

Arrays and Structures

Ikjun Yeom



Contents

Arrays

Dynamically Allocated Arrays

Structures and Unions

Polynomials

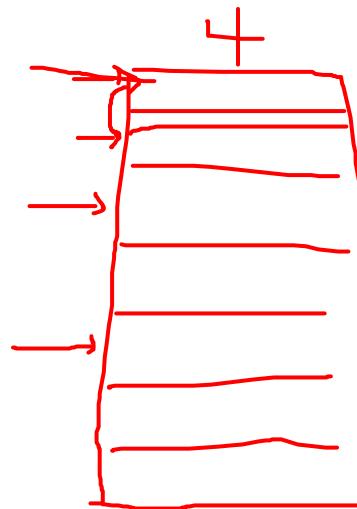
Sparse Matrices

Strings



Definition of Array

- “a set of consecutive memory locations”
- “a collection of data of the same type”
- “a set of pairs, $\langle \text{index}, \text{value} \rangle$, such that each index that is defined has a value associated with it”

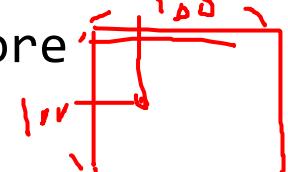


Array as an Abstract Data Type

ADT Array is

objects: A set of pairs $\langle \underline{\text{index}}, \underline{\text{value}} \rangle$ where for each value of index there is a value from the set item .

$\underline{\text{index}}$ is a finite ordered set of one or more dimensions



functions: for all $A \in \text{Array}$, $i \in \text{index}$, $x \in \text{item}$, j ,
size \in integer

$\text{Array Create}(j, \text{List}) ::= \text{return} \dots$

$\text{item Retrieve}(A, i) ::= \text{if } () \text{ return } \dots \text{ else return } \dots$

$\text{Array Store}(A, i, x) ::= \text{if } () \text{ return } \dots \text{ else return } \dots$

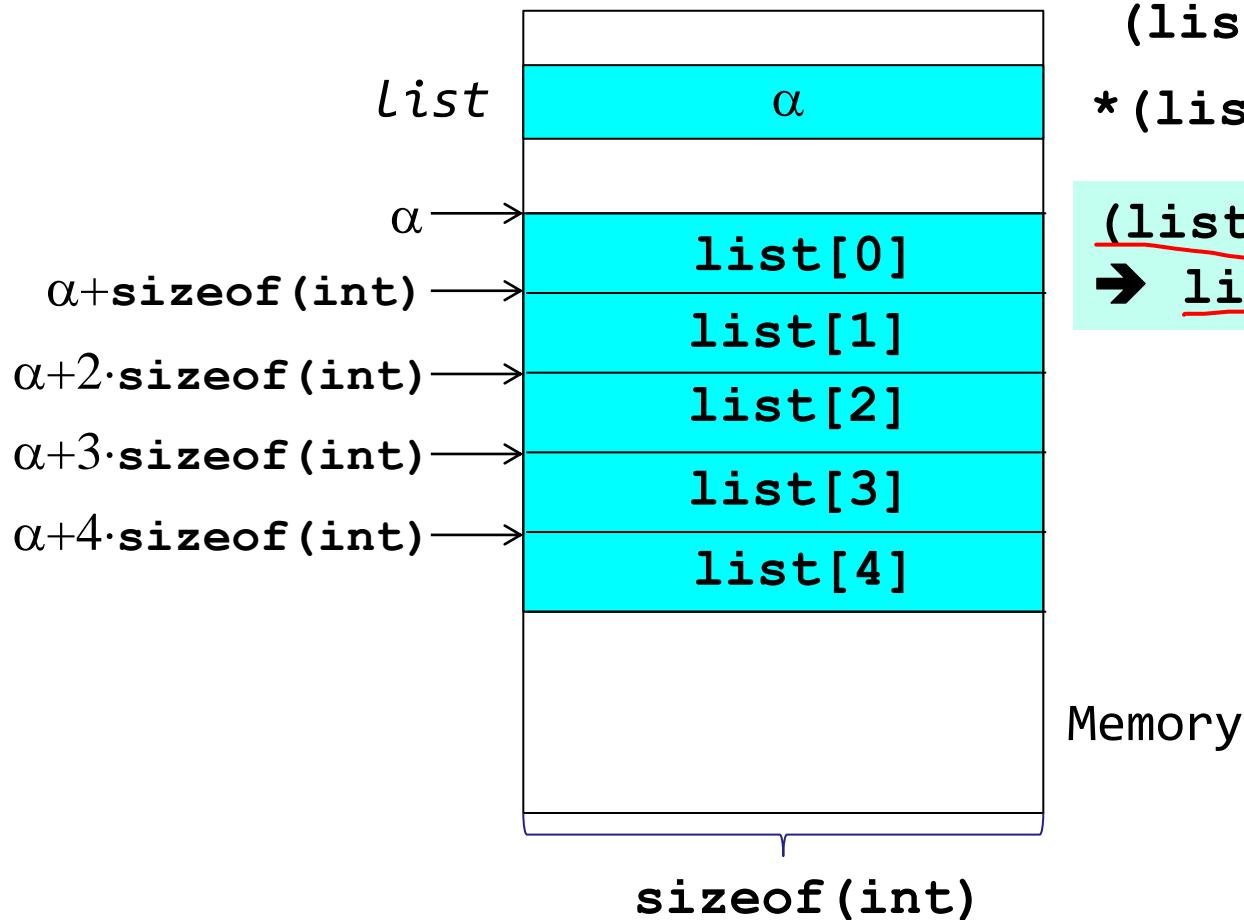
end Array



Array Representation in C

Data structure for one dimensional array

- `int list[5] → int* list;`



`(list+2) vs. &list[2]`

`* (list+2) vs. list[2]`

~~`(list+2)`~~

~~$\rightarrow \text{list} + 2 * \text{sizeof}(\text{int})$~~

Memory



Example Array Program

```
#define MAX_SIZE 100
float sum(float [], int);
float input[MAX_SIZE], answer;
int i;

void main(void)
{
    for (i = 0; i < MAX_SIZE; i++)
        input[i] = i;
    answer = sum(input, MAX_SIZE);
    printf("The sum is: %f\n", answer);
}

float sum(float list[], int n)
{
    int i;
    float tempsum = 0;
    for (i = 0; i < n; i++)
        tempsum += list[i];
    return tempsum;
}
```



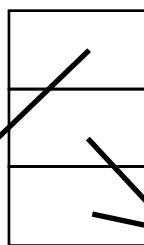
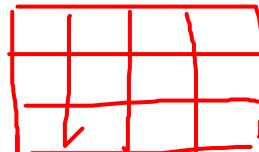
2D Array Representation In C

```
int x[3][4];
```

```
x = x[0]
```

```
x[1]
```

```
x[2]
```



Array-of-arrays representation

- Requires contiguous memory of size 3, 4, 4, and 4 for the 4 1D arrays
- One memory block of size number of rows and number of rows blocks of size number of columns



Dynamically Allocated Arrays – One Dimension (1)

```
#include <math.h>
#include <stdio.h>
#define MAX_SIZE 101

main()
{
    int i, n, list[MAX_SIZE];

    printf("Enter the number of numbers to generate: ");
    scanf("%d", &n);

    for (i=0;i<n;i++)
    {
        list[i]=rand()%1000;
        printf ("%d\n", list[i]);
    }
}
```

What if we change MAX_SIZE to a very large number to avoid run time error??

- We increase the likelihood that program may fail to compile due to lack of memory



Dynamically Allocated Arrays – One Dimension (2)

```
int i, n, *list;

printf("Enter the number of numbers to generate: ");
scanf("%d", &n);

if (n < 1) {
    fprintf(stderr, "Improper value of n \n");
    exit (EXIT_FAILURE);
}

list=malloc(n * sizeof(int));
if (list==NULL) {
    fprintf(stderr, "lack of memory\n");
    exit (EXIT_FAILURE);
}
```

It fails only when $n < 1$ or we do not have sufficient memory to hold the list of numbers



Dynamically Allocated Arrays

- Two Dimension

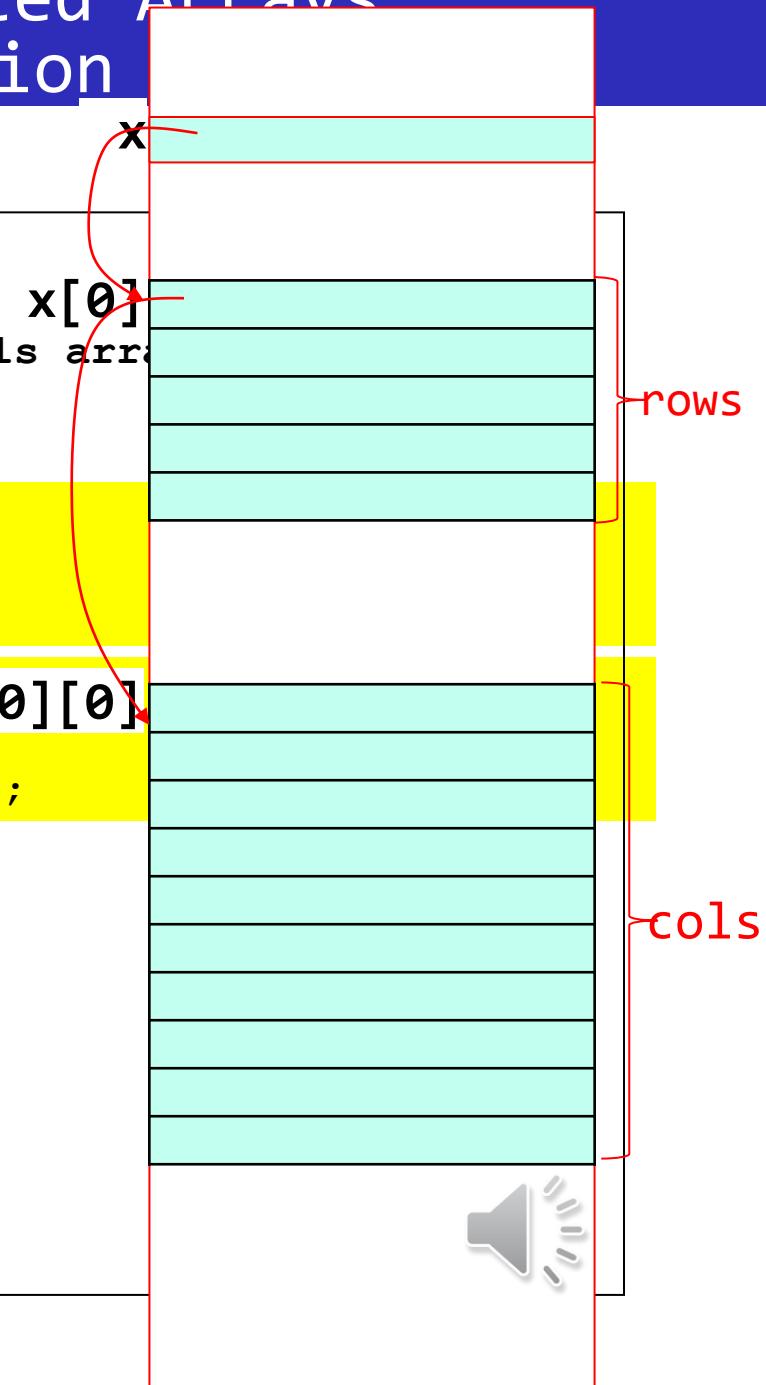
```
int** make2dArray(int rows, int cols)
{
/* create a two dimensional rows × cols array */

    int** x, i;

    /* get memory for row pointers */
    MALLOC(x, rows * sizeof (*int));

    /* get memory for each row */
    for (i = 0; i < rows; i++)
        MALLOC(x[i], cols * sizeof(int));
    return x;
}

void main()
{
    int** myArray;
    myArray=make2dArray(5,10);
    myArray[2][4]=6;
}
```



Quiz 4

Name and student ID

int x[10][10];

Let $\text{length}[i]$ be the desired length of row i of a two dimensional array.

Write a function similar to `make2dArray()` to create a two dimensional array such that row i has $\text{length}[i]$ elements.



Structures (1)

A collection of data items, where each item is identified as to its type and name

```
struct {  
    char name[10];  
    int age;  
    float salary;  
} person;
```

```
strcpy(person.name, "korykang");  
person.age=34;  
person.salary=35000;
```



Structures (2)

To create own structure data type

```
typedef struct humanBeing {  
    char name[10];  
    int age;  
    float salary;  
};
```

or

```
typedef struct {  
    char name[10];  
    int age;  
    float salary;  
} humanBeing;
```

```
humanBeing person1, person2;
```

Assignments

```
person1=person2;
```

Or

```
strcpy(person1.name, person2.name);  
person1.age=person2.age;  
person1.salary=person2.salary;
```

```
if (person1 == person2) {  
    ...}  
        ???  
(Chapter 2 p.61)
```



Structures (3)

To embed a structure within a structure

```
typedef struct {  
    int month;  
    int day;  
    int year;  
} date;
```

```
typedef struct humanBeing{  
    char name[10];  
    int age;  
    float salary;  
    date dob;  
} humanBeing;
```

```
person1.dob.month=12; person1.dob.day=3;  
person1.dob.year=1969;
```



Quiz 5

Name and student ID

Develop a structure to represent each of the following geometric objects: rectangle, triangle, and circle.



Applications of Array

Ordered list

- One dimensional array
- Polynomial: components will be numeric
- String: components can be non-numeric

Matrix

- Standard: multi-dimensional array
- Sparse matrix: one dimensional array
- Components may be numeric



Ordered List (1)

Example

- Days of a week

(Sunday, Monday, Tuesday, Wednesday, Thursday, Friday, Saturday)

- Values in a deck of cards

(Ace, 2, 3, 4, 5, 6, 7, 8, 9, 10, Jack, Queen, King)

- Years Switzerland fought in World War II: ()



Ordered List (2)

Operations

- Finding the length, n , of a list
- Reading the items in a list from left to right (or reversely)
- Retrieving the i -th item from a list, $0 \leq i \leq n$
- Replacing the item in the i -th position of a list,
 $0 \leq i \leq n$
- Inserting a new item in the i -th position of a list,
 $0 \leq i \leq n$
- Deleting an item from the i -th position of a list,
 $0 \leq i \leq n$



Polynomial Abstract Data Type (1)

Example of polynomials

$$\begin{aligned} A(x) &= 3x^6 + 2x^5 + 4, \quad B(x) = x^4 + 10x^3 + 3x^2 + 1 \\ \rightarrow A(x) + B(x) &= 3x^6 + 2x^5 + x^4 + 10x^3 + 3x^2 + 5 \end{aligned}$$

Manipulation of symbolic polynomials

$$A(x) = \sum_{i=0}^n a_i x^i \quad B(x) = \sum_{j=0}^m b_j x^j$$

$$A(x) + B(x) = \sum_{i=0}^{\max(n,m)} (a_i + b_i) \cdot x^i$$

$$A(x) \cdot B(x) = \sum_{i=0}^n (a_i x^i \sum_{j=0}^m (b_j \cdot x^j))$$



Polynomial Abstract Data Type (2)

ADT *Polynomial* is

objects : $p(x) = a_1x^{e_1} + \dots + a_nx^{e_n}$: a set of ordered pairs of $\langle e_i, a_i \rangle$, where a_i in *Coefficients* and e_i in *Exponents*, e_i are integers ≥ 0 .

functions : for all $\text{poly}, \text{poly1}, \text{poly2} \in \text{Polynomial}$,
 $\text{coef} \in \text{Coefficients}$, $\text{expon} \in \text{Exponents}$

Polynomial Zero() ::= ...

Boolean IsZero(*poly*) ::= ...

Coefficient Coef(*poly*, *expon*) ::= ...

Exponent LeadExp(*poly*) ::= ...

Polynomial Attach(*poly*, *coef*, *expon*) ::= ...

Polynomial Remove(*poly*, *expon*) ::= ...

Polynomial SingleMult(*poly*, *coef*, *expon*) ::= ...

Polynomial Add(*poly1*, *poly2*) ::= ...

Polynomial Mult(*poly1*, *poly2*) ::= ...

end *Polynomial*



Polynomial Abstract Data Type (3)

```
/* d = a + b, where a, b, and d are polynomials */
d = Zero();
while (! IsZero(a) && ! IsZero(b)) do {
    switch COMPARE(LeadExp(a), LeadExp(b)) {
        case -1: /* if LeadExp(a) < LeadExp(b) */
            d = Attach(d, Coef(b, LeadExp(b)), LeadExp(b));
            b = Remove(b, LeadExp(b));
            break;
        case 0: /* if LeadExp(a) == LeadExp(b) */
            sum = Coef(a, LeadExp(a)) + Coef(b, LeadExp(b));
            if (sum) {
                d = Attach(d, sum, LeadExp(a));
                a = Remove(a, LeadExp(a));
                b = Remove(b, LeadExp(b));
            }
            break;
        case 1: /* if LeadExp(a) > LeadExp(b) */
            d = Attach(d, Coef(a, LeadExp(a)), LeadExp(a));
            a = Remove(a, LeadExp(a));
    }
}
```

Insert any remaining terms of a or b into d

$$A(x) = 3x^6 + 2x^5 + 4, \quad B(x) = x^4 + 10x^3 + 3x^2 + 5$$



Polynomial Representation (1)

```
#define MAX_DEGREE 101
```

```
typedef struct {
```

```
    int degree;
```

```
    float coef[MAX_DEGREE];
```

```
} polynomial;
```

$$n < \text{MAX_DEGREE} \quad A(x) = \sum_{i=0}^n a_i x^i$$

```
polynomial a;
```

```
a.degree = n,    a.coef[i] = a_{n-i}, 0 \leq i \leq n
```

$$A(x) = 3x^6 + 2x^5 + 41$$

How about $2x^{1000} + 1$?

| | | | | | | | | |
|---|--|---|---|---|---|---|---|----|
| a | 6 | 3 | 2 | 0 | 0 | 0 | 0 | 41 |
| |  | | | | | | | |



Polynomial Representation (2)

```
#define MAX_TERMS 100
typedef struct {
    float coef;
    int expon;
} polynomial;
polynomial terms[MAX_TERMS];
int starta, finisha, startb, finishb, avail;
starta=0; finisha=1; startb=2; finishb=5; avail=6;
```

```
#define MAX_DEGREE 101
```

$$A(x) = 2x^{1000} + 1,$$

$$B(x) = x^4 + 10x^3 + 3x^2 + 1$$

Diagram illustrating the polynomial representation in memory:

| | finisha | | finishb | | avail | | avail | | | | | |
|---------|---------|-----|---------|-----|-------|-----|-------|-----|-----|-----|------|------|
| terms[] | ↓ | ↓ | ↓ | ↓ | ↓ | ↓ | ↓ | | | | | |
| coef | 2 | 1 | 1 | 10 | 3 | 1 | 2 | 1 | 10 | 3 | 2 | |
| expon | 1000 | 0 | 4 | 3 | 2 | 0 | 1000 | 4 | 3 | 2 | 0 | |
| | [0] | [1] | [2] | [3] | [4] | [5] | [6] | [7] | [8] | [9] | [10] | [11] |



Quiz 6

Name and student ID

Implement Remove() and Attach() in slide 21 using polynomial representation in slide 22.



Polynomial Addition (1)

```

void padd (int starta, int finisha, int startb, int finishb,
           int* startd, int* finishd)
{
    float coefficient;
    *startd = avail;
    while (starta <= finisha && startb <= finishb) {
        switch (COMPARE(terms[starta].expon, terms[startb].expon)) {
            case -1: /* a.expon < b.expon */
                attach(terms[startb].coef, terms[startb].expon);
                startb++;
                break;
            case 0: /* equal exponents */
                coefficient = terms[starta].coef + terms[startb].coef;
                if (coefficient)
                    attach(coefficient, terms[starta].expon);
                starta++;
                startb++;
                break;
            case 1: /* a.expon > b.expon */
                attach(terms[starta].coef, terms[starta].expon);
                starta++;
        } /* switch */
    } /* while */
}

```

b

D

5

| terms[] | | finisha | | startb | | finishb | | avail | | | | |
|---------|------|---------|-----|--------|-----|---------|-----|-------|-----|-----|------|------|
| coef | 2 | 1 | 1 | 10 | 3 | 2 | 1 | 0 | | | | |
| expon | 1000 | 0 | 4 | 3 | 2 | 2 | 0 | 0 | | | | |
| | [0] | [1] | [2] | [3] | [4] | [5] | [6] | [7] | [8] | [9] | [10] | [11] |

$$A(x) = 2x^{1000} + 1,$$

$$B(x) = x^4 + 10x^3 + 3x^2 + 1$$



Polynomial Addition (2)

```
/* add in remaining terms of A(x) */
for(; starta <= finisha; starta++) {
    attach(terms[starta].coef, terms[starta].expon);
}
/* add in remaining terms of B(x) */
for(; startb <= finishb; startb++) {
    attach(terms[startb].coef, terms[startb].expon);
}
*finished = avail-1;
} /* end of padd */
```

```
void attach(float coefficient, int exponent)
{
    /* add a new term to the polynomial */
    if (avail >= MAX_TERMS) {
        fprintf(stderr, "too many terms!!!\n");
        exit(1);
    }
    terms[avail].coef = coefficient;
    terms[avail++].expon = exponent;
}
```

$$A(x) = 2x^{10} + 2x^3 + x + 1,$$
$$B(x) = x^4 + 10x^3 + 3x^2$$

$$A(x) = 2x^5 + 2x^4,$$
$$B(x) = x^4 + 5x^3 + x^2 + 4$$



Matrix

- m rows and n columns
- Standard representation : two-dimensional array as
a [MAX_ROWS] [MAX_COLS]

$$\begin{bmatrix} -27 & 3 & 4 \\ 6 & 82 & -2 \\ 109 & -64 & 11 \\ 12 & 8 & 9 \\ 48 & 27 & 47 \end{bmatrix}$$

a [0] [0]=-27;
a [1] [2]=-2;
a [3] [1]=8;



Sparse Matrix

- Sparse matrix has many zero entries

$$\frac{\text{no. of non-zero elements}}{\text{no. of total elements}} \ll \text{small}$$

- two-dimensional array wastes space
- Represent using the triple <row, col, value>

$\frac{15}{15}$

| | col0 | col1 | col2 |
|------|------|------|------|
| row0 | -27 | 3 | 4 |
| row1 | 6 | 82 | -2 |
| row2 | 109 | -64 | 11 |
| row3 | 12 | 8 | 9 |
| row4 | 48 | 27 | 47 |

$\frac{8}{36}$

| | col0 | col1 | col2 | col3 | col4 | col5 |
|------|------|------|------|------|------|------|
| row0 | 15 | 0 | 0 | 22 | 0 | -15 |
| row1 | 0 | 11 | 3 | 0 | 0 | 0 |
| row2 | 0 | 0 | 0 | -6 | 0 | 0 |
| row3 | 0 | 0 | 0 | 0 | 0 | 0 |
| row4 | 91 | 0 | 0 | 0 | 0 | 0 |
| row5 | 0 | 0 | 28 | 0 | 0 | 0 |



Sparse Matrix Abstract Data Type

ADT *SparseMatrix* is

objects: a set of triples, $\langle \text{row}, \text{column}, \text{value} \rangle$, where row and column are integers and from a unique combination, and value comes from the set item functions:

for all $a, b \in SparseMatrix$, $x \in item$, $i, j, max_col, max_row \in index$

SparseMatrix Create($maxRow, maxCol$) ::=

SparseMatrix Transpose(a) ::=

SparseMatrix Add(a, b) ::=

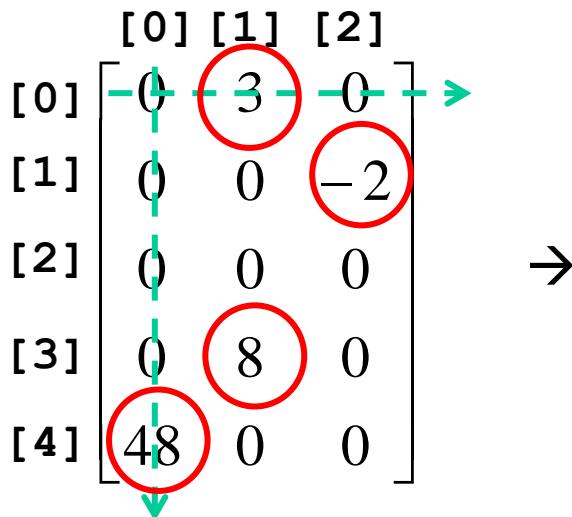
SparseMatrix Mutiply(a, b) ::=

end *SparseMatrix*



Sparse Matrix Representation

```
#define MAX_TERMS 101
typedef struct {
    int row;
    int col;
    int value;
} term;
term a[MAX_TERMS];
```



Number of elements

Number of columns

Number of rows

| | row | col | value |
|------|-----|-----|-------|
| a[0] | 5 | 3 | 4 |
| a[1] | 0 | 1 | 3 |
| a[2] | 1 | 2 | -2 |
| a[3] | 3 | 1 | 8 |
| a[4] | 4 | 0 | 48 |



Approximate Memory Requirements

500 x 500 matrix with 1994 nonzero elements,
4 bytes per element

2D array: $500 \times 500 \times 4 = 1\text{M bytes}$

1D array of triples: $3 \times 1994 \times 4 \approx 23\text{K bytes}$



Quiz 7

Name and student ID

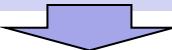
We have a 500×500 matrix. Find the smallest number of nonzero elements such that the memory space of 2-D array representation becomes less than that of the sparse matrix representation.



Transposing a Matrix (1)

Transpose operation ($A(i,j) \rightarrow A(j,i)$)

| | | | | | |
|----|----|----|----|---|-----|
| 15 | 0 | 0 | 22 | 0 | -15 |
| 0 | 11 | 3 | 0 | 0 | 0 |
| 0 | 0 | 0 | -6 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 |
| 91 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 28 | 0 | 0 | 0 |



| | | | | | |
|-----|----|----|---|----|----|
| 15 | 0 | 0 | 0 | 91 | 0 |
| 0 | 11 | 0 | 0 | 0 | 0 |
| 0 | 3 | 0 | 0 | 0 | 28 |
| 22 | 0 | -6 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 |
| -15 | 0 | 0 | 0 | 0 | 0 |

| | row | col | value |
|------|-----|-----|-------|
| a[0] | 6 | 6 | 8 |
| a[1] | 0 | 0 | 15 |
| a[2] | 0 | 3 | 22 |
| a[3] | 0 | 5 | -15 |
| a[4] | 1 | 1 | 11 |
| a[5] | 1 | 2 | 3 |
| a[6] | 2 | 3 | -6 |
| a[7] | 4 | 0 | 91 |
| a[8] | 5 | 2 | 28 |

| | row | col | value |
|------|-----|-----|-------|
| b[0] | 6 | 6 | 8 |
| b[1] | 0 | 0 | 15 |
| b[2] | 0 | 4 | 91 |
| b[3] | 1 | 1 | 11 |
| b[4] | 2 | 1 | 3 |
| b[5] | 2 | 5 | 28 |
| b[6] | 3 | 0 | 22 |
| b[7] | 3 | 2 | -6 |
| b[8] | 5 | 0 | -15 |

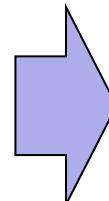


Transposing a Matrix (2)

for each row i

take element $\langle i, j, \text{value} \rangle$ and store it as element
 $\langle j, i, \text{value} \rangle$

| | row | col | value |
|------|-----|-----|-------|
| a[0] | 6 | 6 | 8 |
| a[1] | 0 | 0 | 15 |
| a[2] | 0 | 3 | 22 |
| a[3] | 0 | 5 | -15 |
| a[4] | 1 | 1 | 11 |
| a[5] | 1 | 2 | 3 |
| a[6] | 2 | 3 | -6 |
| a[7] | 4 | 0 | 91 |
| a[8] | 5 | 2 | 28 |



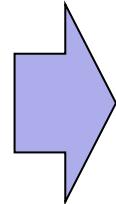
| | row | col | value |
|------|-----|-----|-------|
| a[0] | | | |
| a[1] | | | |
| a[2] | | | |
| a[3] | | | |
| a[4] | | | |
| a[5] | | | |
| a[6] | | | |
| a[7] | | | |
| a[8] | | | |

The table shows the transpose of the original matrix. The element at index a[3] (row 0, col 5) is highlighted in yellow and moved to index a[3] (row 5, col 0). A speaker icon indicates that the transpose operation has been completed.

Transposing a Matrix (3)

for all elements in column j
place element $\langle i, j, \text{value} \rangle$ in element $\langle j, i, \text{value} \rangle$

| | row | col | value |
|------|-----|-----|-------|
| a[0] | 6 | 6 | 8 |
| a[1] | 0 | 0 | 15 |
| a[2] | 0 | 3 | 22 |
| a[3] | 0 | 5 | -15 |
| a[4] | 1 | 1 | 11 |
| a[5] | 1 | 2 | 3 |
| a[6] | 2 | 3 | -6 |
| a[7] | 4 | 0 | 91 |
| a[8] | 5 | 2 | 28 |



| | row | col | value |
|------|-----|-----|-------|
| a[0] | 6 | 6 | 8 |
| a[1] | 0 | 0 | 15 |
| a[2] | 0 | 4 | 91 |
| a[3] | 1 | 1 | 11 |
| a[4] | 2 | 1 | 3 |
| a[5] | 2 | 5 | 28 |
| a[6] | 3 | 0 | 22 |
| a[7] | 3 | 2 | -6 |
| a[8] | 5 | 0 | -15 |



Transposing a Matrix (3)

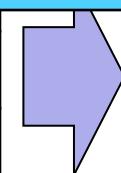
```
void transpose(term a[], term b[])
{
    /* b is set to the transpose of a*/
    int n, i, j, currentb;
    n = a[0].value;      /* total number of elements */
    b[0].row = a[0].col; /* rows in b = columns in a */
    b[0].col = a[0].row; /* columns in b = rows in a */
    b[0].value = n;
    if (n > 0) { /* non-zero matrix*/
        currentb = 1;
        for (i=0; i<a[0].col; i++){ /*transpose by the column*/
            for (j = 1; j <= n; j++){
                if (a[j].col == i){
                    b[currentb].row = a[j].col;
                    b[currentb].col = a[j].row;
                    b[currentb].value = a[j].value;
                    currentb++;
                } /* if a[j].col */
            } /* for j */
        } /* for i */
    } /* if n */
}
```

$\rightarrow O(\text{columns} \cdot \text{elements})$



Fast Matrix Transpose (1)

| | row | col | value |
|------|-----|-----|-------|
| a[0] | 6 | 6 | 8 |
| a[1] | 0 | 0 | 15 |
| a[2] | 0 | 3 | 22 |
| a[3] | 0 | 5 | 3 |
| a[4] | 1 | 1 | 11 |
| a[5] | 1 | 2 | 3 |
| a[6] | 2 | 3 | -6 |
| a[7] | 4 | 0 | 91 |
| a[8] | 5 | 2 | 28 |



| | row | col | value |
|------|-----|-----|-------|
| a[0] | 6 | 6 | 8 |
| a[1] | 0 | 0 | 15 |
| a[2] | | | |
| a[3] | | | |
| a[4] | | | |
| a[5] | | | |
| a[6] | 3 | 0 | 22 |
| a[7] | | | |
| a[8] | | | |

2 elements in row 0
1 element in row 1
2 elements in row 2



Fast Matrix Transpose (2)

Step 1: #non-zero in each row of transpose = #non-zero
in each column of original matrix

Step2: Starting position of each row of transpose
= sum of size of preceding rows of transpose

Step 3: Move elements, left to right, from original
matrix to transpose matrix



Original matrix

| | row | col | value |
|------|-----|-----|-------|
| a[0] | 6 | 6 | 8 |
| a[1] | 0 | 0 | 15 |
| a[2] | 0 | 3 | 22 |
| a[3] | 0 | 5 | -15 |
| a[4] | 1 | 1 | 11 |
| a[5] | 1 | 2 | 3 |
| a[6] | 2 | 3 | -6 |
| a[7] | 4 | 0 | 91 |
| a[8] | 5 | 2 | 28 |

Number of non-zero elements in each row of transpose matrix

rowTerms

| | |
|-----|---|
| [0] | 2 |
| [1] | 1 |
| [2] | 2 |
| [3] | 2 |
| [4] | 0 |
| [5] | 1 |

Starting position of each row of transpose matrix

startingPos

| | |
|-----|---|
| [0] | 1 |
| [1] | 3 |
| [2] | 4 |
| [3] | 6 |
| [4] | 8 |
| [5] | 8 |

```
startingPos[0] = 1;
for(i = 1; i < num_cols; i++) {
    startingPos[i]=
        startingPos[i-1]+
        rowTerms[i-1];
}
```

a[2]: <0,3,22> → b[startingPos[3]]: <3,0,22>



```

for(i = 1; i <= numTerms; i++) {
    j = startingPos[a[i].col];
    b[j].row=a[i].col; b[j].col=a[i].row; b[j].value=a[i].value;
    startingPos[a[i].col]++;
}

```

| | [0] | [1] | [2] | [3] | [4] | [5] |
|-------------|-----|-----|-----|-----|-----|-----|
| startingPos | 3 | 4 | 6 | 8 | 8 | 9 |

Original matrix

| | row | col | value |
|------|-----|-----|-------|
| a[0] | 6 | 6 | 8 |
| a[1] | 0 | 0 | 15 |
| a[2] | 0 | 3 | 22 |
| a[3] | 0 | 5 | -15 |
| a[4] | 1 | 1 | 11 |
| a[5] | 1 | 2 | 3 |
| a[6] | 2 | 3 | -6 |
| a[7] | 4 | 0 | 91 |
| a[8] | 5 | 2 | 28 |

Transpose matrix

| | row | col | value |
|------|-----|-----|-------|
| a[0] | 6 | 6 | 8 |
| a[1] | 0 | 0 | 15 |
| a[2] | 0 | 4 | 91 |
| a[3] | 1 | 1 | 11 |
| a[4] | 2 | 1 | 3 |
| a[5] | 2 | 5 | 28 |
| a[6] | 3 | 0 | 22 |
| a[7] | 3 | 2 | -6 |
| a[8] | 5 | 0 | -15 |

→ 0 (columns + elements)

Fast Matrix Transpose (2)

```
void fastTranspose(term a[], term b[])
{
    int rowTerms[MAX_COL], startingPos[MAX_COL];
    int i,j, numCols = a[0].col, numTerms = a[0].value;
    b[0].row = numCols; b[0].col = a[0].row;
    b[0].value = numTerms;
    if (numTerms > 0) /* non-zero matrix*/
        for(i = 0; i < numCols; i++) { rowTerms[i] = 0; }
        for(i=1; i<= numTerms; i++) {rowTerms[a[i].col]++; }
        startingPos[0] = 1;
        for(i = 1; i < numCols; i++){
            startingPos[i]=startingPos[i-1]+rowTerms[i-1];
        }
        for(i = 1; i <= numTerms; i++) {
            j = startingPos[a[i].col];
            b[j].row = a[i].col; b[j].col = a[i].row;
            b[j].value = a[i].value;
            startingPos[a[i].col]++;
        }
    }
}
```

→ O (columns + elements)



Quiz 8

Name and student ID

Rewrite the following codes without `rowTerms[]` to save memory.

Hint: Reuse `startingPos[]`.

```
for(i=1; i<= numTerms; i++) {rowTerms [a[i].col]++;}  
startingPos[0] = 1;  
for(i = 1; i < numCols; i++){  
    startingPos[i]=startingPos[i-1]+rowTerms[i-1];}
```



String Abstract Data Type (1)

ADT *String* is

objects: a finite set of zero or more characters

functions:

for all $s, t \in String, i, j, m \in$ non-negative integers

String Null(m) ::= ...

Integer Compare(s, t) ::= ...

Boolean IsNull(s) ::= ...

Integer Length(s) ::= ...

String Concat(s, t) ::= ...

String Substr(s, i, j) ::= ...

end *String*

String representation in C

```
#define MAX_SIZE 100  
char s[MAX_SIZE] = {"dog"};
```

| | | | |
|---|---|---|----|
| d | o | g | \0 |
|---|---|---|----|



String Abstract Data Type (2)

C String functions

- `char *strcat(char *dest, char *src)`
- `char *strncat(char *dest, char *src, int n)`
- `int strcmp(char *str1, char *str2)`
- `int strncmp(char *str1, char *str2, int n)`
- `char *strcpy(char *dest, char *src)` *a = b*
- `char *strncpy(char *dest, char *src, int n)`
- `size_t strlen(char *s)`
- `char *strchr(char *s, int c)` *'c'* 
- `char *strtok(char *s, char *delimiters)`
- `char *strstr(char *s, char *pat)`
- `size_t strspn(char *s, char *spanset)`
- `size_t strcspn(char *s, char *spanset)`
- `char *strpbrk(char *s, char *spanset)`



String Insertion (1)

```
void stringins (char *s, char *t, int i)
{
    char string[MAX_SIZE], *temp=string;
    memset(string, 0, sizeof(string));

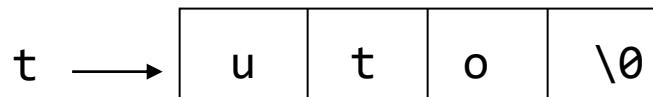
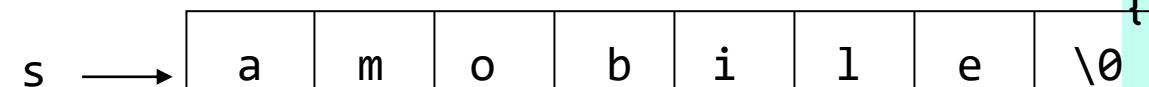
    if (i<0 || i>strlen(s)) {
        fprintf (stderr, "position is out of bounds\n");
        exit(1);
    }
    if (!strlen(s)) strcpy(s,t);
    else if (strlen(t)){
        strncpy(temp, s, i);
        strcat(temp, t);
        strcat(temp, s+i);
        strcpy(s, temp);
    }
}
```



String Insertion (2)

```
void main()
{
    char s[MAX_SIZE] = "amobile";
    char t[MAX_SIZE] = "uto";
    stringins(s, t, 1);
    printf("%s\n", s);

    memset(string, 0, sizeof(string));
}
```



strncpy(temp, s, i); char string[MAX_SIZE], *temp=string;



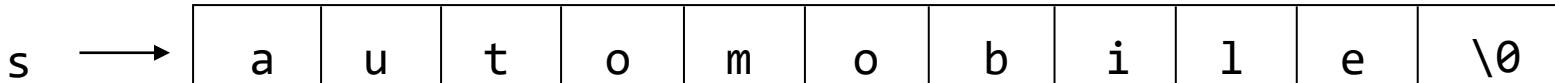
strcat(temp, t);



strcat(temp, s+i);



strcpy(s, temp);

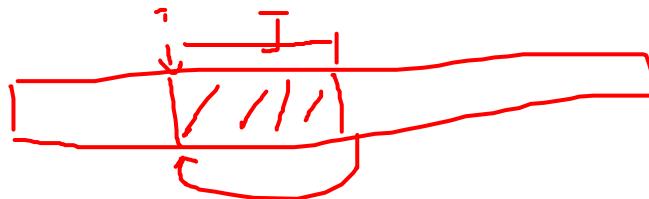


Quiz 9

Name and student ID

Write the string remove function to remove j characters beginning from i in string s.

```
void stringremove (char *s, int i, int j) {
```

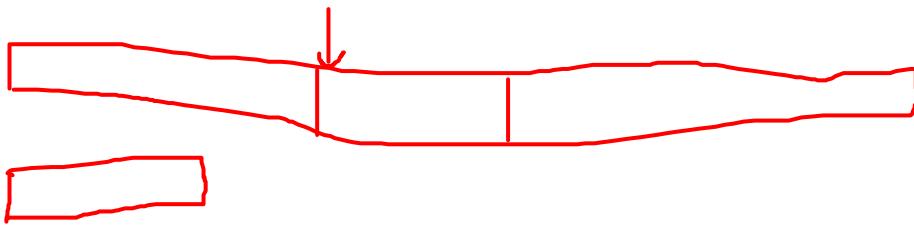


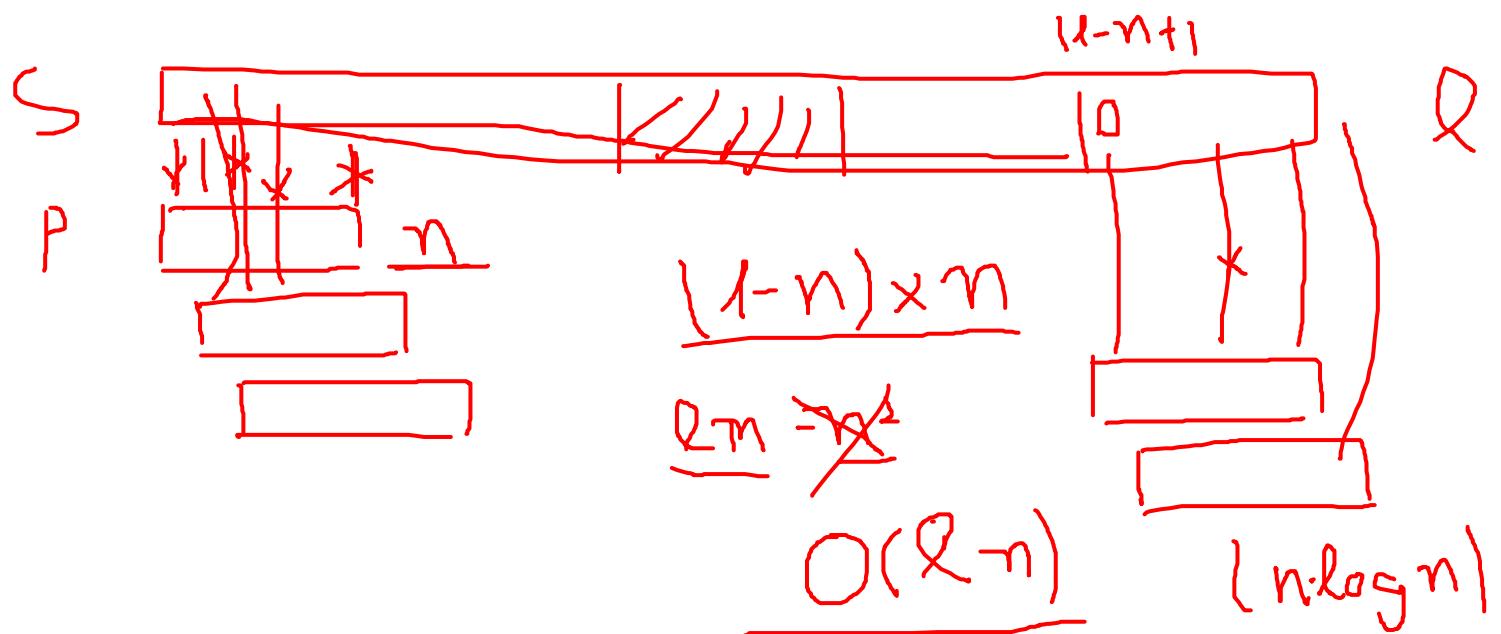
}



Pattern Matching by C-library

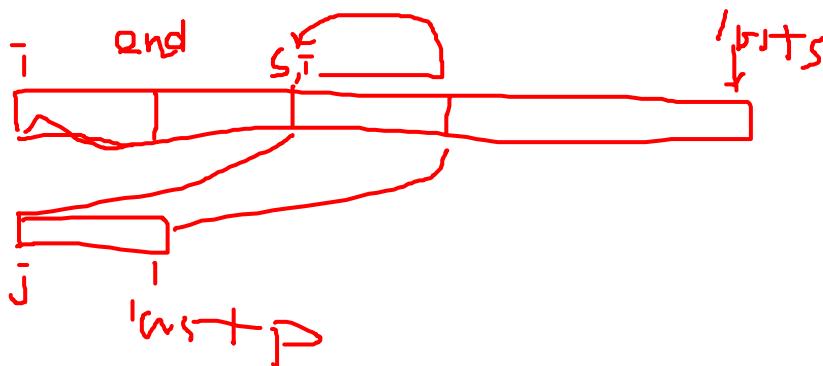
```
#include <string.h>
...
if (t=strstr(string, pat))
    printf("The string from strstr is : %\n", t);
else
    printf("The pattern was not found with strstr\n");
...
(Chapter 3, p.91)
```





Pattern Matching by Checking End Indices First

```
int nfind (char *string, char *pat)
{
    int i, j, start = 0
    int lasts = strlen(string)-1, lastp = strlen(pat)-1;
    int endmatch = lastp;
    for (i = 0; endmatch<=lasts; endmatch++, start++) {
        if (string[endmatch] == pat[lastp]){
            for (j=0, i=start;
                 j<lastp && string[i]==pat[j]; i++, j++);
            if (j == lastp)
                return start;
        }
    }
    return -1;      → O(length of string · length of pattern))
}
```



Quiz 10

Name and student ID

Write a function strnchar() that takes a string and a character as input, and returns the number of occurrences of the character in the string.

```
int strnchar(char *s, char p) {
```

```
}
```



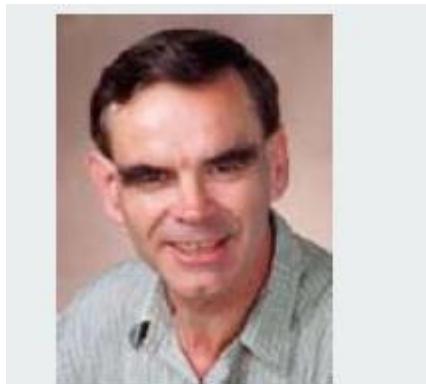
Pattern Matching (Knuth, Morris, Pratt)



Don Knuth



Jim Morris

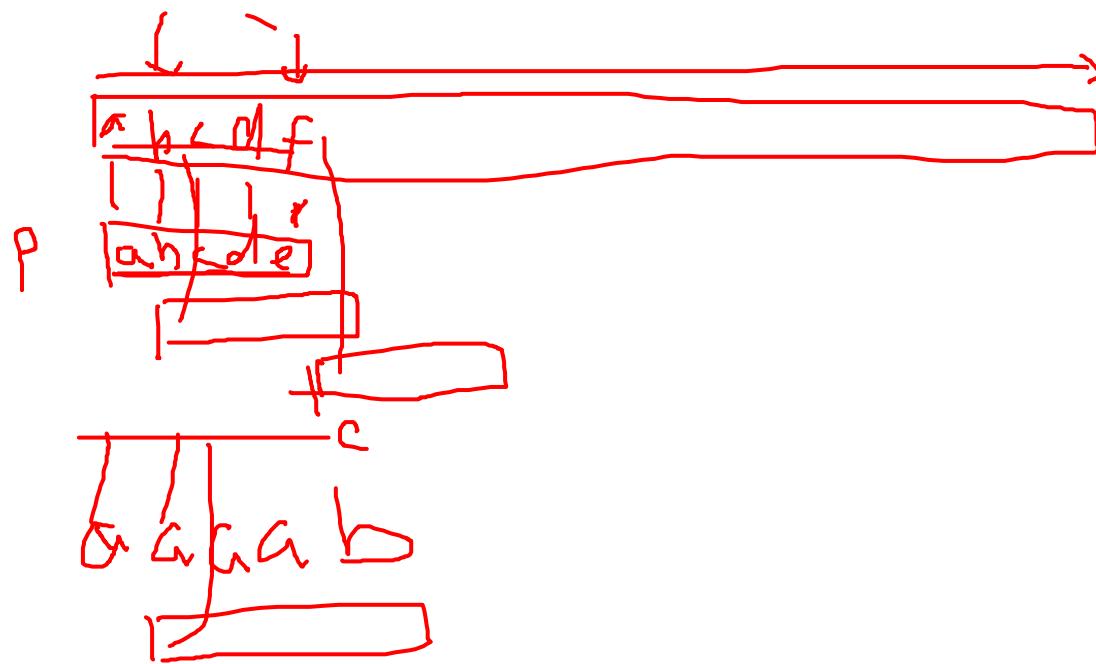
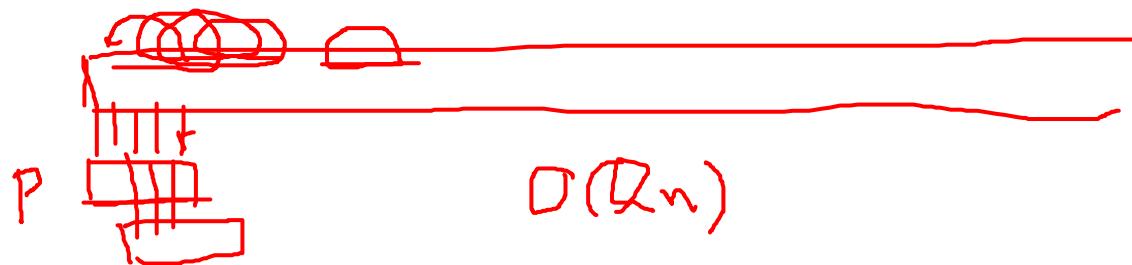


Vaughan Pratt

Donald E. Knuth, James H. Morris, JR, and Vaughan R. Pratt, “Fast Pattern Matching in Strings,” SIAM Journal of Computing, vol.6, no.2, 1977.
(Chapter 2, p.98)

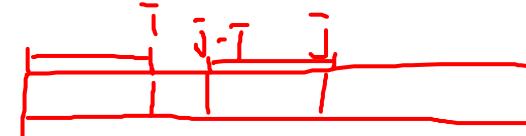


KMP Pattern Matching Idea



Failure Function

$$f(j) = \begin{cases} \text{largest } i < j, \text{ such that } p_0 p_1 \dots p_i = p_{j-i} p_{j-i+1} \dots p_j \\ \quad \text{if such an } i \geq 0 \text{ exists} \\ -1, \text{ otherwise} \end{cases}$$



Failure function example 1)

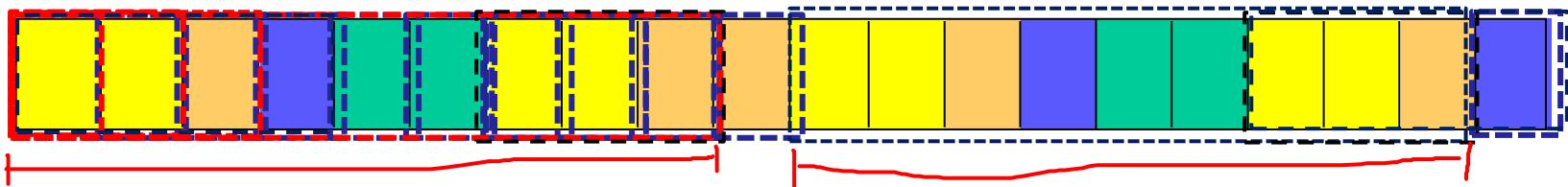
| j | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|--------|----|----|----|---|---|---|---|----|
| pat | a | b | c | | a | b | c | a |
| $f(j)$ | -1 | -1 | -1 | 0 | 1 | 2 | 3 | -1 |



Failure Function Example

Pattern

[0] [1] [2] [3] [4] [5] [6] [7] [8] [9] [10] [11] [12][13][14][15][16][17][18][19]



Failure function $f(j)$

[0] [1] [2] [3] [4] [5] [6] [7] [8] [9] [10] [11] [12][13][14][15][16][17][18][19]

| | | | | | | | | | | | | | | | | | | | |
|----|---|----|----|----|----|---|---|---|----|---|---|---|---|---|---|---|---|---|---|
| -1 | 0 | -1 | -1 | -1 | -1 | 0 | 1 | 2 | -1 | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 3 |
|----|---|----|----|----|----|---|---|---|----|---|---|---|---|---|---|---|---|---|---|

$$f(7)=f(6)+1$$

$$f(8)=f(7)+1$$

$$f(19)=f(f(18))+1$$

(Chapter 2, Program 2.15, p97)



Quiz 11

Name and student ID

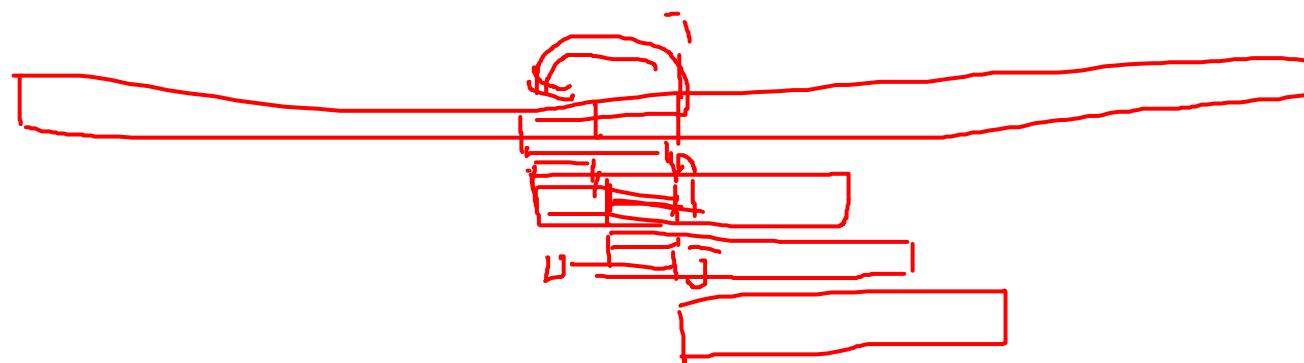
Compute the failure function for each of the following patterns.

- (a) aaaab
- (b) ababaa
- (c) abaabaab

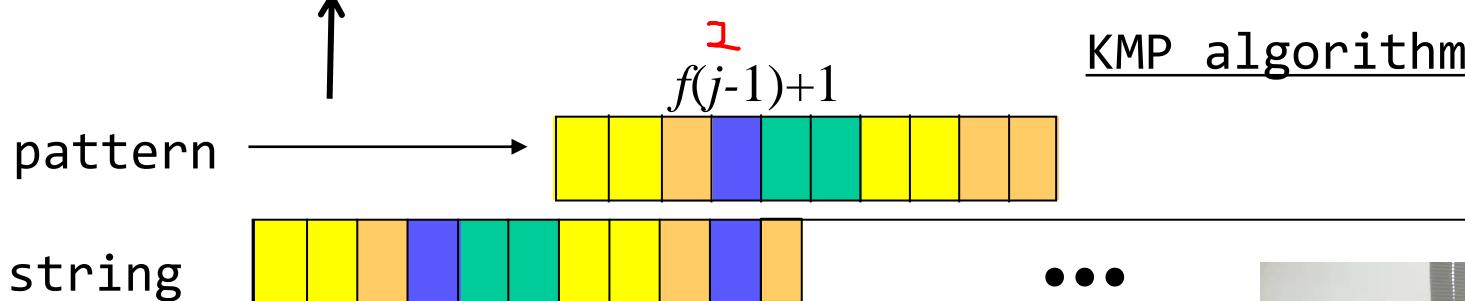
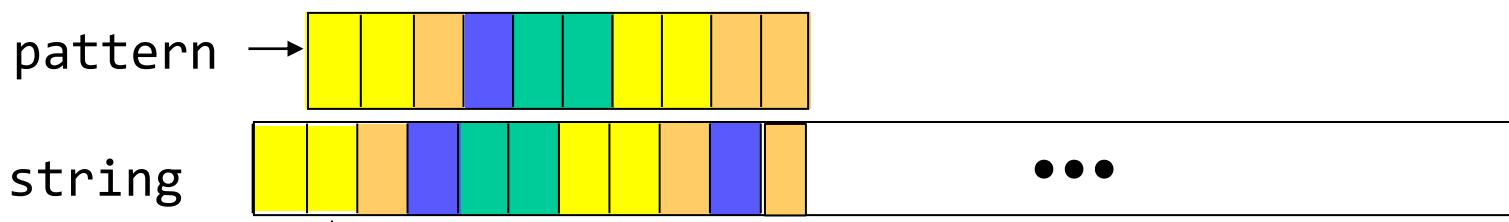
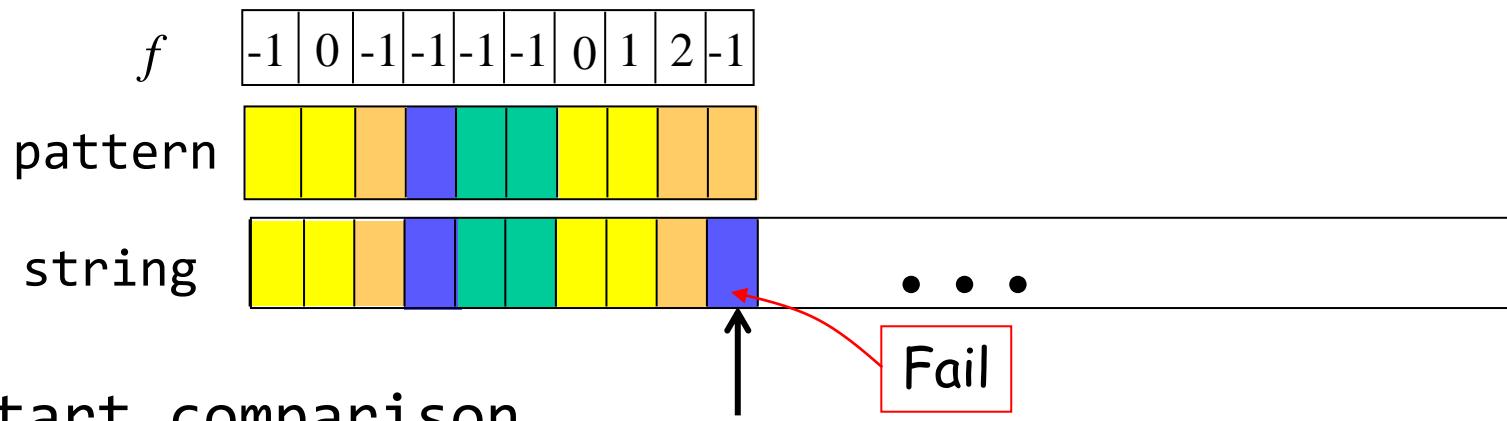


KMP Pattern Matching Rules

- If a partial match is found such that $s_{i-j} \dots s_{i-1} = p_0 p_1 \dots p_{j-1}$ and $s_i \neq p_j$,
then matching may be resumed by comparing
 s_i and $p_{f(j-1)+1}$, if $j \neq 0$
-1
- If $j=0$, we may continue comparing s_{i+1} and p_0



Pattern Matching Comparison



Pattern Matching Algorithm (1)

```
int pmatch (char* string, char* pat)
{
    int i=0; int j=0;
    int lens=strlen(string); int lenp=strlen(pat);
    while (i < lens && j < lenp) {
        if (string[i] == pat[j]) { i++; j++; }
        else if (j == 0) i++;
        else j = failure[j-1]+1;
    }
    return (j == lenp ? i-lenp : -1);
}
```

→ O(length of string)

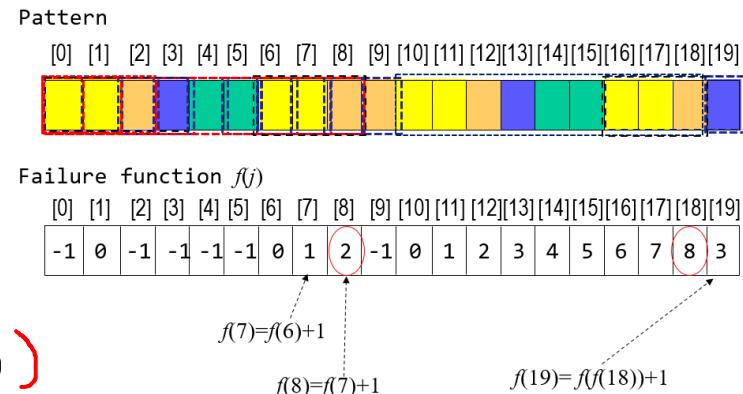


Pattern Matching Algorithm (2)

```
void fail (char *pat)
{
    int n = strlen(pat);
    failure[0] = -1;           j= 19
    for (j = 1; j < n; j++) {
        i = failure[j-1];     8
        while (pat[j] != pat[i+1] && i>=0) i = failure[i]; 2
        if (pat[j] == pat[i+1]) failure[j] = i+1;
        else failure[j] = -1;
    }
}
```

→ O(length of pattern)

The complexity of KMP algorithm is
O(length of string) + ~~length of pattern~~
~~X~~



Quiz 12

Name and student ID

Rewrite pmatch() in slide 12 using for loop instead of while loop.



