



Data Structures and its Applications

V R BADRI PRASAD

Department of Computer Science & Engineering

DATA STRUCTURES AND ITS APPLICATIONS

Introduction to TRIE Trees

V R BADRI PRASAD

Department of Computer Science & Engineering

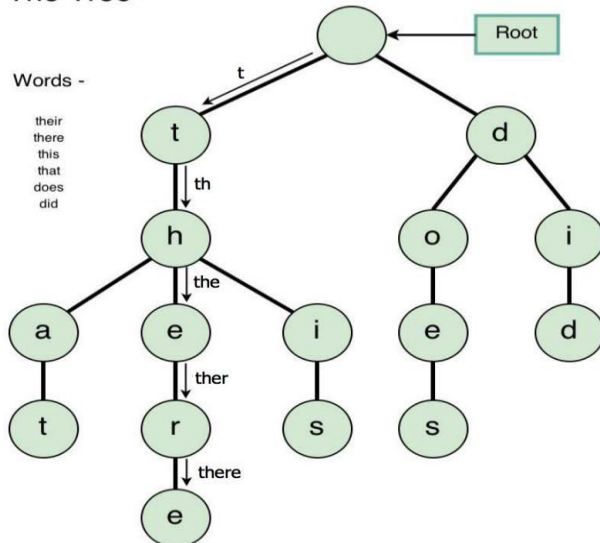
Data Structures and its Applications

TRIE Trees – An Introduction

- **TRIE** tree is a digital search tree, need not be implemented as a binary tree.
- Each node in the tree can contain 'm' pointers – corresponding to 'm' possible symbols in each position of the key.
- Generally used to store strings.

Examples:

Trie Tree -

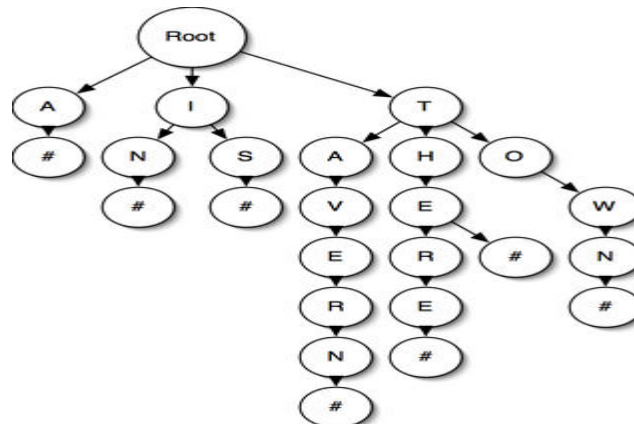


Data Structures and its Applications

TRIE Trees – An Introduction

- A trie, pronounced “try”, is a tree that exploits some structure in the keys
 - e.g. if the keys are strings, a binary search tree would compare the entire strings but a trie would look at their individual characters
 - A trie is a tree where each node stores a bit indicating whether the string spelled out to this point is in the set

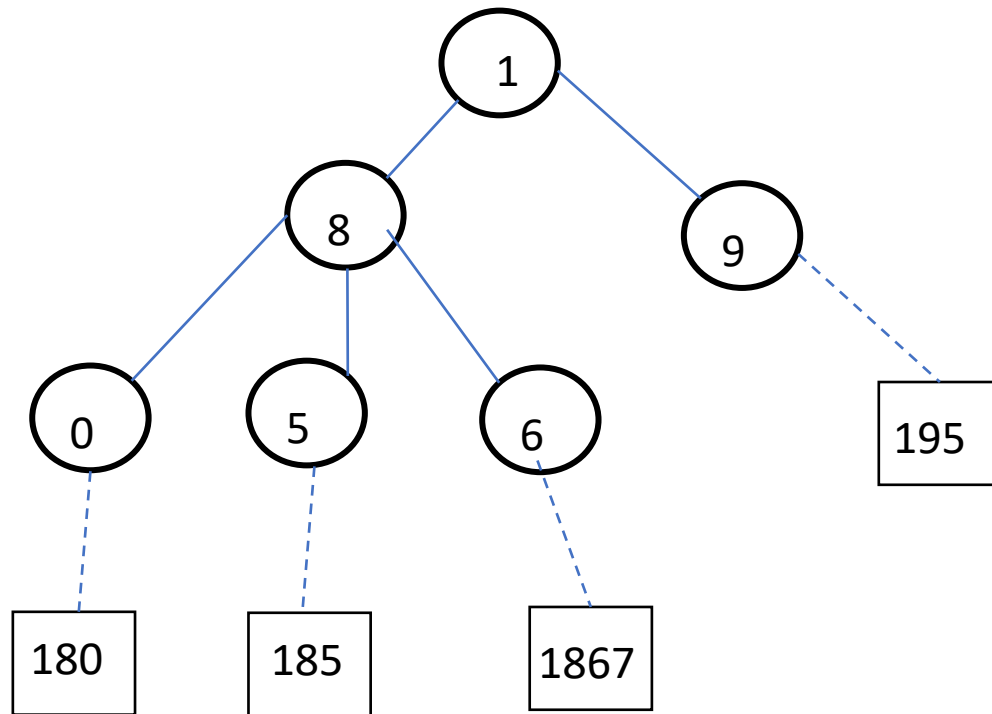
-Examples:



Data Structures and its Applications

TRIE Trees – Numeric Keys : Example2

- If the keys are numeric, there would be 10 pointers in a node.



Data Structures and its Applications

TRIE Trees – Numeric Keys : Example1

- If the keys are numeric, there would be 10 pointers in a node.
- Consider the SSN number as shown.

Name | **Social Security Number (SS#)**

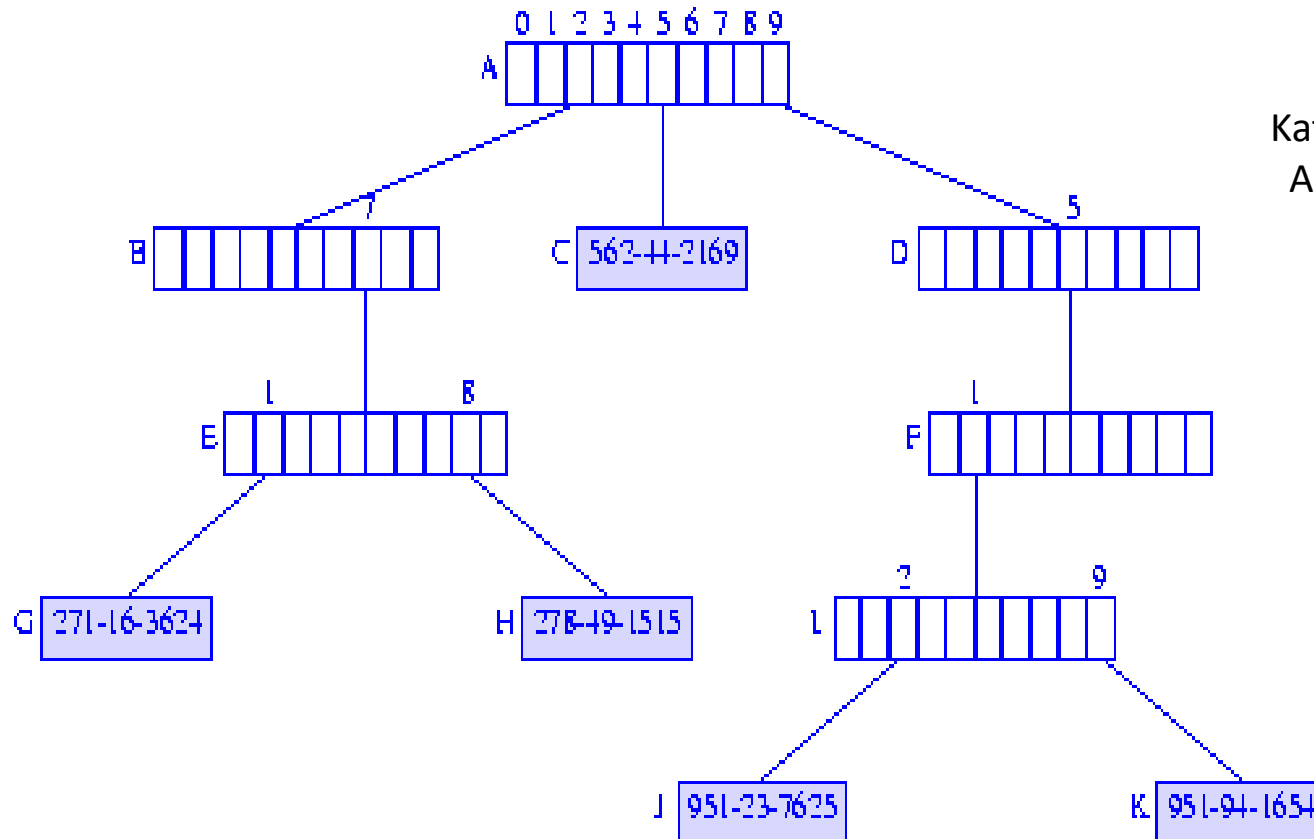
Jack | 951-94-1654

Jill | 562-44-2169

Bill | 271-16-3624

Kathy | 278-49-1515

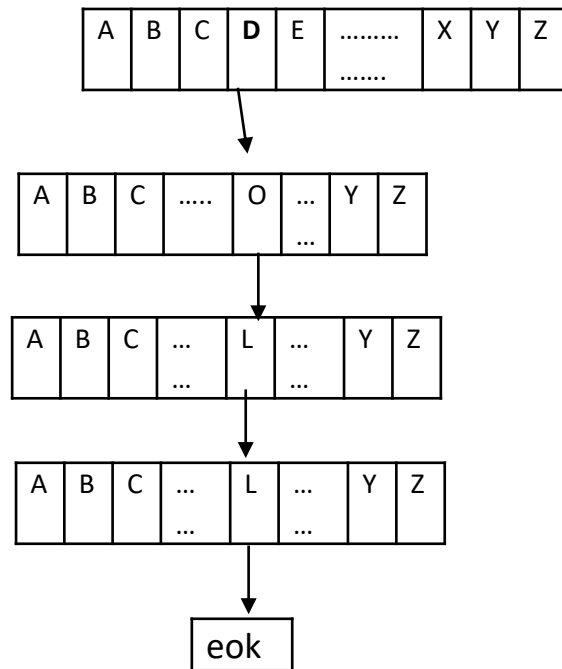
April | 951-23-7625



Data Structures and its Applications

TRIE Trees – An Introduction

- If the keys are Alphabetic, there would be 26 pointers.



Ex: The word DOLL has been stored as shown in the figure.

