CS 2124: DATA STRUCTURES
Spring 2024

Lecture 13

Topics: Hash Tables

Information About Final Exam And Remaining Points

- Final exam is remote
- Final Exam is from Lesson – 7 (Intro. To trees) to Lesson 13 (Hash Tables)

- Class Participation 5-points (will be updated once the course evaluation list is available)
    - 3 Points Class participation
    - 2 points course evaluation

    - Extra Credit points

Section	Review Vote	Extra Credit Review	No Vote	Total Votes
0C3	13	17	2 - but selected topic to present	32
0D4	12	13	6 - but selected topic to present	31
0E1	5	16	3- but selected topic to present	24

Topics (Presentation – 5 minutes Max) – Extra Credit

1. B- Tree (Explain Order and Creation - order 3 or 5)
2. B- Tree (Insertion and deletion operation - order 3 or 5)
3. Segment tree (Array to Tree and tree to array)
4. Dijkstra’s Shortest Path Algorithm (use a graph to show how the algorithm works)
5. Hashing (Avoid Collisions)
6. Warshall's Algorithm (use a graph to show how the algorithm’s working)
7. BFS traversal
8. DFS Traversal
9. MST (use a graph to show how the algorithm works)
10. Kruskal Algorithm (use a graph to show how the algorithm works)
11. RB- Tree (Explain Tree and Creation)
12. RB- Tree (Insertion and deletion operation)
13. Building Huffman Tree using Heap
14. AVL Tree (Insertion - use a tree to show how the it works)

OC3	
Topic #	# of Students
1	2
3	2
4	5
5	4
6	2
7	3
8	1
10	4
11	2
13	4
14	3

OD4	
Topic #	# of Students
1	1
2	1
3	4
4	2
5	2
6	2
7	3
8	2
9	3
10	2
11	1
12	1
13	3
14	4

OE1	
Topic #	# of Students
1	1
2	2
3	2
4	1
6	1
7	2
8	2
11	1
12	1
13	4
14	7

Presentation:

Presentations will be during 14th Week

- 30th April – Tuesday – During Lecture Timings
- 2nd May – Thursday – During Lecture Timings

In person using Power point or white board

Material from the lecture slides is not allowed

Each topic must include :

- Defining the topic (2 points)
- 1 - Example for the topic (1.5 points)
- 1 - Practical or real world example of the topic (1.5 points)

- Max 8 minutes to present the topic
- Presentation sequence is as per the selected topic

Open Chaining (Avoiding Collusion)

```c
#include<stdio.h>
#include<stdlib.h>
#define size 7
struct node
{
    int data;
    struct node *next;
};
struct node *chain[size];
void init()
{
    int i;
    for(i = 0; i < size; i++)
        chain[i] = NULL;
}
void insert(int value)
{
    //create a newnode with value
    struct node *newNode = malloc(sizeof(struct node));
    newNode->data = value;
    newNode->next = NULL;
    int key = value % size; //calculate hash key
    //check if chain[key] is empty
    if(chain[key] == NULL)
        chain[key] = newNode;
    else    //collision
    {
        //add the node at the end of chain[key].
        struct node *temp = chain[key];
        while(temp->next)
        {
            temp = temp->next;
        }

        temp->next = newNode;
    }
}
void print()
{
    int i;
    for(i = 0; i < size; i++)
    {
        struct node *temp = chain[i];
        printf("Index[%d]-->",i);
        while(temp)
        {
            printf("%d -->",temp->data);
            temp = temp->next;
        }
        printf("NULL\n");
    }
}
```

Note: This code will not be part of quiz or exam. It is only for implementation and understanding

Source

Open Chaining (Avoiding Collusion)

```c
int main()
{
    init();
    insert(7);
    insert(0);
    insert(3);
    insert(10);
    insert(4);
    insert(5);
    print();
}
```

```
Index[0]-->7 -->0 -->NULL
Index[1]-->NULL
Index[2]-->NULL
Index[3]-->3 -->10 -->NULL
Index[4]-->4 -->NULL
Index[5]-->5 -->NULL
Index[6]-->NULL
```