Summary

The Array as an Abstract Data Type

- Data structure
- Functions

Structures and Unions

- Similarity and Difference

The Polynomial Abstract Data Type

- Data structure
- Addition

The Sparse Matrix Abstract Data Type

- Data structure
- Transpose

The String Abstract Data Type

- Pattern matching algorithm



Stacks and Queues

Ikjun Yeom



Contents

Stacks

Queues

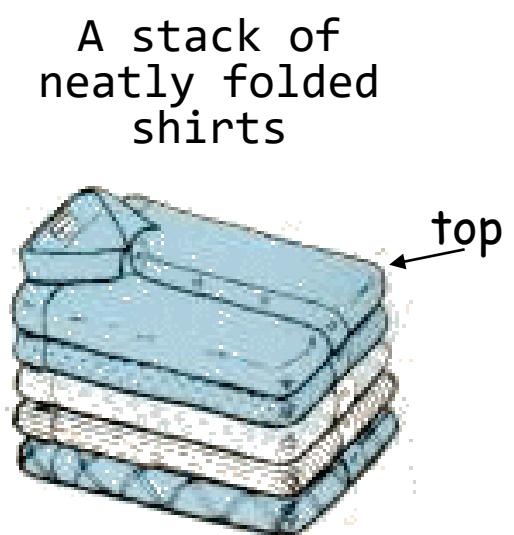
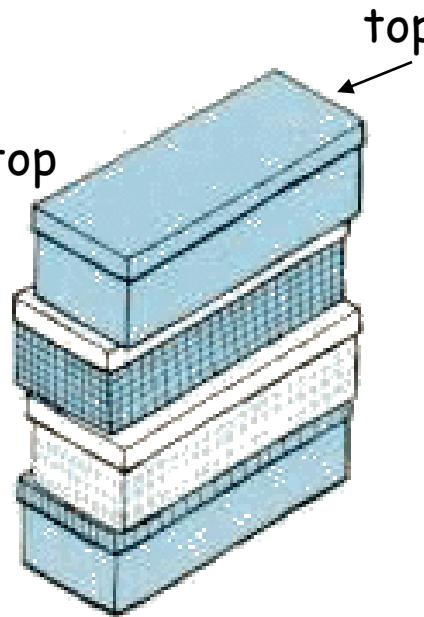
Circular Queues

A Maze Problem

Evaluation of Expressions



Examples of Stack



Stack

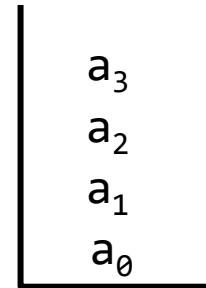
Definition

- An ordered list in which insertions and deletions are made at one end called the *top*
- Given a stack $S=(a_0, \dots, a_{n-1})$

- a_0 is bottom element
- a_{n-1} is top element
- a_i is on top of element a_{i-1} , $0 < i < n$

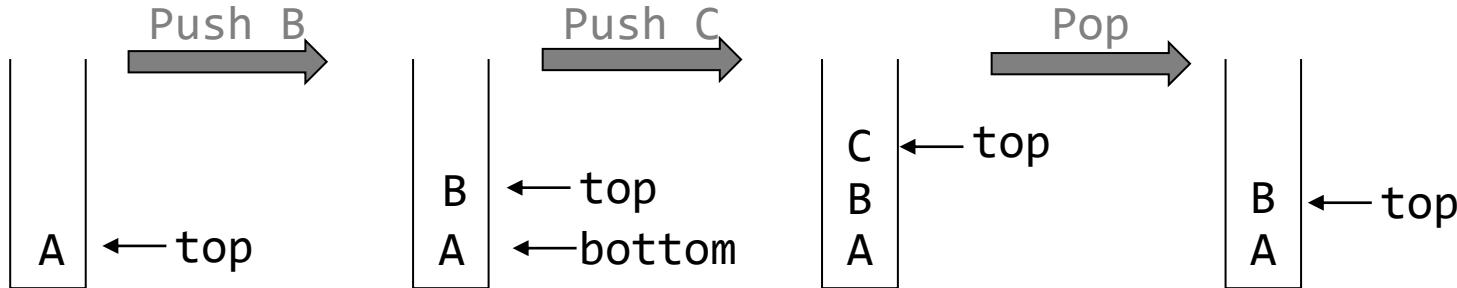
- *Last-In-First-Out (LIFO)*

- Insert the new element into the stack on the top end
- We can only delete and get the top element of the stack

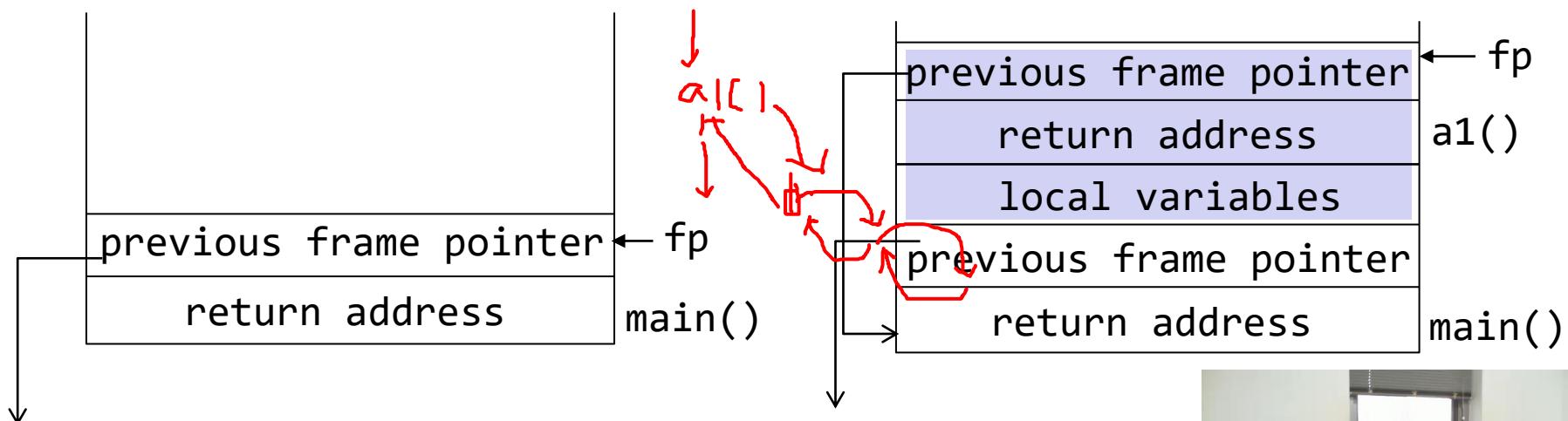


Examples of Stack

A thin box



Process stack (p.108, Exmaple 3.1)



Stack Abstract Data Type

ADT Stack is

objects: A finite ordered list with zero or more elements

functions: for all $stack \in \text{Stack}, item \in \text{element}$,

$\max_{\text{stack}} \text{size} \in \text{positive integer}$

*Stack CreateS(maxStackSize) ::= create an empty stack
whose maximum size is maxStackSize*

Boolean IsFull(stack, maxStackSize) ::= ...

Stack Push(stack, item) ::=

if (isFull(stack)) stackFull

else insert item into top of stack and return

Boolean IsEmpty(stack) ::=

Element Pop(stack) ::=

if (isEmpty(stack)) return

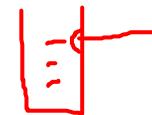
*else remove and return the element at the top of the
 stack*



Stack Implementation (1)

```
#define MAX_STACK_SIZE 100 /*maximum stack size */
typedef struct {
    int key;
    /* other fields may be added*/
} element;
element stack[MAX_STACK_SIZE];
int top = -1;
```

```
void push (element item)
{
    /* add an item to the global stack */
    if (top >= MAX_STACK_SIZE-1) {
        stackFull();
        return;
    }
    stack[++top] = item;
}
```



```
void stackFull()
{
    fprintf(stderr, "Stack is full, cannot
    exit(EXIT_FAILURE);
}
```



Stack Implementation (2)

```
element pop ()  
{  
/* return the top element from the stack, so called 'pop'  
 */  
if (top == -1)  
    return stackEmpty(); /*returns an error key */  
return stack[top--];  
}  
}
```

```
main()  
{  
    element e,f;  
  
    e.key=3;    push(e);  
    e.key=2;    push(e);  
    f=pop();  
    printf ("%d %d\n", top, f.key);  
}
```

[0] [1] [2] [3] [4] [5] [6] [7]

stack

| | | | | | | | | |
|---|---|--|--|--|--|--|--|--|
| 3 | 2 | | | | | | | |
|---|---|--|--|--|--|--|--|--|

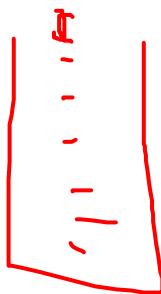
0 2



Quiz 13

Name and student ID

Using the stack implementation in Slide 7 and 8, write a program to generate ten random integer numbers, and push them to a stack. Then print out in the order of first-in-first-out (not last-in-first-out).



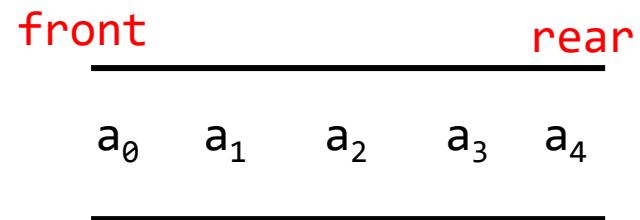
Queue Abstract Data Type

Definition

- An ordered list in which all insertions take place at one end (**rear**) and all deletions take place at the opposite end (**front**)

- Given a queue $Q = (a_0, \dots, a_{n-1})$

- a_0 is front element
- a_{n-1} is rear element
- a_i is behind a_{i-1} , $0 < i < n$



- *First-In-First-Out (FIFO)*

- Insert the new element into the queue on the rear side
- We can only delete/get the front element of the queue



Example of Queue

In a shop



Job scheduling

- Frequently used in computer programming
- Job queue by an operating system
- The jobs are processed in the order they enter the system



Queue Abstract Data Type

ADT Queue is

objects: A finite ordered list with zero or more elements

functions: for $queue \in Queue, item \in element,$
 $maxQueueSize \in \text{positive integer}$

$Queue CreateQ(queue, maxQueueSize) ::= ...$

$Boolean IsFullQ(queue) ::= ...$

$Queue AddQ(queue, item) ::=$
 if ($IsFullQ(queue)$) queueFull
 else insert item at rear of queue and return queue

enqueue

$Boolean IsEmptyQ(queue) ::= ...$

$Element DeleteQ(queue) ::=$
 if ($IsEmpty(queue)$) return
 else remove item and return the item at the front of queue

deq



Queue Implementation (1)

```
#define MAX_QUEUE_SIZE 100 /*maximum queue size */
typedef struct {
    int key;
    /* other fields may be added*/
} element;
element queue[MAX_QUEUE_SIZE];

int rear = -1; int front = -1;

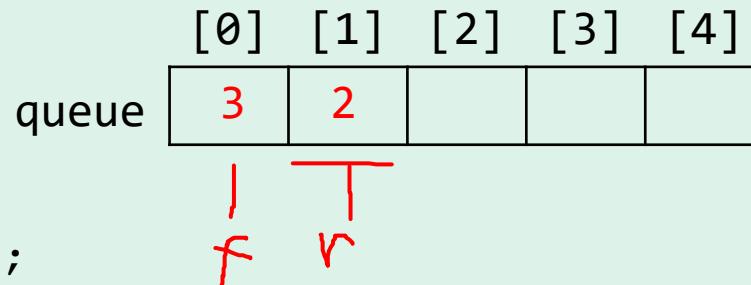
void addq(element item)
{
/* insert an item into a queue, so called 'enqueue' */
    if (rear == MAX_QUEUE_SIZE-1)
        queueFull();
    queue[++rear] = item;
}
```



Queue Implementation (2)

```
element deleteq()
{
    /* delete an item at the front of the queue, so called
     'dequeue' */
    if (front == rear)
        return queueEmpty(); /* return an error key */
    return queue[++front];
}
```

```
main()
{
    element e,f;
    e.key=3;    addq(e);
    e.key=2;    addq(e);
    f=deleteq();
    printf("%d %d %d\n", front, rear, f.key);
}
```



0 1 3



Job Scheduling Example

Insertion and deletion from a sequential queue

| front | rear | Q[0] | Q[1] | Q[2] | Q[3] | Comments |
|-------|------|------|------|------|------|-----------------|
| -1 | -1 | | | | | Queue is empty |
| -1 | 0 | J1 | | | | Job1 is added |
| -1 | 1 | J1 | J2 | | | Job2 is added |
| -1 | 2 | J1 | J2 | J3 | | Job3 is added |
| 0 | 2 | | J2 | J3 | | Job1 is deleted |
| 1 | 2 | | | J3 | | Job2 is deleted |
| 1 | 3 | | | J3 | J4 | Job4 is added |

- The queue gradually shifts to the right
- No available space to add a new item when rear is `(MAX_QUEUE_SIZE - 1)`
- Circular representation is more efficient to avoid the problem



Circular Queue (1)

A queue wraps around the end of the array

Array positions are arranged in a circle rather than in a straight line

- The position next to position MAX_QUEUE_SIZE-1 is 0
- The position precedes 0 is MAX_QUEUE_SIZE-1

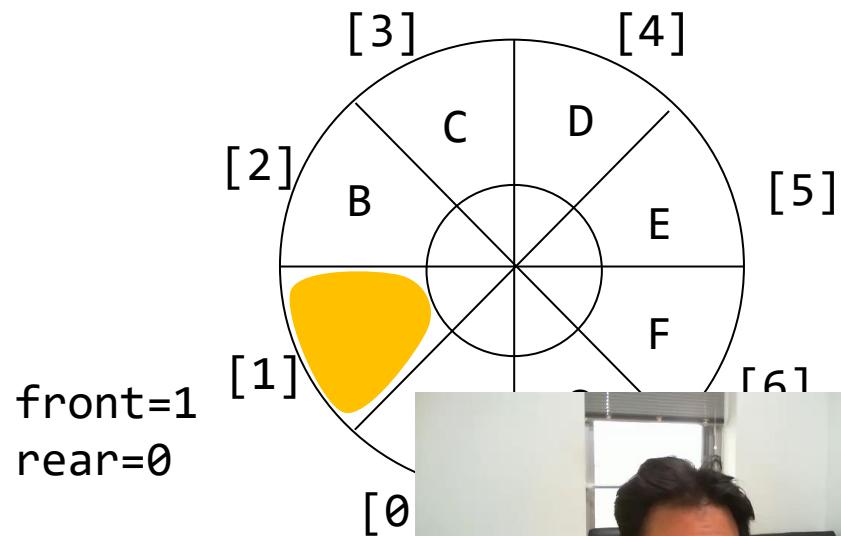
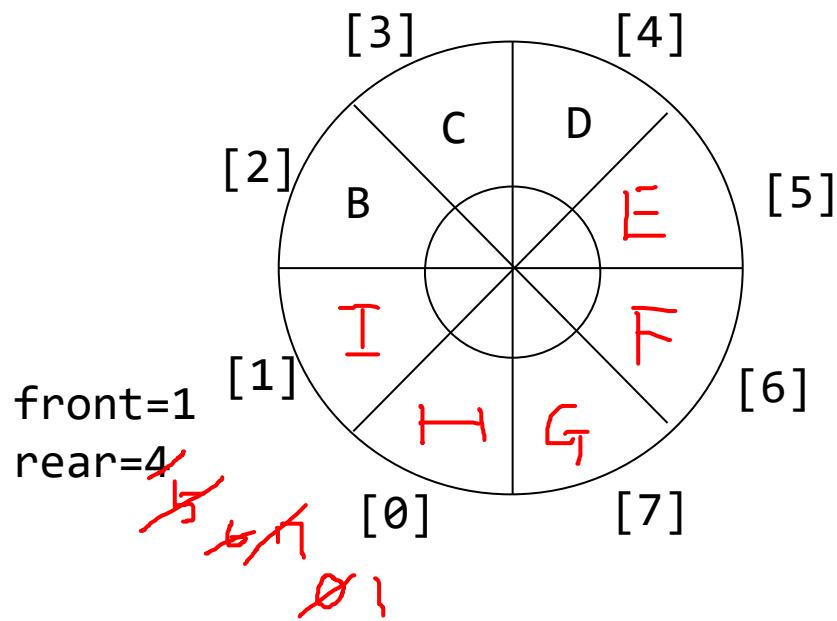
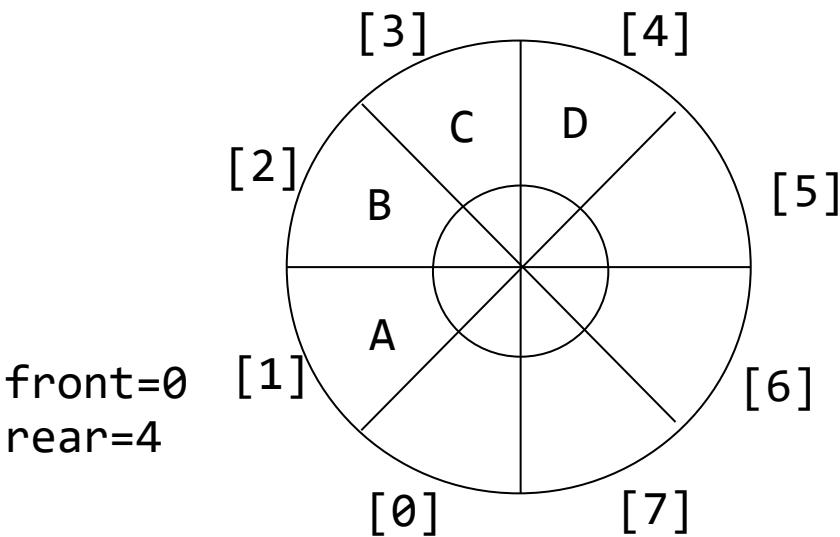
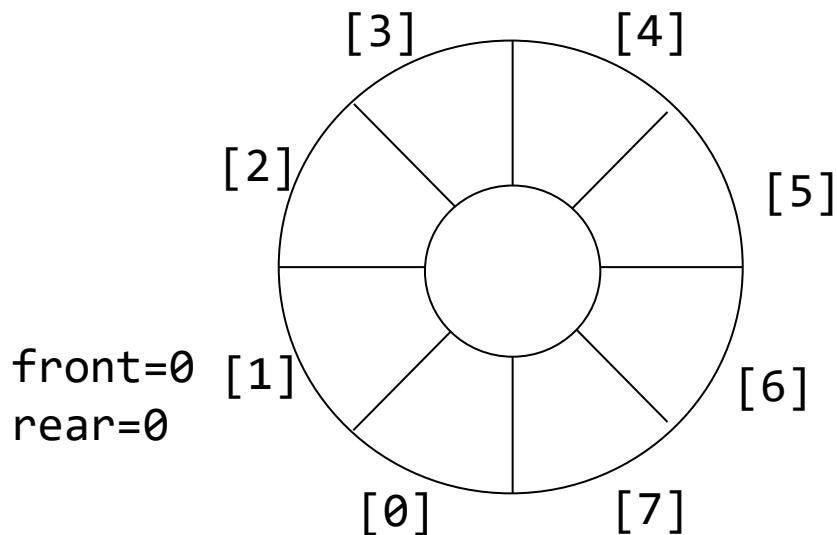
```
( if (rear==MAX_QUEUE_SIZE-1) rear = 0;  
    else rear++;  
→ rear = (rear + 1) % MAX_QUEUE_SIZE
```

qq

| index | 0 | 1 | ... | MAX_QUEUE_SIZE-1 |
|-------|----|---|-----|------------------|
| | 35 | | | 34 |



Circular Queue Operation



Circular Queue (2)

```
element queue[MAX_QUEUE_SIZE];
int front=0, rear=0;

void addq(element item)
{
    rear = (rear+1) % MAX_QUEUE_SIZE;
    if (front == rear) { rear--;
        queueFull(rear); /* print error and exit */
    queue[rear] = item;
}

element deleteq() {
    if (front == rear)
        return queueEmpty();
    front = (front+1) % MAX_QUEUE_SIZE;
    return queue[front];
}
```



Circular Queue Operation Example

| | | | |
|--------|-------------------------|-------------------|-------------------|
| queue | [0] [1] [2] [3] | | front=0 rear=0 |
| Add 3 | [0] [1] [2] [3] | | front=0 rear=1 |
| Add 5 | [0] [1] [2] [3] | | front=0 rear=2 |
| Add 7 | [0] [1] [2] [3] | | front=0 rear=3 |
| Add 8 | Error: Queue is full!!! | front=0 rear=0 | |
| Delete | [0] [1] [2] [3] | | front=1 rear=3 |
| Add 10 | [0] [1] [2] [3] | | front=1 rear=0 |



Quiz 14

Name and student ID

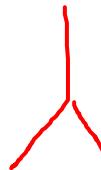
Modify the circular queue implementation in Slide 18 so that we can utilize all the space.



Applications of Stack

Many application areas use stacks:

- Line editing
- Bracket matching
- Maze problem
- Expression evaluation



a b c d ←

{ a , (b + f [4]) * 3 , d + f [5] }



A Maze Problem

Problem

- A value 1 implies a blocked path, and 0 means one can walk right on through.
- Find the way to go out

```
int maze[MAX_ROWS][MAX_COLS];
```

| | | | |
|----------|---|--|--------|
| Entrance | → | 0 1 0 0 0 1 1 0 0 0 1 1 1 1 1 1 0 0 0 1 1 0 1 1 1 0 0 1 1 1 0 1 1 0 0 0 0 1 1 1 1 0 0 1 1 1 1 0 1 1 1 1 0 1 1 0 1 1 0 0 1 1 0 1 0 0 1 0 1 1 1 1 1 1 1 0 0 1 1 0 1 1 1 0 1 0 0 1 0 1 0 1 1 1 1 0 0 1 1 1 1 1 1 1 1 0 0 1 1 0 1 1 0 1 1 1 1 1 0 1 1 1 0 0 0 1 1 0 1 1 0 0 0 0 0 0 0 1 1 1 1 0 0 0 1 1 1 1 0 | → Exit |
|----------|---|--|--------|



Strategy

We may have the chance to go in several directions
Pick one and save our current position and the direction of the next move in the list (stack)
If we have taken a false path, we can return and try another direction by getting the top element of the stack



Allowable Moves (1)

Allowable moves

| | | |
|------------------|---------------|------------------|
| NW [i-1][j-1] | N [i-1][j] | NE [i-1][j+1] |
| W [i][j-1] | X [i][j] | E [i][j+1] |
| SW [i+1][j-1] | S [i+1][j] | SE [i+1][j+1] |

```
typedef struct {
    short int vert; /* -1, 0, +1 */
    short int horiz; /* -1, 0, +1 */
} offsets;
```



Allowable Moves (2)

Table of moves

| Name | Dir | Move[dir].vert | Move[dir].horiz |
|------|-----|----------------|-----------------|
| N | 0 | -1 | 0 |
| NE | 1 | -1 | 1 |
| E | 2 | 0 | 1 |
| SE | 3 | 1 | 1 |
| S | 4 | 1 | 0 |
| SW | 5 | 1 | -1 |
| W | 6 | 0 | -1 |
| NW | 7 | 1 | -1 |

```
offsets move[8];  
nextRow = row + move[dir].vert;  
nextCol = col + move[dir].horiz;
```



Movement history

```
#define MAX_STACK_SIZE 100  
  
typedef struct {  
    short int row; /* current position */  
    short int col; /* current position */  
    short int dir; /* direction of next move */  
} element;  
  
element stack[MAX_STACK_SIZE]
```

Pick one and save our current position and the direction of the next move in the list (stack).

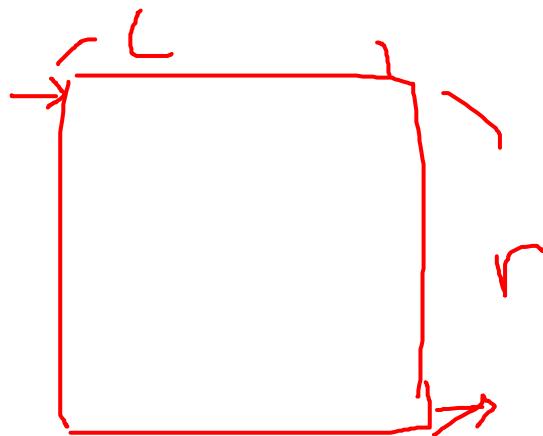
If we have taken a false path, we can return and try another direction by getting the top element of the stack



Quiz 15

Name and student ID

What is the maximum path length from start to finish for any maze of dimensions rows x columns?

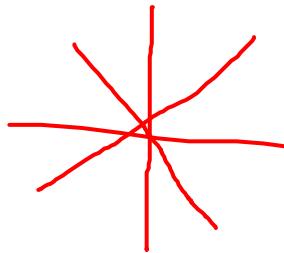


Strategy

We may have the chance to go in several directions

Pick one and save our current position and the direction of the next move in the list (stack)

If we have taken a false path, we can return and try another direction by getting the top element of the stack



Initial Maze Algorithm

Initialize a stack to the maze's entrance coordinates and direction to north;

```
int mark[MAX_ROWS][MAX_COLS];
```

```
while (stack is not empty) {
    <row, col, dir> = delete from top of stack;
    while (there are more moves from current position) {
        dir = direction of move;
        <nextRow, nextCol> = coordinate of next move;
        if ((nextRow==EXIT_ROW) && (nextCol==EXIT_COL))
            success;
        if (maze[nextRow][nextCol] == 0) &&
            mark[nextRow][nextCol] == 0) {
            mark[nextRow][nextCol] = 1;
            add <row, col, next dir> to the top of the stack;
            row = nextRow; col = nextCol; dir = north;
        }
    }
    printf("No path found");
```



Maze(Examined)

[0] [1] [2] [3] [4] [5] [6]

| | | | | | | | |
|-----|--|---|---|---|---|---|---|
| [0] | | 1 | 1 | 1 | 1 | 1 | 1 |
| [1] | | 0 | 0 | 1 | 1 | 0 | 1 |
| [2] | | | 0 | 0 | 1 | 1 | |
| [3] | | | 0 | 1 | 1 | | |
| [4] | | | | 0 | 1 | | |
| [5] | | | | | 0 | | |
| [6] | | | | | | | |

ENTRY = (1,1), EXIT = (5,5)

| | | |
|-------|---------------|--------------|
| (1,1) | PUSH (1,1,ES) | mark[1][2]=1 |
| (1,2) | PUSH (1,2,S) | mark[2][3]=1 |
| (2,3) | PUSH (2,3,ES) | mark[2][4]=1 |
| (2,4) | PUSH (2,4,E) | mark[1][5]=1 |
| (1,5) | POP | |
| (2,4) | PUSH (2,4,W) | mark[3][3]=1 |
| (3,3) | PUSH (3,3,S) | mark[4][4]=1 |
| (4,4) | FOUND!!! | |

(1,1)(1,2)(2,3)(2,4)(3,3)(4,4)(5,5)

Maze (Original)

[0] [1] [2] [3] [4] [5] [6]

| | | | | | | | |
|-----|---|---|---|---|---|---|---|
| [0] | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| [1] | 1 | 0 | 0 | 1 | 1 | 0 | 1 |
| [2] | 1 | 1 | 1 | 0 | 0 | 1 | 1 |
| [3] | 1 | 1 | 1 | 0 | 1 | 1 | 1 |
| [4] | 1 | 1 | 0 | 1 | 0 | 1 | 1 |
| [5] | 1 | 0 | 1 | 1 | 0 | 0 | 1 |
| [6] | 1 | 1 | 1 | 1 | 1 | 1 | 1 |

STACK

| | |
|-----|----------|
| [5] | |
| [4] | (3,3,S) |
| [3] | (2,4,W) |
| [2] | (2,3,ES) |
| [1] | (1,2,S) |
| [0] | (1,1,ES) |



Maze Search Function (1)

```
elements stack[MAX_STACK_SIZE];
offset move[8];
int maze[MAX_ROWS] [MAX_COLS], mark[MAX_ROWS] [MAX_COLS];
int top;

void path(void)
{/* output a path through the maze if such a path exists*/
    int i, row, col, nextRow, nextCol, dir, found=FALSE;
    element position;

    mark[1][1]=1; top=0;
    stack[0].row=1; stack[0].col=1; stack[0].dir=1;

    while (top>-1 && !found) {
        position = pop();
        row = position.row;
        col = position.col;
        dir = position.dir;
```



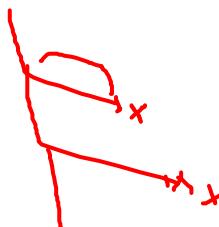
Maze Search Function (2)

```
while (dir < 8 && !found) {  
    /* move in direction dir*/  
    nextRow = row + move[dir].vert;  
    nextCol = col + move[dir].horiz;  
    if (nextRow==EXIT_ROW && nextCol==EXIT_COL)  
        found = TRUE;  
    else if ( !maze[nextRow] [nextCol]  
              && !mark[nextRow] [nextCol]) {  
        mark[nextRow] [nextCol] = 1;  
        position.row = row; position.col = col;  
        position.dir = ++dir;  
        push(position);  
        row = nextRow; col = nextCol; dir = 0;  
    }  
    else ++dir;  
} /* while (dir < 8 & !found)  
} /* while (top>-1 && !found) */
```



Maze Search Function (3)

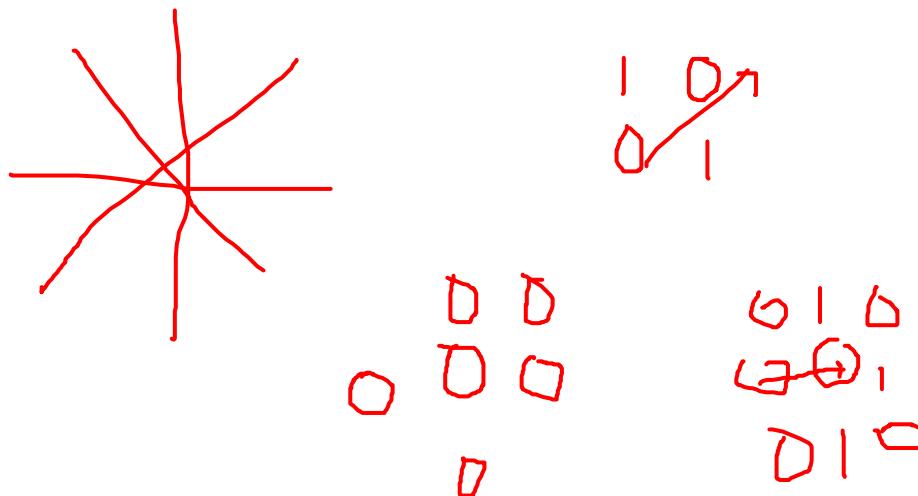
```
if (found) {  
    printf("The path is:\n");  
    printf("row  col\n");  
    for (i=0; i<=top; i++)  
        printf("%2d%5d", stack[i].row, stack[i].col");  
    printf("%2d%5d\n", row, col);  
    printf("%2d%5d\n", EXIT_ROW, EXIT_COL);  
}  
else printf("The maze does not have a path\n");  
}
```



Quiz 16

Name and student ID

Suppose a slightly modified maze problem such that we can move to only four directions (North, East, South, and West). Then, how do we need to modify path() ?



Evaluation of Expressions (1)

Example of expressions

- `((rear+1==front) || ((rear==MAX_QUEUE_SIZE-1 && !front))`
- `x=a/b-c+d*e-a*c;`

Order in evaluating the expression

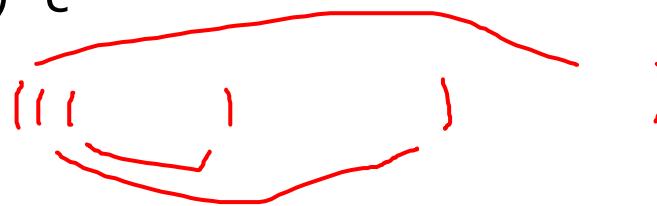
- Precedence hierarchy of operators
- Associativity of an operator (with the same precedence)
- E.g. `C=a++*2+4%3**size` → `C=((a++)*2)+((4%3)*(*size))`



Evaluation of Expressions (2)

Infix notation

- Binary operator is in-between its two operands
- e.g. $a+b$, $a-c*d$, $(a+b)*c$



Prefix notation

- Operator appears before its operands – normally used in the arithmetic expressions of Lisp
- e.g. $+ab$, $-a*cd$, $*+abc$



Postfix notation

- Each operator appears after its operands - used by compiler
- e.g. $ab+$, $acd*-$, $ab+c*$



Evaluating Postfix Expression

Scan the Postfix string from left to right

Initialize an empty stack 

Place the operands on a stack until we find an operator

If we find an operator,

1. Remove, from the stack, the correct number of operands for the operator
2. Perform the operation
3. Place the result back on the stack

Continue this fashion until we reach the end of the expression

Then, remove the answer from the top of the stack



Parsing the Expression

```
precedence getToken (char *symbol, int * n)
{
    *symbol = expr[(*n)++];
    switch (*symbol) {
        case '(' : return lparen;
        case ')' : return rparen;
        case '+' : return plus;
        case '-' : return minus;
        case '/' : return divide;
        case '*' : return times;
        case '%' : return mod;
        case ' ' : return eos;
        default : return operand;
    }
}
```

9000

3 1



Evaluating of Postfix Expression

```
int eval(void) {
    precedence token;  char symbol;  int op1,op2;
    int n = 0; top = -1;
    token = getToken(&symbol, &n);
    while (token != eos) {
        if (token==operand) push(symbol-'0');
        else {
            op2=pop(); op1=pop();
            switch(token) {
                case plus: push(op1+op2); break;
                case minus: push(op1-op2); break;
                case times: push(op1*op2); break;
                case divide: push(op1/op2); break;
                case mod: push(op1%op2);
            } /* switch */
        } /* else */
        token = getToken(&symbol, &n);
    } /* while */
    return pop(); /* return result */
}
```

" " [] ' ' []



Evaluating Postfix Expression Example

Example: 62/3-42*+

| Token | Stack [0] | [1] | [2] | Top |
|-------|--------------|-----|-----|-----|
| 6 | 6 | | | 0 |
| 2 | 6 | 2 | | 1 |
| / | 3 | | | 0 |
| 3 | 3 | 3 | | 1 |
| - | 0 | | | 0 |
| 4 | 0 | 4 | | 1 |
| 2 | 0 | 4 | 2 | 2 |
| * | 0 | 8 | | 1 |
| + | 8 | | | 0 |
| eos | | | | -1 |



Quiz 17

Name and student ID

Evaluate the following postfix expressions.

(assume that all the operands are single digit)

(a) 7 3 2 + * 3 -

(b) 3 4 5 * 4 / +

(c) 9 3 * 4 5 * + 3 -



Infix to Postfix (1)

Infix to postfix

- Fully parenthesize the expression
- Move all binary operators so that they replace their corresponding right parentheses
- Delete all parentheses

e.g. $x = A/B - C + D * E - A * C$

7+3-2 73+2-

→ $x = (((A/B) - C) + (D * E)) - (A * C)$

7+3*2 732*+

→ $(((A/B) - C) + (D * E)) (A * C) -$

(7+3)*2 73+2*

→ $(A/B) C - DE * + AC * -$

→ $AB/C - DE * + AC * -$



Infix to Postfix (2)

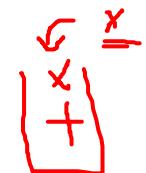
Scan the infix expression from left to right

Operands are passed to the output expression as they are encountered

1 1 1 () () $\sqrt{a+b} \times c$ abcxt

Save the operators until we know their correct placement
and output the higher precedence operators first

→ Stack operators as long as the precedence of the operator at the top of the stack is less than the precedence of the incoming operator



Unstack when we reach the end of the expression



Infix to Postfix (3)

Left parenthesis is placed in the stack whenever it is found in the expression

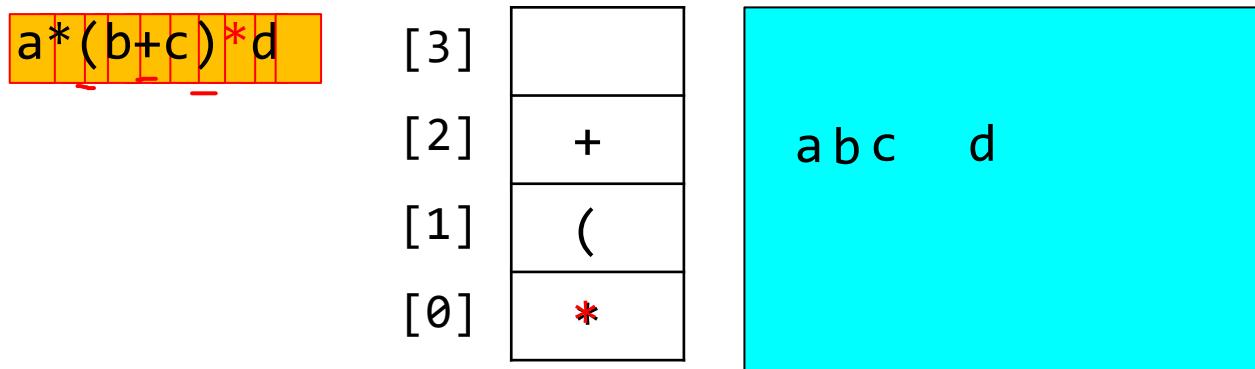
Stack operators until we reach the right parenthesis

Left parenthesis is unstacked only when its matching right parenthesis is found

Right parenthesis is never placed in the stack

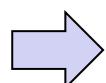


Infix to Postfix Example



The left parenthesis is placed in the stack whenever found
→ High precedence operator

The left parenthesis is stacked until its matching right parenthesis is found
→ Low precedence operator



Two types of precedence :
in-stack precedence and *incoming precedence*



Infix to Postfix Example

Example : $a*(b+c-d)\%e \rightarrow abc+d-*e\%$

| Token | Stack | | | | | | Top | Output |
|-------|-------|-----|-----|--|--|--|-----|---------|
| | [0] | [1] | [2] | | | | | |
| a | | | | | | | -1 | a |
| * | * | | | | | | 0 | a |
| (| * | (| | | | | 1 | a |
| b | * | (| | | | | 1 | ab |
| + | * | (| + | | | | 2 | ab |
| c | * | (| + | | | | 2 | abc |
| - | * | (| - | | | | 2 | abc+ |
| d | * | (| - | | | | 2 | abc+d |
|) | * | | | | | | 0 | abc+d- |
| % | % | | | | | | 0 | abc+d-* |
| e | % | | | | | | 0 | |
| eos | | | | | | | -1 | |



Infix to Postfix Implementation (1)

```
#define MAX_STACK_SIZE 100 /* maximum stack size */  
#define MAX_EXPR_SIZE 100 /* max size of expression */  
typedef enum {lparen, rparen, plus, minus, times, divide,  
             mod, eos, operand } precedence;  
char expr[MAX_EXPR_SIZE]; /* input string */  
  
precedence stack[MAX_STACK_SIZE];  
/* isp and icp arrays - index is value of precedence  
lparen, rparen, plus, minus, times, divide, mod, eos */  
/* isp: in stack precedence, icp: incoming precedence */  
static int isp[] = { 0, 19, 12, 12, 13, 13, 13, 0 };  
static int icp[] = { 20, 19, 12, 12, 13, 13, 13, 0 };
```

Left parenthesis has the highest priority when incoming
the lowest priority while in the stack



Infix to Postfix Implementation (2)

```
void postfix(void) {  
    char symbol; precedence token; int n = 0; int top = 0;  
    stack[0] = eos;  
    for (token=getToken(&symbol,&n); token!=eos;  
         token=getToken(&symbol,&n)) {  
        if (token == operand) printf("%c", symbol);  
        else if (token == rparen) {  
            while (stack[top] != lparen)  
                printToken(pop());  
            pop(); /* discard the left parenthesis */  
        } else {  
            while (isp[stack[top]] >= icp[token])  
                printToken(pop());  
            push(token);  
        }  
    }  
    while ((token=pop()) != eos) printToken(token);  
}
```

plus +

+ - [



Quiz 18

Name and student ID

Write the postfix form of the following expressions:

(a) $a * b * c$

$$\begin{array}{c} a + b \\ \underline{-} \end{array} \quad \begin{array}{c} ab \\ \underline{\underline{+}} \end{array}$$

(b) $-a + b - c + d$

(c) $a * -b + c$

(d) $(a + b) * d + e / (f + a * d) + c$

$$\frac{(a + b)}{d}$$

$$a \quad a b - +$$



Lists



Content

Singly Linked Lists and Chains

Representing Chains in C

Linked Stacks and Queues

Polynomials

Additional List Operations

Equivalence Relations

Sparse Matrices

Doubly Linked Lists



Why Linked List

(BAT, CAT, EAT, FAT, HAT, JAT, LAT, MAT, OAT, PAT, RAT,
SAT, VAT, WAT) Sorted ordered list

If sequential mapping is used,

| | | | | | | | | | | | | | | |
|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|--|
| BAT | CAT | EAT | FAT | HAT | JAT | LAT | MAT | OAT | PAT | RAT | SAT | VAT | WAT | |
|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|--|



How about inserting GAT?

| | | | | | | | | | | | | | | |
|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|
| BAT | CAT | EAT | FAT | GAT | HAT | JAT | LAT | MAT | OAT | PAT | RAT | SAT | VAT | WAT |
|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|

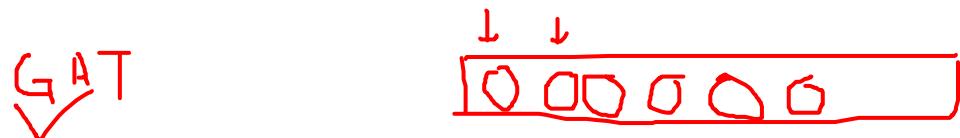
How about deleting LAT?

| | | | | | | | | | | | | | | |
|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|--|
| BAT | CAT | EAT | FAT | GAT | HAT | JAT | MAT | OAT | PAT | RAT | SAT | VAT | WAT | |
|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|--|



Linked Lists (1)

- List elements are stored, in memory, in an arbitrary order
- Explicit information (called a *Link*) is used to go from one element to the next

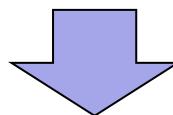


(BAT, CAT, EAT, FAT, HAT, VAT, WAT)

| | [1] | [2] | [3] | [4] | [5] | [6] | [7] | [8] | [9] | [10] | [11] |
|---------|-----|-----|-----|----------|-----|-----|-----|-----|-----|------|------|
| data | HAT | | CAT | EAT | | | WAT | BAT | FAT | HAT | VAT |
| link | 11 | | 4 | 9 | | | 0 | 3 | 10 | 1 | 7 |
| first=8 | | | 1 | <u>2</u> | | | 0 | | | | |

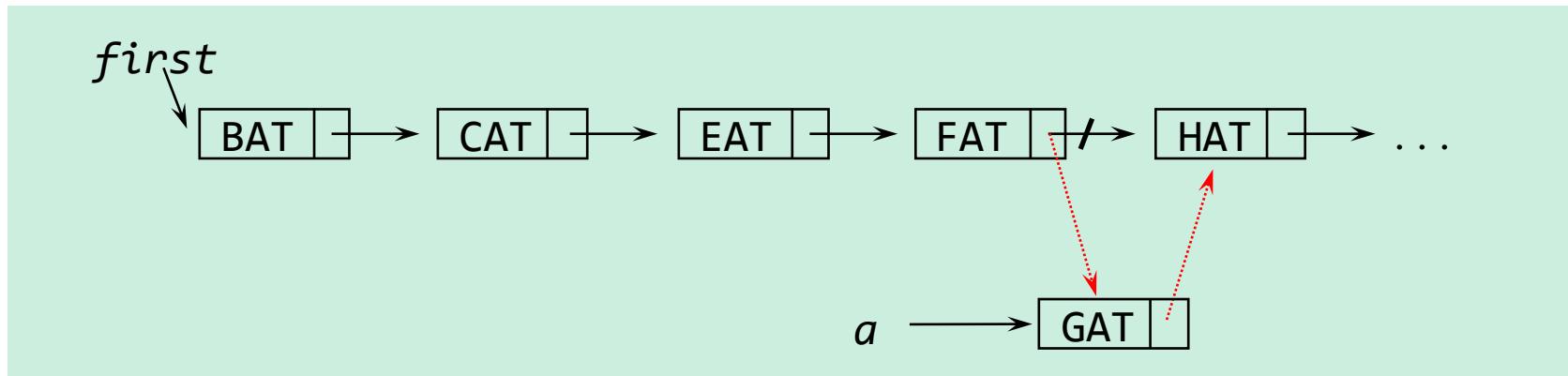


Linked Lists (2)



Inserting GAT into data[5]

| | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 |
|-------------|-----|---|-----|-----|-----|---|-----|-----|-----|----|-----|
| <i>data</i> | HAT | | CAT | EAT | GAT | | WAT | BAT | FAT | | VAT |
| <i>link</i> | 11 | | 4 | 9 | 1 | | 0 | 3 | 5 | | 7 |



Representing Chains in C (1)

A chain is a linked list in which each node represents one element

There is a link or pointer from one element to the next

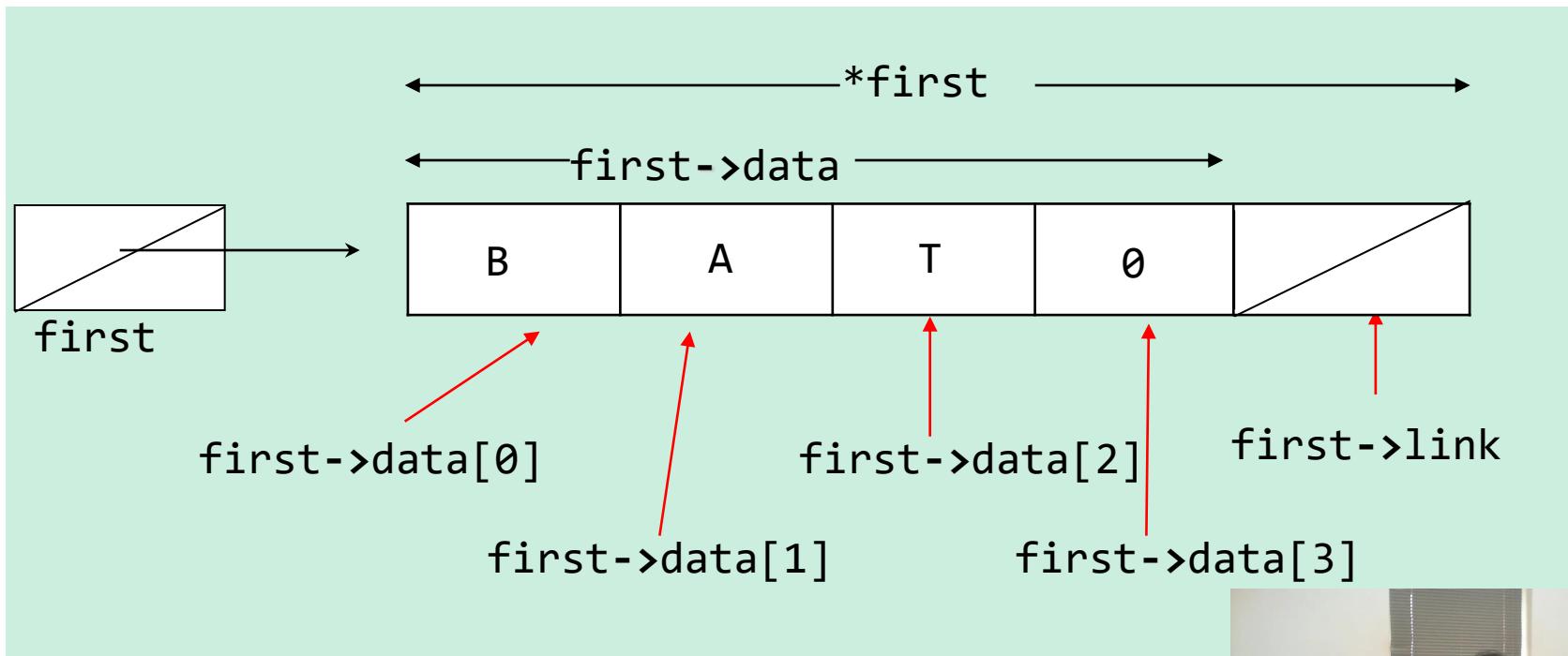
```
typedef struct listNode* listPointer  
typedef struct listNode {  
    char data[4];  
    listPointer link;  
};
```



Representing Chains in C (2)

```
listPointer first = NULL;  
MALLOC(first, sizeof(*first))  
strcpy(first->data, "BAT");  
first->link=NULL;
```

```
typedef struct listNode *listPointer;  
typedef struct listNode {  
    char data[4];  
    listPointer link;  
};
```



Two Nodes Linked List

```
listPointer create2()
{ /* Create a linked list with two nodes */
    listPointer first, second;      listPointer ptr=create2();
```

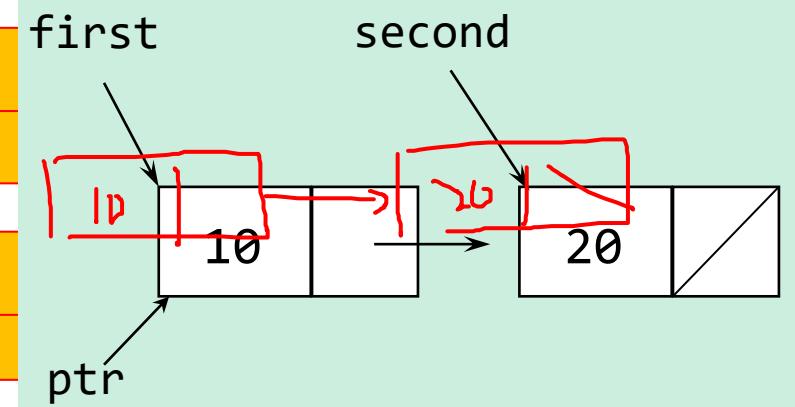
```
MALLOC(first, sizeof(*first));
MALLOC(second, sizeof(*second));
```

```
second->link = NULL;
second->data = 20;
```

```
first->data = 10;
first->link = second;
```

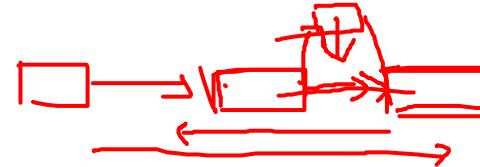
```
return first;
```

```
}
```



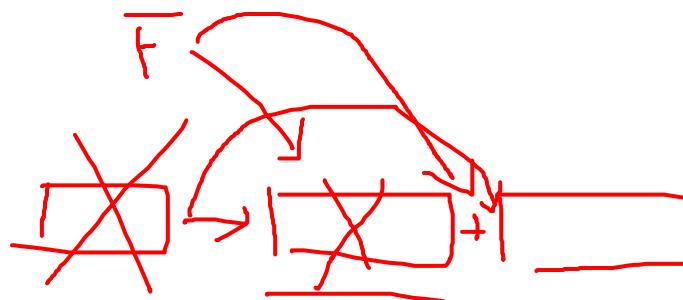
List Insertion

```
void insert50(listPointer* first, listPointer x) (p. 153, Figure 4.7)
{ /* insert a new node with data=50 into the chain first
   after node x */
    listPointer temp;
    MALLOC(temp, sizeof(*temp));
    temp->data = 50;
    if(*first) { /* if (*first != NULL) */
        temp->link = x->link;
        x->link = temp;
    }
    else { /* if (*first == NULL) */
        temp->link = NULL;
        *first = temp;
    }
}
```



List Delete

```
void delete(listPointer *first, listPointer trail,
    listPointer x)                                (p. 154, Figure 4.9)
{ /* 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);
}
delete(&list, list, y);
```

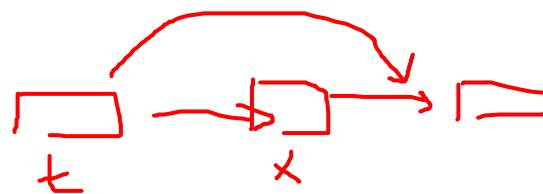


Quiz 19

Name and student ID

Modify delete() function in the previous slide so that it takes only two arguments, first and x without trail.

```
void delete(listPointer *first, listPointer x) {
```

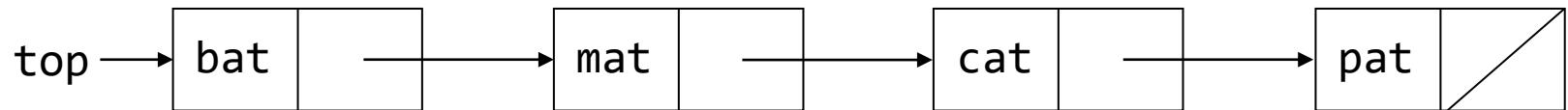


}

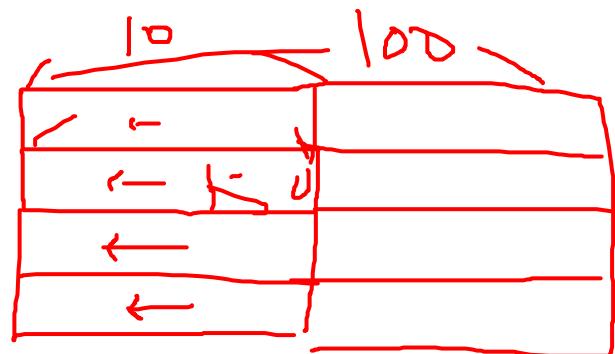
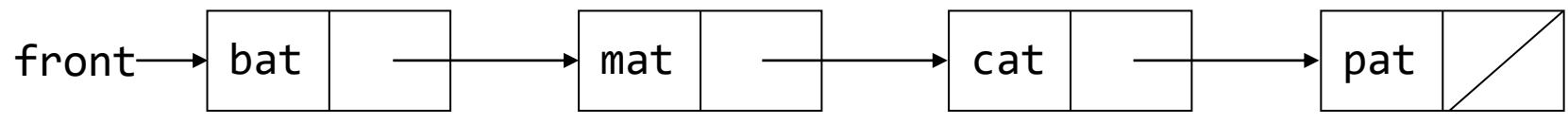


Linked Stacks and Queues

Linked Stack



Linked queue

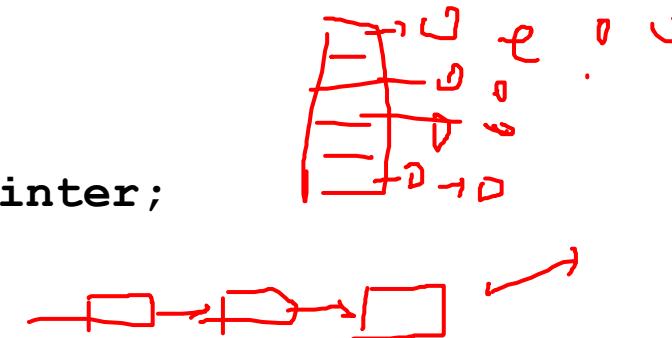


↑
rear



Multiple 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]; /* multiple stacks */
```



Initialization

- $\text{top}[i] = \text{NULL}$, $0 \leq i < \text{MAX_STACKS}$

Boundary conditions

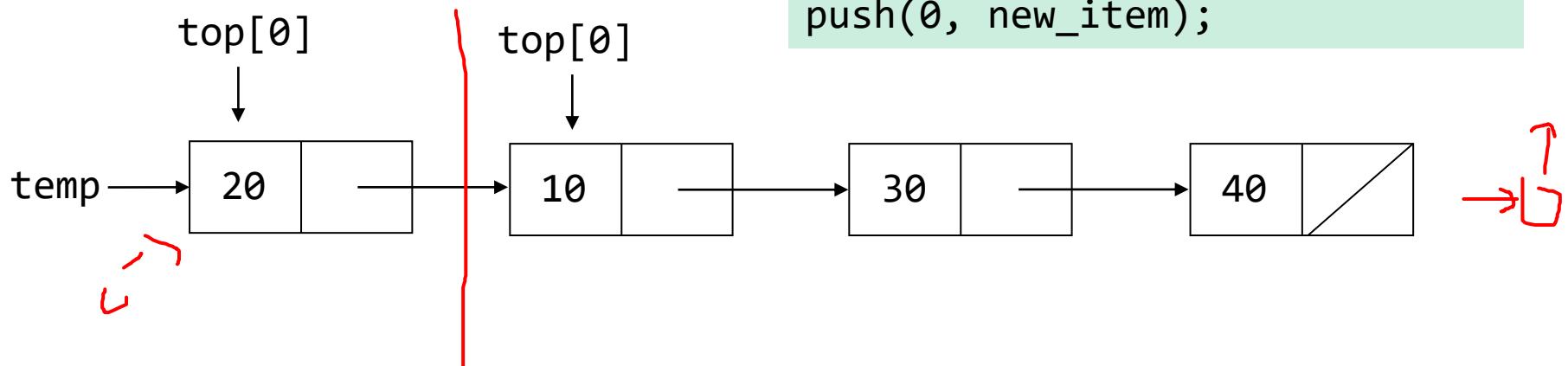
- $\text{IS_EMPTY}(\text{top}[i])$: no item is available in i -th stack
- $\text{IS_FULL}(\text{temp})$: no memory space is available



Stack Operations (1)

```
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;
}
```

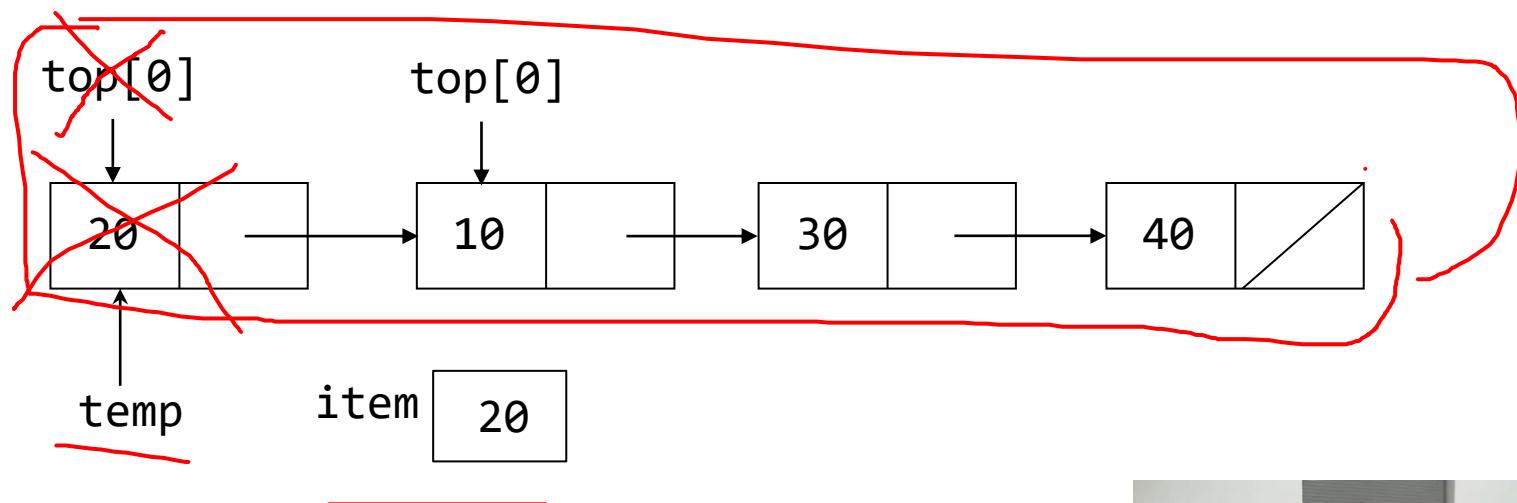
```
element new_item;
new_item.key=20;
push(0, new_item);
```



Stack Operations (2)

```
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 item;
}
```

```
item = pop(0);
printf("%d\n", item.key);
```



Multiple Queues

```
#define MAX_QUEUE 10      /* Maximum Number of Queues */  
typedef struct queue* queuePointer;  
typedef struct queue {  
    element data;  
    queuePointer link;  
};  
queuePointer front[MAX_QUEUES], rear[MAX_QUEUES];
```

Initial condition

- $\text{front}[i] = \text{NULL}$, $0 \leq i < \text{MAX_QUEUES}$



Boundary conditions

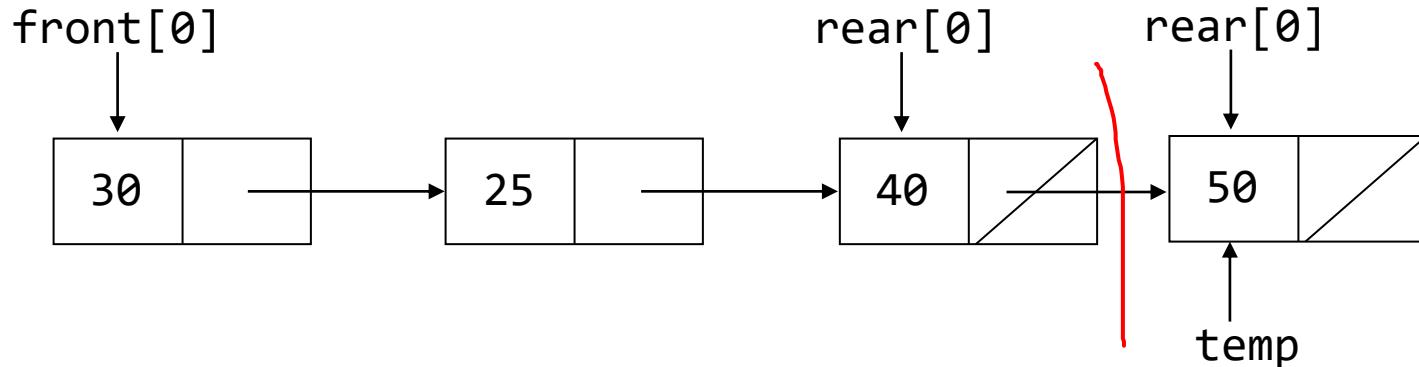
- $\text{IS_EMPTY}(\text{front}[i])$: no item is available in i -th queue
- $\text{IS_FULL}(\text{temp})$: no memory space is available



Queue Operations (1)

```
void addq (int i, element item)
{ /* insert an item at 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;
}
```

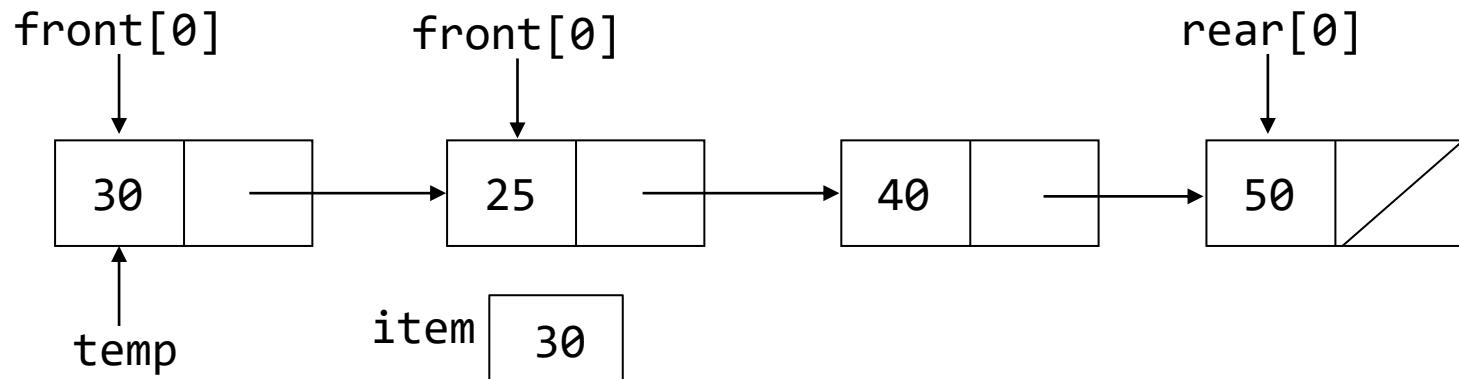
element item;
item.key=50;
addq(0, item);



Queue Operations (2)

```
element deleteq (int i)
{ /* Delete an item from queue i */
queuePointer temp = front[i];
element item;
if (!temp) return queueEmpty();
item = temp->data;
front[i] = temp->link;
free(temp);
return item;
}
```

```
item=deleteq(0);
printf("%d\n", item.key);
```

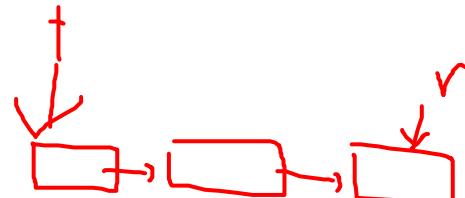


Quiz 20

Name and student ID

In slide 17, modify addq() with the assumption that we have only front not rear.

