# Linked Lists

## Data Structures & Algorithms

# CONTENT

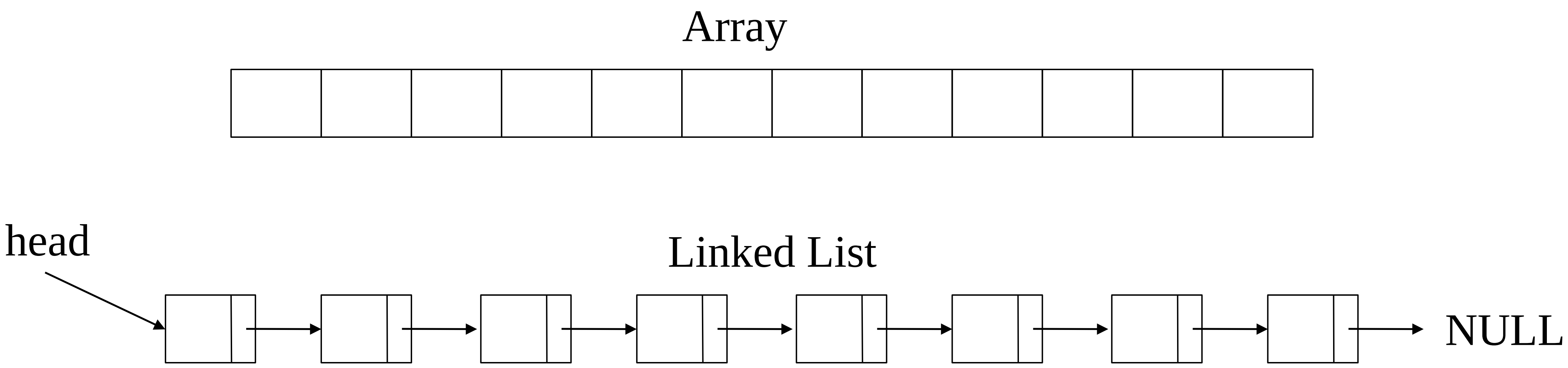
- **Linked List Basics**
- **Singly linked lists**
- **Circularly linked list**
- **Doubly linked lists**
- **Applications**
  - Polynomial representation
  - Equivalence relations
  - Sparse matrices

# CONTENT

- **Linked List Basics**
- **Singly linked lists**
- **Circularly linked list**
- **Doubly linked lists**
- **Applications**
  - Polynomial representation
  - Equivalence relations
  - Sparse matrices

- Linked lists & arrays are similar - Both store collections of data.
- **Array**: features all follow from its strategy of allocating sequentially its elements.
- **Linked lists**: use an entirely different strategy: linked lists allocate memory for each element separately and only when necessary.
  - Linked lists are used to store a collection of information (like arrays)
  - A linked list is made of nodes that are pointing to each other
  - We only know the address of the first node (head)
  - Other nodes are reached by following the “next” pointers
  - The last node points to NULL

- **Linked List vs. Array**
  - Elements of array are contiguous
  - In a linked list, nodes are not necessarily contiguous in memory (each node is allocated with a separate “new” call)



- **Linked List vs. Array**

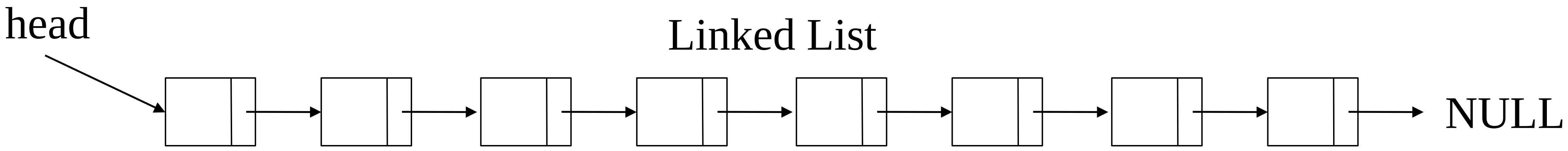
Array:

- **Advantages:**
  - Easy to use
  - A good choice for a small list
  - O(1) access time
- **Disadvantages:**
  - Fixed size
  - Memory wasting
  - Still space and time wasting in dynamic array

Linked List:

- **Advantages:**
  - Arbitrary size
  - No shift required
- **Disadvantages:**
  - Necessity to allocate next
  - O(N) access time

- **we use linked lists if...**
  - The number of elements that will be stored cannot be predicted at compile time
  - Elements may be inserted in the middle or deleted from the middle
  - We are less likely to make random access into the data structure (because random access is expensive for linked lists)

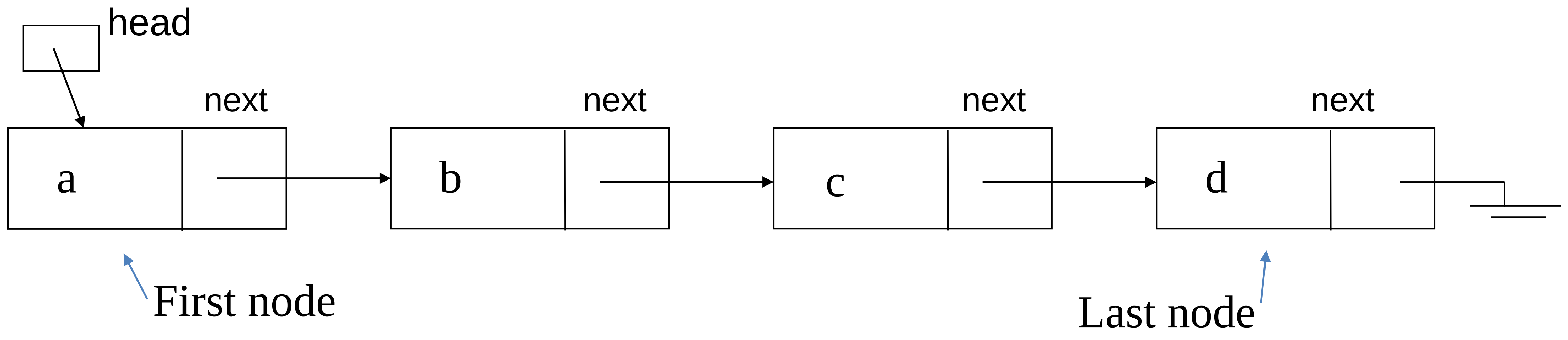


- Operations on Linked List
  - **Generic methods:** size(), isEmpty()
  - **Query methods:** isFirst(p), isLast(p)
  - **Accessor methods:** first(), last()
    before(p), after(p)
  - **Update methods:** insertFirst(e), insertLast(e)
    insertBefore(p,e), insertAfter(p,e)
    removeAfter(p)
    invert(p)
    replaceElement(p,e)
    swapElements(p,q),
    …

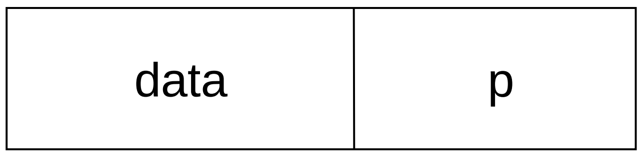
# CONTENT

- Linked List Basics
- **Singly linked lists**
- Circularly linked list
- Doubly linked lists
- Applications
  - Polynomial representation
  - Equivalence relations
  - Sparse matrices