Data Structures and its Applications

TRIE Trees – An Introduction



- An extra pointer corresponding to eok (end of key) or a flag with each pointer indicating that it point to a record rather than to a tree node. (normally \$ symbol is used).
- A pointer in the node is associated with a particular symbol value based on its position in the node.
 - First pointer corresponds to the lowest value.
 - Second pointer to the second lowest and so forth.
- This way of implementation of a digital search tree is called a **TRIE** tree.
- The word **TRIE** is extracted from re**trie**val word.

- Tries are extremely special and useful data-structure that are based on the *prefix of a string*.
- Strings are stored in a top to bottom manner on the basis of their prefix in a TRIE.
- All prefixes of length 1 are stored at until level 1, all prefixes of length 2 are sorted at until level 2 and so on.

Suffix Trie:

- Suffix Trie is a space-efficient data structure to store a string that allows many kinds of queries to be answered quickly.
- Example:
Text is “**banana**\" data-bbox="40 578 728 801"/>

- A Trivial Algorithm for building a suffix tree.
 - Step1 : Generate all suffixes of a given text
 - Step2: Consider all suffixes as individual words and build a compressed trie.

- Example1:

Text is “**banana**\" data-bbox="95 455 418 539"/>

Following are the suffixes of Text

“**banana**\" data-bbox="82 632 190 670"/>

“**anana**\" data-bbox="91 676 188 714"/>

“**nana**\" data-bbox="100 720 188 758"/>

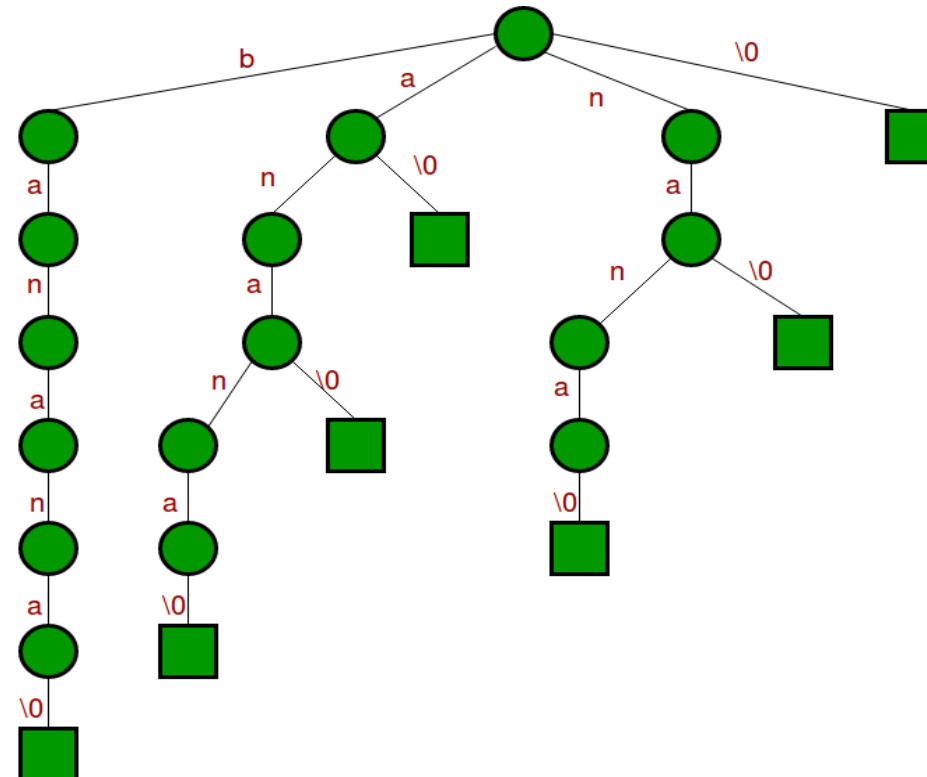
“**ana**\" data-bbox="108 764 186 802"/>

“**na**\" data-bbox="118 808 186 847"/>

“**a**\" data-bbox="128 852 183 890"/>

“**/**\" data-bbox="138 895 183 936"/>

Suffix trie




Data Structures and its Applications

Suffix Trie – Building



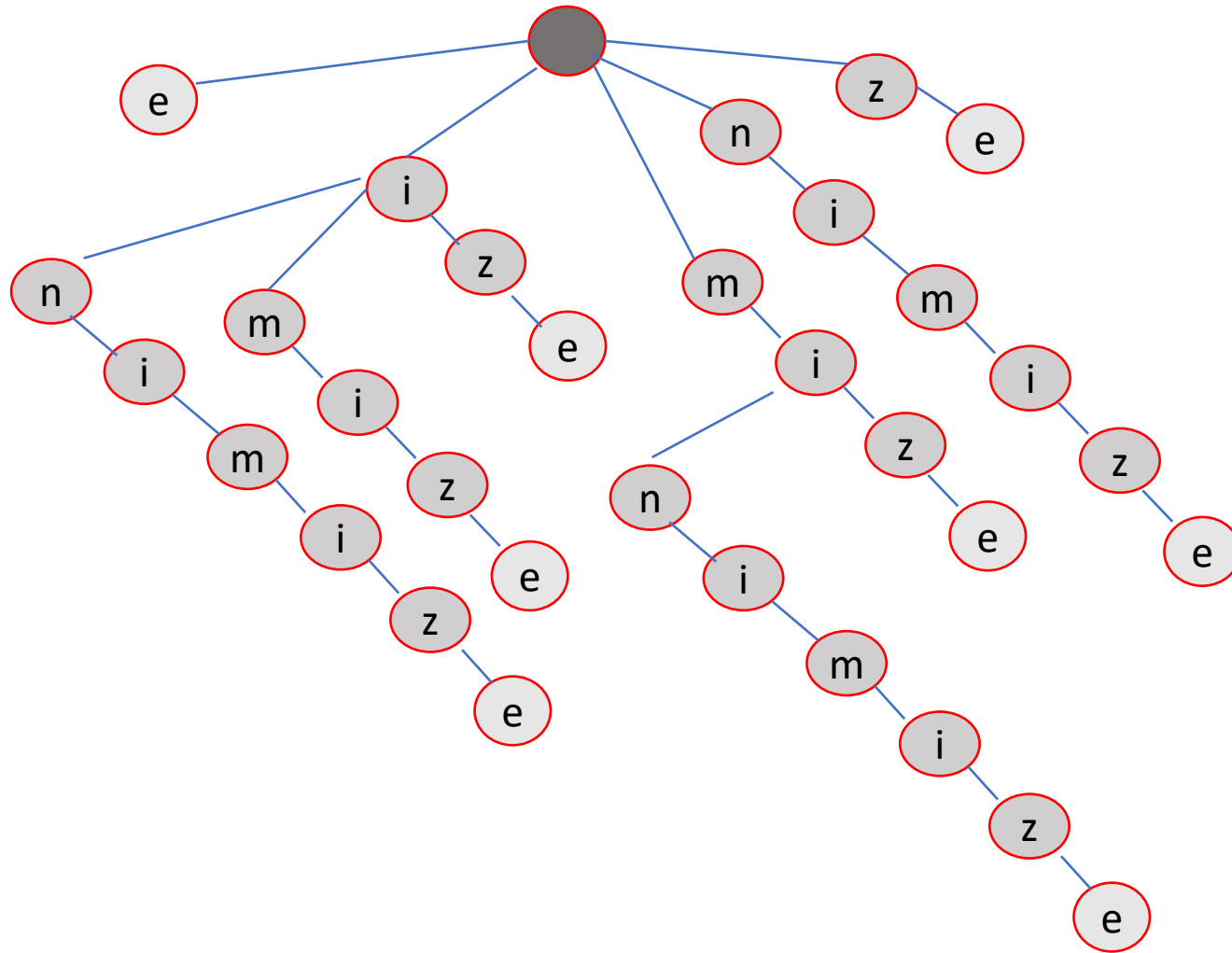
Example 2 : Generate the suffix trie for the word **minimize**

Step1. Generate all the suffixes of the word **minimize**.

 e
ze
ize
mize
imize
nimize
inimize
minimize

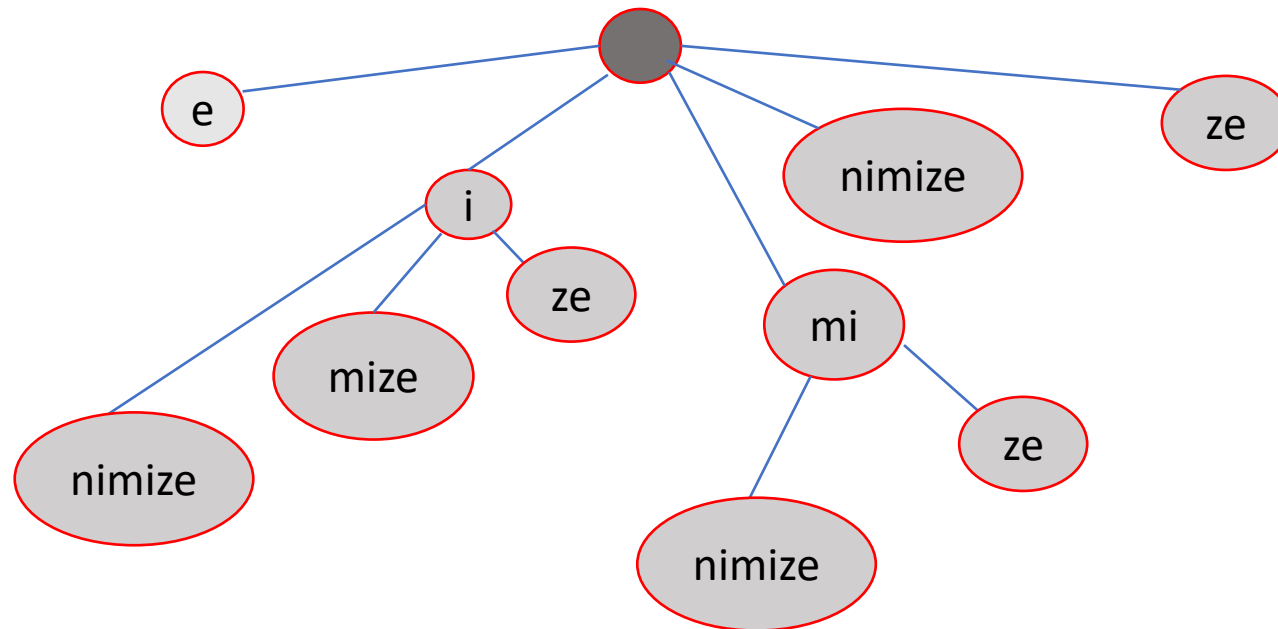
S - set of strings to
include in the suffix trie.

