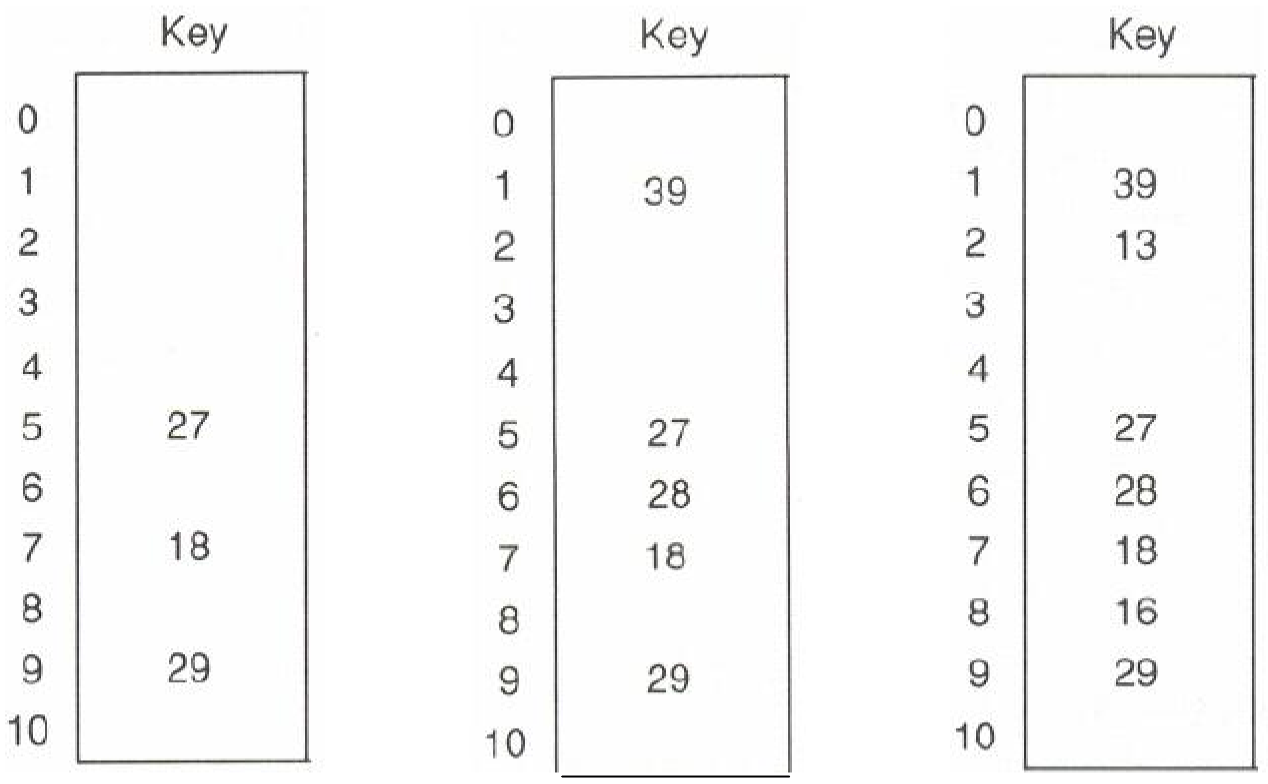


Linear Quotient: An Example

- Keys: 27, 18, 29, 28, 39, 13, 16
- H_1(key) = key mod 11
- H_2 = Quotient (key / 11) mod 11

Linear Quotient: An Example

- Keys: 27, 18, 29, 28, 39, 13, 16
- $H_1(key)$ = key mod 11
- H_2 = Quotient (key / 11) mod 11



	Key
0	
1	
2	
3	
4	
5	27
6	
7	18
8	
9	29
10	

	Key
0	
1	39
2	
3	
4	
5	27
6	28
7	18
8	
9	29
10	

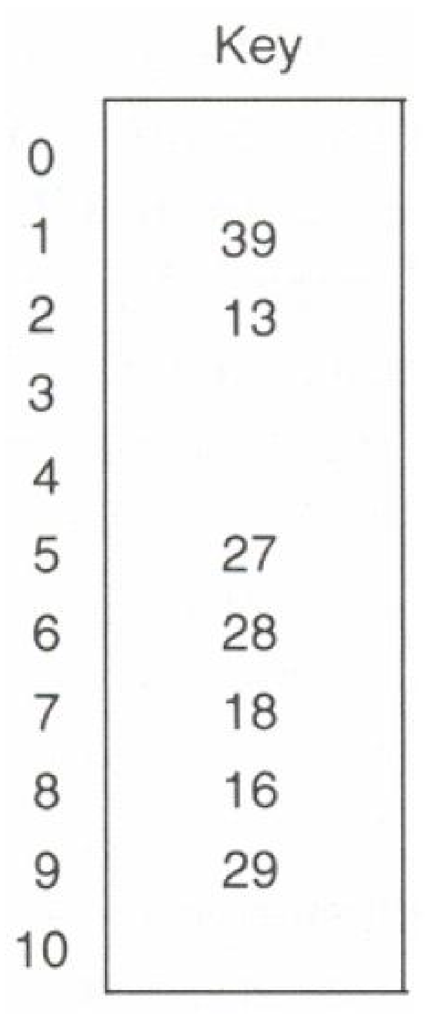
	Key
0	
1	39
2	13
3	
4	
5	27
6	28
7	18
8	16
9	29
10	

Linear Quotient: Discussion

- In the example, retrieving 16 requires only four probes compared with the six with progressive overflow
- Overall we need an average of 13/7=1.9 probes for retrieving the records inserted via linear quotient vs. the 2.3 average probes for progressive overflow and 1.4 for coalesced hashing with LISCH
- The variable increment has allowed us to break up the secondary clusters of records which improves the performance of both successful and unsuccessful searches

Linear Quotient: Discussion... 2

- We can improve upon LQ by observing that the number of retrieval probes is dependent upon the placement of the records
- For example, if we want to insert a record with a key of 67, it collides with 39
- Then, we try to insert 67 at locations 7, 2, and 8 until we finally discover an empty slot at location 3
- What if 39 had not been stored at location 1?



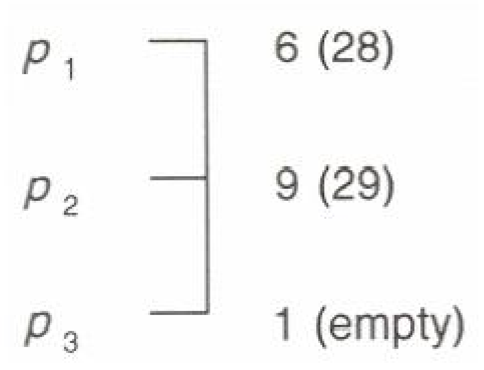
	Key
0	
1	39
2	13
3	
4	
5	27
6	28
7	18
8	16
9	29
10	

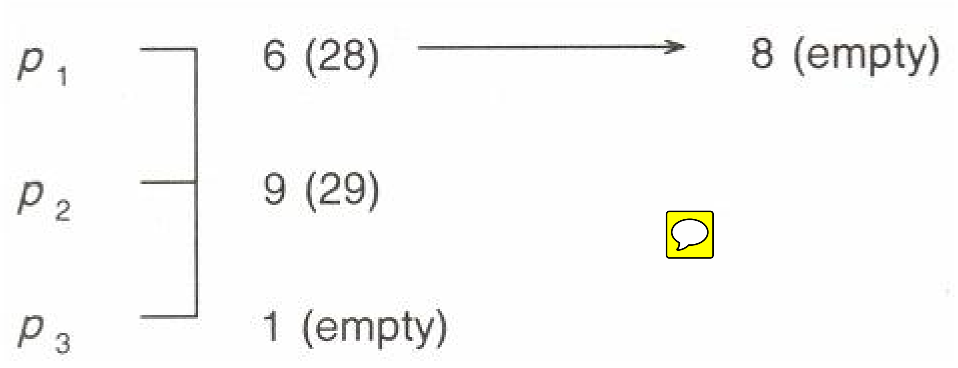
Dynamic methods…

- All the collision resolution methods that we have considered so far are *static* methods
- In static methods, we do not move an item after it is stored
- Dynamic methods, on the other hand, can move any item, and therefore, <u>require additional processing</u> when inserting a record into the table but <u>reduce the number of probes needed for retrieval</u>
- Justification: We usually insert an item into a table only once but retrieve it many times

Brent's Method

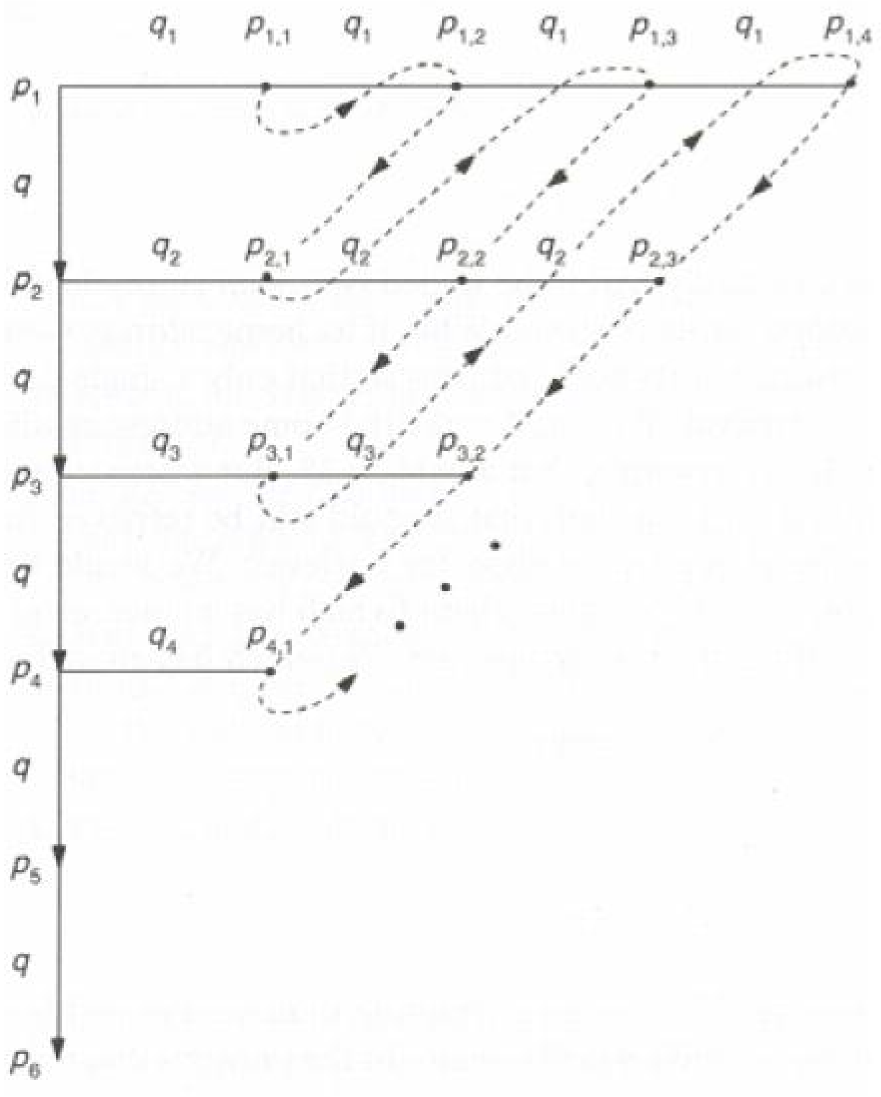
- This is a dynamic method
- **Primary probe chain** of a record is the sequence of locations visited during the insertion or retrieval of the record
- **Secondary probe chain** of the record is the sequence of locations visited when attempting to move a record from the primary probe chain





