Polynomial
&
Sparse Matrix

Amiya Ranjan Panda

Polynomial using Linked List

- Polynomial is a mathematical expression that consists of variables and coefficients.
  - Example: $x^2 - 4x + 7$

- In the Polynomial using linked list, the node contains the coefficients and exponents of the polynomial.

Nodes in a Polynomial

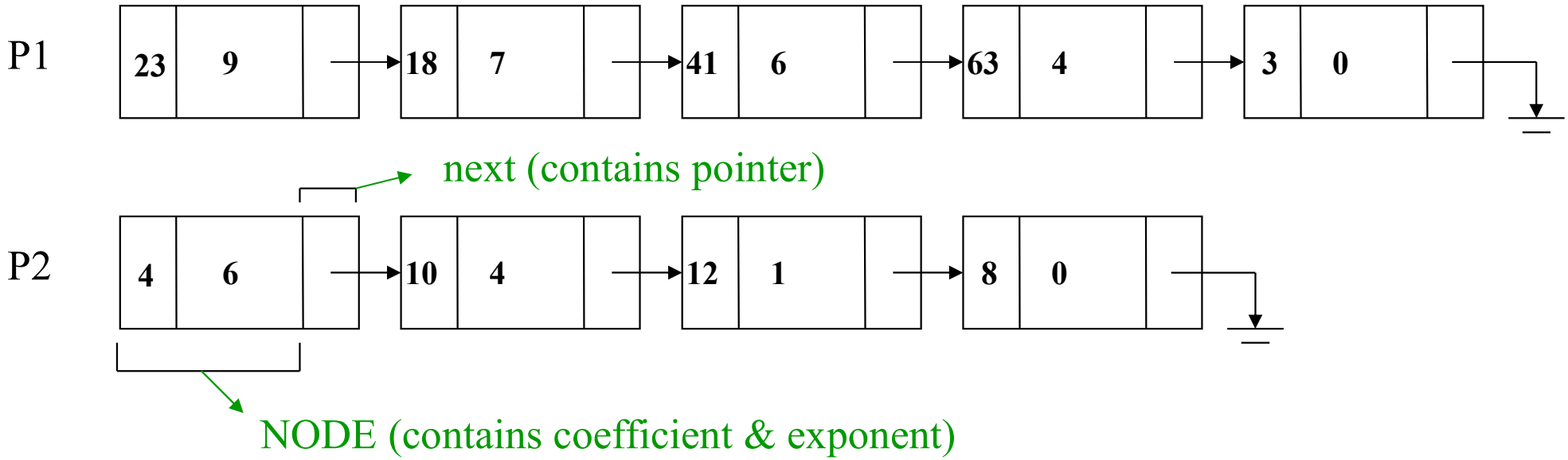
```
struct Node {
    int coeff;
    int exp;
    struct Node *next;
};
```

Implementation of Polynomial

Linked list Implementation:

$p1(x) = 23x^9 + 18x^7 + 41x^6 + 63x^4 + 3$

$p2(x) = 4x^6 + 10x^4 + 12x + 8$



Create Polynomial using Linked List

```
void createPoly() {
   struct Node *curr, *newNode;
   int ch=1;
   while(ch)  {
     newNode =(struct Node*)malloc(sizeof(struct Node));
     printf("Enter the coefficient :");
     scanf("%d", &newNode->coeff);
     printf("Enter the exponent: ");
     scanf("%d", &newNode->exp);
     newNode->next = NULL;
     if( head == NULL) head = newNode;
     else curr->next = newNode;
     curr = newNode;
     printf("Any more node to be added (1/0): ");
     scanf("%d", &ch);
   }  }
```