Suffix Trie – Building - for the word minimize



e
ze
ize
mize
imize
nimize
inimize
minimize

S - set of strings to include in the suffix trie

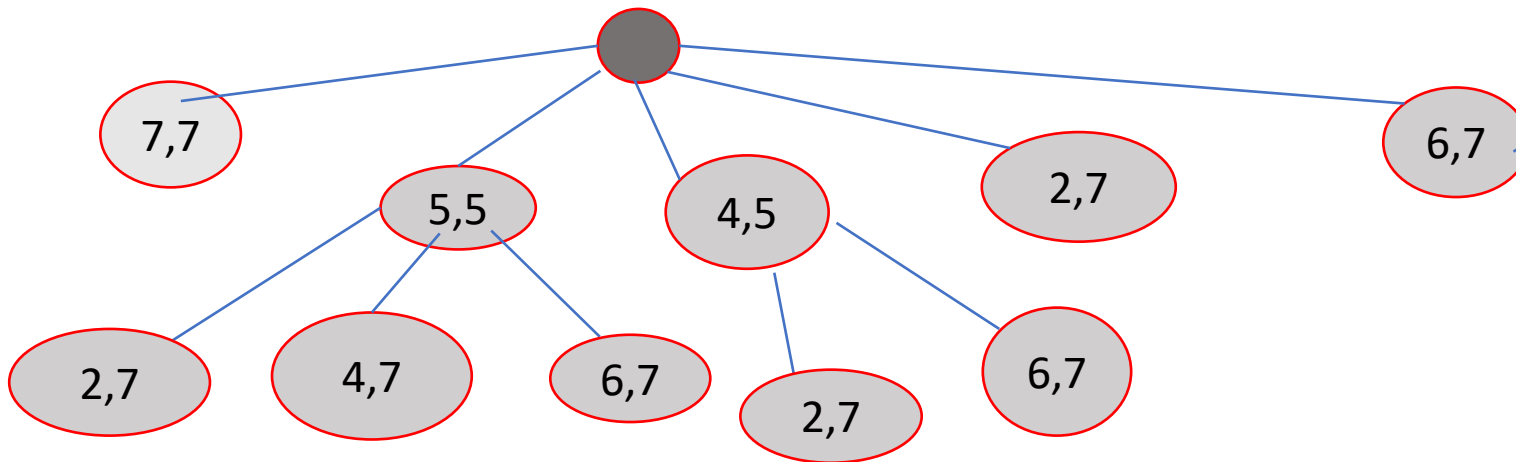


Data Structures and its Applications

Suffix Trie – Compressed Trie – using numbers

- Representation of Compressed trie using numbers - (Indexes)
- The indexes of the word is ...

0	1	2	3	4	5	6	7
m	i	n	i	m	i	z	e



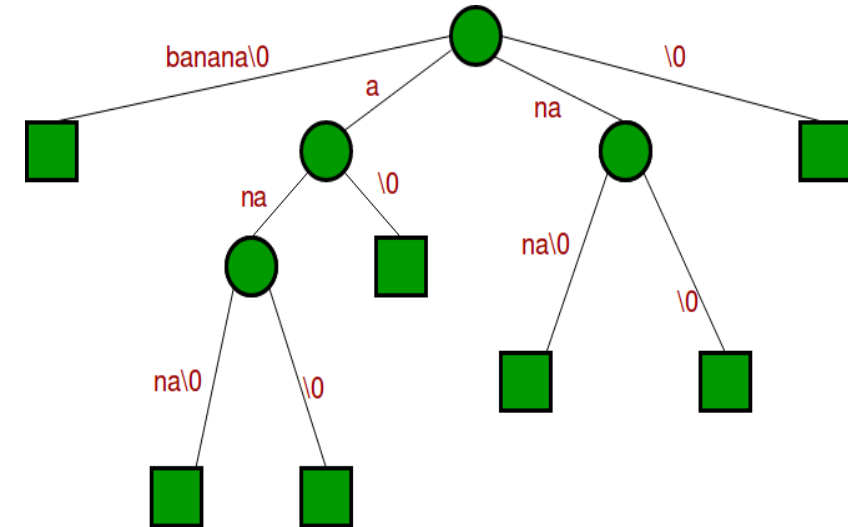
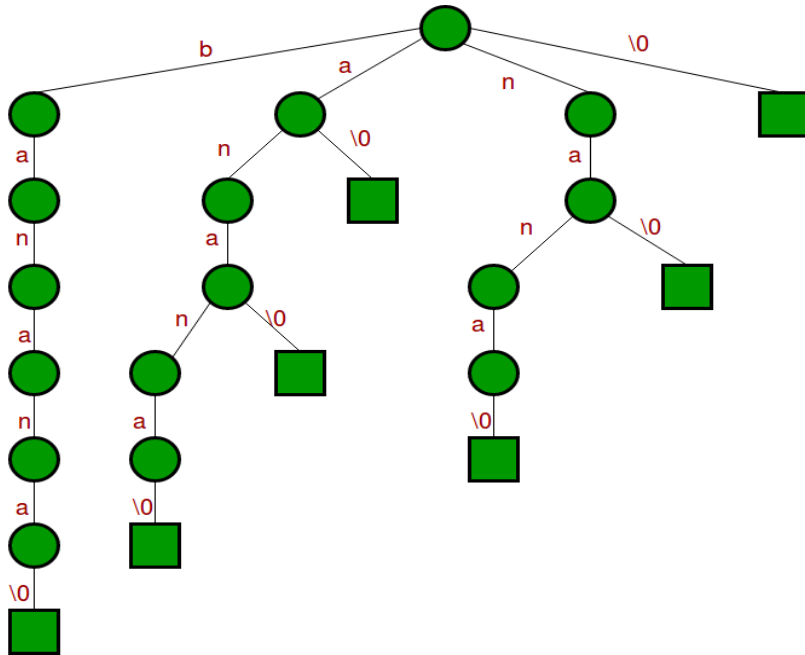
- A simple data structure for string searching
- It's a compressed Trie Tree
- Allow many fast implementations of many important string operations
- Properties of a suffix trees:
 - ✓ A suffix tree for a text X of size n from an alphabet of size d .
 - ✓ Stores all the $n(n-1)$ suffixes of X .
 - ✓ Supports arbitrary pattern matching and prefix matching queries

Example – Banana\$

Data Structures and its Applications

Suffix Trees – Introduction

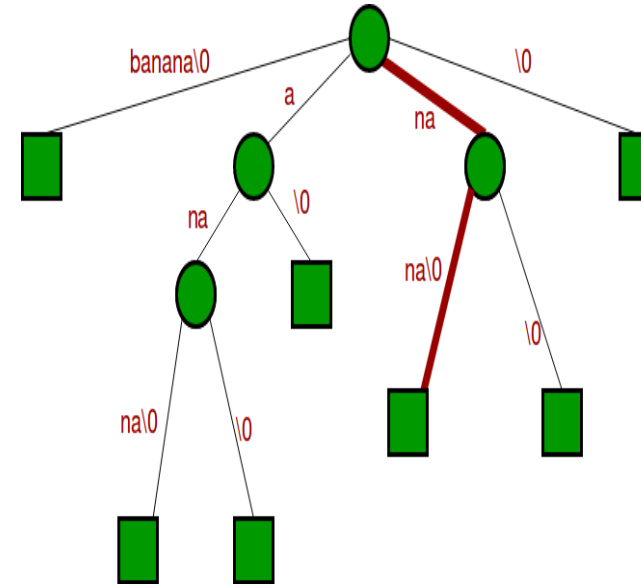
- Join chains of single nodes, to get the following compressed trie, which is the Suffix tree for given text “banana\0”



Data Structures and its Applications

Search for a substring in a Suffix Tree

- 1) Starting from the first character of the pattern and root of Suffix Tree, do following for every character.
 - i) For the current character of pattern, if there is an edge from the current node of suffix tree, follow the edge.
 - ii) If there is no edge, print “pattern doesn’t exist in text” and return.
- 2) If all characters of pattern have been processed, i.e., there is a path from root for characters of the given pattern, then print “Pattern found”.



Data Structures and its Applications

TRIE Trees – Applications, advantages and disadvantages



Applications:

- English dictionary
- Predictive text
- Auto-complete dictionary found on Mobile phones and other gadgets.

Advantages:

- Faster than BST
- Printing of all the strings in the alphabetical order easily.
- Prefix search can be done (Auto complete).

Disadvantages:

- Need for a lot of memory to store the strings,
- Storing of too many node pointers.



THANK YOU

V R BADRI PRASAD

Department of Computer Science & Engineering

badriprasad@pes.edu



Data Structures and its Applications

V R BADRI PRASAD

Department of Computer Science & Engineering

DATA STRUCTURES AND ITS APPLICATIONS

Implementation of TRIE Trees

V R BADRI PRASAD

Department of Computer Science & Engineering

Structure of a node in a TRIE Tree :

- A node of a TRIE tree is represented as shown below.
- One field for each alphabet(A – Z), 26 columns.
- Each column is a pointer to another TRIE node or carries NULL and
- One field for end of word (key).

A	B	C	D	E	F	W	X	Y	Z
F1	F2	F3	F4	F5	F6	F23	F24	F25	F26
End of Word / (eok)										

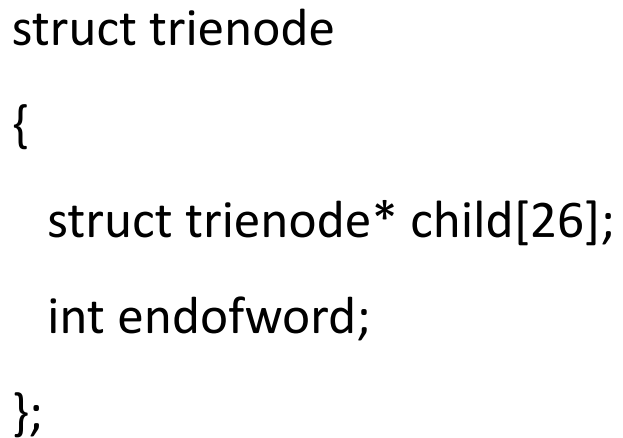
Address of the next node (reference for us)

Field number – for user's reference no field is created , No memory is allocated

End of word / key field

```
struct trienode
{
    struct trienode* child[26];
    int endofword;
};
```

TRIE Trees – Implementation



A	B	C	D	E	F	W	X	Y	Z	Address of the next node (reference for us)
F1	F2	F3	F4	F5	F6	F23	F24	F25	F26	Field number
End of Word / (eok - \$)											End of word / key field

Creation of a node in a TRIE Tree using malloc() function.

```
struct trienode *getnode()
```

```
{
```

```
    int i;
```

```
    struct trienode *temp;
```

```
    temp=(struct trienode*)(malloc(sizeof(struct trienode)));
```