- **Linked Lists**

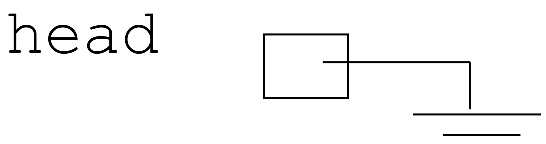



- **Each node has (at least) 2 fields:**
  - Data
  - Pointer to the next node

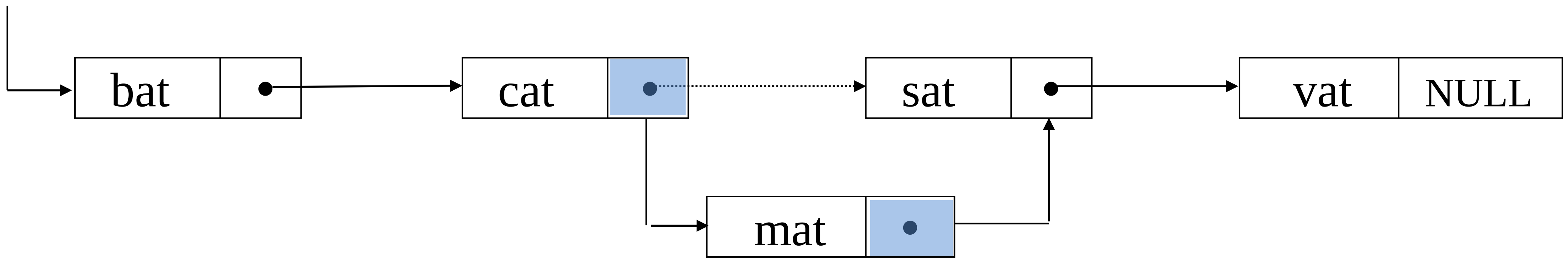



- **Empty linked**

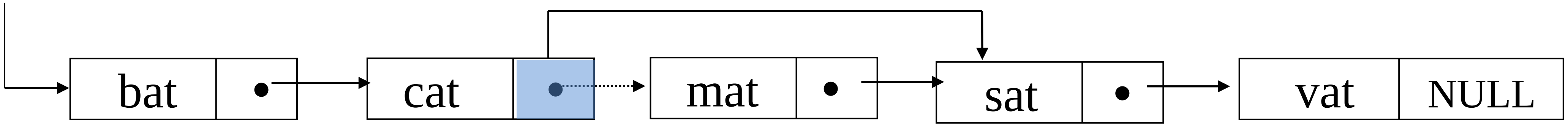
```
head = NULL;
```



- **Insert mat** between **cat** & ***sat***
    1. Get a node that is currently unused; let its address be paddr.
    2. Set the data field of this node to *mat*.
    3. Set paddr's link field to point to the address found in the link field of the node containing *sat*.
    4. Set the link field of the node containing *cat* to point to paddr.



- **Delete mat** from the list
  - Find the element that immediately precedes **mat**, which is **cat**, and set its link field to point to **mat's** link field



- **Implementation**

– Declaration

```
typedef struct  node   *pnode;
typedef struct   node {
        char data [4];
        pnode next;
};
```

– Creation

```
pnode head =NULL;
```

– Testing

```
#define IS_EMPTY(ptr) (!(ptr))
#define IS_FULL(ptr)  (!(ptr))
```

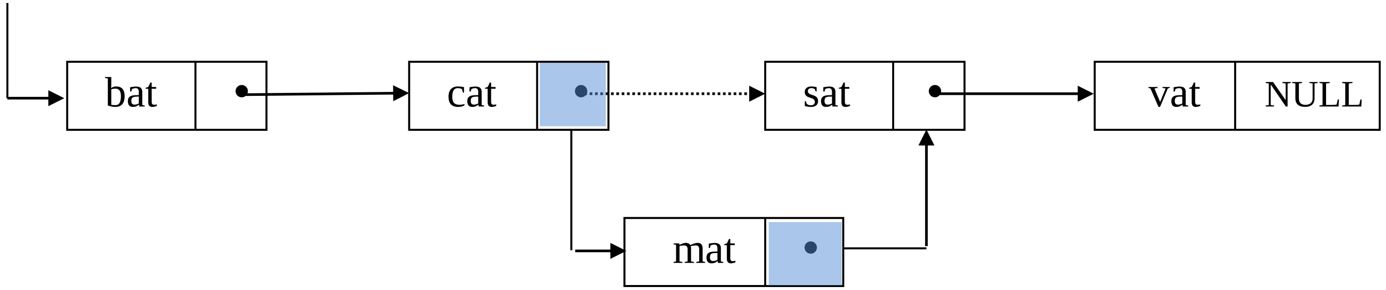
– Traverse a list

```
void traverseList(pnode head){
  printf("The list contains: ");
  for (pnode p = head ; p!=NULL ;
                        p = p->next)
            printf("%s\n", p->data);
    }
```

```
p=head;
while (p!=NULL){
   cout<<P->data;
   p=p->next;
}
```

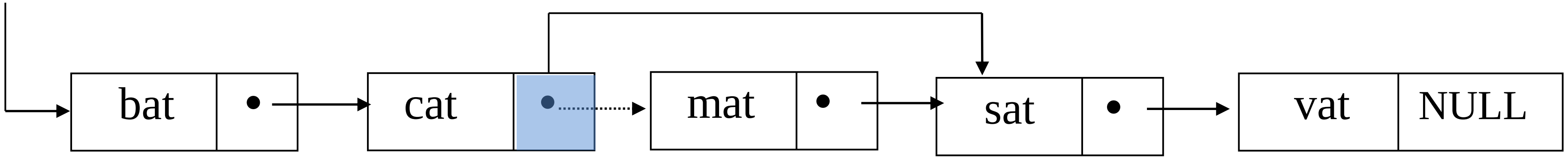
- Implementation - **Insert after a specific position**

```
void insertAfter(pnode p, char* data){
    /* insert a new node with data into the list ptr after node */
    pnode temp;
    temp = (pnode) malloc(sizeof(node));
    if (IS_FULL(temp)){
            fprintf(stderr, “The memory is full\n”);
             exit (1);
    }
    strcpy(temp->data, data);
    if (p) {     //noempty list
             temp->next=p->next;
            p->next= temp;
    }else {                //empty list
            temp->next= NULL;
            p =temp;
    }
}
```



- Implementation - **Delete a node after a specific position**

```
void removeAfter(pnode p){
        /* delete what follows after node p in the list */
        pnode tmp;
        if (p) {
                tmp = p -> next;
                p->next = p->next->next;
                free(tmp);
        }
}
```



- Implementation - **Inverting a list**

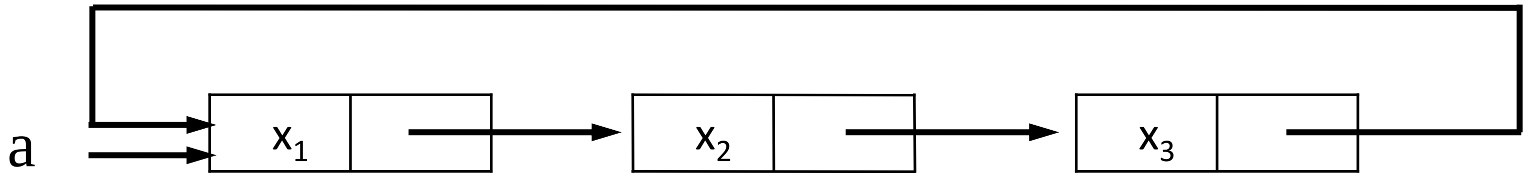
```
pnode  invertList(pnode lead){
        /* invert the chain pointed to by lead */
        pnode middle, trail;
        middle = NULL;
        while (lead) {
                trail = middle;
                middle = lead;
                lead = lead->next;
                middle->next = trail
        }
        return middle;
}
```