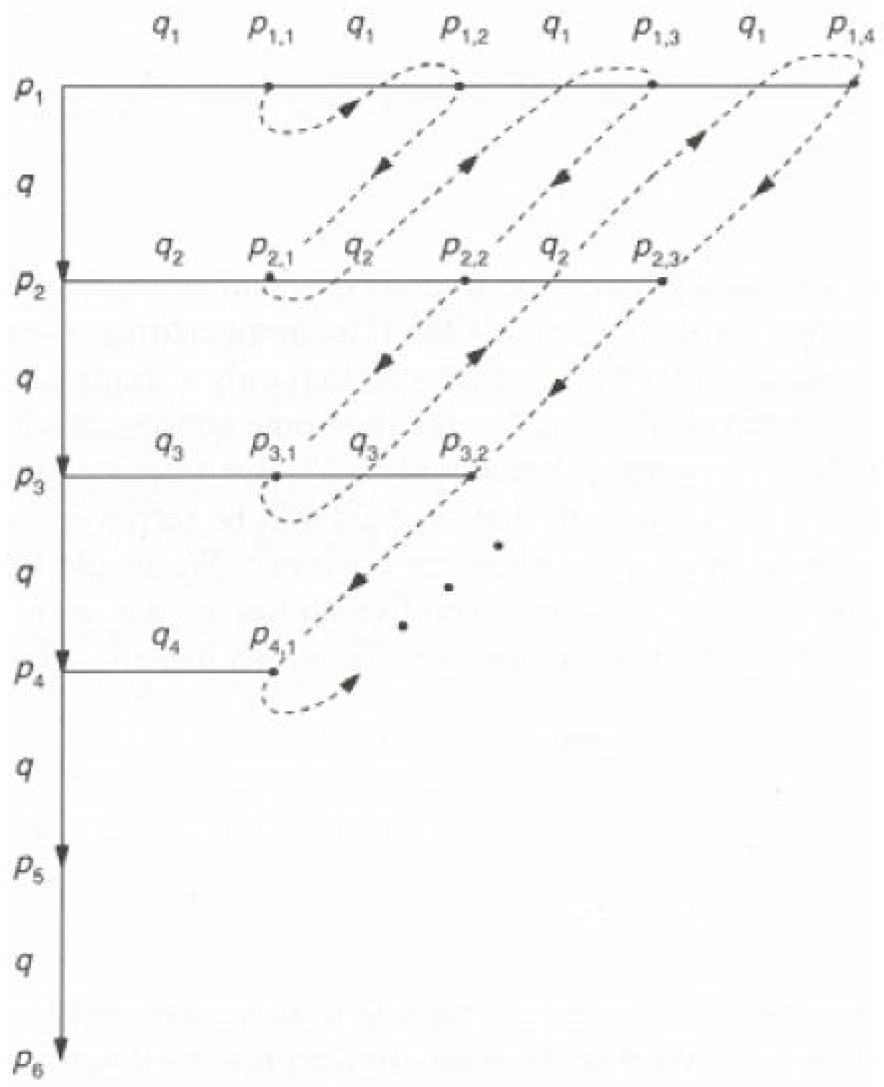
Brent's Method…2

- The solid vertical line represents the primary probe chain, that is, the addresses that would be considered in storing an item using the LQ scheme
- The horizontal lines represent the secondary probe chains, that is, the addresses that would be searched in attempting to move an item from a position along the primary chain



Brent's Method…3

- The *q* value along the primary probe chain is the increment for the item being inserted
- The q_i's along the secondary probe chains represent the increments associated with the item being moved

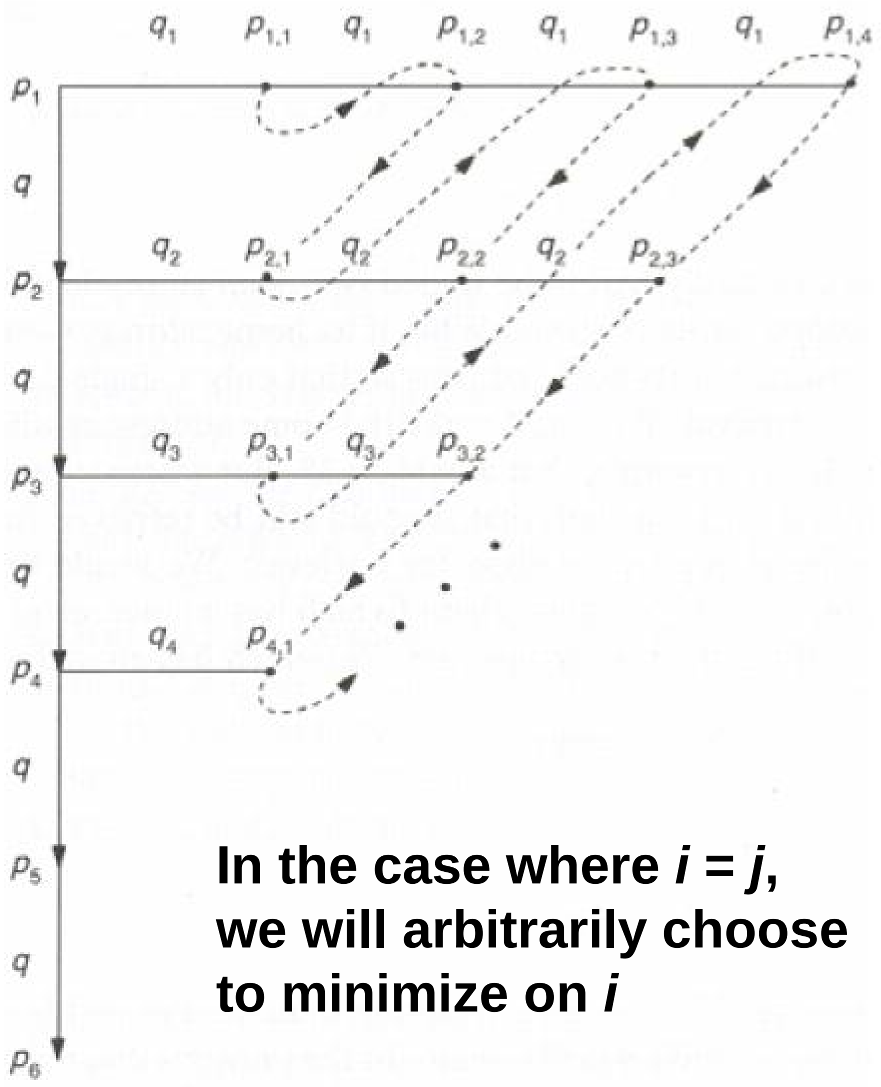


q_1 $p_{1,1}$ q_1 $p_{1,2}$ q_1 $p_{1,3}$ q_1 $p_{1,4}$
p_1
q
q_2 $p_{2,1}$ q_2 $p_{2,2}$ q_2 $p_{2,3}$
p_2
q
q_3 $p_{3,1}$ q_3 $p_{3,2}$
p_3
q
q_4 $p_{4,1}$
p_4
q
p_5
q
p_6

Brent's Method…3

$P_{i,j}$

- The subscript *i* gives the number of probes needed to retrieve the item being inserted along its primary probe chain
- The subscript *j* gives the number of additional probes needed to retrieve the item being moved along its secondary probe chain
- To minimize the number of retrieval probes, we want to minimize (*i* + *j*)



In the case where *i* = *j*, we will arbitrarily choose to minimize on *i*

Brent’s Method: The Algorithm…

- Hash the key of the record to be inserted to obtain the home address for storing the record
- If the home address is empty, insert the record at that location, else
  - Compute the next potential address for storing the incoming record. Initialize *s* ← 2.
  - While the potential storage address is not empty,
    - Check if it is the home address. If it is, the table is full, terminate with a “full table” message.
    - If the record stored at the potential storage address is the same as the incoming record, terminate with a “duplicate record” message.
    - Compute the next potential address for storing the incoming record. Set *s* ← *s* + 1

  /* Attempt to move a record previously inserted */
  - Initialize i ← 1 and j ← 1
  - While (i + j < s)
    - Determine if the record stored at the ith position on the primary probe chain can be moved j offsets along its secondary probe chain.
    - If it can be moved, then
      - Move it and insert the incoming record into the vacated position i along its primary probe chain; terminate with a successful insertion, else
      - Vary i and/or j to minimize the sum of (i + j); if i = j, minimize on i.

  /* Moving has failed */
  - Insert the incoming record at position s on its primary probe chain; terminate with a successful insertion

Brent’s Method: The Algorithm…1

- Hash the key of the record to be inserted to obtain the home address for storing the record
- If the home address is empty, insert the record at that location, else
  - Compute the next potential address for storing the incoming record. Initialize $s \leftarrow 2$.
  - While the potential storage address is not empty,
    - Check if it is the home address. If it is, the table is full, terminate with a “full table” message.
    - If the record stored at the potential storage address is the same as the incoming record, terminate with a “duplicate record” message.
    - Compute the next potential address for storing the incoming record. Set $s \leftarrow s + 1$

Brent’s Method: The Algorithm…2

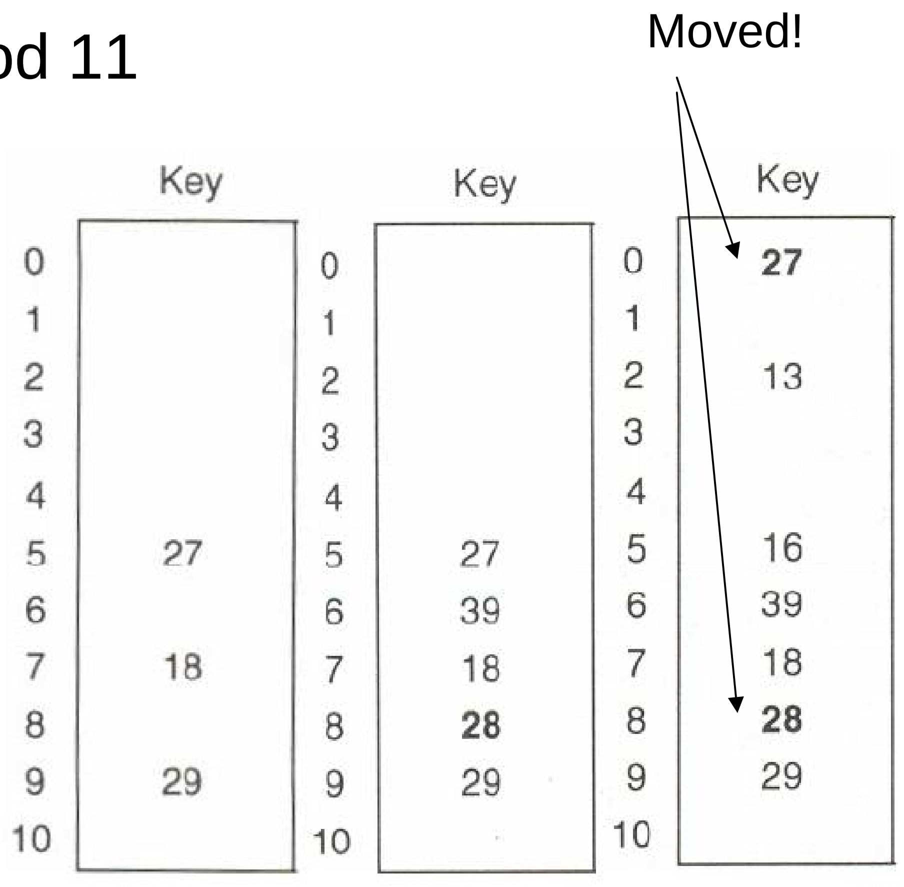
```
/* Attempt to move a record previously inserted */
□ Initialize i ← 1 and j ← 1
□ While (i + j < s)
    ■ Determine if the record stored at the ith position on the primary
      probe chain can be moved j offsets along its secondary probe
      chain.
    ■ If it can be moved, then
        □ Move it and insert the incoming record into the vacated position i
          along its primary probe chain; terminate with a successful
          insertion, else
        □ Vary i and/or j to minimize the sum of (i + j); if i = j, minimize on i.
/* Moving has failed */
□ Insert the incoming record at position s on its primary
  probe chain; terminate with a successful insertion
```

Brent’s Method: An Example

- Keys: 27, 18, 29, 28, 39, 13, 16
- H_1(key) = key mod 11
- H_2 = Quotient (key / 11) mod 11

Brent’s Method: An Example

- Keys: 27, 18, 29, 28, 39, 13, 16
- $H_1(\text{key})$ = key mod 11
- H_2 = Quotient (key / 11) mod 11



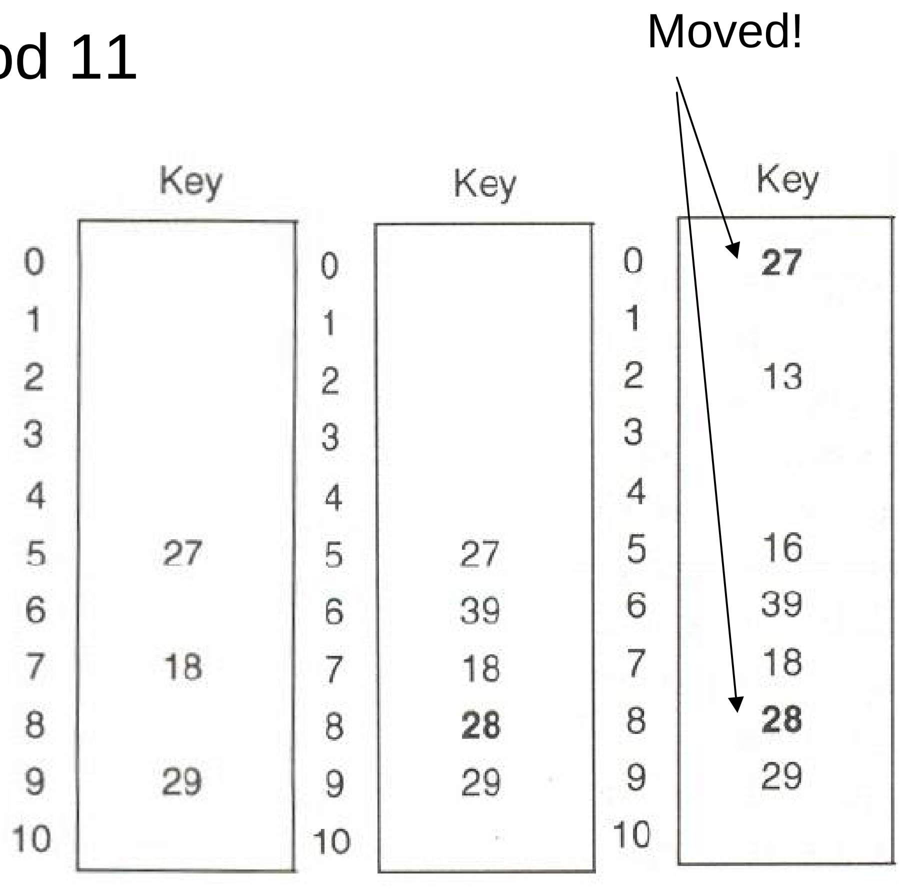
	Key		Key		Key
0		0		0	**27**
1		1		1	
2		2		2	13
3		3		3	
4		4		4	
5	27	5	27	5	16
6		6	39	6	39
7	18	7	18	7	18
8		8	**28**	8	**28**
9	29	9	29	9	29
10		10		10	