Note: This code will not be part of quiz or exam. It is only for implementation and understanding

Source

Open Addressing (Avoiding Collusion)

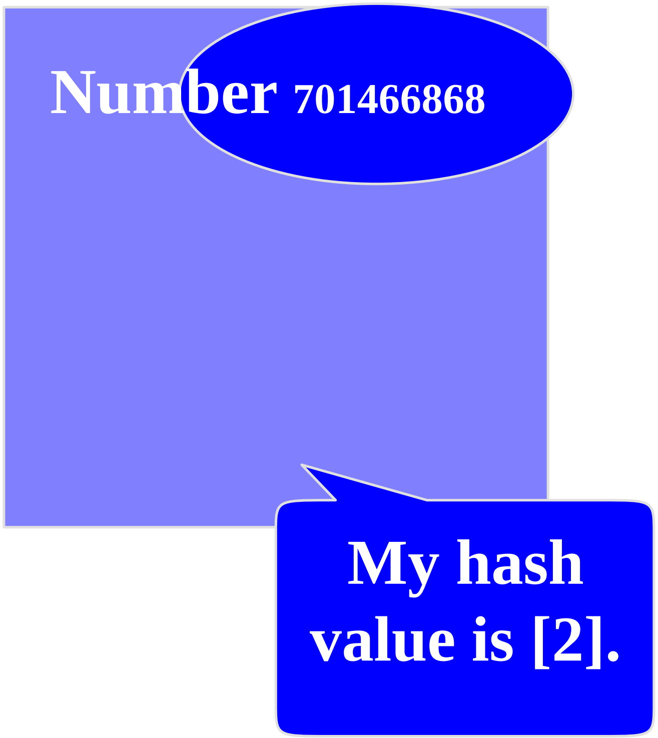
```c
// Function to add key value pair
void insert(int key, int V)
{

    struct HashNode* temp
        = (struct HashNode*)malloc(sizeof(struct HashNode));
    temp->key = key;
    temp->value = V;

    // Apply hash function to find
    // index for given key
    int hashIndex = key % capacity;

    // Find next free space
    while (arr[hashIndex] != NULL && arr[hashIndex]->key != key && arr[hashIndex]->key != -1)
    {
        hashIndex++;
        hashIndex %= capacity;
    }
```

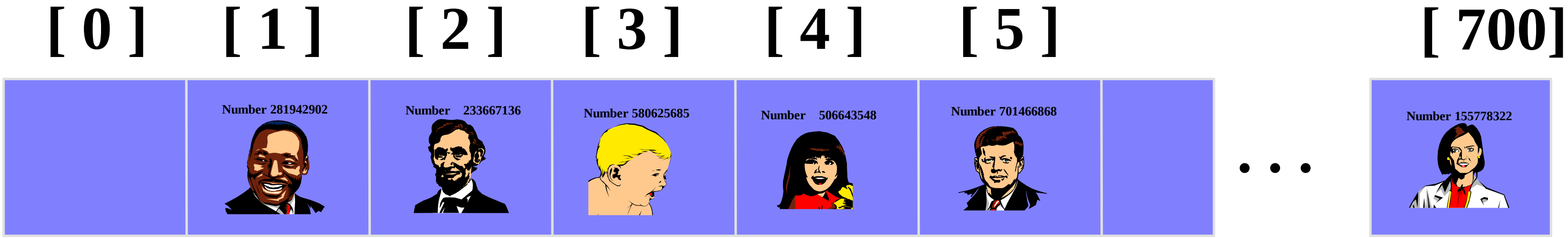
Note: This code will not be part of quiz or exam. It is only for implementation and understanding