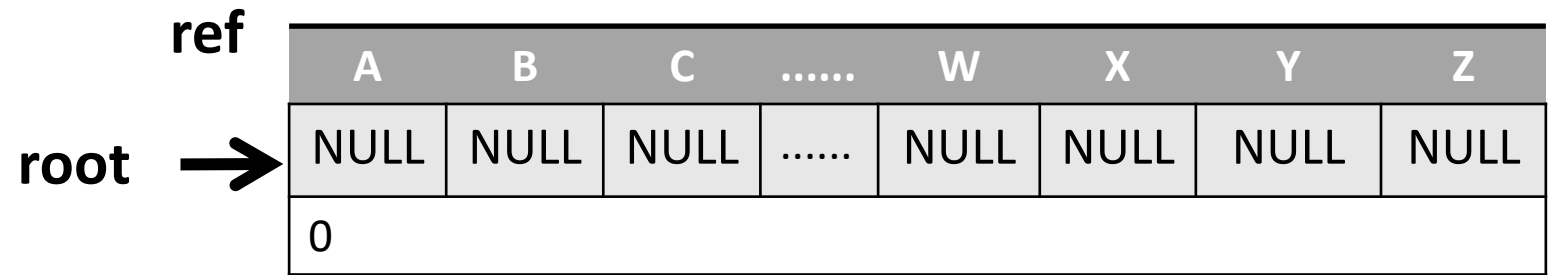
```
    for(i=0;i<26;i++)
```

```
        temp->child[i]=NULL;    // Initialize all the fields to NULL.
```

```
    temp->endofword=0;    // Initialize endofword to 0.
```

```
    return temp;
```

```
}
```



root=getnode();

Insertion of a node in a TRIE Tree

```
void insert(struct trienode *root, char *key)
{
    struct trienode *curr;
    int i,index;

    curr=root;
    for(i=0;key[i]!='\0';i++)
    {
        index=key[i]-'a';
        if(curr->child[index]==NULL)
            curr->child[index]=getnode();
        curr=curr->child[index];
    }
    curr->endofword=1;
}
```



Insertion of a node in a TRIE Tree

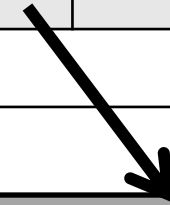
On function call insert, the given string “HELLO” is inserted into the TRIE tree as shown below.

```
insert(root,key);    root
```

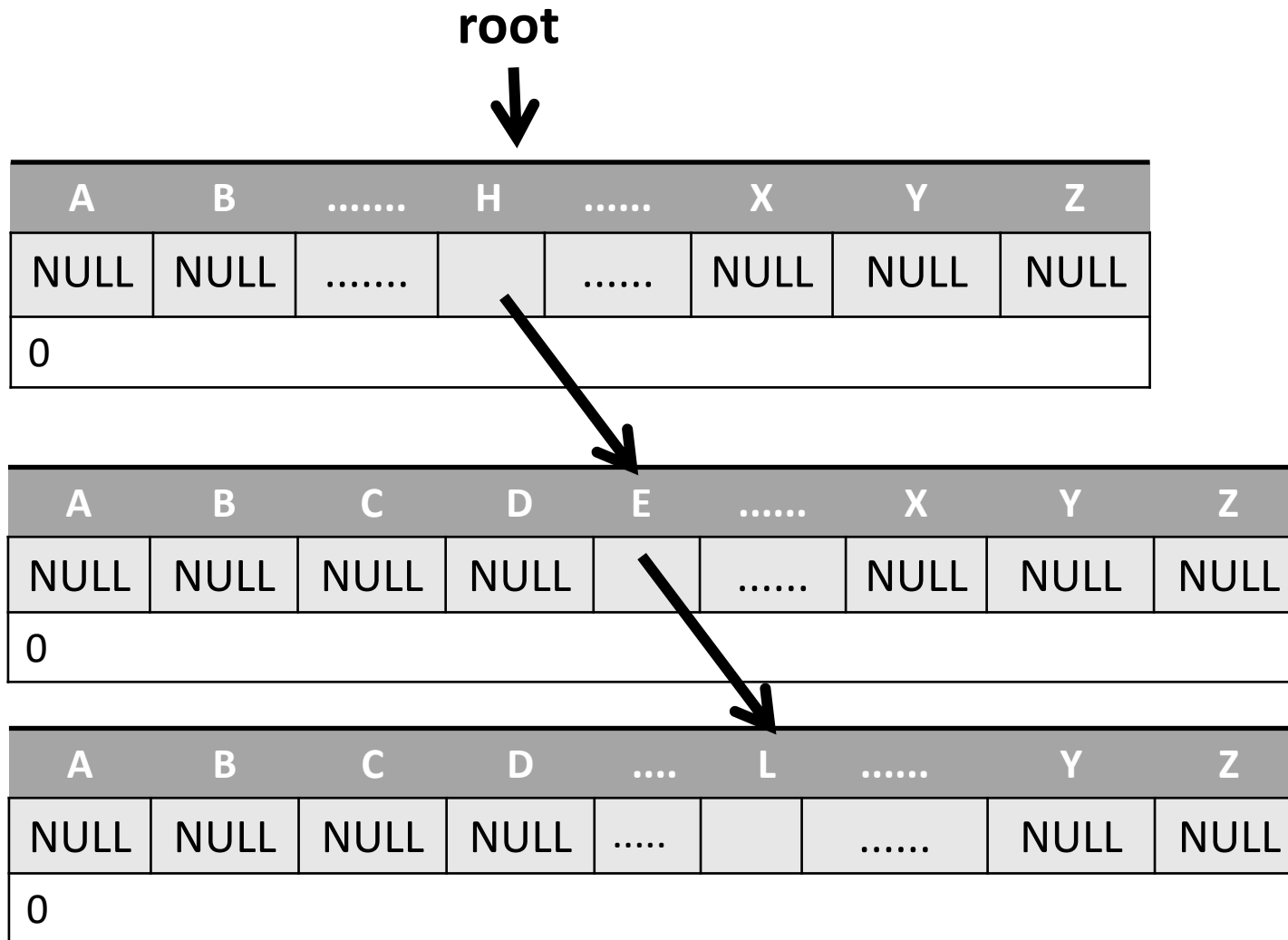


A	B	H	X	Y	Z
NULL	NULL	NULL	NULL	NULL

0



A	B	C	D	E	X	Y	Z
NULL	NULL	NULL	NULL		NULL	NULL	NULL
0								





THANK YOU

V R BADRI PRASAD

Department of Computer Science & Engineering

badriprasad@pes.edu



Data Structures and its Applications

V R BADRI PRASAD

Department of Computer Science & Engineering

DATA STRUCTURES AND ITS APPLICATIONS

Implementation of TRIE Trees :

- Display Operation
- Deletion Operation
- Search Operation

V R BADRI PRASAD

Department of Computer Science & Engineering

```
void display(struct trienode *curr)
{
    int i,j;
    for(i=0;i<255;i++)
    {
        if(curr->child[i]!=NULL)
        {
            word[length++]=i;
            if(curr->child[i]->endofword==1)//if end of word
            {
                printf("\n");
                for(j=0;j<length;j++)
                    printf("%c",word[j]);
            }
            display(curr->child[i]);
        }
    }
    length--;
    return ;
}
```

```
void delete_trie(struct trienode *root, char *key)
{
    int i,index,k;
    struct trienode *curr;
    struct stack x;
    curr=root;

    for(i=0;key[i]!='\0';i++)
    {
        index=key[i];
        if(curr->child[index]==NULL)
        {
            printf("The word not found..\n");
            return;
        }
        push(curr,index);
        curr=curr->child[index];
    }
    curr->endofword=0;
    push(curr,-1);
```



```
while(1)
{
    x=pop();
    if(x.index!=-1)
        x.m->child[x.index]=NULL;
    if(x.m==root)//if root
        break;
    k=check(x.m);
    if((k>=1) || (x.m->endofword==1))
        break;
    else
        free(x.m);
}
return;
}
```

```
int search(struct trienode * root,char *key)
{

    int i,index;
    struct trienode *curr;
    curr=root;

    for(i=0;key[i]!='\0';i++)
    {
        index=key[i];

        if(curr->child[index]==NULL)
            return 0;
        curr=curr->child[index];
    }
    if(curr->endofword==1)
        return 1;
    return 0;
}
```



THANK YOU

V R BADRI PRASAD

Department of Computer Science & Engineering

badriprasad@pes.edu



Data Structures and its Applications

V R BADRI PRASAD

Department of Computer Science & Engineering

DATA STRUCTURES AND ITS APPLICATIONS

Introduction to Hashing :

- Hash Function
- Hash Table
- Creation of Hash Table

V R BADRI PRASAD

Department of Computer Science & Engineering

- Sequential search algorithm takes lot of time for searching $O(n)$.
- BST improves performance by $\log(n)$.
 - To achieve this speed, BST should be balanced.

Consider an employee database of 10000 records.

Using

- Linked list would take $O(n)$ time.
- Using Balanced BST would take $O(\log n)$ time.
- However, using arrays, would take $O(1)$ time will lead to a lot of space wasted.
- Is there a way to get the data retrieved with $O(1)$ time without memory being wasted?
- The solution is HASHING.

- Implementing Dictionaries
- Takes equal time for operation
- Efficient techniques for retrieval of data would be one that takes less number of comparisons.
- A **hash table**, or a **hash map**, is a data structure that associates keys (names) with values (attributes).
- Use hash function to map keys to hash tables.
- Key is stored at a memory location , the address of the location is computed using hash function.

Example:

- Consider a key 496000. Suppose the hash table has 10 memory locations, then the key is stored at location which has an address computed using hash function $\text{key} \bmod 10$.

Address(index) is : $496005 \bmod 10 = 5$.

The data 496000 is stored at location with index five.

Hashing – Hash Function and Hash Table

- A good hash function is one that distributes keys evenly among all slots / index (locations).
- Design of a hash function is an art more than science.



Hash Table	
Index / hash	DATA
0	15
1	46
2	72
3	18
4	34

- Consider key elements as 34, 46, 72, 15, 18, 26, 93
- Hash function is **key mod 5**.
- Index value for the keys are generated using the given hash function
- $34 \bmod 5 = 4$, 34 is stored at index 4.
- $46 \bmod 5 = 1$, 46 is stored at index 1.
- $72 \bmod 5 = 2$, 72 is stored at index 2.
- $15 \bmod 5 = 0$, 15 is stored at index 0.
- $18 \bmod 5 = 3$, 18 is stored at index 3.
- This technique is called **closed hashing**

Hash Table	
Index	DATA
0	15
1	46
2	72
3	18
4	34

- Consider key elements as 34, 46, 72, 15, 18, 26, 93
- Hash function is **key mod 5**.
- Index value for the keys are generated using the given hash function
 - $34 \bmod 5 = 4$, 34 is stored at index 4.
 - $46 \bmod 5 = 1$, 46 is stored at index 1.
 - $72 \bmod 5 = 2$, 72 is stored at index 2.
 - $15 \bmod 5 = 0$, 15 is stored at index 0.
 - $18 \bmod 5 = 3$, 18 is stored at index 3.
 - **$26 \bmod 5 = 1$, but location 1 is already occupied.**

Hence results in clash / collision.

