Chapter 4
LINKED LISTS

Singly Linked Lists and Chains
Representing Chains in C
Linked Stacks and Queues
Polynomials
Additional List Operations
Equivalence Classes
Sparse Matrices
Doubly Linked Lists

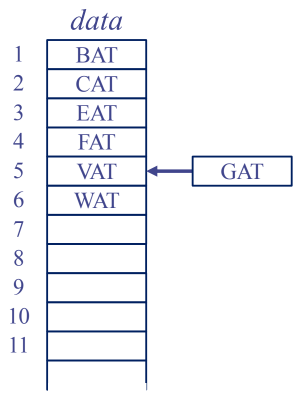
Heung-Il Suk

https://milab.korea.ac.kr

hisuk (AT) korea.ac.kr

Singly Linked Lists and Chains

- Representation of ordered lists
- *Sequential* representation
  - Successive items of a list are located a fixed distance apart
  - Insertion and deletion of arbitrary elements become expensive
- *Linked* representation
  - Items may be *placed anywhere in memory*
  - To access list elements
    - store the *address or location of the next element* in that list



three-letter words

Singly Linked Lists and Chains

- <*data*[*i*], *link*[*i*]> pair comprise a node
  - *data*
    - Elements are no longer in sequential order
  - *link*
    - Values are *pointers* to elements in the *data* array

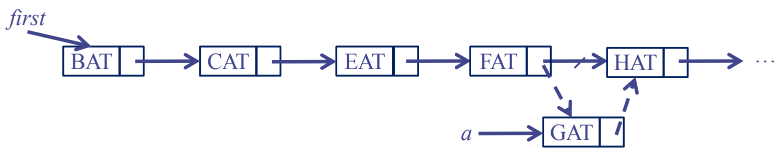
	data	*link*
1	HAT	15
2		
3	CAT	4
4	EAT	9
5		
6		
7	WAT	0
8	BAT	3
9	FAT	1
10		
11	VAT	7

- The list starts at *data*[8]=BAT
  - *first*=8
  - *link*[8]=3, which means it points to *data*[3], which contains CAT
- When we have come to the end of the ordered list
  - *link* equals zero

Usual way to draw a linked list

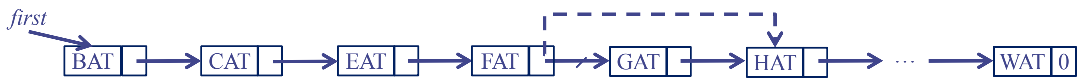
Singly Linked Lists and Chains

- Insertion
  - Inserting GAT into the list
    - *not have to move any elements*



	data	*link*
1	HAT	15
2		
3	CAT	4
4	EAT	9
5	GAT	1
6		
7	WAT	0
8	BAT	3
9	FAT	5
10		
11	VAT	7

- Deletion
  - Deleting GAT from the list
    - Even though the link of GAT still contains a pointer to HAT, GAT is no longer in the list as it cannot be reached by starting at the first element of the list



Representing Chains in C

- Need the following capabilities
  - A mechanism for defining a node's structure, i.e., the fields it contains
    - *Self-referential structures*
  - A way to create new nodes when we need them
    - MALLOC
  - A way to remove nodes that we no longer need
    - free
- Example [*List of words*]

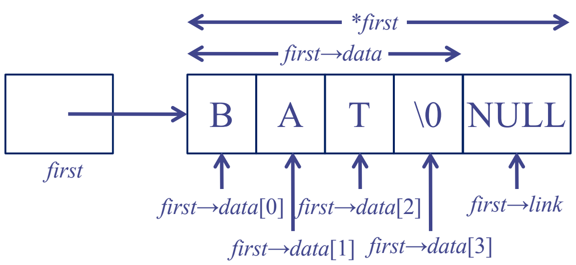
```
typedef struct listNode *listPointer;
typedef struct listNode {
        char          data[4];
        listPointer   link;        /* self-referential structure */
};
```

Defined the pointer (listPointer) to the **struct** before we defined the **struct** (listNode)

C allows us to create a pointer to a type that does not yet exist because otherwise we would face a paradox:
we cannot define a pointer to a nonexistent type,
but to define the new type we must include a pointer to the type