Source

Searching for a Key

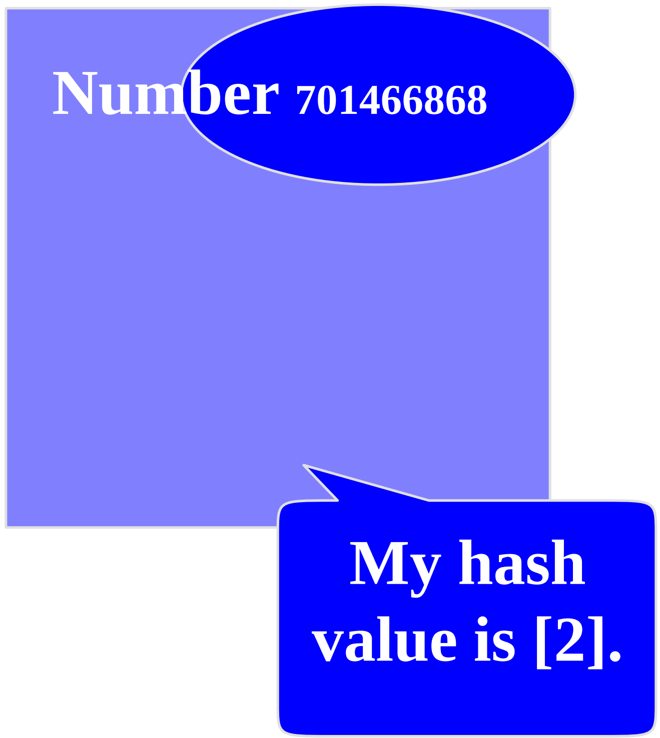
- The data that's attached to a key can be found fairly quickly.

