Lecture # 20

Hash Tables

- Fast searching is the goal of hash table, in which the location of an item is determined directly as a function of the item itself rather than by a sequence of trial- and - error comparisons.

- Under ideal circumstances, the time required to locate an item in a hash table is constant and does not depend on the number of items stored.

Hash function

- As an illustration, suppose that up to 25 integers in the range 0 to 999 are to be stored in the hash table.
- If we want to store any integer in an array (hash table), then we can define a function like following and initialize whole array by -1:

$$h(i) = i$$

- Then finds the location of an item i in the hash table and store the number i in the hash table at location i is called hash function.

- The above hash function:
  - Is time efficient
  - But surely not space efficient.

- Because only 25 of 1000 locations are utilized.

- Since it is possible to store 25 values in 25 locations, we can try to improve the space utilization by using array of size 25.

- But then the original hash function

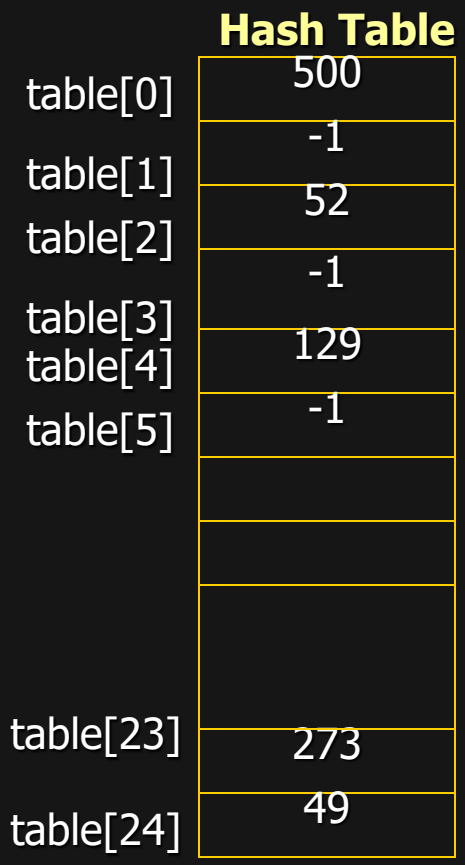
  h(i) = i

  could not be used. But we might use

  h(i) = i modulo 25 or

  h(i) = i % 25

- Because this function will always produce an integer in the range 0 and 24



- The integer 52 thus is stored in the table[2], since h(52) = 52 % 25 = 2. similarly 129, 500, 273 and 49 are stored in location 4,0, 23, and 24 *respectively*.

Collision strategies

- There is an obvious problem with the preceding hash table, namely, that collisions may occur. For example, if 77 is to be stored , it should be placed at location h(77) = 77 % 25 = 2, but this location is already occupied by 52.

- In the same way, many other values may collide at a given position , for example, 2, 27, 102; and , in fact, all integers of the form 25k + 2 hash to location 2. obviously , some strategy is needed to resolve such collisions.

- One simple strategy for handling collisions is known as **Linear Probing.**

- In this scheme, a linear search of the table begins at the location where a collision occurs and continues until an empty slot is found in which the item can be stored.

- Thus, in the preceding example, when 77 collides with the value 52 at location 2, we simply put 77 in position 3; to insert 102, we follow the probe sequence consisting of location 2,3,4, and 4 to find the first available location and thus store 102 in table [5].

- If the search reaches the bottom of the table, we continue at the first location. For example 123 is stored in location 1, since it collides with 273 at location 23, and the probe sequence 23, 24, 0, 1 locates the first empty slot at position 1.

- To determine (search) a specified value is in this hash table, we first apply the hash function to compute the position at which this value should be found. There are three cases to consider:

- First, if this location is empty, we can conclude immediately that the value is not in the table.

- Secondly, if this location contains the specified value, the search is immediately successful.