Representing Chains in C

- Example [*List of words*] (cont.)
  - Create a new empty list
    - listPointer first = NULL;   /* contains the address of the start of the list */
  - Macro to test for an empty list
    - #define IS_EMPTY(first)   (!(first))
  - Create a new node
    - MALLOC(first, sizeof(*first));
  - Place the word BAT into the list
    - strcpy(first→data, "BAT");
    - first→link = NULL;



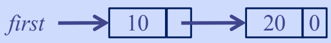
Representing Chains in C

- Example [*Two-node linked list*]
  - a linked list of integers

```
typedef struct listNode *listPointer;
typedef struct listNode {
        int             data;
        listPointer     link;       /* self-referential structure */
};
```

```
void printList( listPointer first )
{
        printf( "The list contains: " );
        for( ; first; first = first→link )
                printf( "%4D", first->data );
        printf( "\n" );
}
```

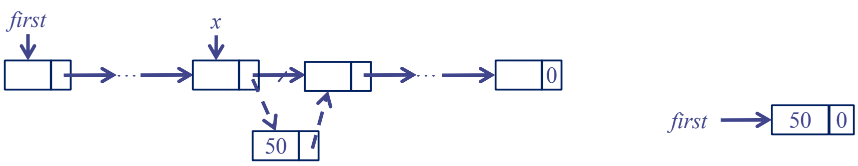
```
listPointer create2()
{       /* create a linked list with two nodes */
        listPointer first, second;
        MALLOC( first, sizeof(*first) );
        MALLOC( second, sizeof(*second) );
        second→link = NULL;
        second→data = 20;
        first→data=10;
        first→link = second;
        return first;
}
```



Representing Chains in C

- Example [*List insertion*]

```
void insert( listPointer *first, listPointer x )
{       /* insert a new node with data=50 into the chain first after node x */
        listPointer temp;
        MALLOC( temp, sizeof(*temp) );
        temp→data = 50;
    (A) if( *first ) {
                temp→link = x→link;
                x→link = temp;
        }
    (B) else {
                temp→link = NULL;
                *first = temp;
        }
}
```



(A) (B)

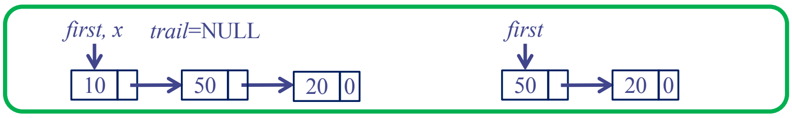
Inserting into an empty and nonempty list

Representing Chains in C

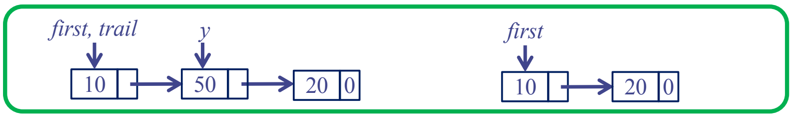
- Example [*List deletion*]

```
void delete( listPointer *first, listPointer trail, listPointer x )
{               /* delete x from the list, trail is the preceding node and *first is the front of the list */
                if( trail )
                                trail→link = x→link;
                else
                                *first = (*first)→link;

                free(x);
}
```



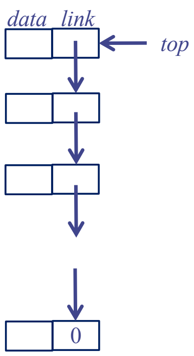
List before and the function call delete(&*first*, NULL, *first*);



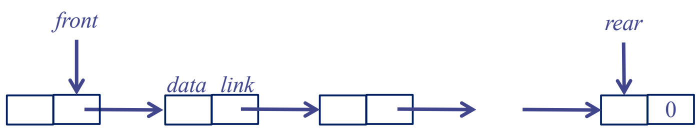
List after the function call delete(&*first*, *y*, *y*→*link*);

Linked Stacks and Queues

- When several stacks and queues coexisted there was no efficient way to represent them sequentially
- Linked stack and queue
  - The direction of links for both the stack and the queue facilitate easy insertion and deletion of nodes



Linked stack



Linked queue

Representation *n* Stacks

```
#define MAX_STACKS 10   /* maximum number of stacks */
typedef struct {
        int     key;
        /* other fields */
} element;

typedef struct stack *stackPointer;
typedef struct stack {
        element data;
        stackPointer link;
};

stackPointer top[MAX_STACKS];
```