# CONTENT

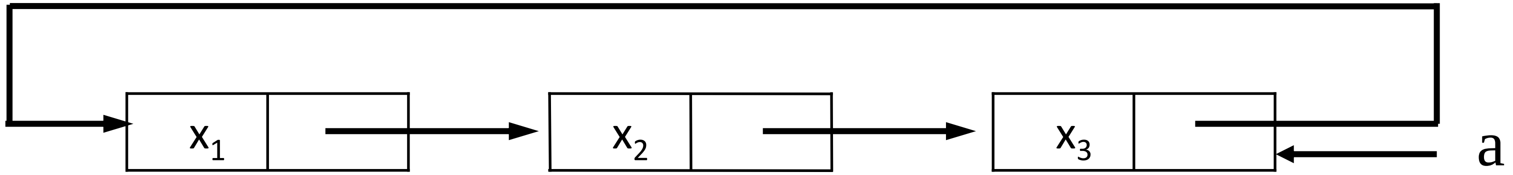
- Linked List Basics
- Singly linked lists
- **Circularly linked list**
- Doubly linked lists
- Applications
  - Polynomial representation
  - Equivalence relations
  - Sparse matrices

- Circularly linked list

  - The link field of the last node points to the first node in the list



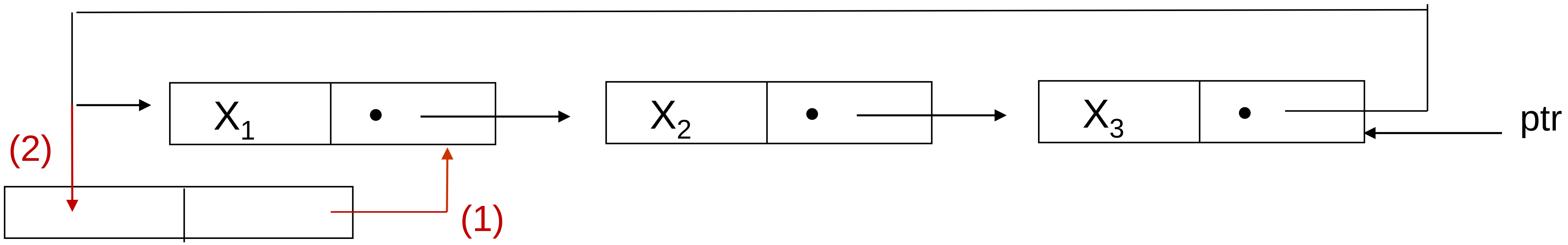

  - It is more convenient when insert a new node if the name of the circular list points to the last node



- **Insert a node**

```
void insertFront (pnode* ptr, pnode node){
        /* insert a node in the list with head (*ptr)->next */
        if (IS_EMPTY(*ptr)){
                *ptr= node;
                node->next = node;      /* circular link */
        }
        else {
                node->next = (*ptr)->next;     (1)
                (*ptr)->next = node;           (2)
        }
}
```

```
typedef struct node *pnode;
typedef struct node {
        char data;
        pnode next;
};
```



- **List length**

```
int length(pnode ptr){
    pnode temp;
    int count = 0;
    if (ptr) {
        temp = ptr;
        do {
            count++;
            temp = temp->next;
        } while (temp!=ptr);
    }
    return count;
}
```

- **Print list**

```
void printList(pnode start, pnode ptr){
    if (start == ptr) return;
    if (ptr) printf("%c  ", ptr->data);
    printList(start, ptr->next);
}
```

⇨**Use:** printList(start, start->next);

- **Other operations**
  - Create a node (with data)
  - Delete a node (with data)
  - Find a node

# CONTENT

- Linked List Basics
- Singly linked lists
- Circularly linked list
- **Doubly linked lists**
- Applications
  - Polynomial representation
  - Equivalence relations
  - Sparse matrices

- **Singly linked lists**
  - Some operations are expensive
    - insertLast, removeAfter
    - Why? ⇨ **need for traversing the list**

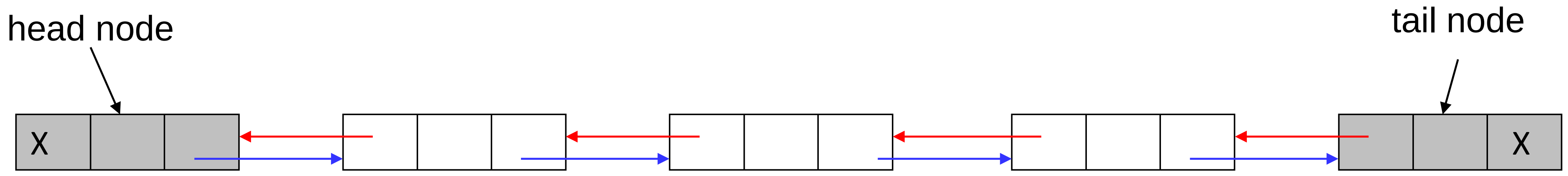
  ⇨Solution: add previous link to elements

- **Doubly linked list has at least three fields:**
  - a left link field (llink)
  - a data field (item)
  - a right link field (rlink)

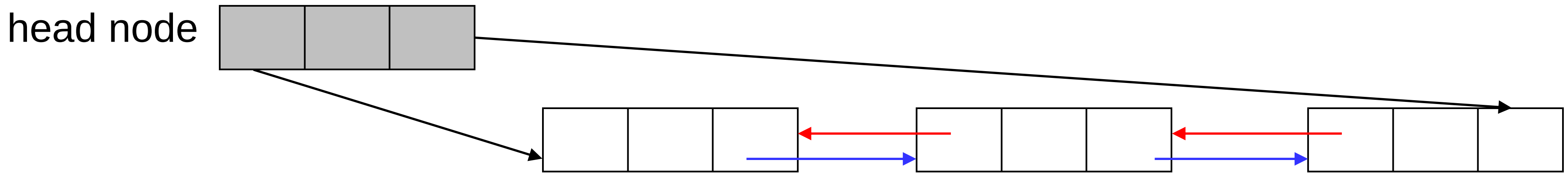
- **Declarations**

```
typedef struct node *node_pointer;
typedef struct node {
        node_pointer llink;
        element item;
        node_pointer rlink;
}
```

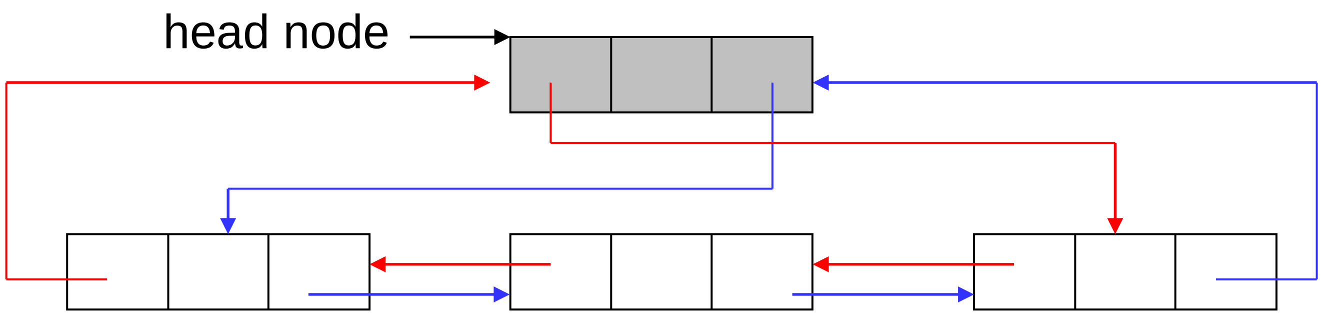
- **Different uses**
  - Doubly linked list with a head node and a tail node




  - Doubly linked list with a head node points to the first node in the list *and* to the last node in the list

