Lecture # 5

- ➢ Elementary Data Structures
- ➢ Hashing

Divide-and-Conquer Algorithm

➢ The simplest application of divide-and-conquer algorithm is Binary Search algorithm.

➢ Before going to binary search we analyze linear search as it takes $\Omega(n)$ time in *worst case* and $O(1)$ in the *best case*.

➢ The *pre-requisite* for binary search algorithm is that the list must be sorted in some order.

```cpp
void b_search :: search(int m,int n)
{
  low=m;
  high=n;
  temp = mid;
  mid=(low+high)/2;
  if( (temp == mid) && (a[mid] < num) )
          mid++;
  if( (temp == mid) && (a[mid] > num) )
          mid--;
  if(a[mid]==num)
   {
      test=1;
      return;     }
  if(a[mid]>num)
     high=mid;
  if(a[mid]<num)
     low=mid;
  if(low>=high)
     return;
  search(low,high);
}
```

Subject to improve …

➢ In above algorithm (code) each pass to this b_search (int m, int n) function reduces the array to be searched by at least one half the size of the sub list still to be searched.

➢ The last pass occurs when size of the sub list reaches one.

➢ Thus the total number of recursive function calls is 1 plus the number k passes required to produce a list of size 1.

➢ Since the size of the sub list after k passes is at most $n/2^k$. We must have

$$n/2^k < 2$$

i.e.

$n < 2 * 2^k$

$n < 2^{k+1}$

Taking $\log_2$ on both sides

$\log_2 n < \log_2 2^{k+1}$

$\log_2 n < K+1 * \log_2 2$

$\log_2 n < K+1 * 1$

$\log_2 n < K+1$

so

➢ So the required number of passes, therefore is the smallest integer that satisfies this inequality.

➢ So the Time Complexity of binary search is $O(\log_2 n)$.

➢ How the two discussed divide - and - conquer algorithms differ – Merge sort and Binary Search **???**

➢ What would be the recurrence relation for binary search algorithm **???**

$$T(n) = \begin{cases} \Theta(1) & n \leq 2 \\ T\left(\dfrac{n}{2}\right) + \Theta(1) & n > 2 \end{cases}$$

Elementary Data Structures

- A stack is an ordered collection of items into which new items may be inserted and from which items may be deleted at one end only, called **top** of the stack. i.e. deletion/insertion can only be done from **top** of the stack.

- The insert operation :- **Push**
- The delete operation :- **Pop**

➤ Figure showing stack…..