- Assume that the initial condition for the stack
  - ✓ top[i] = NULL, 0<= i < MAX_STACKS
- boundary condition
  - ✓ top[i] = NULL iff the ith stack is empty

Machine Intelligence Lab

Push and Pop in the Linked Stacks

```
void push( int i, element item )
{
        /* add item to the ith stack */
        stackPointer temp;
        MALLOC( temp, sizeof(*temp) );

        temp→data = item;
        temp→link = top[i];
        top[i] = temp;
}

element pop( int i )
{
        /* remove top element from the ith stack */
        stackPointer temp = top[i];
        element item;
        if( !temp )
                return stackEmpty();
        item = temp→data;
        top[i] = temp→link;
        free( temp );           /* return to system memory */

        return item;
}
```

Machine Intelligence Lab

Representation *m* Queues

```
#define MAX_QUEUES 10  /* maximum number of queues */
typedef struct {
            int       key;
            /* other fields */
} element;

typedef struct queue *queuePointer;
typedef struct queue {
            element data;
            queuePointer link;
};

queuePointer front[MAX_STACKS], rear[MAX_STACKS];
```

- Assume that the initial condition for the queue
  - ✓ front[i] = NULL, 0<= i < MAX_QUEUES
- boundary condition
  - ✓ front[i] = NULL iff the ith queue is empty

Machine Intelligence Lab

Add to the Rear of a Linked Queue

```
void addq( int i, element item )
{
                /* add item to the rear of queue i */
                queuePointer temp;
                MALLOC( temp, sizeof(*temp) );
                temp→data = item;
                temp→link = NULL;

                if( front[i] )
                                rear[i]→link = temp;
                else
                                front[i] = temp;

                rear[i] = temp;
}
```

Machine Intelligence Lab

Polynomials

- Representing polynomials using linked lists

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

where the a_i are nonzero coefficients and the e_i are nonnegative integer exponents such that $e_{m-1} > e_{m-2} > \ldots > e_1 > e_0 \geq 0$

- We will represent each term as a node containing
  - *coefficient field*
  - *exponent fields*
  - *a pointer to the next term*

Representation of Polynomial

- Declaration of polynomial terms

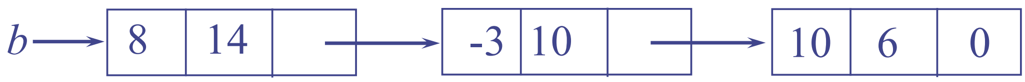
```
typedef struct polyNode *polyPointer;
typedef struct polyNode {
            int coef;
            int expon;
            polyPointer link;
};
polyPointer a, b, c;
```