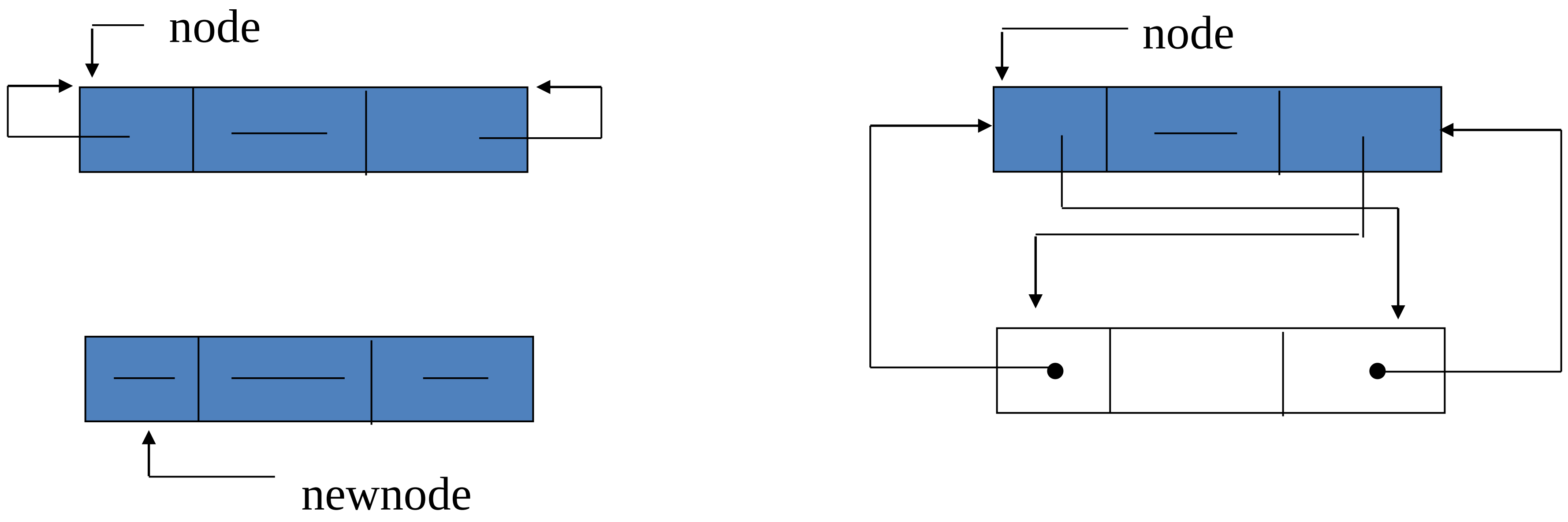


- **Different uses**
  - Doubly linked circular list with head node

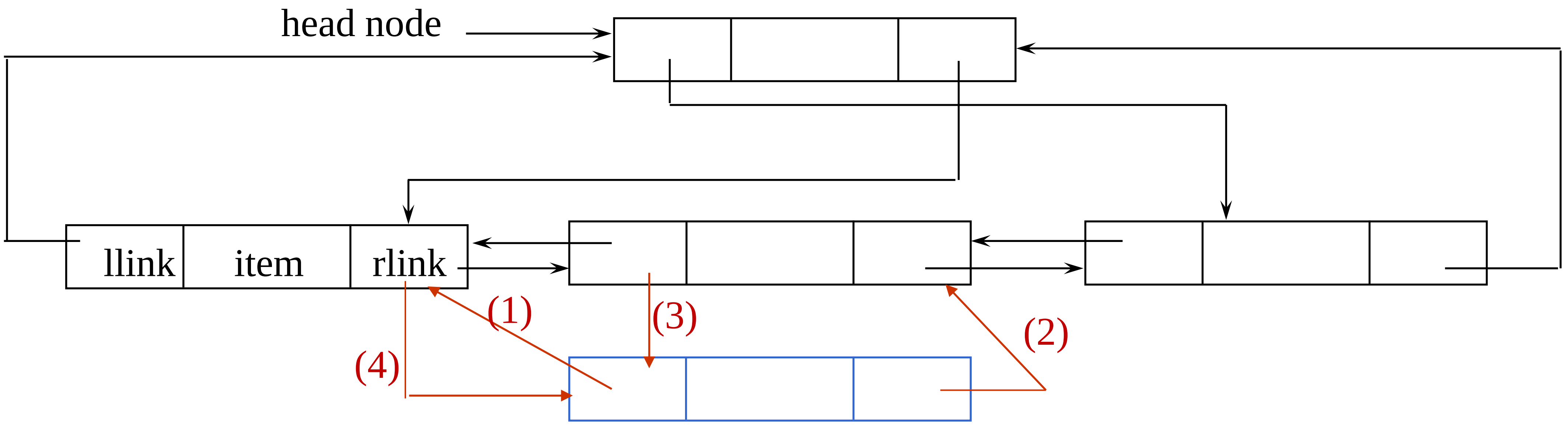



  - Empty doubly linked circular list with head node

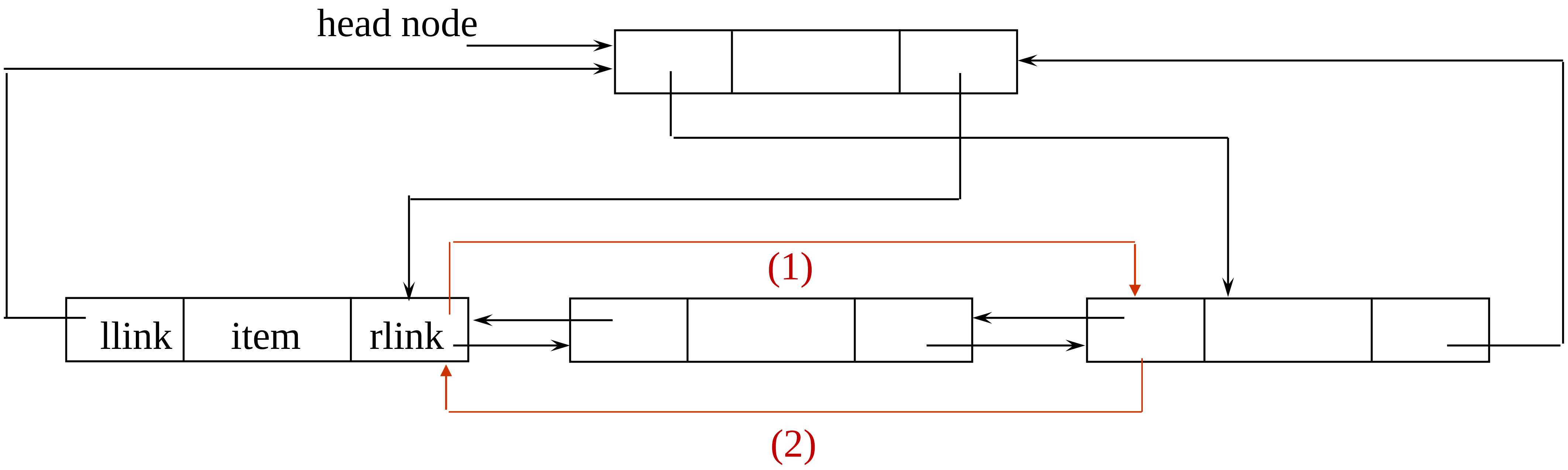
- **Different uses**
  - Doubly linked circular list with head node
    - Insertion into an empty doubly linked circular list



- **Different uses**
  - Doubly linked circular list with head node
    - Insertion into a doubly linked circular list



- **Different uses**
  - Doubly linked circular list with head node
    - Deletion from a doubly linked circular list



## CONTENT

- Linked List Basics
- Singly linked lists
- Stacks and Queues
- Circularly linked list
- Doubly linked lists
- **Applications**
  - Polynomial representation
  - Equivalence relations
  - Sparse matrices

- **Polynomials representation**
  - Representing polynomials as singly linked lists

$$A(x) = a_{m-1}x^{e_{m-1}} + a_{m-2}x^{e_{m-2}} + \ldots + a_0 x^{e_0}$$

$a_i$ are nonzero coefficients, $e_i$ are nonnegative integer exponents such that

$$e_{m-1} > e_{m-2} > \ldots > e_1 > e_0 \geqq 0$$

  - Each term as a node containing coefficient , exponent, as well as a pointer to the next term