Brent’s Method: An Example

- Keys: 27, 18, 29, 28, 39, 13, 16
- $H_1(\text{key}) = \text{key mod } 11$
- $H_2 = \text{Quotient (key / 11) mod } 11$

Average number of probes is 1.7

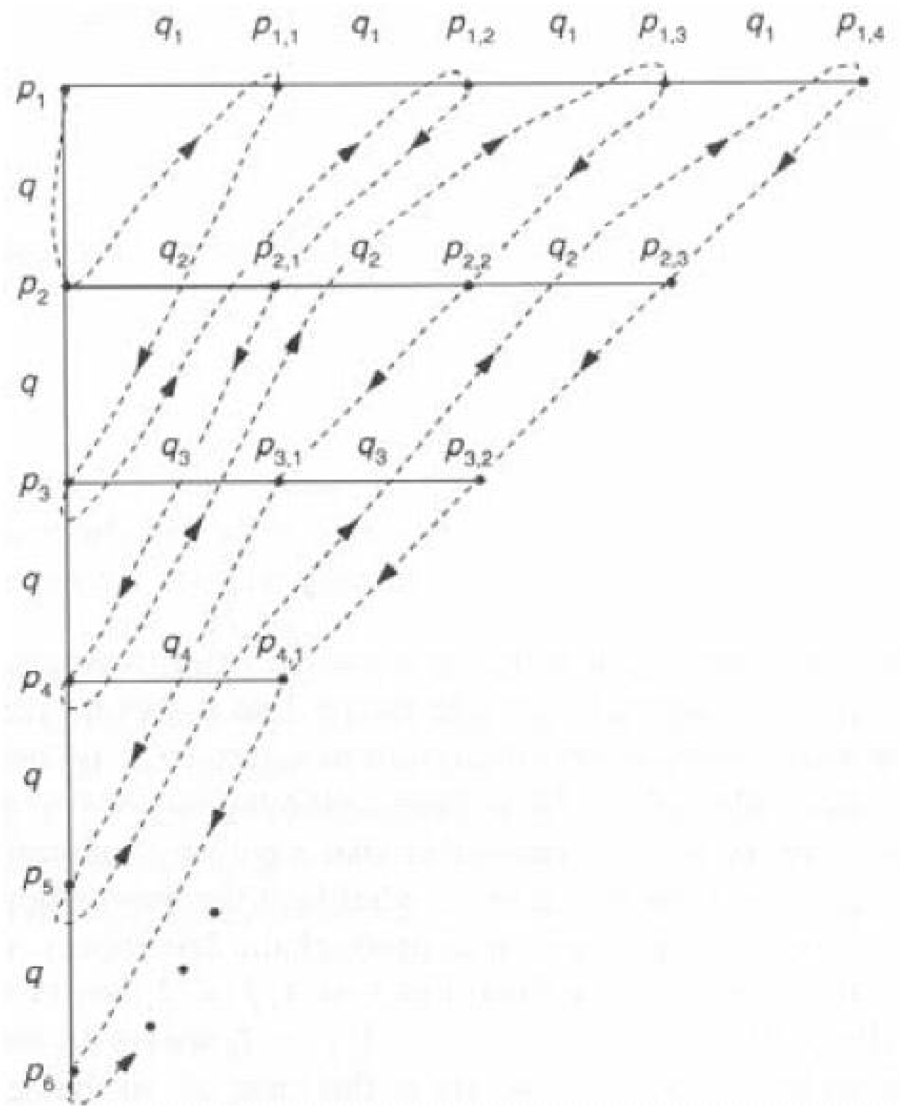
It was 1.9 for LQ, 2.3 for PO and 1.4 for LISCH



	Key		Key		Key
0		0		0	**27**
1		1		1	
2		2		2	13
3		3		3	
4		4		4	
5	27	5	27	5	16
6		6	39	6	39
7	18	7	18	7	18
8		8	**28**	8	**28**
9	29	9	29	9	29
10		10		10	

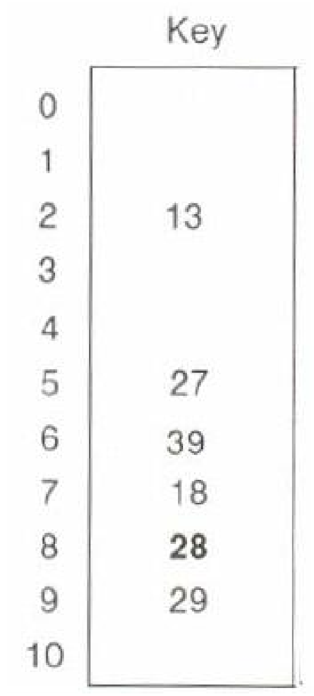
Brent's Method…Optimizing with a road map

- The insertion process can be modified to eliminate the need to compute s prior to trying the various move possibilities
- In doing so we also eliminate the need to traverse the primary probe chain when computing s
- Eliminating this traversal improves the insertion performance

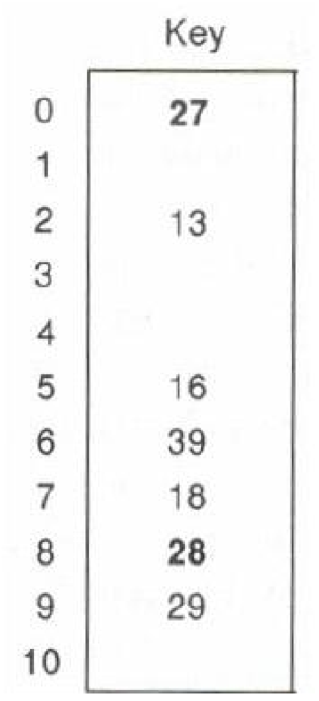


Brent’s Method…Optimizing with a road map

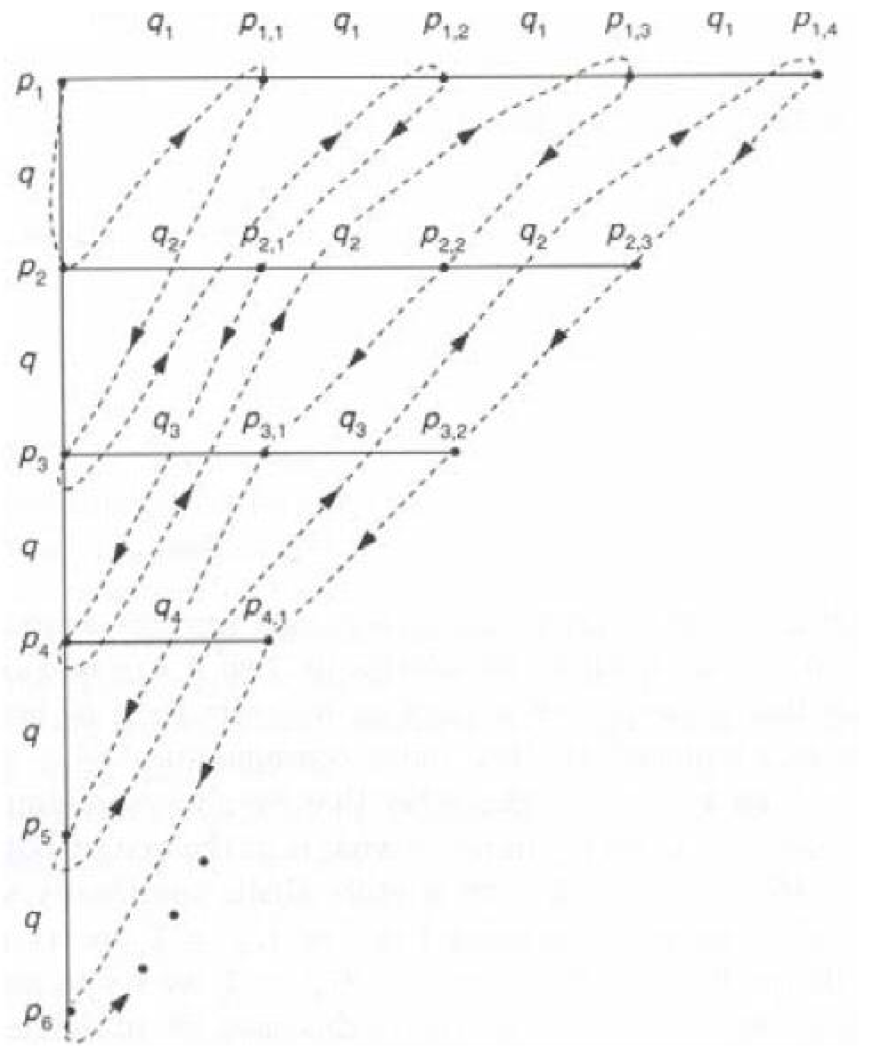
- If we insert 16



	Key
0	
1	
2	13
3	
4	
5	27
6	39
7	18
8	**28**
9	29
10	



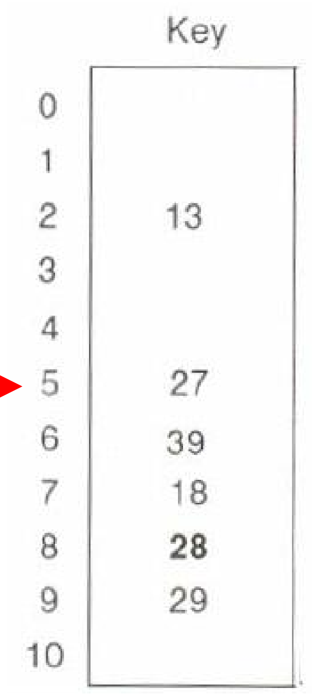
	Key
0	**27**
1	
2	13
3	
4	
5	16
6	39
7	18
8	**28**
9	29
10	



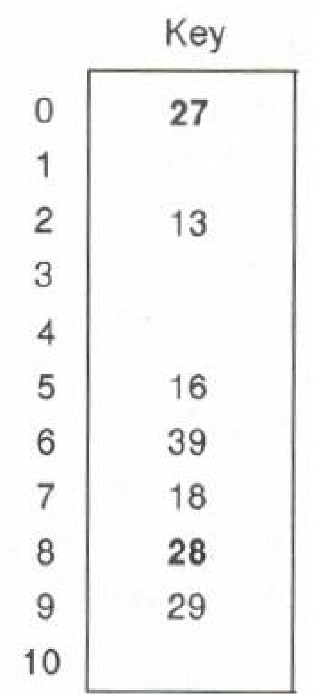
Brent’s Method…Optimizing with a road map

