# SET DATA STRUCTURE
# (Part 2)

# Representation of Sets

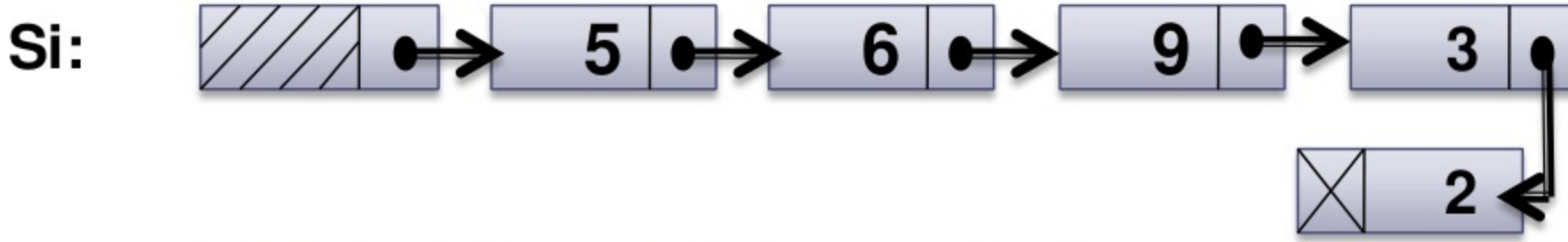
- LIST
- HASH TABLE
- BIT VECTORS
- TREE

# LIST REPRESENTATION

- Simplest and straight forward
- Best suited for dynamic storage facility.
- This allow multiplicity of elements ie; Bag structure.
- All operations can be easily implemented and performance of these operations are as good as compared to other representations.
- Ex: set S = { 5,6,9,3,2,7,1} using linked list structure is

# Operations on List Representation of sets

**UNION:**



## ALGORITHM : UNION_LIST_SETS(Si,Sj;S)

- **Input**: $S_i$ and $S_j$ are header of two single linked list representing two distinct sets.
- **Output**: S is the union of $S_i$ and $S_j$.
- **Data structure**: Linked list representation of set.

## STEPS

*/* to get a header note for S and initialize it*/*

1. S= GETNODE(NODE)
2. S.LINK= NULL, S.DATA = NULL

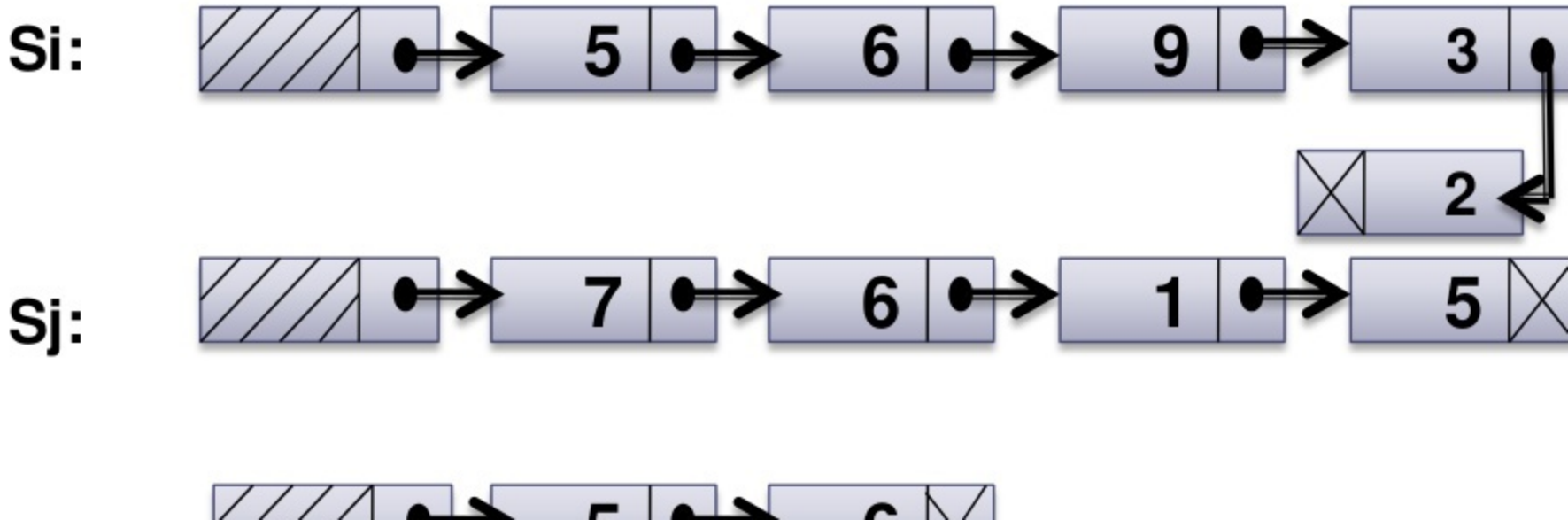
*/* to copy the entire list of Si into S*/*

