Collision Resolution

- The hashing we've looked at so far does have problems with multiple keys hashing to the same location in the table
- For example, consider a function that places names in a table based on hashing the ASCII code of the first letter of the name
- Using this function, all names beginning with the same letter would hash to the same position
- If we attempt to improve the function by hashing the first two letters, we achieve better results, but still have problems
- In fact, even if we used all the letters in the name, there is still a possibility of collisions
- Also, while using all the letters of the name gives a better distribution, if the table only has 26 positions there is no improvement in using the other versions

Collision Resolution (continued)

- So in addition to using more efficient functions, we also need to consider the size of the table being hashed into
- Even then, we cannot guarantee to eliminate collisions; we have to consider approaches that assure a solution
- A number of methods have been developed; we will consider a few in the following slides

Collision Resolution (continued)

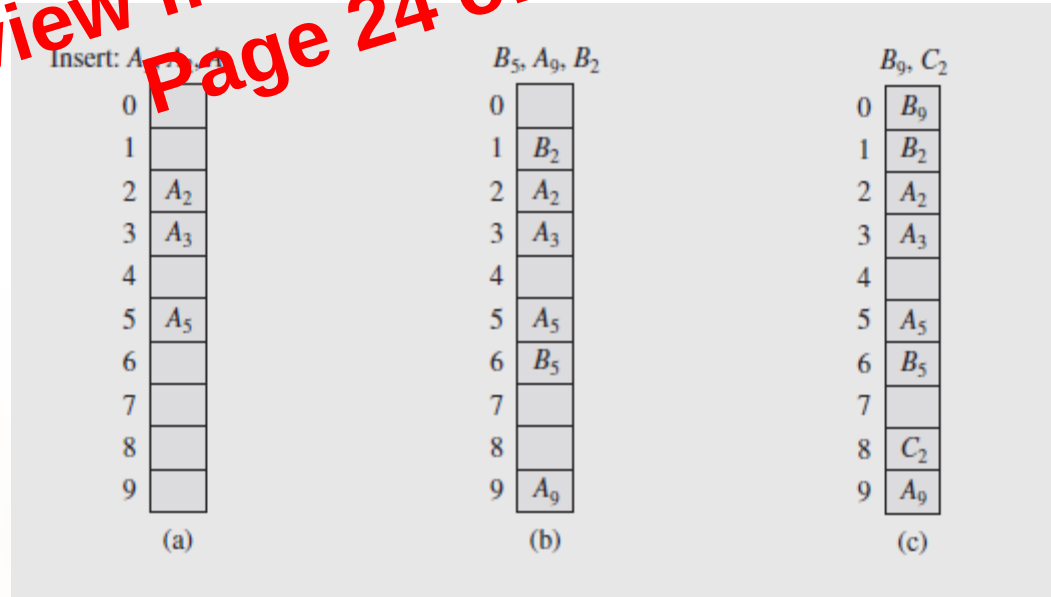

Fig. 10.2 Using quadratic probing for collision resolution

Collision Resolution (continued)

- Chaining
  - In ***chaining***, the keys are not stored in the table, but in the `info` portion of a linked list of nodes associated with each table position
  - This technique is called ***separate chaining***, and the table is called a ***scatter table***
  - This was the table never overflows, as the lists are extended when new keys arrive, as can be seen in Figure 10.5
  - This is very fast for short lists, but as they increase in size, performance can degrade sharply
  - Gains in performance can be made if the lists are ordered so unsuccessful searches don't traverse the entire list, or by using self-organizing linked lists
  - This approach requires additional space for the pointers, so if there are a large number of keys involved, space requirements can be high

Collision Resolution (continued)

- Chaining (continued)
  - This way the next element in the list can be accessed with doing a sequential search down the list
  - Each position in the table is an object that consists of two elements, `info` for the key and `next` which is the index of the key that collided with this position
  - Available positions can be marked in their `next` fields, and a unique value can be used to mark the end of a chain
  - This does require less space than chaining, but the number of keys that can be hashed is limited by table size
  - For keys for which there is no room in the table, an overflow area known as the ***cellar*** can be dynamically located in the arrays
  - Figure 10.6 shows an example of coalesced hashing where a key ends up in the last position in the table