- If we insert 16

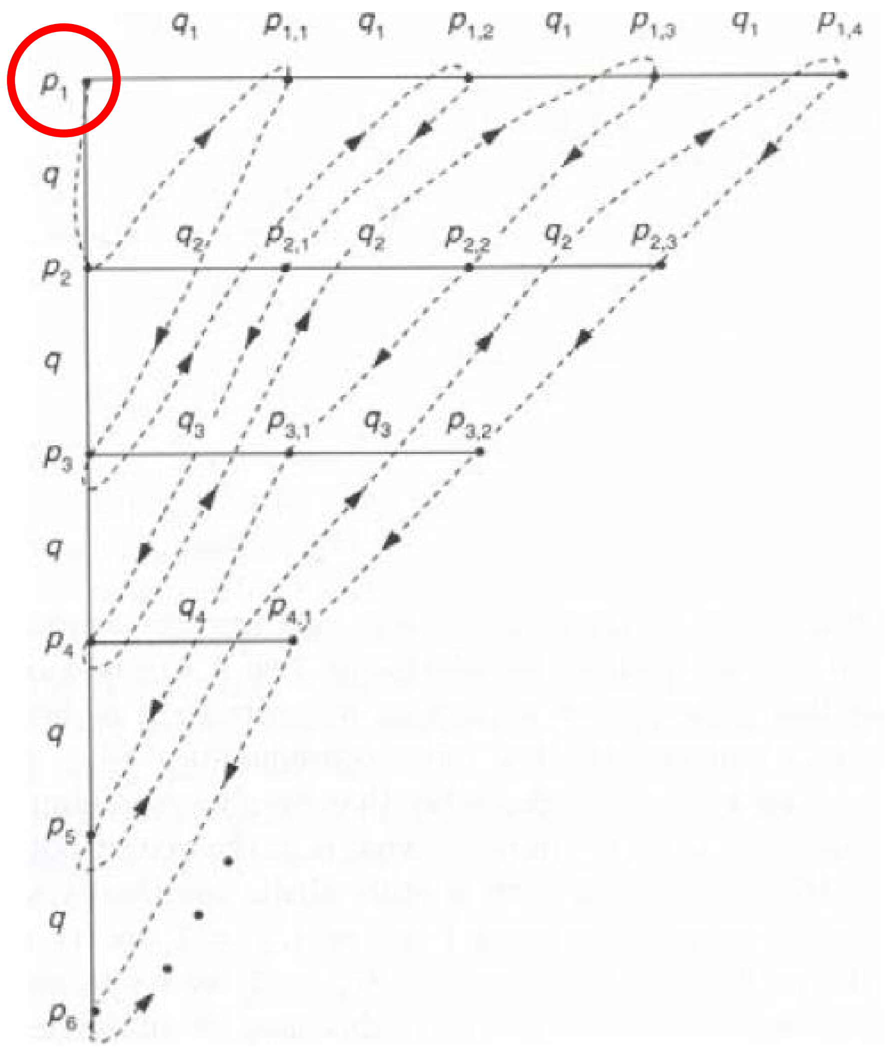
Hash(16)=16 mod 11 = 5



	Key
0	
1	
2	13
3	
4	
5	27
6	39
7	18
8	**28**
9	29
10	



	Key
0	**27**
1	
2	13
3	
4	
5	16
6	39
7	18
8	**28**
9	29
10	



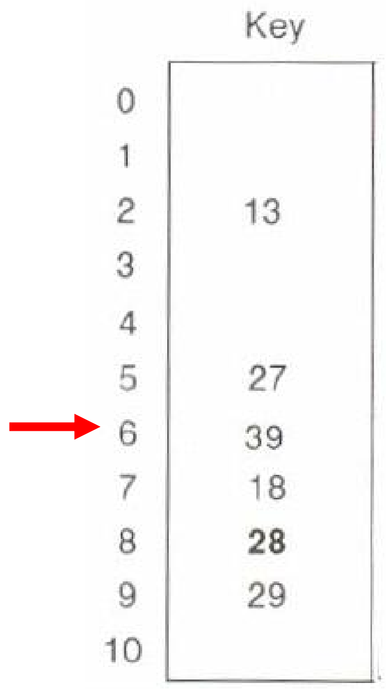
Brent’s Method…Optimizing with a road map

- If we insert 16

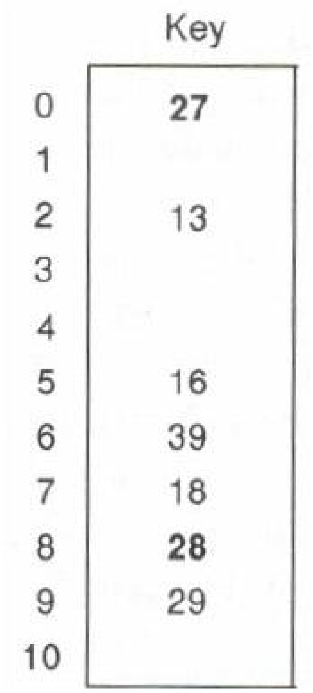
Hash(16)=16 mod 11 = 5

i(16)=Q(16/11) mod 11 = 1

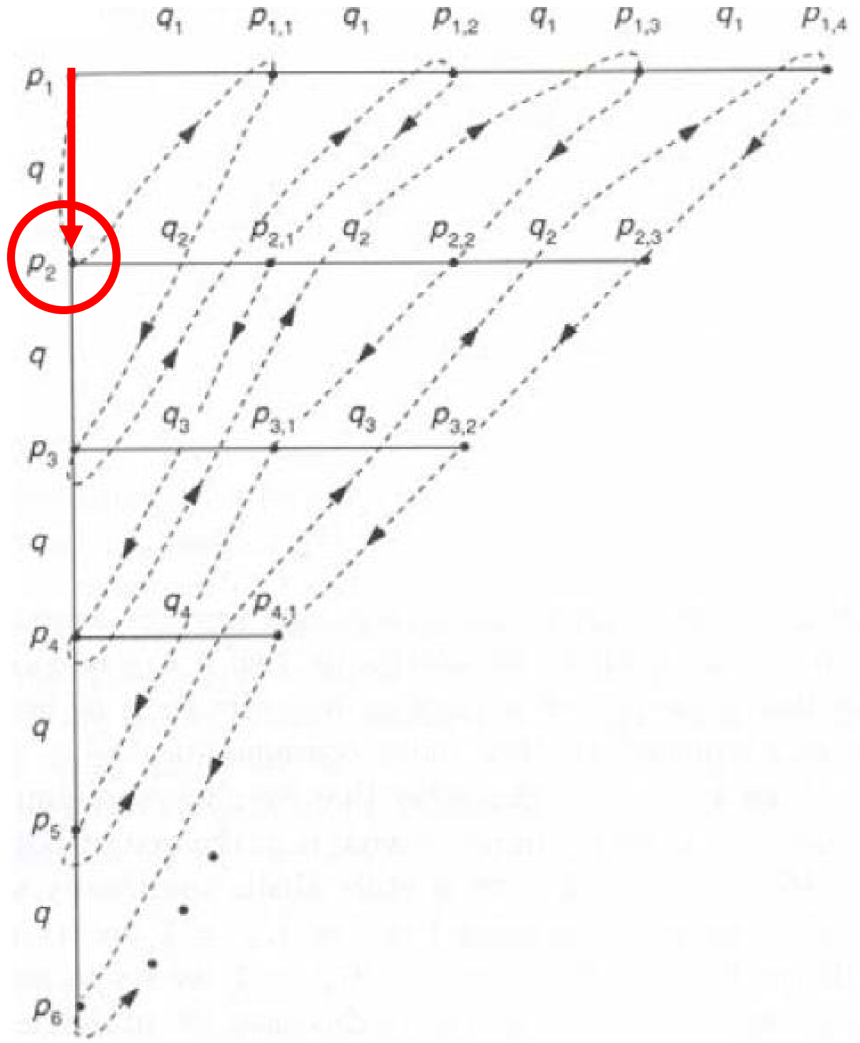
5+1 = 6



	Key
0	
1	
2	13
3	
4	
5	27
6	39
7	18
8	**28**
9	29
10	



	Key
0	**27**
1	
2	13
3	
4	
5	16
6	39
7	18
8	**28**
9	29
10	



q_1 $p_{1,1}$ q_1 $p_{1,2}$ q_1 $p_{1,3}$ q_1 $p_{1,4}$
p_1
q
p_2 q_2 $p_{2,1}$ q_2 $p_{2,2}$ q_2 $p_{2,3}$
q
p_3 q_3 $p_{3,1}$ q_3 $p_{3,2}$
q
p_4 q_4 $p_{4,1}$
q
p_5
q
p_6

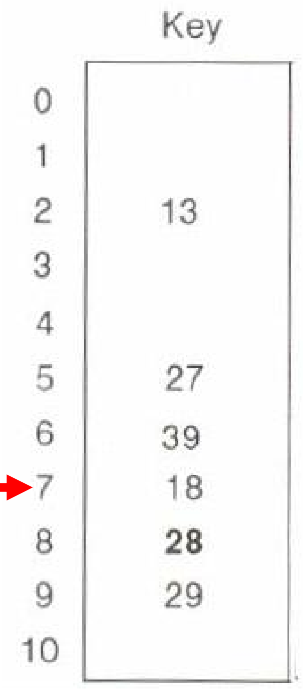
Brent's Method…Optimizing with a road map

- If we insert 16

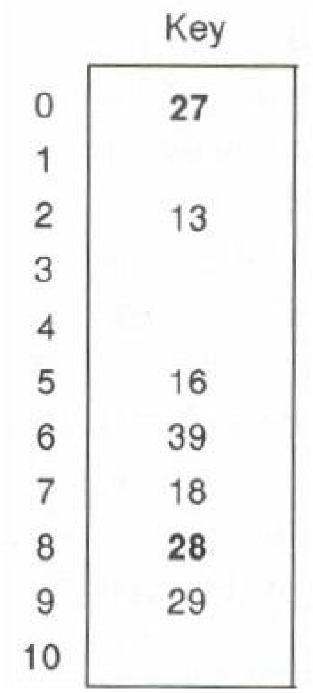
Try to move 27

i(27)=Q(27/11) mod 11 = 2

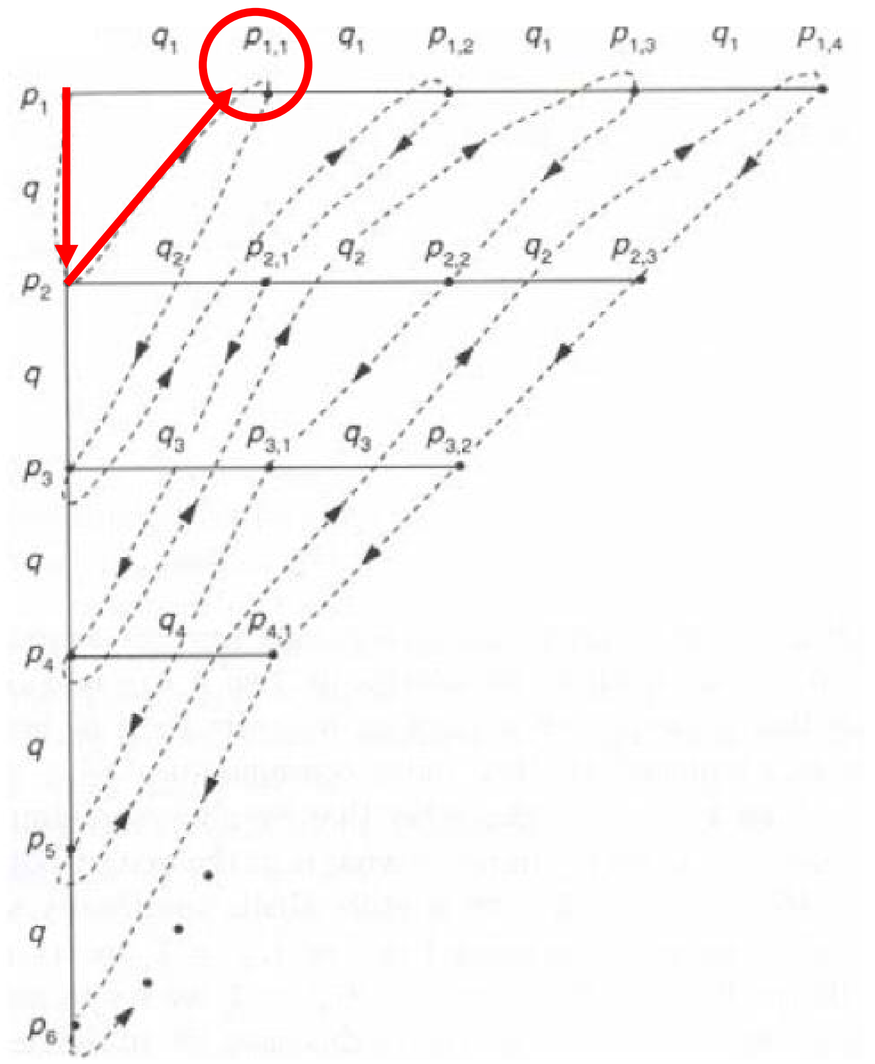
5+2 = 7



	Key
0	
1	
2	13
3	
4	
5	27
6	39
7	18
8	**28**
9	29
10	



	Key
0	**27**
1	
2	13
3	
4	
5	16
6	39
7	18
8	**28**
9	29
10	



q_1 $p_{1,1}$ q_1 $p_{1,2}$ q_1 $p_{1,3}$ q_1 $p_{1,4}$
p_1
q
q_2 $p_{2,1}$ q_2 $p_{2,2}$ q_2 $p_{2,3}$
p_2
q
q_3 $p_{3,1}$ q_3 $p_{3,2}$
p_3
q
q_4 $p_{4,1}$
p_4
q
p_5
q
p_6

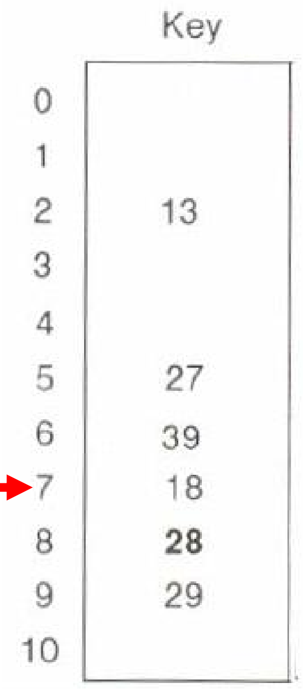
Brent’s Method…Optimizing with a road map