```
void addq (int i, element item) {
```



```
}
```

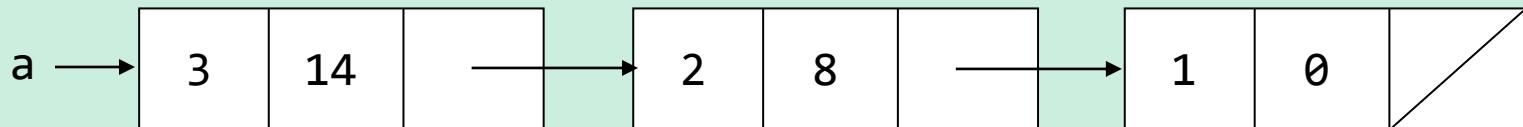


Polynomials - Representation

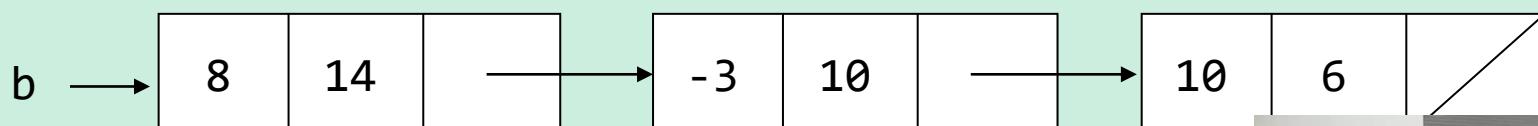
```
typedef struct polyNode* polyPointer;  
typedef struct polyNode {  
    int coef;  
    int expon;  
    polyPointer link;  
};  
polyPointer a,b;
```



$$A(x) = 3x^{14} + 2x^8 + 1$$

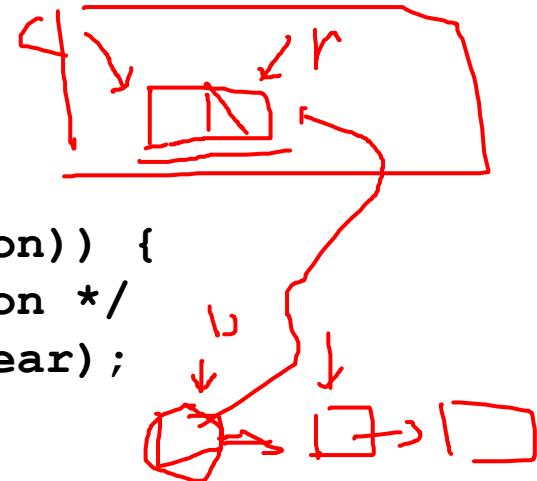


$$B(x) = 8x^{14} - 3x^{10} + 10x^6$$



Add Two Polynomials (1)

```
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;
    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 > a->expon */
                attach(a->coef,a->expon,&rear);
                a = a->link;
        }
    }
}
```



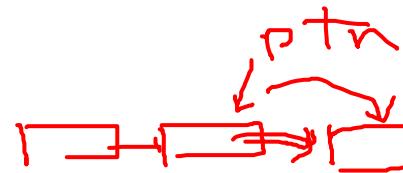
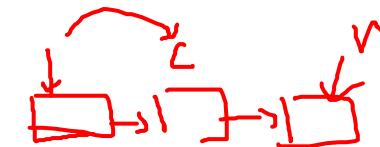
Add Two Polynomials (2)

```
/* copy the rest of list a and 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 the useless initial node */
temp = c;      c = c->link;      free(temp);
return c;
}

void attach(float coefficient, int exponent, polyPointer*
ptr)
{
    polyPointer temp;
    MALLOC(temp, sizeof(*ptr));

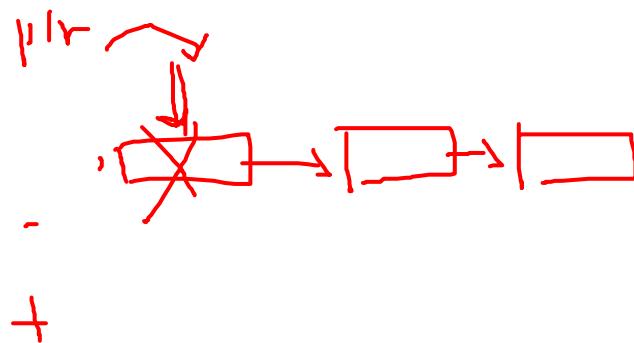
    temp->coef = coefficient;
    temp->expon = exponent;
    (*ptr)->link = temp;
    *ptr = temp;
}
```



Erasing Polynomials

It would be useful to reclaim the nodes that are used temporarily

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



Quiz 21

Name and student ID

In the previous slide, modify `erase()` using `for` loop instead of `while()`.

```
void erase(polyPointer* ptr) {
```

```
}
```



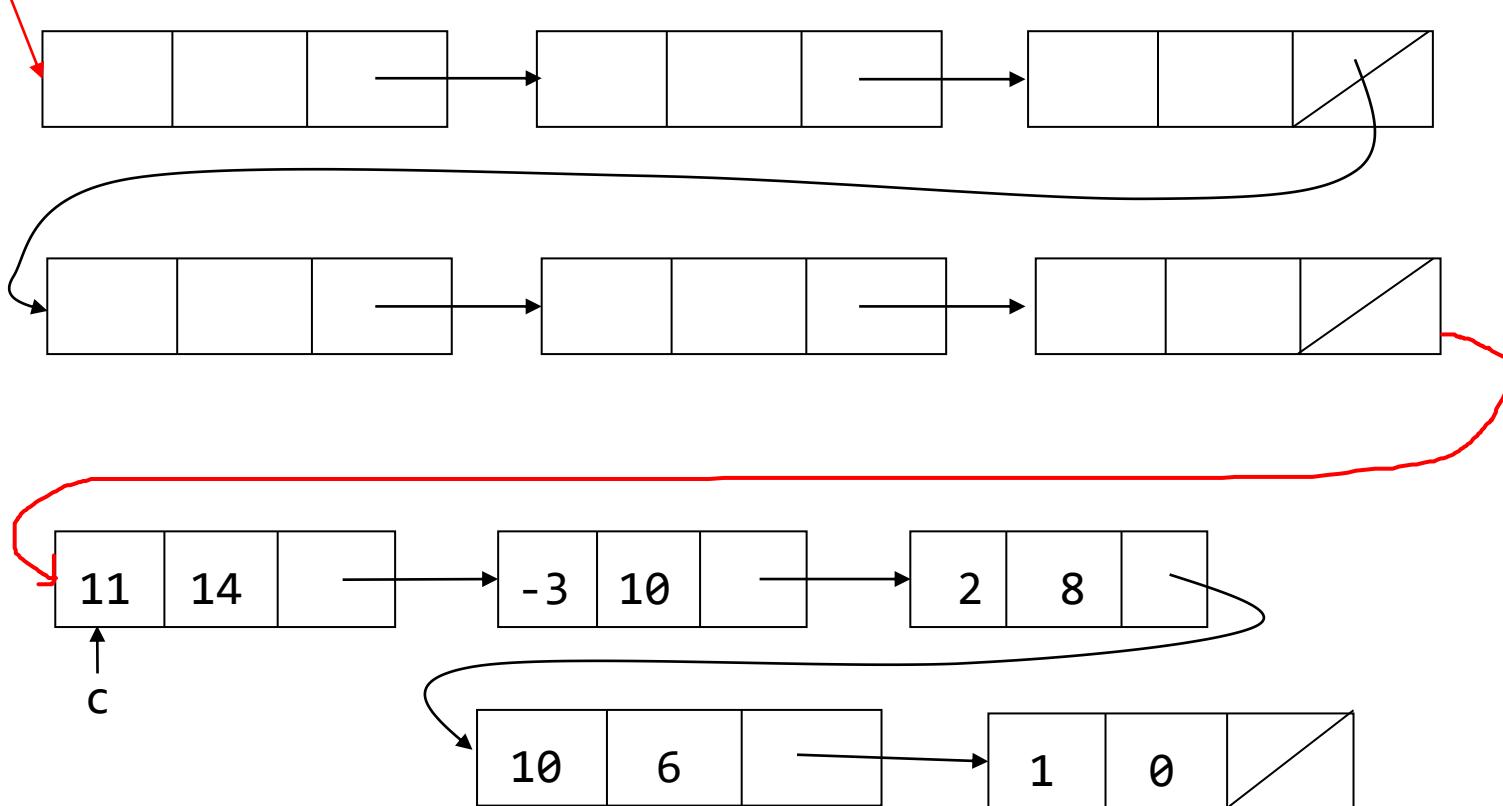
Lists Part 2



Memory Reuse (1)

```
/* a global variable that points to the first node  
of the free nodes list */
```

avail

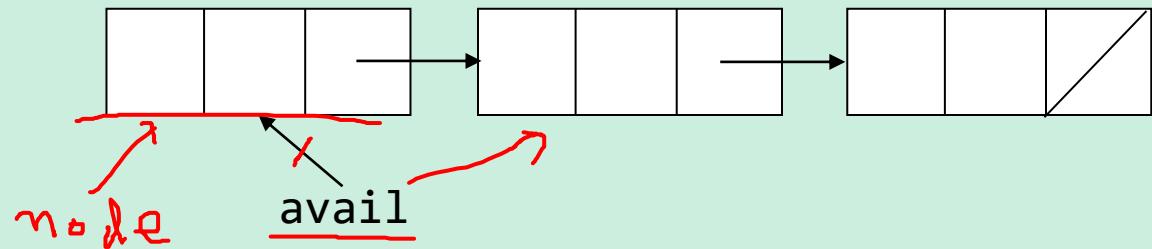


Memory Reuse (2)

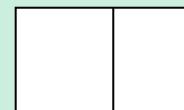
```
polyPointer getNode(void)
{ /* provide a node for use*/
    polyPointer node;
    if (avail) {
        node = avail;
        avail = avail->link;
    }
    else MALLOC(node, sizeof(*node));
    return node;
}
```

```
polyPointer avail;
/* a global variable */
```

```
polyPointer temp=getNode();
```



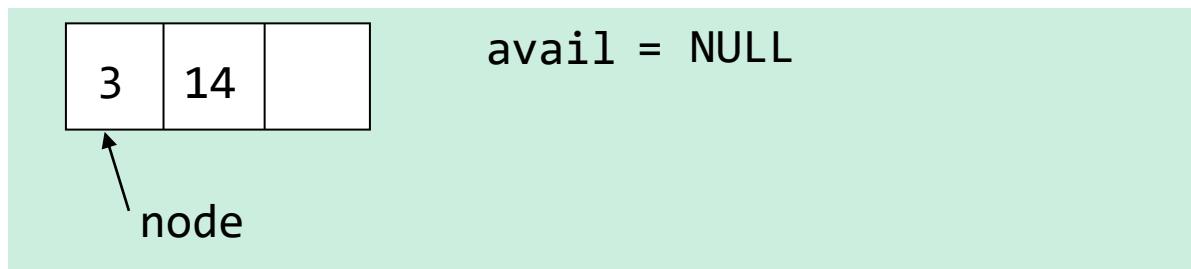
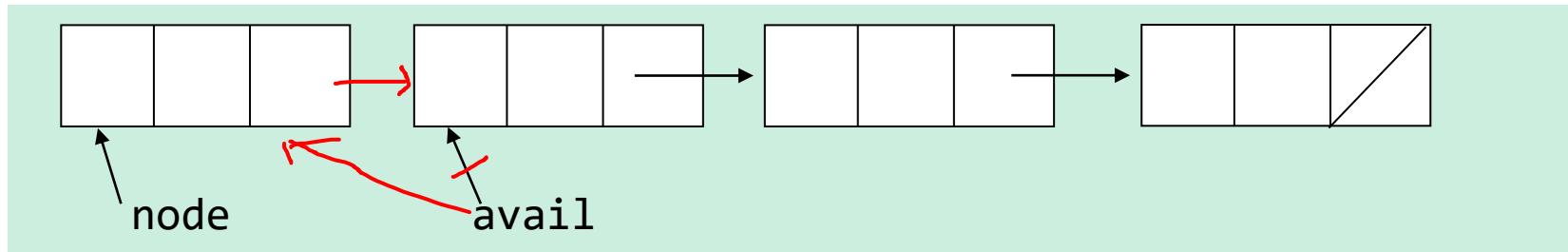
```
avail=NULL
```



Memory Reuse (3)

```
void retNode(polyPointer node)
{ /* return a node to the available list */
    node->link = avail;
    avail = node;
}
```

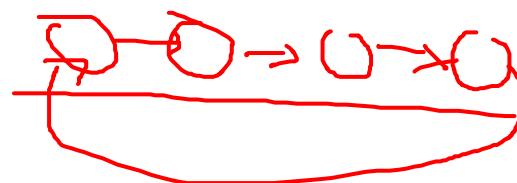
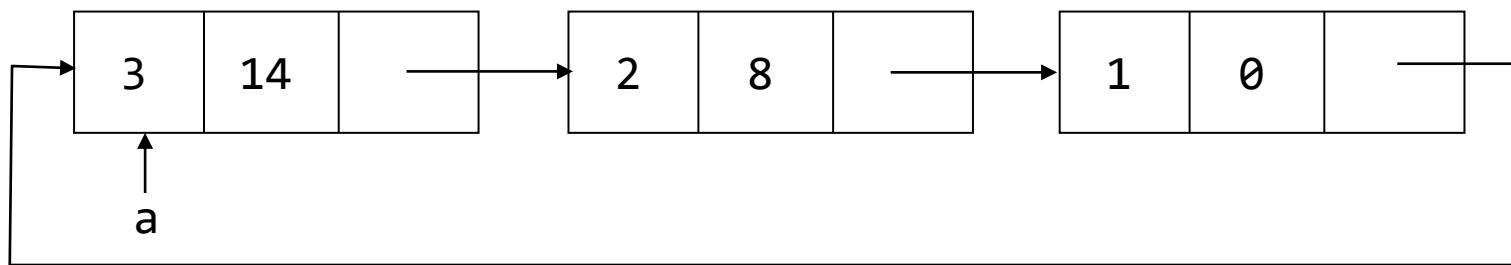
```
polyPointer avail;
/* a global variable that
points to the first node of
the free nodes list */
```



Circular Representation of Polynomials

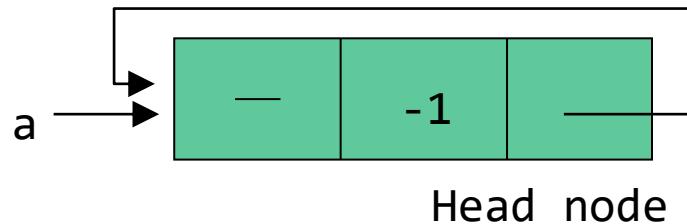
Circular representation

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

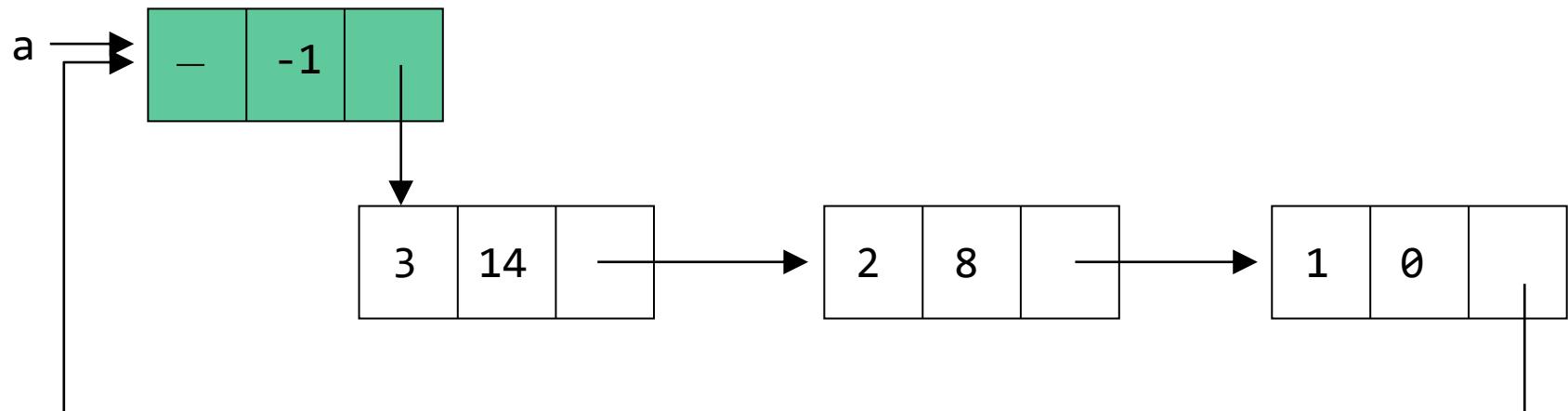


Polynomial : Circularly Linked List (1)

Zero polynomial

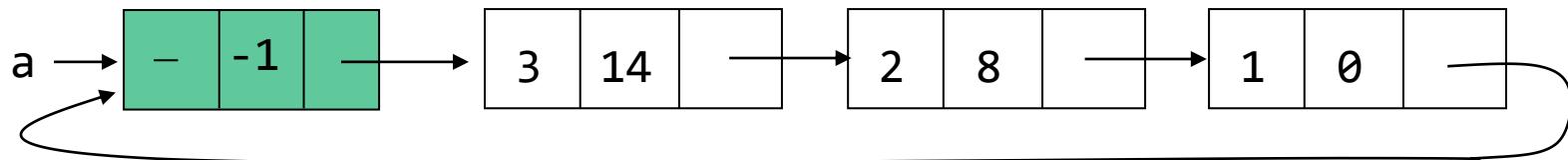


$$3x^{14} + 2x^8 + 1$$

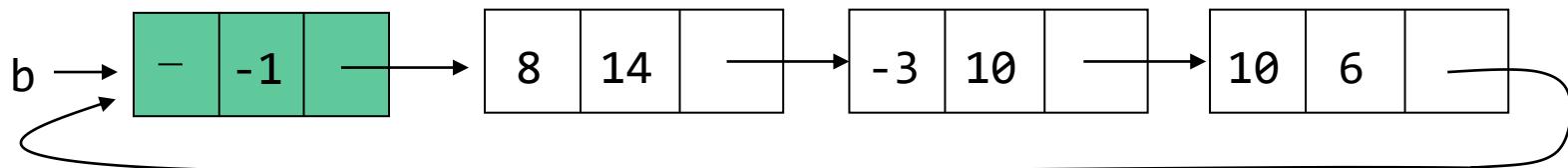


Polynomial : Circularly Linked List (2)

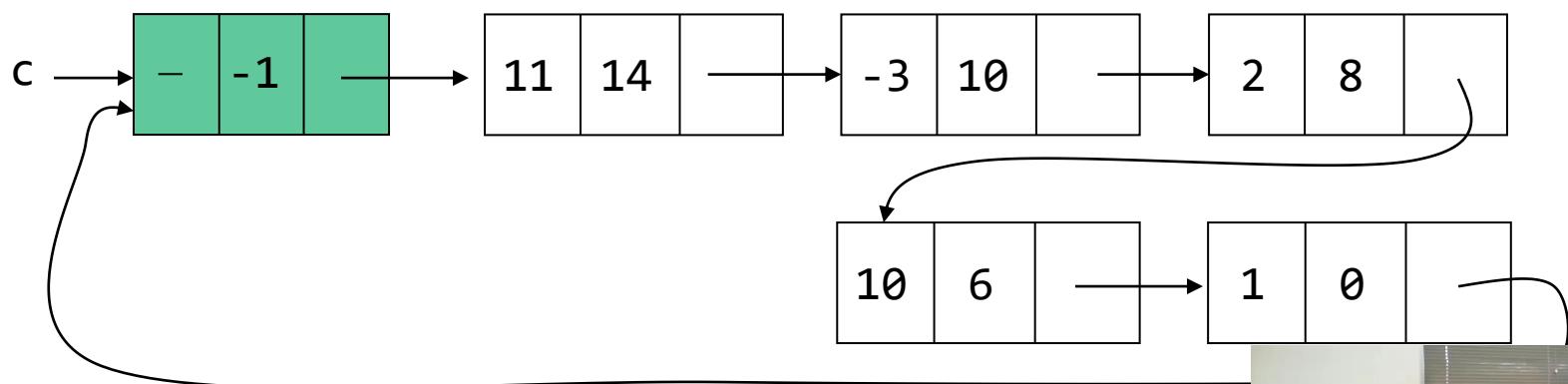
$$A(x) = 3x^{14} + 2x^8 + 1$$



$$B(x) = 8x^{14} - 3x^{10} + 10x^6$$



$$C(x) = 11x^{14} - 3x^{10} + 2x^8 + 10x^6 + 1$$



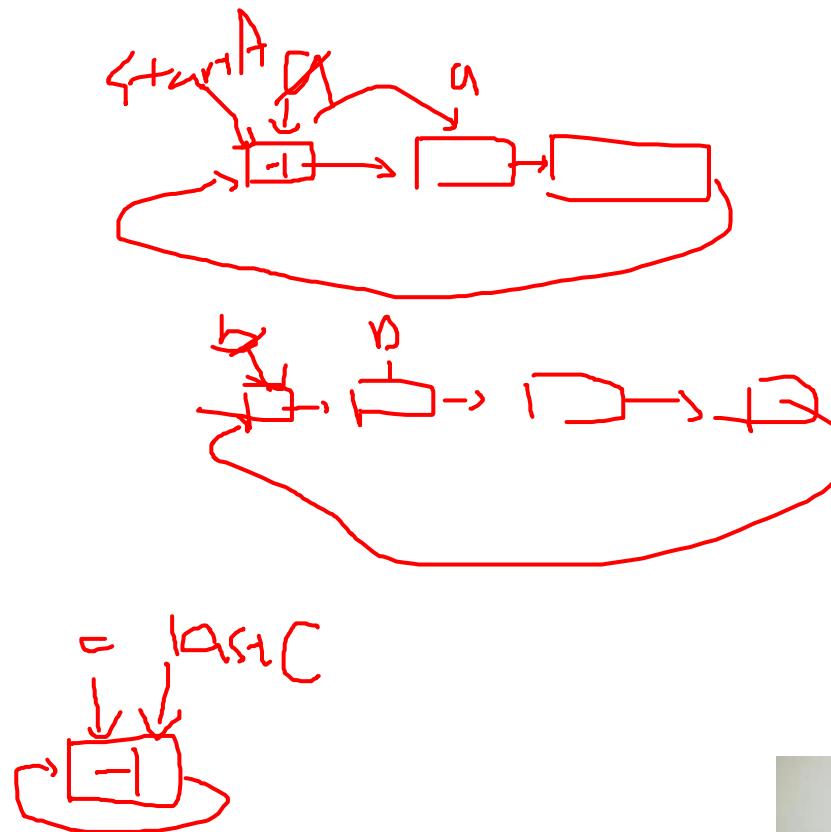
Polynomial Addition - Circular List (1)

```
polyPointer cpadd (polyPointer a, polyPointer b)
{
    polyPointer startA, c, lastC;
    int sum, done = FALSE;
    startA = a;

    {
        a = a->link;
        b = b->link;

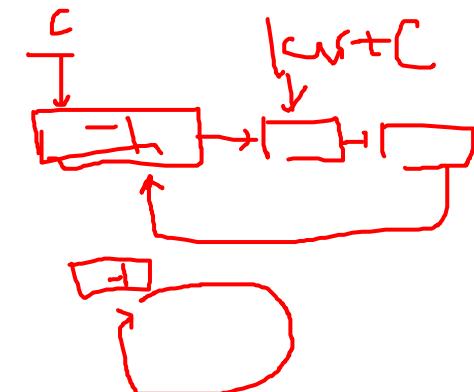
        c = getNode();
        c->expon = -1;

        lastC = c;
    }
}
```



Polynomial Addition - Circular List (2)

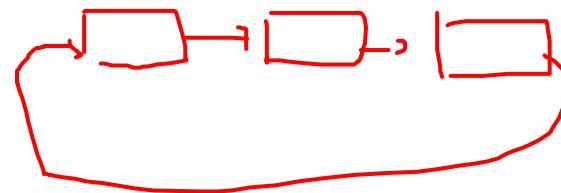
```
do {
    switch (COMPARE(a->expon, b->expon)) {
        case -1: /* a->expon < b->expon */
            attach(b->coef, b->expon, &lastC);
            b = b->link;
            break;
        case 0: /* a->expon == b->expon */
            if (startA == a) done = TRUE;
            else {
                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;
}
```



Quiz 22

Name and student ID

In the previous slide, explain how to modify cpadd() if there is no special node indicating the first node. You do not need to submit the actual code.

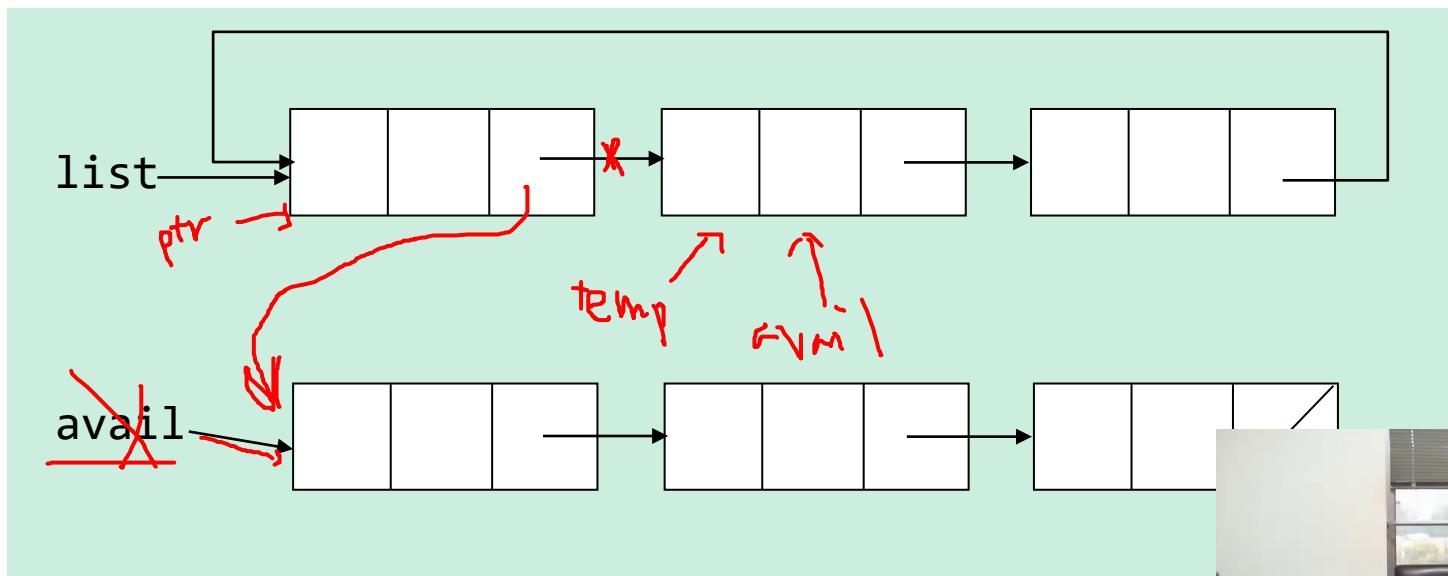


Erasing a Circular List

```
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;
    }
}
```

```
polyPointer avail;
/* a global variable that
points to the first node of
the free nodes list */
```

```
cerase(&list);
```



Additional List Operations(1)

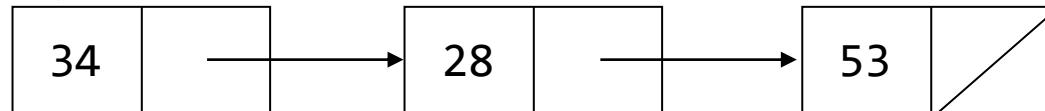
Base data structure

```
typedef struct listNode* listPointer;  
Typedef struct listNode {  
    int data;  
    listPointer link;  
};  
listPointer list_head, list_head2;
```

list_head



list_head2

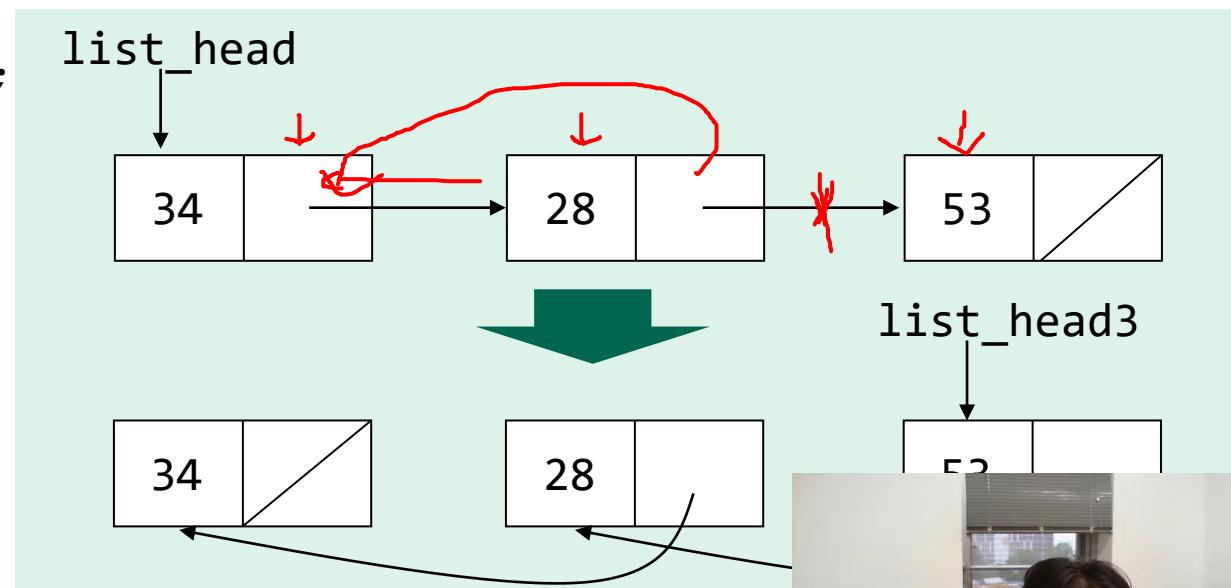


Additional Operations on Lists (2)

Inverting

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

```
listPointer list_head3 =
    invert(list_head);
```



Additional Operations on Lists (3)

Concatenating

```
listPointer concatenate (listPointer ptr1, listPointer
ptr2)
{ /* produce a new list that contains the list ptr1
followed by the list ptr2 */

listPointer temp;

if (!ptr1)    return ptr2;
if (!ptr2) return ptr1;

/* find end of first list */
for (temp = ptr1; temp->link; temp = temp->link);

temp->link = ptr2; /* link end of first to start of
second*/

return ptr1;

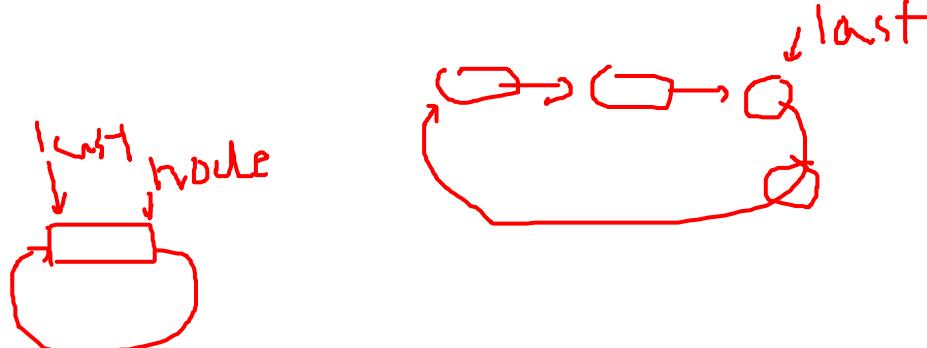
}
```



Additional Operations on Lists (4)

Inserting at the front (circular list)

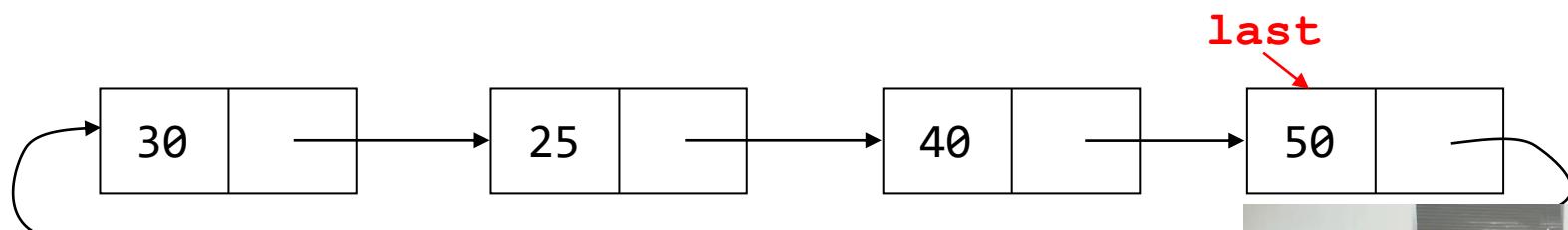
```
void insertFront (listPointer* last, listPointer node)
{
    /* insert node at the front of the circular list whose
    last node is pointed to by *last */
    if (!(*last)) {
        *last = node;
        node->link = node;
    } else {
        node->link = (*last)->link;
        (*last)->link = node;
    }
}
```



Additional Operations on Lists (5)

Finding the length (circular list)

```
int length (listPointer last)
{
    listPointer temp;
    int count = 0;
    if (last) {
        temp = last;
        do {
            count++;
            temp = temp->link;
        } while (temp != last);
    }
    return count;
}
```



Quiz 23

Name and student ID

Implement a function, invertedCopyList(), for a given list, to copy the list, invert it, and return.

```
listPointer invertedCopyList(listPointer ptr){
```

```
}
```



Equivalence Classes

Equivalence relation \equiv \subseteq

- For any polygon x , $x \equiv x$ (Chapter 4, p.174)
- For any two polygons, x and y , if $x \equiv y$ then $y \equiv x$
- For any three polygons, x, y and z , if $x \equiv y$ and $y \equiv z$, $x \equiv z$

Problem

- To partition a set S into equivalent classes such that two members x and y of S are in the same equivalence class iff $x \equiv y$
- Example

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

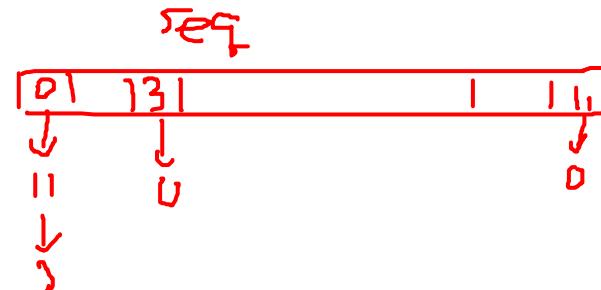
$\rightarrow \{0, 2, 4, 7, 11\}; \{1, 3, 5\}; \{6, 8, 9, 10\}$



Equivalence Relations Algorithm

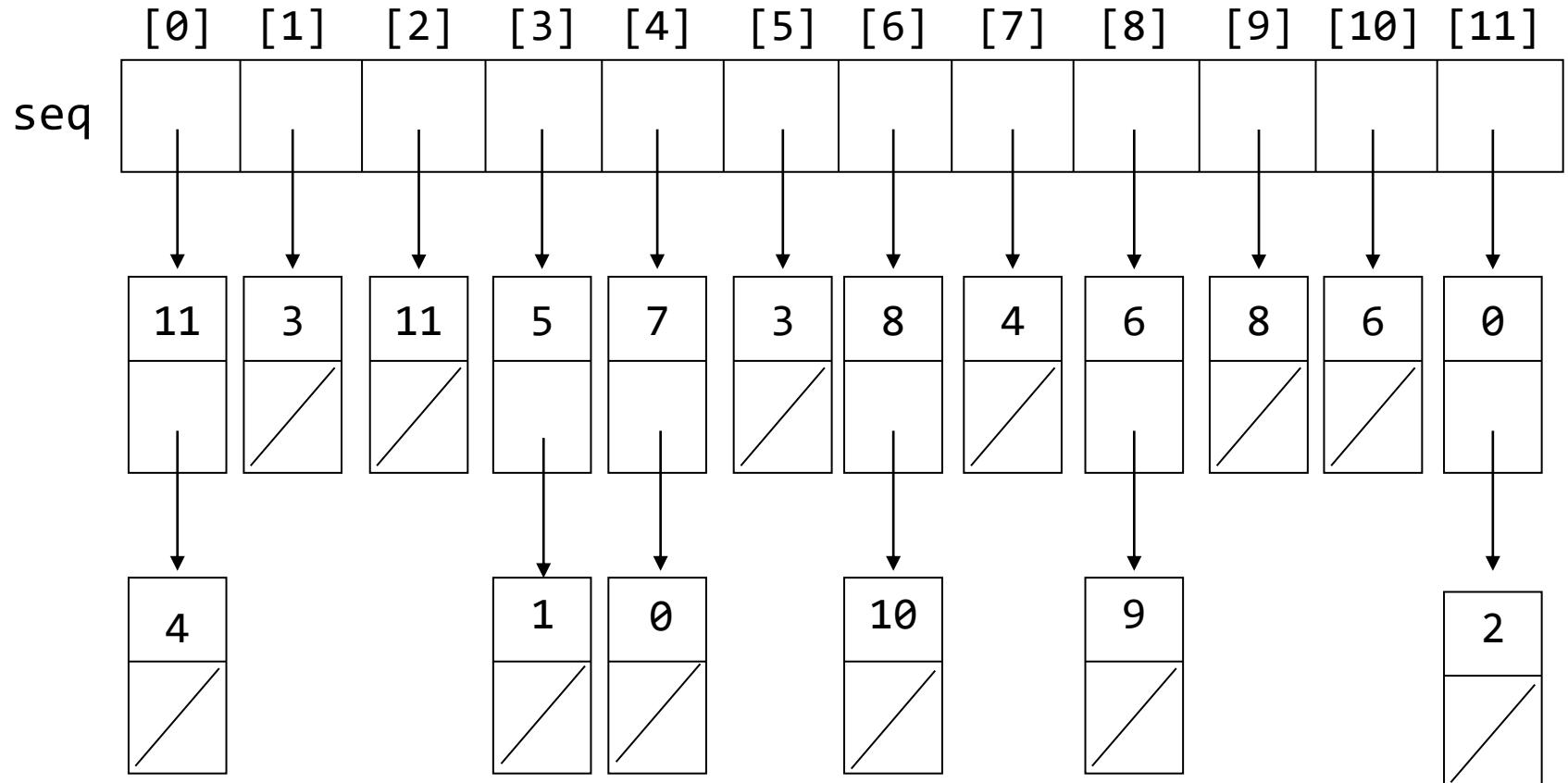
Assume that m is the number of pairs, n is the number of objects

```
void equivalence ()  
{  
    initialize seq to NULL and out to TRUE;  
    while (there are more pairs) {  
        read the next pair  $\langle i, j \rangle$ ;  
        put  $j$  on the  $\text{seq}[i]$  list;  
        put  $i$  on the  $\text{seq}[j]$  list;  
    }  
    for ( $i = 0$ ;  $i < n$ ;  $i++$ )  
        if ( $\text{out}[i]$ ) {  
             $\text{out}[i] = \text{FALSE}$ ;  
            output this equivalence class;  
        }  
}
```



Example of Equivalence Relations

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



Implementation of Equivalence Relations (1)

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

void main (void)
{
    short int out[MAX_SIZE];
    nodePointer seq[MAX_SIZE];
    nodePointer 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;
    }
}
```

$$0 \equiv 4, 3 \equiv 1, 6 \equiv 10, 7 \equiv 4, 6 \equiv 8$$

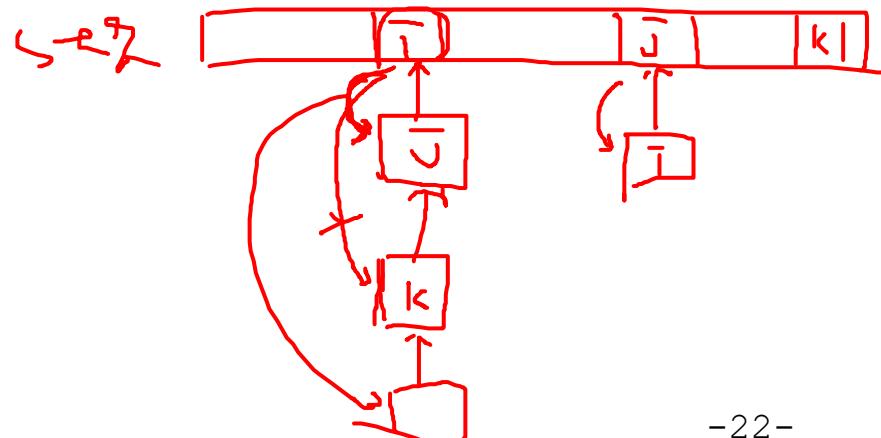
| | [0] | [1] | [2] | [3] | [4] | [5] | [6] | [7] | [8] | [9] | [10] | [11] |
|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|------|------|
| seq | | | | | | | | | | | | |
| out | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | | | |



Implementation of Equivalence Relations (2)

```
printf("Enter a pair of numbers (-1 -1 to quit): ");
scanf("%d%d", &i, &j);

/* phase 1: input the equivalence class */
while (i >= 0) {
    MALLOC(x, sizeof(*x));
    x->data = j; x->link = seq[i]; seq[i] = x;
    MALLOC(x, sizeof(*x));
    x->data = i; x->link = seq[j]; seq[j] = x;
    printf("Enter a pair of numbers (-1 -1 to quit): ");
    scanf("%d%d", &i, &j);
}
```

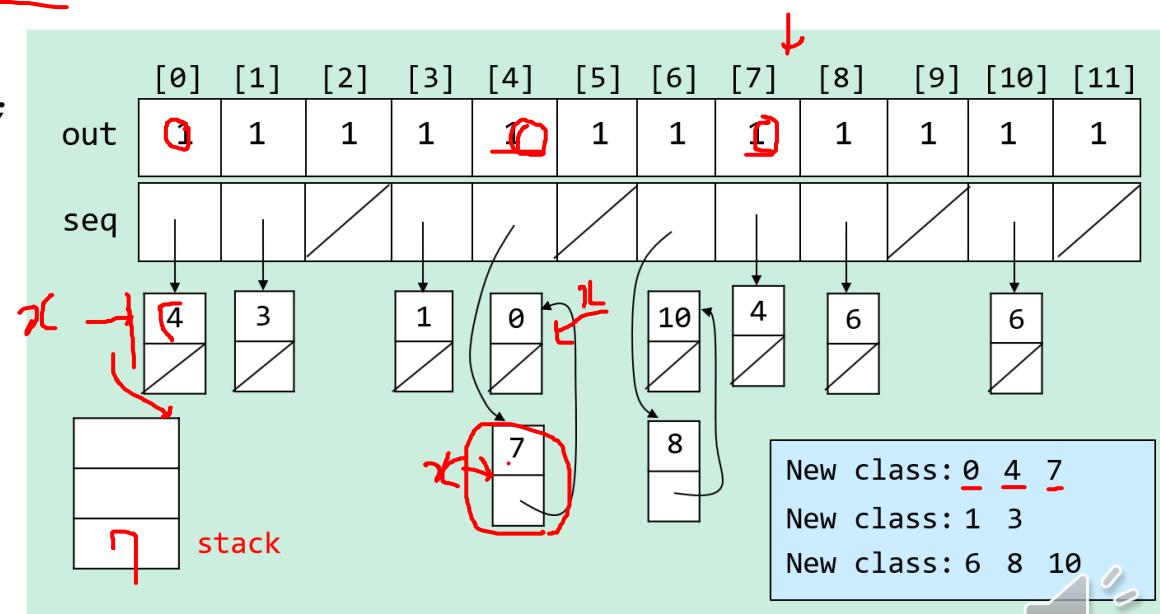


Implementation of Equivalence Relations (3)

```

/* phase 2: print out the equivalent classes */
for (i = 0; i < n; i++) {
    if (out[i]) {
        printf("\nNew class: %5d", i);
        out[i] = FALSE; x = seq[i]; top = NULL;
        for (;;) {
            while (x) {
                j = x->data; K X 0
                if (out[j]) {
                    printf("%5d", 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;
        } /* for (;;) */
    } /* for (i) */
} /* main() */

```



Quiz 24

Name and student ID

```
count = 0;
for (i = 0; i < n; i++)
    if (out[i]) {
        count++;
        printf("\nNew class: %5d", i);
        out[i] = FALSE; x = seq[i]; top = NULL;
        for (;;) {
            while (x) {
                j = x->data;
                if (out[j]) {
                    printf("%5d", 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;
        } /* for (;;) */
    } /* for (i) */
printf("%d\n", count); → ?
} /* main() */
```



Trees - Part 1

Contents

Introduction

Binary Trees

Binary Tree Traversals

Additional Binary Tree Operations

Threaded Binary Trees

Heaps

Binary Search Trees

Forests

Set Representation

Counting Binary Trees

Definition

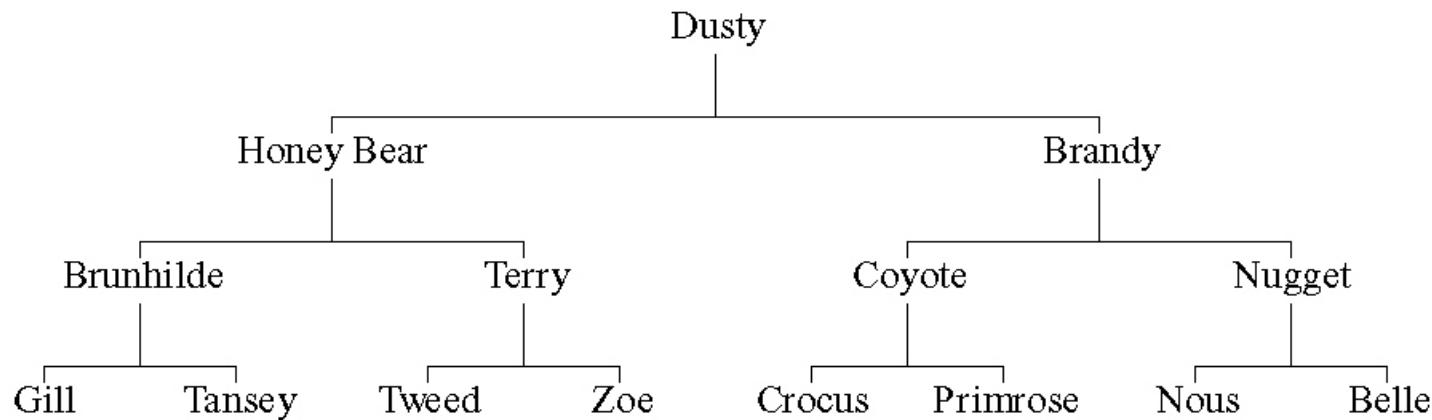
Definitions of Tree

- A finite set of one or more nodes s.t.:
 - There is a specially designated node called the **root**
 - The remaining nodes are partitioned into $n \geq 0$ disjoint sets T_1, \dots, T_n , where each of these sets is a tree.

