# Linked List

## Equivalence Relations, Sparse Matrices & Doubly Linked Lists

Joseph Chuang-Chieh Lin (林莊傑)

Department of Computer Science & Engineering,
National Taiwan Ocean University

Fall 2024

# Outline

1. Equivalence Relations
2. Sparse Matrices Revisted
3. Doubly Linked Lists

# Outline

1. Equivalence Relations
2. Sparse Matrices Revisted
3. Doubly Linked Lists

## Equivalence Relation

A relation over a set $S$ is said to be an equivalence relation over $S$ iff it is symmertric, reflexive, and transitive over $S$.

- reflexive: $x \equiv x$ for each $x \in S$.
- symmetric: for $x, y \in S$, if $x \equiv y$, then $y \equiv x$.
- transitive: for $x, y, z$, if $x \equiv y$ and $y \equiv z$, then $x \equiv z$.

## Equivalence Relation

A relation over a set $S$ is said to be an equivalence relation over $S$ iff it is symmertric, reflexive, and transitive over $S$.

- reflexive: $x \equiv x$ for each $x \in S$.
- symmetric: for $x, y \in S$, if $x \equiv y$, then $y \equiv x$.
- transitive: for $x, y, z$, if $x \equiv y$ and $y \equiv z$, then $x \equiv z$.

## Example

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

## Equivalence Relation

A relation over a set $S$ is said to be an equivalence relation over $S$ iff it is symmertric, reflexive, and transitive over $S$.

- reflexive: $x \equiv x$ for each $x \in S$.
- symmetric: for $x, y \in S$, if $x \equiv y$, then $y \equiv x$.
- transitive: for $x, y, z$, if $x \equiv y$ and $y \equiv z$, then $x \equiv z$.

## Example

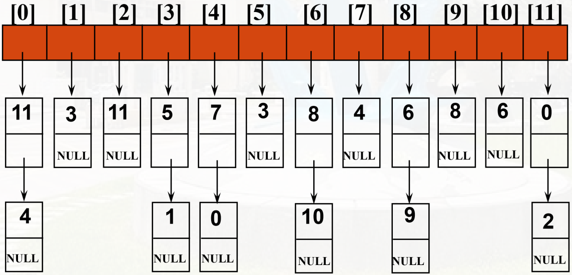
Given $0 \equiv 4$, $3 \equiv 1$, $6 \equiv 10$, $8 \equiv 9$, $7 \equiv 4$, $6 \equiv 8$, $3 \equiv 5$, $2 \equiv 11$, $11 \equiv 0$. We have three equivalent classes:

$$\{0, 2, 4, 7, 11\}, \{1, 3, 5\}, \{6, 8, 9, 10\}.$$

# Lists after Giving Pairs as the Input

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

```c
typedef struct node *nodePointer;
typedef struct node {
    int data;
    nodePointer link;
};
```



# The Equivalence Algorithm (Full Code HERE)

seq[n]: hold the head nodes of the $n$ lists; out[n]: whether the object $i$ has been printed.

```
void equivalence() {
    initialize seq[i] = NULL, out[i] to true for each i
    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++) { // the 2nd phase
        if (out[i]) {
            out[i] = false;
            output this equivalence class;
        }
    }
}
```

# The Equivalence Algorithm (second phase in detail)

```
for (i=0; i<n; i++) { // the 2nd phase
    if (out[i]) {
        printf("\nNew Classes: %d", i);
        out[i] = false;
        x = seq[i]; top = NULL; // initial stack
        while (1) { // find the rest of the class
            while (x) { // process the list
                j = x->data;
                if (out[j]) {
                    printf("%d", j);  out[j] = false;
                    y = x->link; x->link = top; top = x; x = y;
                } else x = x->link;
            }
            if (!top) break;
            x = seq[top->data]; top = top->link; // pop the stack
        }
    }
}
```

# Outline

1 Equivalence Relations

2 Sparse Matrices Revisted

3 Doubly Linked Lists

## Issues for Previous Representation

- When we performed matrix operations such as $+$, $-$, or $*$, the number of **nonzero terms** varied.
- The sequential representation of sparse matrices suffered from the same inadequacies as the similar representation of polynomials.

Solution:

- Linked list representation for sparse matrices.

## Issues for Previous Representation

- When we performed matrix operations such as $+$, $-$, or $*$, the number of **nonzero terms** varied.
- The sequential representation of sparse matrices suffered from the same inadequacies as the similar representation of polynomials.