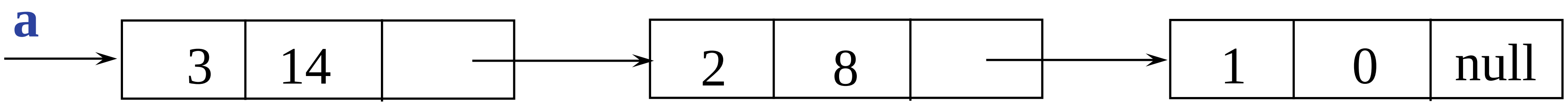
| | | |
|---|---|---|
| coef | expon | link |

- **Declarations**

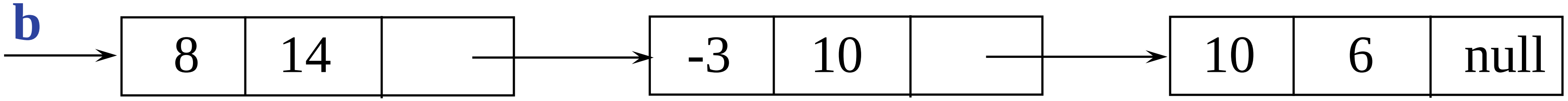
```
typedef struct poly_node    *poly_pointer;
typedef struct poly_node {
            int coef;
            int expon;
            poly_pointer  link;
 };
poly_pointer   a , b, c;
```

- **Polynomials representation - Example**

$$a = 3x^{14} + 2x^{8} + 1$$




$$b = 8x^{14} - 3x^{10} + 10x^{6}$$



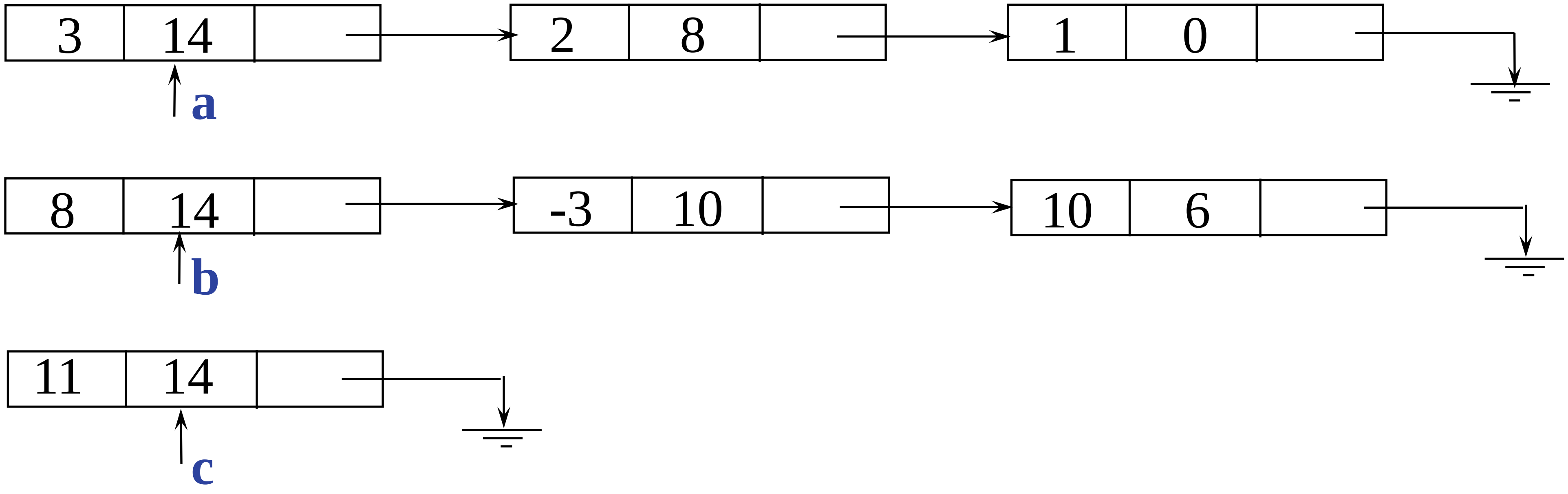
- **Add two polynomials**

To add 2 polynomials, we examine their terms starting at the nodes pointed to by **a** and **b**, there is 3 cases:

1. If the exponents of the two terms are equal, we add the two coefficients and create a new term c for the result
2. If the exponent of the current term in a is less than the exponent of the current term in b, then we create a duplicate term of b, attach this term to the result, called c, and advance the pointer to the next term in b.
3. Take a similar action on a if a->expon > b->expon

## 1. a->expon == b->expon

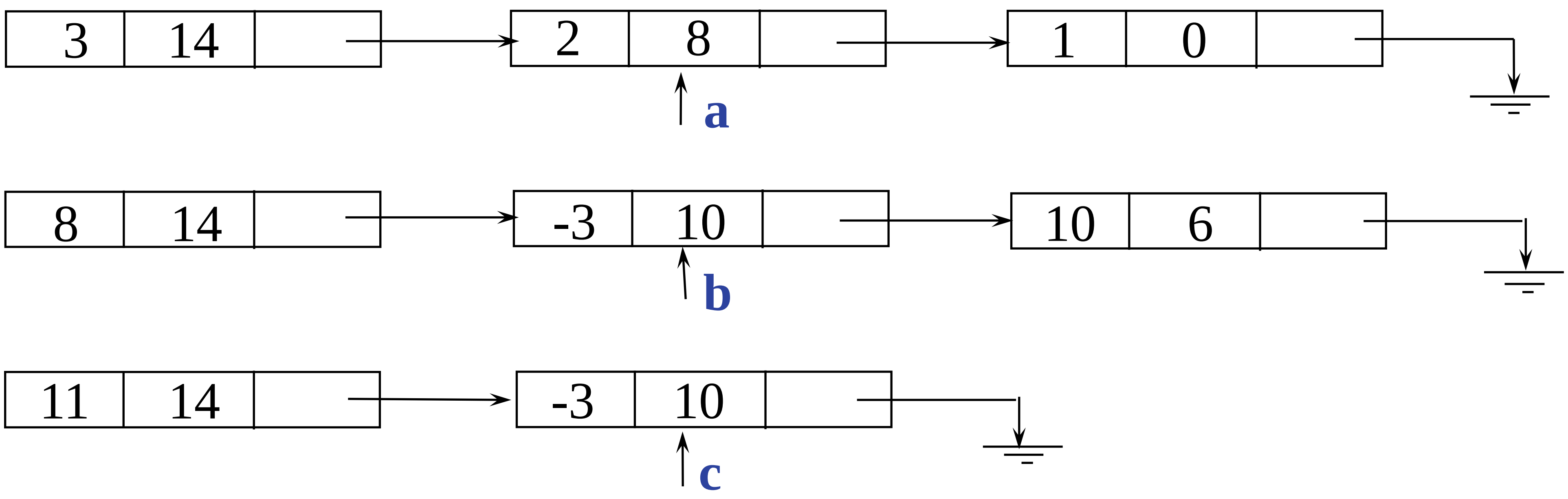
If the exponents of the two terms are equal, we add the two coefficients and create a new term c for the result




```
a->expon == b->expon
```