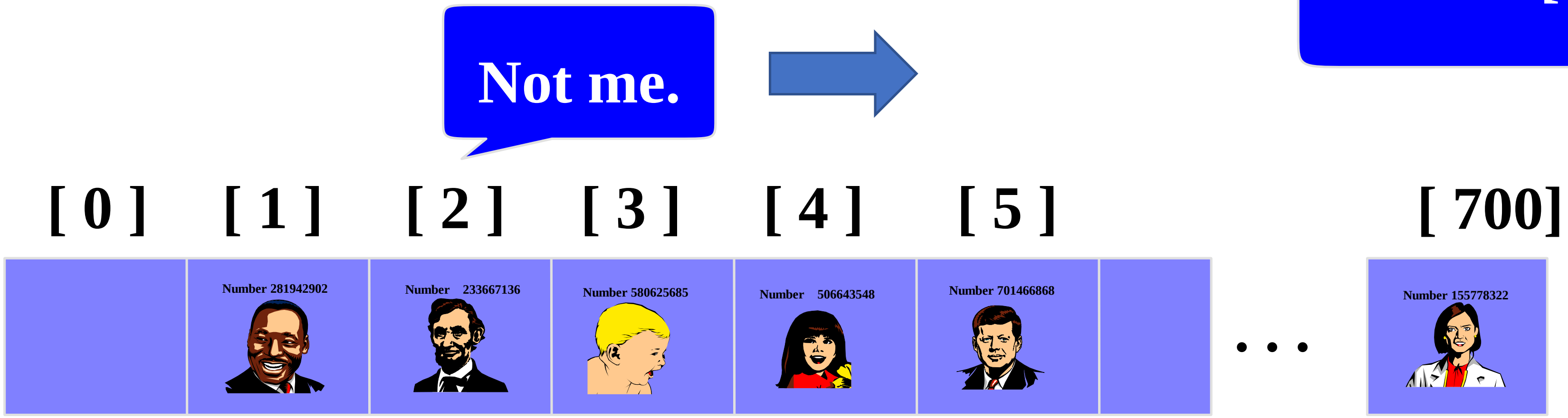




Searching for a Key

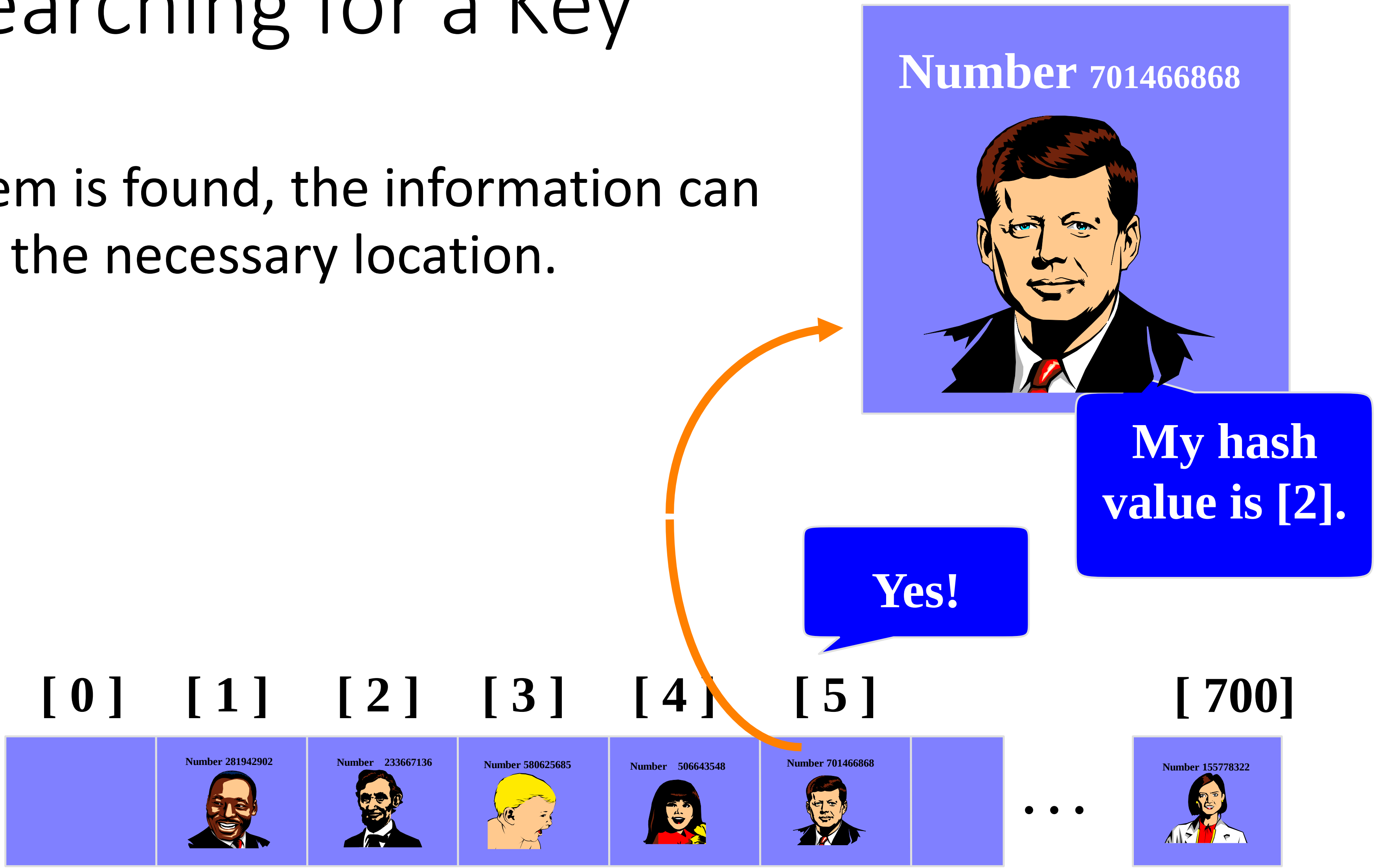
- Calculate the hash value.
- Check that location of the array for the key.
- If location 2 has a different key than the one you are looking for, then move forward...





Searching for a Key

- When the item is found, the information can be copied to the necessary location.