3. ptri = si.LINK
4. While (ptri !=NULL) do
   1. Data = ptri.data
   2. INSERT_SL_FRONT(S, DATA)
   3. ptri= ptri.LINK

5 End while

7. While (ptrj!=NULL) do
   ptri=Si.link
   while (ptri. DATA != ptrj. DATA) do
      1. ptri=ptri.LINK
8. Endwhile
9.If (ptri=NULL) then
      INSERT_SL_FRONT(S,ptrj.DATA)
10. EndIf
11. ptrj=ptrj.LINK
12. Endwhile
13. Return (S)
14. stop

# INTERSECTION



# ALGORITHM : INTERSECTION_LIST_SETS(Si,Sj;S)

- **Input**: $S_i$ and $S_j$ are header of two single linked list representing two distinct sets.
- **Output**: S is the intersection of $S_i$ and $S_j$.
- **Data structure**: Linked list representation of set.

**STEPS:**

*/* To get a header node for S and initialize it */*

1. S= GETNODE(NODE)
2. S. LINK= NULL, S. DATE= NULL

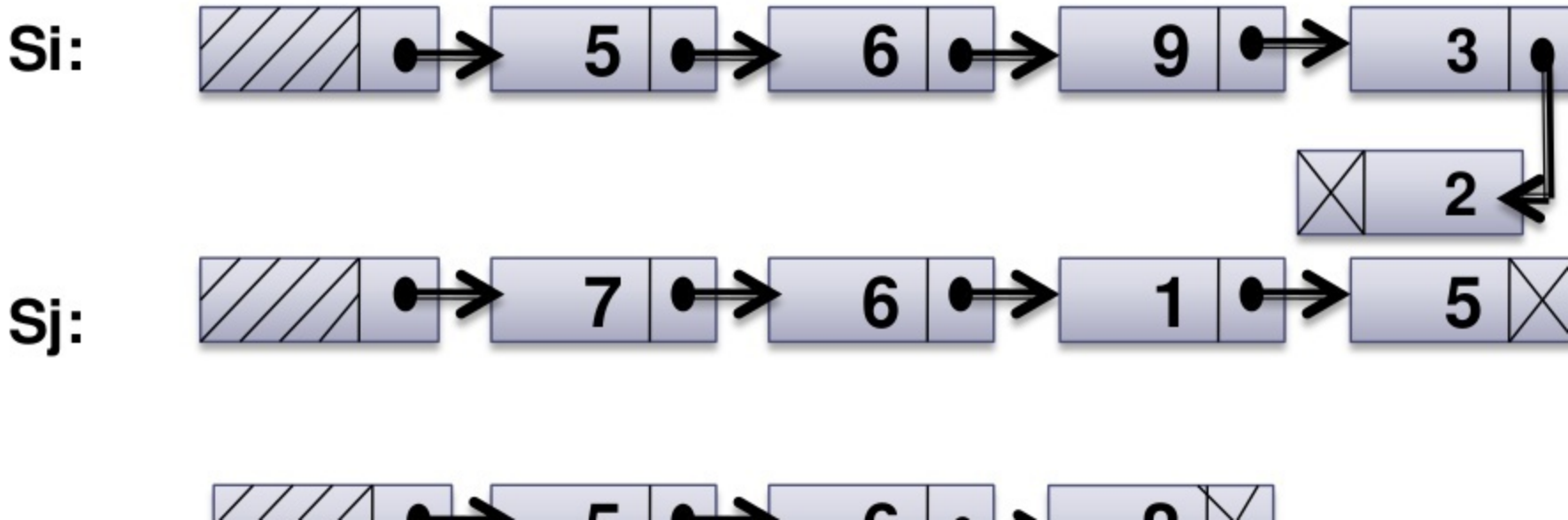
*/* search the list Sj, for each element in Si */*

3. ptri= Si.LINK
4. While (ptri!= NULL) do
    1. ptrj= Sj.LINK
    2. While(ptri.DATA!= ptrj.DATA) and(ptrj.!=NULL ) do

3. Endwhile.
4. If (ptrj!=NULL) then // when the element is found in Sj
    1. INSERT_SL_FRONT(S,ptrj,DATA)
5. EndIf
6. ptri = Si.LINK

5. Endwhile
6. Return(S)
7.Stop.

## DIFFERENCE:



# ALGORITHM : DIFFERENCE_LIST_SETS(Si,Sj;S)

- **Input**: $S_i$ and $S_j$ are header of two single linked list representing two distinct sets.
- **Output**: S is the difference of $S_i$ and $S_j$.
- **Data structure**: Linked list representation of set.

**STEPS:**

*/* Get a header node forS and initialize it*/*

1.S= GETNODE(NODE )

2. S.LINK= NULL,S. DATA =NULL

*/* Get S' the intersection of Si, and Sj*/*

3. S'= INTERSECTION _LIST_SET_(Si, Sj)

*/* Copy the entire list Si into S*/*

4.ptri= Si. LINK

5. While (ptri.LINK!=NULL) do

*/* For each element in S'. Delete it from S if it is there*/*

7.ptr= S'.LINK

8.While (ptr!=NULL) do

   1. DELETE_SL_ANY(S,ptr.DATA)

   2. ptr=ptr.LINK

9. Endwhile

10.Return (S)

11.Stop.