- If we insert 16

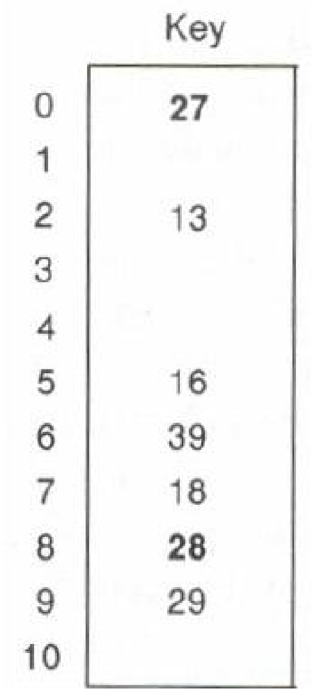
Hash(16)=16 mod 11 = 5

i(16)=Q(16/11) mod 11 = 1

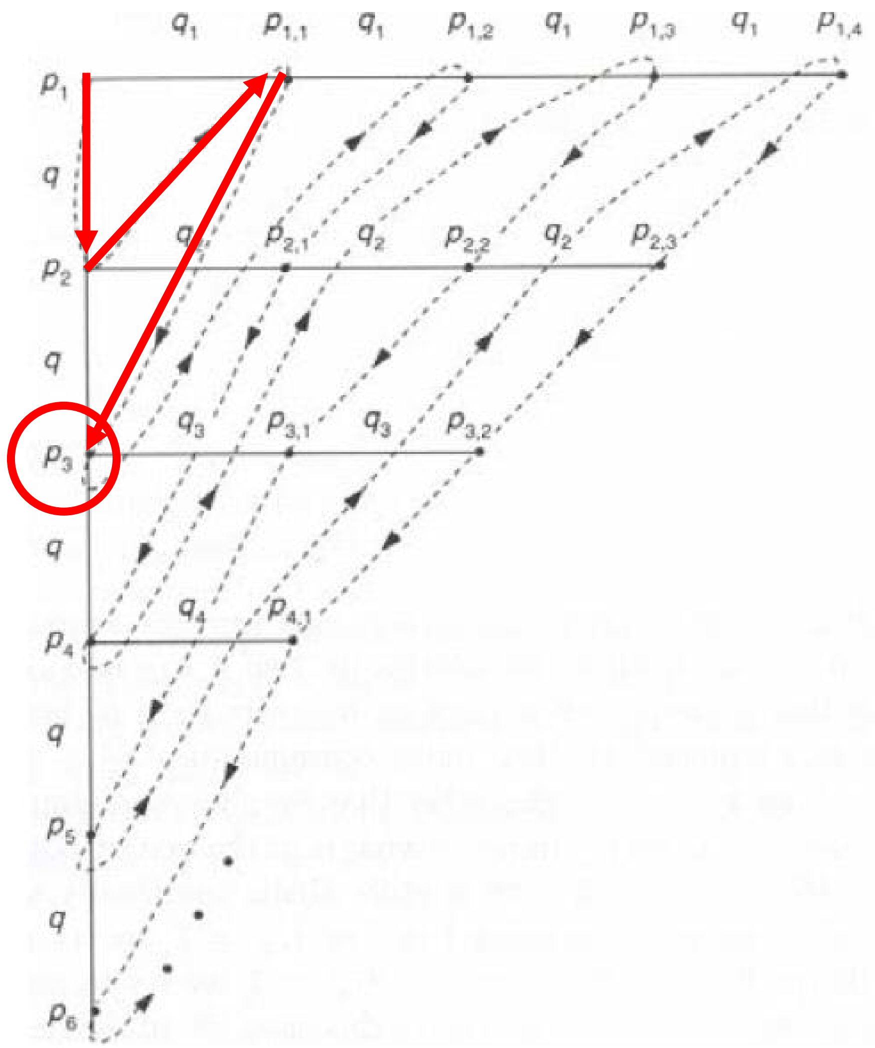
5+1+1 = 7



	Key
0	
1	
2	13
3	
4	
5	27
6	39
7	18
8	**28**
9	29
10	



	Key
0	**27**
1	
2	13
3	
4	
5	16
6	39
7	18
8	**28**
9	29
10	



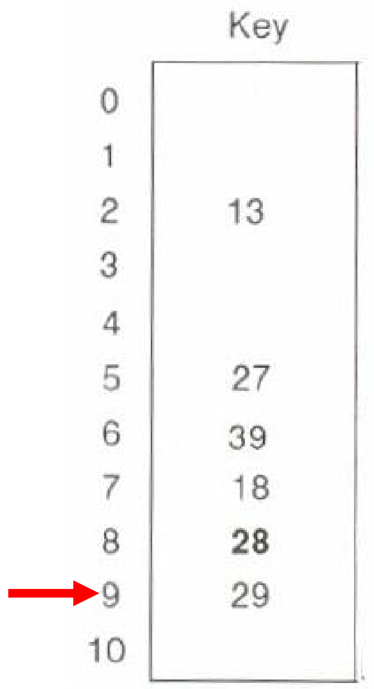
Brent’s Method…Optimizing with a road map

- If we insert 16

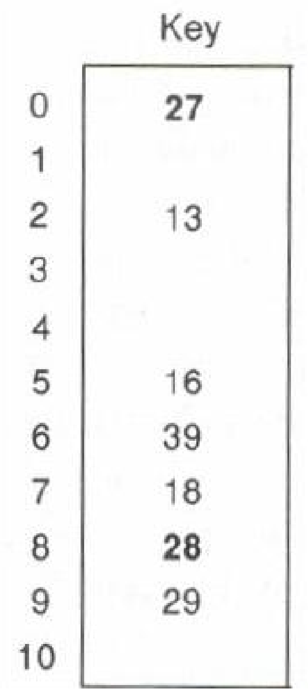
Try to move 27

i(27)=Q(27/11) mod 11 = 2

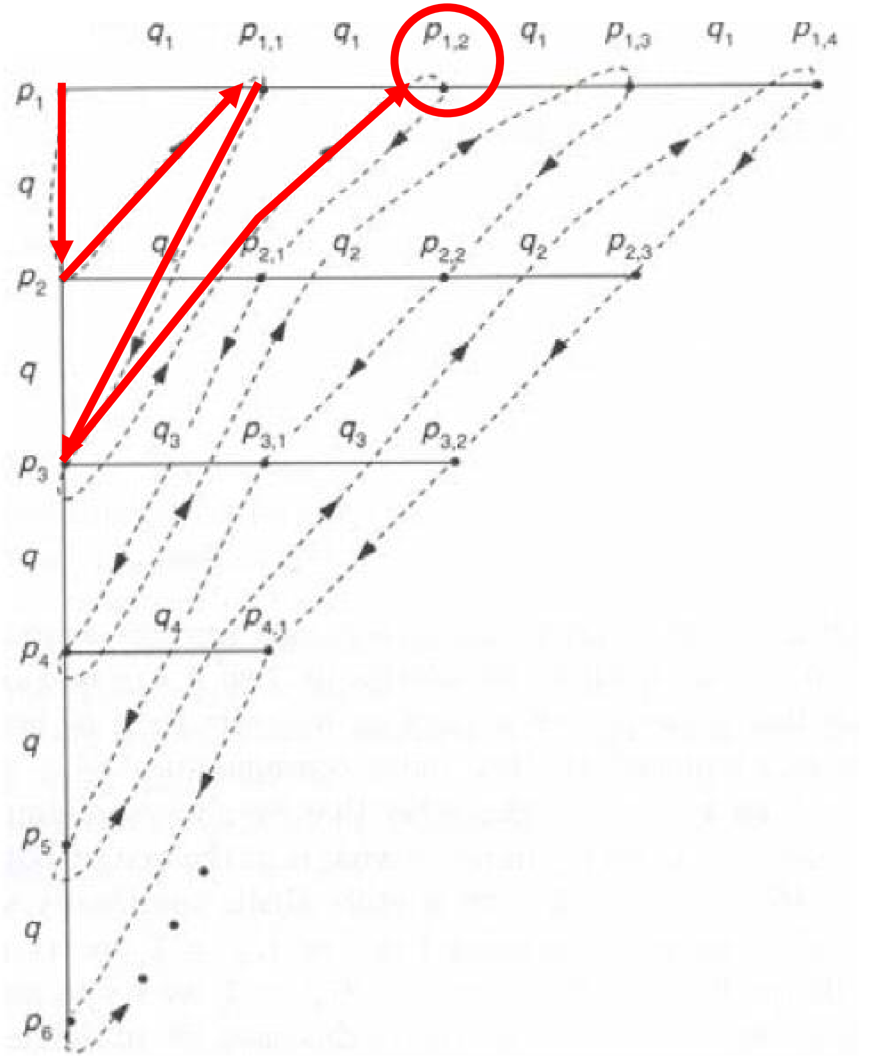
5+2+2 = 9



	Key
0	
1	
2	13
3	
4	
5	27
6	39
7	18
8	**28**
9	29
10	



	Key
0	**27**
1	
2	13
3	
4	
5	16
6	39
7	18
8	**28**
9	29
10	



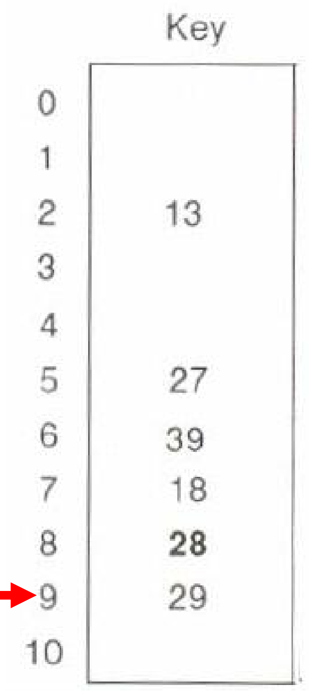
Brent's Method…Optimizing with a road map

- If we insert 16

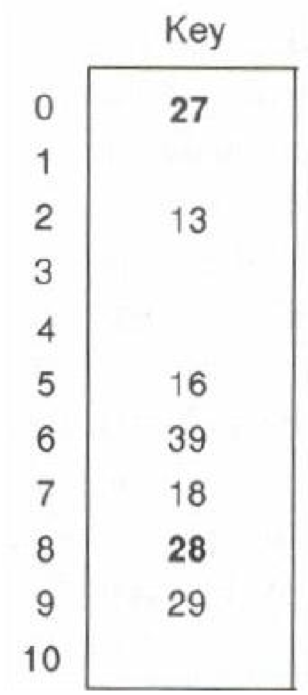
Try to move 39

i(39)=Q(39/11) mod 11 = 3

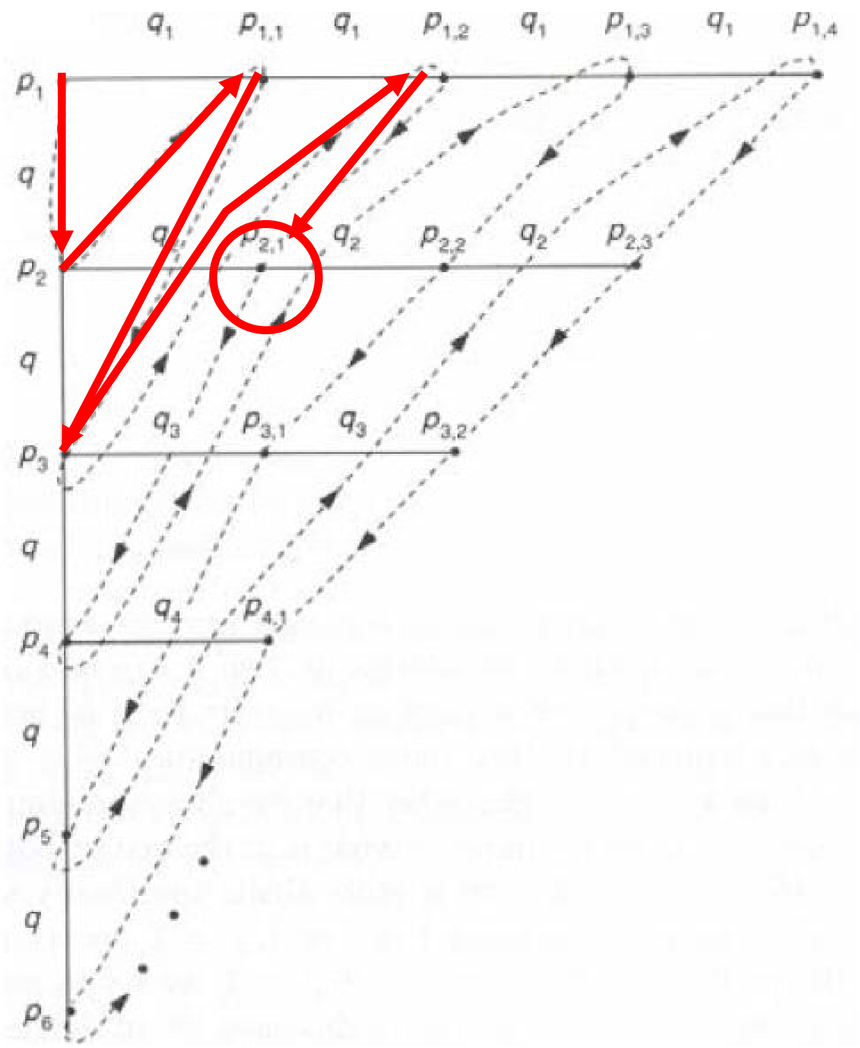
6+3 = 9



	Key
0	
1	
2	13
3	
4	
5	27
6	39
7	18
8	**28**
9	29
10	



	Key
0	**27**
1	
2	13
3	
4	
5	16
6	39
7	18
8	**28**
9	29
10	



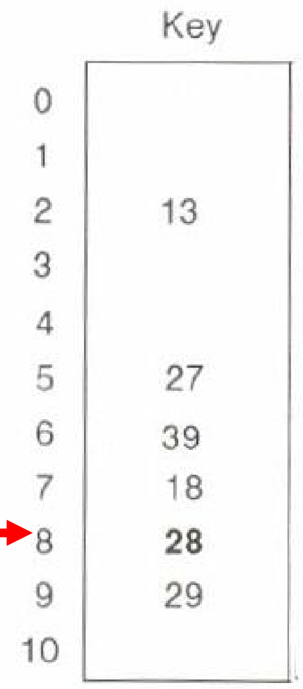
Brent's Method…Optimizing with a road map