- In the third case this location contains the value other than the one for which we are searching, because of the way that collision were resolved in constructing the table. In this case , we begin a " *circular linear search* " at this location and continue until either the item is found or you reach an empty location or the starting location, indicating that the item is not in the table. Thus , the search time in the first two cases is constant, but in this last case, it is not.

Improvements in Hashing

- There seems to be three things, we might do to improve performance

1. Increase the table capacity
2. Use a different strategy for resolving collisions
3. Use a different hash function

- Making the table capacity equal to the number of items to be stored , as in our first example, is usually not practical, but using any smaller table leaves open the possibility of collisions.

- In fact, even though the table is capable for storing considerably more items than necessary, collisions may be quite likely. For example for hash table with 365 locations in which 23 randomly selected items are to be stored, the probability that a collision will occur can be more in worst case.

- Thus, it is clearly unreasonable to expect a hashing scheme to prevent collisions completely. Instead we must be satisfied with hash tables in which reasonably few collisions occur.

- Empirical (experimental/practical) studies suggests using tables, whose capacities are approximately 1.5 to 2 times the number of items that must be stored.

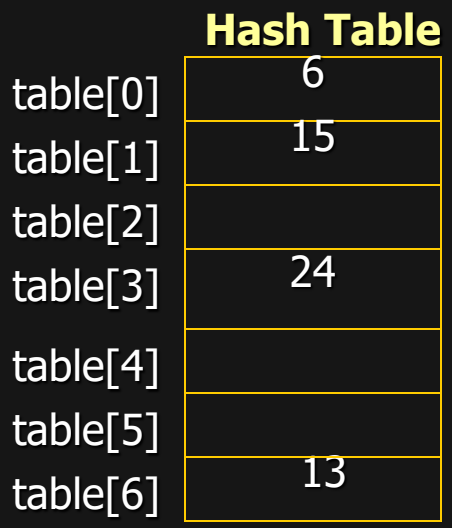
- A second way to improve performance is to design a better method for handling collisions.

- In the linear probe scheme, whenever collisions occur, the colliding values are stored in locations that should be reserved for items that hash directly to these locations. So that approach could not completely solve the problem.

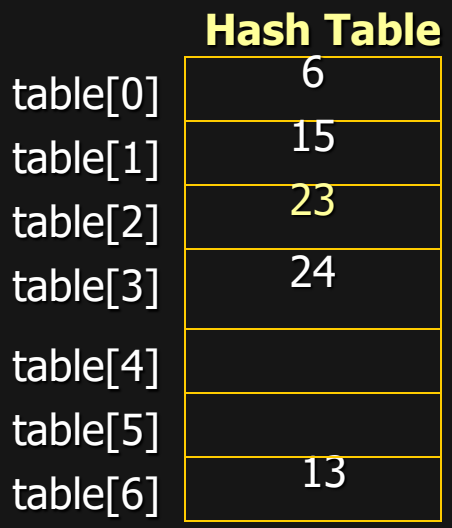
Rehashing

- If the table gets too full, the running time for the operations will start taking too long.
- One of the solution is to build another table that is about twice as big (with associated new hash function) and scan down the entire original hash table, computing the new hash value for all non deleted elements and inserting it into new table.

- As an example, suppose 13, 15, 24 and 6 are inserted into closed hash table of size 7. let the hash function is h(x) = x % 7. suppose linear probing is used to resolve collisions. The resulting hash table is as follows.