# ALGORITHM : EQUALITY_LIST_SETS(Si,Sj)

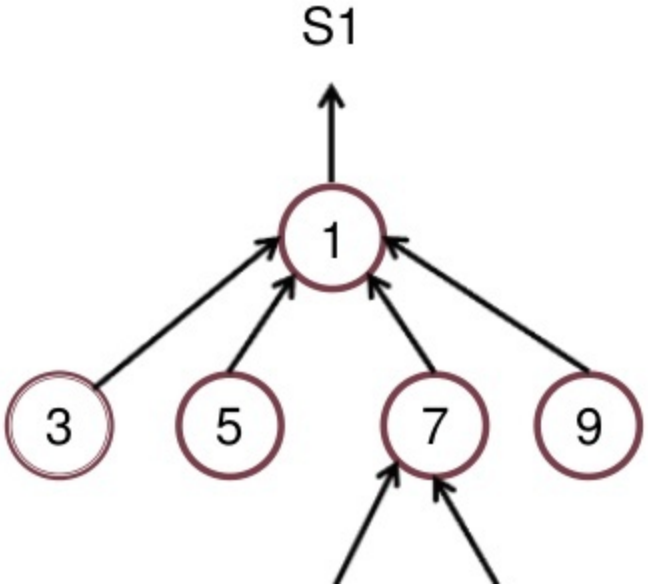
- **Input**: $S_i$ and $S_j$ are header of two single linked list representing two distinct sets.
- **Output**: Return TRUE if two sets $S_i$ and $S_j$ equal else FALSE
- **Data structure**: Linked list representation of set.

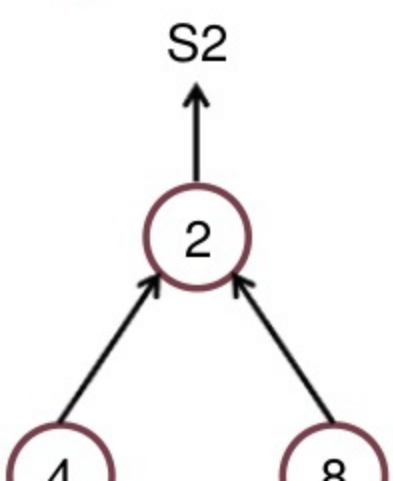
## STEPS

1. li= 0, lj =0
2. ptr=Si.LINK *// to count Si*
3. while (ptr!=NULL) do
    1. li=li+1
    2. ptr=ptr.LINK
4. Endwhile
5. ptr=Sj.LINK *//to count Sj*
6. While (ptr!=NULL) do
    1. lj=lj+1
    2. ptr=ptr.LNIK

10. ptri= Si.LINK,flag=TRUE
11. While (ptril!=NULL )and (flag = TURE) do
    1. ptrj=sj.LINK
    2. while (ptrj.DATA !=ptri.DATA)and
        (ptrj!=NULL)  do
        1.ptrj=ptrj.LINK
    3. Endwhile
    4.ptri=ptri.LINK
    5. If (ptrj= NULL)then
        1. flag= FALSE
    6.Endif

- Here a tree is used to represent one set, and the each element in the set has the same root.
- Each element in a set has pointer to its parent.
- Let us consider sets S1 ={1,3,5,7,9,11,13}
  S2 ={2,4,8}
  S3 ={6}