Solution:

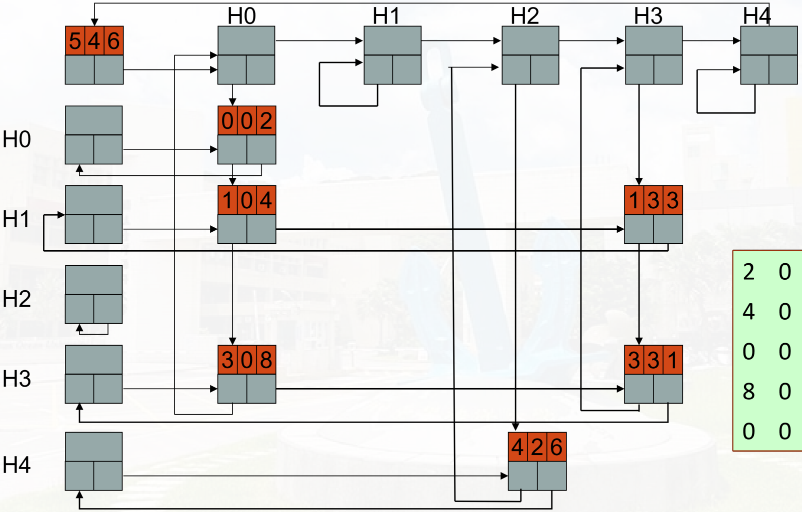
- Linked list representation for sparse matrices.
- Two types of nodes in the representation: header nodes and element nodes.

<table>
<tr><td colspan="2">next</td></tr>
<tr><td>down</td><td>right</td></tr>
</table>

header node

<table>
<tr><td>row</td><td>col</td><td>value</td></tr>
<tr><td colspan="3">down | right</td></tr>
</table>

element node




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

# Sparse Matrix Representation

- We represent each column (row) of a sparse matrix as a circularly linked list with a header node.
- The header node for row *i* is also the header node for column *i*. The number of header nodes is $\max\{\text{numRows}, \text{numCols}\}$.
- Each element node is simultaneously linked into two lists: a row list, and a column list.
- Each head node is belonged to three lists: a row list, a column list, and a header node list.

# Outline

1 Equivalence Relations

2 Sparse Matrices Revisted

3 Doubly Linked Lists

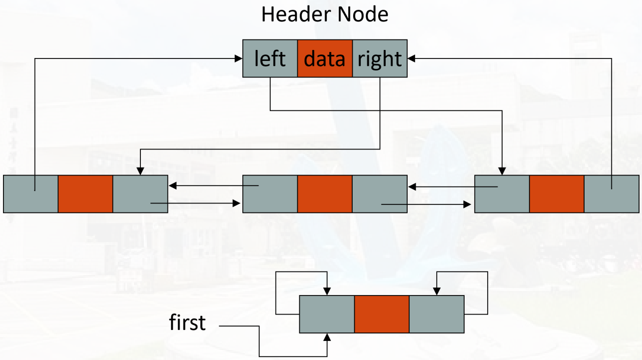
### Issues for Singly Linked Lists

- The only way to find the node that precedes some node $p$ is to start at the beginning of the list.
- Sometimes it is necessary to move in either direction.

Doubly linked lists:

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

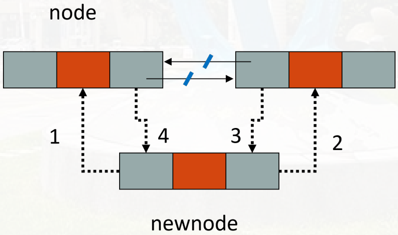
```
ptr = ptr->llink->rlink = ptr->rlink->llink
```




Empty doubly linked circular list with header node

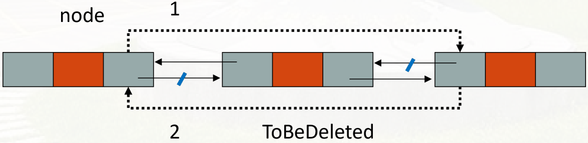
# Insertion into a doubly linked circular List

```
void d_LCL_insert(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
}
```



# Deletion from a doubly linked circular List

```
void d_LCL_delete(nodePointer node, nodePointer ToBeDeleted) {
/* delete from the doubly linked list */
    if (node == ToBeDeleted)
        printf("Deletion of header node not permitted.\n");
    else {
        ToBeDeleted->llink->rlink = ToBeDeleted->rlink;   // 1
        ToBeDeleted->rlink->llink = ToBeDeleted->llink;   // 2
        free(ToBeDeleted);
    }
}
```



# Discussions