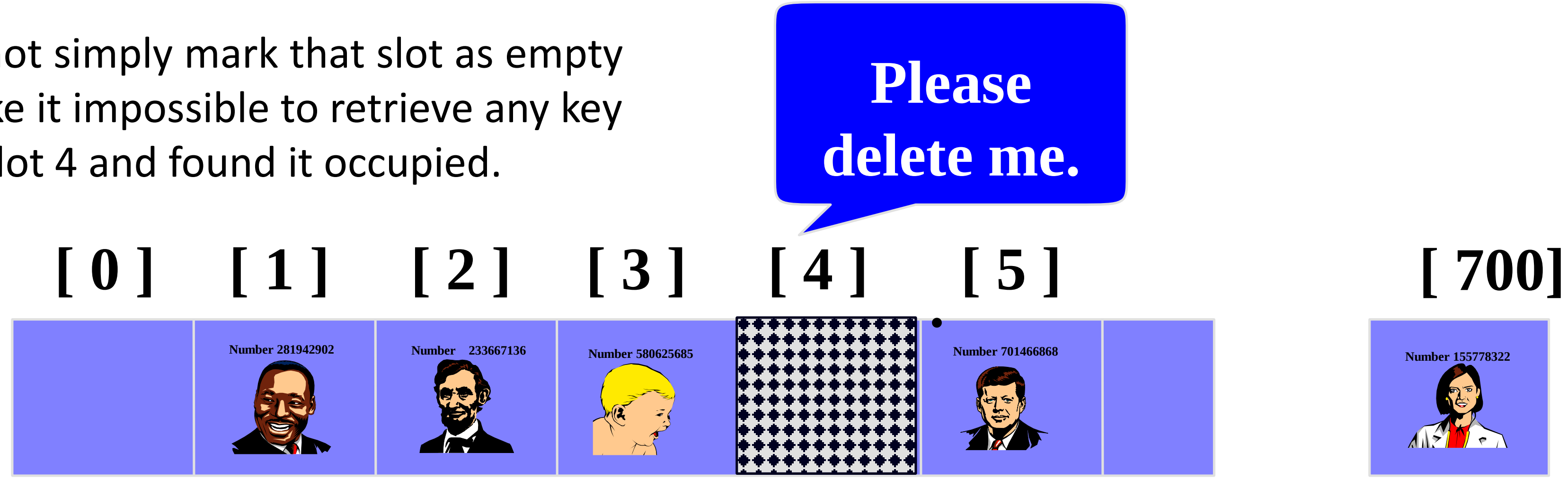
Hash Table
Searching For A Key Or Lookup Issue

- Hash tables store data in pseudo-random locations, so accessing the data in a sorted manner is a very time consuming operation.
- Other data structures such as self-balancing binary search trees generally operate more slowly (since their lookup time is O(log n)) and are rather more complex to implement than hash tables but maintain a sorted data structure at all times
- Although hash table lookups use constant time on average, the time spent can be significant.
- Evaluating a good hash function can be a slow operation.

Deleting a Record

- Records may also be deleted from a hash table.
- But the location must not be left as an ordinary "empty spot" since that could interfere with searches.
- The location must be marked in some special way so that a search can tell that the spot used to have something in it.

When we delete a key from slot 4 , we cannot simply mark that slot as empty by storing {0 or -1} in it. Doing so might make it impossible to retrieve any key (k) during whose insertion we had probed slot 4 and found it occupied.



Hashing Concerns

- Hash tables in general exhibit poor locality of reference i.e. the data to be accessed is distributed seemingly at random in memory. Because hash tables cause access patterns that jump around, this can trigger microprocessor cache misses that cause long delays.