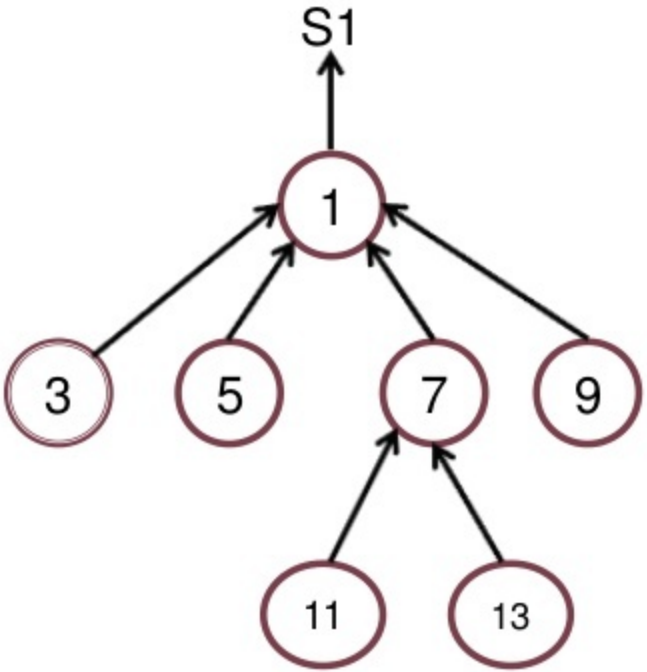


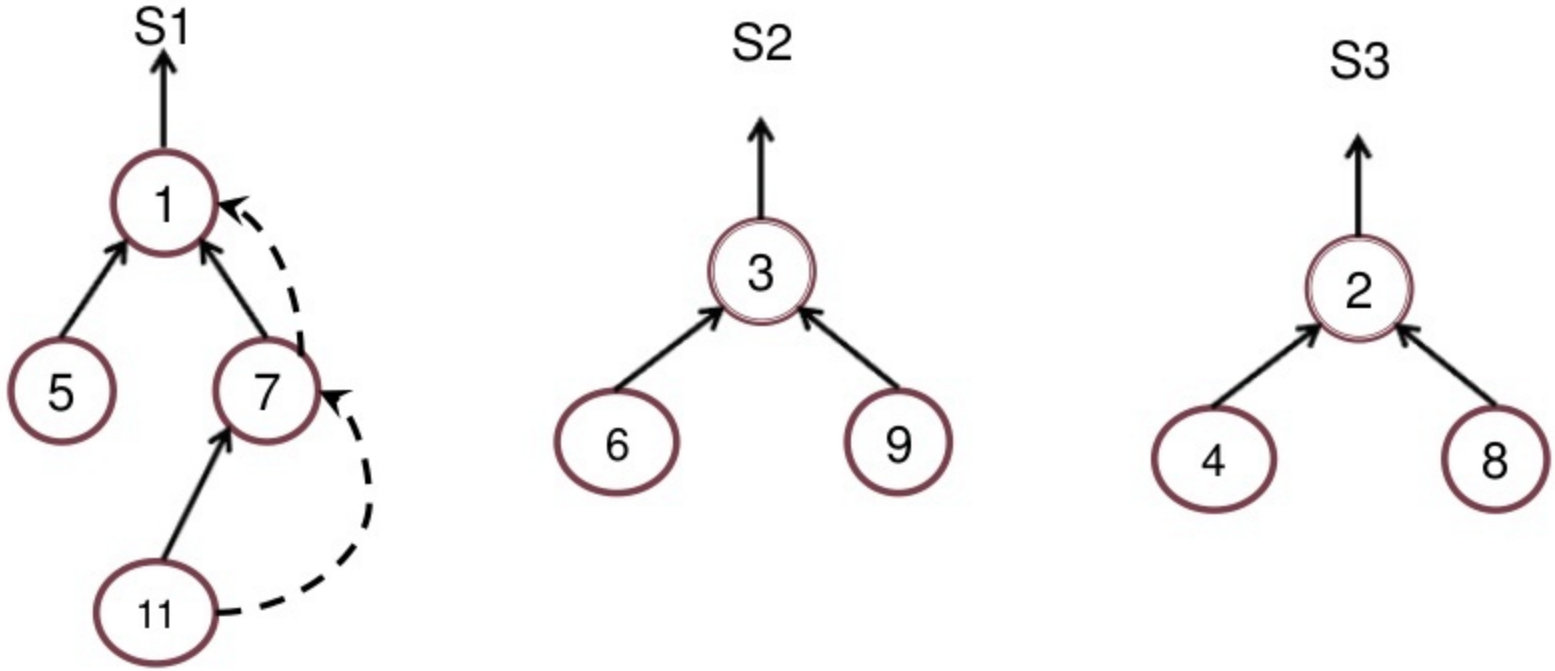


S3 ={6}

# Tree representation of set S1 ={1,3,5,7,9,11,13}

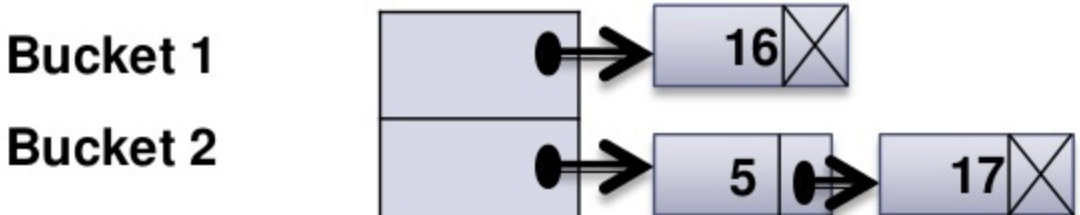


# Illustration of FIND method

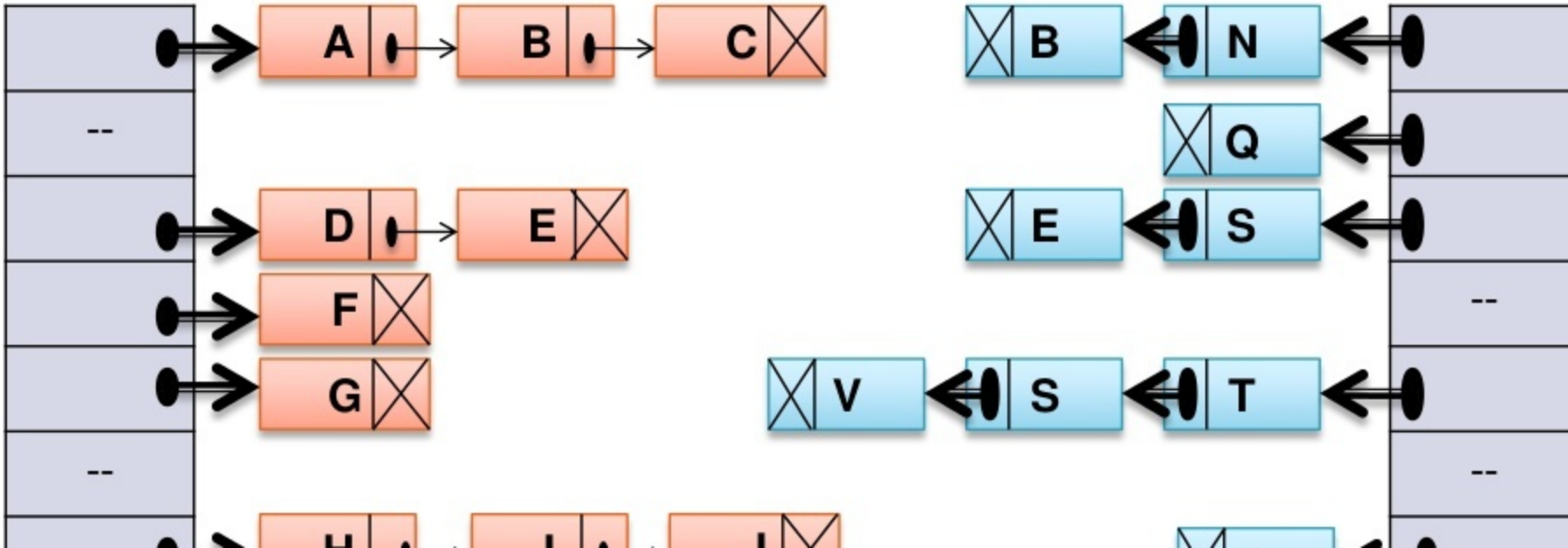


# HASH TABLE

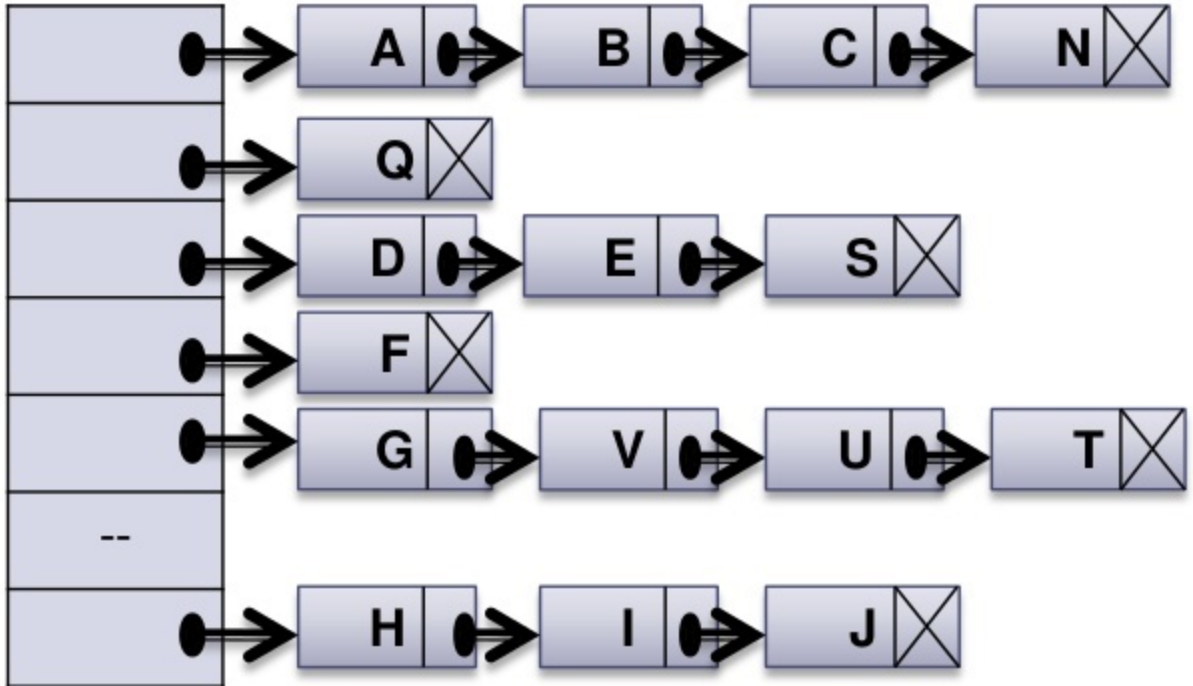
- Here the elements in collection are separated in to number of buckets.
- Each bucket can hold arbitrary number of elements.
- Consider set S ={2,5,7,16,17,23,34,42}
- Here hash table with 4 buckets and H(x) hash function can store which can place element from S to any of the four buckets.