- If we insert 16

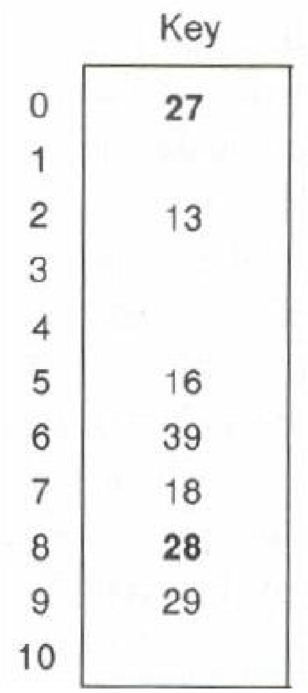
Hash(16)=16 mod 11 = 5

i(16)=Q(16/11) mod 11 = 1

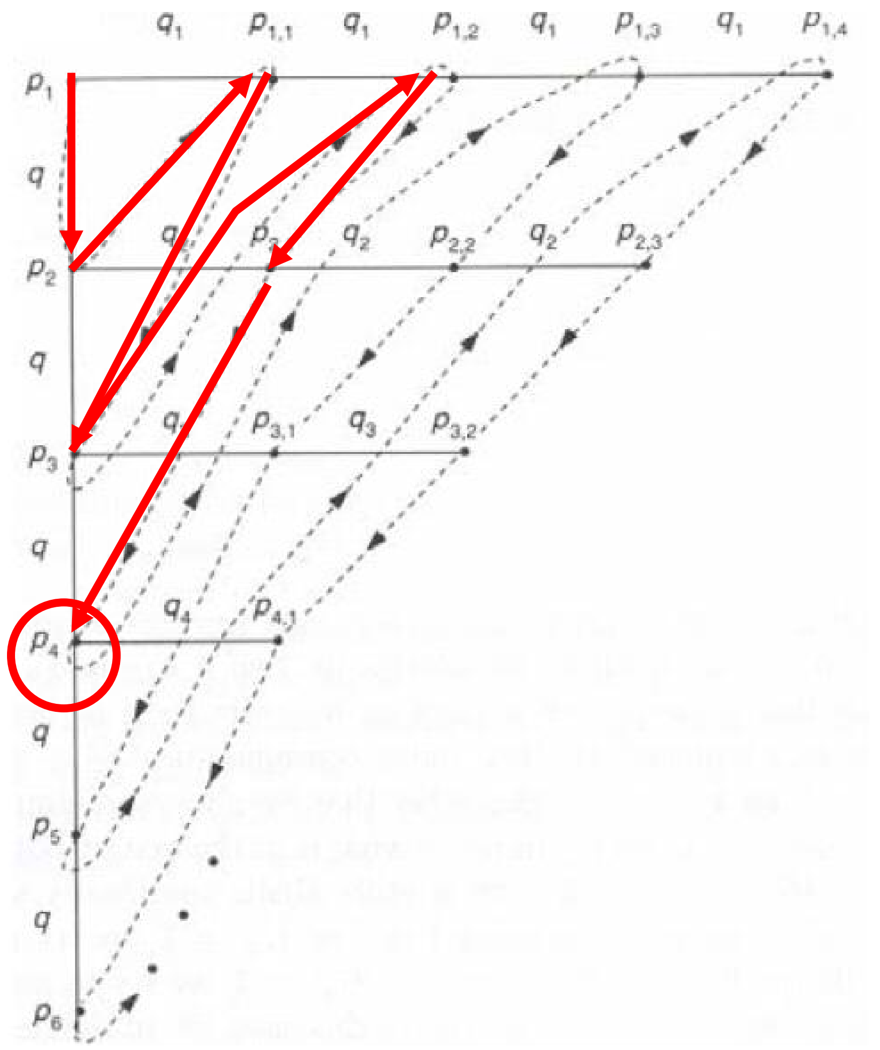
5+1+1+1= 8



	Key
0	
1	
2	13
3	
4	
5	27
6	39
7	18
8	**28**
9	29
10	



	Key
0	**27**
1	
2	13
3	
4	
5	16
6	39
7	18
8	**28**
9	29
10	



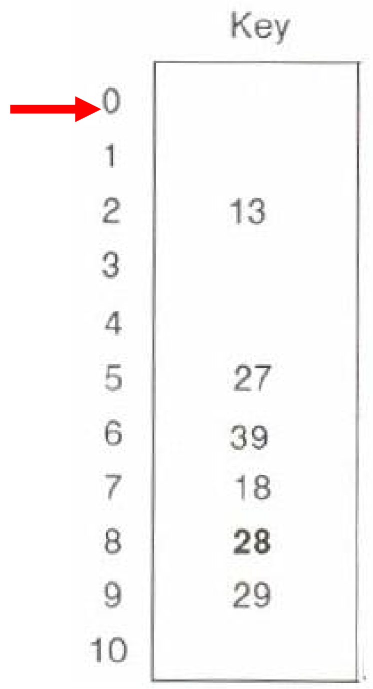
Brent's Method…Optimizing with a road map

- If we insert 16

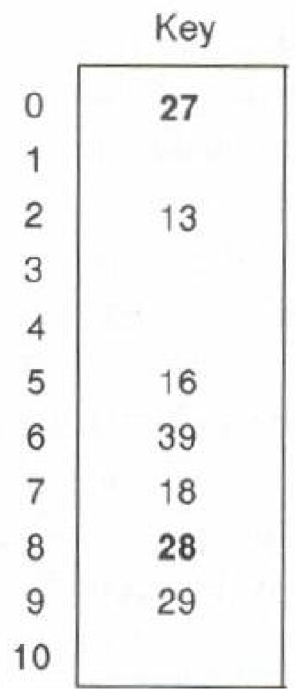
Try to move 27

i(27)=Q(27/11) mod 11 = 2

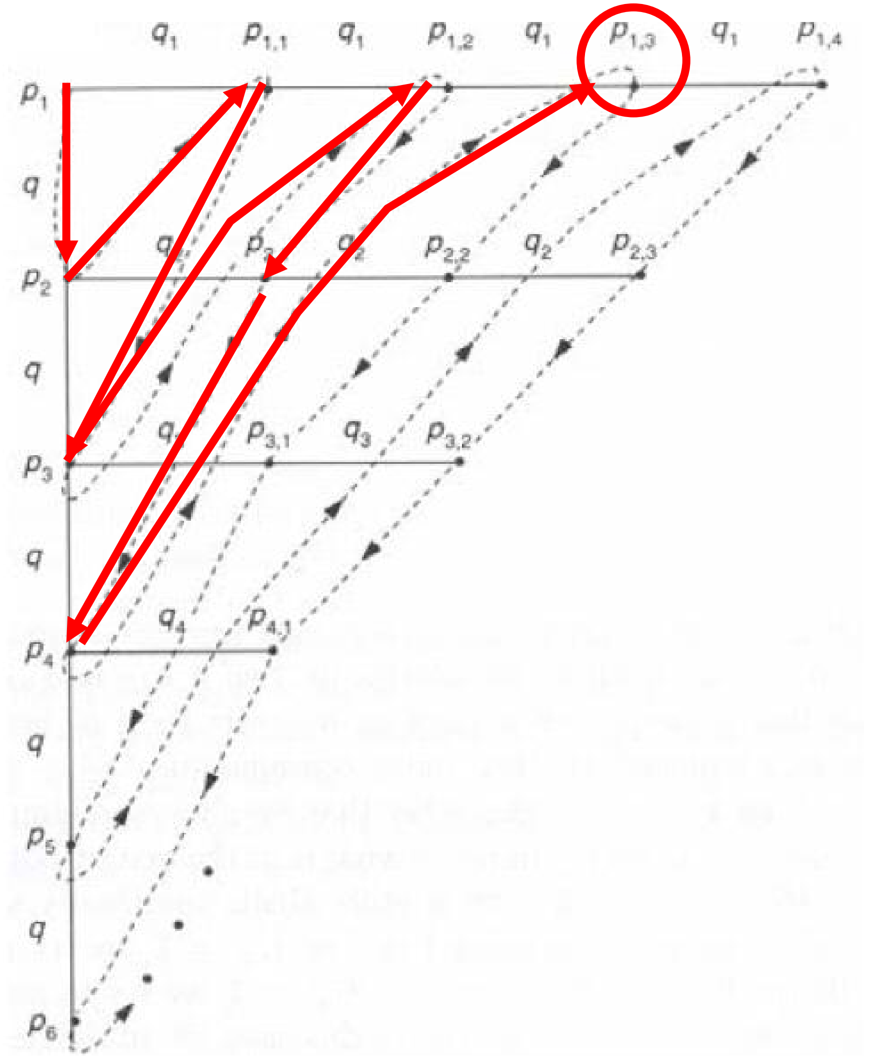
(5+2+2+2) mod 11 = 0



	Key
0	
1	
2	13
3	
4	
5	27
6	39
7	18
8	**28**
9	29
10	



	Key
0	**27**
1	
2	13
3	
4	
5	16
6	39
7	18
8	**28**
9	29
10	



Brent’s Method : Discussion

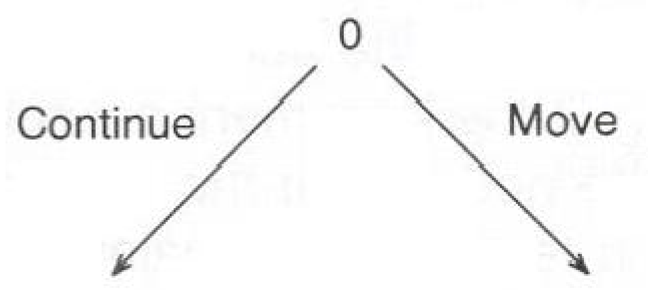
- Avoiding traversal at each insertion improves insertion performance
- Note that the Brent’s Method is only used for insertion
- Because linear quotient is used for retrieval with Brent’s method, deleting a record would require the placement of a tombstone at the position of the deleted record

???

- If it is a good idea to move an item on a primary probe chain, why not carry this concept one step further and move items from secondary and subsequent probe chains?
- Too complex?