Top of the Stack

D

C

B

A

NULL
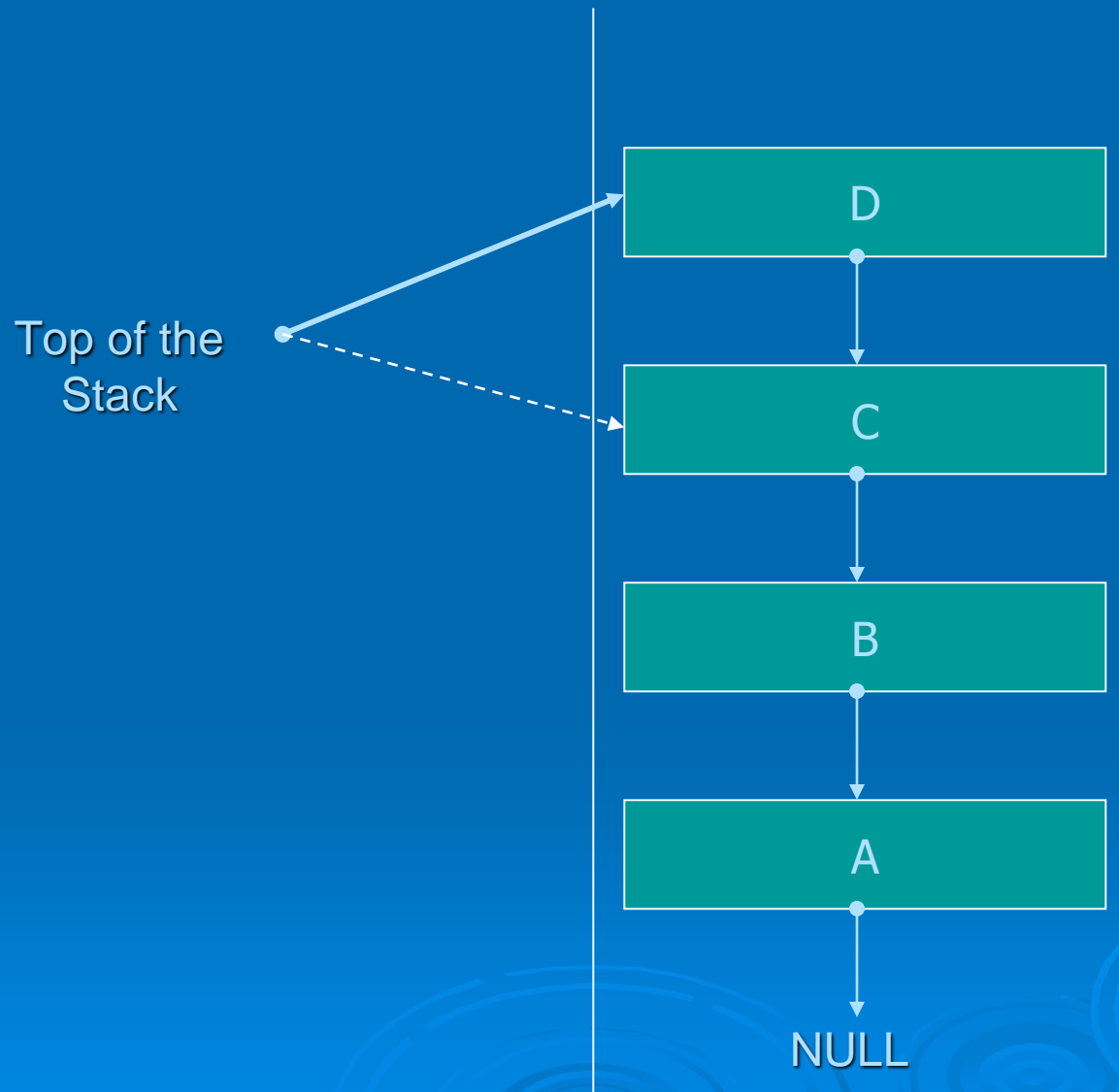
- Stack is also referred as Last-in First-out i.e LIFO.
- empty means stack underflows
- If top of the stack exceeds n, the stack overflows.