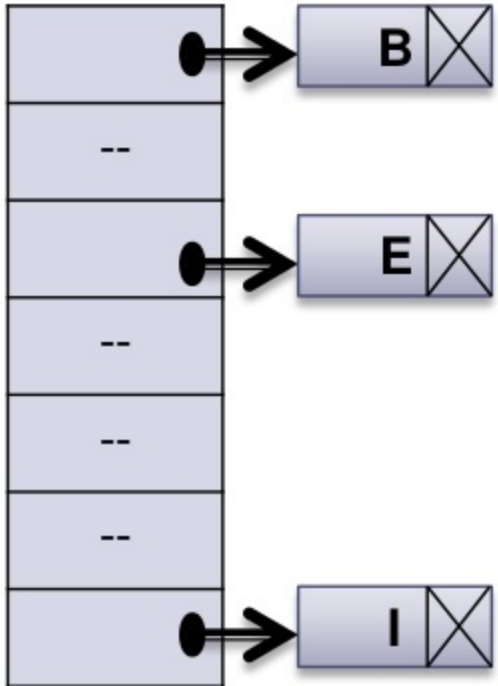
# Operation on Hash table Representation of Sets



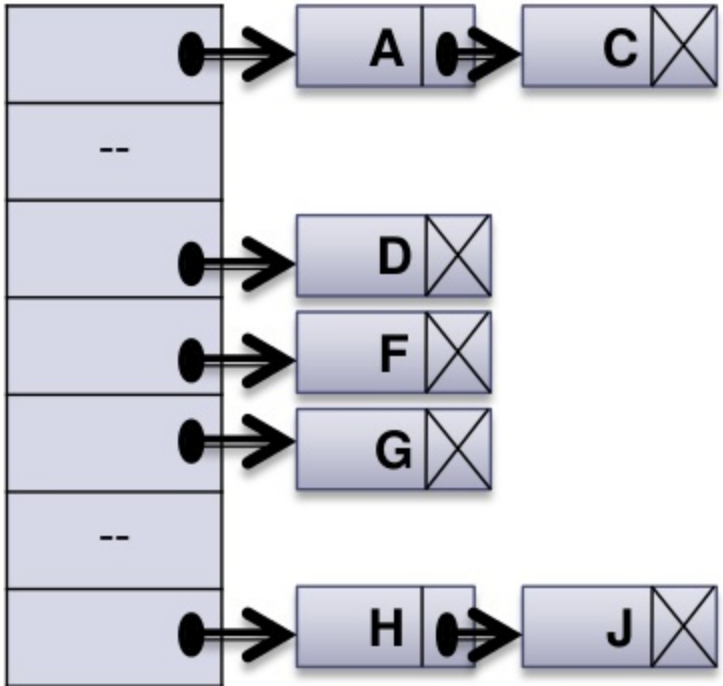
# UNION: S = Si U Sj



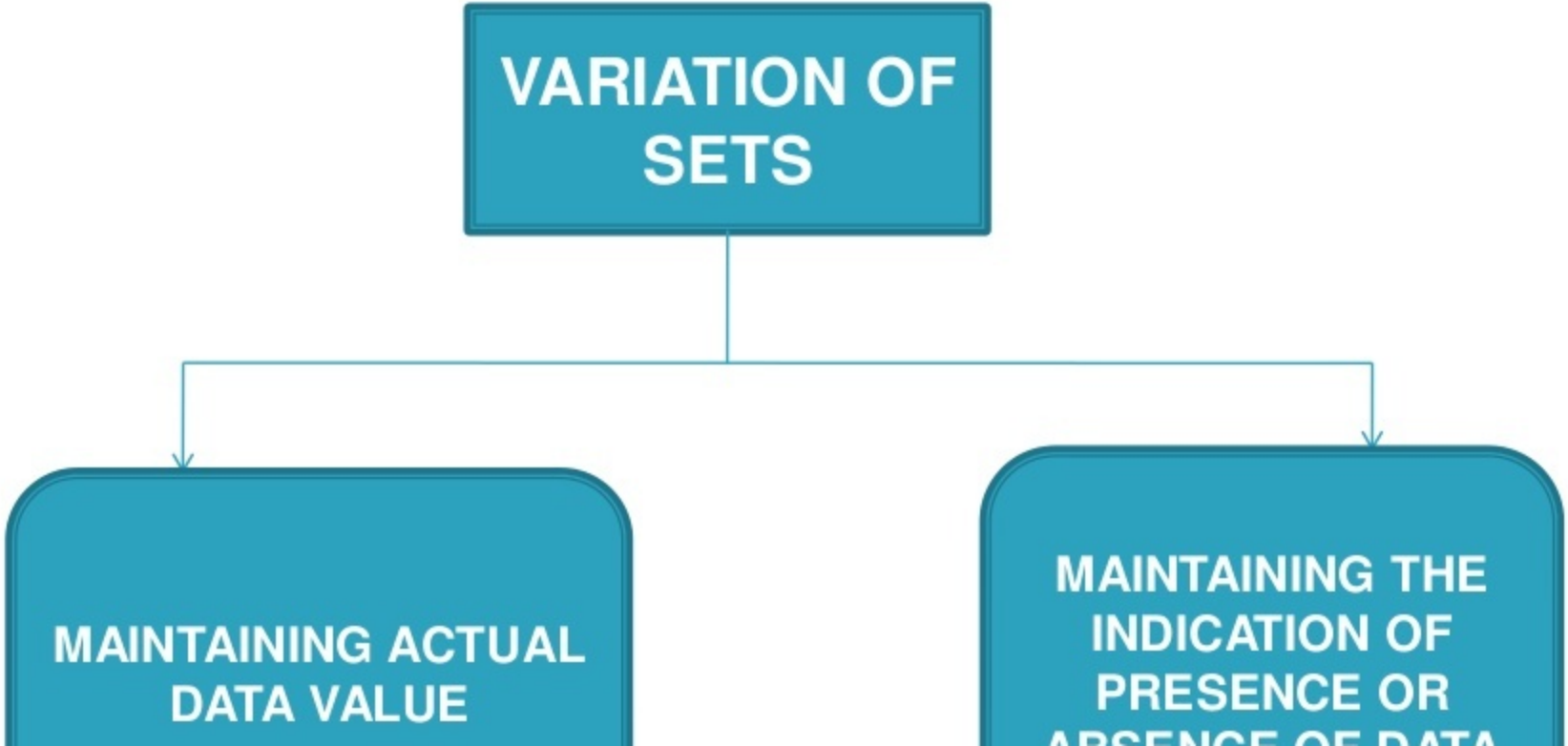
## INTERSECTION




## DIFFERENCE



# BIT VECTOR


VARIATION OF SETS
MAINTAINING ACTUAL DATA VALUE
MAINTAINING THE INDICATION OF PRESENCE OR

- A set, giving the records about the age of cricketer less than or equal to 35 is as given below:

  {0,0,0,0,1,1,1,1,0,1,1}
- Here 1 indicates the presence of records having the age less than or equal to 35.
- 0 indicates the absence of records having the age less than or equal to 35.
- As we have to indicate presence or absence of an element only, so 0 or 1 can be used for indication for saving storage space
- A bit array data structure is known for this purpose.

# Operations on bit vector representation

- It is very easy to implement set operation on the bit array data structure.
- The operations are well defined only if the size of the bit arrays representing two sets under operation are of same size.

# UNION

- To obtain the union of sets $s_i$ and $s_j$, the bit-wise OR operation can be used
- $S_i$ and $S_j$ are given below:

$S_i$ = 1 0 0 1 0 1 1 0 0 1

$S_j$ = 0 0 1 1 1 0 0 1 0 0

# ALGORITHM : UNION_BIT_SETS(Si,Sj;S)

- **Input**: $S_i$ and $S_j$ are two bit array corresponding to two sets.
- **Output**: A bit array S is the result of $S_i$ U $S_j$.
- **Data structure**: Bit vector representation of set.

1. li=LENGTH(Si) //Size of $S_i$.
2. li=LENGTH($S_j$) //Size of $S_j$