Display Polynomial

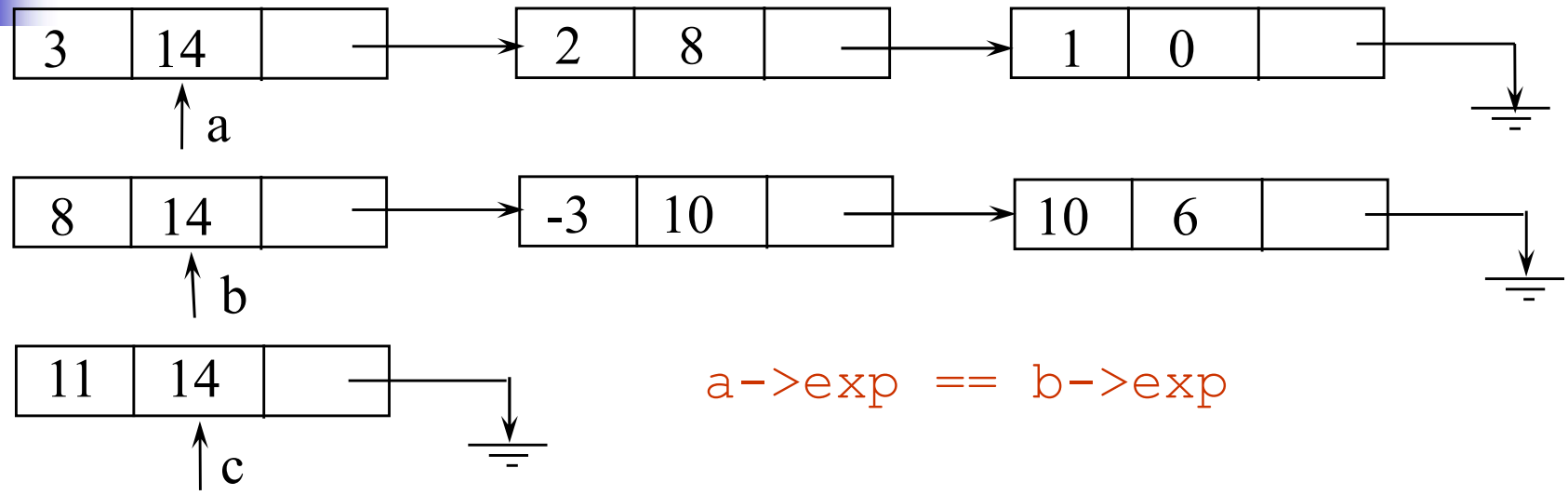
```
void displyPoly() {
   struct Node *curr = head;
   while(curr != NULL) {
      printf("%dx^%d", curr->coeff, curr->exp);
      if(curr->next != NULL)
         printf(" + ");
      curr = curr->next;
   }
}
```