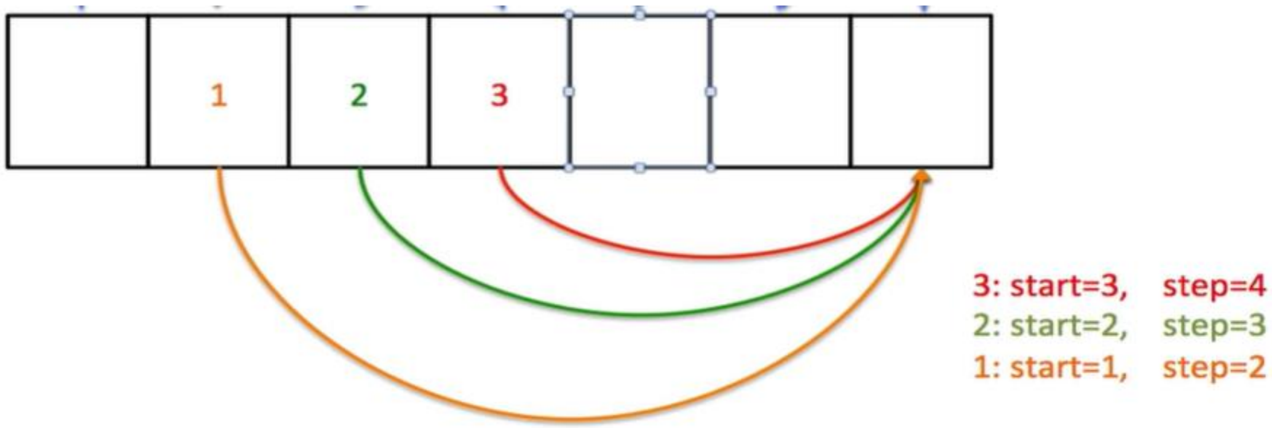
- Hash tables are more difficult and error-prone to write and use. Hash tables require the design of an effective hash function for each key type, which in many situations is more difficult and time-consuming to design and debug

- In some applications, a black hat with knowledge of the hash function may be able to supply information to a hash which creates worst-case behavior by causing excessive collisions, resulting in very poor performance (i.e., a denial of service attack).

Brent’s Method

- This method is a heuristic*. This attempts to minimize the average time for a successful search in a hash table.
- This method was originally applying on double hashing technique, but this can be used on any open addressing techniques like linear and quadratic probing.

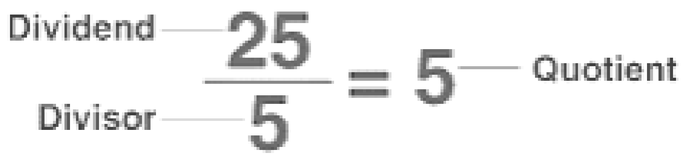
Brent hashing was originally developed to make the double-hashing process more efficient, but it can be successfully applied to any closed hashing process.



** **Heuristic** is a problem-solving strategy or method that is not guaranteed to find the optimal solution, but is designed to find a satisfactory solution in a reasonable amount of time.*

Hashing: Collision Resolution: Brent's Method

- Record Keys: 27, 18, 29, 28, 39
- Table Size = Table
- Hash Function = hash(key) = key mod Table
- Incrementing Function :
  - i(key) = Quotient (Key / Table) mod Table (computed on incoming key)
  - ‘I’ is the function you defined for the increment



Continue next slide ->

Source

Hashing: Collision Resolution: Brent's Method

- Record Keys: 27, 18, 29, 28, 39
- Table Size = Table
- Hash Function = hash(key) = key mod Table
- Incrementing Function :
  - i(key) = Quotient (Key / Table) mod Table (computed on incoming key)
  - 'I' is the function you defined for the increment

1. After inserting 27 and 18, we have a collision on the insertion of 29.
   I. Should we move 18 to reduce the total # of probes?
   II. 18 has 1; 29 has 2;
      a) i(key) = i(18) = 18 has 1 (18 / 11 = > 1.63 % 11);
      b) i(key) = i(29) = 29 has 2 (29/ 11 => 2.63 % 11);

loc	Key	Detail
0		
1		
2		
3		
4		
5	27	27 mod 11 = 5
6		
7	18	18 mod 11 = 7 ; 29 mod 11 = 7 *i(29) = 2 so try 9*
8		
9		
10		

Continue next slide ->

Source

Hashing: Collision Resolution: Brent's Method

- Record Keys: 27, 18, 29, 28, 39
- Table Size = Table
- Hash Function = hash(key) = key mod Table
- Incrementing Function :
  - i(key) = Quotient (Key / Table) mod Table (computed on incoming key)
  - 'I' is the function you defined for the increment