coef	expon	link

Polynomials

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

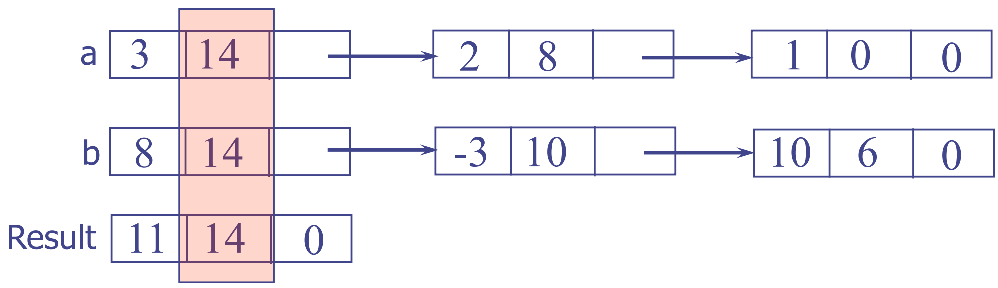
$$b = 8x^{14} - 3x^{10} + 10x^{6}$$



Example: Adding Two Polys

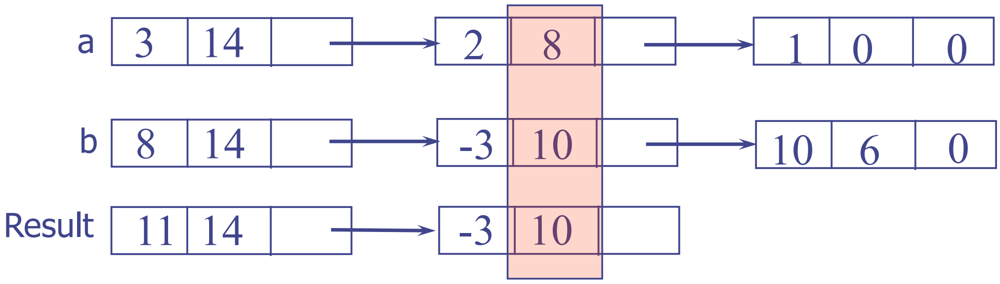
$$a = 3x^{14} + 2x^{8} + 1$$

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

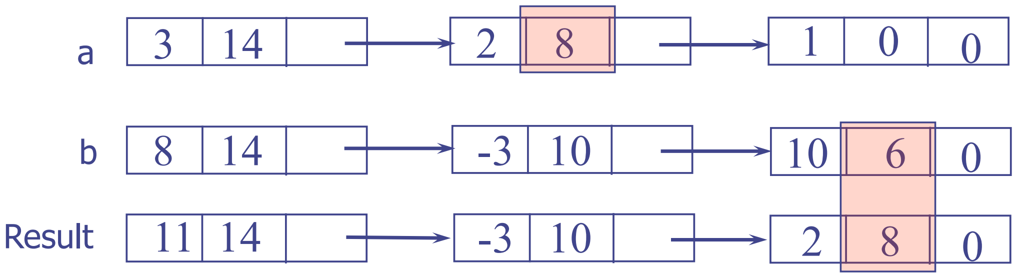


`a→expon == b→expon`

Example (cont.)

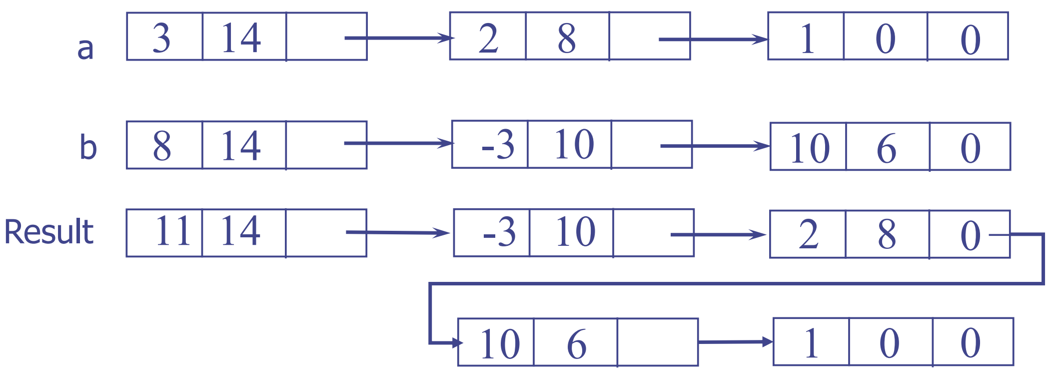


a→expon < b→expon



a→expon > b→expon

Example (cont.)



Add Two Polynomials

```
polyPointer padd(polyPointer a, polyPointer b)
{           /* return a polynomial which is the sum of a and b */
            polyPointer c, rear, temp;
            int sum;
            MALLOC( rear, sizeof(*rear) );
            c = rear;       /* initially give c a single node with no values */
            while (a && b) {
                        switch ( COMPARE(a→expon, b→expon) ) {
                                    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;
                        }
            }
            /* copy rest of list a and then list b */
            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;
            /* delete extra initial node */
            temp = c;     c = c→link;   free(temp);
            return c;
}
```

To avoid having to search for the last node in *c* each time we add a new node, we keep a pointer, *rear*, which points to the current last node in *c*.

Attaching A Term

```
void attach(float coefficient, int exponent, polyPointer *ptr)
{           /* create a new node with coef = coefficient and expon = exponent, attach it to the node
               pointed to by ptr. ptr is updated to point to this new node */

            polyPointer temp;
            MALLOC( temp, sizeof(*temp) );
            temp→coef = coefficient;
            temp→expon = exponent;
            (*ptr)→link = temp;
            *ptr = temp;
}
```

Erasing Polynomials

- Assume to compute e(x) = a(x) * b(x) + d(x)

```
polyPointer a, b, d, e;
...
a = readPoly();
b = readPoly();
d = readPoly();
temp = pmult(a, b); /* only hold a partial result for d(x) */
e = padd(temp, d);
printPoly(e);
```