3. If (li != lj) then
   1. Print “Two sets are not compatible for union”
   2. Exit
4. End if

/* Loop over the under lying bit arrays and bit-wise OR on its constituents data.*/

5. For i=1 to li do

# INTERSECTION

- To obtain the intersection of sets $s_i$ and $s_j$, the bit-wise AND operation can be used
- $S_i$ and $S_j$ are given below:

$$S_i = 10010110001$$

$$S_j = 00111001000$$

$$S_i \bullet S_j = 00010000000$$

# ALGORITHM : INTERSECTION_BIT_SETS(Si,Sj;S)

- **Input**: $S_i$ and $S_j$ are two bit array corresponding to two sets.
- **Output**: A bit array S is the result of $S_i \cap S_j$.
- **Data structure**: Bit vector representation of set.

# STEPS:

1. li=LENGTH($S_i$) //Size of $S_i$
2. li=LENGTH($S_j$) //Size of $S_j$
3. If (li != lj) then
   1.Print "Two sets are not compatible for intersection"
   2.Exit
4. End if

/* Loop over the under lying bit arrays and bit-wise AND on its constituents data.*/

5. For i=1 to li do
   1.S[i] = $S_i$[i] AND $S_j$[i]

# DIFFERENCE

- The difference of $S_i$ from $S_j$ is the set of values that appear in $S_i$ but not in $S_j$. This can be obtained using bit-wise AND on the inverse of $S_j$.
- $S_i$ and $S_j$ are given below:

$S_i$ = 1 0 0 1 0 1 1 0 0 1

$S_j$ = 0 0 1 1 1 0 0 1 0 0

$S_j'$ = 1 1 0 0 0 1 1 0 1 1

# ALGORITHM : DIFFERENCE_BIT_SETS(Si,Sj;S)

- **Input**: $S_i$ and $S_j$ are two bit array corresponding to two sets.
- **Output**: A bit array S is the result of $S_i$ and $S_j$.
- **Data structure**: Bit vector representation of set.