- **Time Complexity:**
  - Push :
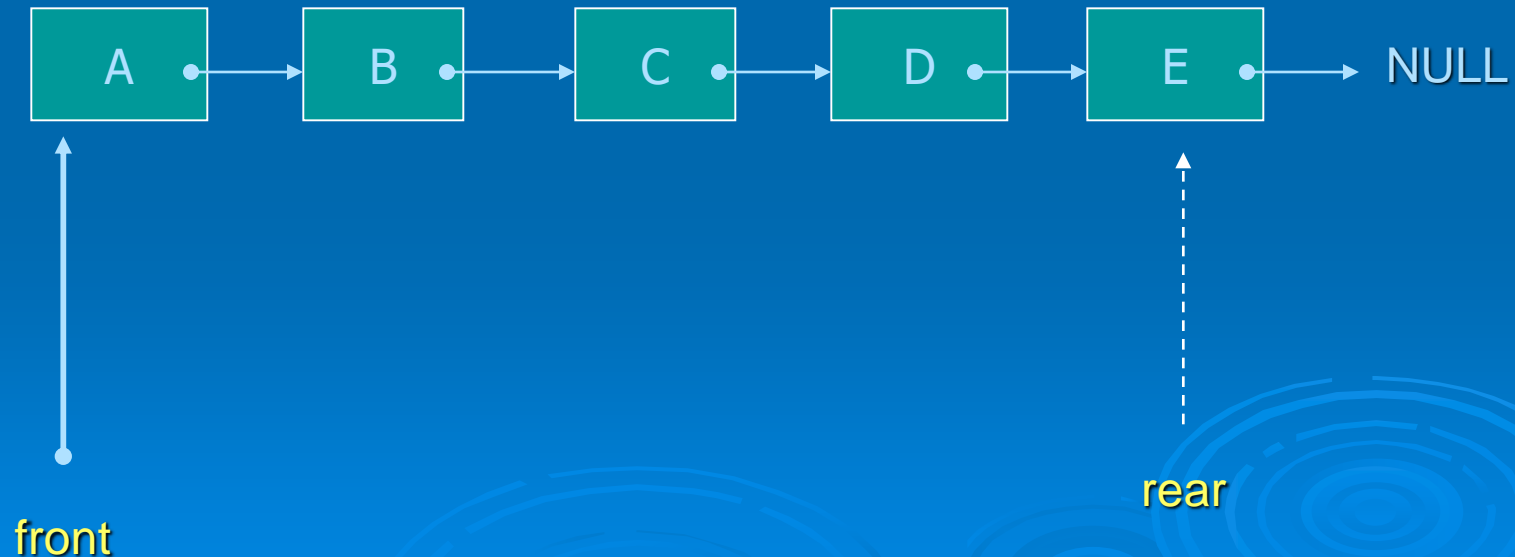
    O (1) as only one operation is required.

  - Pop :

 O (1) as only one operation is required.

  - Is_Empty :

    O (1) as only need to check the top of Stack.

➢ In a queue, the element deleted from the list is the one which inserted first.

➢ Queue is also referred as First-in First-out i.e FIFO.

```
┌─────┐     ┌─────┐     ┌─────┐     ┌─────┐     ┌─────┐
│  A  ●────▶│  B  ●────▶│  C  ●────▶│  D  ●────▶│  E  ●────▶ NULL
└─────┘     └─────┘     └─────┘     └─────┘     └─────┘
   ▲                                               ▲
   │                                               ┊
  front                                           rear
```

- ➢ If an empty queue is deleted, we say queue underflows and

- ➢ If rear of the queue exceeds n, the queue overflows.

- ➢ Insertion to queue is called Enqueue and deletion form queue is called Dequeue.

- ➢ **Time Complexity:**

  - Enqueue :

    O (1) as only one operation is required.

  - Dequeue :

    O (1) as only one operation is required.

➢ Link List is a Data Structure in which elements are explicitly ordered, that is each item contains within itself the address of next item.

➢ Comparison with array Data Structure

node   node   node   node

| Item | next | | Item | next | | Item | next | | Item | next |

NULL

Handle of the LL
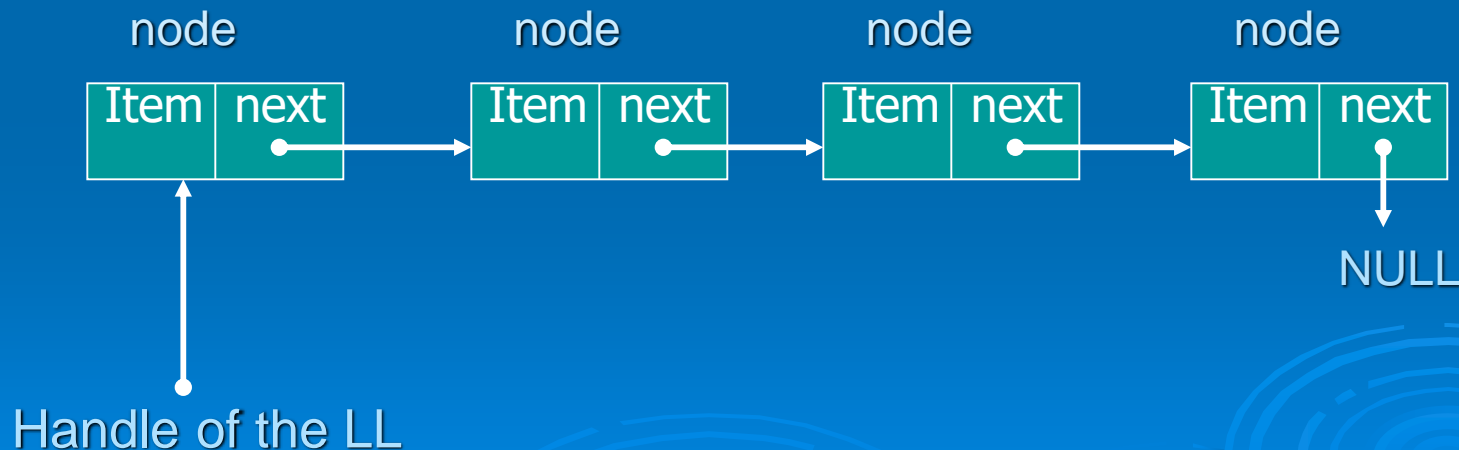
Analysis & Operations of Link List