  - We created *temp(x)* only to hold a partial result for *d(x)*
    - It would be useful to reclaim the nodes that are being used to represent *temp(x)*

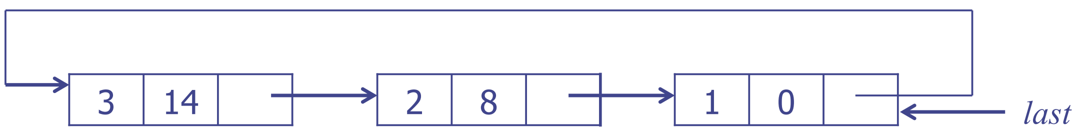
- Erase()

```
void earse(polyPointer *ptr)
{           /* erase the polynomial pointed to by ptr */
            polyPointer temp;
            while (*ptr)      {
                        temp = *ptr;
                        *ptr = (*ptr)→link;
                        free(temp);
            }
}
```


Circular List Representation of Polynomials

- If the link field of the last node points to the first node in the list, all the nodes of a polynomial can be freed more efficiently

- Circular representation of $3x^{14} + 2x^{8} + 1$



Available Space List

- Chain: a singly linked list in which the last node has a null link

- An efficient erase algorithm for circular lists, by maintaining a list (as a chain) of nodes that have been "freed"
  - When a new node is needed, examine this list
  - If the list is not empty, then we may use one of its nodes
  - Only need to use *malloc* to create a new node when the list is empty
- This list is called the available space list or *avail* list
  - Initially, set *avail* to NULL
- Instead of using *malloc* and *free*, now use *getNode* and *retNode*

getNode Function

```
polyPointer getNode(void)
{          /* provide a node for use */
           polyPointer node;
           if ( avail ) {
                      node = avail;
                      avail = avail→link:
           }
           else
                      MALLOC( node, sizeof(*node) );

           return node;
}
```

retNode And *cerase* Function

- retNode
  - Return a node to the available list

```
void retNode(polyPointer node)
{
        node→link = avail;
        avail = node;
}
```

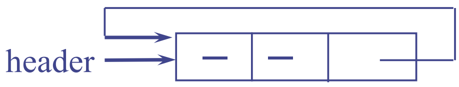
- cerase
  - Erase a circular list in a fixed amount of time independent of the number of nodes in the list

```
void cerase( polyPointer *ptr )
{       /* erase the circular list pointed to by ptr */
        polyPointer temp;
        if (*ptr) {
                temp = (*ptr)→link;
                (*ptr)→link = avail;
                avail = temp;
                *ptr = NULL;
        }
}
```

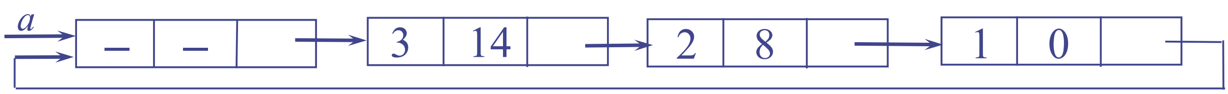

Zero Polynomial

- To avoid the special case of zero polynomial, each polynomial contains one additional node, a header node
  - The *expon* and *coef* fields of this node are irrelevant

- The representation
  - Zero polynomial



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



Summary

- Two polynomials represented as circular lists with header node

```
polyPointer cpadd(polyPointer a, polyPointer b)
{               /* Polynomials a and b are singly linked circular lists with a header node.
                   Return a polynomial which is the sum of a and b */
                polyPointer startA, c, lastC;
                int sum, done = FALSE;
                startA = a;                 /* record start of a */
                a = a→link;                 /* skip header node for a and b */
                b = b→link;
                c = getNode();              /* get a header node for sum */
                c→expon = -1;
                lastC = c;


                /* continued… */
```


Summary(cont.)

```
                do {
                        switch (COMPARE(a→expon, b→expon))
                        {
                                case  -1:       /* a→expon < b→expon */
                                                attach(b→coef, b→expon, &lastC);
                                                b = b→link;
                                                break;

                        case  0:        /* a→expon = b→expon */
                                if (startA == a)
                                                done = TRUE;
                                else {
                                                sum = a→coef + b→coef;
                                                if  (sum)    attach(sum, a→expon, &lastC);
                                                a = a→link;   b = b→link;
                                }
                                break;

                        case  1:        /* a→expon > b→expon */
                                attach(a→coef, a→expon, &lastC);
                                a = a→link;
                        }
                } while ( !done );

                lastC→link = c;
                return c;
}
```