Also, the capacity of the hash table is full.

Hence, 26 cannot be stored in the hash table.

It is true for the next data item 93 as location with index 3 is also occupied.

Which results in clash.

The problem can be resolved by

- Increasing the Memory Capacity.
- Overcoming Collision using
 - Open Addressing / Separate Chaining
 - Closed Addressing :
 - Linear Probing
 - Quadratic Probing
 - Double Hashing

Initially Hash Table contains all 'NULL' values in the address field of the hash table.

- Consider key elements as 34, 46, 72, 15, 18
- Hash function is **key mod 5**.

Hash Table	
Index	address
0	NULL
1	NULL
2	NULL
3	NULL
4	NULL

- $34 \bmod 5 = 4$, 34 is stored at index 4.
- $46 \bmod 5 = 1$, 46 is stored at index 1.
- $72 \bmod 5 = 2$, 72 is stored at index 2.
- $15 \bmod 5 = 0$, 15 is stored at index 0.
- $18 \bmod 5 = 3$, 18 is stored at index 3.

Initially Hash Table contains all 'NULL' values in the address field of the hash table.



- Consider key elements as 34, 46, 72, 15, 18
- Hash function is **key mod 5**.

Hash Table	
Index	address
0	NULL
1	NULL
2	NULL
3	NULL
4	<div><div></div><div>→ 34</div></div>

- **$34 \bmod 5 = 4$, 34 is stored at index 4.**
- $46 \bmod 5 = 1$, 46 is stored at index 1.
- $72 \bmod 5 = 2$, 72 is stored at index 2.
- $15 \bmod 5 = 0$, 15 is stored at index 0.
- $18 \bmod 5 = 3$, 18 is stored at index 3.

Initially Hash Table contains all 'NULL' values in the address field of the hash table.

- Consider key elements as 34, 46, 72, 15, 18
- Hash function is **key mod 5**.

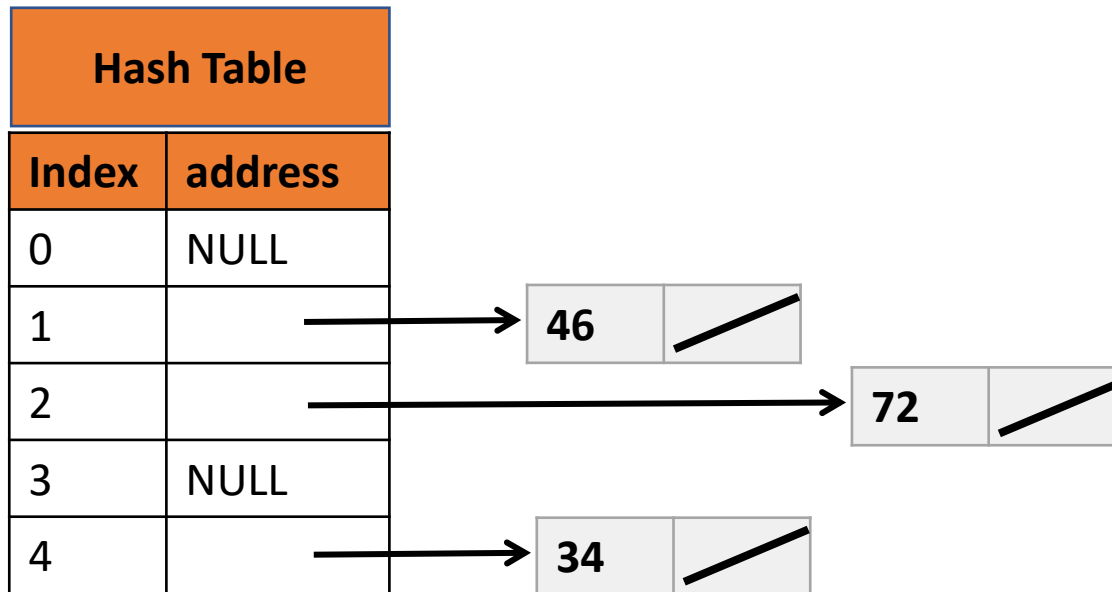
Hash Table	
Index	address
0	NULL
1	
2	NULL
3	NULL
4	

- $34 \bmod 5 = 4$, 34 is stored at index 4.
- **$46 \bmod 5 = 1$, 46 is stored at index 1.**
- $72 \bmod 5 = 2$, 72 is stored at index 2.
- $15 \bmod 5 = 0$, 15 is stored at index 0.
- $18 \bmod 5 = 3$, 18 is stored at index 3.

Initially Hash Table contains all 'NULL' values in the address field of the hash table.





- Consider key elements as 34, 46, 72, 15, 18
- Hash function is **key mod 5**.

- $34 \bmod 5 = 4$, 34 is stored at index 4.
- $46 \bmod 5 = 1$, 46 is stored at index 1.
- **$72 \bmod 5 = 2$, 72 is stored at index 2.**
- $15 \bmod 5 = 0$, 15 is stored at index 0.
- $18 \bmod 5 = 3$, 18 is stored at index 3.



Initially Hash Table contains all 'NULL' values in the address field of the hash table.

- Consider key elements as 34, 46, 72, 15, 18
- Hash function is **key mod 5**.

Hash Table	
Index	address
0	
1	
2	
3	NULL
4	

15

46

72

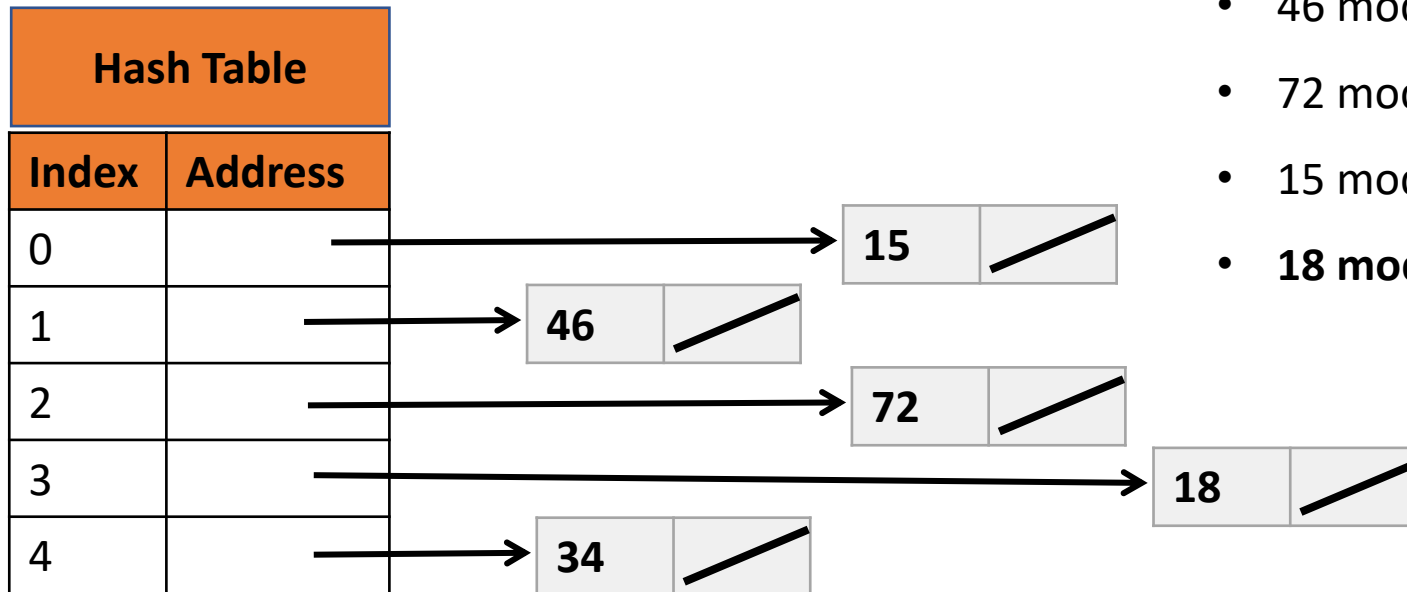
34

- $34 \bmod 5 = 4$, 34 is stored at index 4.
- $46 \bmod 5 = 1$, 46 is stored at index 1.
- $72 \bmod 5 = 2$, 72 is stored at index 2.
- **$15 \bmod 5 = 0$, 15 is stored at index 0.**
- $18 \bmod 5 = 3$, 18 is stored at index 3.

Initially Hash Table contains all 'NULL' values in the address field of the hash table.

- Consider key elements as 34, 46, 72, 15, 18
- Hash function is **key mod 5**.

- $34 \bmod 5 = 4$, 34 is stored at index 4.
- $46 \bmod 5 = 1$, 46 is stored at index 1.
- $72 \bmod 5 = 2$, 72 is stored at index 2.
- $15 \bmod 5 = 0$, 15 is stored at index 0.
- $18 \bmod 5 = 3$, 18 is stored at index 3.**





THANK YOU

V R BADRI PRASAD

Department of Computer Science & Engineering

badriprasad@pes.edu



Data Structures and its Applications

V R BADRI PRASAD

Department of Computer Science & Engineering

DATA STRUCTURES AND ITS APPLICATIONS

Hashing :