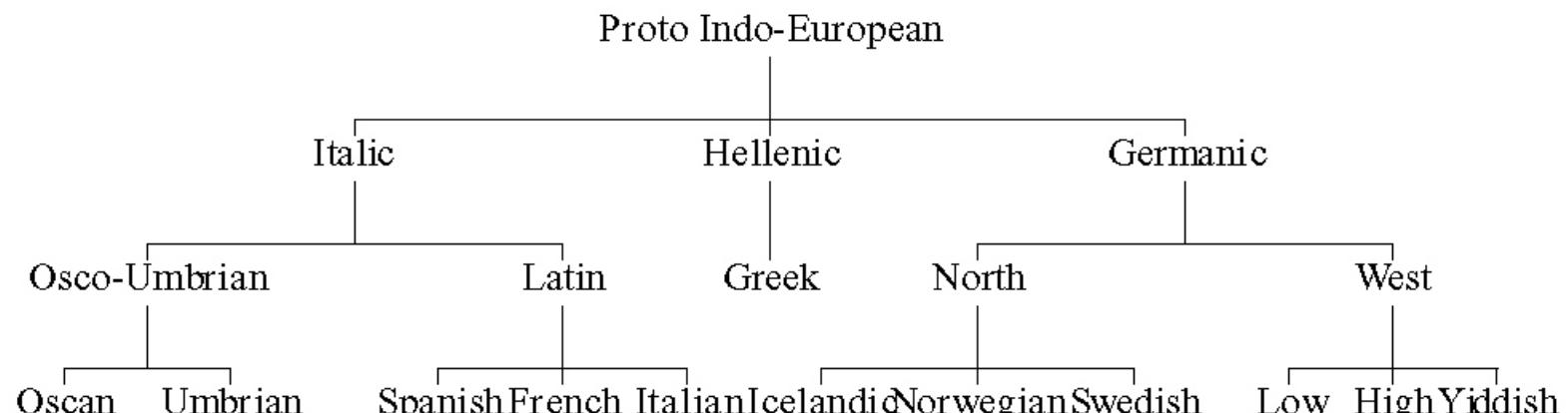
Examples of tree

- Botanic tree
- Family tree

Example of a Tree



(a) Pedigree



(b) Lineal

Terminology (1)

Subtree

- A tree T_1 is called subtree of a tree T_2 if every node of T_1 is also a node of T_2

Size of a tree

(Chapter 5, p.193)

- The number of the nodes

Degree of a node

- The number of subtrees
- Leaf (terminal node) - a node with degree zero

Degree of a tree

- Maximum degree of the nodes in the tree

Terminology (2)

Height (depth) of a tree

- Maximum level(=depth) of any node in the tree
- The length of the longest root-to-leaf path

Width of a tree

- The size of the largest level

Terminology (3)

Parent

Children

Siblings

Grand parent

Grand children

Ancestors

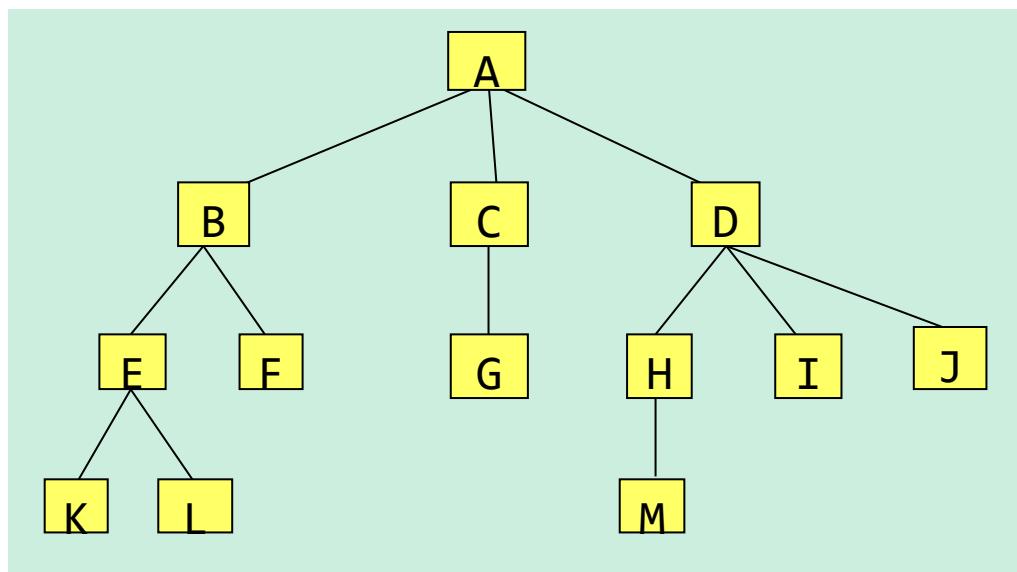
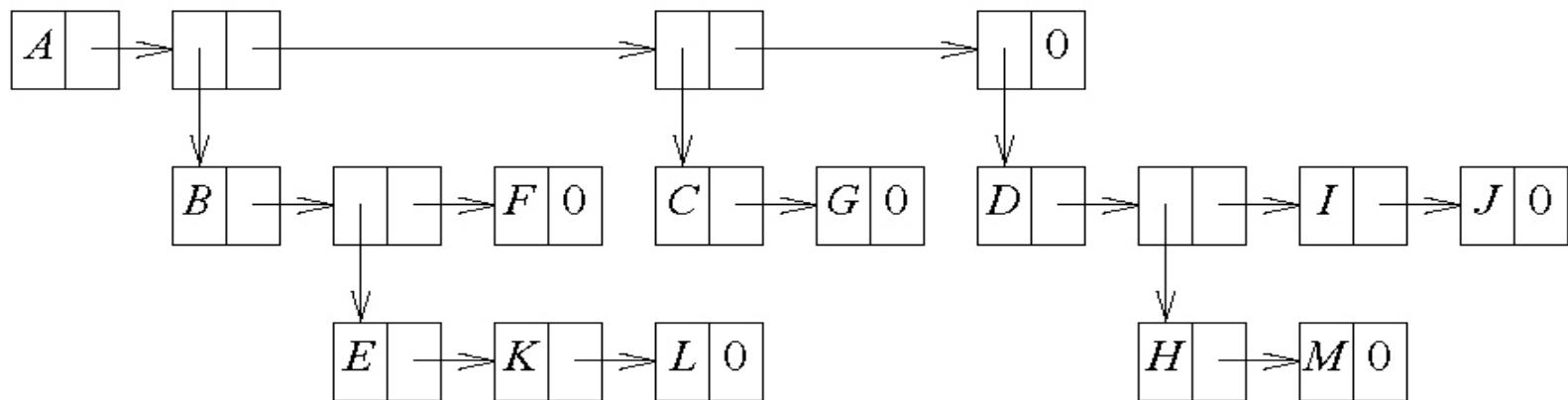
- All the nodes along the path from the root to the node

Descendants

- All the nodes that are in its subtrees

List Representation

(A (B (E (K, L), F), C (G), D (H (M), I, J)))



Quiz 25

Name and student ID

Draw the tree given as the list (a(b(e(k(l),f),c(g,d(h(m,i(j))))))).

K-ary Tree Representation

Node structure for a tree of degree k

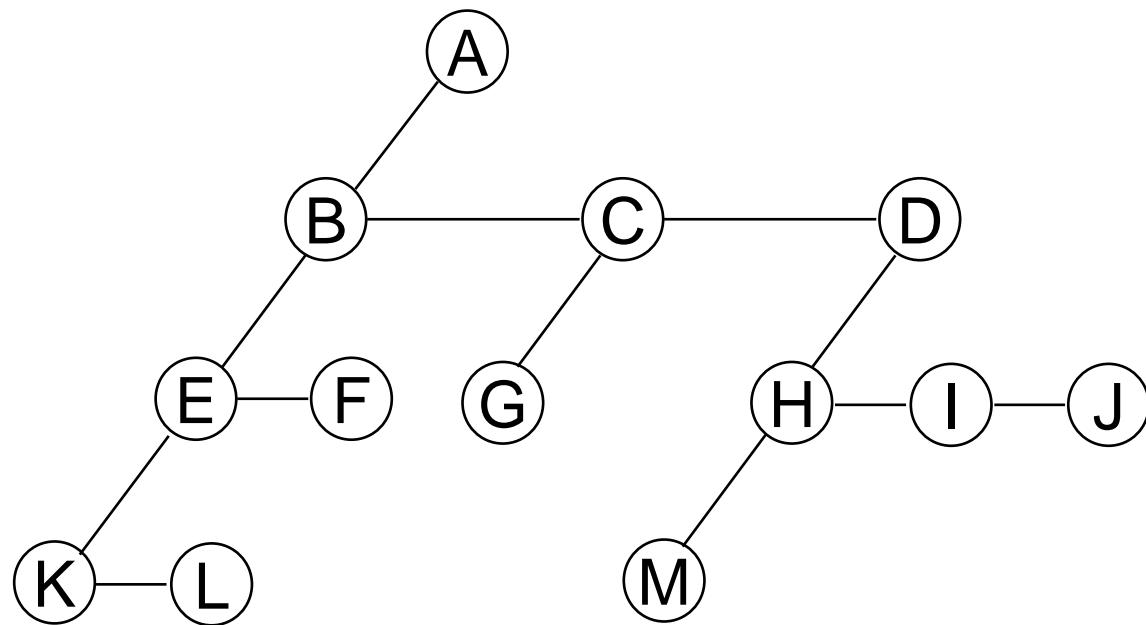
| | | | | | |
|------|--------------------|--------------------|--------------------|-----|--------------------|
| data | child ₁ | child ₂ | child ₃ | ... | child _k |
|------|--------------------|--------------------|--------------------|-----|--------------------|

Lemma 5.1 If T is k -ary tree with n nodes, each having a fixed size, then $n \cdot (k-1) + 1$ of the $n \cdot k$ child fields are 0, $n \geq 1$

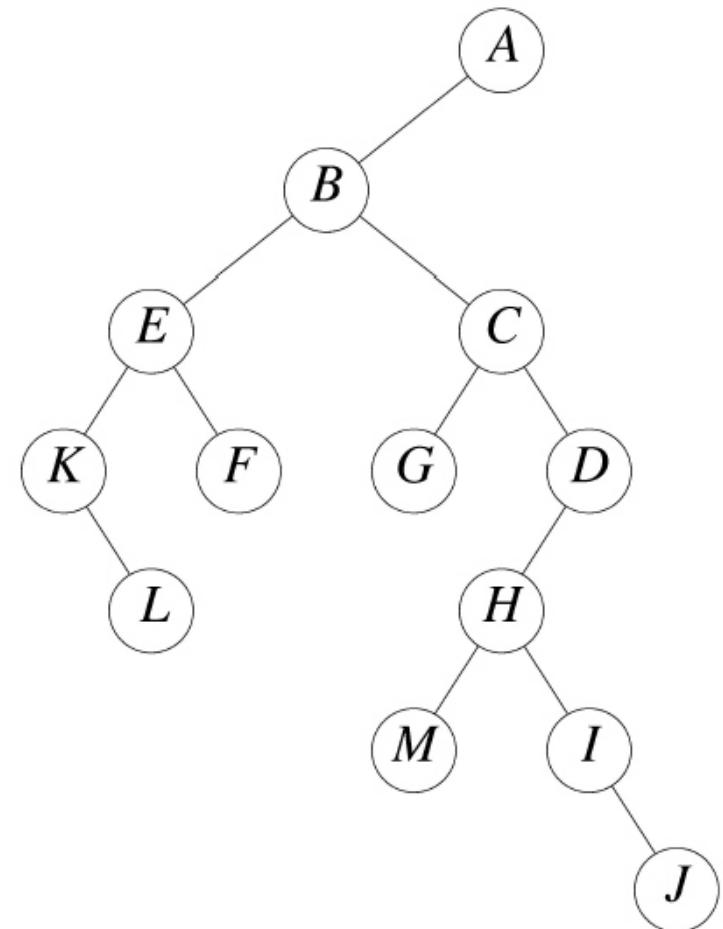
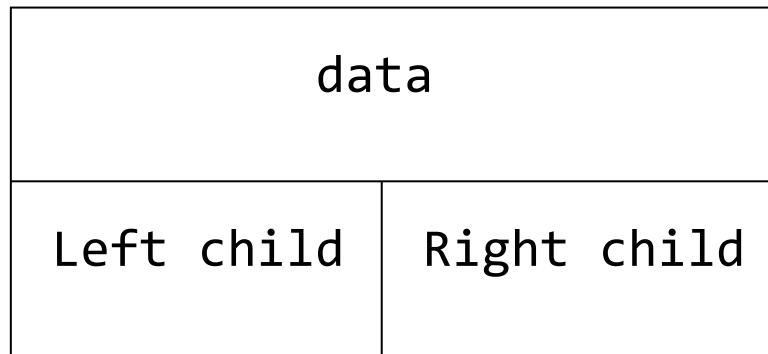
Chapter 5, p.195

Left Child-Right Sibling Representation

| Data | |
|------------|---------------|
| Left child | Right sibling |



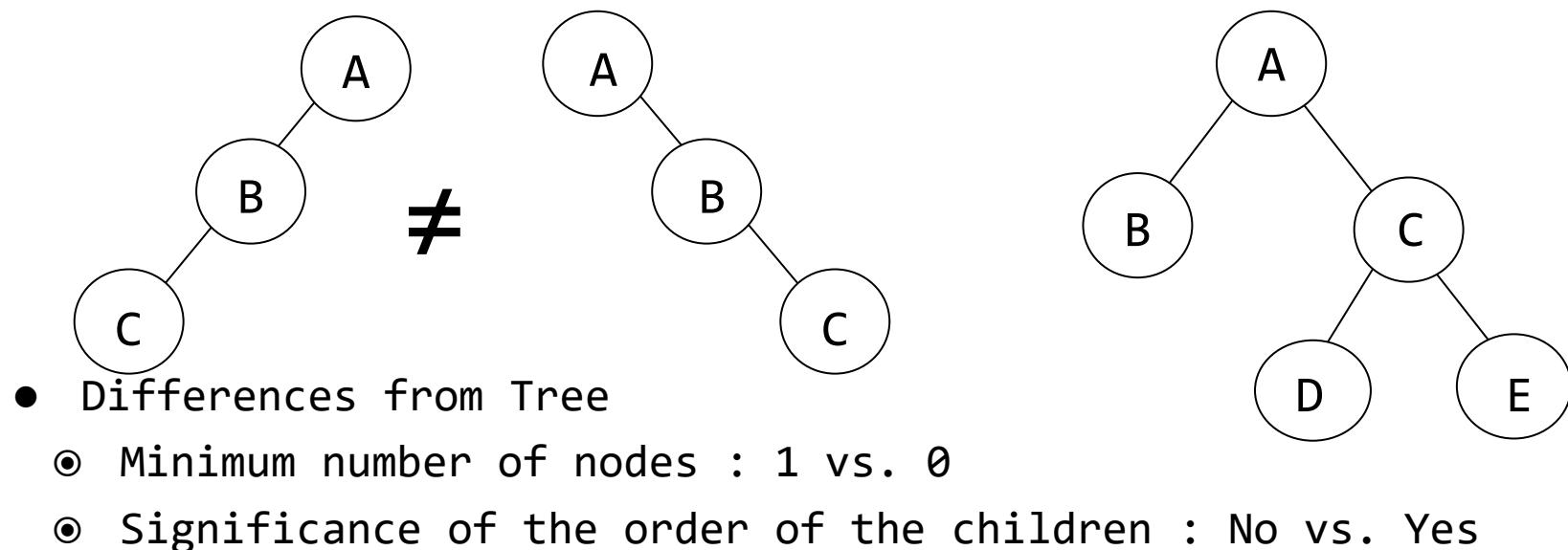
Left Child-Right Child Representation



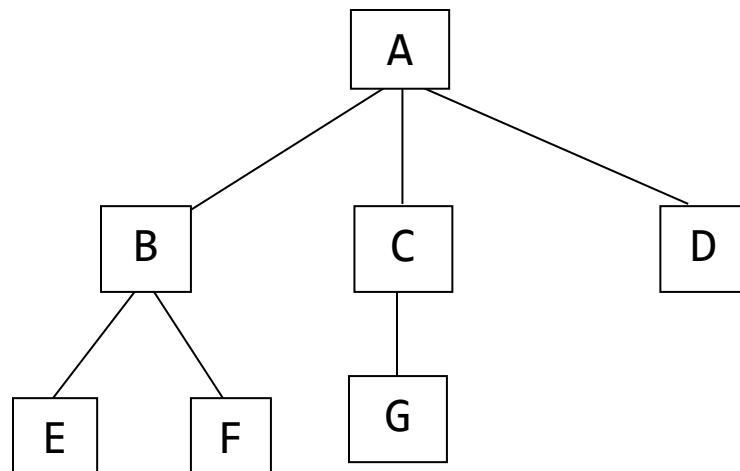
Binary Tree

Definition 1)

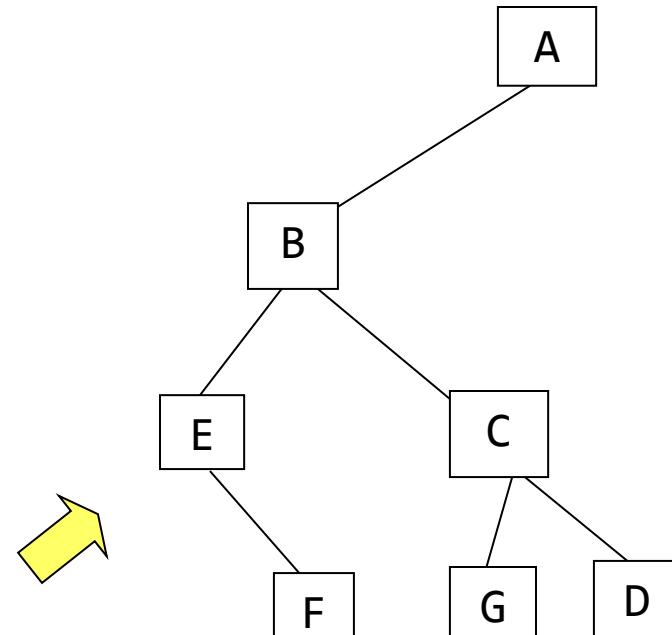
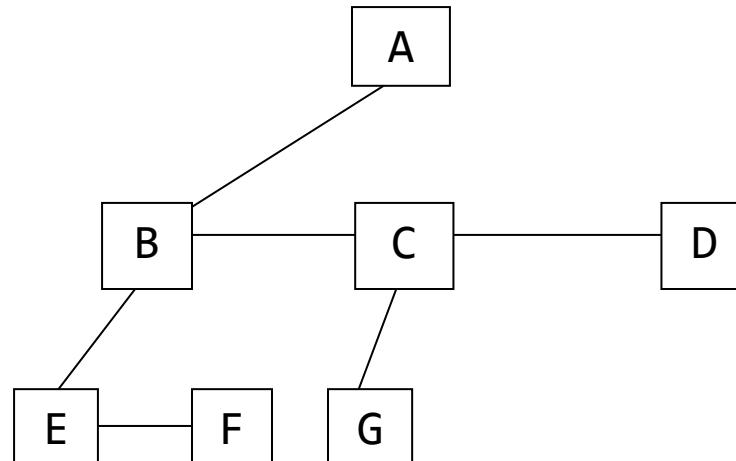
- Finite set of nodes that is either empty or consists of a root and two disjoint binary trees called the left subtree and the right subtree



Binary Tree Representation of Trees



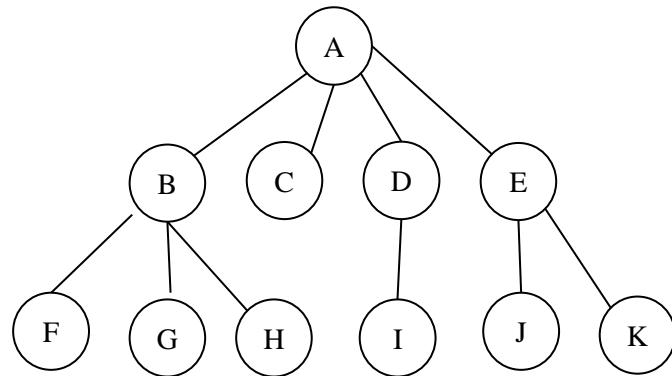
Left child-right sibling representation



Quiz 26

Name and student ID

Convert the tree in the figure into a binary tree.



Abstract Data Type

ADT *Binary_Tree* (abbreviated BinTree) is

objects: a finite set of nodes either empty or consisting of a root node, left *Binary_Tree*, and right *Binary_Tree*

functions:

for all $bt, bt1, bt2 \in \text{BinTree}$, $\text{item} \in \text{element}$

$\text{BinTree Create}() ::= \dots$

$\text{Boolean IsEmpty}(bt) ::= \dots$

$\text{BinTree MakeBT}(bt1, \text{item}, bt2) ::= \dots$

$\text{BinTree Lchild}(bt) ::= \dots$

$\text{element Data}(bt) ::= \dots$

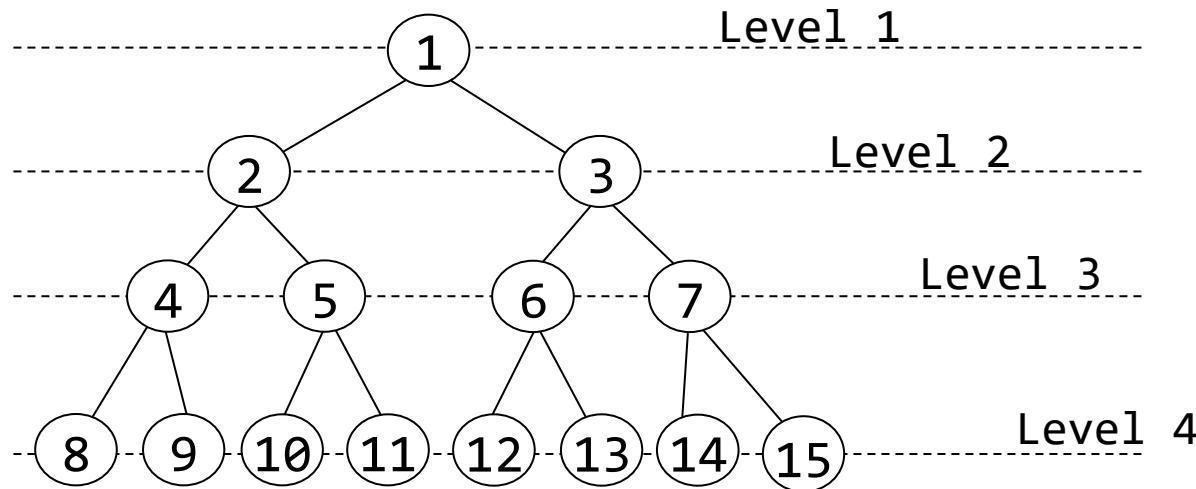
$\text{BinTree Rchild}(bt) ::= \dots$

Properties of Binary Tree (1)

Lemma 5.2 [Maximum number of nodes]

The maximum number of nodes on level i of a binary tree is 2^{i-1} , $i \geq 1$.

The maximum number of nodes in a binary tree of depth k is $2^k - 1$, $k \geq 1$.



Properties of Binary Tree (2)

Lemma 5.3 [Relation between number of leaf nodes and nodes of degree 2]

For any nonempty binary tree T , if n_0 is the number of leaf nodes and n_2 the number of nodes of degree 2,

then $n_0 = n_2 + 1$

(proof) $n = n_0 + n_1 + n_2$

B : the number of branches $\rightarrow n = B + 1$

$$B = n_1 + n_2 \times 2$$

$$\rightarrow n = n_0 + n_2 + n_1 = n_1 + n_2 \times 2 + 1$$

$$\rightarrow n_0 + n_2 + n_1 = n_1 + n_2 + n_2 + 1$$

$$\rightarrow n_0 = n_2 + 1$$

Definitions

Definition 2)

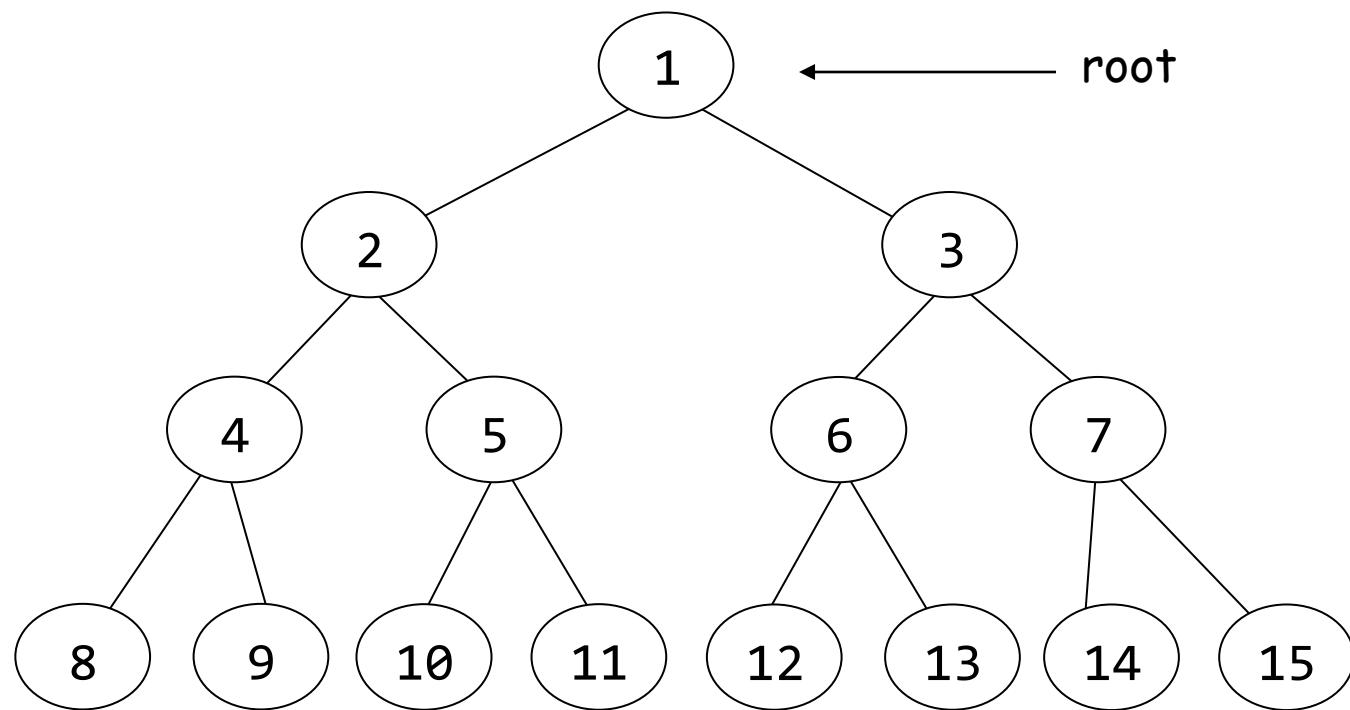
A **full binary tree** of depth k is a binary tree of depth k having $2^k - 1$ nodes, $k \geq 0$. (Assume that the depth of the root node is 1)

A nonempty tree is said to be full if all of its leaves are at the same level and all of its nodes have the same degree

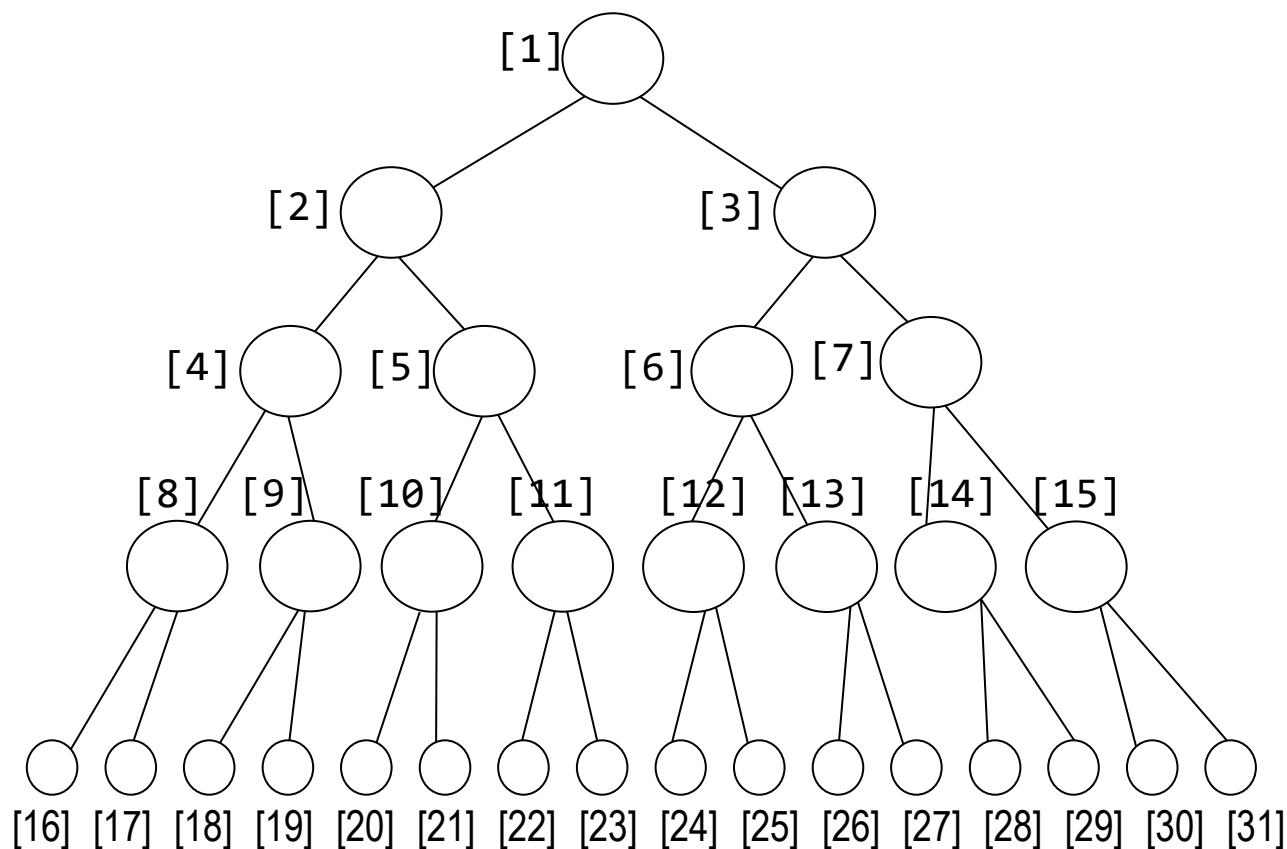
Definition 3)

A **binary tree** with n nodes and depth k is **complete** iff its nodes correspond to the nodes numbered from 1 to n in the full binary tree of depth k .

An Example of Full Binary Tree



Array Representation of Binary Trees (1)



Array Representation of Binary Trees (2)

Lemma 5.4 : If a complete binary tree with n nodes (depth = $\lfloor \log_2 n + 1 \rfloor$, Assume that the depth of root node is 1) is represented sequentially, then for any node with index i , $1 \leq i \leq n$, we have:

1) $\text{parent}(i)$ is at $\lfloor i/2 \rfloor$ if $i \neq 1$,

If $i=1$, it is at the root and has no parent

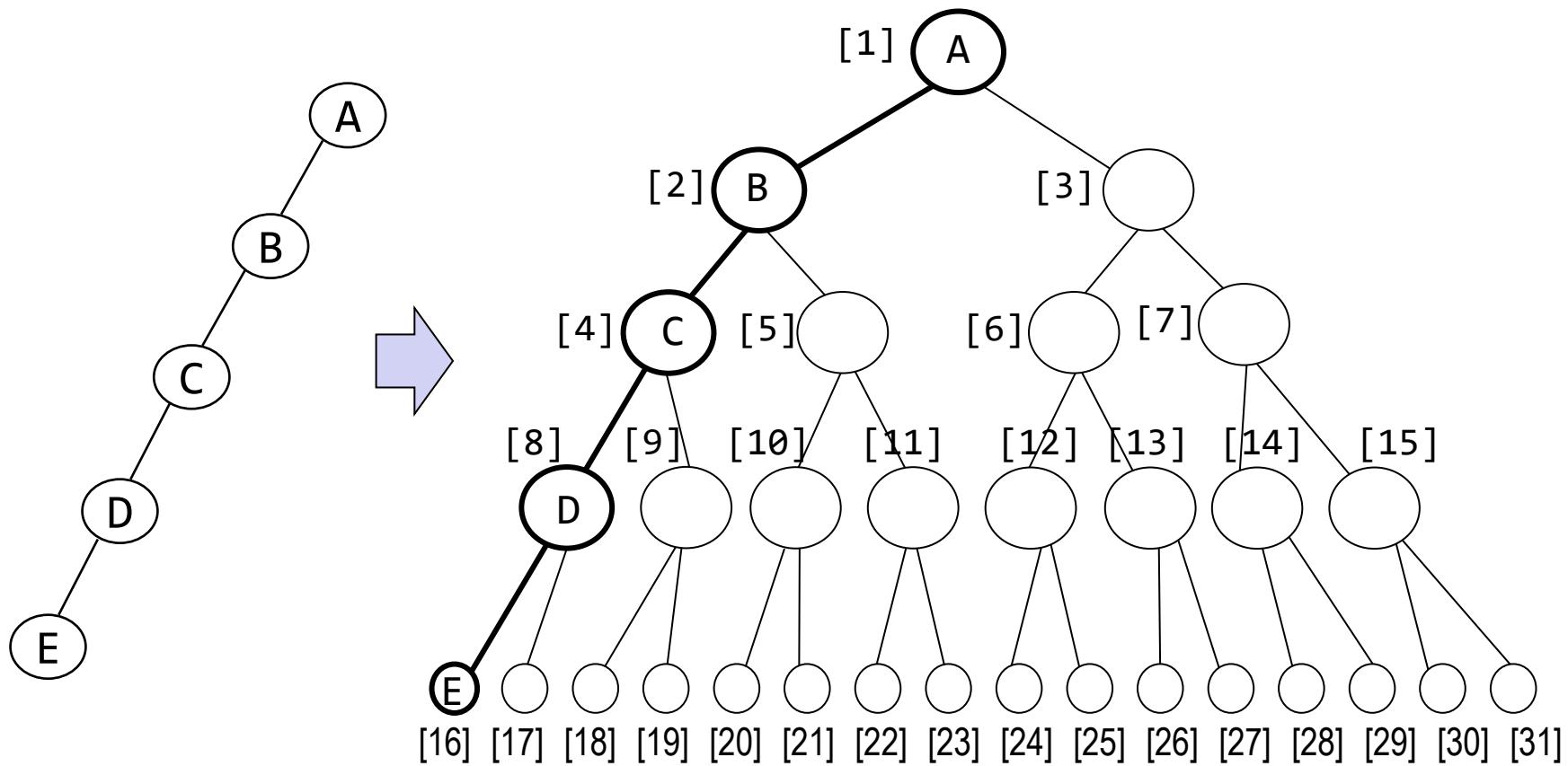
2) $\text{leftChild}(i)$ is at $2i$ if $2i \leq n$.

If $2i > n$, then i has no left child

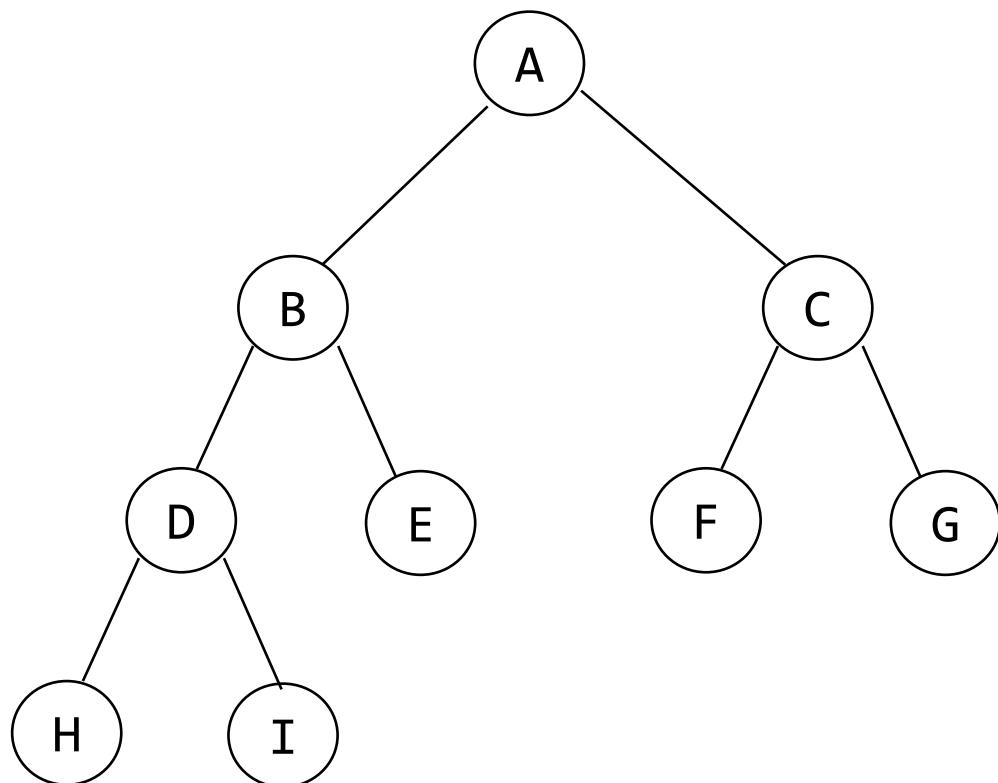
3) $\text{rightChild}(i)$ is at $2i+1$, if $2i+1 \leq n$.

If $2i+1 > n$, then i has no right child

Array Representation of Binary Trees (3)



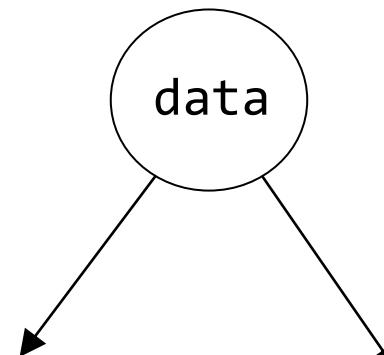
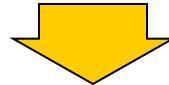
Array Representation of Binary Trees (4)



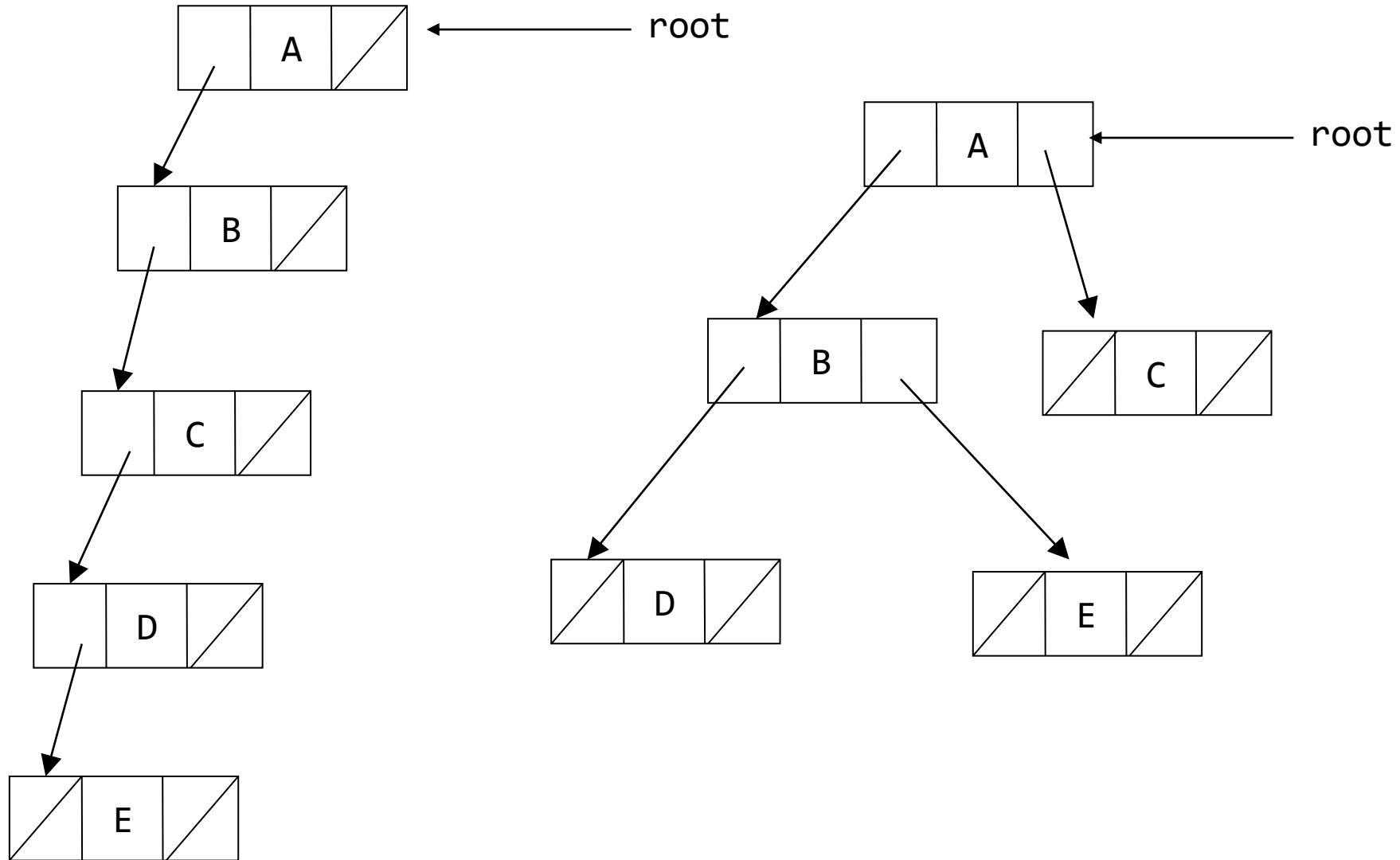
| | |
|-----|---|
| [0] | - |
| [1] | A |
| [2] | B |
| [3] | C |
| [4] | D |
| [5] | E |
| [6] | F |
| [7] | G |
| [8] | H |
| [9] | I |

Linked Representation (1)

```
typedef struct node* treePointer;  
typedef struct node {  
    int data;  
    treePointer leftChild, rightChild;  
};
```



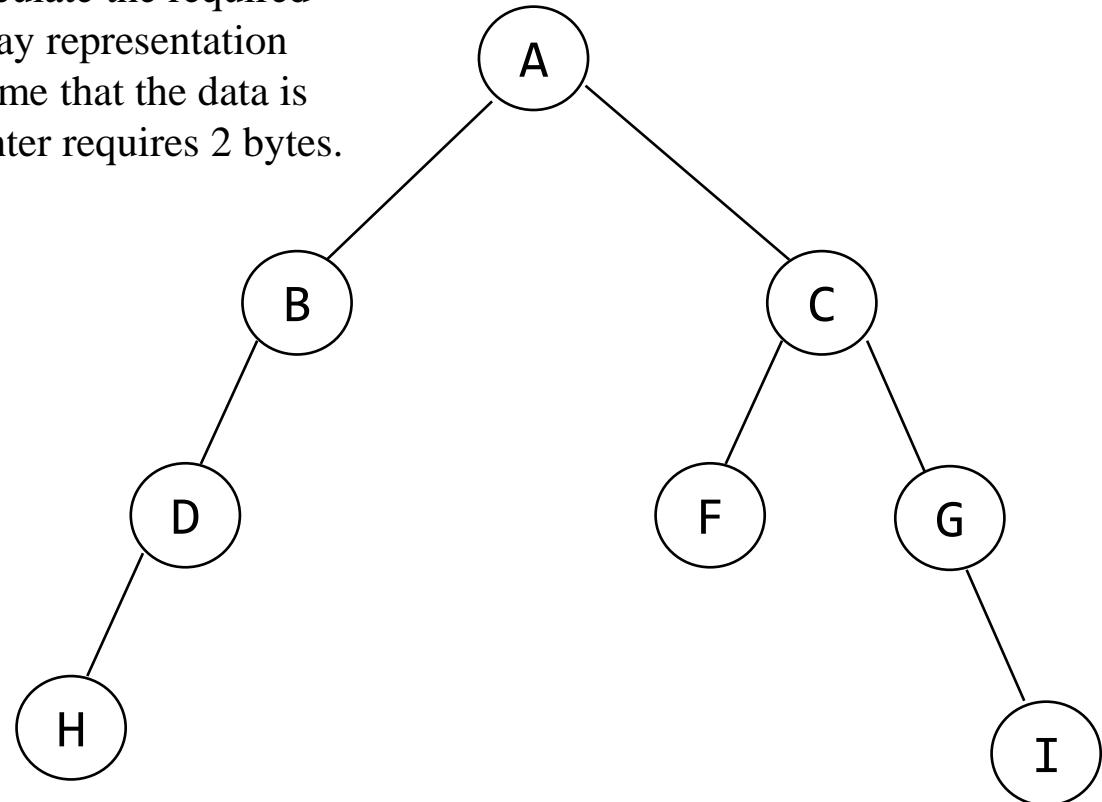
Linked Representation (2)



Quiz 27

Name and student ID

Consider the tree in the figure. Calculate the required memory space when we use (a) array representation and (b) linked representation. Assume that the data is integer (4 bytes), and the node pointer requires 2 bytes.

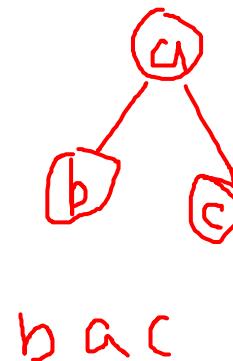


Binary Tree Traversal

- L : moving left
- V : visiting the node
- R : moving right

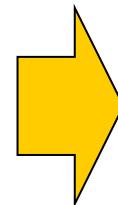
6 possible combinations of traversal

- : LVR (inorder, “depth-first”)
- : LRV (postorder)
- : VLR (preorder)
- : VRL
- : RVL
- : RLV



Arithmetic Expression

A / B * C * D + E

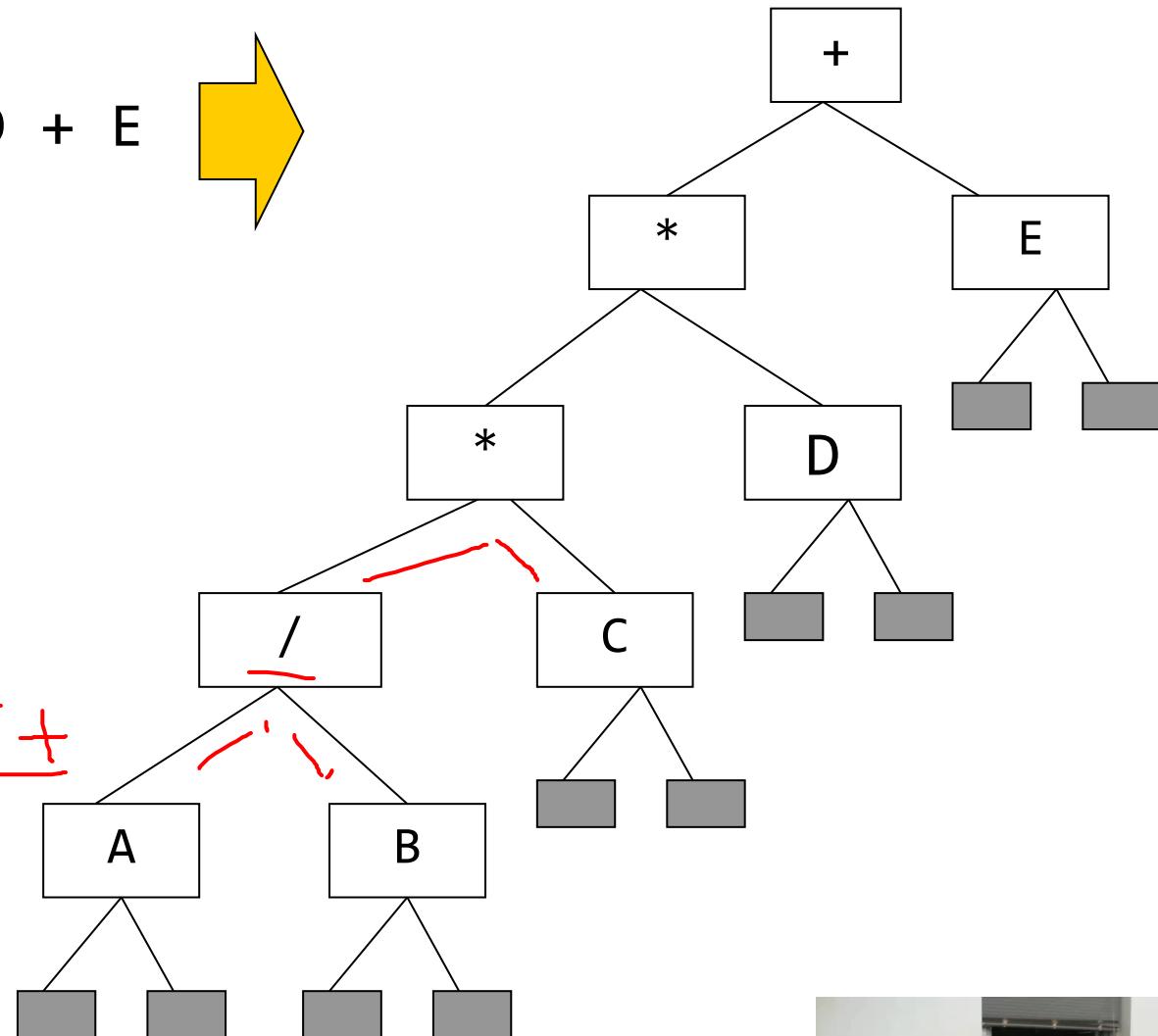


L V R

A / B

LRV

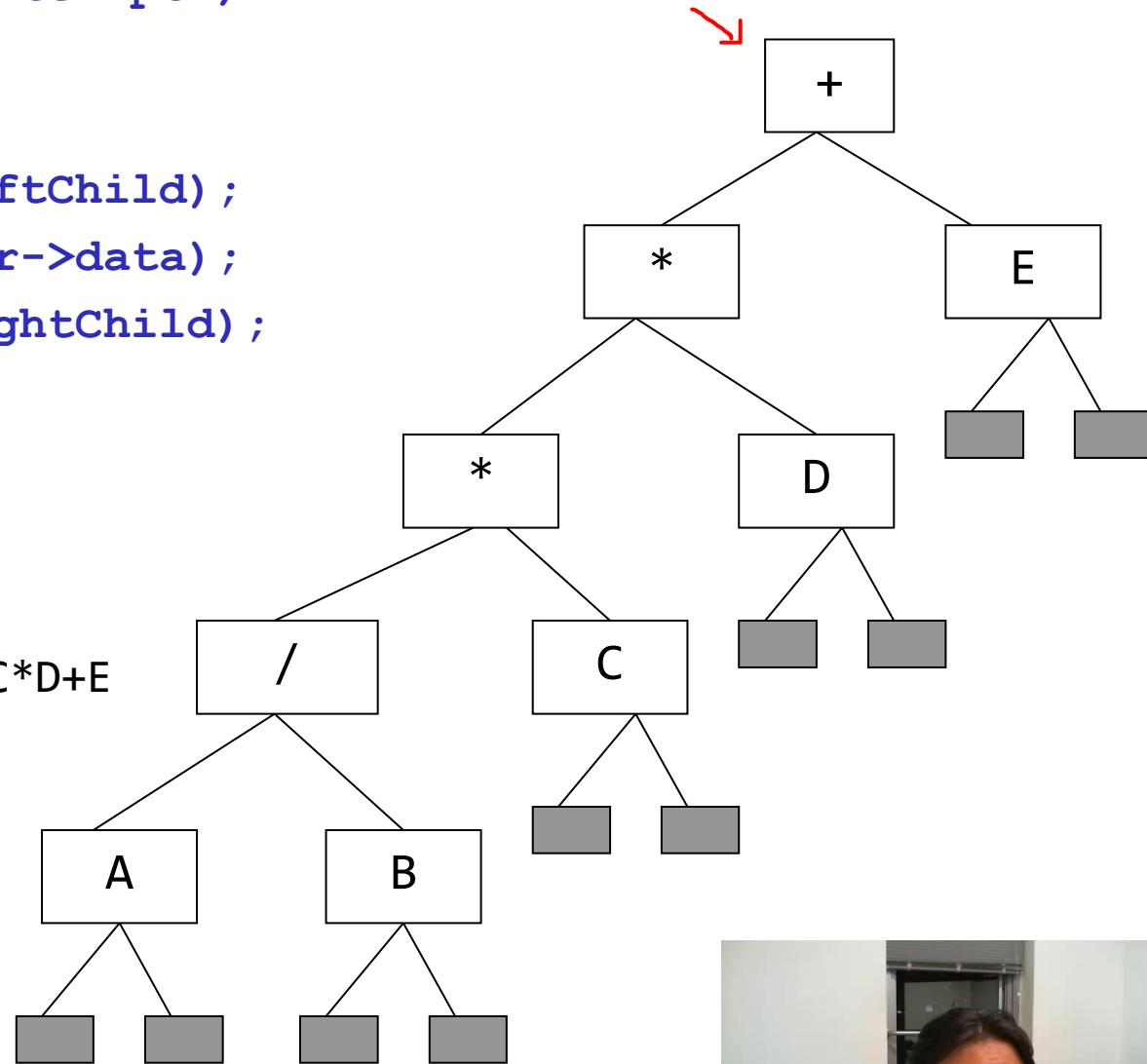
A B / C * D * E +



Inorder Traversal

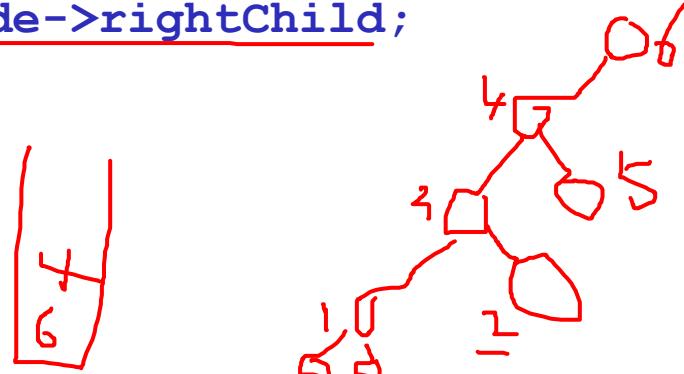
```
void inorder(treePointer ptr)
{
    if (ptr) {
        inorder(ptr->leftChild);
        printf("%c", ptr->data);
        inorder(ptr->rightChild);
    }
    return;
}
```

Inorder traversal : A/B*C*D+E



Iterative Inorder Traversal

```
void iterInorder(treePointer node)
{
    int top = -1;      /* initialize the stack */
    treePointer stack[MAX_STACK_SIZE];
    for (;;) {
        for (; node; node = node->leftChild)
            push(node); /* add to the stack */
        node = pop(); /* delete from the stack */
        if (!node) break; /* empty stack */
        printf("%c", node->data);
        node = node->rightChild;
    }
    return;
}
```



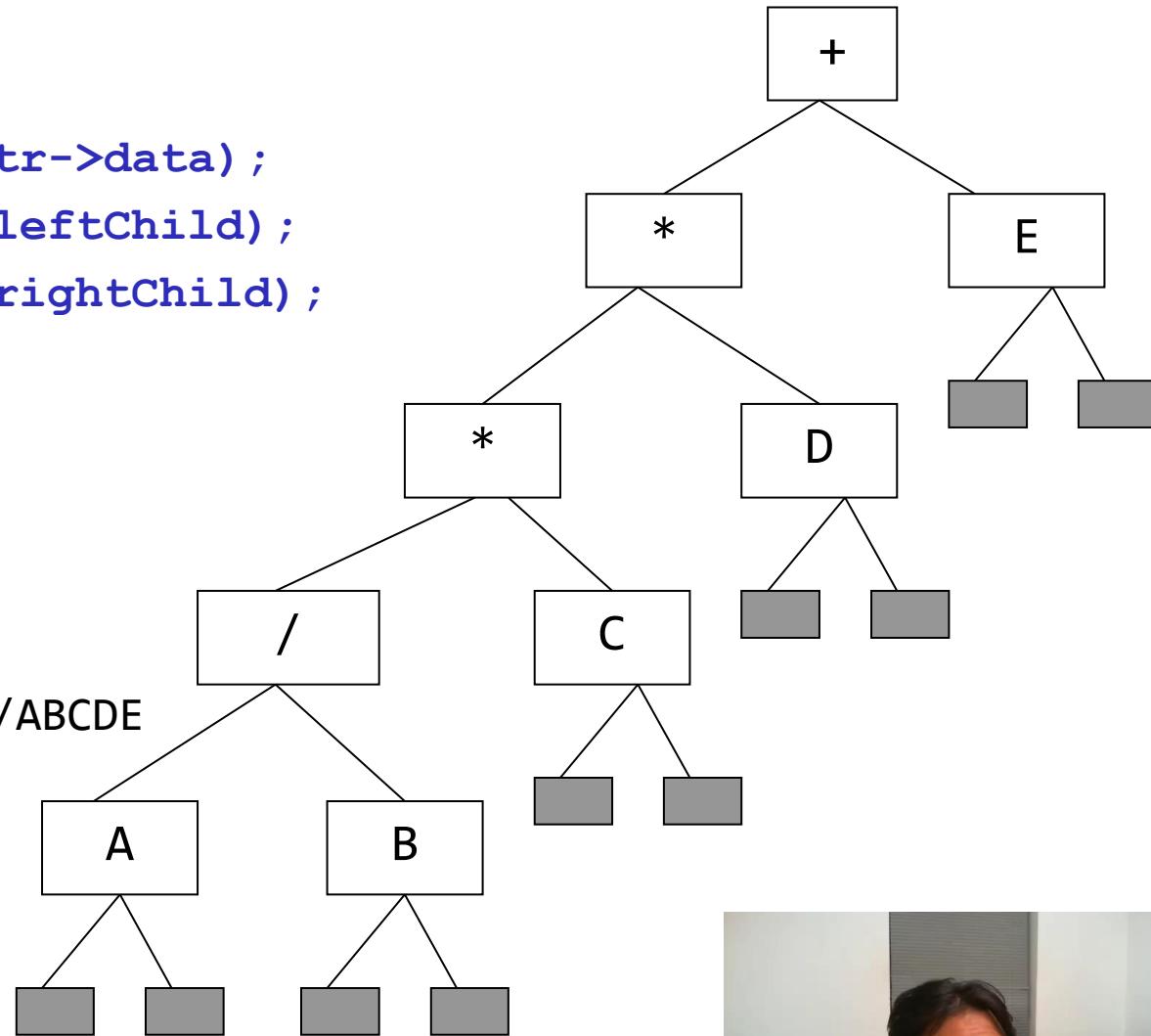
1 3 2



Preorder Traversal

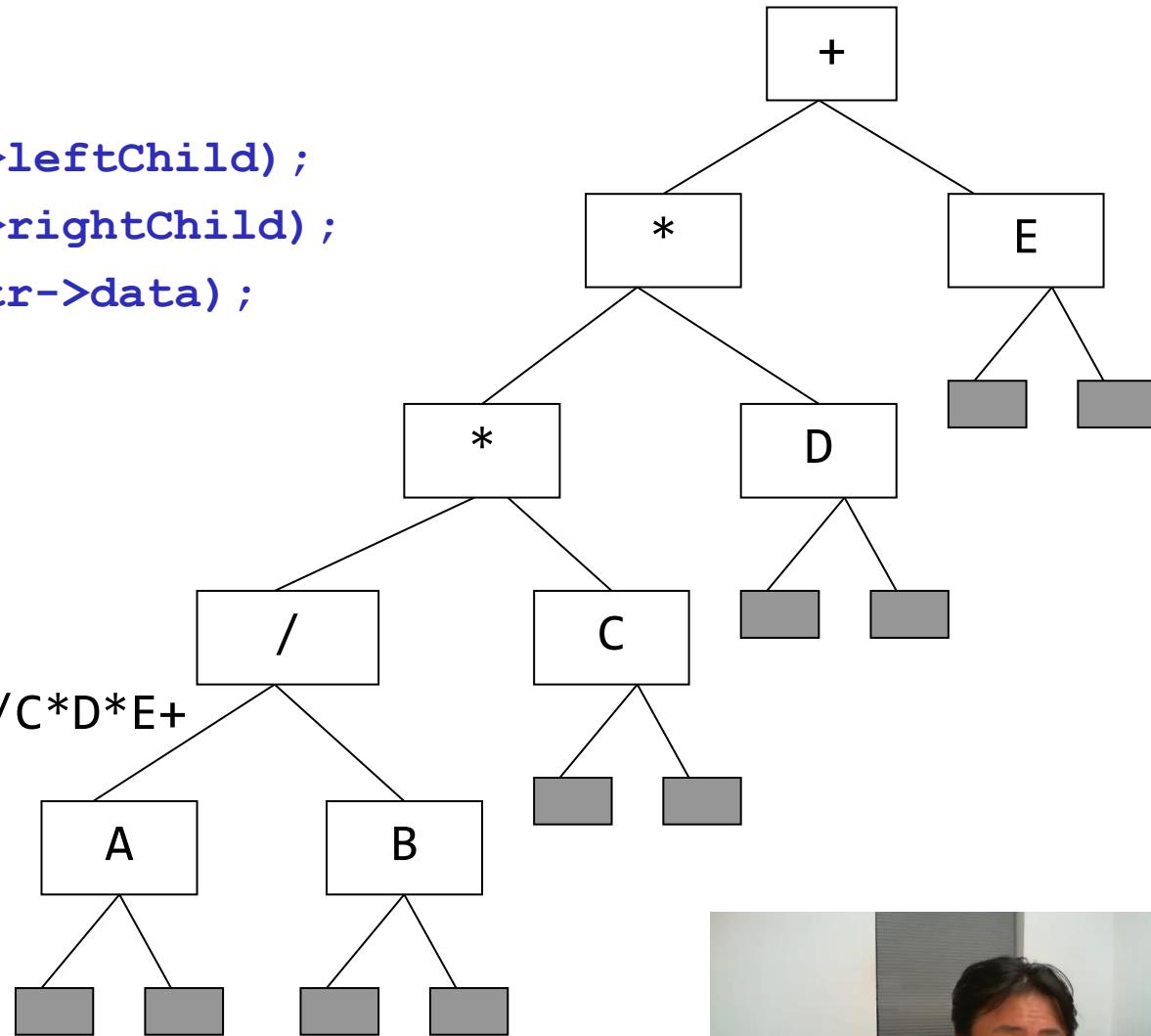
```
void preorder(treePointer ptr)
{
    if (ptr) {
        printf("%c", ptr->data);
        preorder(ptr->leftChild);
        preorder(ptr->rightChild);
    }
    return;
}
```