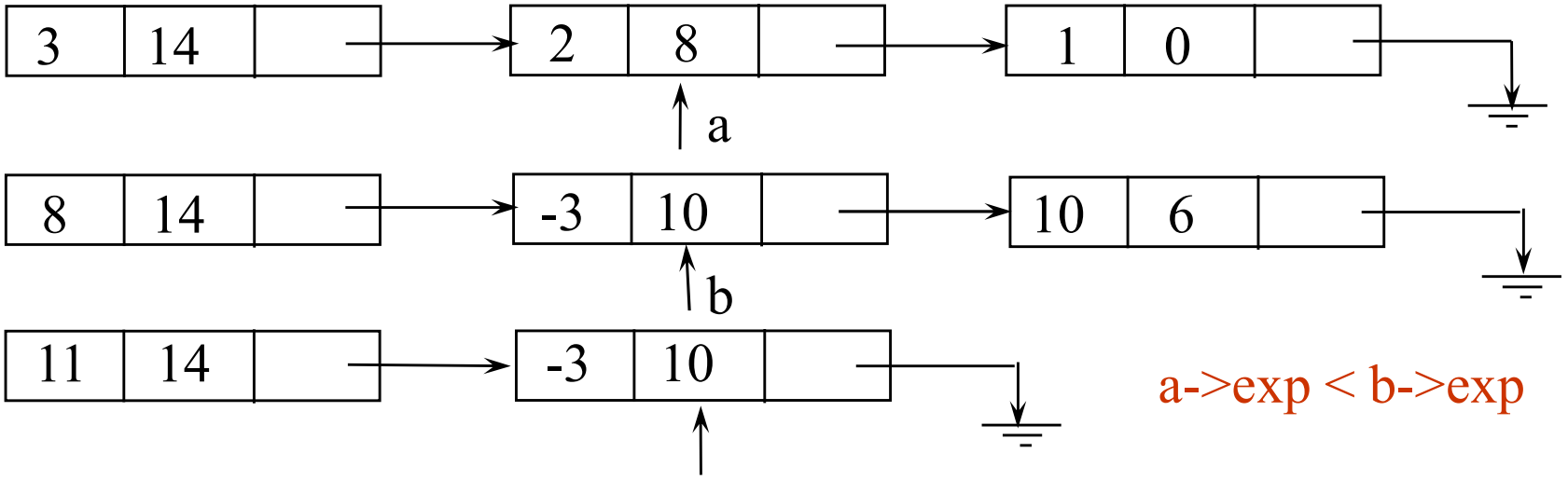
Add two Polynomials

Adding Polynomials

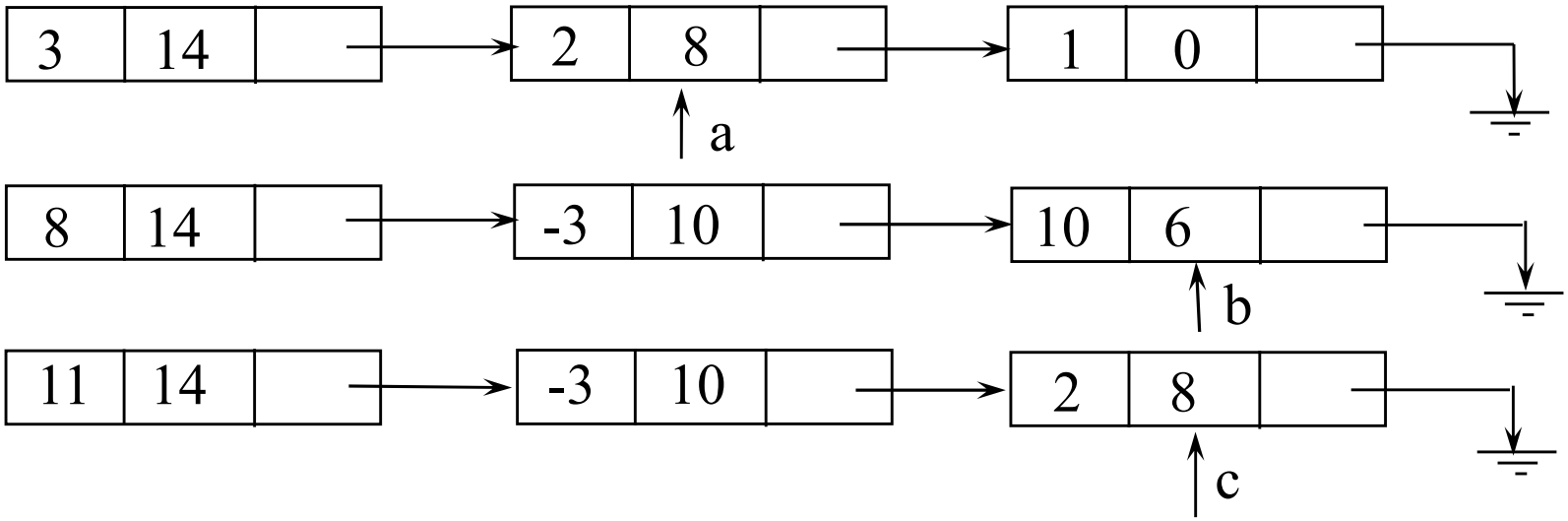
- To add two polynomials, examine their terms starting from the nodes pointed to by *a* and *b*.
  - If the exponents of the two terms are equal
    - add the two coefficients
    - create a new term for the result, named as *c*
  - If the exponent of the current term in *a* is less than *b*
    - create a term to store current term of *b*
    - attach this term to the result(i.e. in *c*)
    - advance the pointer to the next term in *b*
  - Similar action on *a* if *a->exp* > *b->exp*
- Figure generating the first three term of $c = a+b$