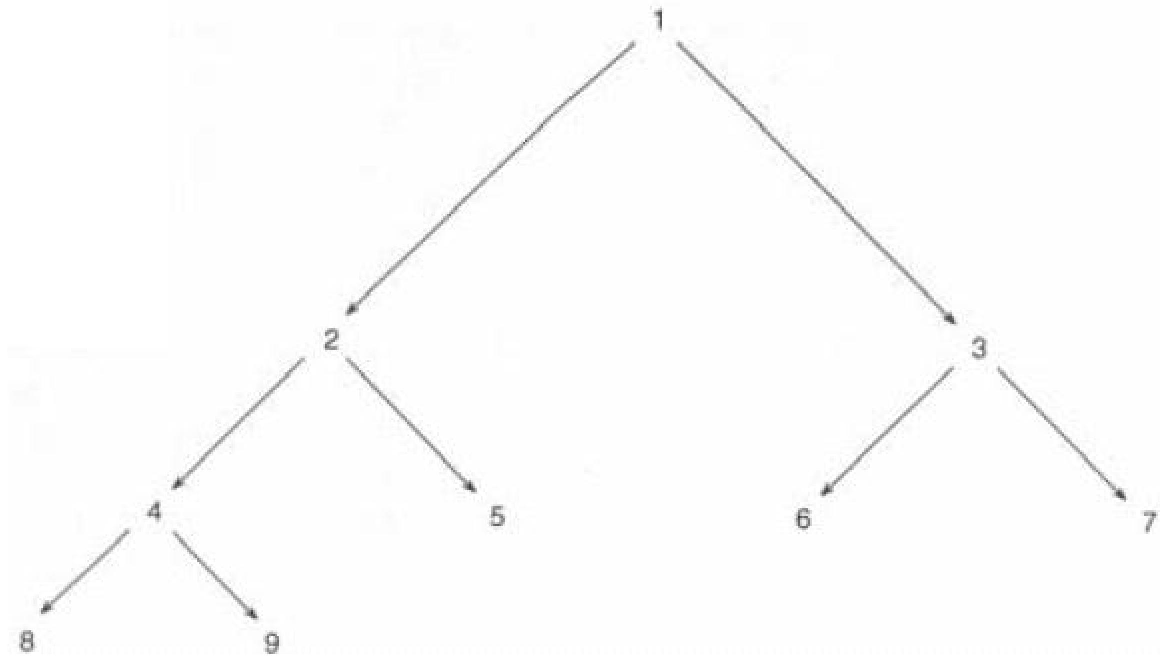
Binary Tree

- A binary tree structure can be used to determine when to move an item and where to move it
- A binary tree is appropriate since there are essentially two choices at each probable storage address
  - *Continue* to the next address along the probe chain of the item being inserted, or
  - *Move* the item stored at that address to the next position on *its* probe chain



Binary Tree ... 2

- The binary decision tree is generated in a breadth first fashion from the top down left to right as shown



Binary Tree ... 3

- The binary tree is used only as a control mechanism in deciding where to store an item and is not used for storing records
- A different binary tree is constructed for each insertion of a record
- Encountering either an empty leaf node in the binary tree or a full table terminates the process
- More preprocessing is done on the file with the binary tree method than with Brent's method
- The special processing is only for insertion of items; retrieval is handled through the LQ method.

Binary Tree : An Example

- Keys: 27, 18, 29, 28, 39, 13, 16, 41, 17, 19
- H_1(key) = key mod 11
- H_2 = Quotient (key / 11) mod 11

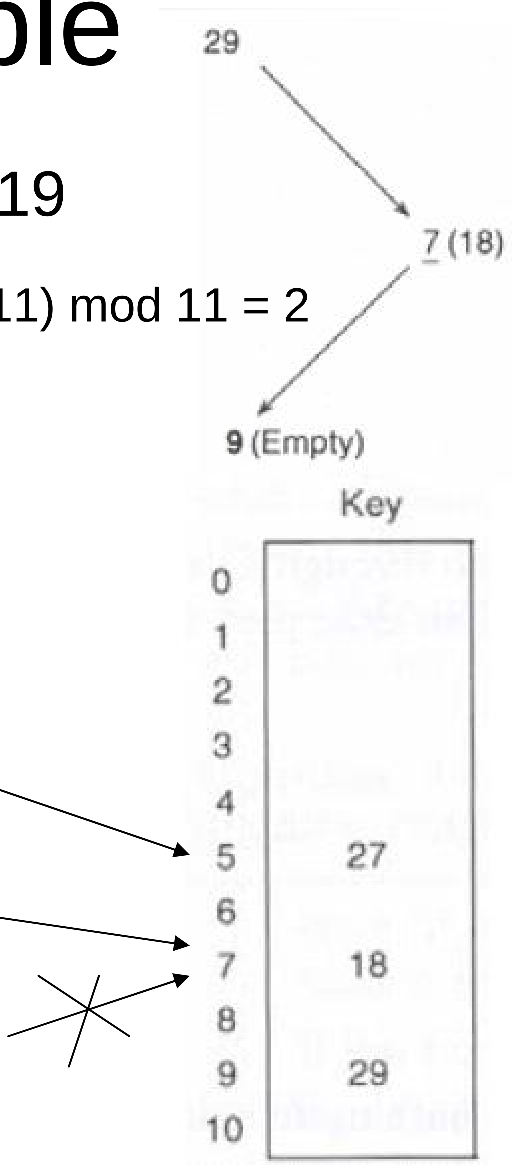
- Inserting 27

  $H_1(27) = 27 \bmod 11 = 5$
- Inserting 18

  $H_1(18) = 18 \bmod 11 = 7$
- Inserting 29

  $H_1(29) = 29 \bmod 11 = 7$ (COLLISION!)

$H_2=Q(29/11) \bmod 11 = 2$



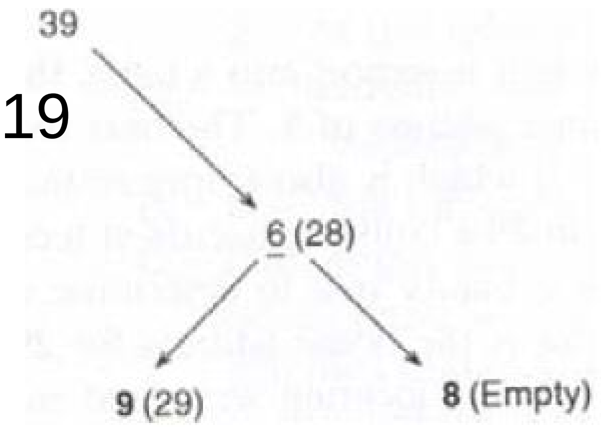
Binary Tree : An Example

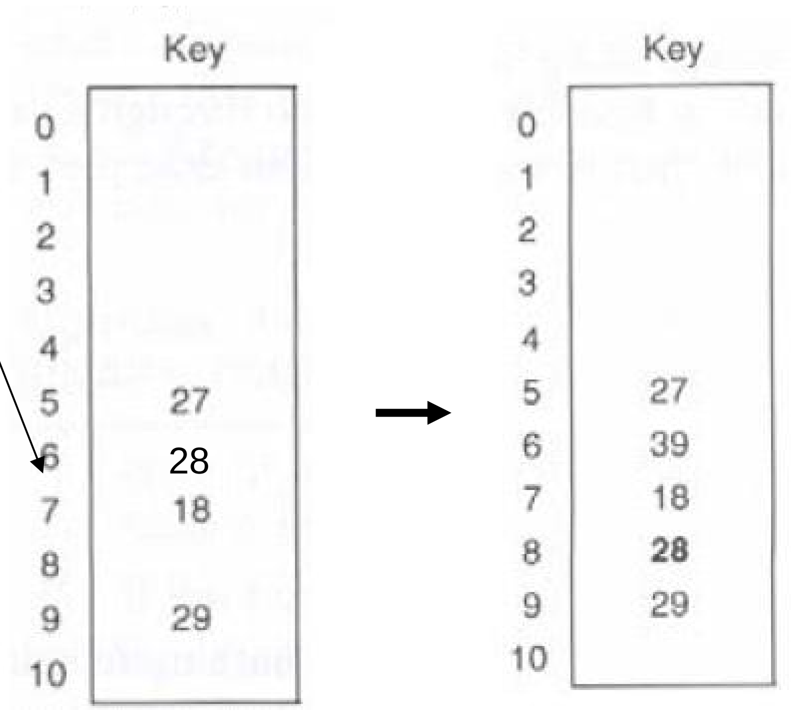
- Keys: 27, 18, 29, 28, 39, 13, 16, 41, 17, 19
- H_1(key) = key mod 11
- H_2 = Quotient (key / 11) mod 11
- Inserting 28

  $H_1(28) = 28 \bmod 11 = 6$
- Inserting 39

  $H_1(39) = 39 \bmod 11 = 6$

(COLLISION!)





	Key
0	
1	
2	
3	
4	
5	27
6	28
7	18
8	
9	29
10	

	Key
0	
1	
2	
3	
4	
5	27
6	39
7	18
8	**28**
9	29
10	

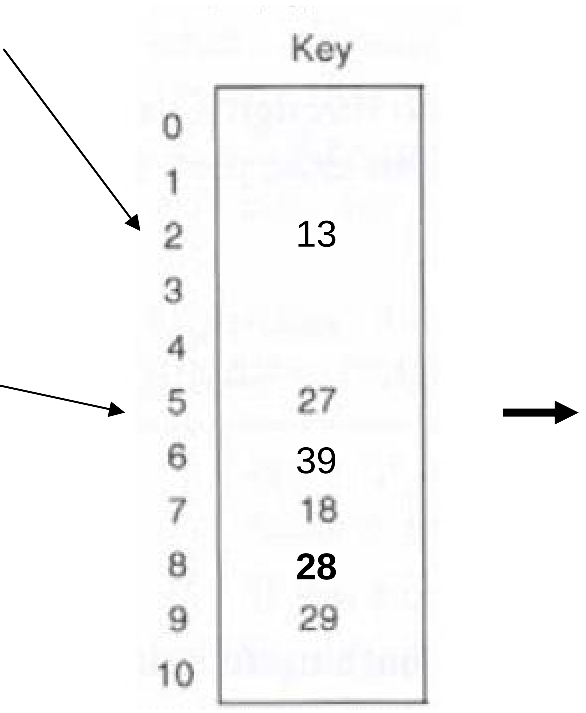
Binary Tree : An Example

- Keys: 27, 18, 29, 28, 39, 13, 16, 41, 17, 19
- $H_1(\text{key}) = \text{key} \bmod 11$
- $H_2 = \text{Quotient}(\text{key} / 11) \bmod 11$
- Inserting 13

  $H_1(13) = 13 \bmod 11 = 2$
- Inserting 16

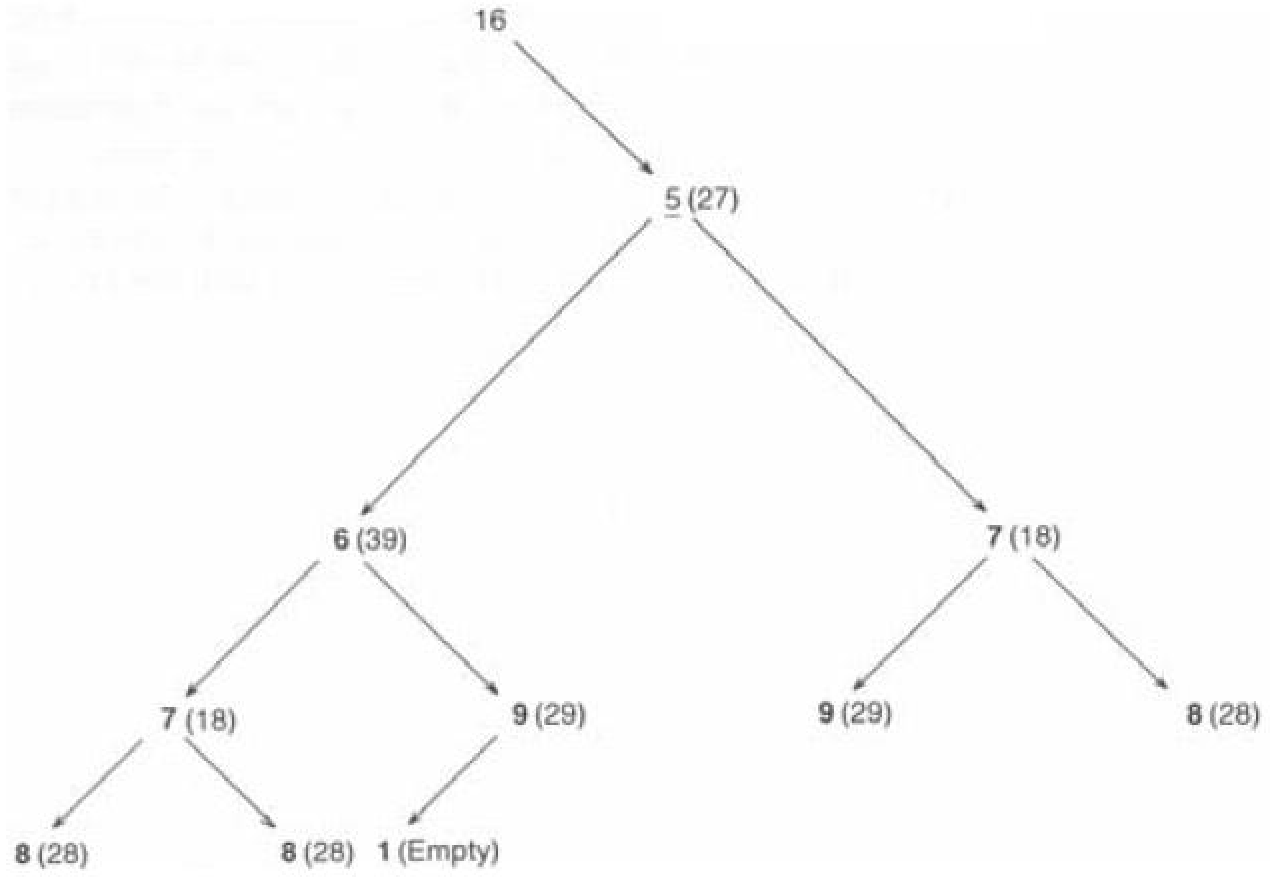
  $H_1(16) = 16 \bmod 11 = 5$

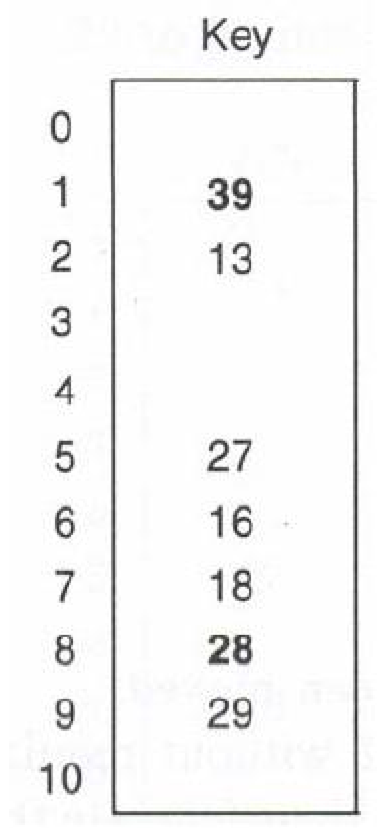
(COLLISION!)