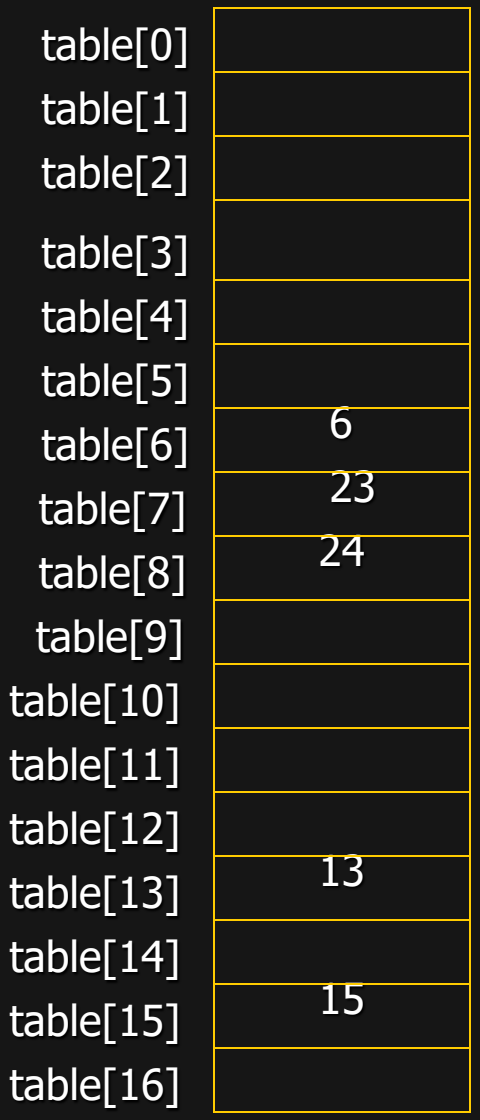


- If lets say, 23 is inserted into the above table, the resulting table will be over 70% full. Since table is mostly full, a new table is created (own criteria in mind).



- The size of new table is 17, because this the first prime that is twice as large as the old table size (one of the criteria).
- The newly hash function is then h(x) = x % 17.
- The old table is scanned and elements : 6, 15, 23, 24 and 13 are inserted into the newly made table.
- The resulting table is as follows.

Hash table after rehashing



- The entire above operation is called *rehashing*.

- This is obviously very expansive operation. There are n elements to rehash and table size is roughly 2n. But actually it is not that much bad as rehashing is done infrequently according to the criteria set in algorithm.

- In particular, there must have been n/2 inserts prior to last rehash.

- If this data structure is a part of the program, the effect is not noticeable. On the other hand, if the hashing is performed as part of an interactive system, then the unfortunate user whose insertion caused a rehash could see a slow down.

- Rehashing can be implemented in many ways:
- One alternative is to rehash as soon as the table is half full.
- The other extreme is to rehash when an insertion fails (e.g. collision occurs).
- A third, a middle of the road strategy is to rehash when table reaches to certain load factor (like 70% filled criteria in above example).

- Since the performance does degrade as the load factor increases, third strategy, implemented with good cutoff, could be best.

Open Hashing (Separate Chaining)

- An another approach, known as ***chaining***, uses a hash table that is an array or vector of linked lists that store the items.