Add Polynomials





Add Polynomials




```
a->exp > b->exp
```

Add two Polynomials

```
void addPoly() {
  struct Node *poly1 = head1, *poly2 = head2, *head3 = NULL;
  struct Node *curr, *newNode;
  while(poly1 && poly2)  {
    newNode =(struct Node*)malloc(sizeof(struct Node));
    if(poly1->exp > poly2->exp) {
      newNode->coeff = poly1->coeff;
      newNode->exp = poly1->exp;
      poly1 = poly1->next;
    }
    else if(poly1->exp < poly2->exp) {
      newNode->coeff = poly2->coeff;
      newNode->exp = poly2->exp;
      poly2 = poly2->next;
    }
```

Add two Polynomials

```
  else {
     newNode->coeff = poly1->coeff +
                              poly2->coeff;
     newNode->exp = poly1->exp;
     poly1 = poly1->next;
     poly2 = poly2->next;
   }
   newNode->next = NULL;
   if( head3 == NULL) head3 = newNode;
   else curr->next = newNode;
   curr = newNode;
}
while(poly1 || poly2)  {
   newNode =(struct Node*)
              malloc(sizeof(struct Node));
   if(poly1) {
     newNode->coeff = poly1->coeff;
     newNode->exp = poly1->exp;
     poly1 = poly1->next;
   }
   if(poly2) {
     newNode->coeff = poly2->coeff;
     newNode->exp = poly2->exp;
     poly2 = poly2->next;
   }
   newNode->next = NULL;
   if( head3 == NULL) head3 = newNode;
   else curr->next = newNode;
   curr = newNode;
} }
```

Add two Polynomials: Example

Input:

1st Polynomial = 5x^2 + 4x^1 + 2x^0

2nd Polynomial = 5x^1 + 5x^0

Output:

5x^2 + 9x^1 + 7x^0

Input:

1st Polynomial = 5x^3 + 4x^2 + 2x^0

2nd Polynomial = 5x^1 + 5x^0

Output:

5x^3 + 4x^2 + 5x^1 + 7x^0

Multiply two Polynomials

```
void multiplyPoly() {
  struct Node *poly1 = head1, *poly2;
  struct Node *curr, *newNode, *prev;
  int coeff, exp;
  while (poly1 != NULL) {
    poly2 = head2;
    while (poly2 != NULL) {
      coeff = poly1->coeff * poly2->coeff;
      exp = poly1->exp + poly2->exp;
      curr = prev = head3;
      while(curr != NULL && curr->exp >= exp) {
        prev = curr;
        curr = curr->next;
      }
      if(curr == NULL || curr->exp != exp)  {
        newNode =(struct Node*)
                  malloc(sizeof(struct Node));
        newNode->coeff = coeff;
        newNode->exp = exp;
        newNode->next = NULL;
      }
      if(head3 == NULL)
        head3 = newNode;
      else if(curr == NULL)
            prev->next = newNode;
      else if(curr->exp != exp){
           prev->next = newNode;
           newNode->next = curr; }
      else  curr->coeff += coeff;
    }
    poly2 = poly2->next;     }
  poly1 = poly1->next;
}   }
```

Multiply two Polynomials: Example

Input: Poly1: 3x^2 + 5x^1 + 6, Poly2: 6x^1 + 8

Output: 18x^3 + 54x^2 + 76x^1 + 48

On multiplying each element of 1st polynomial with elements of 2nd polynomial, we get