Additional List Operations

- Inverting (or reversing) a singly linked list
  - Invert the list pointed to by *lead*

```
listPointer invert(listPointer lead)
{
        listPointer middle, trail;
        middle = NULL;
        while (lead) {
                trail = middle;
                middle = lead;
                lead = lead→link;
                middle→link = trail;
        }

        return middle;
}
```


Additional List Operations

- Concatenating two chains

```
listPointer concatenate( listPointer ptr1, listPointer ptr2 )
{       /* produce a new list that contains the list ptr1 followed by the list ptr2.
           The list pointed to by ptr1 is changed permanently */
        listPointer temp;
        /*check for empty lists */
        if( !ptr1 )     return ptr2;
        if( !ptr2 )     return ptr1;

        /* neither list is empty, find end of first list */
        for( temp=ptr1; temp→link; temp=temp→link ) ;

        /* link end of first to start of second */
        temp→link = ptr2;
}
```

Additional List Operations

- Inserting at the front of a circular list

```
void insertFront( listPointer *last, listPointer node )
{       /* insert node at the front of the circular list whose last node is last */
        if( !(*last) ) {
                /* list is empty, change list to point to new entry */
                *last = node;
                node→link = node;
        }
        else {
                /* list is not empty, add new entry at front */
                node→link = (*last)→link;
                (*last)→link = node;
        }
}
```


Length of Linked List

- Finding the length of a circular list

```
int length( listPointer last )
{       /* find the length of the circular list last */
        listPointer temp;
        int count = 0;
        if( last ) {
                temp = last;
                do {
                        count++;
                        temp = temp→link;
                } while( temp != last );
        }

        return count;
}
```

Equivalence Relations

- Reflexive, $x \equiv x$
- Symmetric, if $x \equiv y$, then $y \equiv x$
- Transitive, if $x \equiv y$ and $y \equiv z$, then $x \equiv z$

- Definition : equivalence relations
  - A relation, $\equiv$, over a set, S, is said to be an *equivalence relation* over S *iff* it is *symmertric*, *reflexive*, and *transitive* over S

- Examples: partition a set S into equivalence classes
  - $0 \equiv 4$, $3 \equiv 1$, $6 \equiv 10$, $8 \equiv 9$, $7 \equiv 4$, $6 \equiv 8$, $3 \equiv 5$, $2 \equiv 11$, $11 \equiv 0$
  - Three equivalent classes : {0, 2, 4, 7, 11}; {1, 3, 5}; {6, 8, 9, 10}

A Rough Algorithm to Determine Equivalence

- Two phases
  - First phase
    - Read in and store the equivalence pairs $<i, j>$
  - Second phase
    - Begin at 0 and find all pairs of the form $<0, j>$
    - All pairs of the form $<j, k>$ imply that k is in the same equivalence class as 0
    - Continue in this way until we have found, marked, and printed the entire equivalence class containing 0

First phase

Second phase

```
void equivalence()
{
    initialize;
    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);
}
```

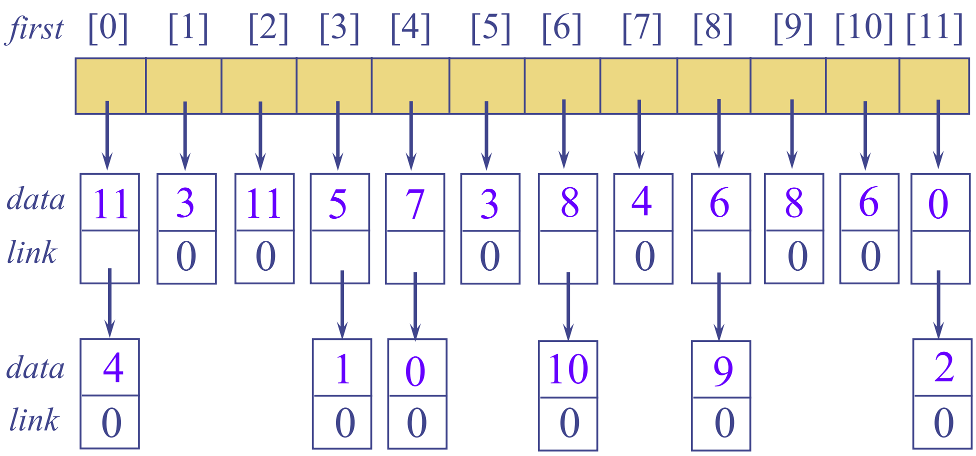
A More Detailed Version of The Equivalence Algorithm