Deletion

- How can data be removed from a hash table?
- If chaining is used, the deletion of an element entails deleting the node from the linked list holding the element
- For the other techniques we've considered, deletion usually involves more careful handling of collision issues, unless a perfect hash function is used
- This is illustrated in Figure 10.10a, which stores keys using linear probing
- In Figure 10.10b, when A_4 is deleted, attempts to find B_4 check location 4, which is empty, indicating B_4 is not in the table
- A similar situation occurs in Figure 10.10c, when A_2 is deleted, causing searches for B_1 to stop at position 2

Deletion (continued)

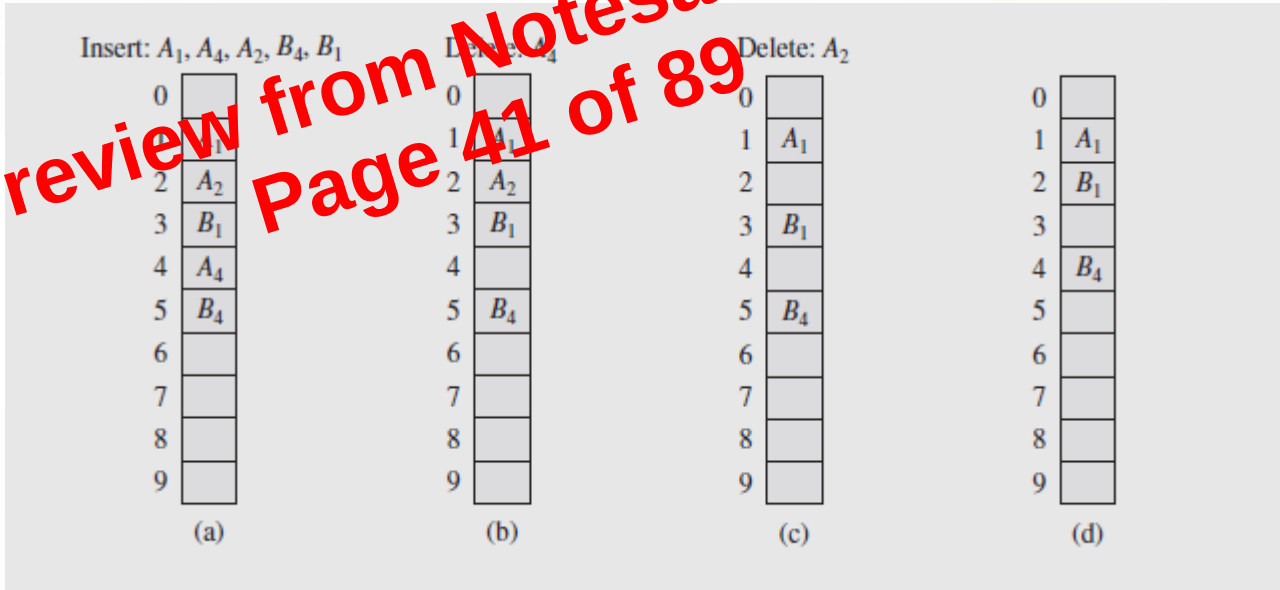

Fig. 10.10 Linear search in the situation where both insertion and deletion of keys are permitted

- A solution to this is to leave the deleted keys in the table with some type of indicator that the keys are not valid
- This way, searches for elements won't terminate prematurely
- When new keys are inserted, they can overwrite the marked keys

Perfect Hash Functions (continued)

- Cichelli's Method
  - Richard J. Cichelli developed one technique for constructing minimal perfect hash functions in 1980
  - It is used to hash fairly small number of reserved words, and has the form

$$h(word) = (length(word) + g(firstletter(word)) + g(lastletter(word))) \bmod TSize$$

  - In the expression, *g* is the function to be constructed; it assigns values to letters to the function *h* returns unique hash values
  - The values that *g* assigns to particular letters do not have to be unique
  - There are three parts to the algorithm: computation of letter occurrences, ordering words, and searching
  - The last step is critical, and uses an auxiliary function, `try()`