Preorder traversal : +**/ABCDE



Postorder Traversal

```
void postorder(treePointer ptr)
{
    if (ptr) {
        postorder(ptr->leftChild);
        postorder(ptr->rightChild);
        printf("%c", ptr->data);
    }
    return;
}
```



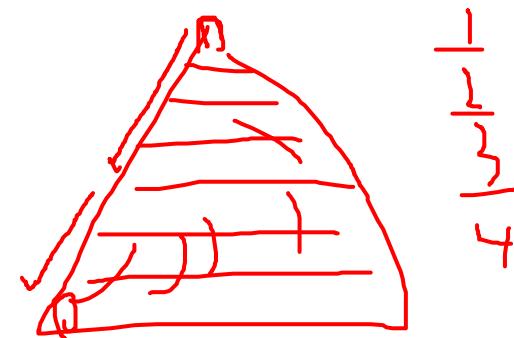
Postorder traversal: AB/C*D*E+



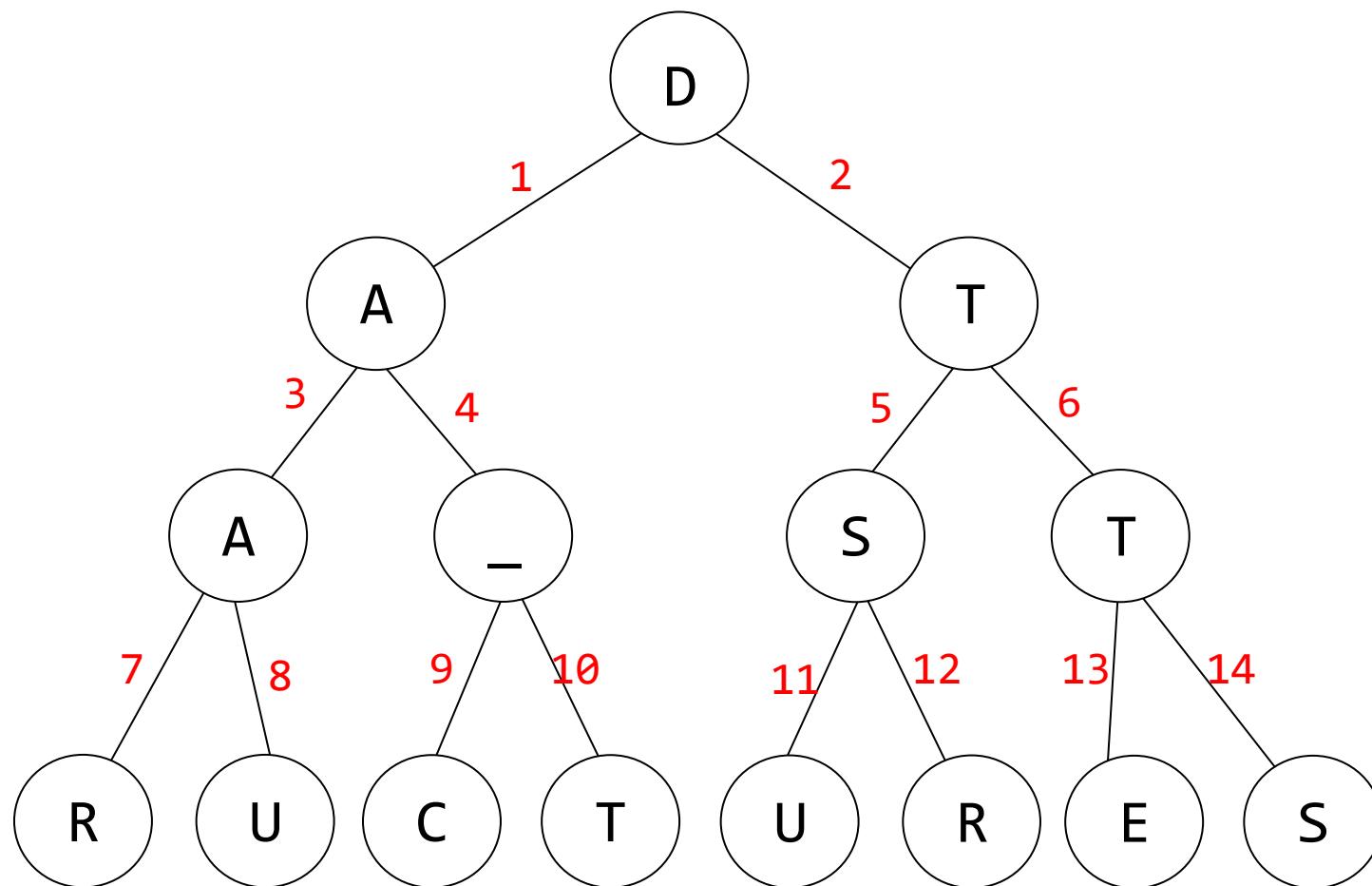
Level-order Traversal ("Breadth-first") (1)

```
void levelOrder(treePointer ptr)
{
    treePointer queue[MAX_QUEUE_SIZE];
    if (!ptr) return; /* empty tree */
    addq(ptr);
    for (;;) {
        ptr = deleteq();
        if (ptr) {
            printf("%c", ptr->data);
            if (ptr->leftChild)
                addq(ptr->leftChild);
            if (ptr->rightChild)
                addq(ptr->rightChild);
        } /* if */
        else break;
    } /* for (;;) */
}
```

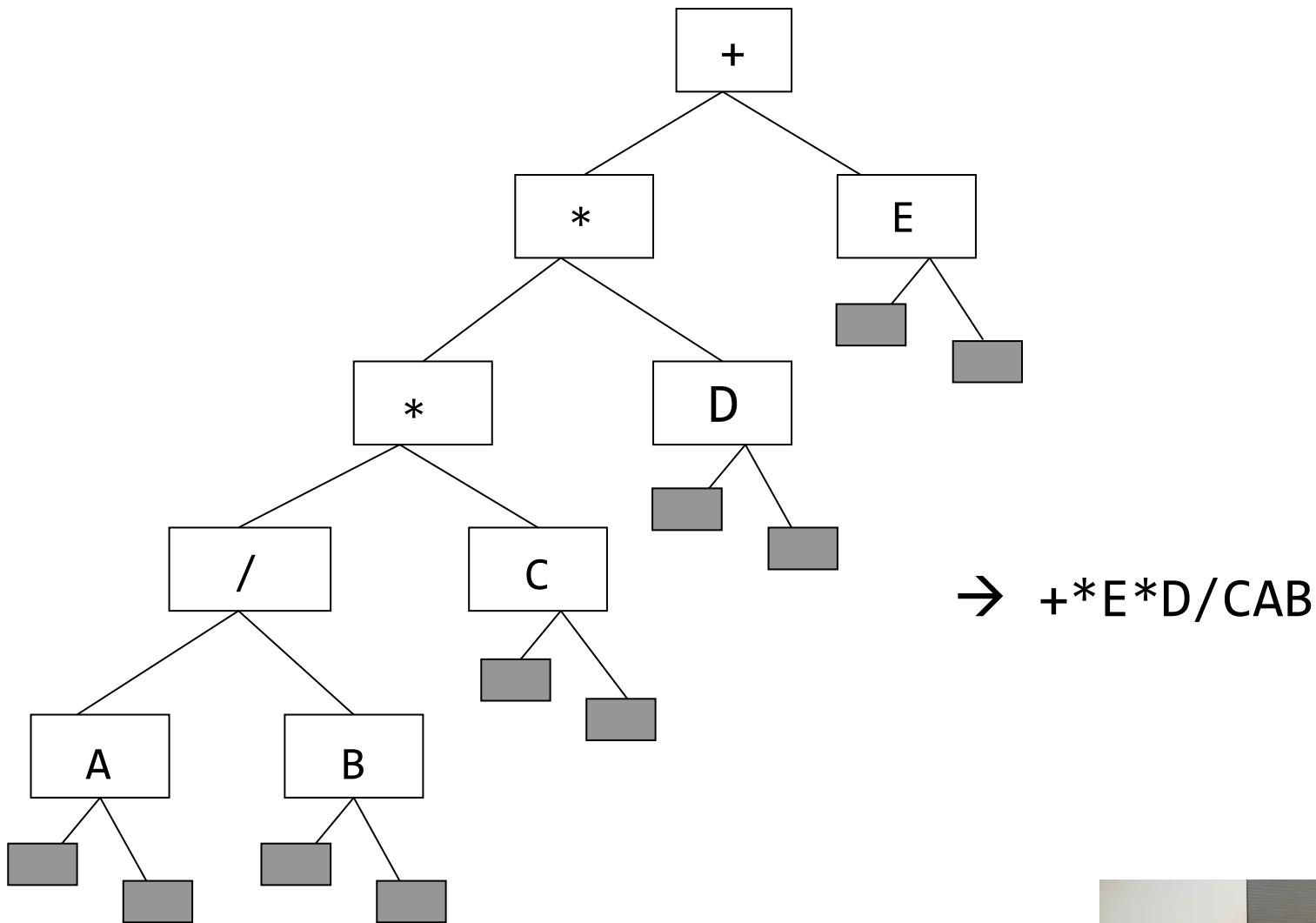
Depth - first



Level-order Traversal (“Breadth-first”) (2)



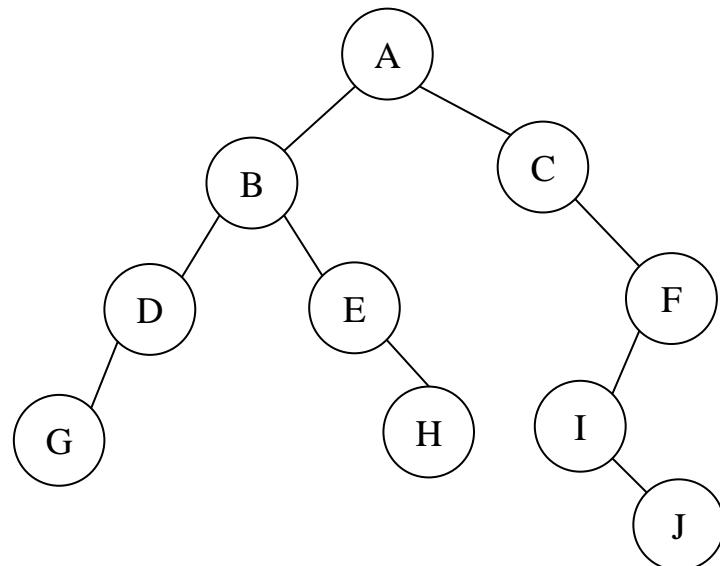
Level-order Traversal (“Breadth-first”) (3)



Quiz 28

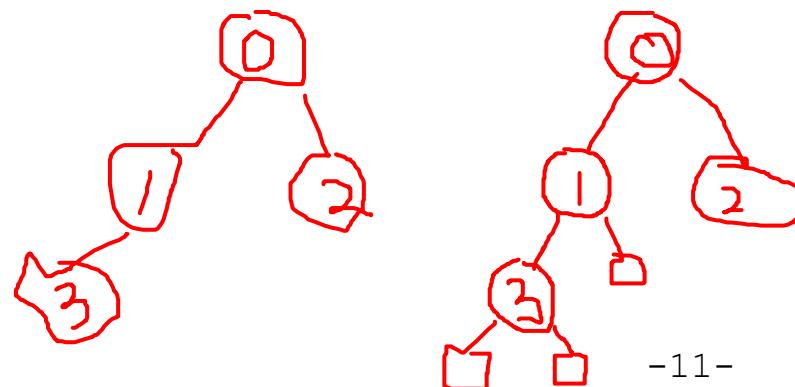
Name and student ID

Write out the inorder, preorder, postorder, and level-order traversals for the given binary tree.



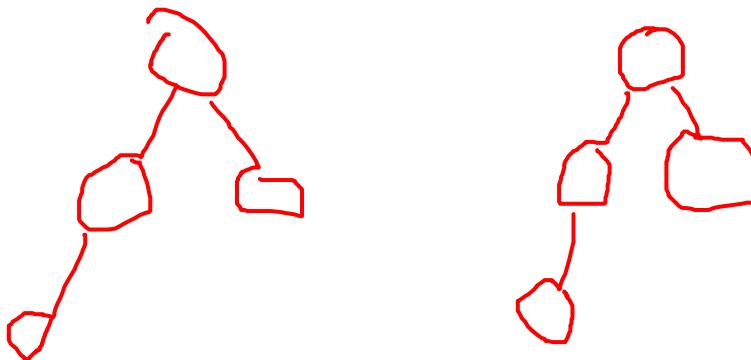
Copying a Binary Tree

```
treePointer copy(treePointer original)
{ /* this function returns a treePointer to an exact copy
   of the original tree */
treePointer temp;
if (original) {
    MALLOC(temp, sizeof(*temp));
    temp->leftChild = copy(original->leftChild);
    temp->rightChild = copy(original->rightChild);
    temp->data = original->data;
    return temp;
}
return NULL;
}
```



Testing for Equality of Binary Trees

```
int equal(treePointer first, treePointer second)
{
    return ((!first && !second) ||
            (first && second && (first->data == second->data) &&
             equal(first->leftChild, second->leftChild) &&
             equal(first->rightChild, second->rightChild)));
}
```



Satisfiability Problem

Problem domain

- The set of formulas constructed by taking variables x_1, x_2, \dots, x_n and operators \wedge (and), \vee (or), and \sim (not)
- The variables can hold one of two possible values, *true* or *false*
- Composition rules;
 - A variable is an expression
 - If x and y are expressions, then, $\sim x$, $x \vee y$, and $x \wedge y$ are expressions
 - Parenthesis can be used to alter the normal order of evaluation, which is \sim before \wedge before \vee

$$\frac{x_1 \wedge x_2 \vee x_3 \vee \sim x_4}{\begin{array}{cccc} T & T & | & | \\ \hline F & F & | & | \end{array}}$$

T - .
T - . .



Evaluation of the Expression (1)

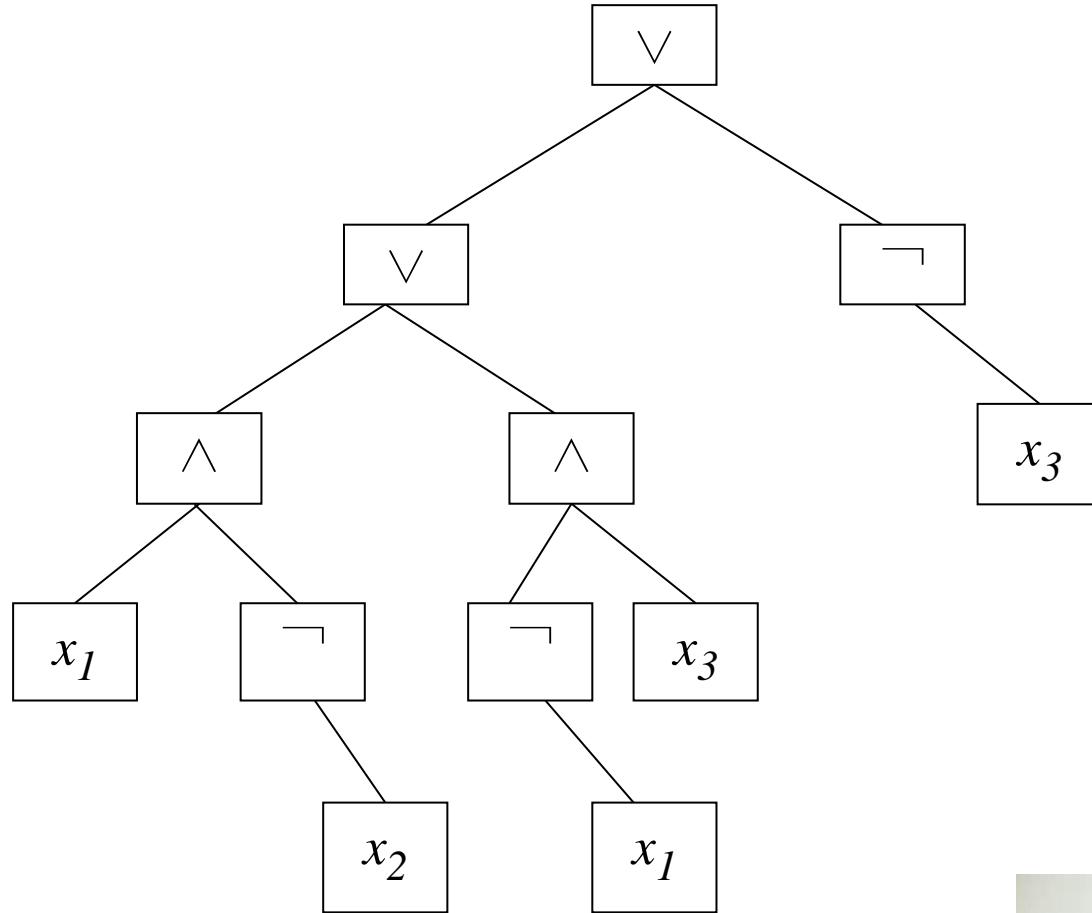
```
typedef enum { not, and, or, true, false} logical;  
typedef struct node* treePointer;  
typedef struct node {  
    treePointer leftChild;  
    logical      data; /* the operator or the  
                        value of the variable */  
    short int     value; /* true or false */  
    treePointer rightChild;  
};
```

| leftChild | data | value | rightChild |
|-----------|------|-------|------------|
| | | | |



An Example of Formula

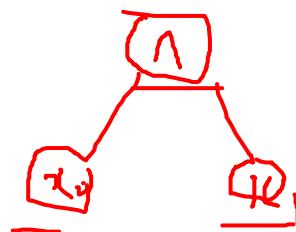
$$(x_1 \wedge \neg x_2) \vee (\neg x_1 \wedge x_3) \vee \neg x_3$$



Evaluation of the Expression (1)

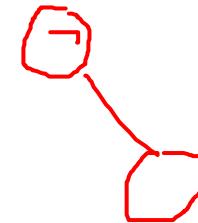
```
for (all 2n possible combinations) {  
    generate the next combination;  
    replace the variables by their values;  
    evaluate root by traversing it in postorder;  
    if (root->value) {  
        printf(<combination>);  
        return;  
    }  
}  
printf("No satisfiable combination");
```

T T
T F
F T
F F



Evaluation of the Expression (2)

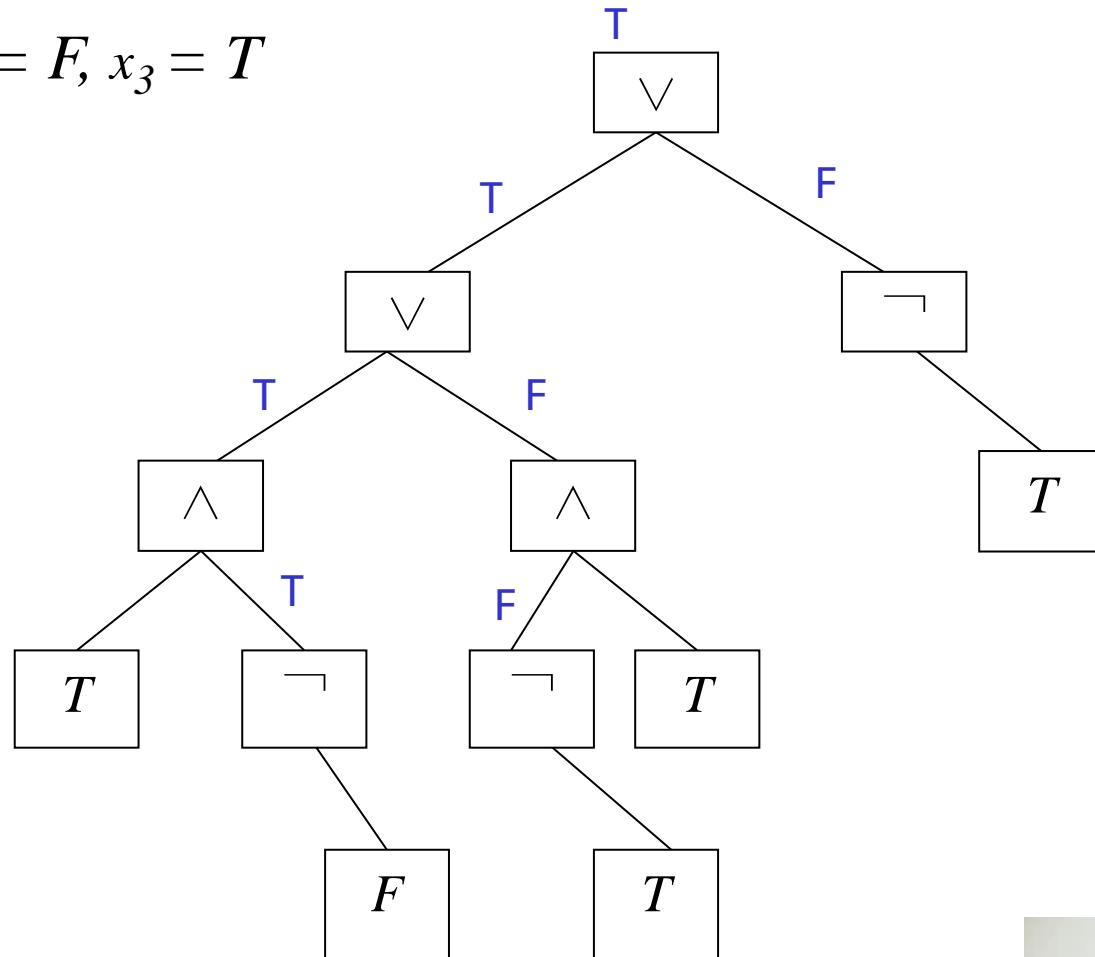
```
void postOrderEval(treePointer node)
{
    if (node) {
        postOrderEval(node->leftChild);
        postOrderEval(node->rightChild);
        switch (node->data) {
            case not: node->value = !node->rightChild->value;
                        break;
            case and: node->value = node->rightChild->value
                        && node->leftChild->value;
                        break;
            case or:  node->value = node->rightChild->value ||
                        node->leftChild->value;
                        break;
            case true: node->value=TRUE; break;
            case false: node->value=FALSE;
        }
    }
}
```



Sample Evaluation

$$(x_1 \wedge \neg x_2) \vee (\neg x_1 \wedge x_3) \vee \neg x_3$$

$x_1 = T, x_2 = F, x_3 = T$



Quiz 29

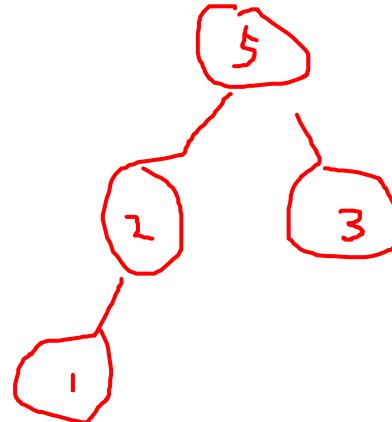
Name and student ID

Write a function treeAdd() which returns the sum of data in a tree.

```
int treeAdd(treePointer tree) {
```

```
    typedef struct node* treePointer;
    typedef struct node {
        int data;
        treePointer leftChild, rightChild;
    };
```

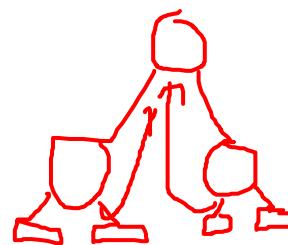
```
}
```



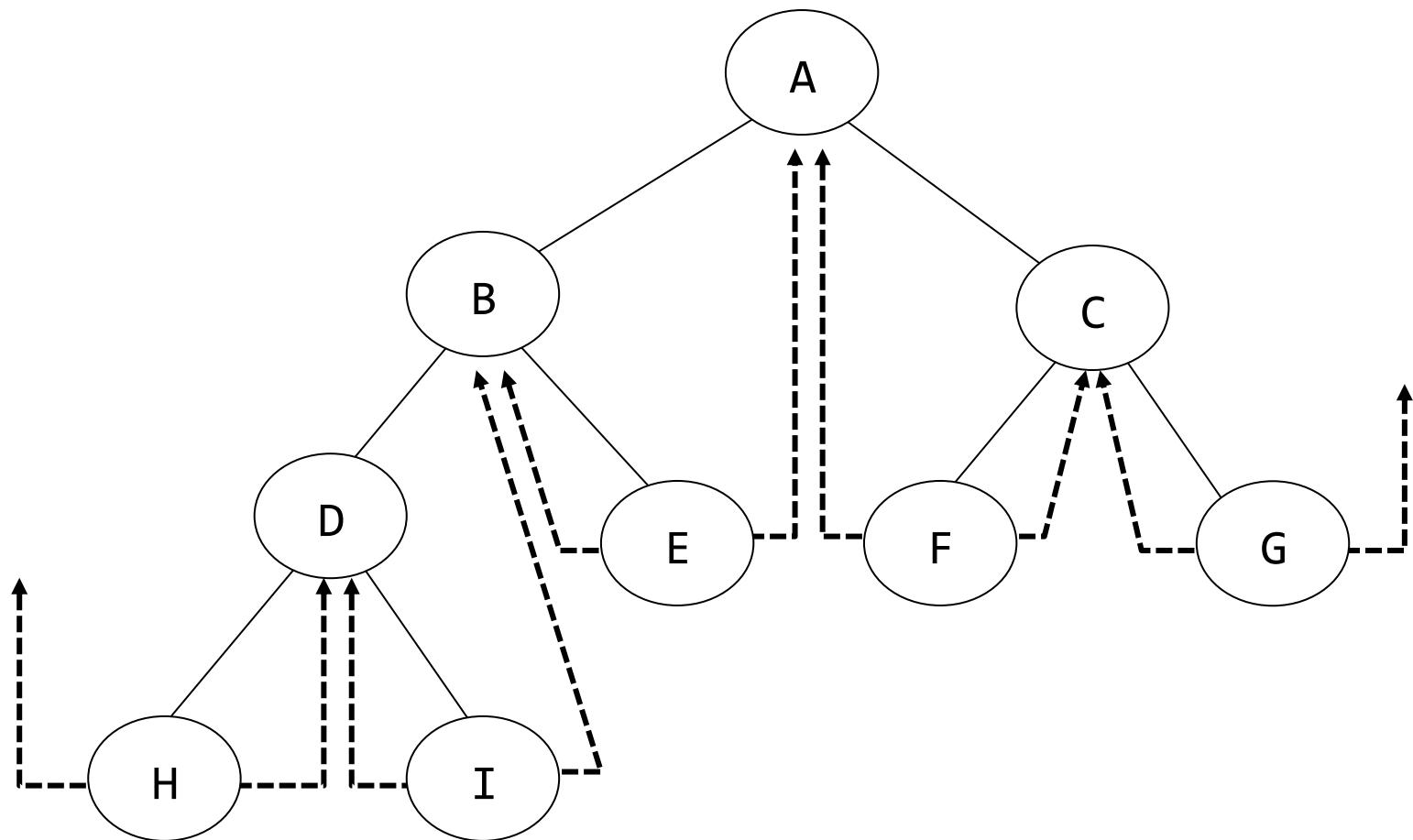
Threaded Binary Tree

Thread

- A way to make use of the null links in the linked representation of a tree
- A null link is replaced by a pointer, called **thread**, to another node in the tree
- If `leftChild` is null, it is replaced by a pointer to its inorder predecessor
- If `rightChild` is null, it is done by one to its inorder successor



An Example of Threaded Binary Tree



H D I B E A F C G



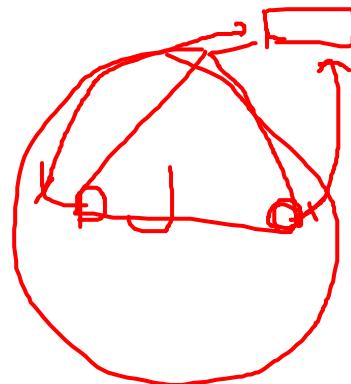
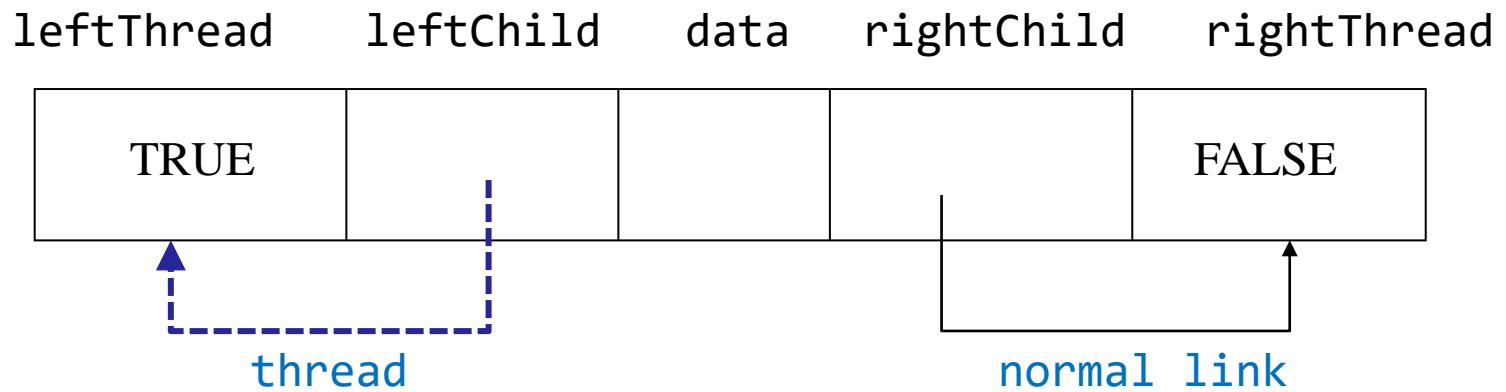
Threaded Binary Tree Representation (1)

```
typedef struct threadedTree *threadedPointer;
typedef struct threadedTree {
    short int leftThread; /* TRUE if leftChild is thread */
    threadedPointer leftChild;
    char data;
    threadedPointer rightChild;
    short int rightThread; /* TRUE if rightChild is thread */
}
```

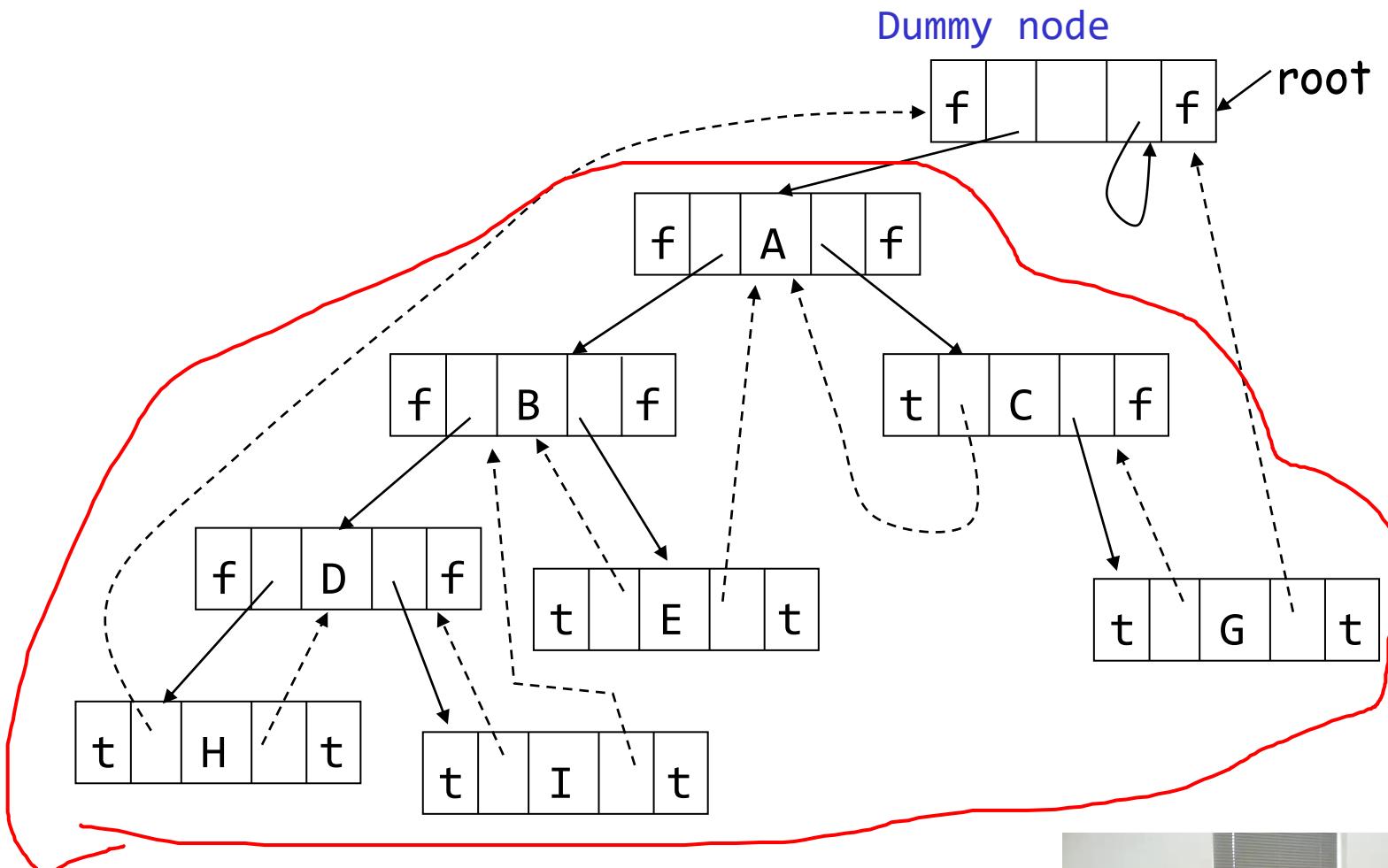


Threaded Binary Tree Representation (2)

An empty threaded tree

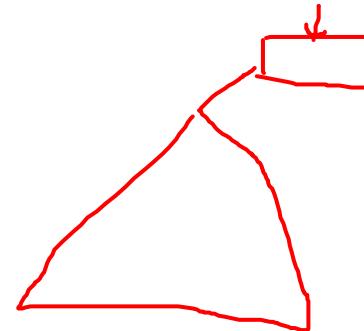


Threaded Binary Tree Representation (3)



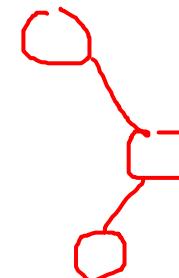
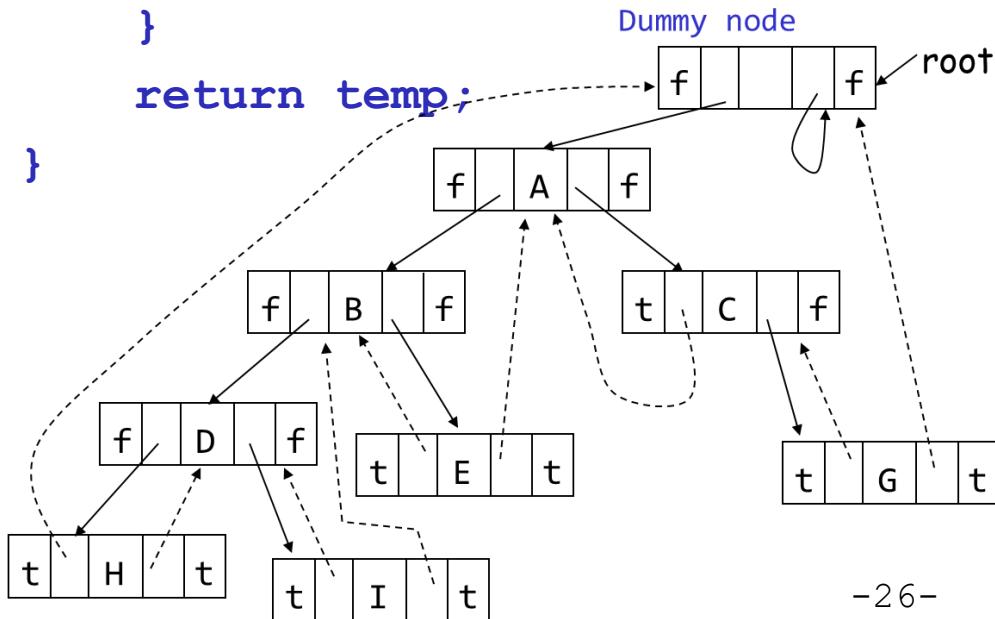
Inorder Traversal of a Threaded Binary Tree

```
void tinorder(threadedPointer tree)
/* traverse the threaded binary tree inorder */
{
    threadedPointer temp = tree;
    for (;;) {
        temp = insucc(temp);
        if (temp == tree) break;
        printf("%3c", temp->data);
    }
}
```

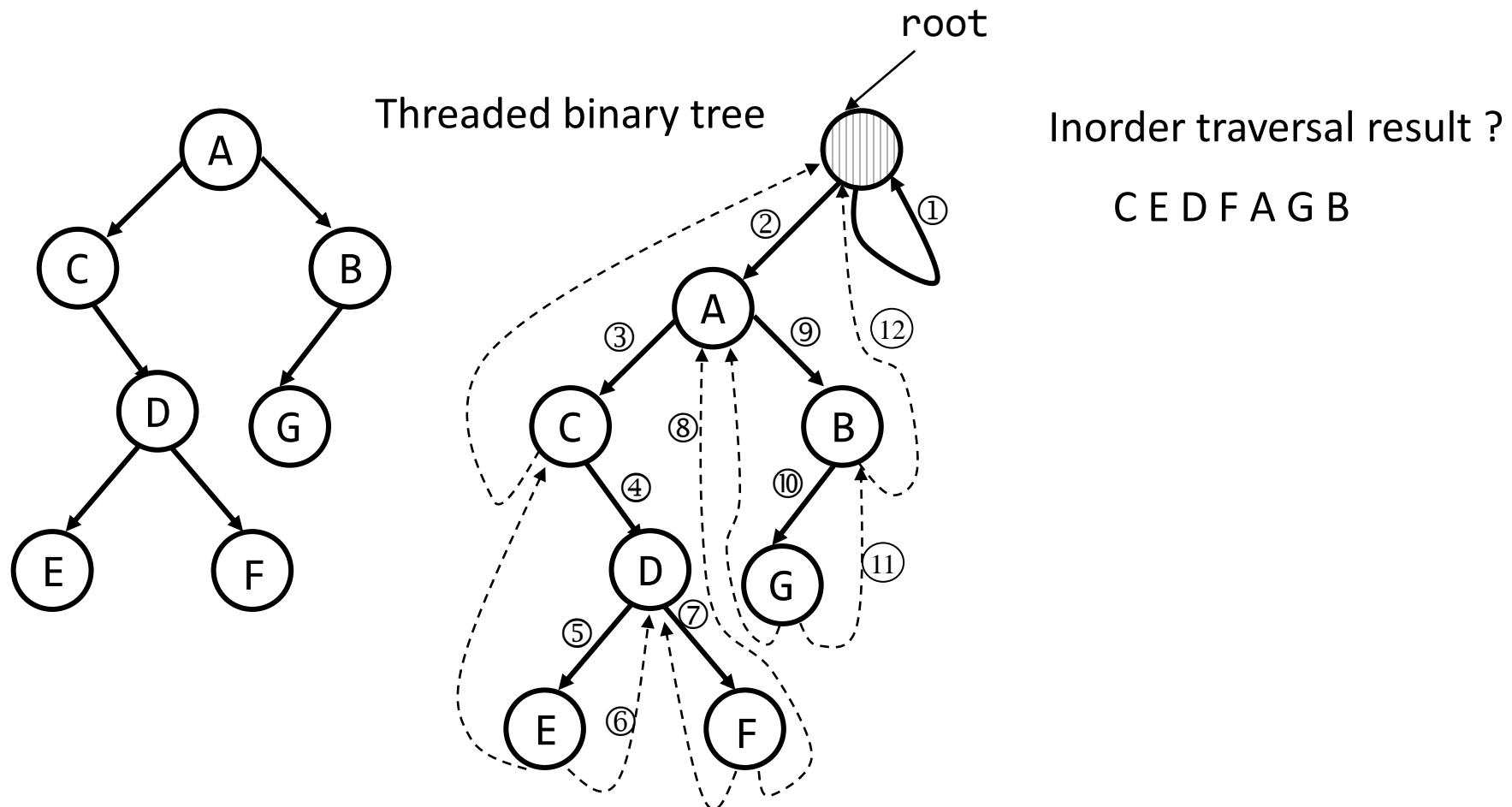


Finding the Inorder Successor

```
threadedPointer insucc(threadedPointer tree)
/* find the inorder successor of tree in a threaded binary
tree */
{
    threadedPointer temp;
    temp = tree->rightChild;
    if (!tree->rightThread){ /* normal link */
        while (!temp->leftThread)
            temp = temp->leftChild;
    }
    return temp;
}
```



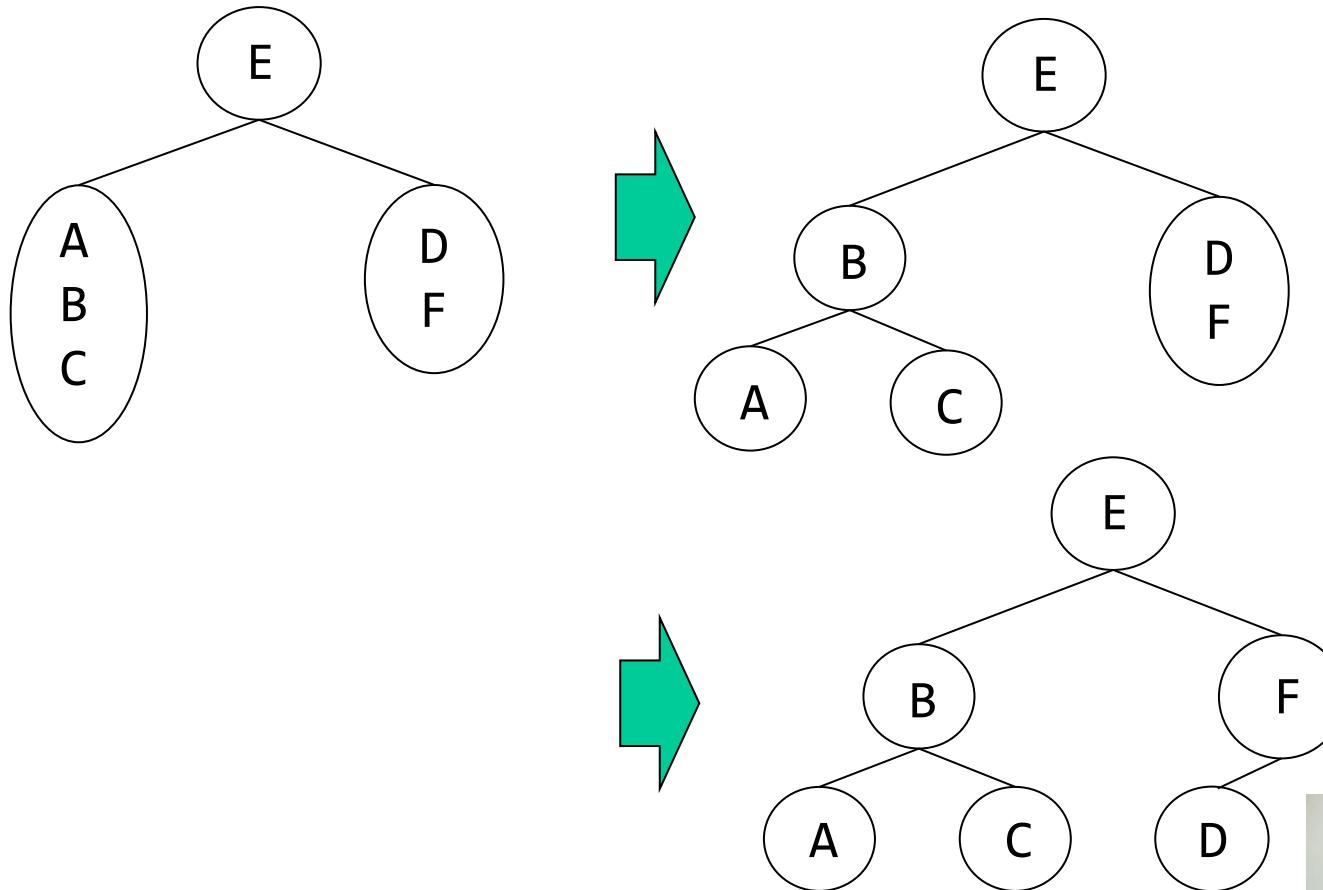
Threaded binary tree example



How to rebuild a tree from traversal results

Inorder traversal A B C E D F

Postorder traversal A C B D F E



Quiz 30

Name and student ID

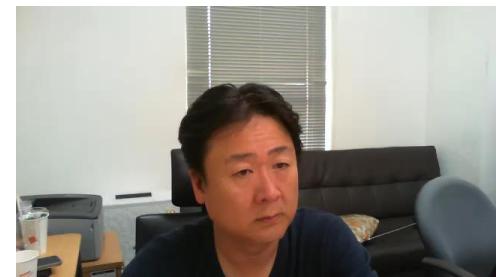
Rebuild a binary tree from the following traversal results:

Inorder traversal: D G B E A F H C

Postorder traversal: G D E B H F C A



Trees - Part 2



Contents

Heaps

Binary Search Trees

Forests

Set Representation

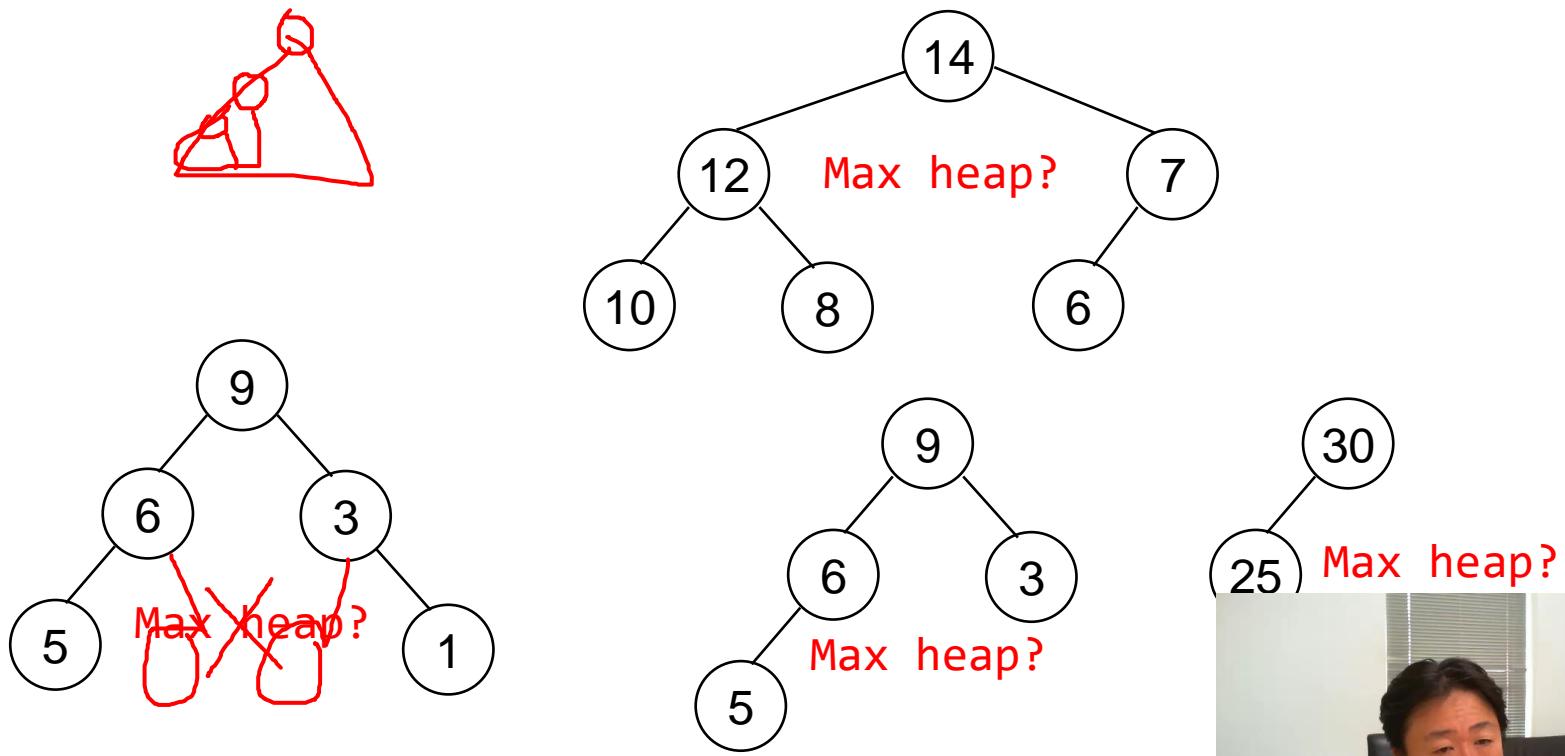
Counting Binary Trees



Max Heap

Definition

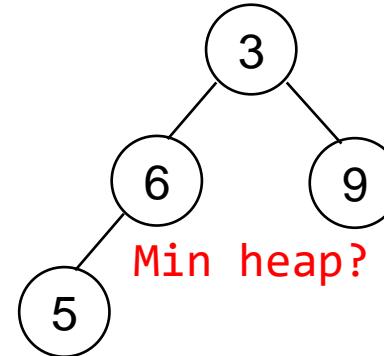
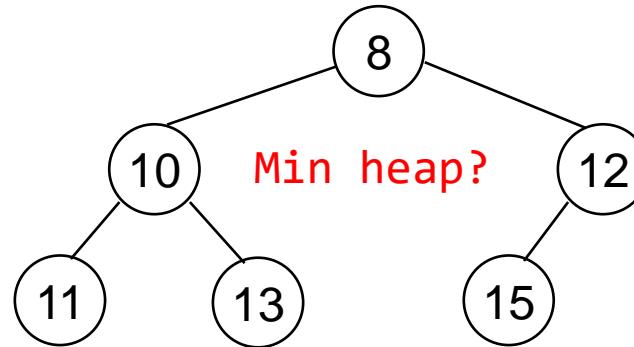
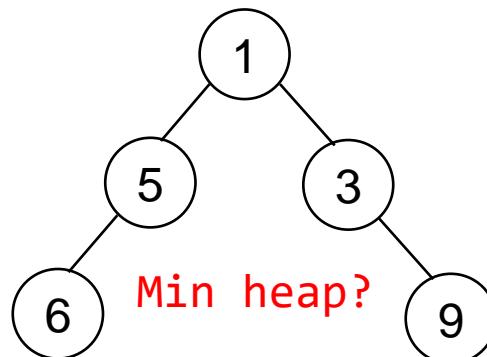
- A max tree is a tree in which the key value in each node is no smaller than the key values in its children (if any)
- A **max heap** is a complete binary tree that is also a max tree



Min Heap

Definition

- A min tree is a tree in which the key value in each node is no larger than the key values in its children (if any)
- A **min heap** is a complete binary tree that is also a min tree



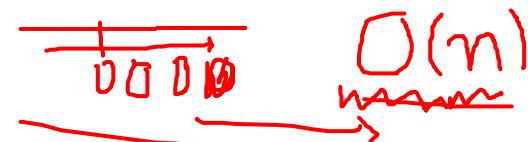
Priority Queues

Priority queue



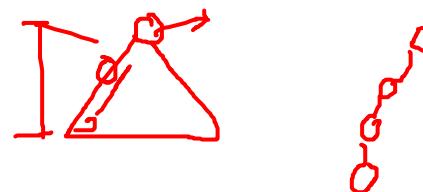
- The items added to a queue have a priority associated with them (payment, importance, ...)
- A queue in which the items are sorted so that the highest priority item is always the next one to be extracted

We could use a tree structure



- It generally provides $O(\log n)$ performance for both insertion and deletion (n is the number of node in a tree)
 - Unfortunately, if the tree becomes unbalanced, performance will degrade to $O(n)$ in pathological cases
- This will probably not be acceptable when dealing with time critical cases.

Heap will provide guaranteed $O(\log n)$ performance for both insertion and deletion



Representations of Priority Queues

| Representation | Insertion | Deletion |
|-----------------------|--------------------------|---------------------------|
| Unordered array | $\Theta(1)$ | $\Theta(n)$ |
| Unordered linked list | $\Theta(1)$ | $\Theta(n)$ |
| Sorted array | $O(n)$ | $\Theta(1)$ |
| Sorted linked list | <u>$O(n)$</u> | $\Theta(1)$ |
| Max heap | $O(\log_2 n)$ | $O(\log_2 n) = O(\log n)$ |

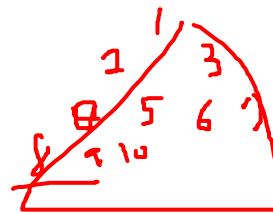
(n is the number of node in a tree)

$$O(n+1) = \underline{\overbrace{O(1) O(1) \dots O(1)}} + O(\log n)$$



Implementation of Max Heap

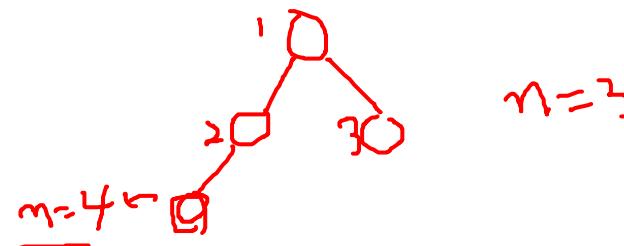
```
#define MAX_ELEMENTS 200 /*maximum size of heap+1*/  
#define HEAP_FULL(n)    (n == MAX_ELEMENTS-1)  
#define HEAP_EMPTY(n)   (!n)  
  
typedef struct {  
    int key;  
    /* other fields */  
} element;  
  
element heap[MAX_ELEMENTS];  
int n = 0;
```



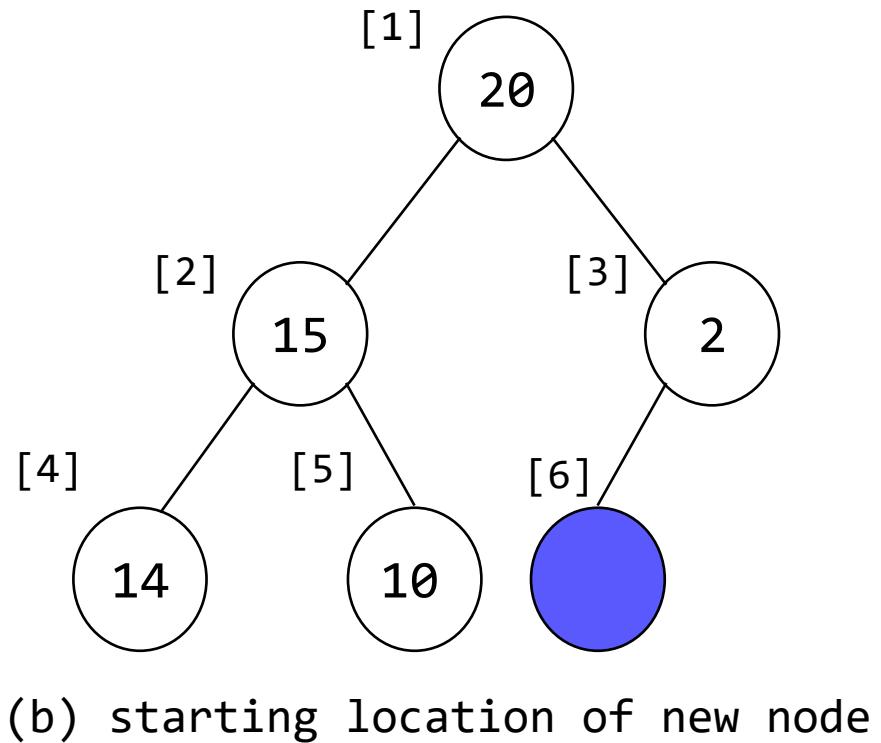
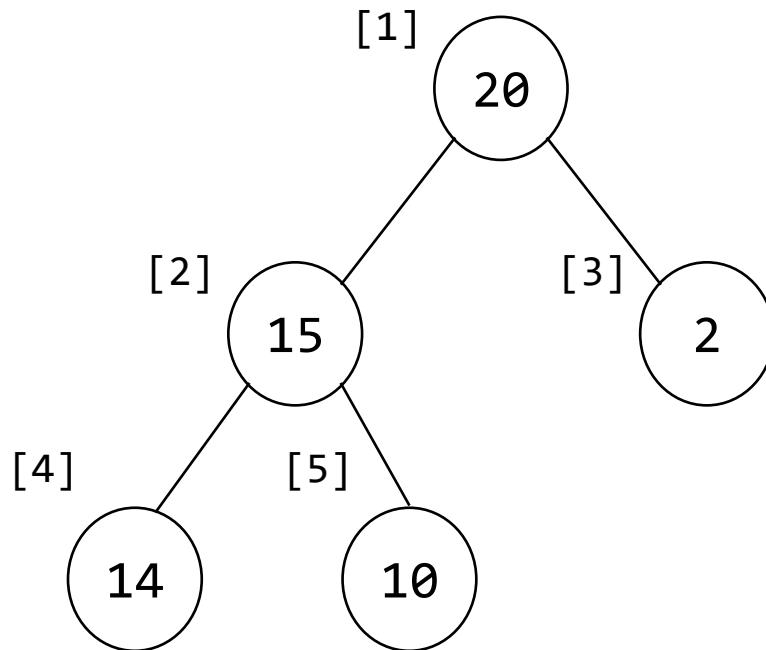
Insertion

```
void push(element item, int *n) {  
    int i;  
    if (HEAP_FULL(*n)) {  
        fprintf(stderr, "The heap is full. ");  
        exit(EXIT_FAILURE);  
    }  
    i = ++(*n);  
    while ((i != 1) && (item.key > heap[i/2].key)) {  
        heap[i] = heap[i/2];  
        i /= 2;  
    }  
    heap[i] = item;  
}
```

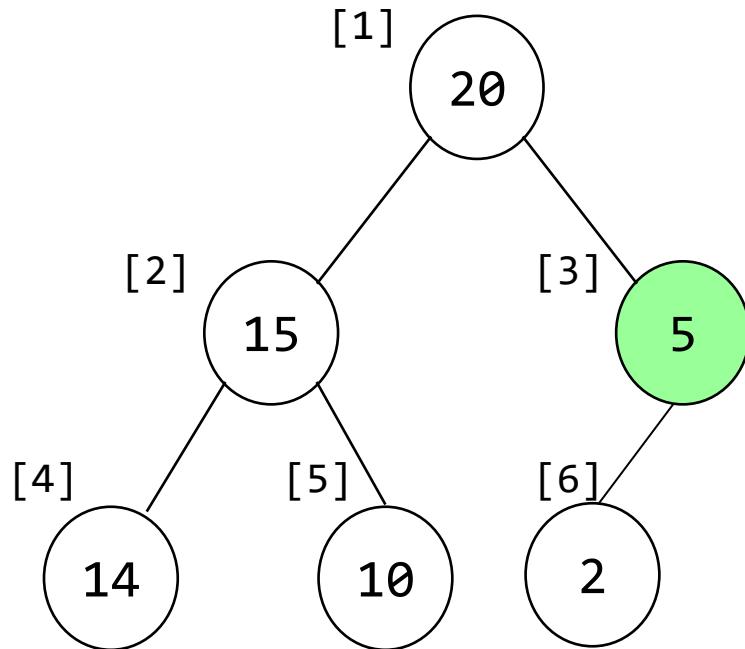
100
! 