## STEPS:

1. li=LENGTH($S_i$) //Size of $S_i$
2. lj=LENGTH($S_j$) //Size of $S_j$
3. If (li != lj) then
   1. Print "Two sets are not compatible for difference"
   2. Exit
4. End if /*To find the inverse (NOT) of $S_j$*/
5. For i=1 to li do
   1. $S_j[i]$ = NOT $S_j[i]$
6. EndFor /*Loop over the under lying bit arrays and bit-wise AND*/

# EQUALITY

- The equality operation is used to determine whether two sets $S_i$ and $S_j$ are equal or not.
- This can be achieved by simple comparison between the pair-wise bit values in two bit arrays.

# ALGORITHM : EQUALITY_BIT_SETS(Si,Sj)

- **Input**: $S_i$ and $S_j$ are two bit array corresponding to two sets.
- **Output**: Return TRUE if they are equal else FALSE.
- **Data structure**: Bit vector representation of set.

# STEPS:

1. li=LENGTH($S_i$) //Size of $S_i$
2. li=LENGTH($S_j$) //Size of $S_j$
3. If (li != lj) then
   1.Return (FALSE) //return with failure
   2.Exit
4. End if

   /*Loop over the under lying bit arrays and compare*/

5. For i=1 to li do
   1.$S_J$[i] != $S_j$[i] then
   1.Return (FALSE) //return with failure
   2.Exit
   2.EndIf

# Application of Set Data Structure

# Information storing using bit string

- Let us consider a technique of storage and retrieval of information using bit strings.
- **A bit string is a set of bits that is a string of 0's and 1's for example 1000110011 is a bit string**.
- Let us now see how the information can be stored and retrieved using bit string.
- Let us assume a simple database to store the

| NAME | REG NO | SEX | DISCIPLINE | MODULE | CATEGORY | ADDRESS |
|---|---|---|---|---|---|---|
| AAA | A1 | M | CS | C | SC | --- |
| BBB | A2 | M | CE | P | GN | --- |
| CCC | A3 | F | ME | D | GN | --- |
| DDD | A4 | F | EC | D | GN | --- |
| EEE | A5 | M | EE | P | ST | --- |
| FFF | A6 | M | AE | C | SC | ---- |
| GGG | A7 | F | ME | C | ST | --- |
| HHH | A8 | M | CE | D | GN | --- |
| III | A9 | F | CS | P | SC | --- |

- **Name** : String of Characters of length 25.
- **RegnNo** : Alpha numeric string of length 15.
- **Sex** : A single character value coded as
  **F**=Female **M**=Male
- **Discipline**: Two character value coded as:
  - **AE**-Agricultural Engineering
  - **CE**-Civil Engineering
  - **CS**-Computer Science and Engineering
  - **EC**-Electrical and Communication Engineering
  - **EE**-Electrical Engineering
  - **ME**-Mechanical
- **Module** : One character value coded as
  **C** = Certificate **P**=Diploma **D**= Degree
- **Category**: Two character value coded as

- **Length of bit string = number of records**(here 10).
- To store a particular column we require **Bit Arrays** storing a set of bit string.
- The number of bit arrays will be determined by different **attributes** that the field may have.
- For ex:
  - Sex : 2 for M or F
  - Discipline : 6 for six different branches
  - Module : 3 for three different streams
  - Category : 4 for different categories
- All together **15 bit arrays** each of length 10 in this case is required to store the information.

| ARRAY | BIT STRING |
| --- | --- |
| M | 1100110101 |
| F | 0011001010 |
| AE | 0000010001 |
| CE | 0100000100 |
| CS | 1000000010 |
| EC | 0001000000 |
| EE | 0000100000 |
| ME | 0000010000 |
| C | 0010001000 |
| P | 0100100011 |
| D | 0011000100 |

# Information retrieval using bit string

- How many students are there in engineering and computer discipline?

  To retrieve this information only bit arrays CS needs to be searched for the number of 1's in it.

- Who are the female students in CS discipline?

  For this information do **F ∩ CS or**

  **[0 0 1 1 0 0 1 0 1 0] ∩ [1 0 0 0 0 0 0 0 1 0] =**

  **[ 0 0 0 0 0 0 0 1 0 ]**

  Thus it gives the 9th record only

# Performance issue of the technique

- Efficient in terms of **storage point of view**

  If **v = number of bit arrays**

  **r = number of records**

  Total bits needed = **v*r;**

  In our example 15*10 = **150** bits.

  In contrast if we are using conventional method we may need **10** bytes for sex and module, **20** bytes for each Discipline and Category thus total **60** bytes=**480**

- From **computation point** of view this technique is efficient because no searching is involved.

- A record can be computed through logical operations like AND,OR,NOT and hence giving **fast computations.**

- One drawback of this technique is that it is **not possible to store all kind of information**. For example , the field where all or nearly all the values are different ,like name, regno, address this technique is in efficient.


Thank