Perfect Hash Functions (continued)

- Cichelli's Method (continued)
  - Another modification partitions the body of data into separate buckets for which minimal perfect hash functions are found
  - The partitioning is carried out by a grouping function, *gr*, which indicates the bucket each word belongs to
  - A general hash function of the form

$$h(word) = bucket_{gr(word)} + h_{gr(word)}(word)$$

    is then generated, as suggested by Ted Lewis and Curtis Cook (1986)
  - The drawback to this approach is the difficulty in finding grouping functions that support the creation of minimal perfect hash functions
  - Both of these approaches are not completely successful if they rely on the algorithm developed by Cichelli
  - Although Cichelli advocated brute force as a last resort, most efforts focus on finding more efficient searching algorithms, such as the FHCD

Perfect Hash Functions (continued)

- The FHCD Algorithm (continued)
  - Next, a random $g(h)$ value must be found that will produce two numbers either than 2 when h computes the hash function for the two words, because location 2 is occupied
  - For purposes of the example, assume the number is 4, so h(Calliope) = 1 and h(Melpomene) = 4
  - A summary of these steps is shown in Figure 10.12d; Figure 10.12e shows the values of the function g
  - By means of this function g, the function h becomes a minimal perfect hash function
  - However, g has to be stored as a table so it can be used every time function h is needed, because it is presented as a table and not a formula
  - This may not be straightforward

Hash Functions for Extendible Files (continued)

- There are some hashing techniques that take into account variable sizes of tables or files
- These fall into two groups: directory and directoryless
- In directory schemes, a directory or index of keys controls access to the keys themselves
- There are a number of techniques that fall into this category
  - ***Expandable hashing***, developed by Gary D. Knott in 1971
  - ***Dynamic hashing***, developed by Per-Âke Larson in 1978
  - ***Extendible hashing***, developed by Ronald Fagin and others in 1979
- All of these distribute keys among buckets in similar ways
- The structure of the directory or index is the main difference

Hash Functions for Extendible Files (continued)

- Extendible Hashing (continued)
  - This is illustrated in Figure 10.14b; when a key with *h* value 11001 arrives, the first two bits send it to the fourth directory position
  - From there it goes to bucket b_1, where keys whose *h*-values start with 1 are stored
  - This causes an overflow, splitting bucket b_1 into b_{10}, which is the new name for b_1, and b_{11}; their local depths are set to two
  - Bucket b_{11} is now pointed at by the pointer in position 11, and b_1's keys are split between b_{10} and b_{11}
  - Things are more complicated if a bucket whose local depth is equal to the depth of the directory overflows
  - Consider what happens when a key with *h*-value 0001 arrives at the table in Figure 10.14b

Hash Functions for Extendible Files (continued)

- Linear hashing (continued)
  - Current loading in Figure 10.16a is 75%, but when key 10 arrives it is hashed to position 1 and loading increases to 83%
  - The first bucket splits and keys are distributed using function h_1, shown in Figure 10.16b
  - Of note is the fact that the first bucket has the lowest load, but was the first one that split
  - Now assume 21 and 36 are hashed to the table, and 25 arrives as seen in Figure 10.16c
  - The loading factor rises to 87%, so another split, this time the second bucket, gives rise to the configuration in Figure 10.16d
  - After hashing 27 and 37 another split occurs, and this new situation is shown in Figure 10.16e

Hash Functions for Extendible Files (continued)

- Linear hashing (continued)
  - In this case, *split* reaches the last allowed value on this level, so it is assigned the value zero, and h_1 is retained for use in further hashing
  - In addition, a new function, h_2 is defined as $K \bmod 4 \cdot TSize$
  - These steps are presented in a table on the next slide
  - Note that since the order of splitting is predetermined, some overflow area is needed with linear hashing
  - If files are used, this may lead to more than one file access
  - The area can be distinct from buckets, but can also work in the fashion of coalesced hashing by using empty space in buckets, as suggested by James Mullin in 1981
  - Overflow areas are not necessary in a directory scheme, but can be used