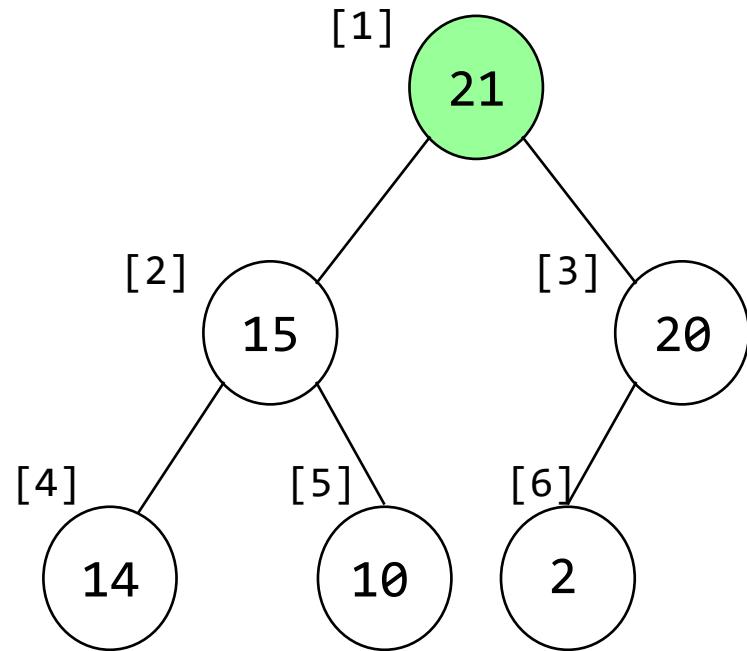
Example of Insertion (1)



Example of Insertion (2)



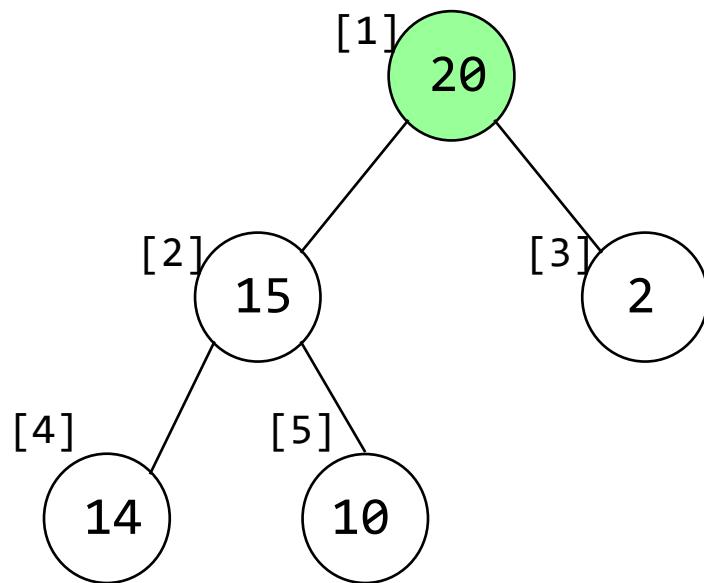
(c) After inserting 5 into the heap of (a)



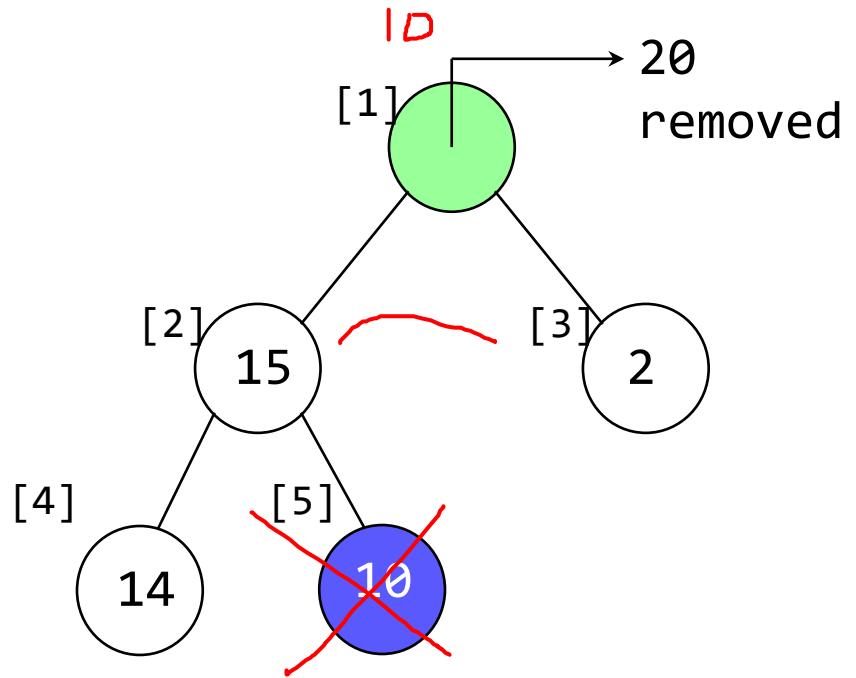
(d) After inserting 21 in into the hean of (a)



Example of Deletion



(a) Initial heap structure

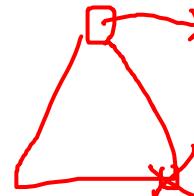


(b) 20 is removed



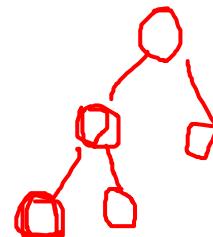
Deletion (1)

```
element pop(int *n)
/* deletes the node of the largest key from the heap */
{
    int parent, child;
    element item, temp;
    if (HEAP_EMPTY(*n)) {
        fprintf(stderr, "The heap is empty");
        exit(EXIT_FAILURE);
    }
    /* store the largest key */
    item = heap[1];
    /* use the last node to reconfigure the heap */
    temp = heap[(*n)--];
    parent = 1;
    child = 2;
```



Deletion (2)

```
while (child <= *n) {  
    /* search the largest child of current parent */  
    if ((child < *n) && (heap[child].key < heap[child+1].key))  
        child++;  
    if (temp.key >= heap[child].key) break;  
    /* move to the lower level */  
    heap[parent] = heap[child];  
    parent = child;  
    child *= 2;  
} /* while */  
heap[parent] = temp;  
return item;  
}
```



Quiz 31

Name and student ID

Draw the max heap after inserting 10, 8, 15, 20, 21, 3, 7, 18 from an empty heap.

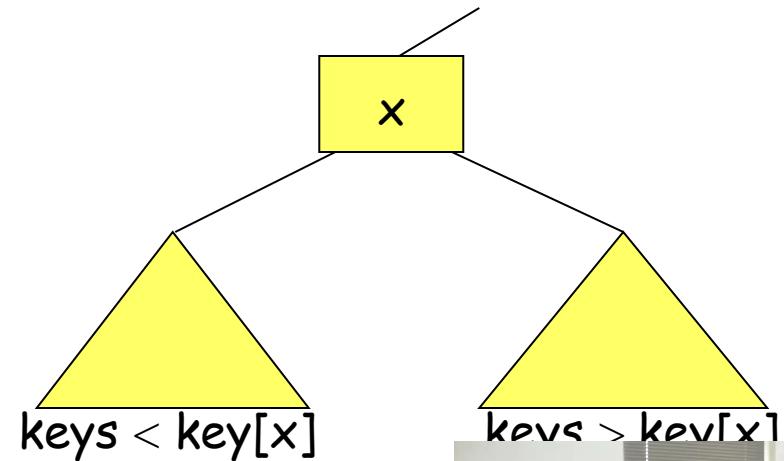


Binary Search Tree

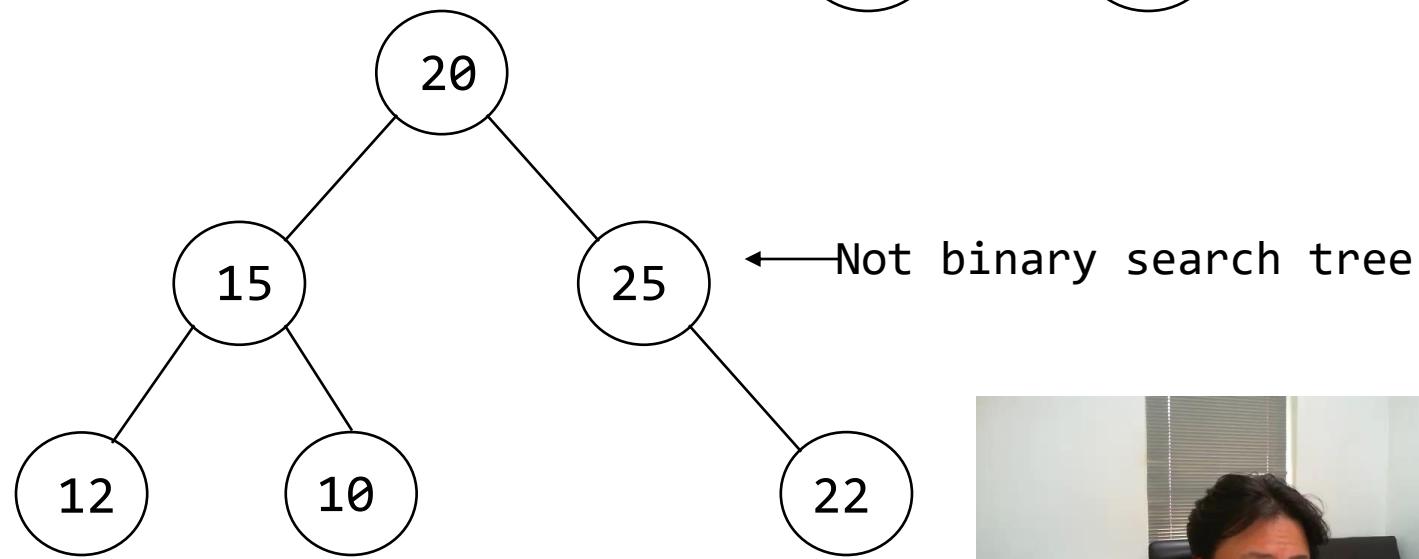
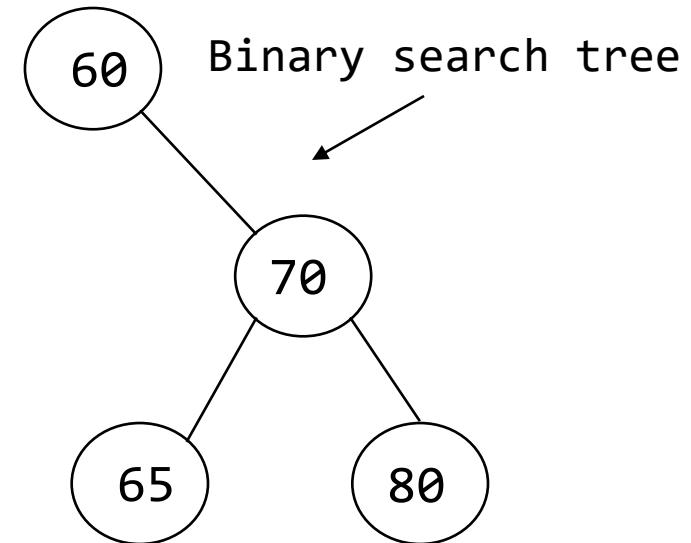
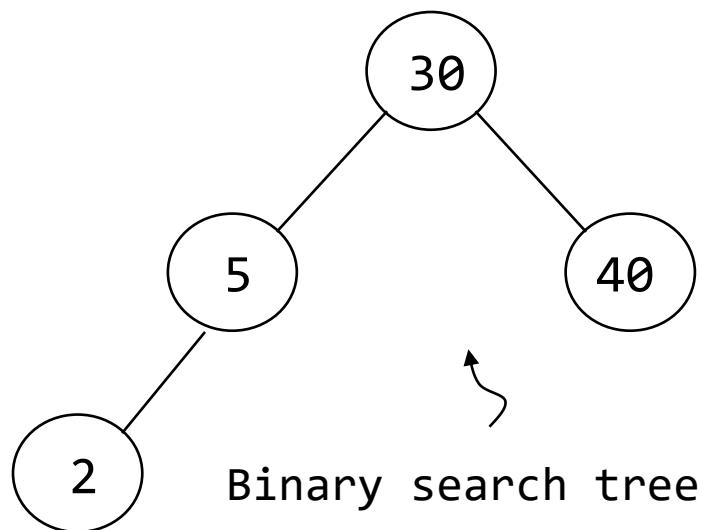
Binary search tree (BST) is a binary tree that is empty or each node satisfies the following properties:

- every element has a key, and no two elements have the same key
- the keys in a nonempty left subtree must be smaller than the key in the root of the subtree
- the keys in a nonempty right subtree must be larger than the key in the root of the subtree
- the left and right subtrees are also BST

```
typedef struct node* treePointer;
typedef struct {
    int key;
} element;
typedef struct node {
    element data;
    treePointer leftChild;
    treePointer rightChild;
};
```



Examples of Binary Search Tree



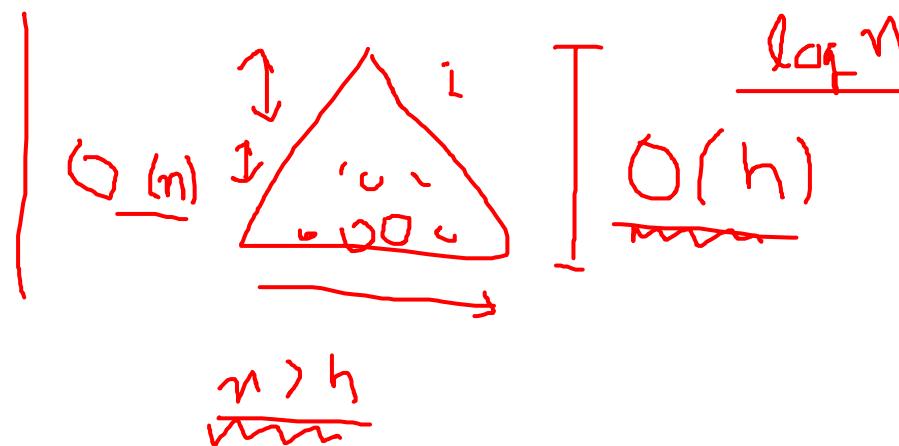
Features of BST

Searching, insertion, deletion is bounded by $O(h)$ where h is the height of the BST

These operations can be performed both

- by key value
 - e.g.) delete the element with key x
- by rank
 - e.g.) delete the fifth smallest element

Inorder traversal of BST generate a sorted list



Searching a BST - Recursive Version

```
element *search(treePointer root,  int k)
{
    if (!root) return NULL;
    if (k == root->data.key) return &(root->data);
    if (k < root->data.key)
        return search(root->leftChild,  k);
    return search(root->rightChild,  k);
}
```



Searching a BST - Iterative Version

```
element *iterSearch (treePointer tree,  int k)
{
    while (tree) {
        if (k == tree->data.key) return &(tree->data);
        if (k < tree->data.key)
            tree = tree->leftChild;
        else
            tree = tree->rightChild;
    } /* while */
    return NULL;
}
```



Time Complexity of Searching BST

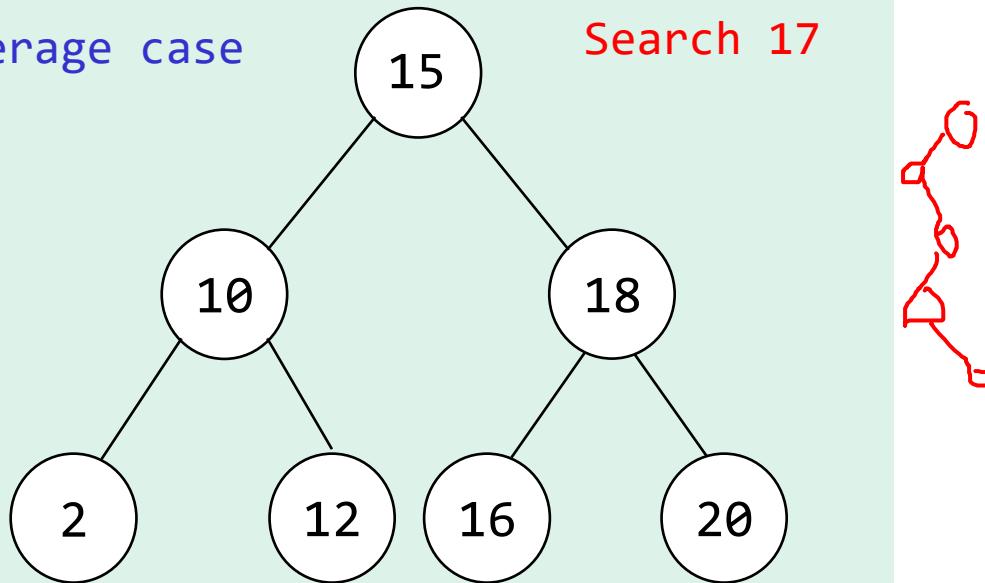
Average case

$O(h)$, where h is the height of BST

Worst case

$O(n)$, where n is the number of nodes

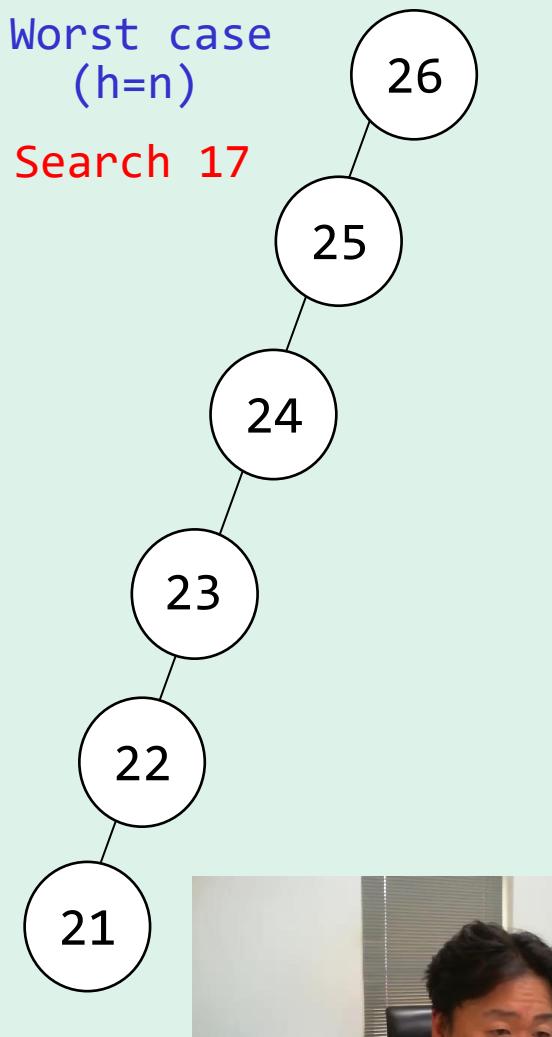
Average case



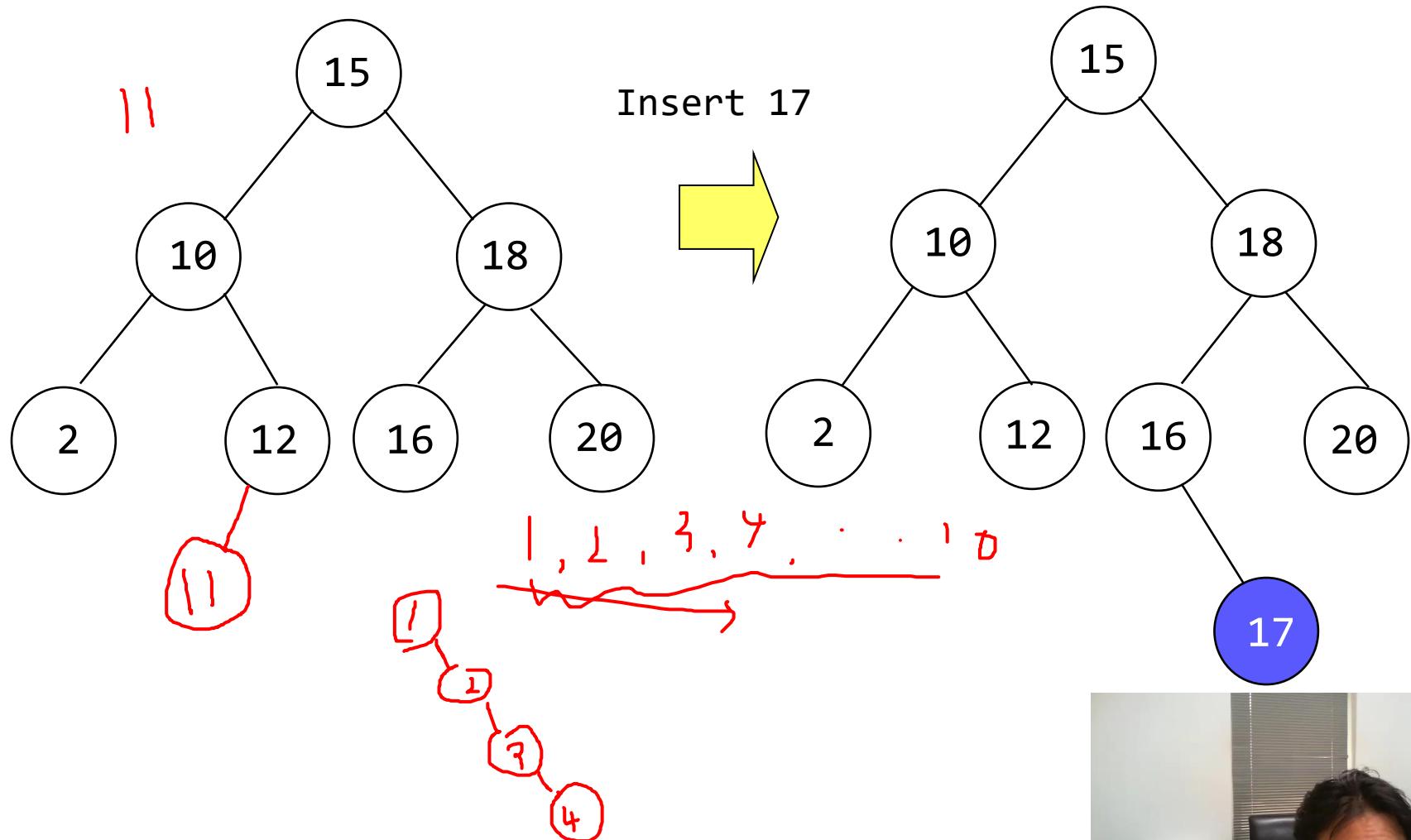
Search 17

Worst case
($h=n$)

Search 17

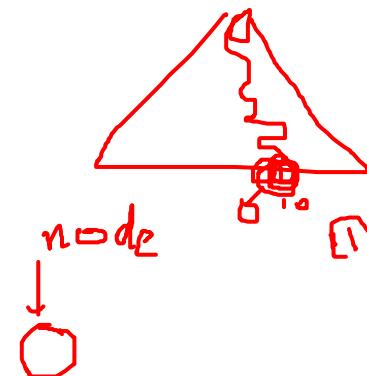


Example of Insertion



Insertion (1)

```
void insert(treePointer *node, int k) {  
    treePointer ptr, temp = modifiedSearch(*node, k);  
    if (temp || !(*node)) {  
        MALLOC(ptr, sizeof (*ptr));  
        ptr->data.key = k;  
        ptr->leftChild = ptr->rightChild = NULL;  
        if (*node) {  
            if (k < temp->data.key) {  
                temp->leftChild = ptr;  
            } else {  
                temp->rightChild = ptr; }  
            } else { *node = ptr; }  
    }  
}
```

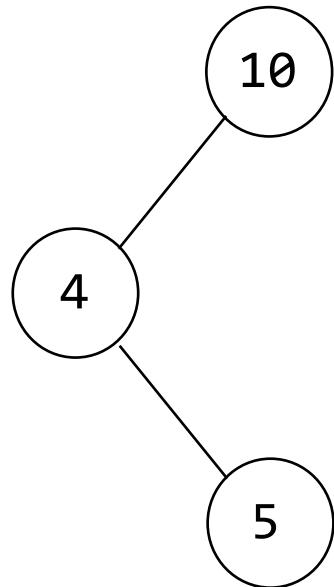


modifiedSearch() is slightly modified version of function iterSearch
return NULL, if the tree is empty or the key is present
return a pointer to the last node of the tree that was encountered during

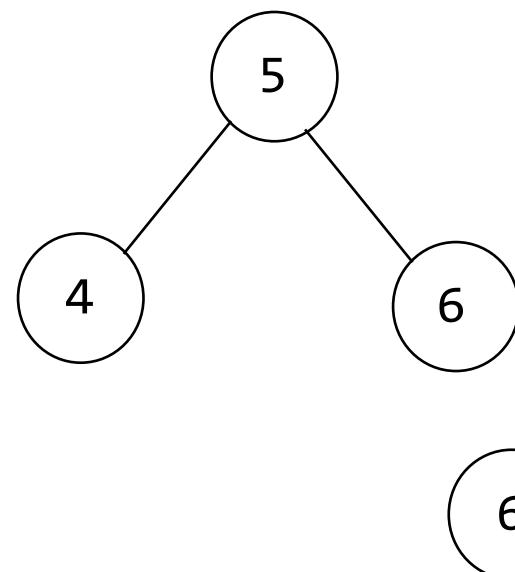


Example of Insertion

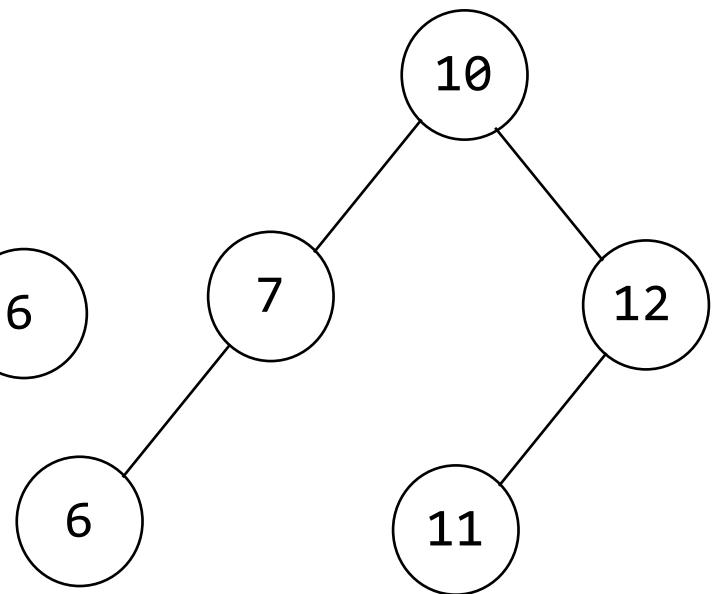
10 4 5



5 6 4



10 12 7 11 6



Quiz 32

Name and student ID

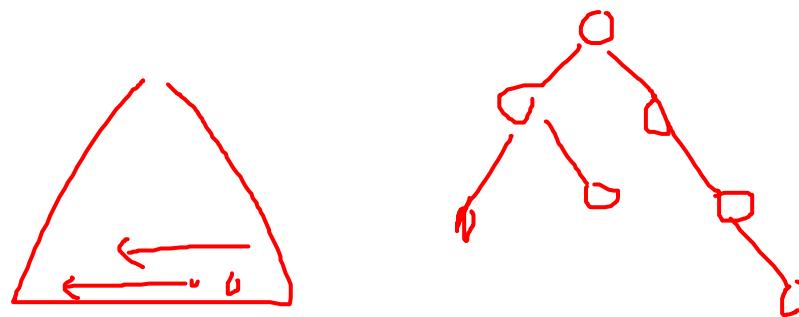
Draw the BST after inserting 10, 8, 15, 20, 21, 3, 7, 18 from an empty BST.



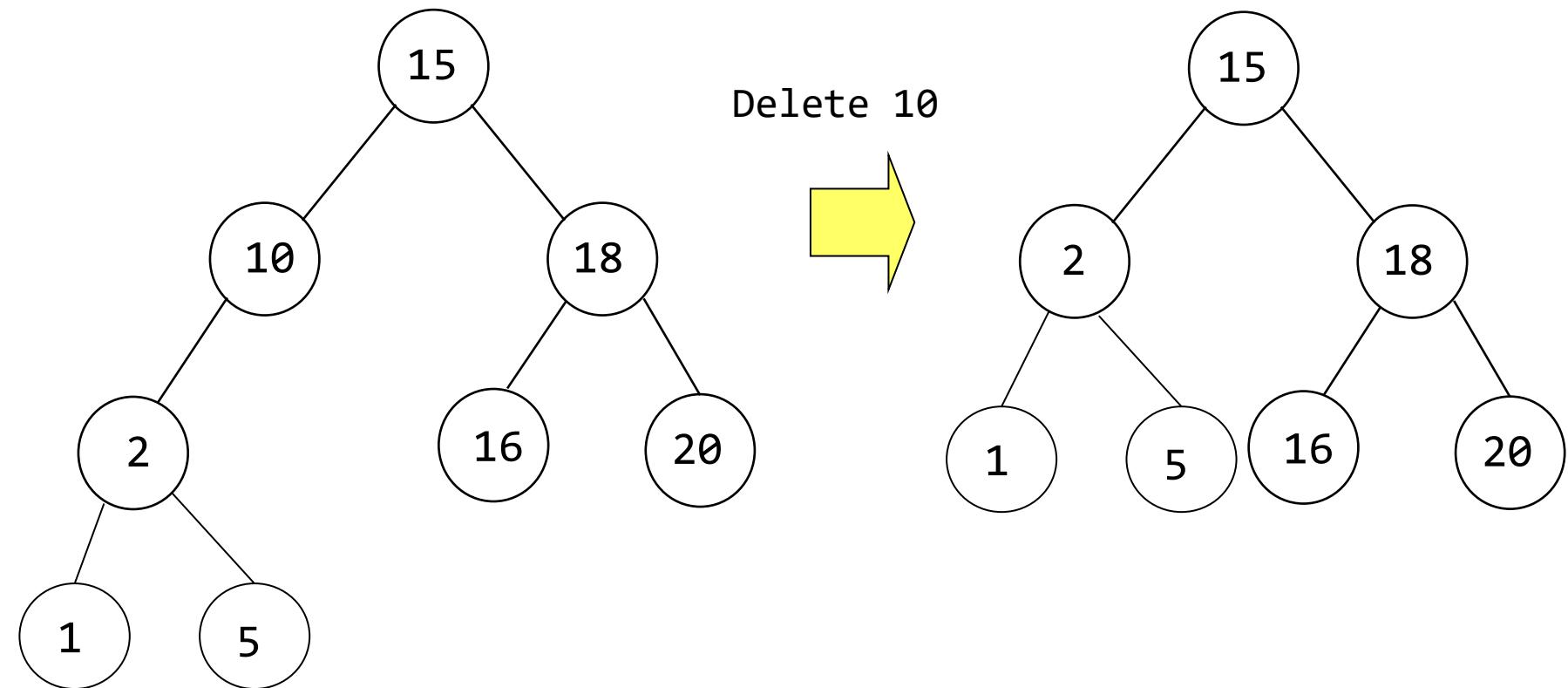
Deletion

Deleting from a BST

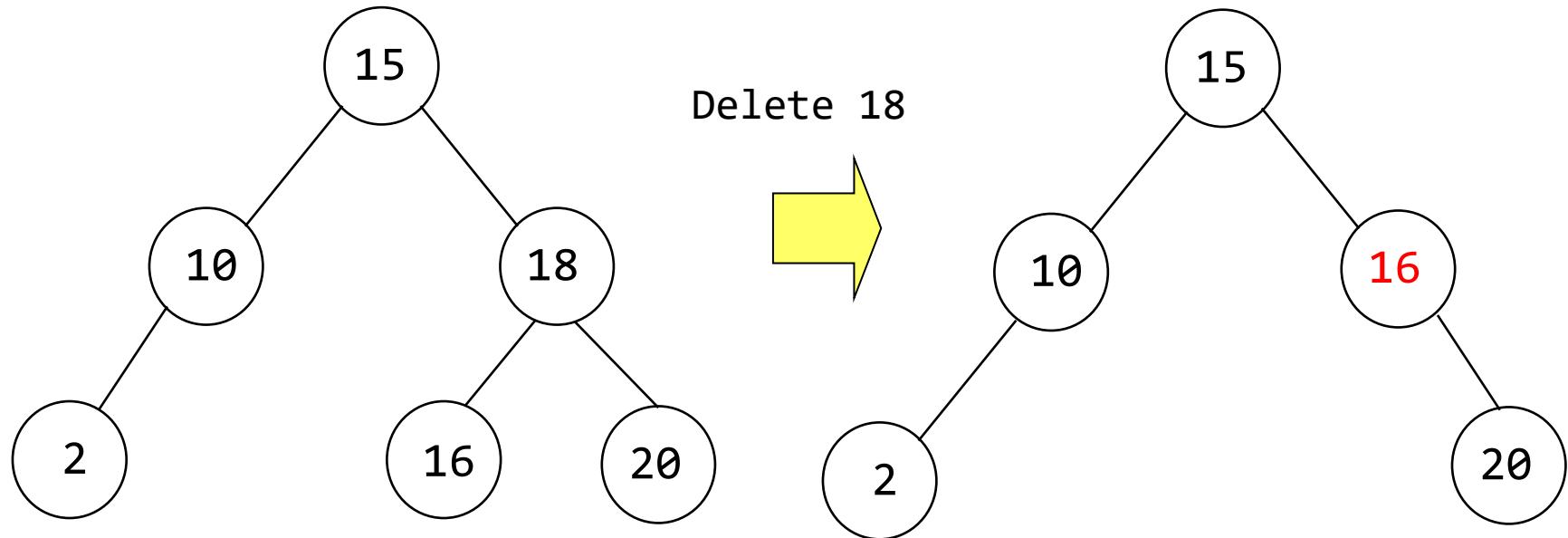
- deletion of a leaf node
- deletion of a node with 1 child
- deletion of a node with 2 children



Deleting a Node with 1 Child Node



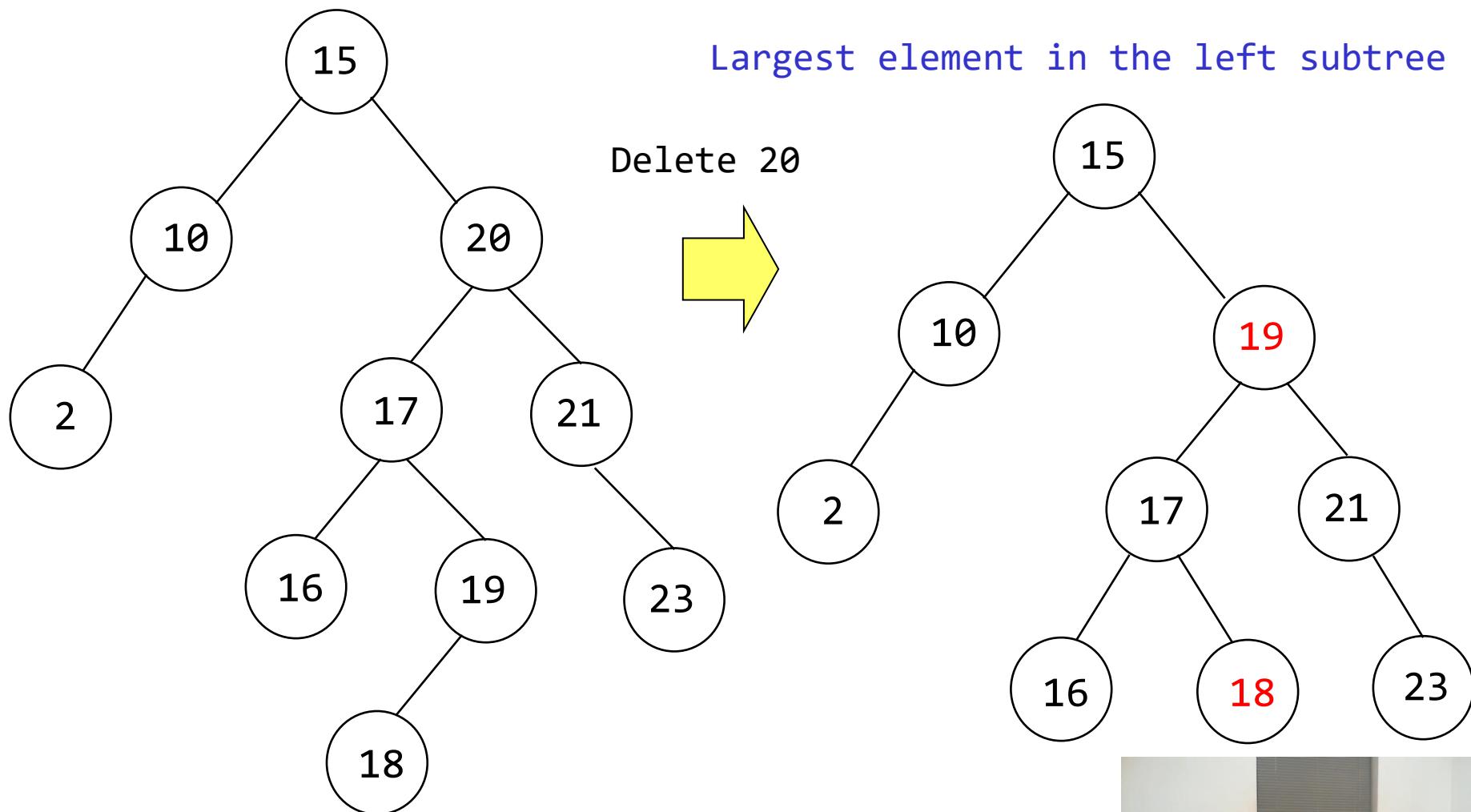
Deleting a Node with 2 Child Nodes (1)



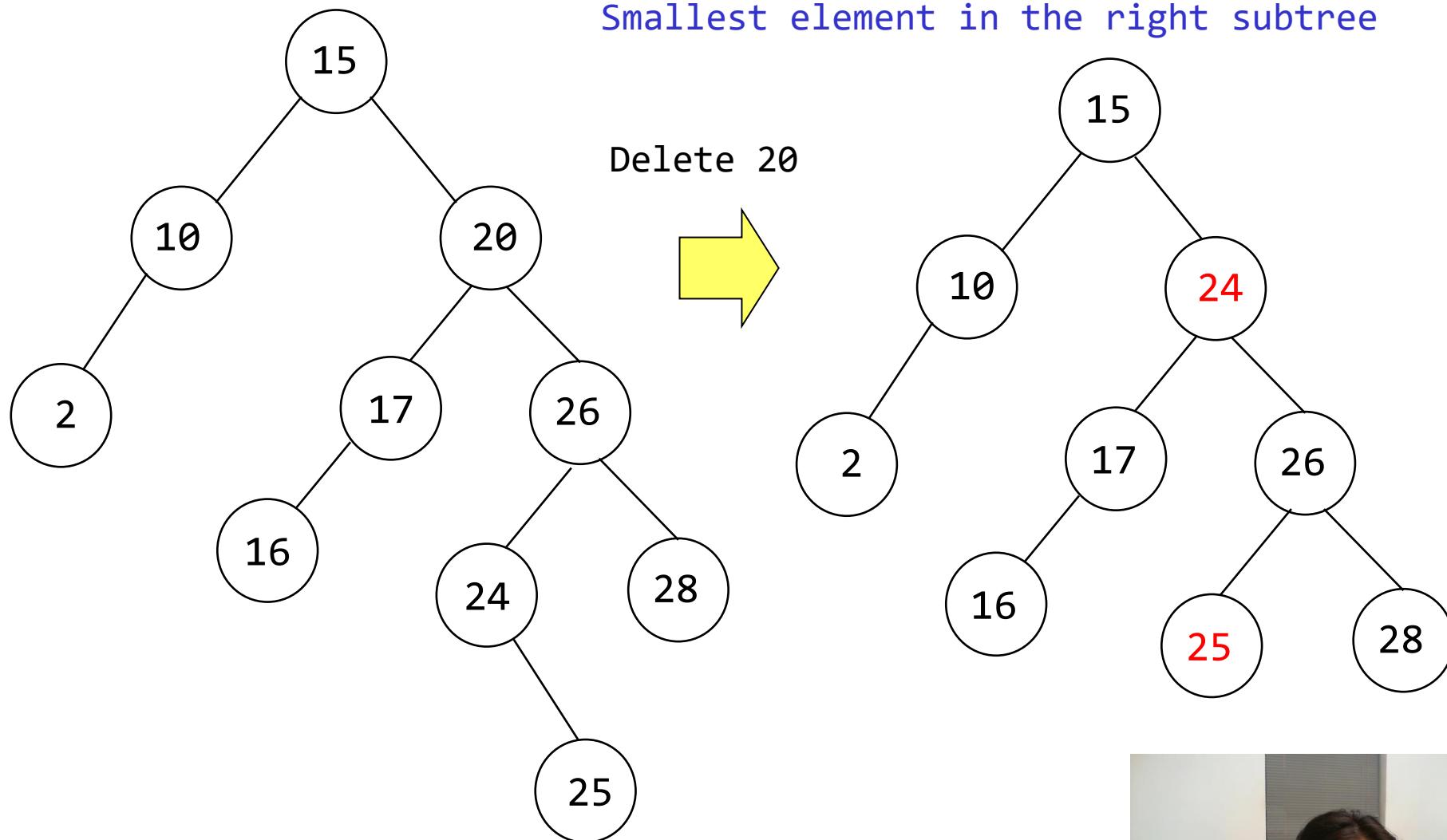
→ Time complexity : $O(h)$, where h is the height of BST



Deleting a Node with 2 Child Nodes (2)



Deleting a Node with 2 Child Nodes (3)



Height of BST

The height of a BST with n elements

- average case: $O(\log_2 n)$
- worst case: $O(n)$

e.g.) Use `insert()` to insert the keys $1, 2, 3, \dots, n$ into an initially empty BST

Balanced Search Trees

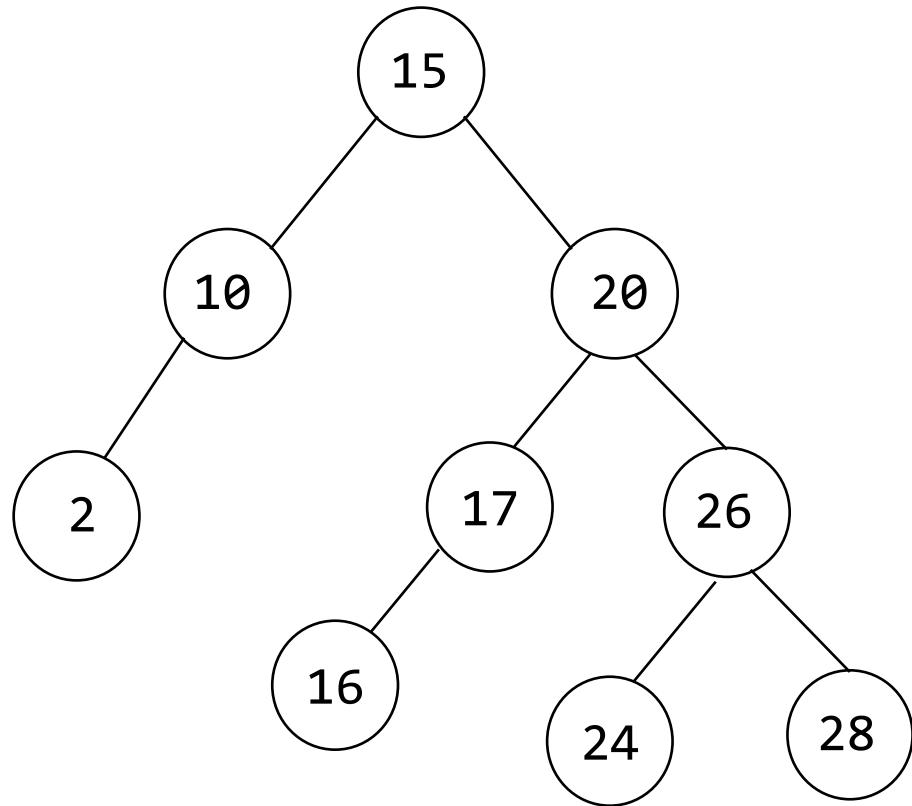
- Worst case height : $O(\log_2 n)$
 - Searching, insertion, deletion is bounded by $O(h)$, where h is the height of a binary tree
- e.g.) AVL tree, 2-3 tree, red-black tree



Quiz 33

Name and student ID

Draw the resulting tree after deleting 15.



Graph - part 1



Contents

Graph Abstract Data Type

Elementary Operations

Spanning Trees

Shortest Paths

Activity Networks



GRAPH ABSTRACT DATA TYPE

Introduction

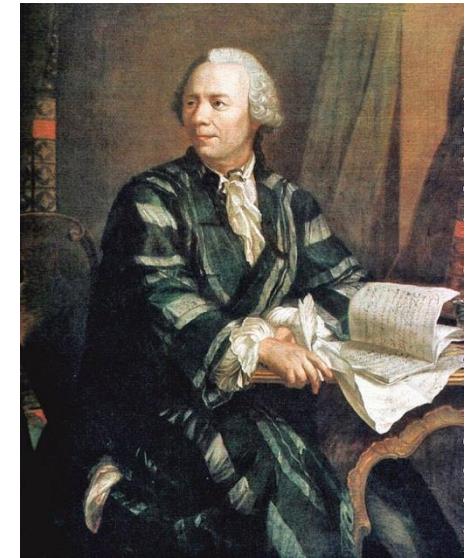
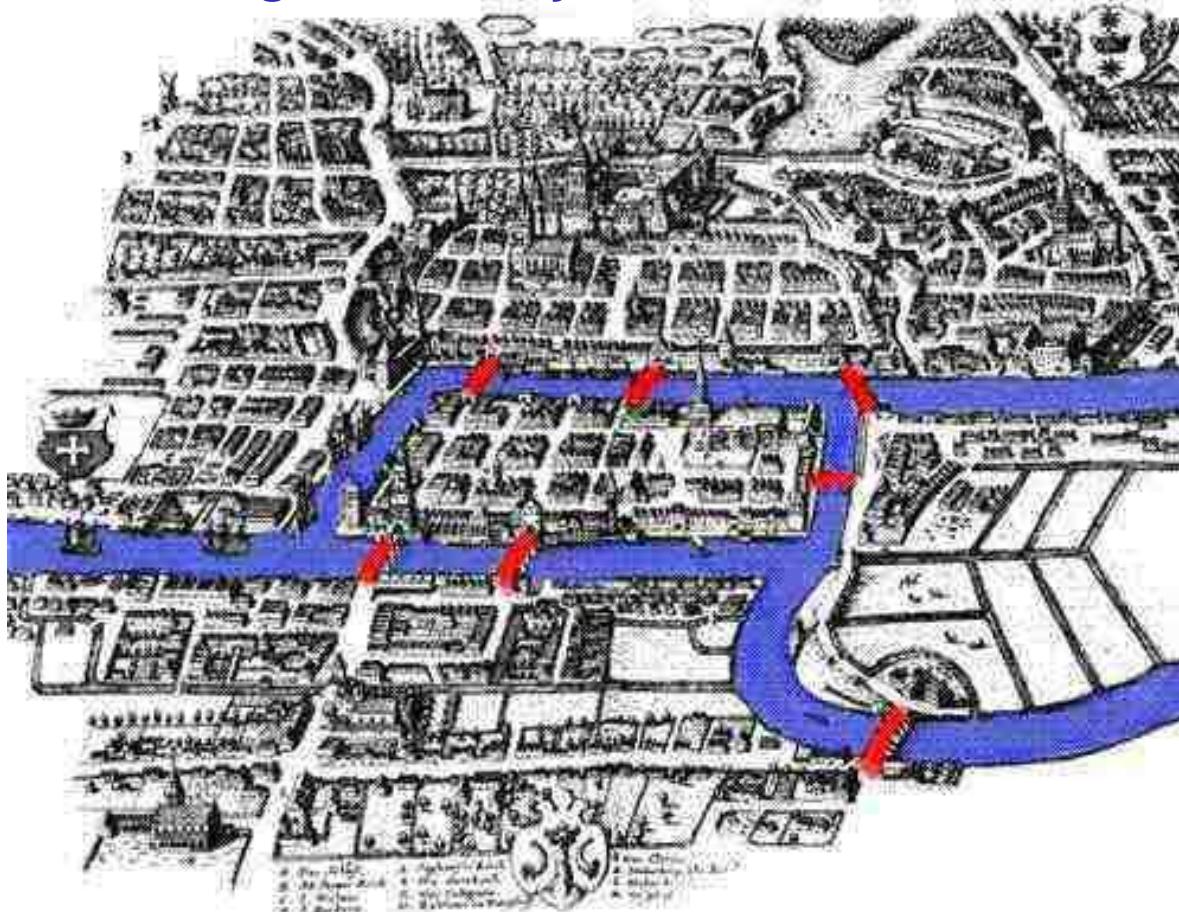
Definitions

Graph representations



The First Use of Graph

Starting at some land area, is it possible to return to our starting location after walking across each of the bridges exactly once ?