- Insert Operation
- Display Operation

V R BADRI PRASAD

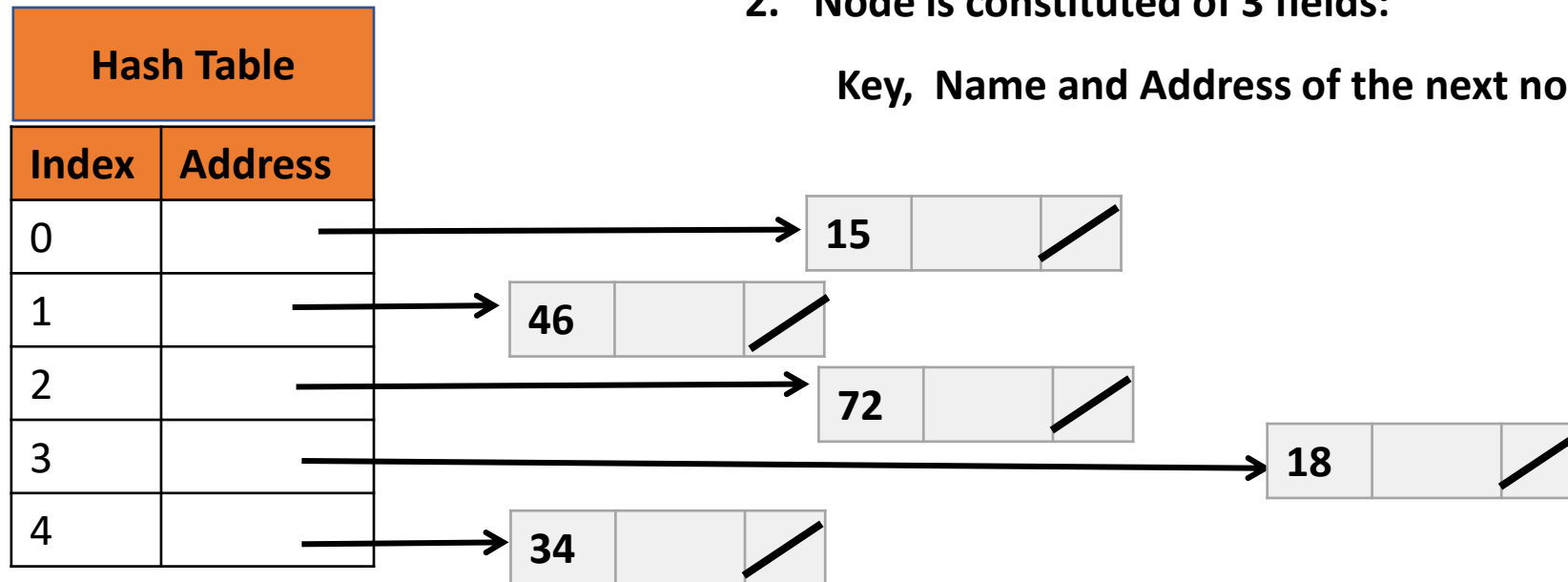
Department of Computer Science & Engineering

- Consider key elements as 34, 46, 72, 15, 18
- Hash function is **key mod 5**.

1. Hash Table contains Index, Address fields.

2. Node is constituted of 3 fields:

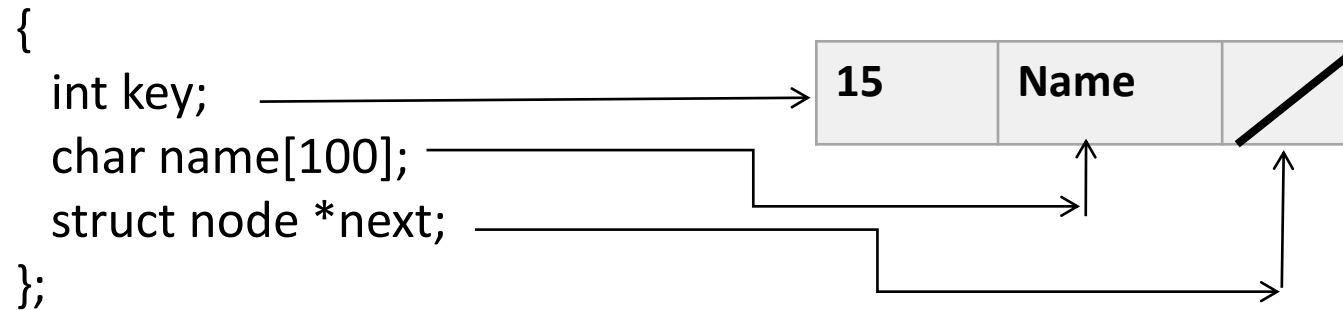
Key, Name and Address of the next node.



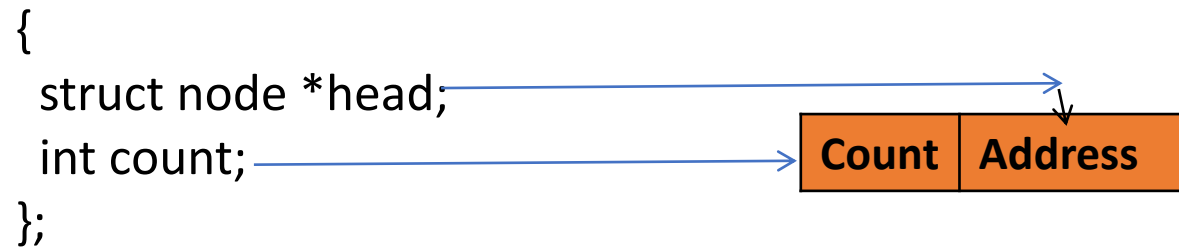
Data Structures and its Applications

Hashing : Node Creation – Separate Chaining

struct node



struct hash



```
void insert_to_hash(struct hash *ht, int size, int key, char* name)
{
    int index;
    struct node *temp;
```


```
// Create a node and store the starting address in temp variable.
```

```
temp=(struct node*)(malloc(sizeof(struct node)));
temp->key=key;
strcpy(temp->name,name);
temp->next=NULL;
```

// Insert node at the beginning of Singly linked list as shown in the figure.

```
index=key%size;  
temp->next=ht[index].head;  
ht[index].head=temp;  
ht[index].count++;  
}
```

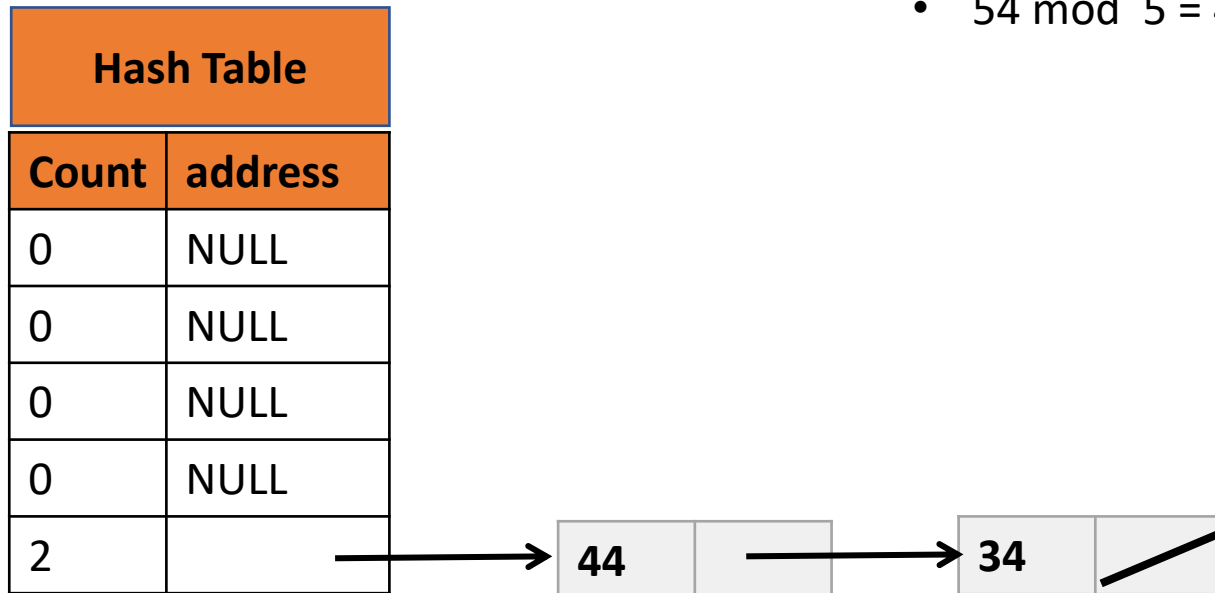
- $34 \bmod 5 = 4$, 34 is stored at index 4.
- $44 \bmod 5 = 4$, 44 is stored at index 4.
- $54 \bmod 5 = 4$, 54 is stored at index 4.

Hash Table	
Count	address
0	NULL
0	NULL
0	NULL
0	NULL
1	

// Insert node at the beginning of Singly linked list as shown in the figure.

```
index=key%size;  
temp->next=ht[index].head;  
ht[index].head=temp;  
ht[index].count++;  
}
```

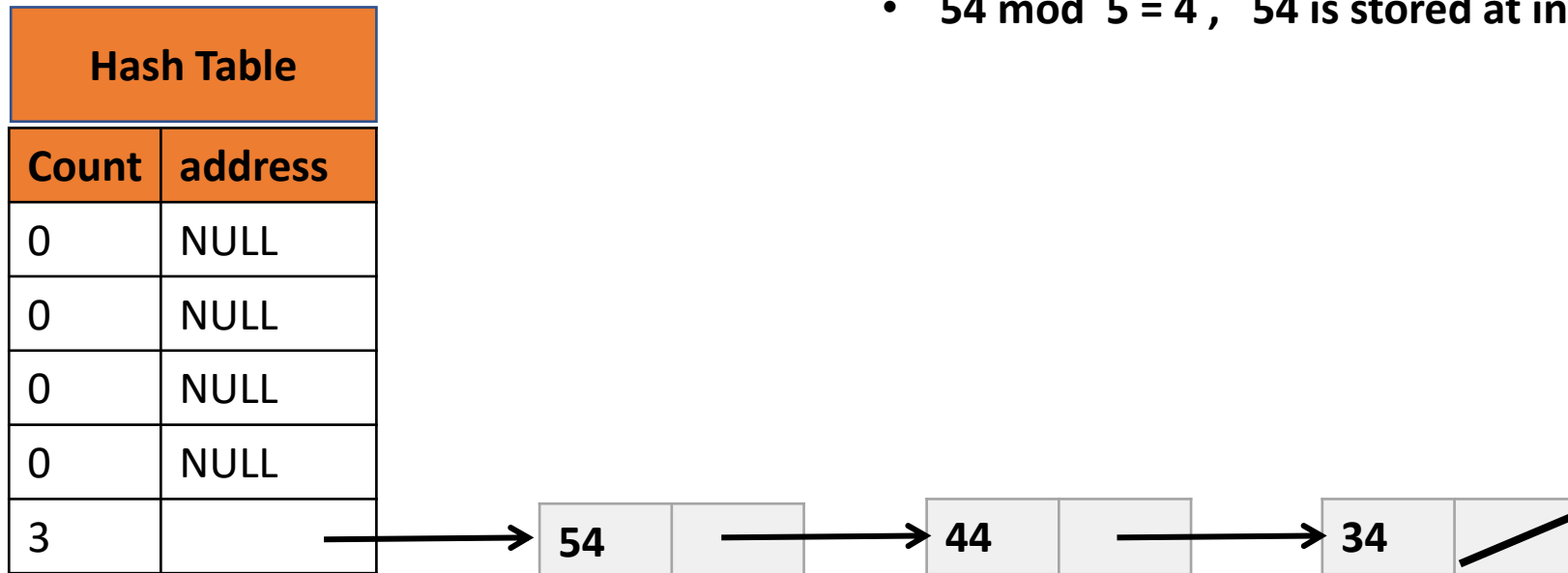
- $34 \bmod 5 = 4$, 34 is stored at index 4.
- **$44 \bmod 5 = 4$, 44 is stored at index 4.**
- $54 \bmod 5 = 4$, 54 is stored at index 4.