➢ Adding (Inserting) to a Link List (LL) :
   Time Complexity : *O* (1) or Θ(n)

➢ Searching a Link List :

   Time complexity : Θ(n)

➢ Deletion from Link List :

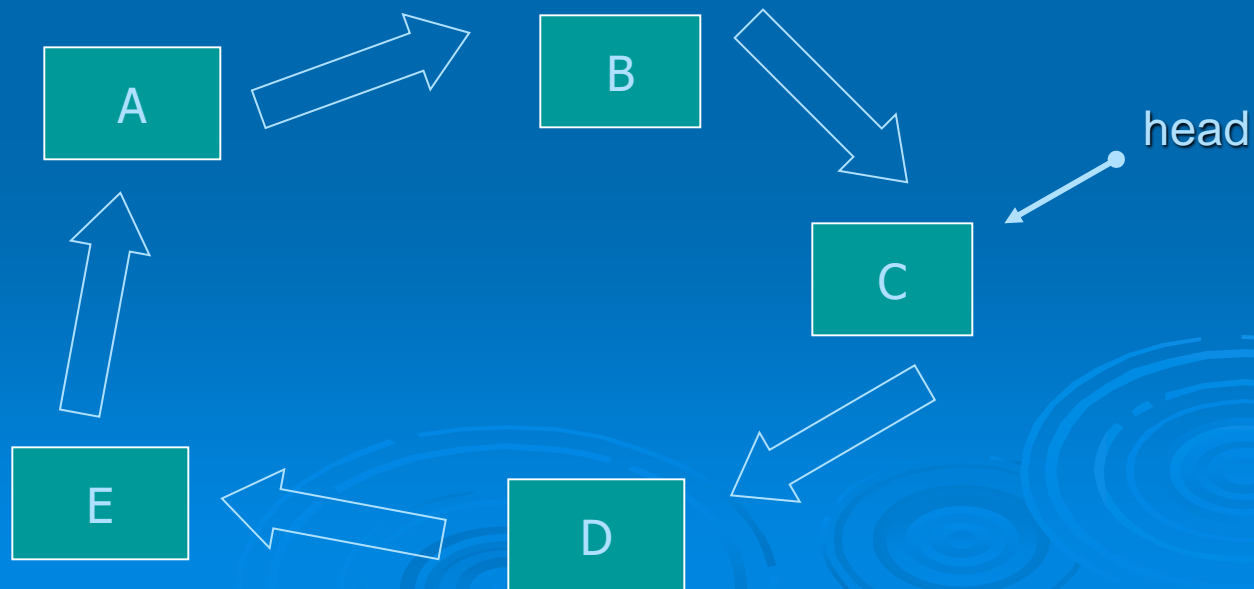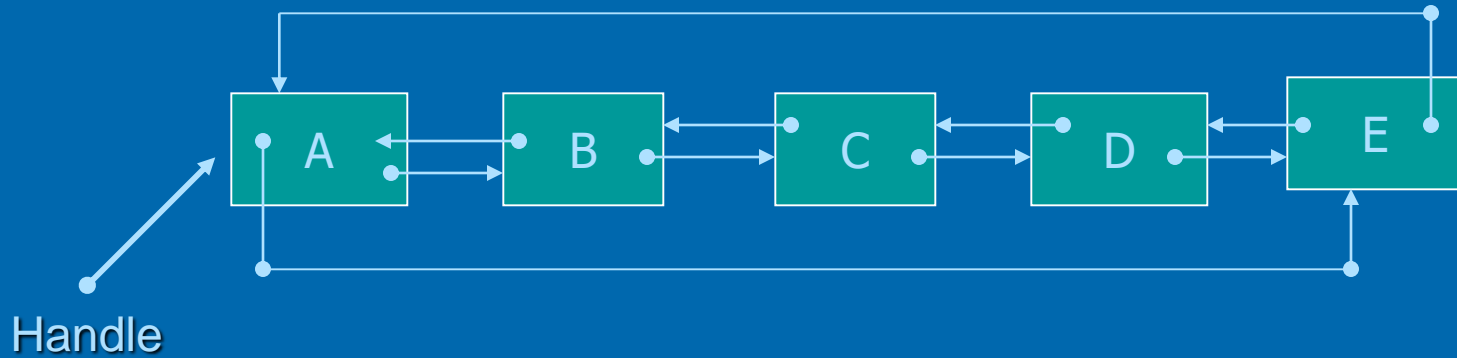   From the start of link list.