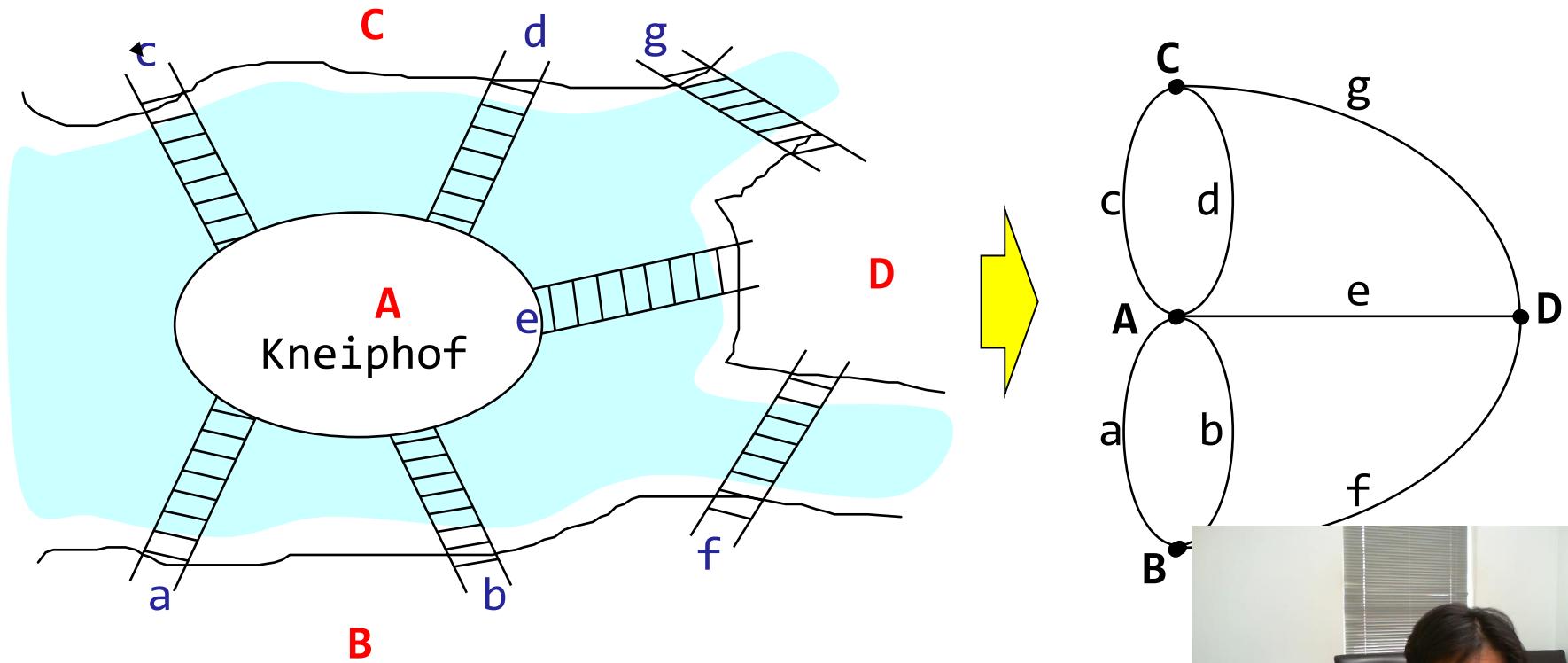
Leonardo Euler
(1707-1783)



Euler's Walk

Abstraction of the problem

- Land areas are transformed into vertices and the bridges into edges
- The degree of a vertex is defined as the number of edges incident on it



Applications of Graph

Dealing with problems which have a fairly natural graph/network structure, for example:

- Road networks (subway networks)
 - nodes = towns/road junctions
 - arcs = roads
- Communication networks
 - Designing telephone systems
 - Computer network planning
 - Routing tree building
- Computer systems
 - Circuit equation
- Foreign exchange/multinational tax planning (network of fiscal flows)
- Social graph



Definitions

$G = (V, E)$, where

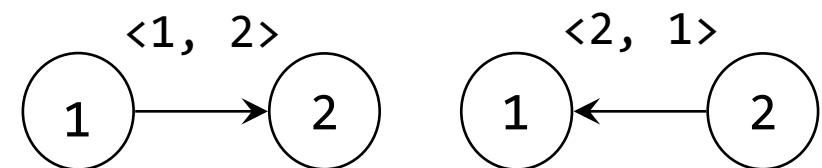
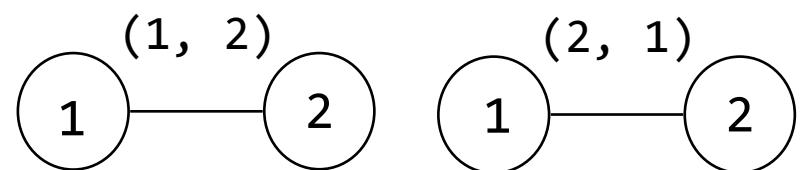
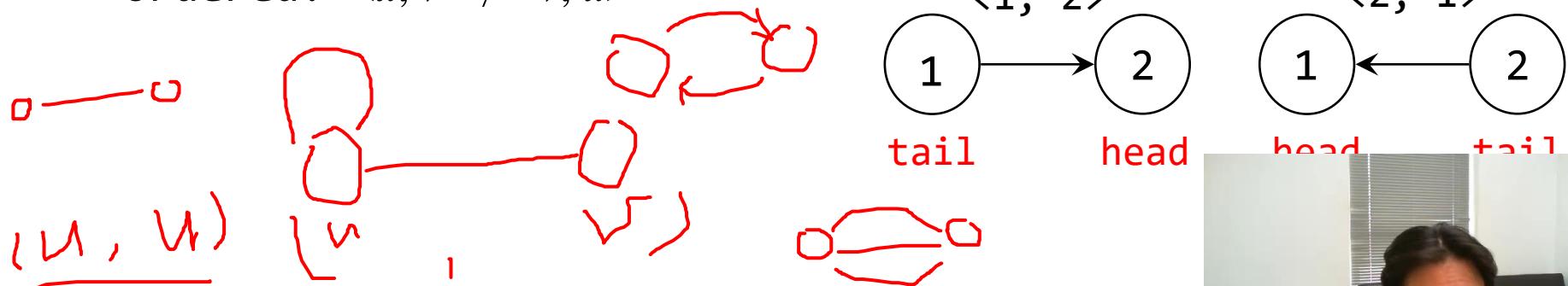
- $V(G)$: set of vertices - finite and nonempty
- $E(G)$: set of edges - finite and possibly empty
- Restrictions
 - A graph may not have an edge from a vertex, i , back to itself
 - A graph may not have multiple occurrence of the same edge

Undirected graph

- unordered: $(u, v) = (v, u)$

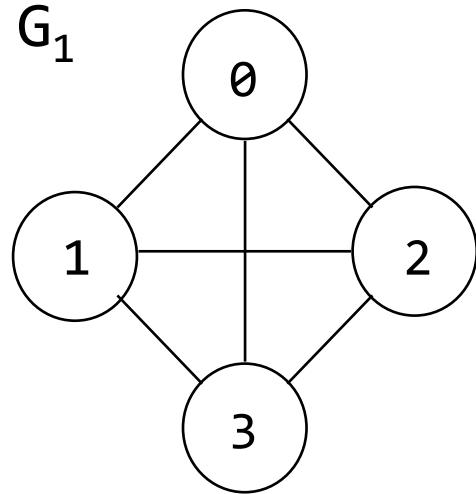
Directed graph (digraph)

- ordered: $\langle u, v \rangle \neq \langle v, u \rangle$



Examples of Graph (1)

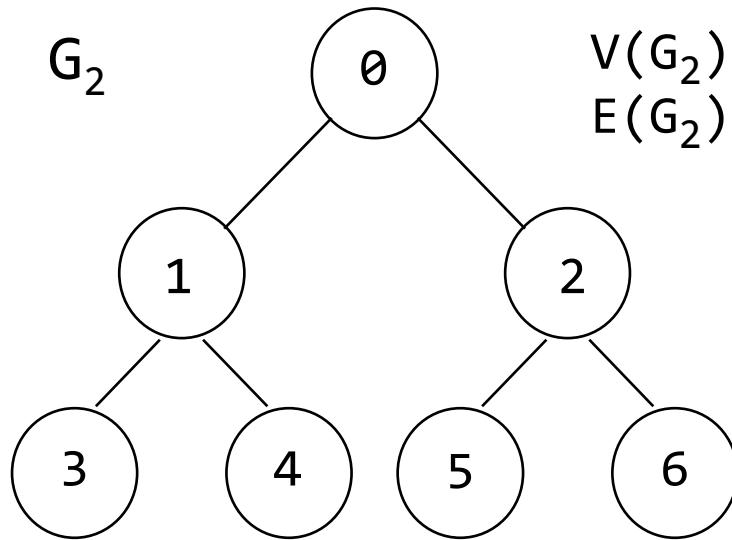
G_1



$$V(G_1) = \{0, 1, 2, 3\}$$

$$E(G_1) = \{(0,1), (0,2), (0,3), (1,2), (1,3), (2,3)\}$$

G_2



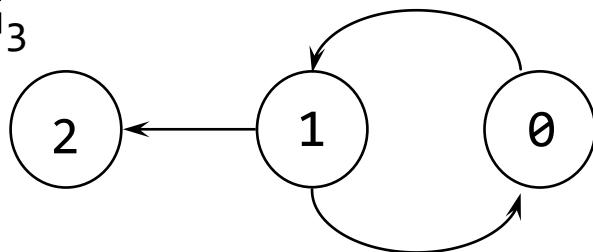
$$V(G_2) = \{0, 1, 2, 3, 4, 5, 6\}$$

$$E(G_2) = \{(0,1), (0,2), (1,3), (1,4), (2,5), (2,6)\}$$



Examples of Graph (2)

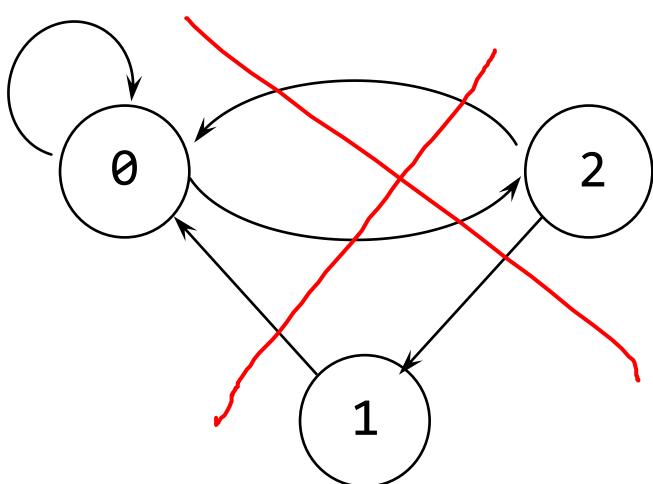
G_3



$$V(G_3) = \{0, 1, 2\}$$

$$E(G_3) = \{\langle 0, 1 \rangle, \langle 1, 0 \rangle, \langle 1, 2 \rangle\}$$

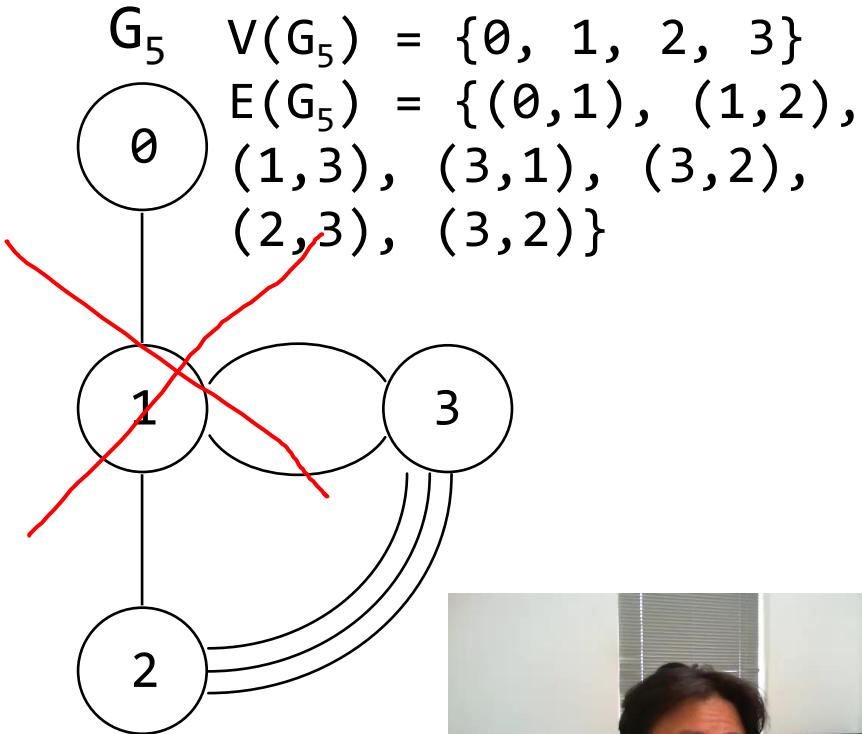
G_4



$$V(G_4) = \{0, 1, 2\}$$

$$E(G_4) = \{ \langle 0, 0 \rangle, \langle 0, 2 \rangle, \langle 1, 0 \rangle, \langle 2, 1 \rangle, \langle 2, 0 \rangle \}$$

G_5



$$V(G_5) = \{0, 1, 2, 3\}$$

$$E(G_5) = \{(0,1), (1,2), (1,3), (3,1), (3,2), (2,3), (3,2)\}$$



Terminology (1)

Complete graph

A graph that has the maximum number of edges

→ For an undirected graph with n vertices,
the maximum number of the edges = $n(n-1)/2$

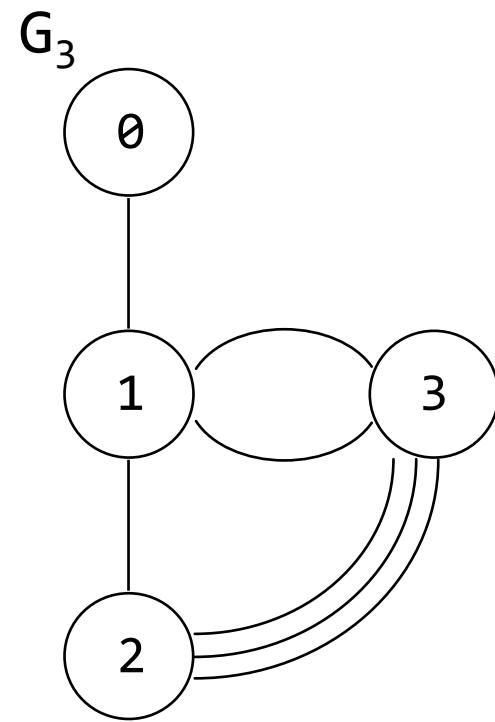
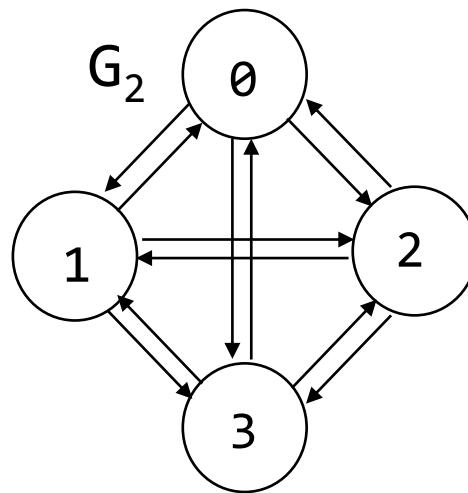
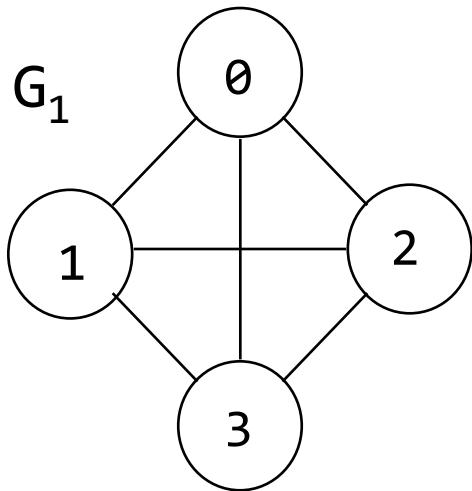
→ For an directed graph with n vertices,
the maximum number of the edges = $n(n-1)$

Multigraph

A graph whose edges are unordered pairs of vertexes,
and the same pair of vertexes can be connected by multiple edges



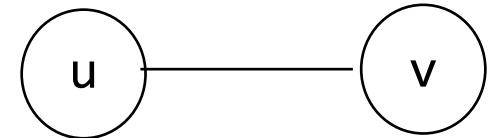
Terminology - Example



Terminology (2)

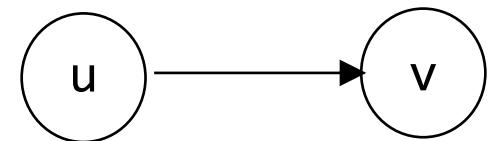
If (u, v) is an edge of an **undirected graph**,

- The vertices u and v are **adjacent**
- The edge (u, v) is **incident on** u and v



If $\langle u, v \rangle$ is a **directed edge**,

- The vertex u is **adjacent to** v
- The vertex v is **adjacent from** u
- The edge $\langle u, v \rangle$ is **incident on** u and v

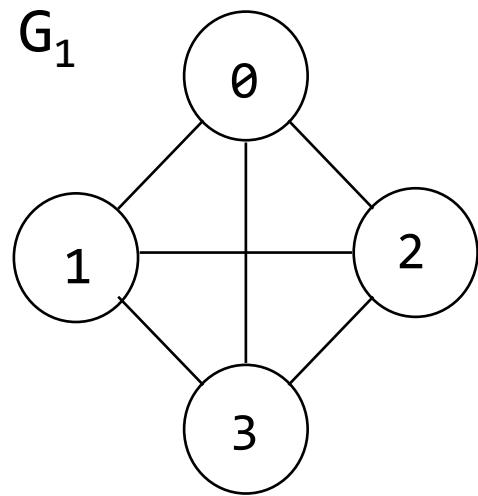


A **subgraph** of G is a graph G' such that
 $V(G') \subseteq V(G)$ and $E(G') \subseteq E(G)$

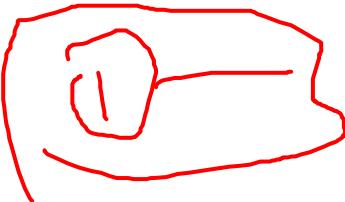
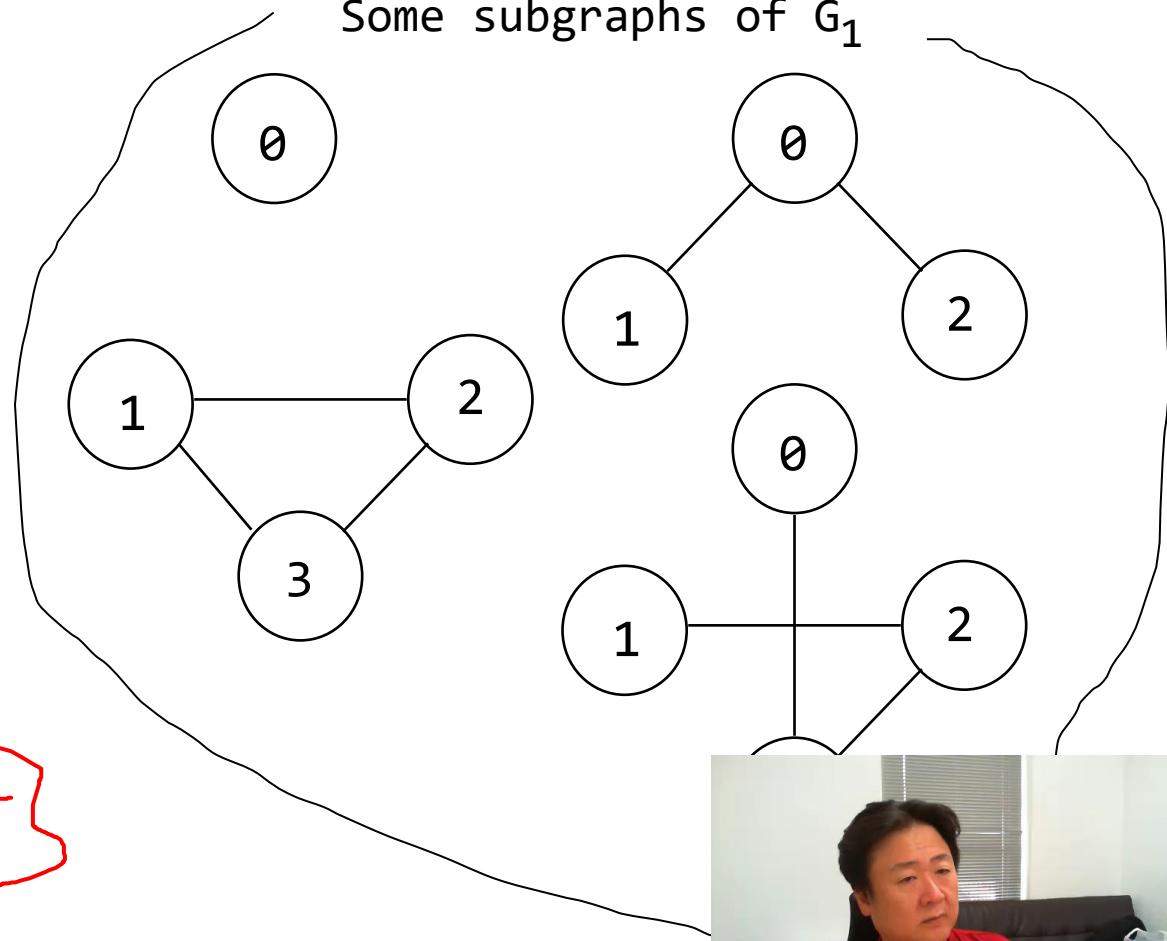


Terminology (3)

G_1



Some subgraphs of G_1



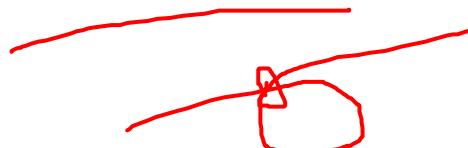
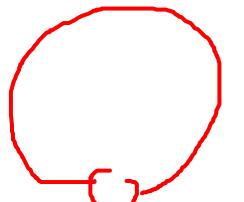
Terminology (4)

A **path** from vertex u to v in graph G is a sequence of vertices, $u, i_1, i_2, \dots, i_k, v$ such that $(u, i_1), (i_1, i_2), \dots, (i_k, v)$ are edges in an undirected graph

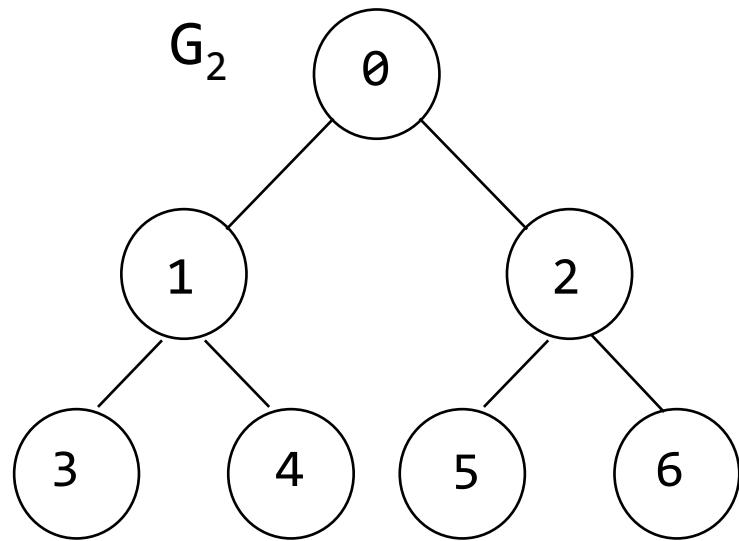
- If G' is a directed graph, the path consists of $\langle u, i_1 \rangle, \langle i_1, i_2 \rangle, \dots, \langle i_k, v \rangle$
- The length of a path is the number of the edges on it

A **simple path** is a path in which all vertices, except possibly the first and the last, are distinct

A **cycle** is a simple path in which the first and the last vertices are the same

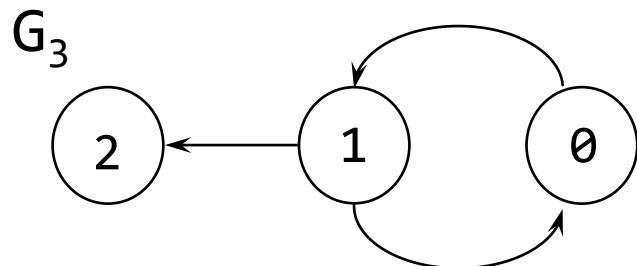


Terminology (5)



$$V(G_2) = \{0, 1, 2, 3, 4, 5, 6\}$$
$$E(G_2) = \{(0,1), (0,2), (1,3), (1,4), (2,5), (2,6)\}$$

Simple path: 0, 2, 6
0, 1, 4
2, 5
...



$$V(G_3) = \{0, 1, 2\}$$
$$E(G_3) = \{<0,1>, <1,0>, <1,2>\}$$

Simple path: 0, 1, 2
1, 0
1, 2

Cycle: 0, 1, 0



Terminology (6)

In an undirected graph G ,

two vertices u and v are **connected** if there is a path in G from u and v

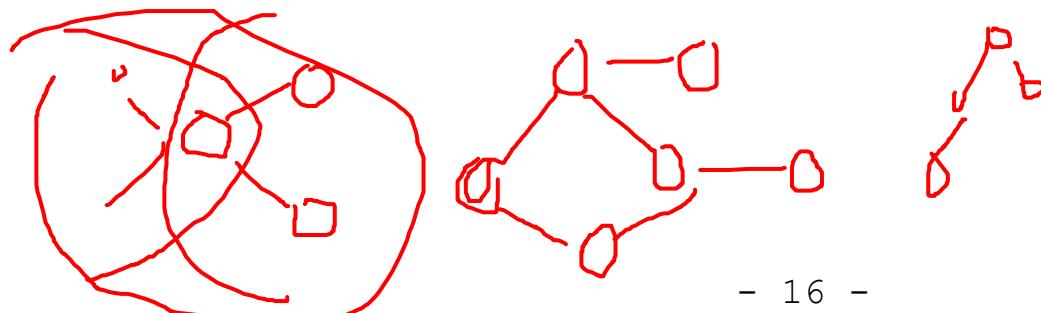
A **connected component** or simply a **component** of an undirected graph is a maximal connected subgraph

Let $G=(V,E)$ be a graph and $G_1=(V_1,E_1) \dots, G_m=(V_m, E_m)$ be its connected components

→ For all i,j , $V_i \cap V_j = \{\}$.

Further, $V = V_1 \cup \dots \cup V_m$ and $E = E_1 \cup \dots \cup E_m$

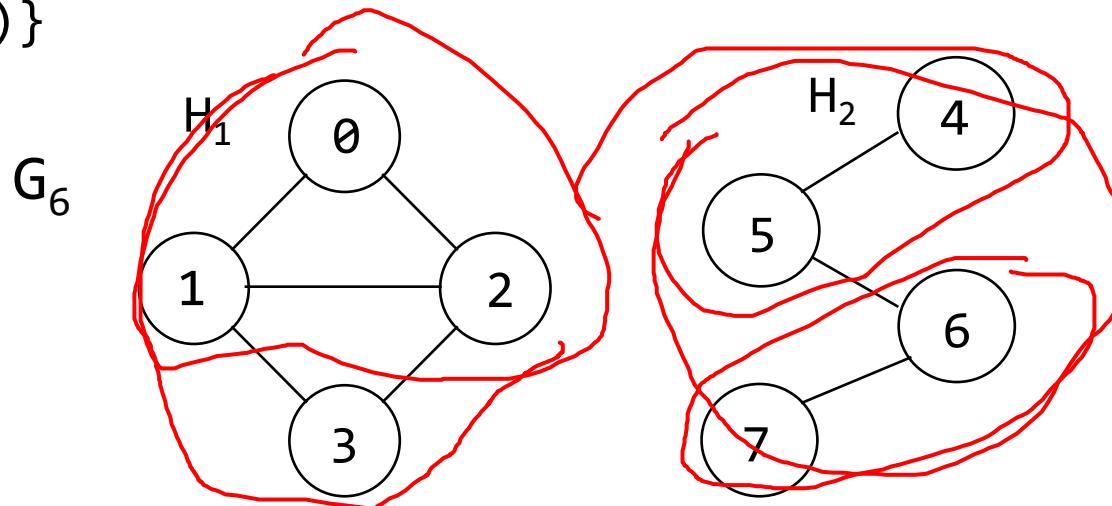
A tree is a graph that is connected and acyclic



Terminology (7)

$$V(G_6) = \{0, 1, 2, 3, 4, 5, 6, 7\}$$

$$E(G_6) = \{(0,1), (0,2), (1,2), (1,3), (2,3), (4,5), (5,6), (6,7)\}$$



Connected components

$$V(H_1) = \{0, 1, 2, 3\}$$

$$E(H_1) = \{(0,1), (0,2), (1,2), (1,3), (2,3)\}$$

$$V(H_2) = \{4, 5, 6, 7\}$$

$$E(H_2) = \{(4,5), (5,6), (6,7)\}$$



Terminology (8)

A directed graph is **strongly connected** if, for every pair of vertices, u and v in $V(G)$, there is a directed path from u and v and also from v to u

A strongly connected component is a maximal subgraph that is strongly connected



Given a directed graph $D=(V, E)$,

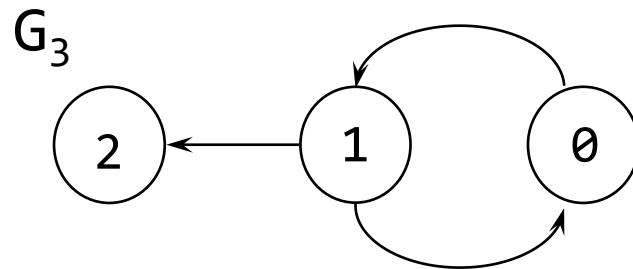
a subgraph $S=(V', E')$ is a strongly connected component if S is strongly connected, and

for all vertexes u such that $u \in V$ and $u \notin V'$

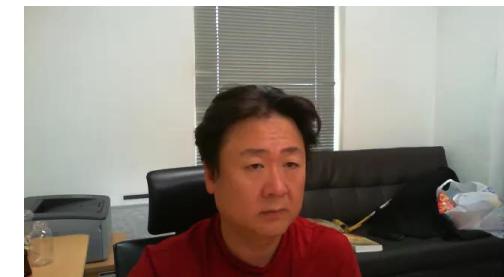
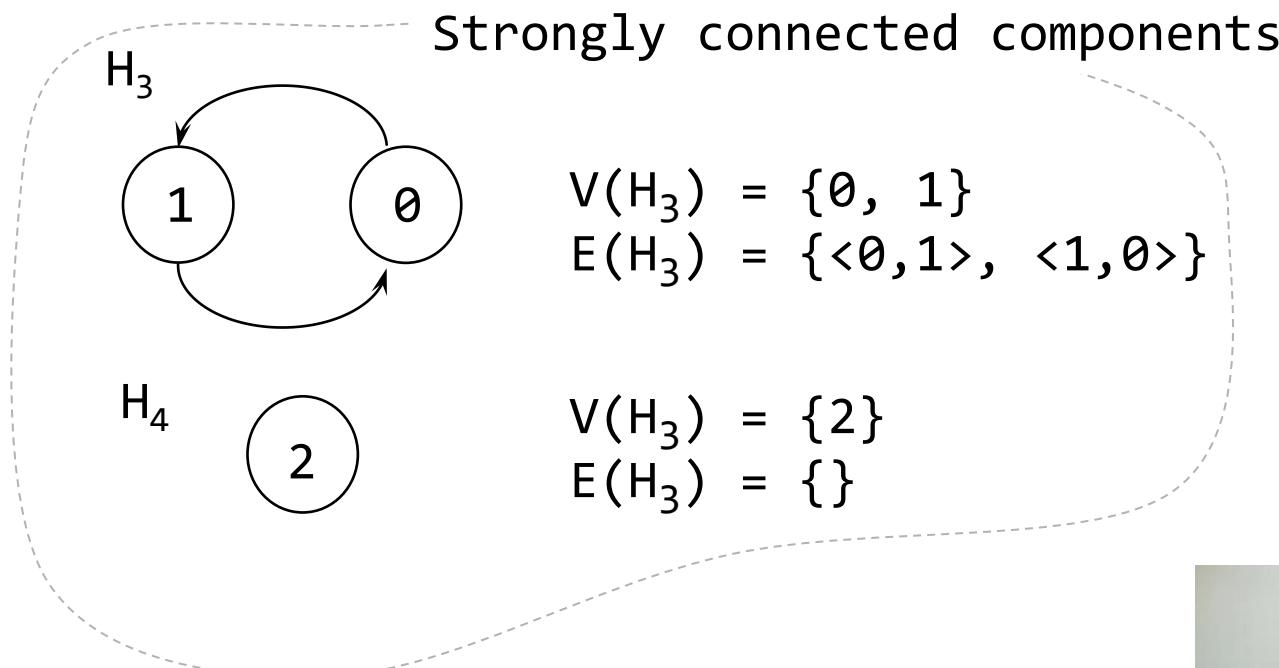
there is no vertex $v \in V'$ for which $(u, v) \in E$



Terminology (9)



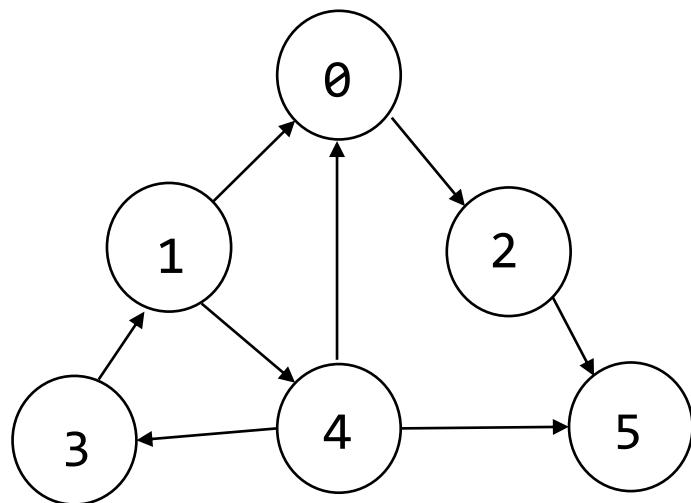
$$V(G_3) = \{0, 1, 2\}$$
$$E(G_3) = \{\langle 0, 1 \rangle, \langle 1, 0 \rangle, \langle 1, 2 \rangle\}$$



Quiz 34

Name and student ID

Draw the strongly connected components of the following graph.



Terminology (10)

The degree of a vertex is the number of edges incident to that vertex

For a directed graph,

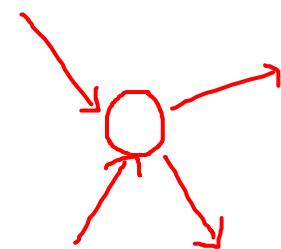
- the **in-degree** of a vertex v is defined as the number of edges that have v as the head
- the **out-degree** of a vertex v is defined as the number of edges that have v as the tail

Property

e : the number of edges

d_i : the degree of a vertex i in graph G

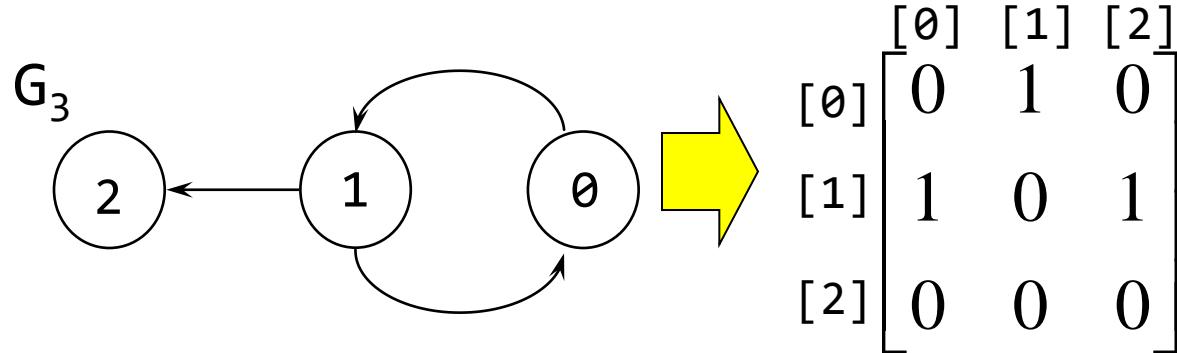
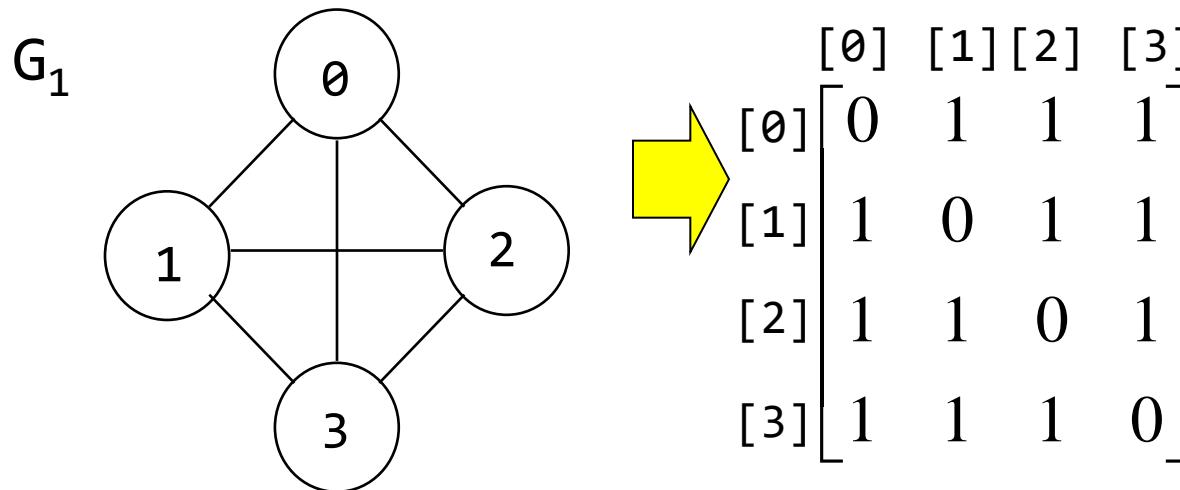
$$e = \left(\sum_{i=0}^{n-1} d_i \right) / 2$$



Adjacency Matrix

Two-dimensional $n \times n$ array, when the number of nodes is n

$$a[i][j] = \begin{cases} 1, & \text{if } (i, j) \text{ is adjacent} \\ 0, & \text{otherwise} \end{cases}$$



Property of Adjacency Matrix

The adjacency matrix for an undirected graph is symmetric

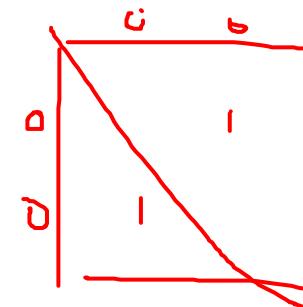
The adjacency matrix for a digraph need not be symmetric

For an undirected graph, the degree of any vertex i is its row sum

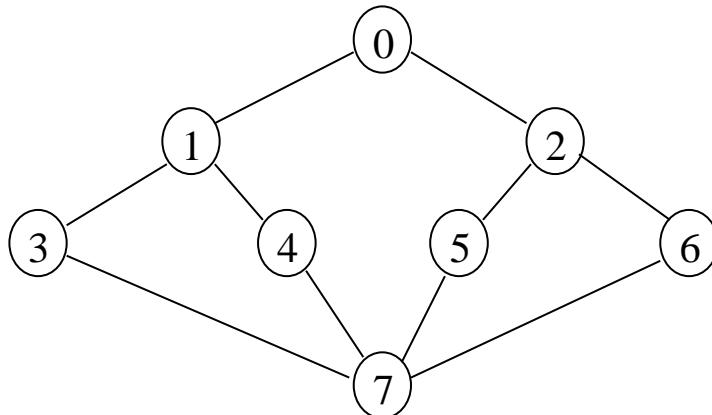
For a directed graph,

the row sum is the out-degree;

the column sum is the in-degree



Inefficiency of Adjacency Matrix



Utilization = $20/64 = 31(\%)$

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 0 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 |
| 1 | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 0 |
| 1 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 0 |
| 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 1 |
| 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 1 |
| 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 1 |
| 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 1 |
| 0 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 0 |



Adjacency Lists

Replace n rows of adjacency matrix with n linked lists

Every vertex i in G has one list

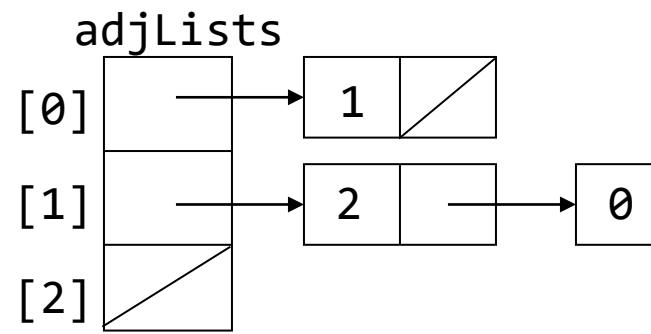
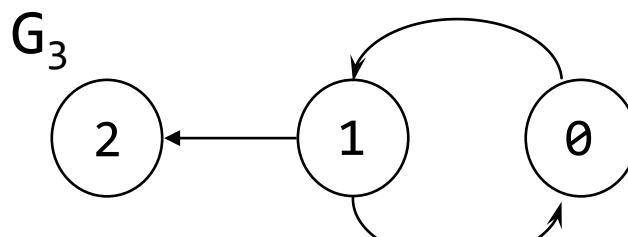
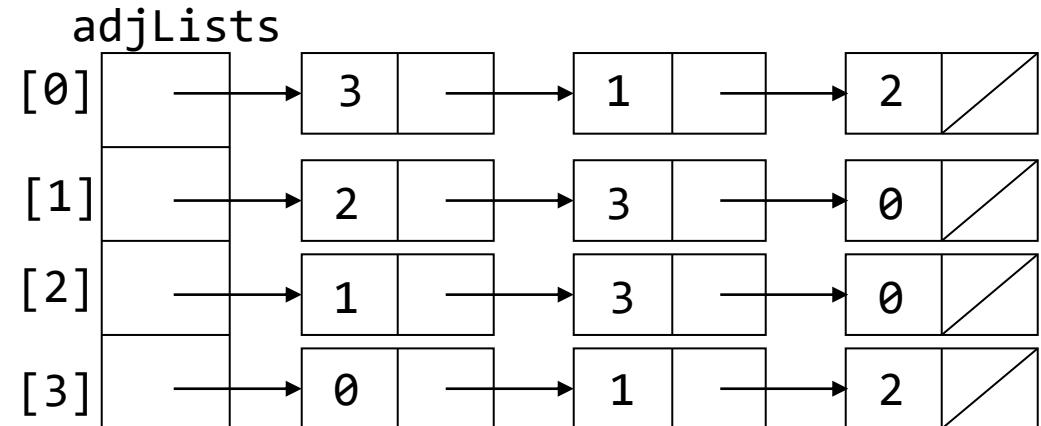
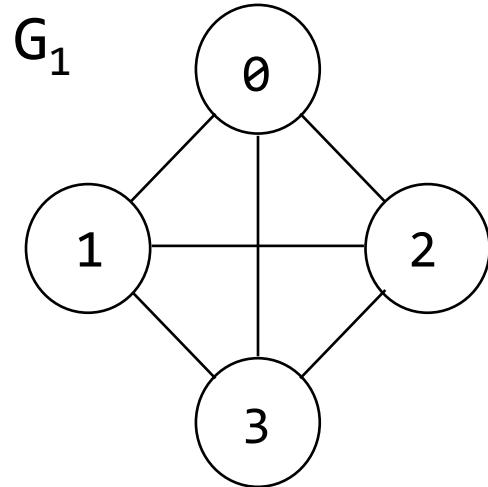
The nodes in chain i represent the vertices that are adjacent from i

The vertices in each chain are not required to be ordered

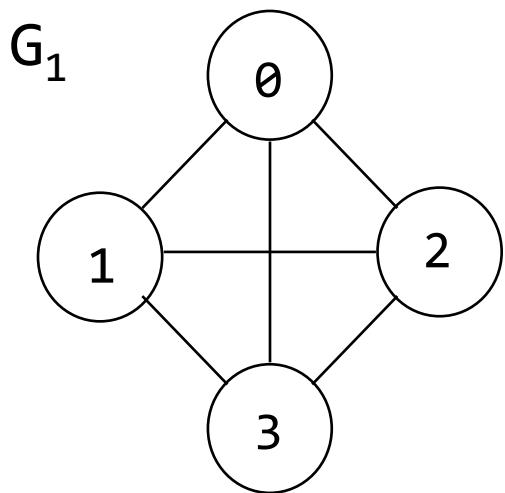


Adjacency Lists in C

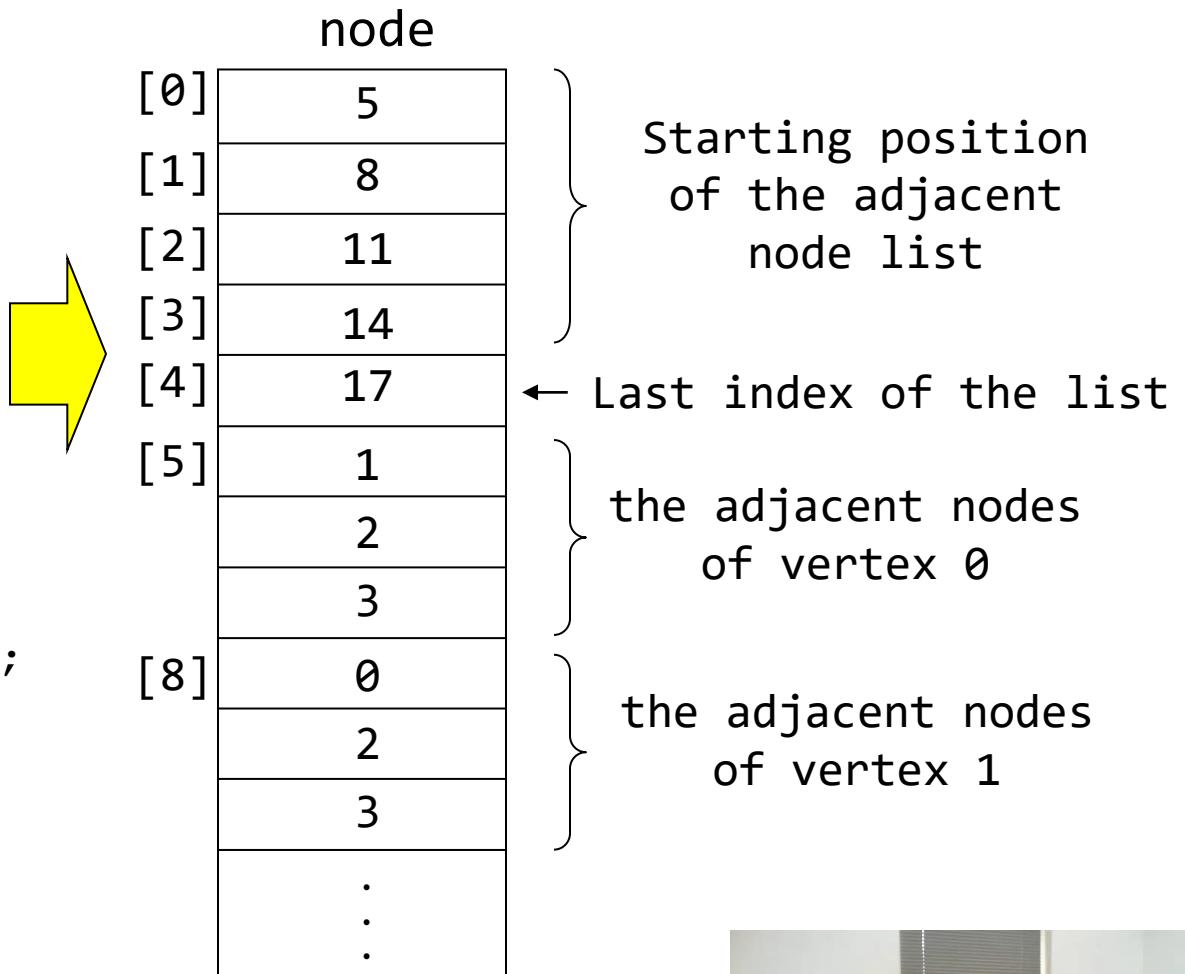
```
typedef struct node *nodePointer;
typedef struct node {
    int vertex; nodePointer link;
};
nodePointer adjLists[MAX_NODES];
```



Sequential Representation



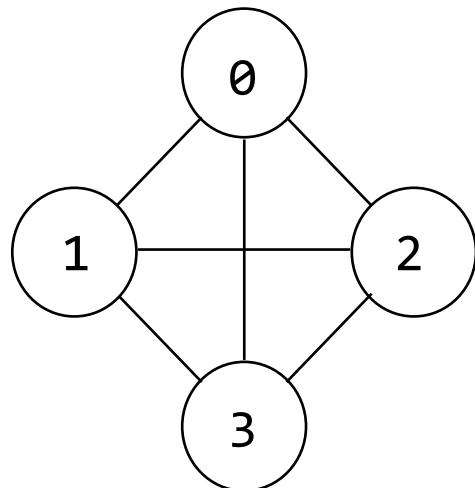
```
int nodes[n+2*e+1];
```



Quiz 35

Name and student ID

Calculate the required memory to represent the following graph using adjacency matrix, adjacency list, and sequential representation, respectively.

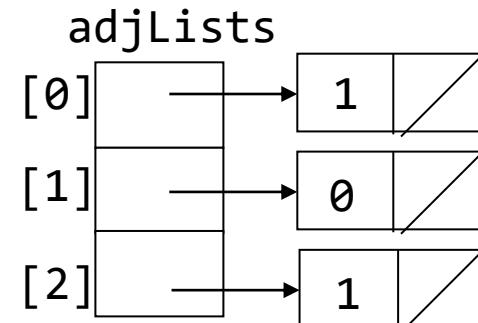
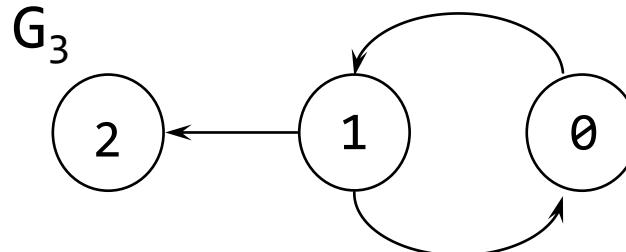


Inverse Adjacency List

Useful for finding in-degree of a vertex in digraphs

Contains one list for each vertex

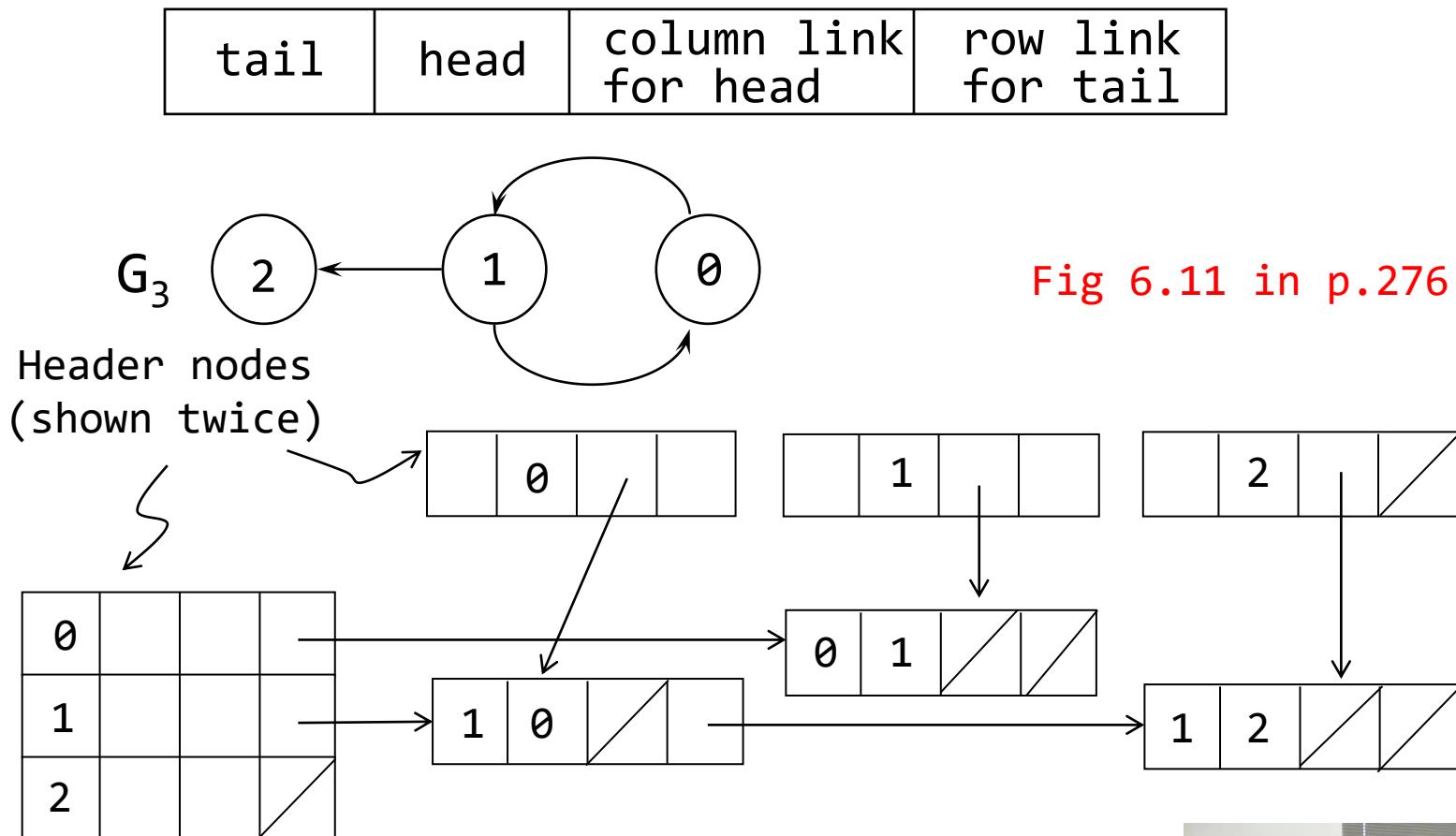
- Each list contains a node for each vertex adjacent to the vertex
- Vertices that are adjacent to vertex i



```
typedef struct node *nodePointer;
typedef struct node {
    int vertex;
    nodePointer link;
} ;
nodePointer adjLists[MAX_NODES] ;
```



Orthogonal Representation



Adjacency Multilists

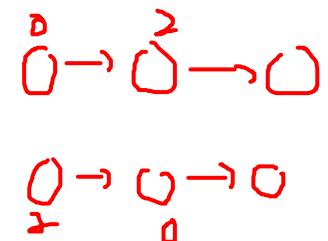
Lists in which nodes are shared among several lists

Exactly one node for each edge

The node is on the adjacency list for each of the two vertices it is incident to

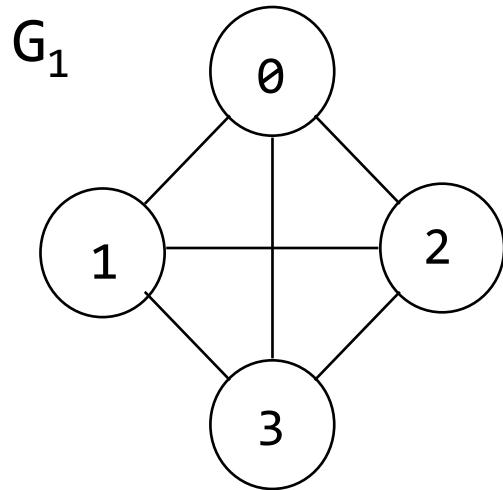
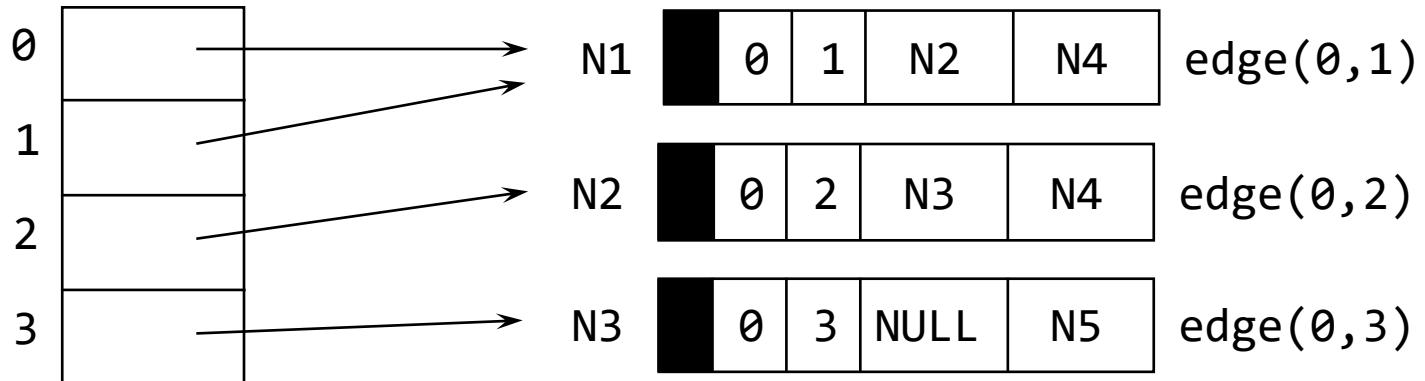
| | | | | |
|--------|---|---|-------|-------|
| marked | 0 | 2 | path1 | path2 |
|--------|---|---|-------|-------|

```
typedef struct edge *edgePointer;  
typedef struct edge {  
    short int marked;  
    int vertex1;  
    int vertex2;  
    edgePointer path1; /* next entry that has vertex1 */  
    edgePointer path2; /* next entry that has vertex2 */  
};
```



Adjacency Multilist Example

adjLists



| | | | | | |
|----|---|---|------|------|-----------|
| N4 | 1 | 2 | N5 | N6 | edge(1,2) |
| N5 | 1 | 3 | NULL | N6 | edge(1,3) |
| N6 | 2 | 3 | NULL | NULL | edge(2,3) |



Weighted Edges

Assign weights to edges of a graph

- Distance from one vertex to another, or
- Cost of going from one vertex to an adjacent vertex

Modify representation to signify an edge with the weight of the edge

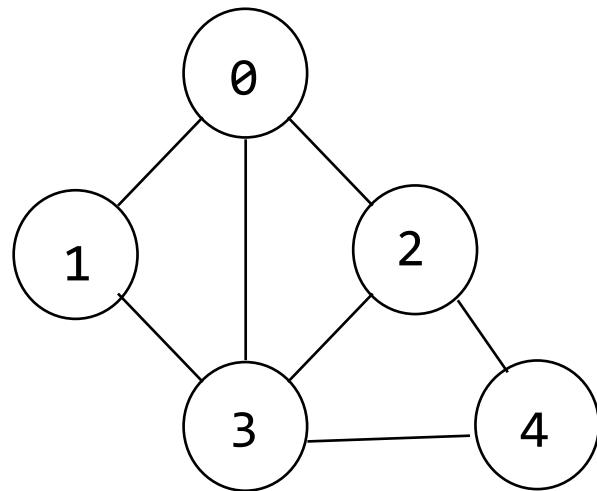
- for adjacency matrix : weight instead of 1
- for adjacency list : add weight field



Quiz 36

Name and student ID

Show the adjacency multilist for the following graph (slide 32).