18x^3 + 24x^2 + 30x^2 + 40x^1 + 36x^1 + 48

On adding values with same power of x,

18x^3 + 54x^2 + 76x^1 + 48

Input: Poly1: 3x^3 + 6x^1 + 9, Poly2: 9x^3 + 8x^2 + 7x^1 + 2

Output: 27x^6 + 24x^5 + 75x^4 + 135x^3 + 114x^2 + 75x^1 + 18

Nodes in a Sparse Matrix

```
struct Node {
    int row;
    int col;
    int val;
    struct Node *next;
};
```

Create a Sparse Matrix

```
void createSparse() {
  int i, j, row = 4, col = 5, count = 0;
  int sm[row][col] = {  {0 , 0 , 3 , 0 , 4 }, {0 , 0 , 5 , 7 , 0 },
                        {0 , 0 , 0 , 0 , 0 }, {0 , 2 , 6 , 0 , 0 }  };
  struct Node *curr = head, *newNode;
  for(i = 0; i < 4; i++)
    for(j = 0; j < 5; j++)
      if(sm[i][j] != 0) {
        newNode =(struct Node*) malloc(sizeof(struct Node));
        newNode->row = i;   newNode->col = j;
        newNode->val = sm[i][j];
        newNode->next = NULL;  count++;
        if(head == NULL) head = newNode;
        else curr->next = newNode;
        curr = newNode;    }
  if(count != 0)  {
    newNode =(struct Node*)
      malloc(sizeof(struct Node));
    newNode->row = row;
    newNode->col = col;
    newNode->val = count;
    newNode->next = head;
    head = newNode;
  }
}
```

Display Sparse Matrix

```
void displaySparse() {
    struct Node *temp = head;
    printf(\nRow #\tCol #\tValue\n");
    while(temp != NULL) {
      printf("%d\t\t%d\t\t%d\n", temp->row, temp->col, temp->val);
      temp = temp->next;
    }
}
```

Add Two Sparse Matrices

```
void input()  {
    int rows, cols, r, c, v, nos, val;
    printf("Enter No. of rows and coluns: ";
    scanf("%d%d", &rows, &cols");
    printf("Enter No. of non-zero values: ");
    scanf("%d", &nos);
    head=(Sparse *) malloc(sizeof(Sparse));
    head->row=rows;
    head->col=cols;
    head->val=nos;
    head->next=NULL;
    v=0;
    while(v<nos)  {
        curr=(Sparse *) malloc(sizeof(Sparse));
        printf("Enter row, col & value:  ";
        scanf("%d%d%d", &curr->row,
                        &curr->col, &curr->val);
        curr->next=NULL;
        if(head->next==NULL) head->next=curr;
        else last->next=curr;
        last=curr;
        v++;
    }
}
```

Add Two Sparse Matrices

```
void addMatrix()  {
    Sparse *t1, *t2, *t3=NULL, *curr, *last;
    int count = 0;
    t1=head1, t2=head2;
    if(t1->row!=t2->row||t1->col!=t2->col)  {
        printf("Invalid Addition...\n");
        return;
    }
    curr=(Sparse *) malloc(sizeof(Sparse));
    curr->row=head1->row;
    curr->col=head1->col;
    curr->next=NULL;
    head3=curr;
    last=curr;

    t1=t1->next, t2=t2->next;
```

```
while(t1!=NULL && t2!=NULL)  {
    curr=(Sparse *) malloc(sizeof(Sparse));
    curr->next=NULL;
    if(t1->row < t2->row) {
        curr->row=t1->row;
        curr->col=t1->col;
        curr->val=t1->val;
        t1=t1->next;
         }
    else if(t1->row > t2->row) {
        curr->row=t2->row;
        curr->col=t2->col;
        curr->val=t2->val;
        t2=t2->next;
         }
```