                Time Complexity : *O*(1)

   Deletion of particular node (block) with given key

                Time Complexity : Θ(n)

➤ Analysis of Ring **/** Doubly Ring …..

Handle

head

➢ Can *binary search* kind of algorithm be applied on *doubly linked list* to reduce the time of an algorithm.

- If yes how …
- If no why …

Hash Table

- Given a table *T* and a record *x*, (with key and satellite data), we need to support:
  - Insert (*T*, *x*)
  - Delete (*T*, *x*)
  - Search(*T*, *x*)

- We want these to be fast, but don't care about sorting the records

Direct-address table

➢ If the keys are drawn from the reasoning small universe U = {0, 1, . . . , m-1} of keys, a solution is to use a Table T[0, . m-1], indexed by keys.

➢ To represent the dynamic set, we use an array, or direct-address table, denoted by T[0 . . m-1], in which each slot corresponds to a key in the universe.

- Following figure illustrates the approach.
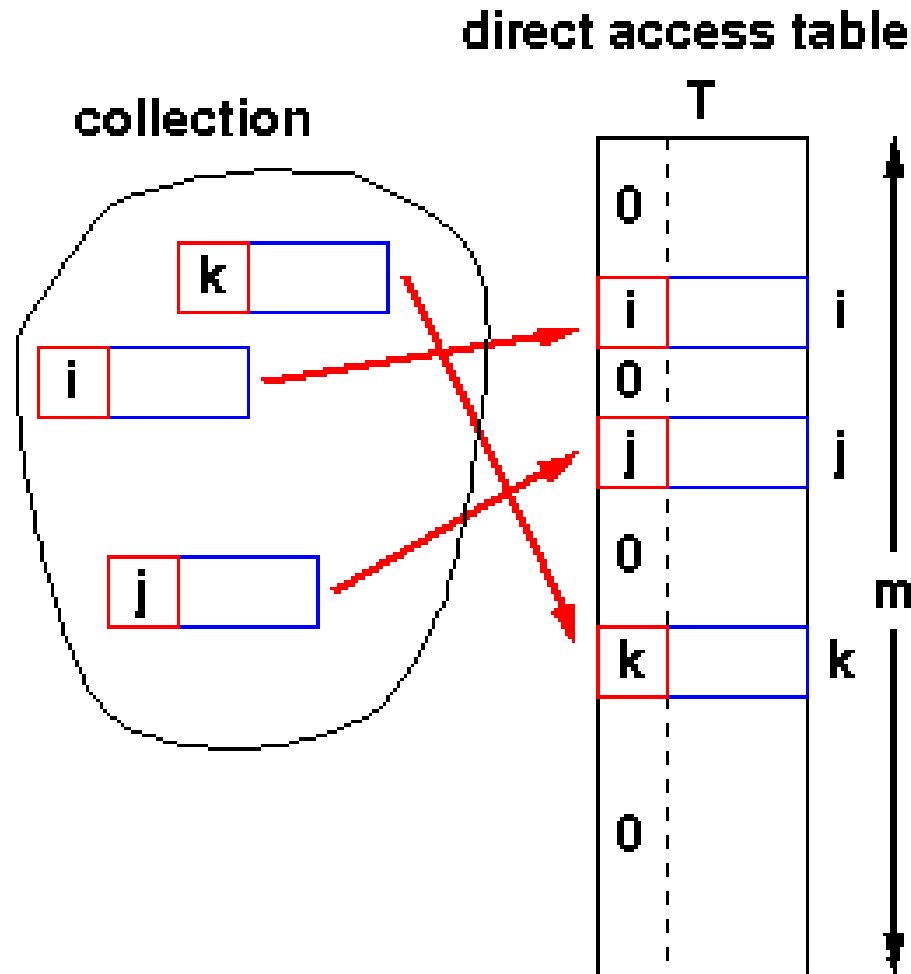
➤ Each key in the universe U {Collection} corresponds to an index in the table T[0 . . m-1]