1. After inserting 27 and 18, we have a collision on the insertion of 29.
   I. Should we move 18 to reduce the total # of probes?
   II. 18 has 1; 29 has 2;
   III. Is there any combination of i + j < 2? No, so don't move anything.
   IV. Only if s (# of probes required to retrieve the item, if nothing is moved) is 3 or more do we try to move.
2. i(key) = i(29) = 29 has 2;
3. Move 29 to loc9 (i.e. 7+2 = 9); 7 is the original index, 2 is the quotient

loc	Key	Detail
0		
1		
2		
3		
4		
5	27	27 mod 11 = 5
6		
7	18	18 mod 11 = 7 ; 29 mod 11 = 7 *i(29) = 2 so try 9*
8		
9		
10		

Continue next slide ->

Source

Hashing: Collision Resolution: Brent's Method

- Record Keys: 27, 18, 29, 28, 39
- Table Size = Table;
- Hash Function = hash(key) = key mod Table
- Incrementing Function :
- i(key) = Quotient (Key / Table) mod Table (computed on incoming key)
  - ‘I’ is the function you defined for the increment

1. After inserting 27 and 18, we have a collision on the insertion of 29.
   I. i(key) = i(29) = 29 has 2;
   II. Move 29 to loc9 (i.e. 7+2 = 9)
2. Insert 28 at loc6
3. Insert 39 at loc6 (collision)
   I. s value of 39 is 3 (i(39) = 3; try loc9; then loc1 so we need 3 probes to find 39) - try to reduce this; start with i = 1 and j = 1 ; try moving what is at the home address one offset along its chain i.e. move 28 to (i(28) = 2; so offset is 2) loc8.
   II. This works, so move 28 to loc8 and put 39 in loc6

loc	Key	
0		
1		
2		
3		
4		
5	27	
6	28	28 mod 11 = 6; 39 mod 11 = 6; collision
7	18	
8		
9	29	
10		

Continue next slide ->

Source

Hashing: Collision Resolution: Brent's Method

- Record Keys: 27, 18, 29, 28, 39
- Table Size = Table;
- Hash Function = hash(key) = key mod Table
- Incrementing Function :
- i(key) = Quotient (Key / Table) mod Table (computed on incoming key)
  - ‘I’ is the function you defined for the increment

1. After inserting 27 and 18, we have a collision on the insertion of 29.
   I. i(key) = i(29) = 29 has 2;
   II. Move 29 to 9 (i.e. 7+2 = 9)
2. Insert 28 at 6
3. Insert 39 at 6 (collision) (As i(39) = 3 which is higher then i(28) = 2, meaning keeping 39 at loc6 will result is less proving as compared to value 28)
   I. Move 28 to loc8 and put 39 in loc6

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

Source

Brent’s Method (Do it your self)

- What will be the final locations of the following elements if Brent's Method is use to avoid collision:
  - Record Keys: 20, 10, 30, 40, 31, 50
  - Table Size = 10;
  - Hash Function = key divide Table = Key/Table Size
  - Incrementing Function in case of collision = i+(key current loc); were i = 1.

loc	Key	
0		
1		
2		
3		
4		
5		
6		
7		
8		
9		

1. *After inserting 20*
   I. *20/10 = 2; insert 20 at loc2*

   ..

   ..
2. *Case of collision = 31/10 = 3; loc3 contain 30; collision*
   I. *i+(key current loc) = 1+3 = 4; assign loc4 to 31*
3. *Case of collision as 40 is at loc4*
   I. *i+(key current loc) = 1+4 = 5; assign loc5 to 31*

Indexing => loc1 = 10; loc2 = 20; loc3=30; loc4=40; loc5=31; loc6=50

Quiz and Final Exam

- Quiz – 2 - 30th April – 6:00 AM till End of day (11: 58 PM)
- Final Exam - 7th May, 8:00 AM - 3:00 PM
- Presentations (extra credit) will be during 14th Week
    - 30th April – Tuesday – During Lecture Timings
    - 2nd May – Thursday – During Lecture Timings

We completed the course without any disaster
Or
did we !