Add Two Sparse Matrices

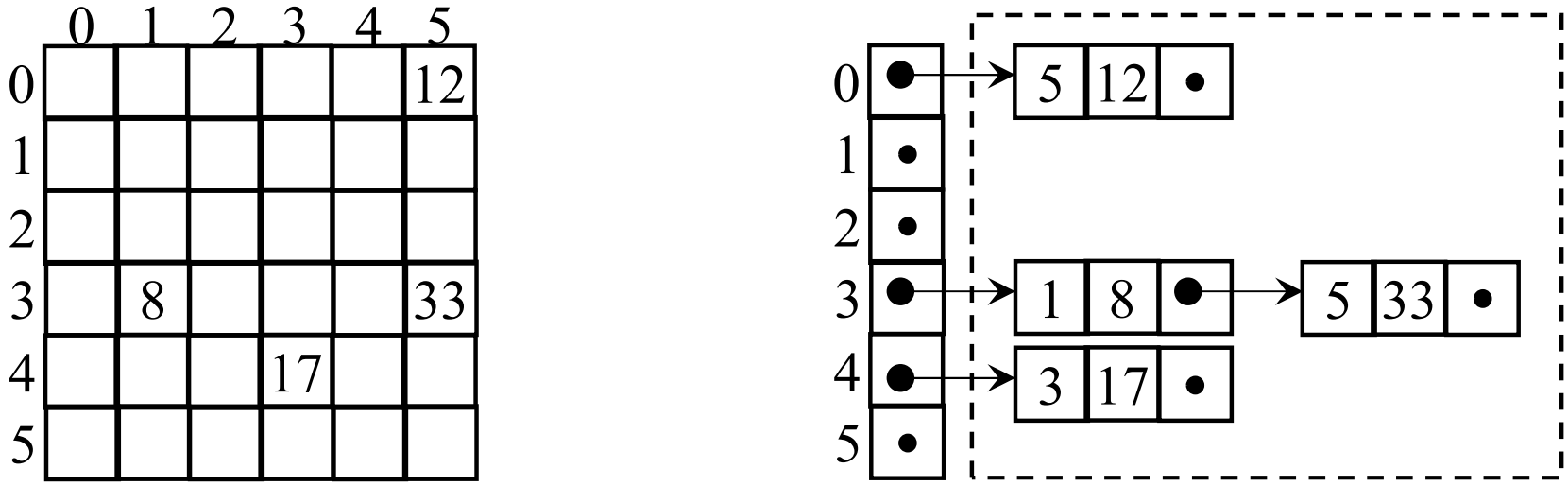
```
else if(t1->col < t2->col) {
    curr->row=t1->row;
    curr->col=t1->col;
    curr->val=t1->val;
    t1=t1->next;
else if(t1->col > t2->col) {
    curr->row=t2->row;
    curr->col=t2->col;
    curr->val=t2->val;
    t2=t2->next;
     }
else {
    curr->row=t1->row;
    curr->col=t1->col;
    curr->val=t1->val + t2->val;
    t1=t1->next, t2=t2->next;
}
    last->next=curr;
    last=curr;
    count++;   }
while(t1!=NULL) {
    curr=(Sparse *) malloc(sizeof(Sparse));
    curr->next=NULL;
    curr->row=t1->row; curr->col=t1->col;
    curr->val=t1->val;    t1=t1->next;
    last->next=curr;      last=curr;
    count++;
}
while(t2!=NULL) {
    curr=(Sparse *) malloc(sizeof(Sparse));
    curr->next=NULL;
    curr->row=t2->row; curr->col=t2->col;
    curr->val=t2->val;    t2=t2->next;
    last->next=curr;      last=curr;
    count++;        }
head3->val = count;
}
```