➤ Using this approach, all three basic operations

Insert (*T*, *x*),

Delete (*T*, *x*), (dictionary operations)

Search(*T*, *x*)

take $\theta(1)$ in the worst case.

Hash Tables

➢ When the size of the universe is much larger the same approach (direct address table) could still work in principle, but the size of the table would make it impractical. A solution is to *map the keys onto a small range*, using a function called a hash function. The resulting data structure is called *hash table*.

➢ With direct addressing, an element with key k is stored in slot k. With hashing, this same element is stored in slot h(k); that is we use a hash function h to compute the slot from the key. Hash function maps the universe *U* of keys into the slot of a hash table T[0 . . .m-1] i.e. h: $U \rightarrow \{0, 1, . . ., m-1\}$

➢ Typical, hash functions generate "random looking" values.

➢ For example, the following function usually works well

$$h(k) = k \bmod m$$

(where m is a prime number)

➢ The point of the hash function is to reduce the range of array indices that need to be handled.

Collision

- As keys are inserted in the table, it is possible that two keys may hash to the same table slot.
  If the hash function distributes the elements uniformly over the table, the number of collisions cannot be too large on the average, but it is very likely that there will be at least one collision, even for a lightly loaded table .
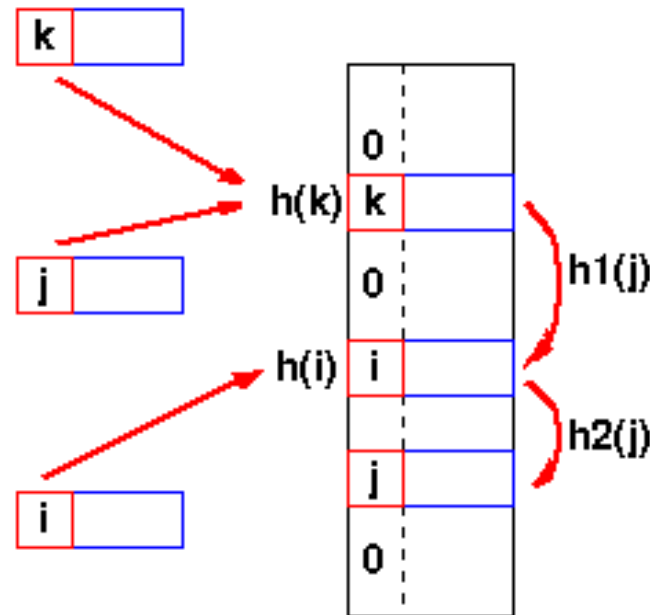
- A hash function h map the keys k and j to the same slot, so they collide .