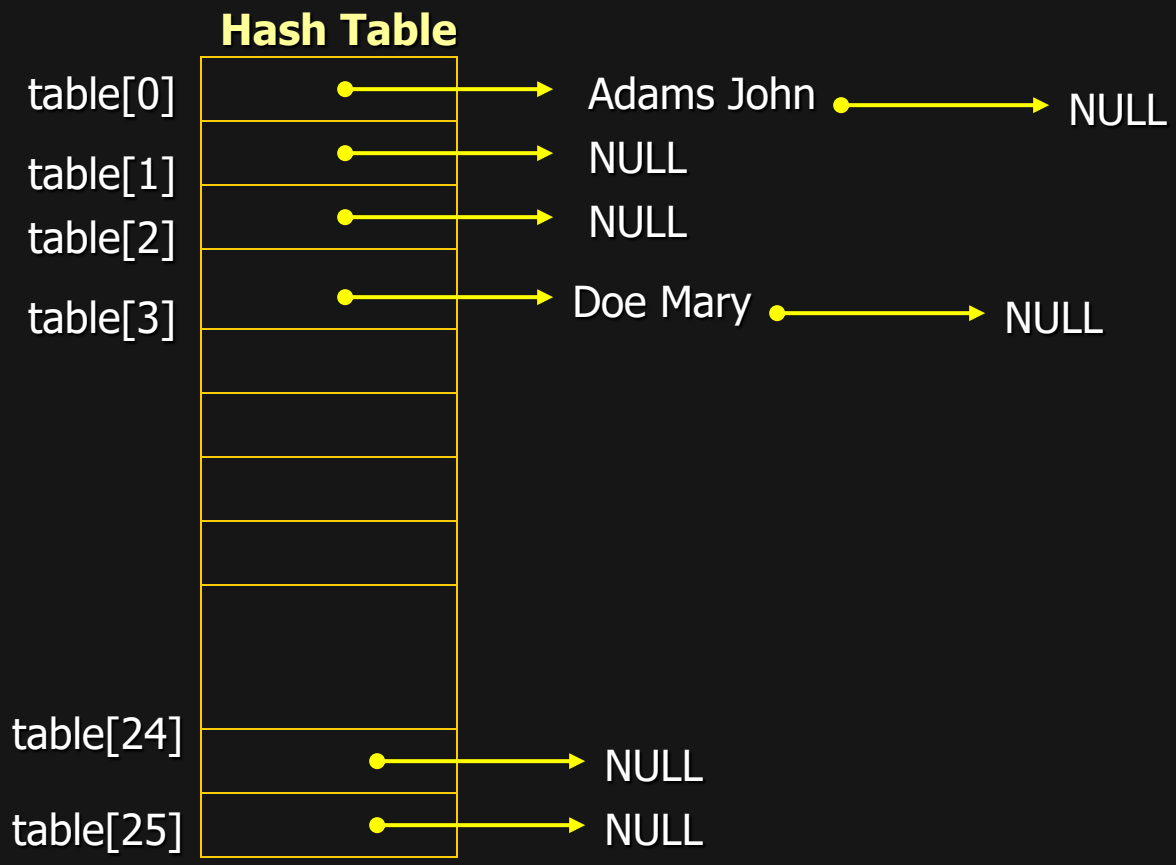
- To illustrate , suppose we wish to store a collection of names . We might use an array table of 26 linked lists, initially empty, and the simple hash function.

  h(name) = name[0] – 'A';