Multiply Two Sparse Matrices

```
void mulMatrix()  {
    Sparse *t1,*t2,*curr, *last;
    int count=0;
    head3=(Sparse *) malloc(sizeof(Sparse));
    head3->row=head1->row;
    head3->col=head2->col;
    head3->next=NULL;
    t1=head1->next;
    t2=head2->next;
    while(t1!=NULL) {
        t2=head2->next;
        while(t2!=NULL) {
            if(t1->col==t2->row) {
                curr=(Sparse *) malloc(sizeof(Sparse));
                curr->row=t1->row;
                curr->col=t2->col;
                curr->val=(t1->val)*(t2->val);
                curr->next=NULL;
                if(head3->next==NULL)
                    head->next=curr;
                else
                    last->next=curr;
                last=curr;
                count++;
            }
            t2=t2->next;
        }
        t1=t1->next;
    }
    head3->val=count;
}
```

Sparse Matrix - Implementation

- Example: Sparse matrix, represented as an *array* of linked lists:

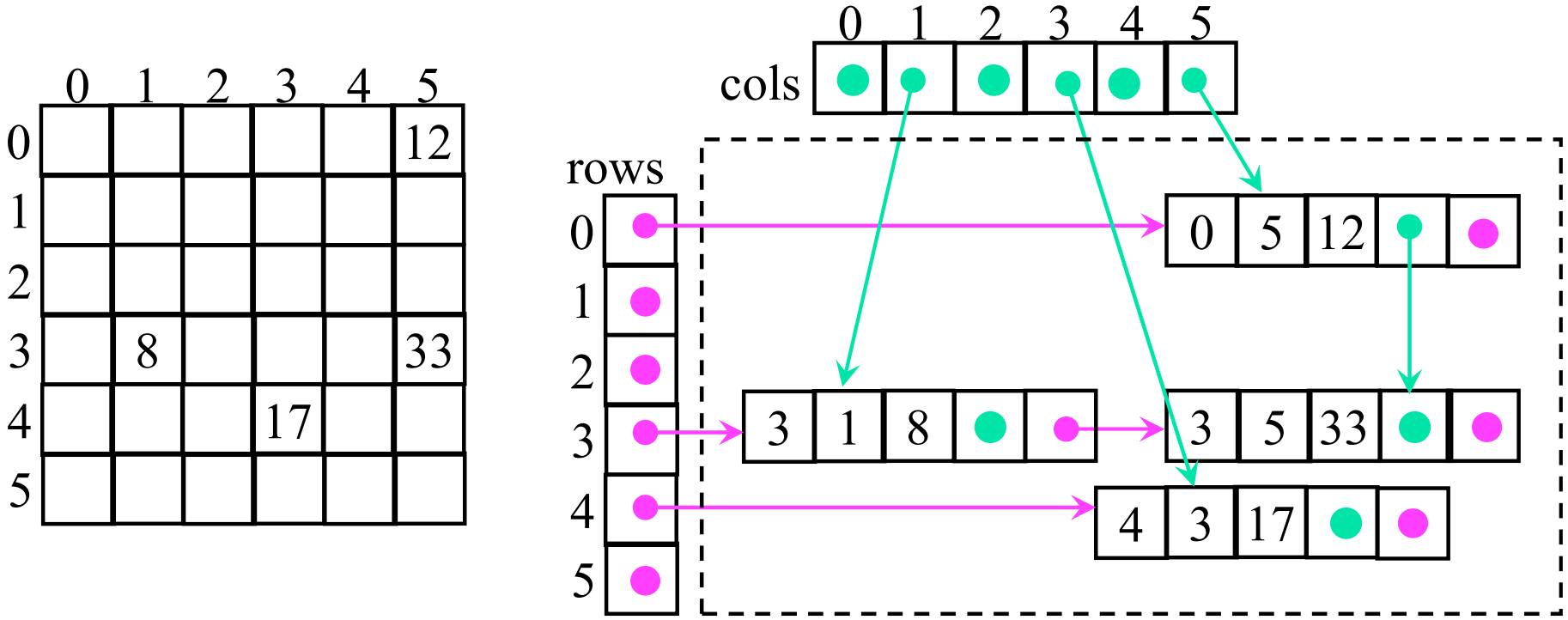


With this representation,

- It is efficient to step through all the elements of a *row*
- It is expensive to step through all the elements of a *column*
- Clearly, it could be linked columns instead of rows
- Why not both?