## 2. a->expon < b->expon

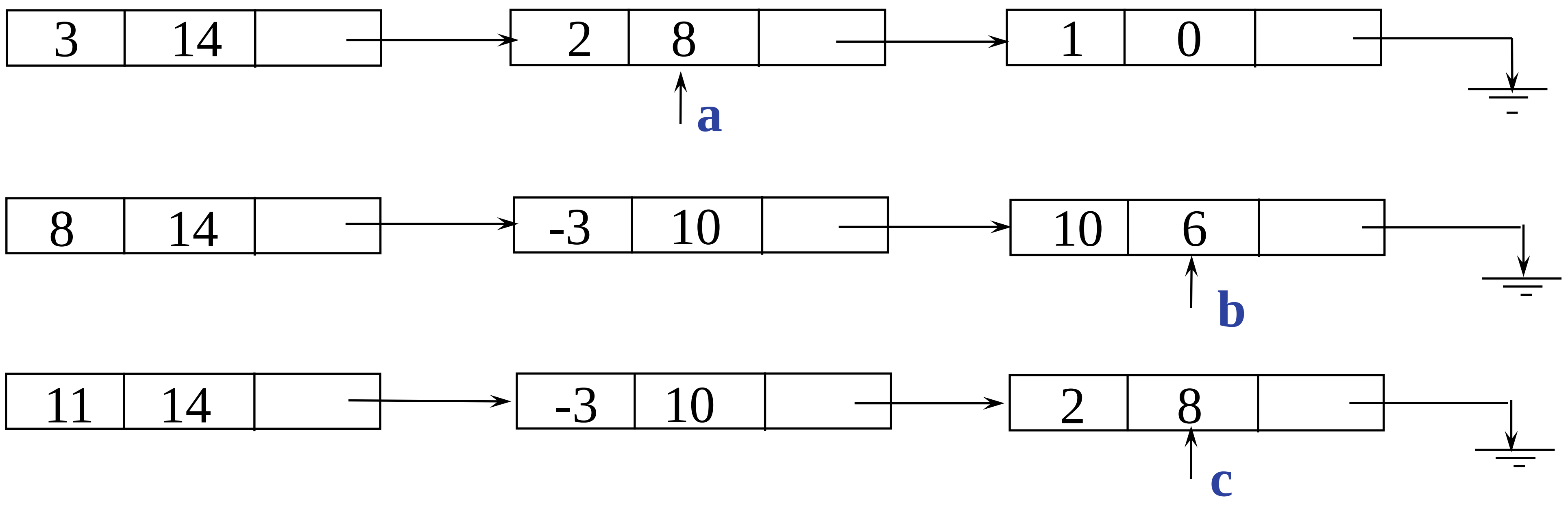
If the exponent of the current term in a is less than the exponent of the current term in b, then we create a duplicate term of b, attach this term to the result, called c, and advance the pointer to the next term in b.




a->expon < b->expon

## 3. a->expon > b->expon

Take a similar action on a if a->expon > b->expon




`a->expon > b->expon`

- **Function Add two polynomials**

```
poly_pointer addPoly(poly_pointer a, poly_pointer b) {
    poly_pointer front, rear, temp;
    int sum;
    rear =(poly_pointer)malloc(sizeof(poly_node));
    if (IS_FULL(rear)) {
        fprintf(stderr, "The memory is full\n");
        exit(1);
    }
    front = rear;
    while (a && b) {
        switch (COMPARE(a->expon, b->expon)) {
```

- **Function Add two polynomials**

```
        case -1:          /* a->expon < b->expon */
                attach(b->coef, b->expon, &rear);
                b= b->link;
                break;
        case 0:           /* a->expon == b->expon */
                sum = a->coef + b->coef;
                if (sum)        attach(sum,a->expon,&rear);
                a = a->link;    b = b->link;
                break;
        case 1:           /* a->expon > b->expon */
                attach(a->coef, a->expon, &rear);
                a = a->link;
        } // end switch
} //end while
```

- **Function Add two polynomials**

```
for (; a; a = a->link)
        attach(a->coef, a->expon, &rear);
 for (; b; b = b->link)
        attach(b->coef, b->expon, &rear);
  rear->link = NULL;
  temp = front;
  front = front->link;
  free(temp);
  return front;
} // end function
```

Delete extra initial node.

- **Attach a term**

```
void attach(float coefficient, int exponent, poly_pointer *ptr){
        /* create a new node attaching to the node pointed to
           by ptr. ptr is updated to point to this new node. */
        poly_pointer temp;
        temp = (poly_pointer) malloc(sizeof(poly_node));
        if (IS_FULL(temp)) {
                fprintf(stderr, “The memory is full\n”);
                exit(1);
        }
        temp->coef = coefficient;
        temp->expon = exponent;
        (*ptr)->link= temp;
        *ptr = temp;
}
```

- **Analysis**

**(1) coefficient additions**

$0 \leq$ additions $\leq \min(m, n)$

where m (n) denotes the number of terms in a (b).

**(2) exponent comparisons**

extreme case

em-1 > fm-1 > em-2 > fm-2 > … > e0 > f0

m+n-1 comparisons

**(3) creation of new nodes**

extreme case

m + n new nodes

summary O(m+n)

- **Erasing polynomials**

```
void erase(poly_pointer *ptr){
     /* erase the polynomial pointed to by ptr */
    poly_pointer temp;
    while (*ptr) {
            temp = *ptr;
            *ptr = (*ptr)->link;
            free(temp);
    }
}
```

- **Equivalence relations**
  - A relation over a set, S, is said to be an *equivalence relation* over S *iff* it is **symmertric**, **reflexive**, and **transitive** over S.
    - Reflexivity: x=x
    - Symmetry: if x=y, then y=x
    - Transitivity: if x=y and y=z, then x=z

- **Example**

0=4, 3=1, 6=10, 8=9, 7=4, 6=8, 3=5, 2=11, 11=0

⇨ three equivalent classes:

{0,2,4,7,11}; {1,3,5}; {6,8,9,10}

- **Algorithm to find Equivalence Classes**

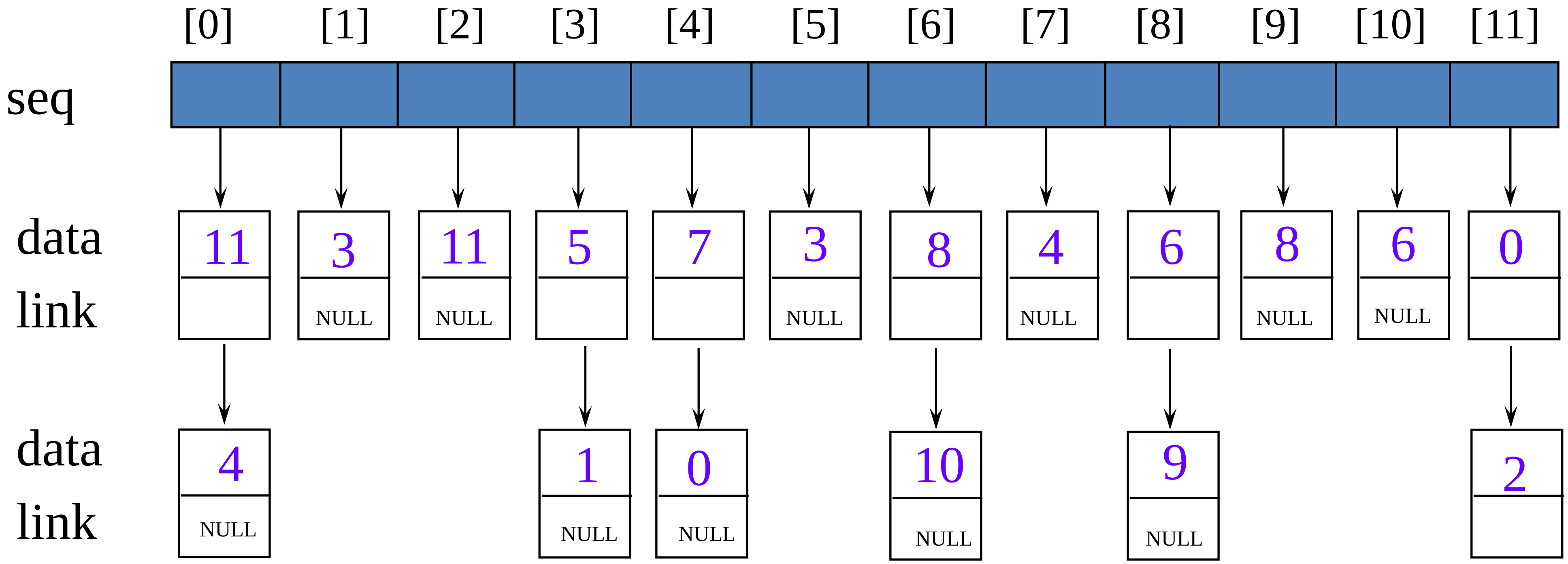
```
void equivalence() {
    initialize data structures;
    while (there are more pairs) {
        read the next pair <i,j>;
        process this pair;
    }
    initialize the output;
    do {
        output a new equivalence class;
    } while (not done);
}
```