  i.e. h(name) is 0 if name[0] is 'A',

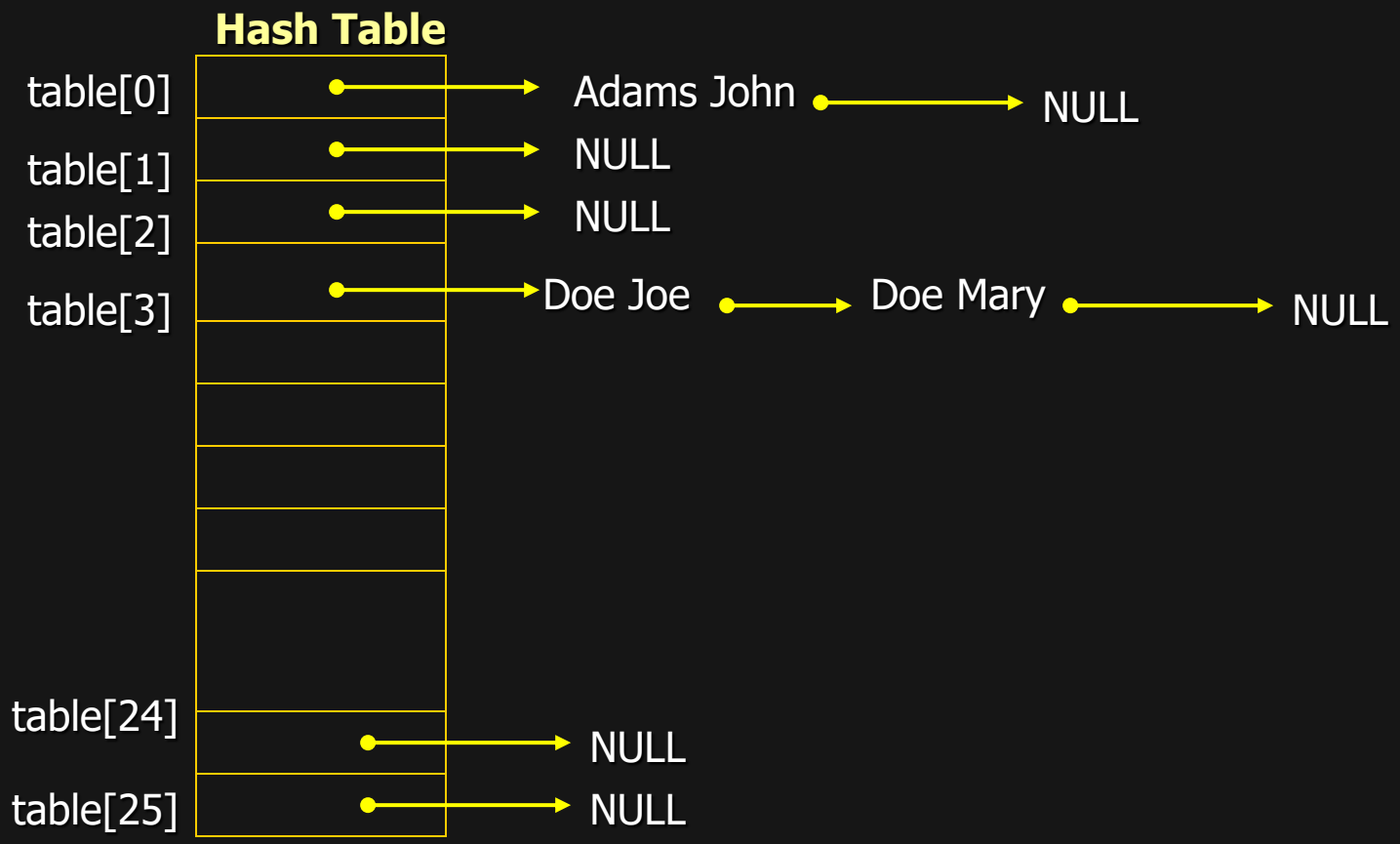
  1 if name[0] is 'B', ……, and

  25 if name[0] is 'Z'.


Hash Table
table[0]
table[1]
table[2]
table[3]
table[24]
table[25]
Adams John
NULL
NULL
NULL
Doe Mary
NULL
NULL
NULL

- Thus for example, “Adams John” and “Doe Mary” are stored in nodes pointed to by table[0] and table[3], respectively.

- When a collision occurs, we simply insert the new item into the appropriate linked list. For example, since h(“Davis Joe”) =h(“Doe Marrie”) = ‘D’ – ‘A’ = 3, a collision occurs when we attempt to store the name “Davis Joe” and thus we add a new node containing this name to the link list pointed to by the table[3].


Hash Table
table[0]
table[1]
table[2]
table[3]
table[24]
table[25]
Adams John
NULL
NULL
NULL
Doe Joe
Doe Mary
NULL
NULL
NULL

- Searching such a hash table is straightforward. We simply apply the hash function to the item being sought and then use one of the efficient search algorithms for Linked lists.

- So by this approach we can reduce reasonably our searching time.

Applications of Hashing

- Compilers use hash tables to keep track of declared variables (symbol table).
- A hash table can be used for on-line spelling checkers — if misspelling detection (rather than correction) is important, an entire dictionary can be hashed and words checked in constant time.

Applications of Hashing

- Hash functions can be used to quickly check for inequality — if two elements hash to different values they must be different.

When is hashing suitable?

- Hash tables are very good if there is a need for many searches in a reasonably stable (means insertions are very rare) table. E.g. for on-line spelling checkers — if misspelling detection (rather than correction)
- Hash tables are not so good if there are many insertions and deletions, or if table traversals are needed — in this case, AVL trees are better.
- Also, hashing is very slow for any operations which require the entries to be sorted
  - e.g. Find the minimum key

Thank You……