Another implementation

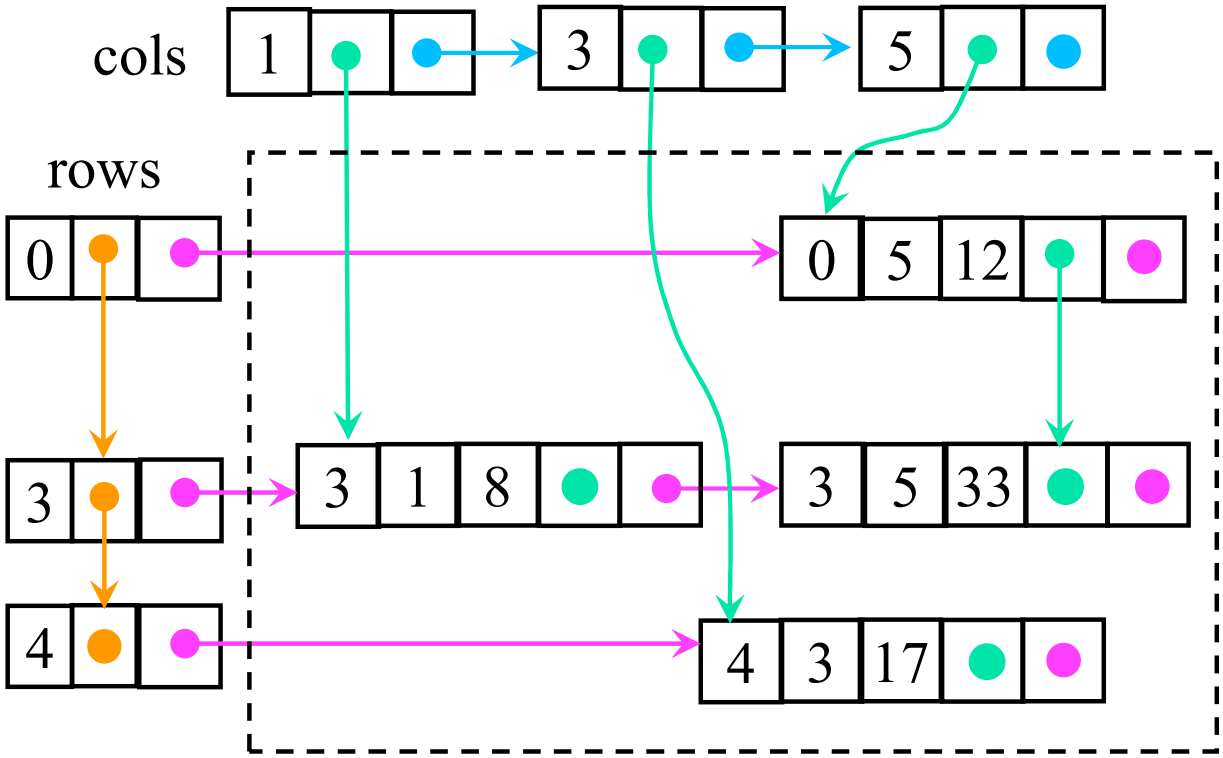
- For efficient access to both rows and columns , another array and additional data in each node is required



- Do we really need the row and column number in each node?

Yet another implementation

- Instead of arrays of pointers to rows and columns, linked lists can be used:



	0	1	2	3	4	5
0						12
1						
2						
3		8				33
4				17		
5						

- Would this be a good data structure?
- This may be the best implementation if most rows and most columns are almost empty