- There are two basic methods for handling collisions in a hash table:
  - Chaining and
  - Open addressing.

Collision Resolution by **Chaining**

- When there is a collision (keys hash to the same slot), the incoming keys is stored in an overflow area and the corresponding record is appeared at the end of the linked list

T[j]
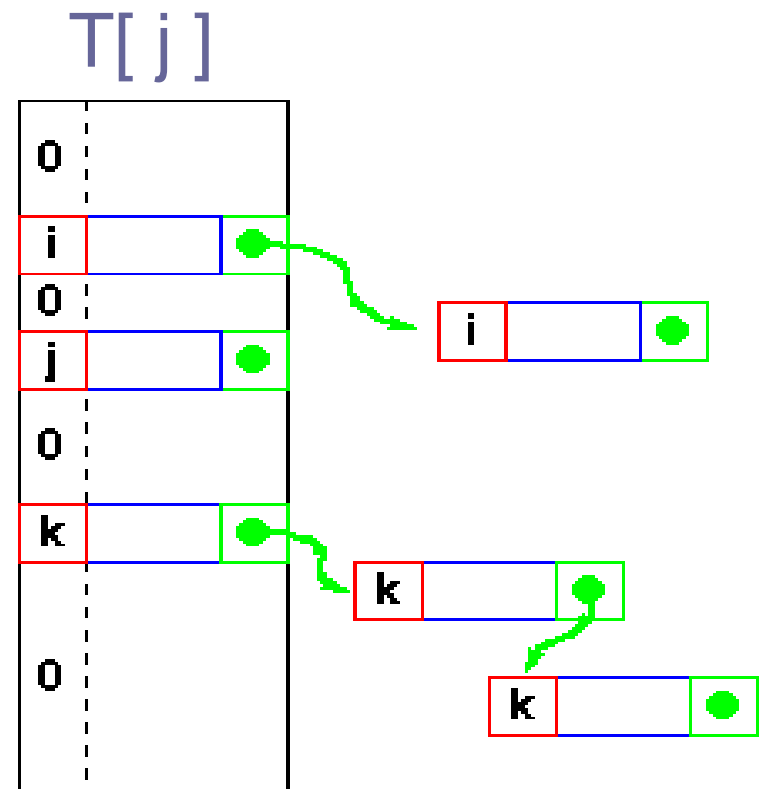
- Each slot T[j] contains a linked list of all the keys whose hash value is j. For example, $h(k1) = h(kn)$ and $h(k5) = h(k2) = h(k7)$.

- For Example:

- keys: 5, 28, 19, 15, 20, 33, 12, 17, 10
  slots: 9
  hash function = $h(k) = k \mod 9$

  $h(5) = 5 \mod 9 = 5$
  $h(28) = 28 \mod 9 = 1$
  $h(19) = 19 \mod 9 = 1$
  $h(15) = 15 \mod 9 = 6$
  $h(20) = 20 \mod 9 = 2$
  $h(33) = 33 \mod 9 = 6$
  $h(12) = 12 \mod 9 = 3$
  $h(17) = 17 \mod 9 = 8$
  $h(10) = 10 \mod 9 = 1$

- The worst case running time for insertion is O(1), if node is added at the head of list.

- In the worst case behavior of chain-hashing, all n keys hash to the same slot, creating a list of length n. The worst-case time for search is thus θ(n) plus the *time to compute the hash function*.

- Deletion of an element x has the same worst-case time as for search operation if the lists are singly linked.

Assignment # 3

➤ You have the two sorted lists of N/2 elements each. Find the median N numbers when two lists are combined. Suggest an *efficient algorithm* for solving the problem and give its analysis (Recurrence Relation / Asymptotic notation etc...)

➤ For example

List 1 :- 80 85   86 87   100

List 2 :- 2   60   69 70   86

Median is :- 80 and 85

Due: Next Week………….