- *seq* holds the header nodes of *n* lists
- *out* tells us whether or not the object *i* has been printed

```
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;
       }
}
```


Lists after pairs have been input

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



Sparse Matrices

- In Chapter 2, we could save space and computing time by retaining only the nonzero terms of sparse matrices

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

Sparse Matrices

- **Each entry node** has five fields in addition to the tag field: *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

next	
down	*right*

header node

row	*col*	*value*
down	*right*	

element node

head field is not shown

Sparse Matrices

- If $a_{ij} \neq 0$, there is a node with tag field = *entry*

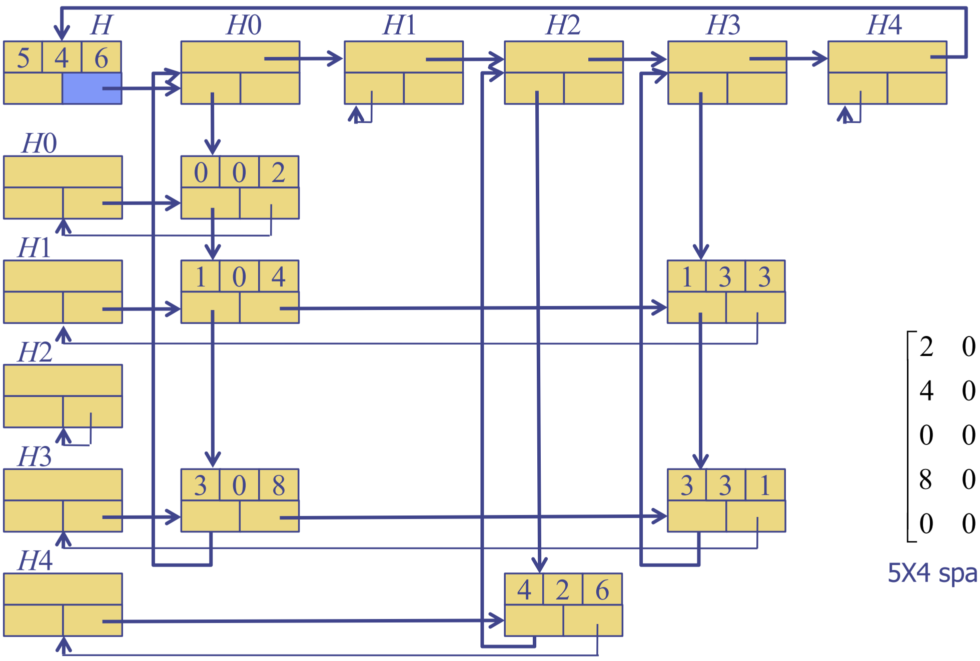
i	j	a_{ij}
down	*right*	

- The list of header nodes has a header node that has the same structure as an entry node

# row	*# col*	*# elements*
down	*right*	

- Total storage
  - A $numRows \times numCols$ matrix with $numTerms$ nonzero terms needs $\max\{numRows\ ,\ numCols\} + numTerms + 1$ nodes
  - Total storage will be less than $numRows \times numCols$ when $numTerms$ is sufficiently small

Linked Representation of The Sparse Matrix



$$\begin{bmatrix} 2 & 0 & 0 & 0 \\ 4 & 0 & 0 & 3 \\ 0 & 0 & 0 & 0 \\ 8 & 0 & 0 & 1 \\ 0 & 0 & 6 & 0 \end{bmatrix}$$

5X4 sparse matrix

Sparse Matrices

- Declarations for matrix representation

```
#define MAX_SIZE 50                /* size of largest matrix */
typedef enum {head, entry} tagfield;
typedef struct matrixNode *matrixPointer;
typedef struct entryNode {
        int row;
        int col;
        int value;
};

typedef struct matrixNode {
        matrixPointer down;
        matrixPointer right;
        tagfield tag;
        union {
                matrixPointer next;
                entryNode entry;
        } u;
};

matrixPointer hdnode[MAX_SIZE];
```