- **More detailed Algorithm to find Equivalence Classes**

```
void equivalence() {
    initialize seq to NULL and out to TRUE;
    while (there are more pairs) {
        read the next pair, <i,j>;
        put j on the seq[i] list;
        put i on the seq[j] list;
    }
    for (i=0; i<n; i++)
        if (out[i]) {
            out[i]= FALSE;
            output this equivalence class;
            compute indirect equivalence using transitivity by using stack;
        }
}
```

- **Illutration**

$0 \equiv 4$
$3 \equiv 1$
$6 \equiv 10$
$8 \equiv 9$
$7 \equiv 4$
$6 \equiv 8$
$3 \equiv 5$
$2 \equiv 11$
$11 \equiv 0$




Example: 0=4, 3=1, 6=10, 8=9, 7=4, 6=8, 3=5, 2=11, 11=0

⇨ three equivalent classes: {0,2,4,7,11}; {1,3,5}; {6,8,9,10}

- **Program** (1/4)

```
#include <stdio.h>
#include <alloc.h>
#define MAX_SIZE 24
#define IS_FULL(ptr)  (!(ptr))
#define FALSE  0
#define TRUE   1
typedef struct node *node_pointer ;
typedef struct node {
        int data;
        node_pointer link;
};
```

- **Program** (2/4)

```
void main(void) {
    short int out[MAX_SIZE];
    node_pointer seq[MAX_SIZE];
    node_pointer x, y, top;
    int i, j, n;
    printf("Enter the size (<= %d) ", MAX_SIZE);
    scanf("%d", &n);
    for (i=0; i<n; i++) {
        out[i]= TRUE;    seq[i]= NULL;
    }
    printf("Enter a pair of numbers (-1 -1 to quit): ");
    scanf("%d%d", &i, &j);
```

- **Program** (3/4)

```
while (i>=0) {                         //Phase 1: input the equivalence pairs:
    x = (node_pointer) malloc(sizeof(node));
    if (IS_FULL(x))
        fprintf(stderr, “memory is full\n”);
        exit(1);
    }
    x->data= j;  x->link= seq[i];  seq[i]= x;      //Insert x to the top of lists seq[i]
    x = (node_pointer) malloc(sizeof(node));
    if (IS_FULL(x))
        fprintf(stderr, “memory is full\n”);
        exit(1);
    }
    x->data= i;  x->link= seq[j];  seq[j]= x;      //Insert x to the top of lists seq[j]
    printf(“Enter a pair of numbers (-1 -1 to quit): “);
    scanf(“%d%d”, &i, &j);
 }
```

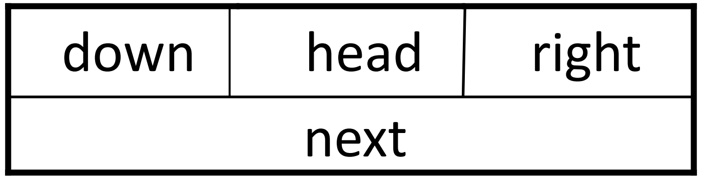
- **Program** (4/4)

```
for (i=0; i<n; i++) {
        if (out[i]) {                                          //Phase 2: output the equivalence classes
                printf("\nNew class: %5d", i);
                out[i]= FALSE;          //mark class as output
                x = seq[i];    top = NULL;          //initialize stack
                for (;;) {    //find the entire class
                        while (x) {                    //process a list
                                        j = x->data;
                                        if (out[j]) {                           // first time, visit this seq[j]
                                        printf("%5d", j);
                                        out[j] = FALSE;
                                        y = x->link;  x->link = top;  //push the linked number to stack if it links to
                                        top = x;  x = y;              // number another
                                } else    x = x->link;
                        }
                        if (!top) break;        //stack empty
                        x = seq[top->data];  top = top->link;      //pop from stack to find the same class number
                }  //for (;;)
}  // for (i=0; i<n; i++)
} //main()
```

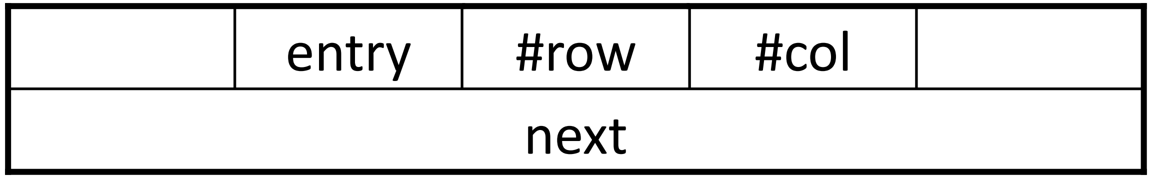
- **Sparse matrices**
  - Each column of a sparse matrix is represented as a circularly linked list with a head node
  - A similar representation for each row of a sparse matrix
  - Each node has a tag field that is used to distinguish between head nodes and entry nodes
  - Each **head node** has three fields: down, right, and next
    - down field: links into a column list
    - right field: links into a row list
    - next field: links the head nodes together
  - The head node for row i is also the head node for column i, and the total number of head nodes is max {number of rows, number of columns}

- **Each entry node has 6 fields: tag, row, col, down, right, value.**
  - down field: links to the next nonzero term in the same column
  - right field: links to the next nonzero term in the same row
  - tag field: entry
  - row field: row index
  - col field: column index
  - value field: nonzero value
- **A *num_rows* × *num_cols* matrix with *num_terms* nonzero terms needs**
  **max{*num_rows, num_cols*} + *num_terms* + 1 node**
  - max{*num_rows, num_cols*}: number of head nodes
  - *num_terms*: number of nonzero terms
  - 1 node: a special head node for the list of row and column head nodes contains the dimensions of the matrix
- **Total storage will be less than *num_rows* × *num_cols* when *num_terms* is sufficiently small**