// Insert node at the beginning of Singly linked list as shown in the figure.

```
index=key%size;  
temp->next=ht[index].head;  
ht[index].head=temp;  
ht[index].count++;  
}
```

- $34 \bmod 5 = 4$, 34 is stored at index 4.
- $44 \bmod 5 = 4$, 44 is stored at index 4.
- $54 \bmod 5 = 4$, 54 is stored at index 4.



```
void display(struct hash* ht, int size)
{
    int i;
    struct node *temp;
    printf("\n");
    for(i=0;i<size;i++)
    {
        printf("%d : ",i)
        if(ht[i].head != NULL)
        {
            temp=ht[i].head;
            while(temp!=NULL)
            {
                printf("%d",temp->key);
                printf("%s->",temp->name);
                temp=temp->next;
            }
        }
        printf("\n");
    }
}
```

Count	address
0	NULL
0	NULL
0	NULL
0	NULL
3	



Display Output :

```
0 :
1 :
2 :
3 :
4 : 54 -> 44 -> 34
```



THANK YOU

V R BADRI PRASAD

Department of Computer Science & Engineering

badriprasad@pes.edu



Data Structures and its Applications

V R BADRI PRASAD

Department of Computer Science & Engineering

DATA STRUCTURES AND ITS APPLICATIONS

Hashing – Closed Hashing – Linear Probing

- Insert Operation
- Display Operation

V R BADRI PRASAD

Department of Computer Science & Engineering

- Consider key elements as 34, 46, 72, 15, 18
- The data is stored in the hash table as shown .

Hash Table	
Index	Data
0	15
1	46
2	72
3	18
4	34

- $34 \bmod 5 = 4$, 34 is stored at index 4.
- $46 \bmod 5 = 1$, 46 is stored at index 1.
- $72 \bmod 5 = 2$, 72 is stored at index 2.
- $15 \bmod 5 = 0$, 15 is stored at index 0.
- $18 \bmod 5 = 3$, 18 is stored at index 3.

Say if 57 is to be stored, then

Compute hash / index value : **$57 \% 5 = 2$** .

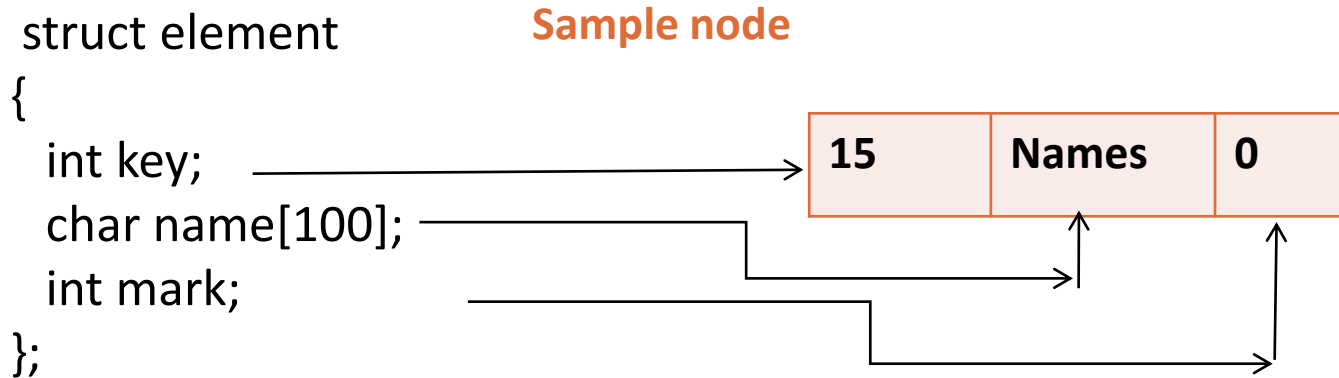
However, it exceeds the capacity of the hash table. Hence it cannot be stored.

To overcome this problem, increase the size of the hash table.

Data Structures and its Applications


Hashing : Linear Probing - Node Creation

The method is same as separate chaining for creation if the node.



Allocation of memory

```
hashtable = (struct element *) (malloc(tableSz * sizeof(struct element)));
```



Key	Name	Mark
		0
....	0
		0

Data Structures and its Applications

Hashing – Linear Probing : Insert Operation

Consider key elements as 34, 46, 72, 15, 18

- $34 \bmod 5 = 4$, 34 is stored at index 4.
- $46 \bmod 5 = 1$, 46 is stored at index 1

Let table size be 5.

if count == 5 then Table is Full. Cannot insert

The code is as follows:

```
void insert_to_hash(struct element *ht, int size, int key, char *name, int *count)
{
    int index;
    if(size==*count)
    {
        printf("Table full.. cannot insert\n");
        return;
    }
}
```



Key	Name	Mark
--	--	0
46	DEF	1
--	--	0
--	--	0
34	MNP	1

Data Structures and its Applications

Hashing – Linear Probing : Insert Operation

Consider key elements as 34, 46, 71, 15, 18

- $34 \bmod 5 = 4$, 34 is stored at index 4.
- $46 \bmod 5 = 1$, 46 is stored at index 1

Next number is 71.

Index is $71 \% 5 = 1$.


Since , index '1' is non empty location, search for first empty location in the sequence.

I.e., location with index value $1+1 = 2$.

Now, Location 2 is empty.

- Store the key value in that location.
- Mark location 2 as 1.

The code for the same is as follows:



Array Index	Key	Name	Mark
0	--	--	0
1	46	DEF	1
2	71	ABC	0
3	--	--	0
4	34	MNP	1

Data Structures and its Applications

Hashing – Linear probing : Insert Operation



Find the index value using the statement

$\text{index} = \text{key} \% \text{size};$

Find the first empty location in the hash table and store the data sent from the main program

```
while(ht[index].mark !=0)
```

```
    index=(index+1)%size;
```

```
    ht[index].key=key;
```

```
    strcpy(ht[index].name,name);
```

```
    ht[index].mark=1;
```

Increment count by 1

```
    (*count)++;
```

```
    return;
```

```
}
```



Key	Name	Mark
15	ABC	1
46	DEF	1
71	GHI	1
18	JKL	1
34	MNP	1

Other elements are stored in the memory as shown.
Mark field is set to 1.

An element is to be removed from the hash table. How to delete..?

Data Structures and its Applications

Hashing – Linear Probing – Deletion of an element



```
void delete_from_hash(struct element *ht,int size,int key, int *count)
```

```
{  
    int i,index;  
    printf("count = %d\n",*count);
```

```
// If Table is empty display table is empty.
```

```
if(*count==0)  
{  
    printf("table empty..\n");  
    return;  
}
```

Key	Name	Mark
15	ABC	0
46	DEF	0
71	GHI	0
18	JKL	0
34	MNP	0

```
// if Mark is '0', indicates the element is not present in the hash table
```

```
// Otherwise:
```

Data Structures and its Applications

Hashing – Linear Probing – Deletion of an element

// Search for the element to be deleted

```
index = key % size;
i=0;
while(i<*count)
{
    if (ht[index].mark==1)
    { // indicates element is present
        if(ht[index].key==key) // if found
        { ht[index].mark=0; // Delete
            (*count)--;
            return;
        }
        i++; }
    index=(index+1)%size; // search for the element in the
                          //consecutive memory location
}

printf("key not found..");
return;
}
```

Key	Name	Mark
15	ABC	1
46	DEF	1
71	GHI	1
18	JKL	1
34	MNP	1

Data Structures and its Applications

Hashing – Linear Probing – Deletion of an element

// Search for the element to be deleted

Ex: Let's delete 71.

First, search for 71 at location $71 \% 5 = 1$.

At memory location 1, check the value stored with the key value.

It is the same.

Now, Set the mark value to '0'.

This is deletion of an element from the hash table.

Key	Name	Mark
15	ABC	1
46	DEF	1
71	GHI	0
18	JKL	1
34	MNP	1



THANK YOU

V R BADRI PRASAD

Department of Computer Science & Engineering

badriprasad@pes.edu



Data Structures and its Applications

V R BADRI PRASAD

Department of Computer Science & Engineering

DATA STRUCTURES AND ITS APPLICATIONS

Quadratic Hashing : Insert, delete, search and Display operations

V R BADRI PRASAD

Department of Computer Science & Engineering

Data Structures and its Applications

Closed Hashing – Quadratic Probing

Hash Table	
Index	Data
0	-
1	34
2	46
3	72
4	-

Double hashing is collision resolving technique in open addressed hash tables.

Double hashing uses the idea of applying a second hash function to key when a collision occurs.

- $((34 \bmod 5) + 1^{**2}) \% 5 = 1$
- $((46 \bmod 5) + 1^{**2}) \% 5 = 2$
- $((72 \bmod 5) + 1^{**2}) \% 5 = 3$
- $((15 \bmod 5) + 1^{**2}) \% 5 = 1$. **There is a clash / collision**
- The next empty location is determined using the hash function with value $i=2$.
- **Index** = $((15 \bmod 5) + 2^{**2}) \% 5 = 0$.

Data Structures and its Applications

Closed Hashing – Quadratic Probing

Hash Table	
Index	Data
0	15
1	34
2	46
3	72
4	-

- Consider key elements as 34, 46, 72, 15, 18
- Hash Function is $hash = ((key \% size) + i * i) \% size$,
where $i=1,2,3,4,\dots$ The data is stored in the hash table as shown.

Since the hash value is computed using the above quadratic equation, the technique is called as quadratic hashing / probing

- $((34 \% 5) + 1^2) \% 5 = 1$
- $((46 \% 5) + 1^2) \% 5 = 2$
- $((72 \% 5) + 1^2) \% 5 = 3$
- $((15 \% 5) + 1^2) \% 5 = 1$. **There is a clash / collision**
- The next empty location is determined using the hash function with value $i=2$.
- Index** = $((15 \% 5) + 2^2) \% 5 = 0$.

Data Structures and its Applications

Closed Hashing – Quadratic Probing

Hash Table	
Index	Data
0	15
1	34
2	46
3	72
4	-

- Consider key elements as 34, 46, 72, 15, 18
- Hash Function is $hash = ((key \% size) + i * i) \% size$,
where $i=1,2,3,4,\dots$ The data is stored in the hash table as shown.