Binary Tree : An Example

- Keys: 27, 18, 29, 28, 39, 13, 16, 41, 17, 19





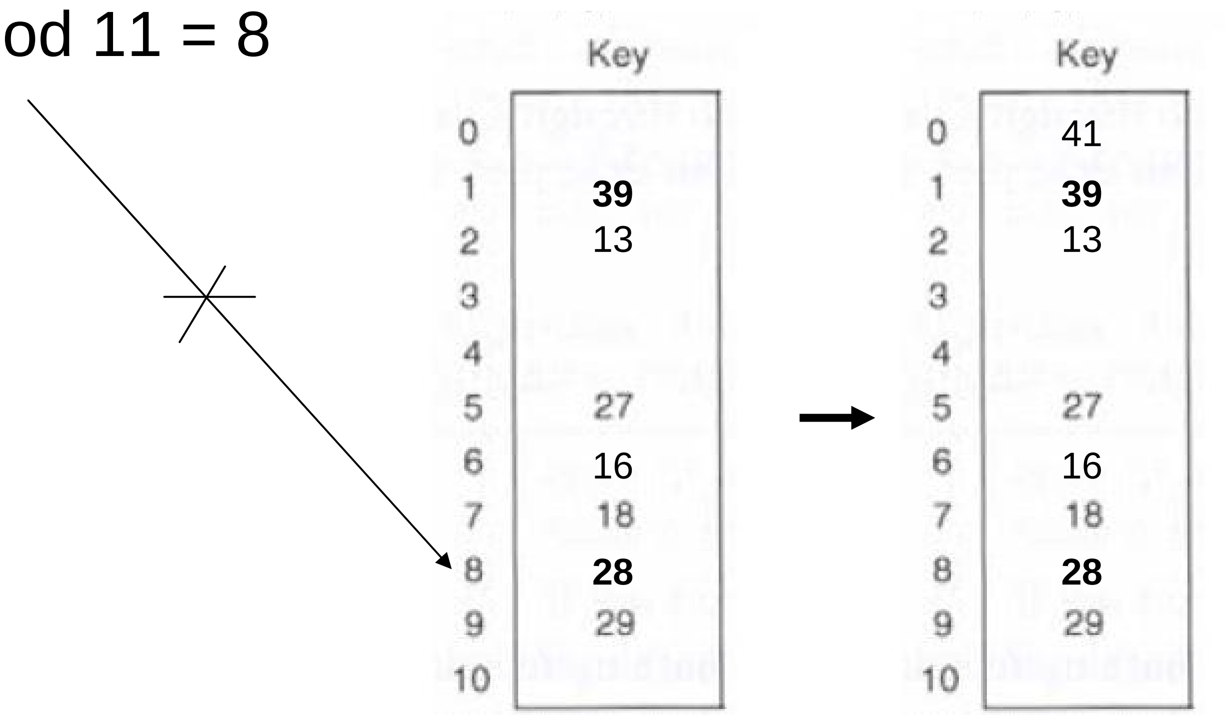
	Key
0	
1	**39**
2	13
3	
4	
5	27
6	16
7	18
8	**28**
9	29
10	

Binary Tree : An Example

- Keys: 27, 18, 29, 28, 39, 13, 16, 41, 17, 19
- $H_1(\text{key})$ = key mod 11
- H_2 = Quotient (key / 11) mod 11
- Inserting 41

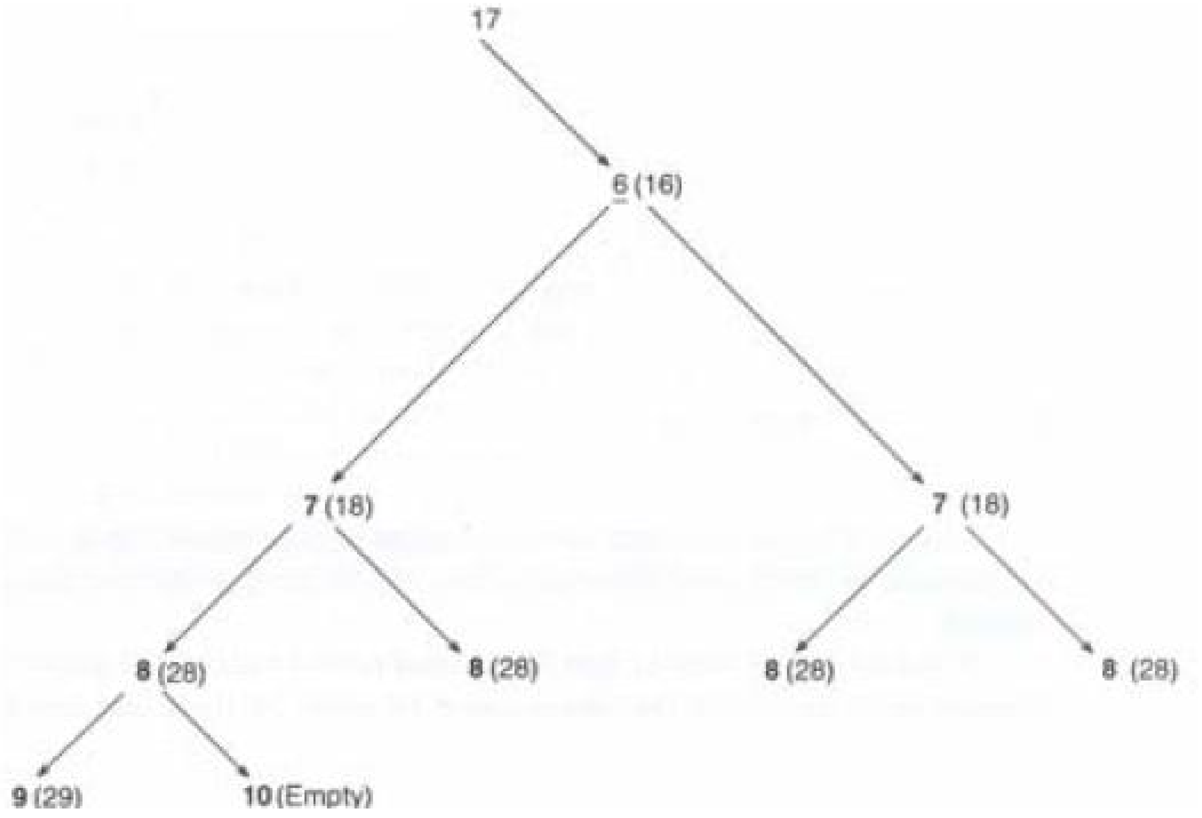
  $H_1(41)$ = 41 mod 11 = 8

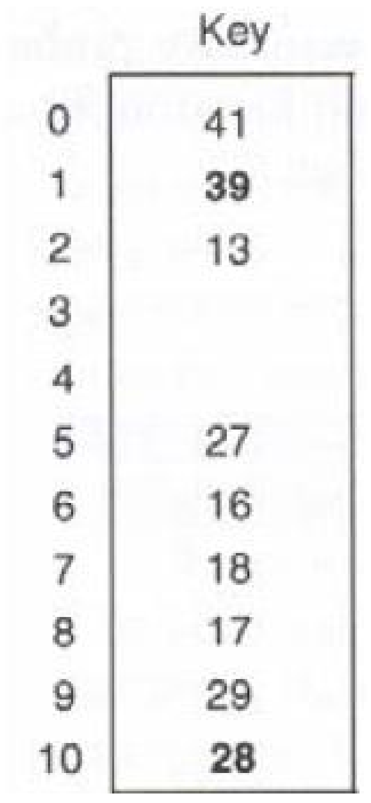
(COLLISION!)



Binary Tree : An Example

- Keys: 27, 18, 29, 28, 39, 13, 16, 41, 17, 19

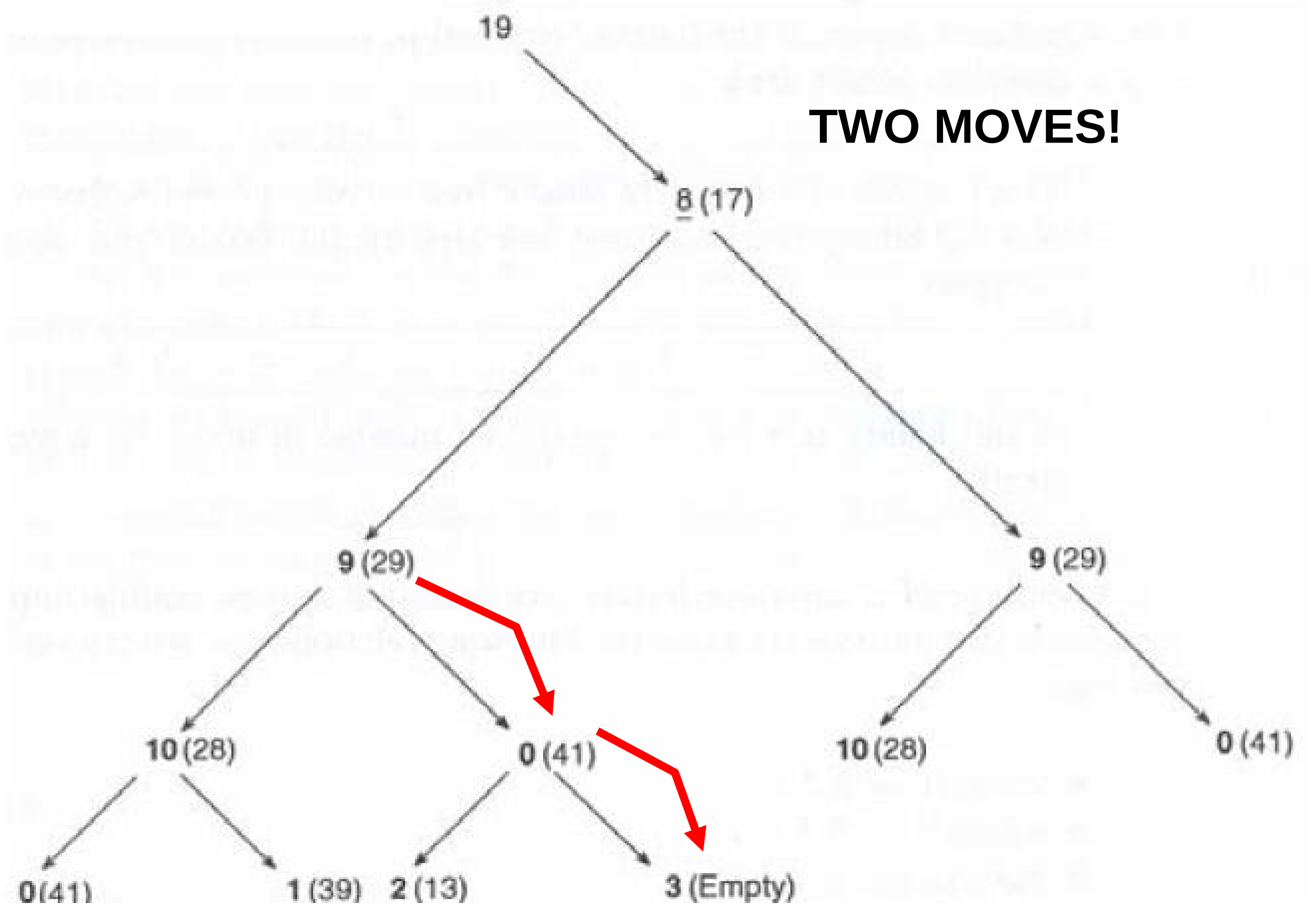




	Key
0	41
1	39
2	13
3	
4	
5	27
6	16
7	18
8	17
9	29
10	28

Binary Tree : An Example

- Keys: 27, 18, 29, 28, 39, 13, 16, 41, 17, 19



	Key
0	29
1	39
2	13
3	41
4	
5	27
6	16
7	18
8	17
9	19
10	28