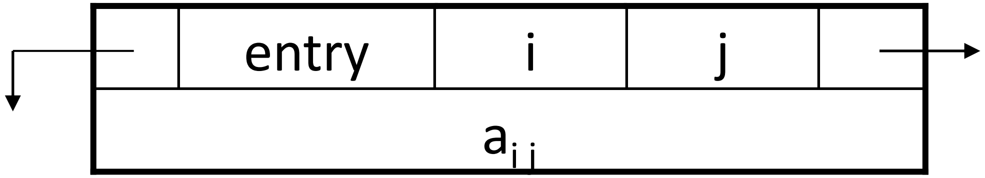
- **Sparse matrices**



head node



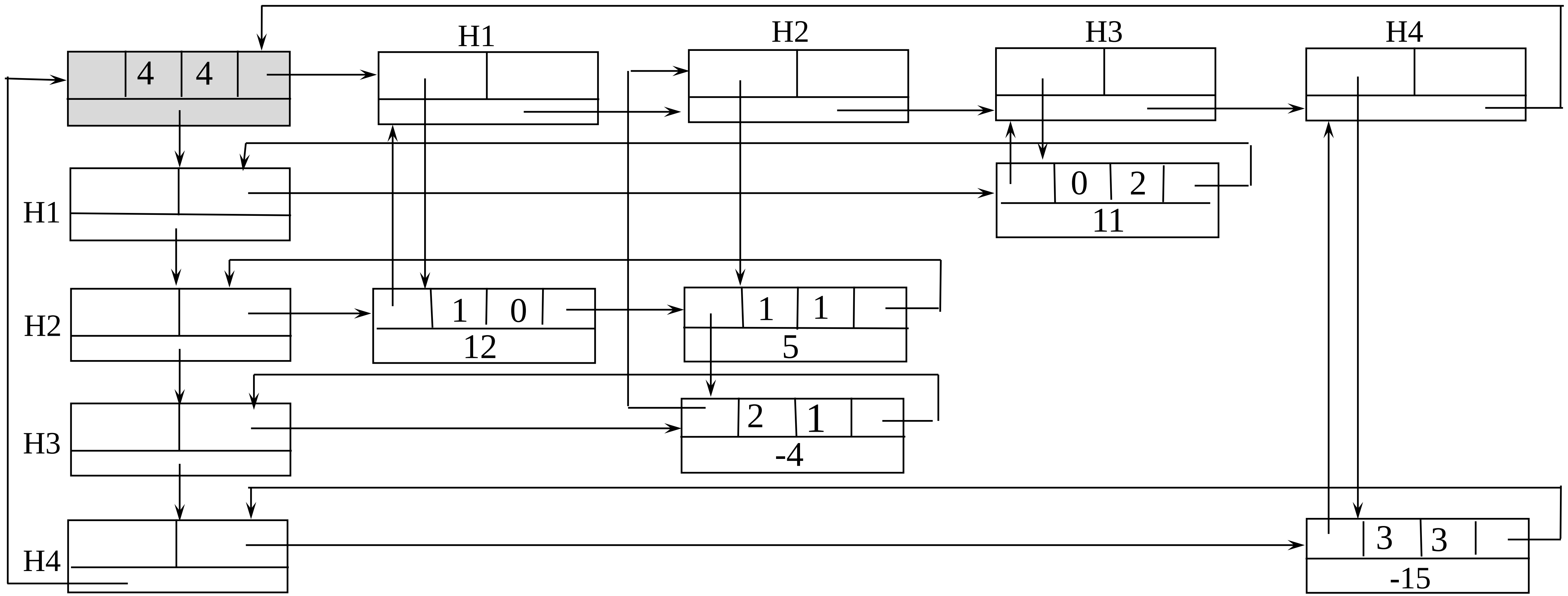
special head node of the list of head nodes



entry of $a_{ij}$

\# of head nodes = max{# of rows, # of columns}

$$\begin{bmatrix} 0 & 0 & 11 & 0 \\ 12 & 5 & 0 & 0 \\ 0 & -4 & 0 & 0 \\ 0 & 0 & 0 & -15 \end{bmatrix}$$



- **Program** (1) **- Declarations**

```
#include <stdio.h>
#include <stdlib.h>
#define MAX_SIZE  25
typedef enum {head,entry} tagfield;
typedef struct MatrixNode *MatrixPointer;
struct EntryNode {
    int row;
    int col;
    int value;
};
```

```
struct MatrixNode {
    MatrixPointer down;
    MatrixPointer right;
    tagfield tag;
    union {
        MatrixPointer next;
        struct EntryNode entry;
    } u;
};
MatrixPointer HdNode[MAX_SIZE];
```

- **Program** (2) **- Read in a matrix and set up its linked representation**

```
MatrixPointer readM(void){
    /* read in a matrix and set up its linked representation.
    An auxilliary global array HdNode is used */
    int NumRows, NumCols,  NumEntries, NumHeads, i;
    int row, col, value, CurrentRow;
    MatrixPointer temp,last,node;

    printf("Enter the number of rows, columns and entries: ");
    scanf("%d,%d,%d",&NumRows, &NumCols, &NumEntries);
    NumHeads = (NumCols > NumRows) ? NumCols : NumRows;
    /* set up head node for the list of head nodes */
    node = (MatrixPointer)malloc(sizeof(struct MatrixNode));
    node->tag = entry;
    node->u.entry.row = NumRows;
    node->u.entry.col = NumCols;
```

- **Program** (3) **- Read in a matrix and set up its linked representation**

```
if (!NumHeads)    node->right = node;         /* when list of head nodes is empty */
else {                                         /* initialize the head nodes */
        for (i = 0; i < NumHeads; i++) {
                temp = (MatrixPointer)malloc(sizeof(struct MatrixNode));
                HdNode[i] = temp;
                HdNode[i]->tag = head;
                HdNode[i]->right = temp;
                HdNode[i]->u.next = temp;
        }
        CurrentRow = 0;
        last = HdNode[0];
        for (i = 0; i < NumEntries; i++) {
                printf("Enter row, column and value: ");
                scanf("%d,%d,%d",&row,&col,&value);
                if (row > CurrentRow) {
                        last->right = HdNode[CurrentRow];
                        CurrentRow = row;
                        last = HdNode[row];
                }
```

- **Program** (4) **- Read in a matrix and set up its linked representation**

```
            temp = (MatrixPointer)malloc(sizeof(struct MatrixNode));
            temp->tag = entry;              temp->u.entry.value = value;
            temp->u.entry.row = row;        temp->u.entry.col = col;
            last->right = temp;         /* link into row list */
            last = temp;
            HdNode[col]->u.next->down = temp;  /* link into column list */
            HdNode[col]->u.next = temp;
        } // for
        last->right = HdNode[CurrentRow];/*close last row */
        for (i = 0; i < NumCols; i++)                /* close all column lists */
            HdNode[i]->u.next->down = HdNode[i];
        for (i = 0; i < NumHeads-1; i++)             /* link all head nodes together */
            HdNode[i]->u.next = HdNode[i+1];
        HdNode[NumHeads-1]->u.next =  node;
        node->right = HdNode[0];
    } // if
    return node;
}
```

- **Program** (5) **- Print out the matrix in each row**

```
void writeM(MatrixPointer node){  /* print out the matrix in row major form */
        int i;
        MatrixPointer temp;
        printf("\n\nNumRows = %d, NumCols = %d\n",
                node->u.entry.row,  node->u.entry.col);
        printf(" The matrix by row, column, and value: \n\n");
        for (i = 0; i < node->u.entry.row; i++) /* print out the entries in each row */
        for (temp = HdNode[i]->right; temp != HdNode[i]; temp = temp->right)
                printf("%5d%5d%5d\n",temp->u.entry.row,
                        temp->u.entry.col,    temp->u.entry.value);
}
```

- **Program (6) - Erase the matrix**

```
   void merase(MatrixPointer *node){            /* erase the matrix, return the pointers to the
heap */
        MatrixPointer x,y;
        int i, NumHeads;
        for (i = 0; i < (*node)->u.entry.row; i++) {/* free the entry pointers by row */
                y = HdNode[i]->right;
                while (y != HdNode[i]) {
                        x = y;
                        y = y->right;
                        free(x);
                }
        }
        /* determine the number of head nodes and free these pointers */
        NumHeads = ((*node)->u.entry.row > (*node)->u.entry.col) ?
                                        (*node)->u.entry.row : (*node)->u.entry.col;
        for (i = 0; i < NumHeads; i++)
                free(HdNode[i]);
        *node = NULL;
   }
```

# SUMMARY

- Linked List Basics
- Singly linked lists
- Circularly linked list
- Doubly linked lists
- Applications

Nhân bản – Phụng sự – Khai phóng

# Enjoy the Course…!