Two different types of nodes

Sparse Matrices

- **[Assignment #2] Programming project (Exercises 6)**
  - Implement a complete linked list system to perform arithmetic on sparse matrices using our linked list representation
  - Create a user-friendly, menu-driven system that performs the following operations
    - *mread* : Read in a sparse matrix (Program 4.23)
    - *mwrite* : Write out a sparse matrix (Program 4.24)
    - *merase* : Erase a sparse matrix (Program 4.25)
    - *madd* : Create the sparse matrix $d = a + b$
    - *mmult* : Create the sparse matrix $d = a * b$
    - *mtranspose :* Create the sparse matrix $b = a^T$
  - Deadline: April 21 (end of the day)

Doubly Linked List

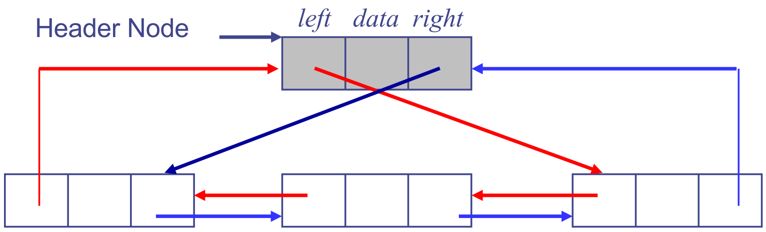
- In singly linked lists, it can move only in the direction of the links
- The only way to find the node that precedes a specific node is to start at the beginning of the list
- It is necessary to move in either direction: doubly linked list

- A node in a doubly linked list has at least three fields, a left link field (*llink*), a data field (*data*), and a right link field (*rlink*)

```
typedef struct node *nodePointer;
typedef struct node   {
            nodePointer llink;
            element data;
            nodePointer rlink;
 }
```

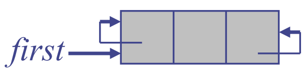

Doubly Linked List

- Doubly linked circular list with a header node



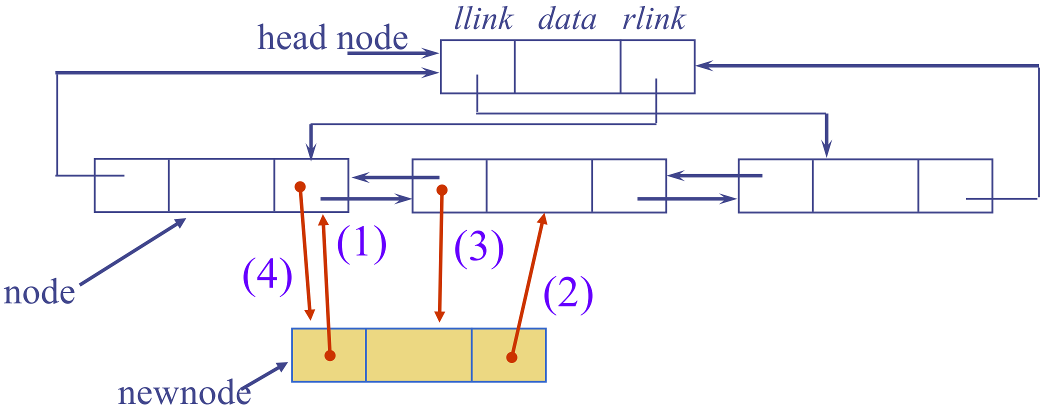
- If *ptr* points to any node in a doubly linked list, then:
  - $ptr = ptr \rightarrow rlink \rightarrow llink = ptr \rightarrow llink \rightarrow rlink$

- An empty doubly linked list



Insert

```
void dinsert(nodePointer node, nodePointer newnode)
{               /* insert newnode to the right of node */
                newnode→llink = node; /* (1) */
                newnode→rlink = node→rlink; /* (2) */
                node→rlink→llink = newnode;  /* (3) */
                node→rlink- = newnode;  /* (4) */
}
```



Delete

```
void ddelete(nodePointer node, nodePointer deleted)
{               /* delete from the double linked list */
                if  (node==deleted)
                                printf("Deletion of head node not permitted.\n");
                else {
                                deleted→llink→rlink= deleted→rlink;      /* (1) */
                                deleted→rlink→llink= deleted→llink;      /* (2) */
                                free(deleted);
                }
}
```