- $((34 \% 5) + 1^2) \% 5 = 1$
- $((46 \% 5) + 1^2) \% 5 = 2$
- $((72 \% 5) + 1^2) \% 5 = 3$
- $((15 \% 5) + 1^2) \% 5 = 1$
- $((18 \% 5) + 1^2) \% 5 = 4$

Store the number 18 in location 4.

Data Structures and its Applications

Closed Hashing – Quadratic Probing

Hash Table	
Index	Data
0	15
1	34
2	46
3	72
4	18

- Consider key elements as 34, 46, 72, 15, 18
- Hash Function is $hash = ((key \% size) + i * i) \% size$,
where $i=1,2,3,4,\dots$ The data is stored in the hash table as shown.
 - $((34 \% 5) + 1^2) \% 5 = 1$
 - $((46 \% 5) + 1^2) \% 5 = 2$
 - $((72 \% 5) + 1^2) \% 5 = 3$
 - $((15 \% 5) + 1^2) \% 5 = 1$
 - $((18 \% 5) + 1^2) \% 5 = 4$

Store the number 18 in location 4.

Data Structures and its Applications

Hashing – Quadratic Probing : Insert Operation

Let table size be 10.

if count == 10 then Table is Full. Cannot insert

The code is as follows:

```
void insert_to_hash(struct element *ht, int size, int key, char *name, int *count)
{
    int index;
    if(size==*count)
    {
        printf("Table full.. cannot insert\n");
        return;
    }
}
```

-Insert the key elements as 25, 85, 95 into the hash table.

Index	Key	Mark
0	15	1
1	34	1
2	46	1
3	72	1
4	18	1
5	-	0
6	-	0
7	-	0
8	-	0
9	-	0

-Insert the key elements as 25, 85, 95 into the hash table.

Next number is 25 and hash / index value is :

Index is $((25 \% 10) + 1 * 1) = 6$.

For 85, it is $((85 \% 10) + 1 * 1) = 6$ clash occurs. Then $i=2$,

The index / hash value = $((85 \% 10) + 2 * 2) \% 10 = 9$.

Instead of searching for consecutive locations which is empty, it is better to search locations that are empty which is determined using the above hash function.

Store 85 at location 9.

Index	Key	Mark
0	15	1
1	34	1
2	46	1
3	72	1
4	18	1
5	-	0
6	25	1
7	-	0
8	-	0
9	-	0

-Insert the key elements as 25, 85, 95 into the hash table.

Next number is 25 and hash / index value is :

Index is $((25 \% 10) + 1 * 1) = 6$.

For 85, it is $((85 \% 10) + 1 * 1) = 6$ clash occurs. Then $i=2$,

The index / hash value = $((85 \% 10) + 2 * 2) \% 10 = 9$.

Store 85 at location 9.

Next element is 95.

$((95 \% 10) + 1 * 1) \% 10 = 6$ clash.

$((95 \% 10) + 2 * 2) \% 10 = 9$ clash

$((95 \% 10) + 3 * 3) \% 10 = 4$ clash

$((95 \% 10) + 4 * 4) \% 10 = 1$ clash

$((95 \% 10) + 5 * 5) \% 10 = 0$ clash

Index	Key	Mark
0	15	1
1	34	1
2	46	1
3	72	1
4	18	1
5	-	0
6	25	1
7	-	0
8	-	0
9	85	1

Data Structures and its Applications

Hashing –Quadratic Probing : Insert Operation

$((95 \% 10) + 6 * 6) \% 10 = 1$ clash

$((95 \% 10) + 7 * 7) \% 10 = 4$ clash

$((95 \% 10) + 8 * 8) \% 10 = 9$ clash

$((95 \% 10) + 9 * 9) \% 10 = 4$ clash

$((95 \% 10) + 10 * 10) \% 10 = 6$ clash and so on...

Index	Key	Mark
0	15	1
1	34	1
2	46	1
3	72	1
4	18	1
5	-	0
6	25	1
7	-	0
8	-	0
9	85	1

Data Structures and its Applications

Hashing –Quadratic Probing : Insert Operation




Find the index value using the statement

$\text{index} = \text{key} \% \text{size};$

Find the first empty location in the hash table and store the data sent from the main program

```
for(h=1;h<=size;h++)
{
    If(ht[index].mark ==0)
    {
        ht[index].key=key;
        strcpy(ht[index].name,name);
        ht[index].mark=1;
        return;
    }
    index=(index+1)%size;
}
printf("cannot insert..\n");
```



Key	Name	Mark
15	ABC	1
46	DEF	1
71	GHI	1
18	JKL	1
34	MNP	1

Other elements are stored in the memory as shown.
Mark field is set to 1.

An element is to be removed from the hash table. How to delete..?

```
void delete_from_hash(struct element *ht,int size,int key, int *count)
{
    int i,index;
    printf("count = %d\n",*count);
```

// If Table is empty display table is empty.

```
    if(*count==0)
    {
        printf("table empty..\n");
        return;
    }
```

// if Mark is '0', indicates the element is not present in the hash table

// Otherwise:

Data Structures and its Applications

Hashing – Quadratic Probing – Deletion of an element

// Search for the element to be deleted

```
index = key % size;
i=index;
h=1;

for(h=1;h<=size;h++)
{
    if (ht[index].mark==1)
    {
        // indicates element is present
        if(ht[i].key==key) // if found
            printf("key Found and deleted.."); // Delete
            ht[i].mark=0;
            return;
    }
    index=(index+1)%size; // search for the element in the
                          //consecutive memory location
}

printf("key not found..");
return;
}
```

Key	Name	Mark
15	ABC	1
46	DEF	0
71	GHI	1
18	JKL	1
34	MNP	1

If key is 46, then key is found.

Then, set mark field to 0.

This indicates that the element is deleted.

Data Structures and its Applications

Hashing – Linear Probing – Deletion of an element

// Search for the element to be deleted

Ex2: Let's delete 85.

First, search for 85 at location $((85 \% 10) + 1 * 1) \% 10 = 6$.

Key 85 doesn't match with 25 in the hash table.

Now check at location $((85 \% 10) + 2 * 2) \% 10 = 9$.

It is the same.

Now, Set the mark value to '0'.

This is deletion of an element from the hash table.

Index	Key	Mark
0	15	1
1	34	1
2	46	1
3	72	1
4	18	1
5	-	0
6	25	1
7	-	0
8	-	0
9	85	0



THANK YOU

V R BADRI PRASAD

Department of Computer Science & Engineering

badriprasad@pes.edu



Data Structures and its Applications

V R BADRI PRASAD

Department of Computer Science & Engineering

DATA STRUCTURES AND ITS APPLICATIONS

Double Hashing

V R BADRI PRASAD

Department of Computer Science & Engineering

- Double hashing is collision resolving technique in open addressed hash tables.
- Double hashing uses the idea of applying a second hash function to key when a collision occurs.

Let $h1(key) = key \bmod 13$ be hash function1.

Let $h2(key) = 7 - key \bmod 7$ be hash function2.

Consider the following elements to be inserted into the hash table of size **TABLESIZE**.

Keys : { 18, 41, 22, 44 }.

Let's now calculate the values of $h1(key)$ and $h2(key)$.

Key	$h1(key)$	$h2(key)$
18	$18 \bmod 13 = 5$	$7 - (18 \bmod 7) = 3$
41	$41 \bmod 13 = 2$	$7 - (41 \bmod 7) = 1$
22	$22 \bmod 13 = 9$	$7 - (22 \bmod 7) = 6$
44	$44 \bmod 13 = 5$	$7 - (44 \bmod 7) = 2$

Data Structures and its Applications

Open Hashing – Double Hashing



Key	$h_1(\text{key})$	$h_2(\text{key})$	Double hash(key)
18	$18 \bmod 13 = 5$	$7 - (18 \bmod 7) = 3$	-
41	$41 \bmod 13 = 2$	$7 - (41 \bmod 7) = 1$	-
22	$22 \bmod 13 = 9$	$7 - (22 \bmod 7) = 6$	-
44	$44 \bmod 13 = 5$	$7 - (44 \bmod 7) = 2$	-

Index	0	1	2	3	4	5	6	7	8	9	10	11	12
Key			41			18				22			

- Key – 18 , using $h_1(\text{key})$ gives 5 as index / hash. Go to location 5. It is Free. Assign 18 to location 5.
- Key – 41 , using $h_1(\text{key})$ gives 2 as index / hash. Go to location 2. It is Free. Assign 41 to location 2.
- Key – 22 , using $h_1(\text{key})$ gives 2 as index / hash. Go to location 2. It is Free. Assign 22 to location 9.
- Key – 44 , using $h_1(\text{key})$ gives 5 as index / hash. Go to location 5. It is not Free.

Use double hashing function.

Data Structures and its Applications

Open Hashing – Double Hashing



Key	$h1(key)$	$h2(key)$	Double hash(key)
18	$18 \bmod 13 = 5$	$7 - (18 \bmod 7) = 3$	-
41	$41 \bmod 13 = 2$	$7 - (41 \bmod 7) = 1$	-
22	$22 \bmod 13 = 9$	$7 - (22 \bmod 7) = 6$	-
44	$44 \bmod 13 = 5$	$7 - (44 \bmod 7) = 5$	-

Index	0	1	2	3	4	5	6	7	8	9	10	11	12
Key			41			18				22			

- Key – 44 , using $h1(key)$ gives 5 as index / hash. Go to location 5. It is not Free.

Use double hashing function. Index/hash = $hash1(key) + j * hash2(key)$, $j=1$ as it has the first collision.

. Index/hash = $(5 + 1 * (5)) \% 13 = 10$

Data Structures and its Applications

Open Hashing – Double Hashing



Key	$h1(key)$	$h2(key)$	Double hash(key)
18	$18 \bmod 13 = 5$	$7 - (18 \bmod 7) = 3$	-
41	$41 \bmod 13 = 2$	$7 - (41 \bmod 7) = 1$	-
22	$22 \bmod 13 = 9$	$7 - (22 \bmod 7) = 6$	-
44	$44 \bmod 13 = 5$	$7 - (44 \bmod 7) = 5$	10

Index	0	1	2	3	4	5	6	7	8	9	10	11	12
Key			41			18				22	44		

- Key – 44 , using $h1(key)$ gives 5 as index / hash. Go to location 5. It is not Free.

Use double hashing function. Index/hash = $hash1(key) + j * hash2(key)$, $j=1$ as it has the first collision.

$$Index/hash = (5 + 1 * (5)) \% 13 = 10.$$

Since location 10 is free, key value 44 is stored in location 10.

The collision is resolved using double hashing.



THANK YOU

V R BADRI PRASAD

Department of Computer Science & Engineering

badriprasad@pes.edu