ELEMENTARY OPERATIONS

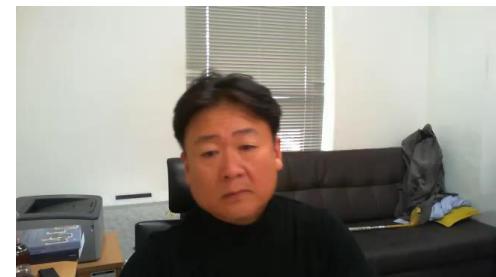
Depth First Search

Breadth First Search

Connected Components

Spanning Trees

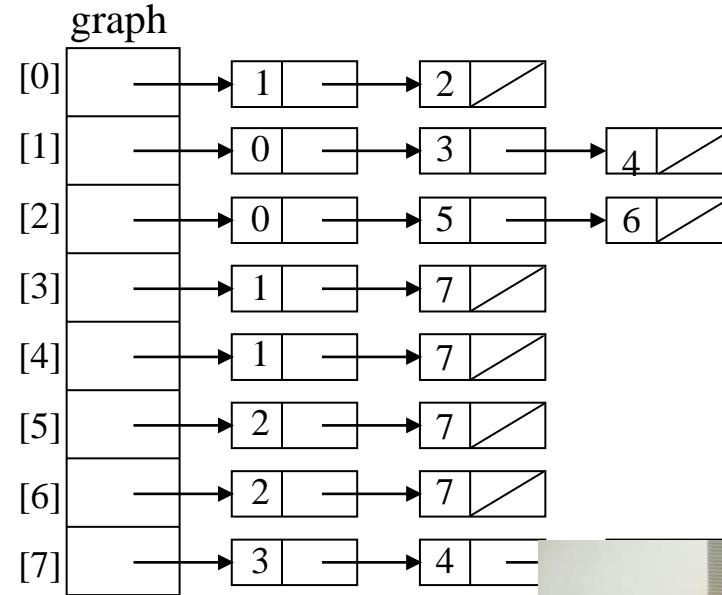
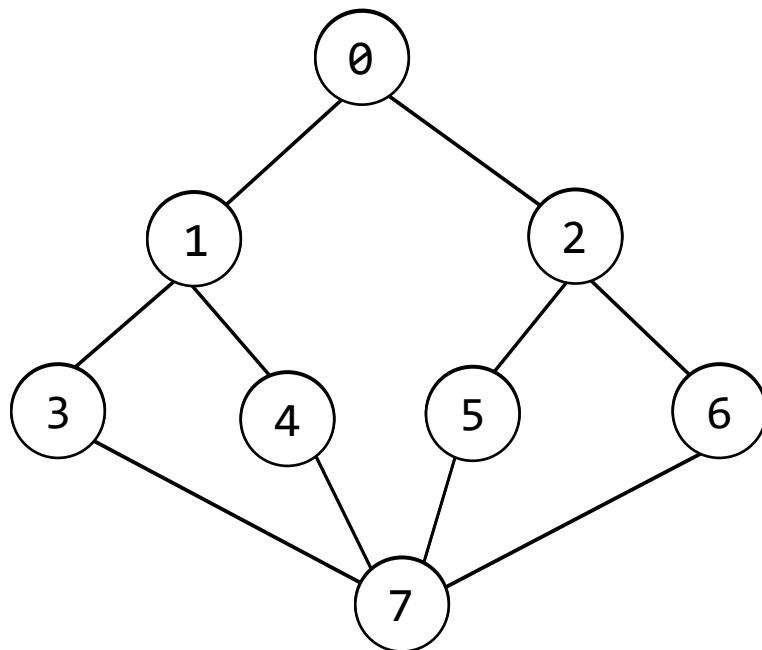
Biconnected components



Graph Traversal

Visit every vertex in a graph

- DFS (Depth First Search) - similar to a preorder tree traversal
- BFS (Breath First Search) - similar to a level-order tree traversal



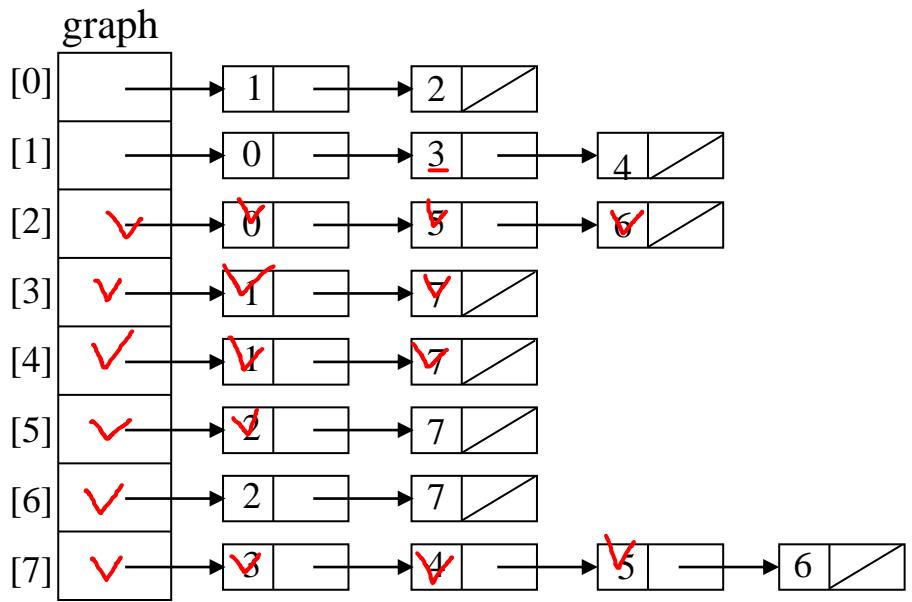
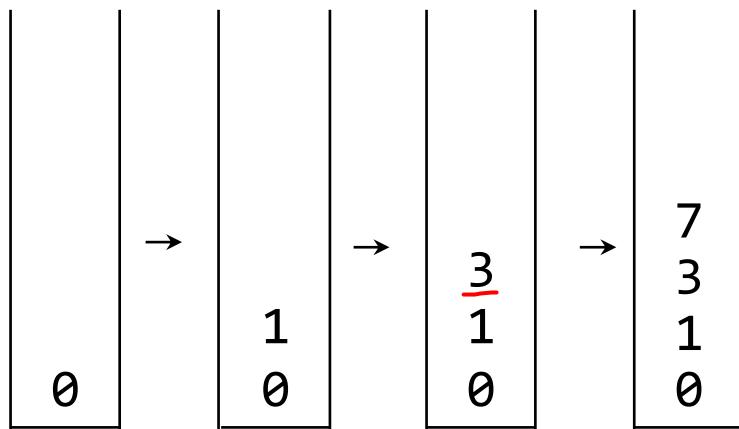
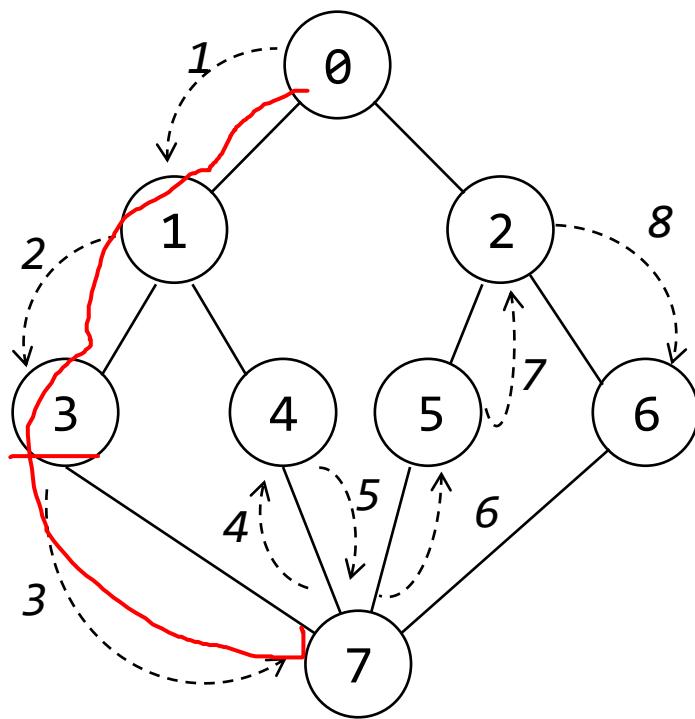
Depth First Search

```
#define FALSE 0
#define TRUE 1
short int visited[MAX_VERTICES];
nodePointer graph[MAX_VERTICES];

void dfs(int v)
{
    nodePointer w;
    visited[v] = TRUE;
    printf("%5d", v);
    for (w = graph[v]; w; w = w->link) {
        if (!visited[w->vertex])
            dfs(w->vertex);
    }
}
```



Example of DFS - $\text{dfs}(0);$



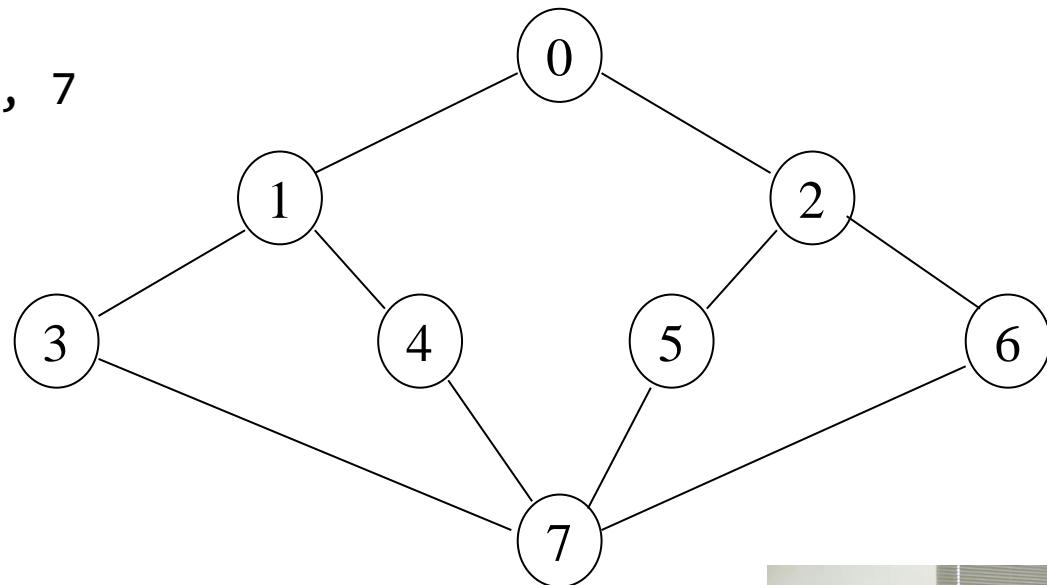
Breadth First Search (1)

BFS starts at vertex v and marks it as visited

It then visits each of vertices on v 's adjacency list

When all the vertices on v 's adjacency list are visited,
all the unvisited vertices are visited that are
adjacent to the first vertex on v 's adjacency list

0, 1, 2, 3, 4, 5, 6, 7



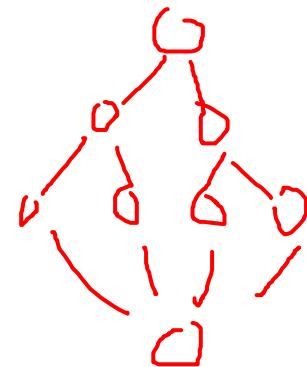
Breadth First Search (2)

```
typedef struct node *queuePointer;  
typedef struct node {  
    int vertex;  
    queuePointer link;  
};  
queuePointer front, rear;
```

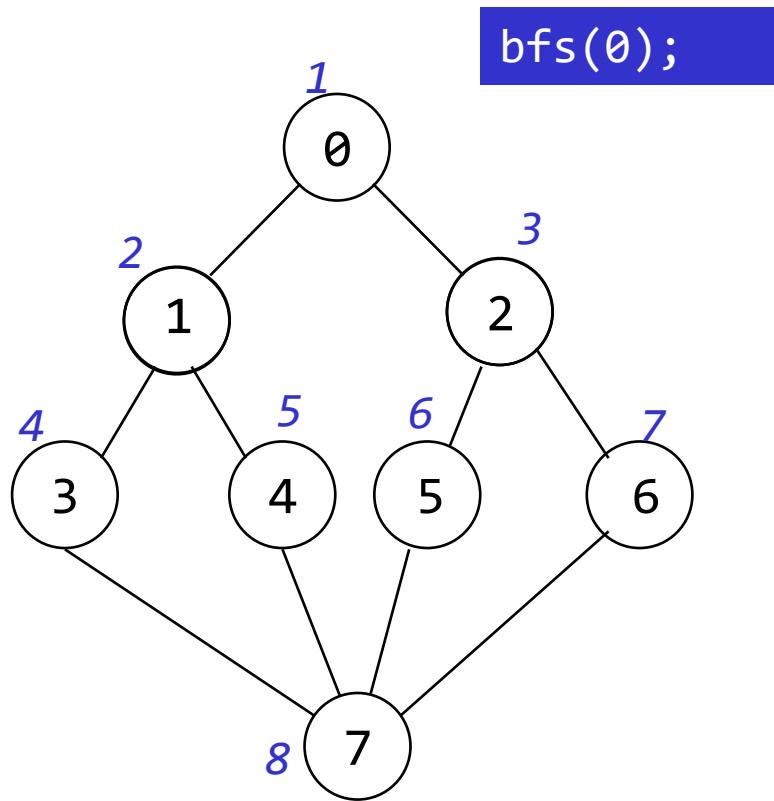


Breadth First Search (3)

```
void bfs(int v)
{
    nodePointer w;
    front = rear = NULL; /* initialize queue */
    printf("%5d", v);
    visited[v] = TRUE;
    addq(v); /* p.159, Chapter 4 */
    while (front) {
        v = deleteq(); /* p.160, Chapter 4 */
        for (w = graph[v]; w; w = w->link)
            if (!visited[w->vertex]) {
                printf("%5d", w->vertex);
                addq(w->vertex);
                visited[w->vertex] = TRUE;
            } /* if */
    } /* while */
}
```



Example of BFS – $bfs(0);$



Connected Components

To determine whether or not an undirected graph is connected

- simply calling *dfs(0)* or *bfs(0)* and then determine if there are unvisited vertices

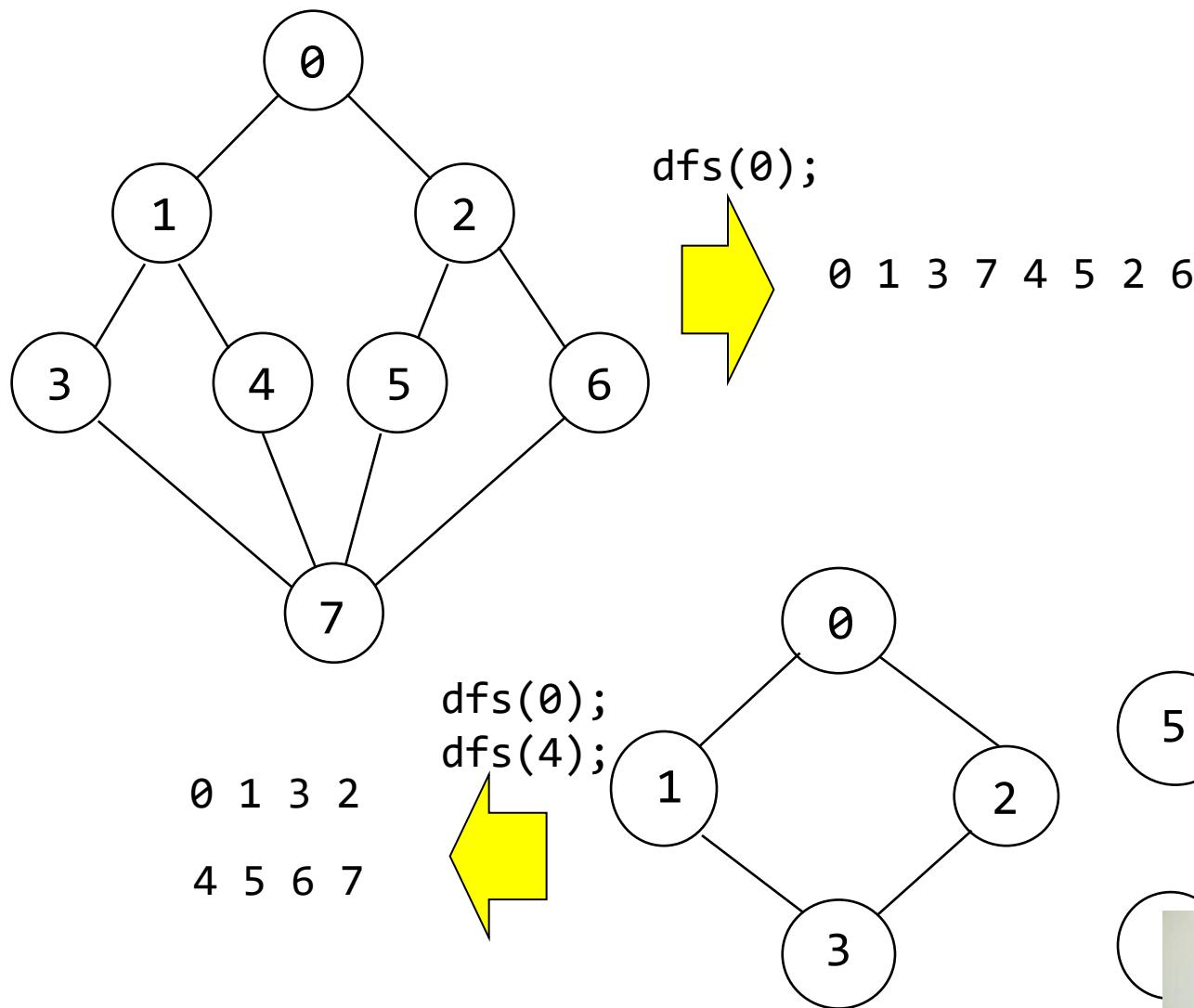
To list the connected components of a graph

- make repeated calls to either *dfs(v)* or *bfs(v)* where v is an unvisited vertex

```
void connected(void)
{ /* determine the connected components of a graph */
    int i;
    for (i = 0; i < n; i++) {
        if (!visited[i])
            dfs(i);
        printf("\n");
    }
}
```



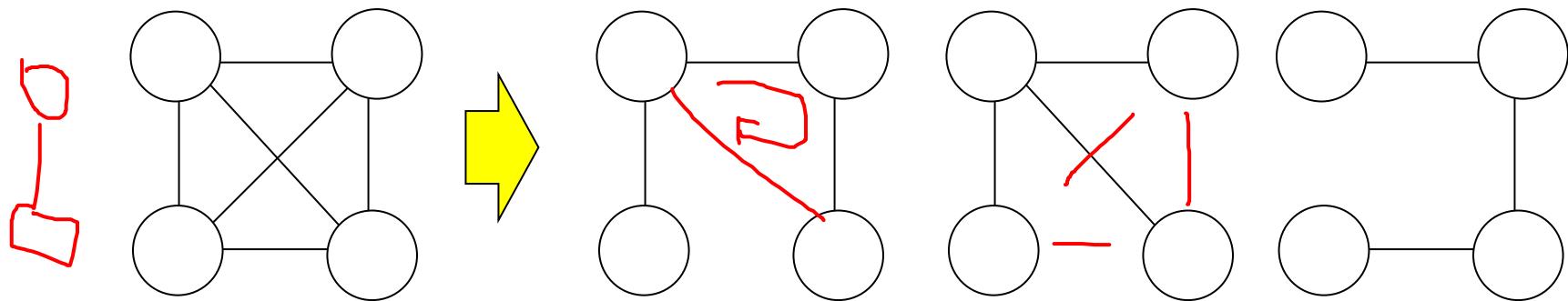
Example of Connected Components



Spanning Trees

A spanning tree is any tree that consists solely of edges in G and that include all the vertices in G

- 1) if we add a nontree edge into a spanning tree \rightarrow cycle
- 2) spanning tree is a minimal subgraph, G' , of G such that $V(G)=V(G')$ and G' is connected

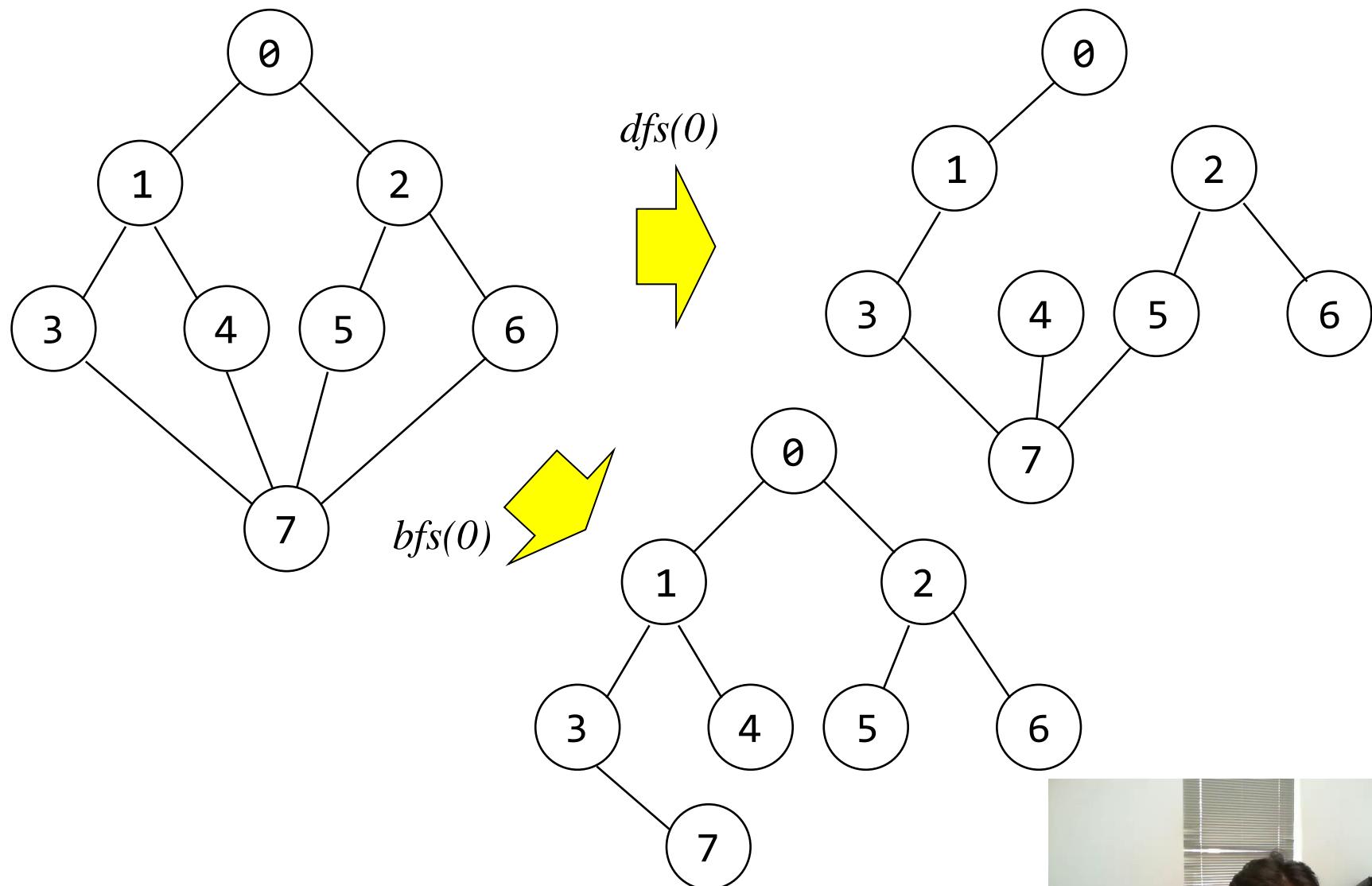


If graph G is connected, *dfs* or *bfs* implicitly partitions the edges in G into two sets:

- T : (for tree edges) set of edges used or traversed during the search
- N : (for nontree edges) set of remaining edges



Example of DFS/BFS Spanning Trees

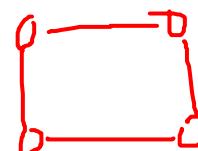


Bi-connected Components and Articulation Points

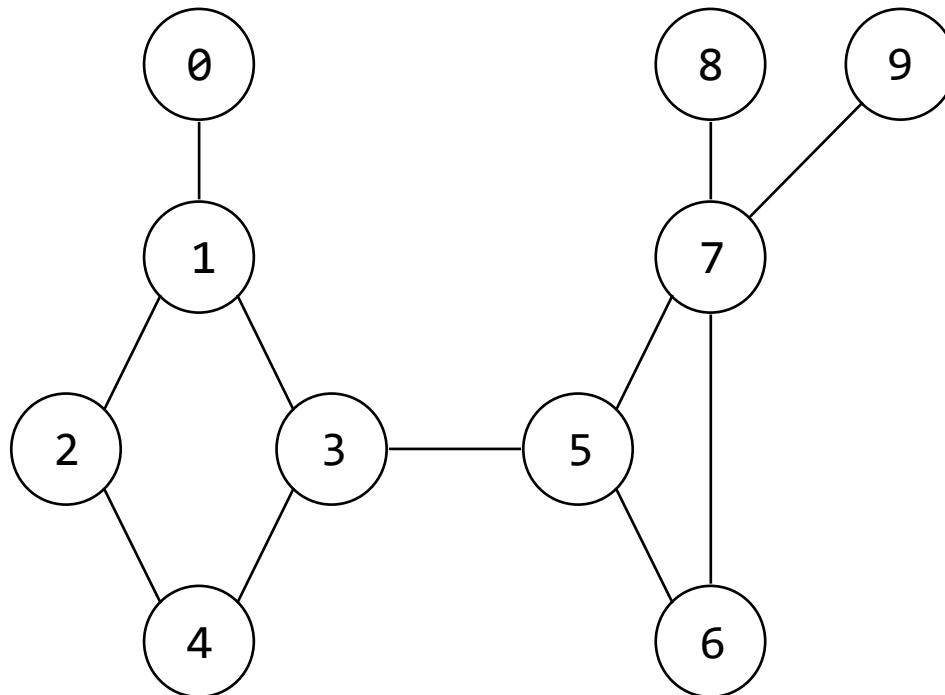
An **articulation point** is a vertex v of G such that the deletion of v , together with all edges incident on v , produces a graph, G' , that has at least two connected components

- A **bi-connected graph** is a connected graph that has no articulation point, in other words, a connected graph that is not broken into disconnected pieces by deleting any single vertex (and incident edges).
- A **bi-connected component** of a connected undirected graph is a maximal biconnected subgraph

- Between any two vertices in a bi-connected graph, there exists at least two disjoint paths
- G has a simple cycle containing the vertices



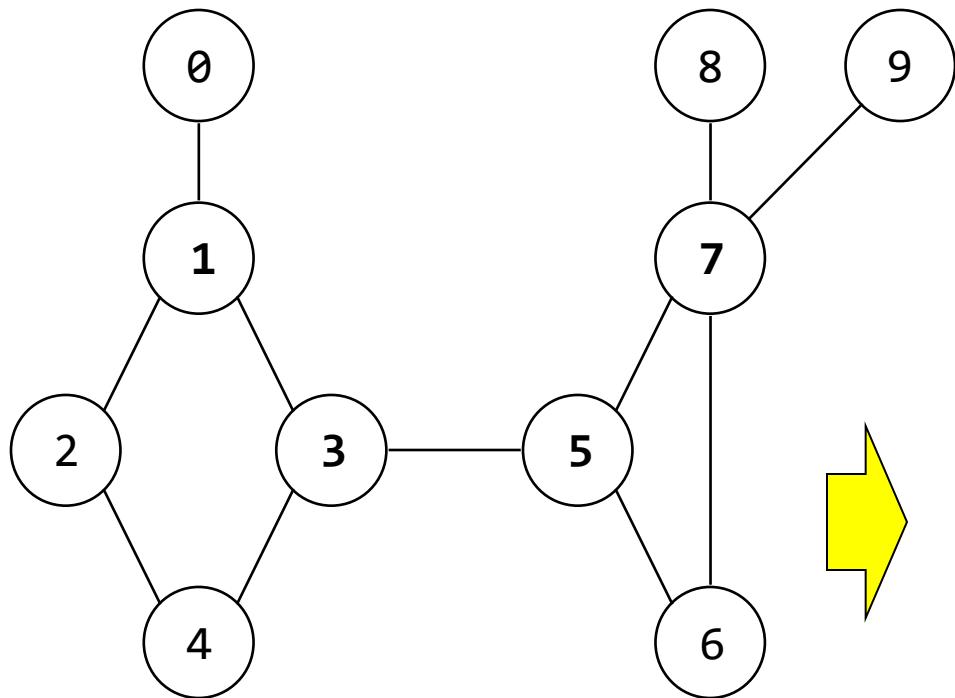
Example of Articulation Point



Connected graph

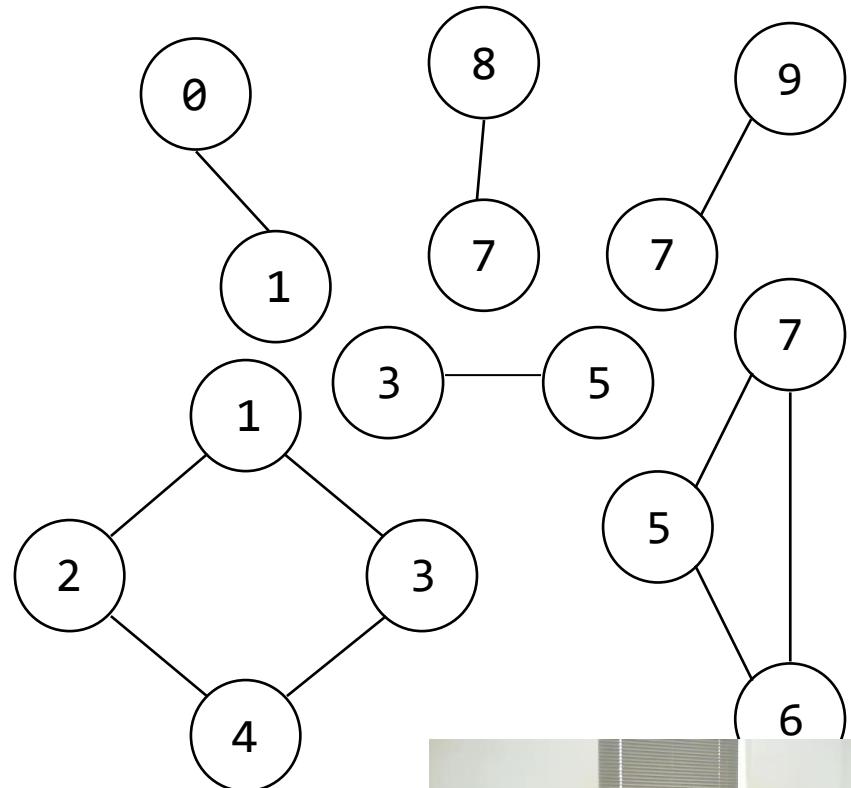


Example of Biconnected Components



Connected graph

Biconnected components



Finding the Biconnected Components (1)

Using any depth first search spanning tree of G ,

- 1) Produce a dfs spanning tree starting from any vertex
- 2) Assign a depth first number that gives the sequence in which the vertices are visited during the depth first search

→ If u is an ancestor of v in the depth first spanning tree, $\text{dfn}(u) < \text{dfn}(v)$

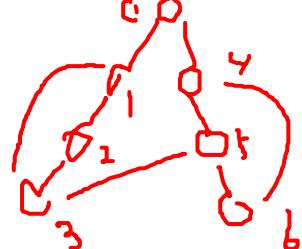
3) A nontree edge (u, v) is a **back edge** iff either u is an ancestor of v or v is an ancestor of u

4) Determine the lowest depth first number (low) that we can reach from a vertex u using a path of descendants followed by at most one back edge

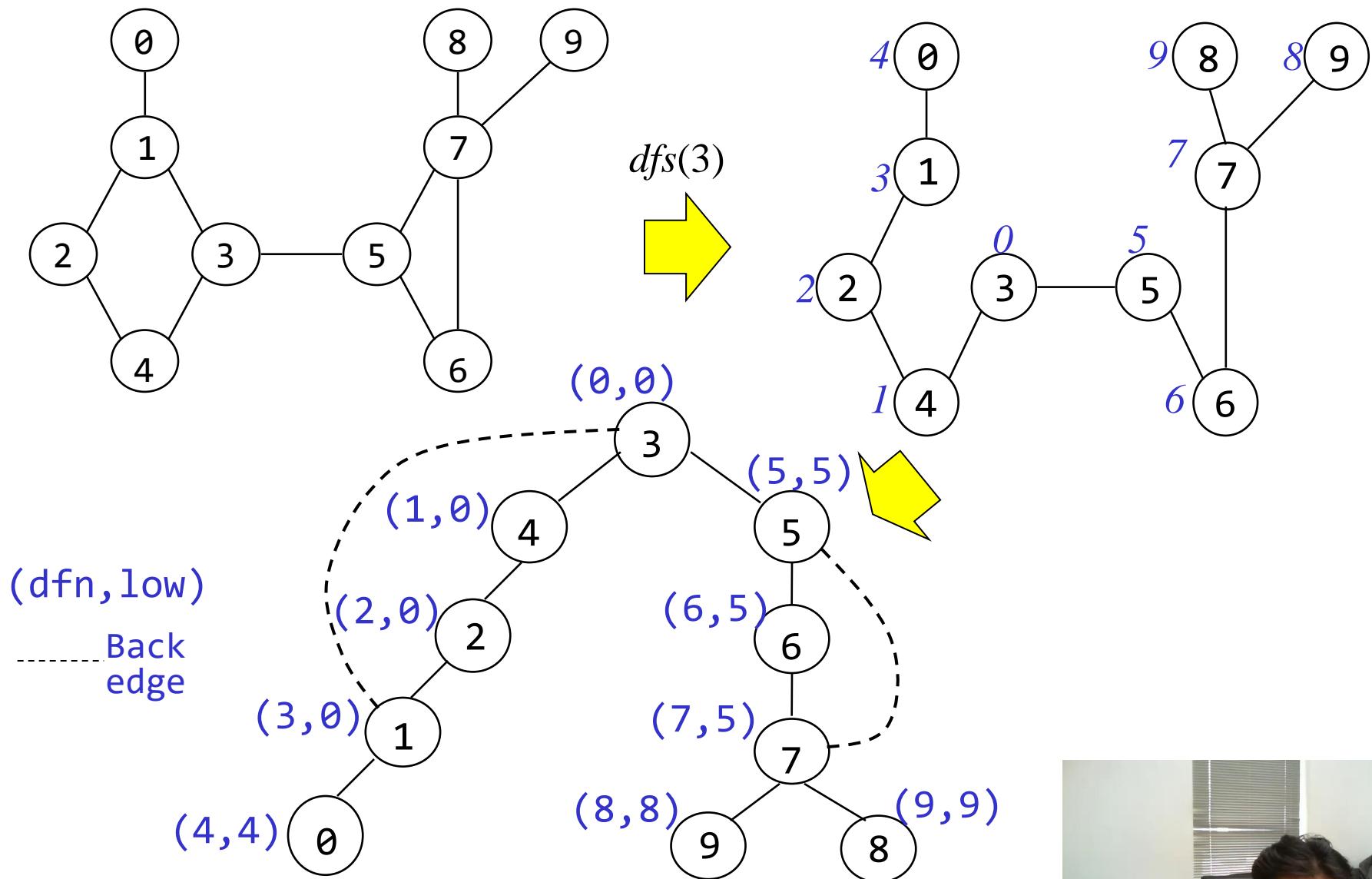
$$\underline{\underline{\text{low}}}(u) = \min\{\underline{\underline{\text{dfn}}}(u),$$

$$\min\{\underline{\underline{\text{low}}}(w) \mid w \text{ is a } \underline{\underline{\text{child of}}} u\},$$

$$\min\{\underline{\underline{\text{dfn}}}(w) \mid (u, w) \text{ is a } \underline{\underline{\text{back edge}}}\}\}$$



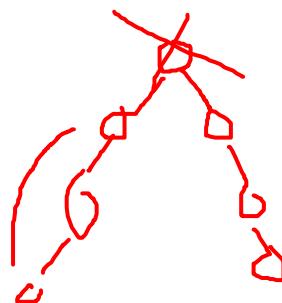
Example of DFN and LOW



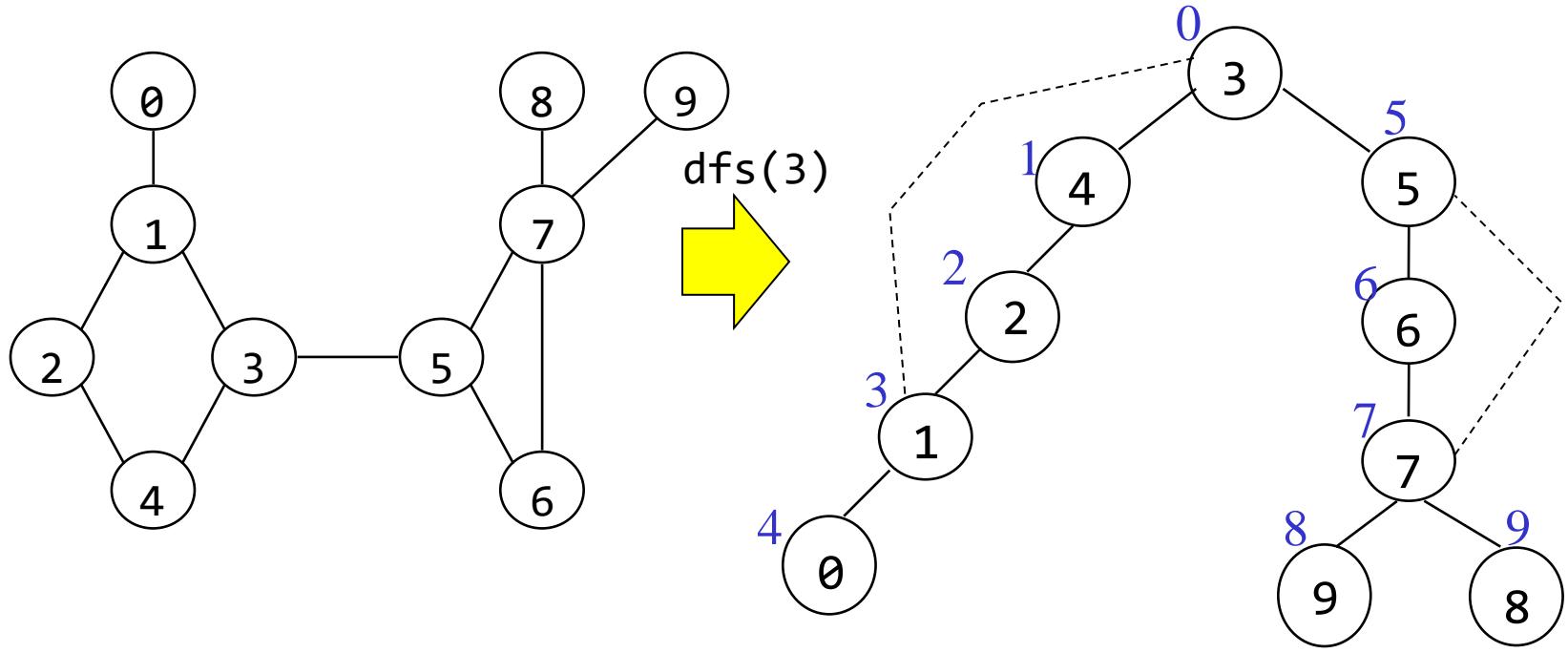
Finding the Biconnected Components (2)

Articulation point

- The root of a depth first spanning tree is an articulation point iff it has at least two children
- Any vertex u other than the root is an articulation point iff it has at least one child w such that we can not reach an ancestor of u using a path that consists of only w , descendants of w , and a single back edge
→ u is not the root and u has child w such that $low(w) \geq dfn(u)$



Example of DFN and LOW



| vertex | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|--------|---|---|---|---|---|---|---|---|---|---|
| dfn | 4 | 3 | 2 | 0 | 1 | 5 | 6 | 7 | 9 | 8 |
| low | 4 | 0 | 0 | 0 | 0 | 5 | 5 | 5 | 9 | 8 |

Articulation points



MINIMUM COST SPANNING TREES



Minimum Cost Spanning Tree

Minimum cost spanning tree

- A spanning tree of least cost
- Kruskal's, Prim's, and Sollin's algorithms

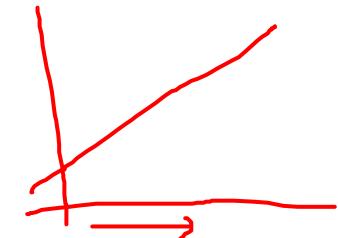


Greedy method

- Make the best decision, local optimum, at each stage using some criterion
- When the algorithm terminates, we hope that the local optimum is equal to the global optimum.

Spanning tree construction constraints

- use only edges within the graph
- use exactly $n-1$ edges
- may not use edges that would produce a cycle



Applications

- Network design: telephone, electrical, water



Kruskal's Algorithm

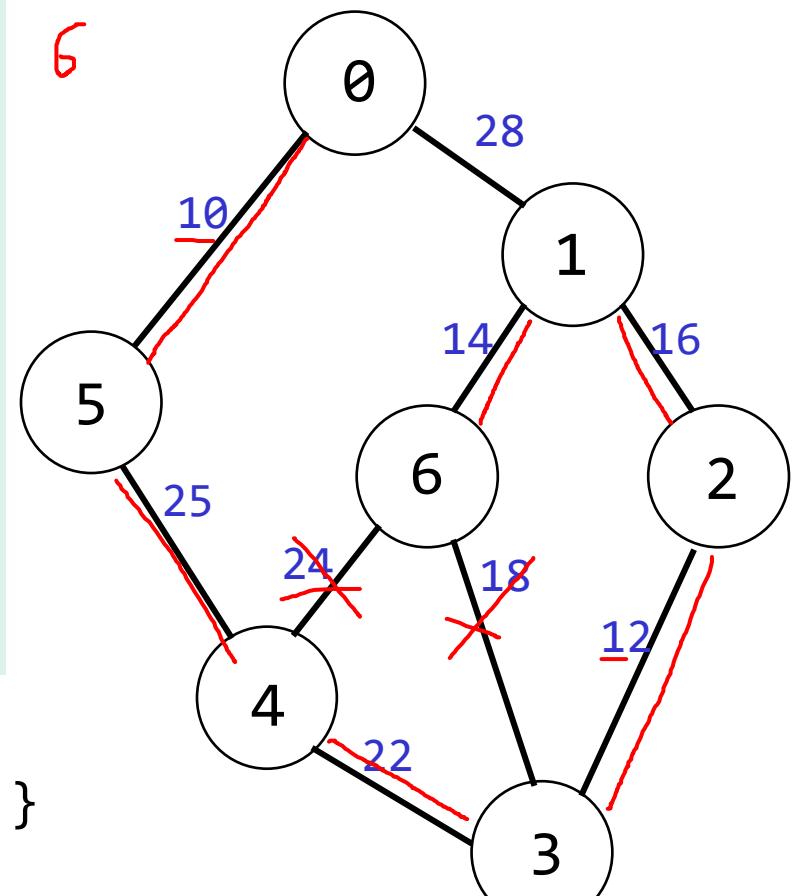
```
T = {};  
while (T contains less than (n-1) edges &&  
      E is not empty)  
{  
    choose a least cost edge (v,w) from E;  
    delete (v,w) from E;  
    if ((v,w) does not create a cycle in T)  
        add(v,w) to T;  
    else  
        discard (v,w);  
}  
if (T contains fewer than (n-1) edges)  
printf("no spanning tree\n");
```



An Example of Kruskal's Algorithm

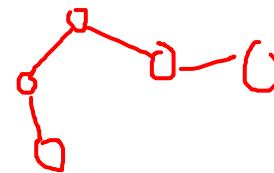
```
T = {};  
while (T contains less than (n-1) edges &&  
      E is not empty)  
{  
    choose a least cost edge (v,w) from E;  
    delete (v,w) from E;  
    if ((v,w) does not create a cycle in T)  
        add(v,w) to T;  
    else  
        discard (v,w);  
}  
if (T contains fewer than (n-1) edges)  
printf("no spanning tree\n");
```

T = {(0,5)(2,3)(1,6)(1,2)(3,4) (5,4)}



Prim's Algorithm

```
T = {};  
TV = {0}; /* start with vertex 0 and no edge*/  
while (T contains fewer than n-1 edges)  
{  
    let (u,v) be a least cost edge such that u ∈ TV  
        and v ∉ TV;  
    if (there is no such edge) break;  
    add v to TV;  
    add (u,v) to T;  
}  
if (T contains fewer than n-1 edges)  
printf("no spanning tree\n");
```

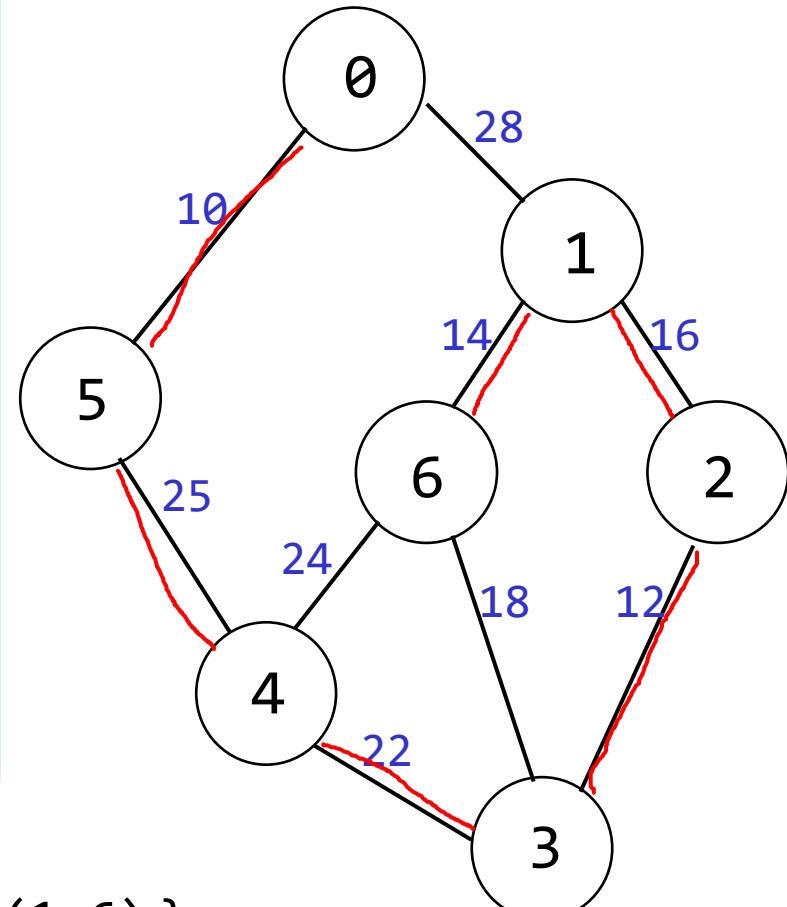


An Example of Prim's Algorithm

```
T = {};
TV = {0};
while (T contains fewer than n-1 edges)
{
    let (u,v) be a least cost edge such
    that u ∈ TV and v ∉ TV;
    if (there is no such edge)
        break;
    add v to TV;
    add (u,v) to T;
}
if (T contains fewer than n-1 edges)
    printf("no spanning tree\n");
```

TV={ 0 5 4 3 2 1 6 }

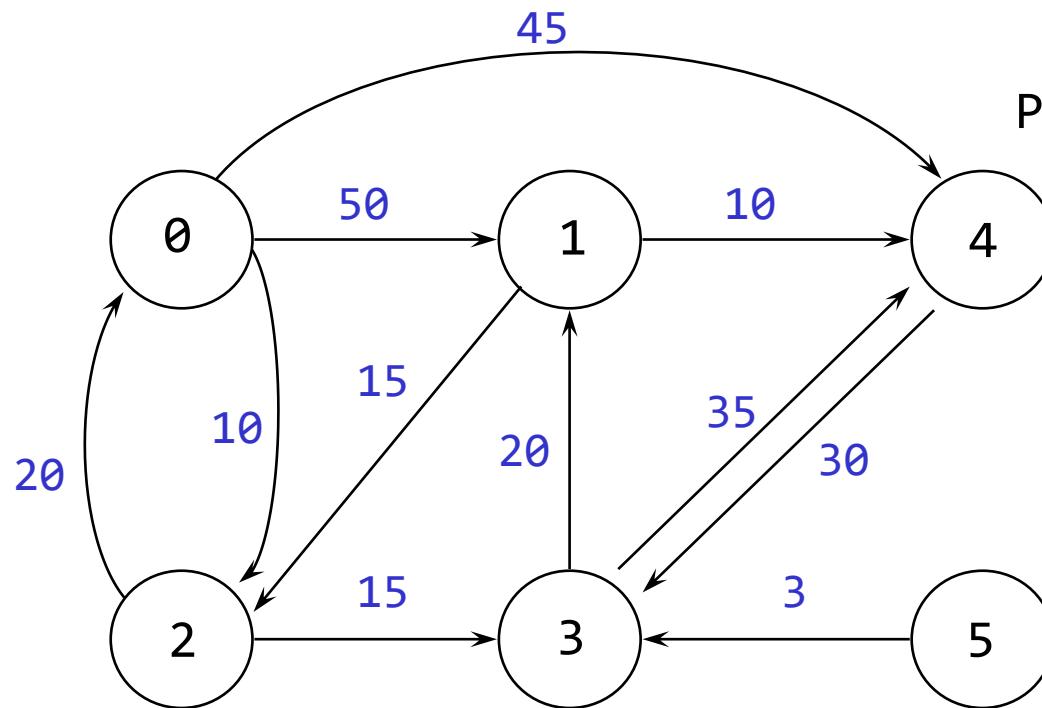
T={(0,5) (5,4) (4,3) (3,2) (2,1) (1,6) }



SHORTEST PATHS FINDING



Shortest Path Example



Path from vertex 0 Length

- | | |
|------------|----|
| 1) 0 2 | 10 |
| 2) 0 2 3 | 25 |
| 3) 0 2 3 1 | 45 |
| 4) 0 4 | 45 |



Dijkstra's Algorithm Basic Outline

S : the set of vertices whose shortest paths from the source have already been determined

$V-S$: the remaining vertices.

d : array of best estimates of shortest path to each vertex

π : an array of predecessors for each vertex

The basic mode of operation is:

Initialize d and π ,

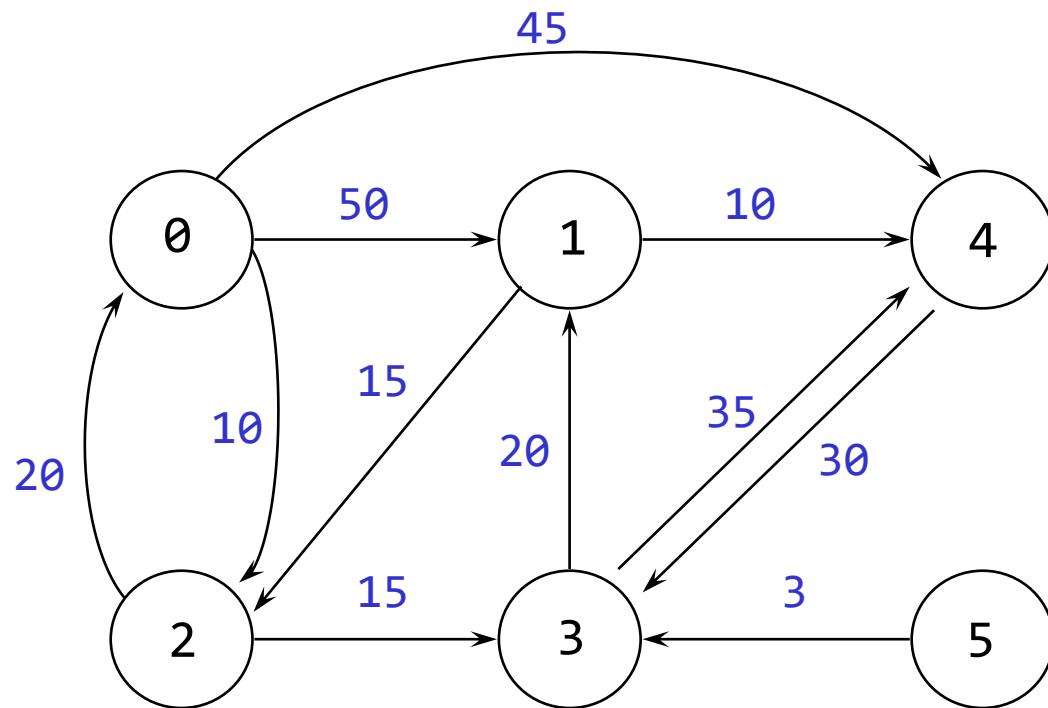
Set S to empty,

While there are still vertices in $V-S$,

- Sort the vertices in $V-S$ according to the current best estimate of their distance from the source
- Add u , the closest vertex in $V-S$, to S
- Update the costs of all the vertices, v , still in $V-S$ and connected to u

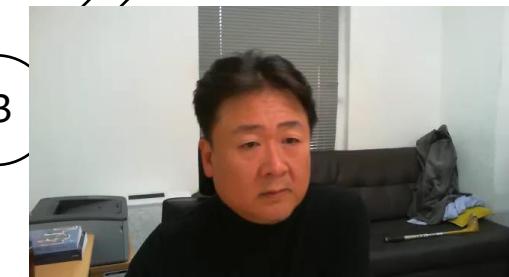
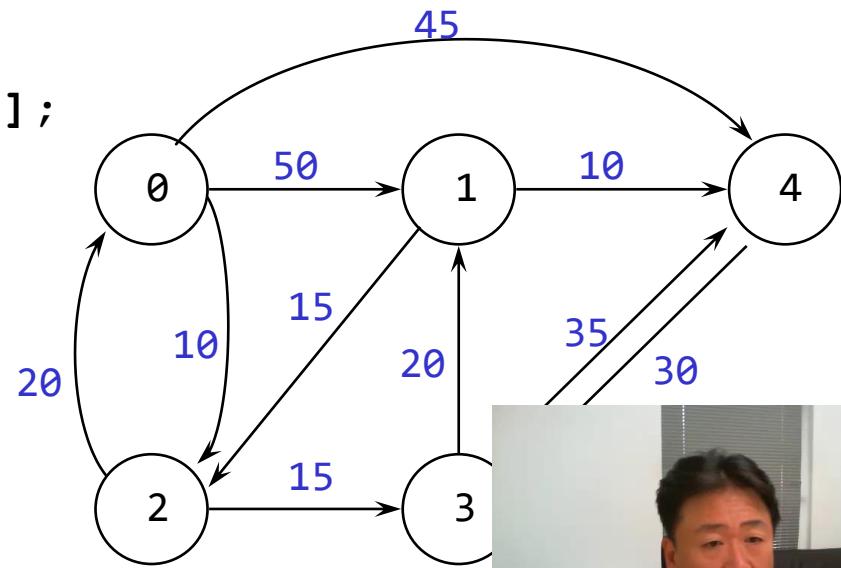


Shortest Path Example



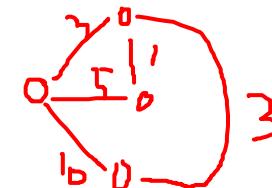
Implementation of Dijkstra's Algorithm in C (1)

```
#define MAX_VERTICES 6  
  
int cost[][][MAX_VERTICES] =  
{{ 0, 50, 10, 1000, 45, 1000},  
{1000, 0, 15, 1000, 10, 1000},  
{ 20, 1000, 0, 15, 1000, 1000},  
{1000, 20, 1000, 0, 35, 1000},  
{1000, 1000, 30, 1000, 0, 1000},  
{1000, 1000, 1000, 3, 1000, 0}};  
  
int distance[MAX_VERTICES];  
short int found[MAX_VERTICES];  
int n = MAX_VERTICES;
```



Implementation of Dijkstra's Algorithm in C (2)

```
void shortestPath(int v, int cost[][][MAX_VERTICES],
                  int distance[], int n, short int found[])
{
    int i, u, w;
    for (i = 0; i < n; i++) {
        found[i] = FALSE;
        distance[i] = cost[v][i];
    }
    found[v] = TRUE;    distance[v] = 0;
    for (i = 0; i < n-2; i++) {
        u = choose(distance, n, found);
        found[u] = TRUE;
        for (w = 0; w < n; w++)
            if (!found[w])
                if (distance[u]+cost[u][w] < distance[w])
                    distance[w] = distance[u]+cost[u][w];
    } /* for i */
}
```



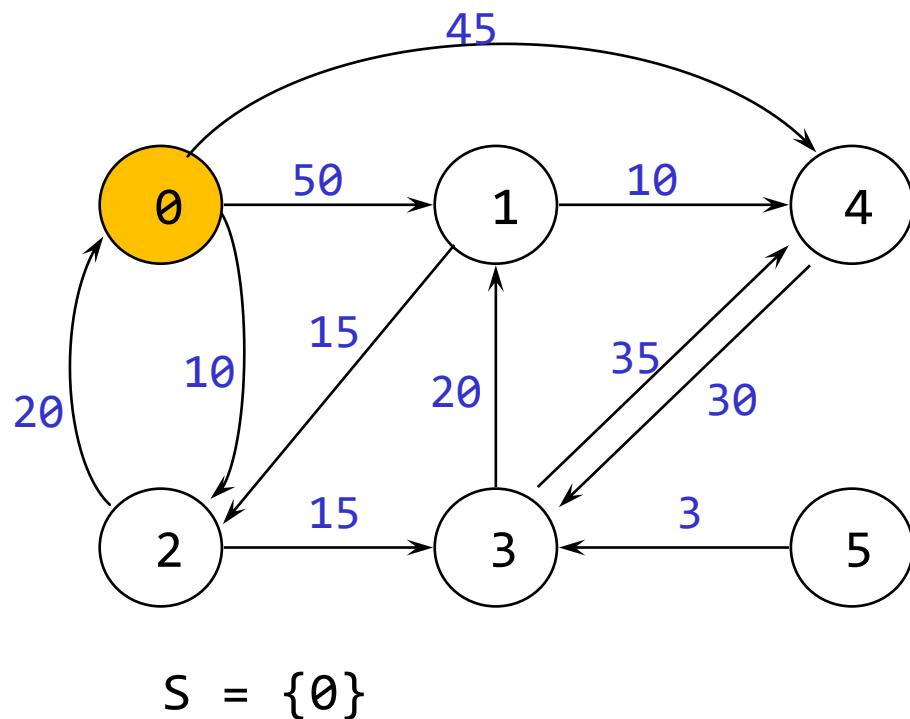
Implementation of Dijkstra's Algorithm in C (3)

```
int choose(int distance[], int n, int found[])
{
    /* find smallest distance not yet checked */
    int i, min, minpos;
    min = INT_MAX;
    minpos = -1;
    for (i = 0; i < n; i++)
        if (distance[i] < min && !found[i]) {
            min = distance[i];
            minpos = i;
        }
    return minpos;
}
```



Dijkstra's Algorithm Example 1

Initial condition

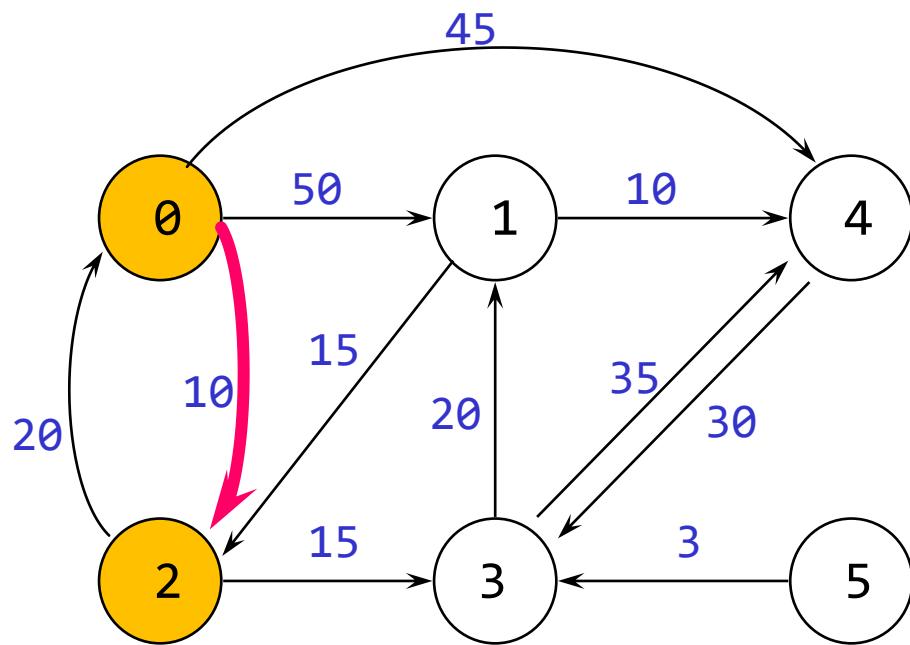


Cost adjacency matrix

| [0] | [1] | [2] | [3] | [4] | [5] | |
|-----|----------|----------|----------|----------|----------|----------|
| [0] | 0 | 50 | 10 | ∞ | 45 | ∞ |
| [1] | ∞ | 0 | 15 | ∞ | 10 | ∞ |
| [2] | 20 | ∞ | 0 | 15 | ∞ | ∞ |
| [3] | ∞ | 20 | ∞ | 0 | 35 | ∞ |
| [4] | ∞ | ∞ | ∞ | 30 | 0 | ∞ |
| [5] | ∞ | ∞ | ∞ | 3 | ∞ | 0 |



Dijkstra's Algorithm Example 1 - 1st Step



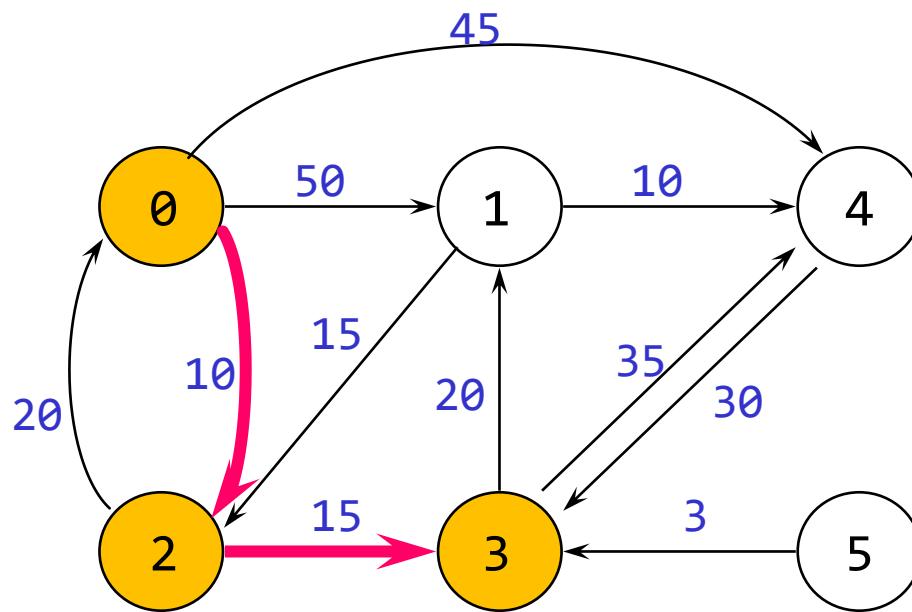
| [0] | [1] | [2] | [3] | [4] | [5] | |
|-----|----------|----------|----------|----------|----------|----------|
| [0] | 0 | 50 | 10 | ∞ | 45 | ∞ |
| [1] | ∞ | 0 | 15 | ∞ | 10 | ∞ |
| [2] | 20 | ∞ | 0 | 15 | ∞ | ∞ |
| [3] | ∞ | 20 | ∞ | 0 | 35 | ∞ |
| [4] | ∞ | ∞ | ∞ | 30 | 0 | ∞ |
| [5] | ∞ | ∞ | ∞ | 3 | ∞ | 0 |

$$S = \{0, 2\}$$

$0 \rightarrow 2: (0, 2), 10$



Dijkstra's Algorithm Example 1 - 2nd Step



| [0] | [1] | [2] | [3] | [4] | [5] | |
|-----|----------|----------|----------|----------|----------|----------|
| [0] | 0 | 45 | 10 | 25 | 45 | ∞ |
| [1] | ∞ | 0 | 15 | ∞ | 10 | ∞ |
| [2] | 20 | ∞ | 0 | 15 | ∞ | ∞ |
| [3] | ∞ | 20 | ∞ | 0 | 35 | ∞ |
| [4] | ∞ | ∞ | ∞ | 30 | 0 | ∞ |
| [5] | ∞ | ∞ | ∞ | 3 | ∞ | 0 |

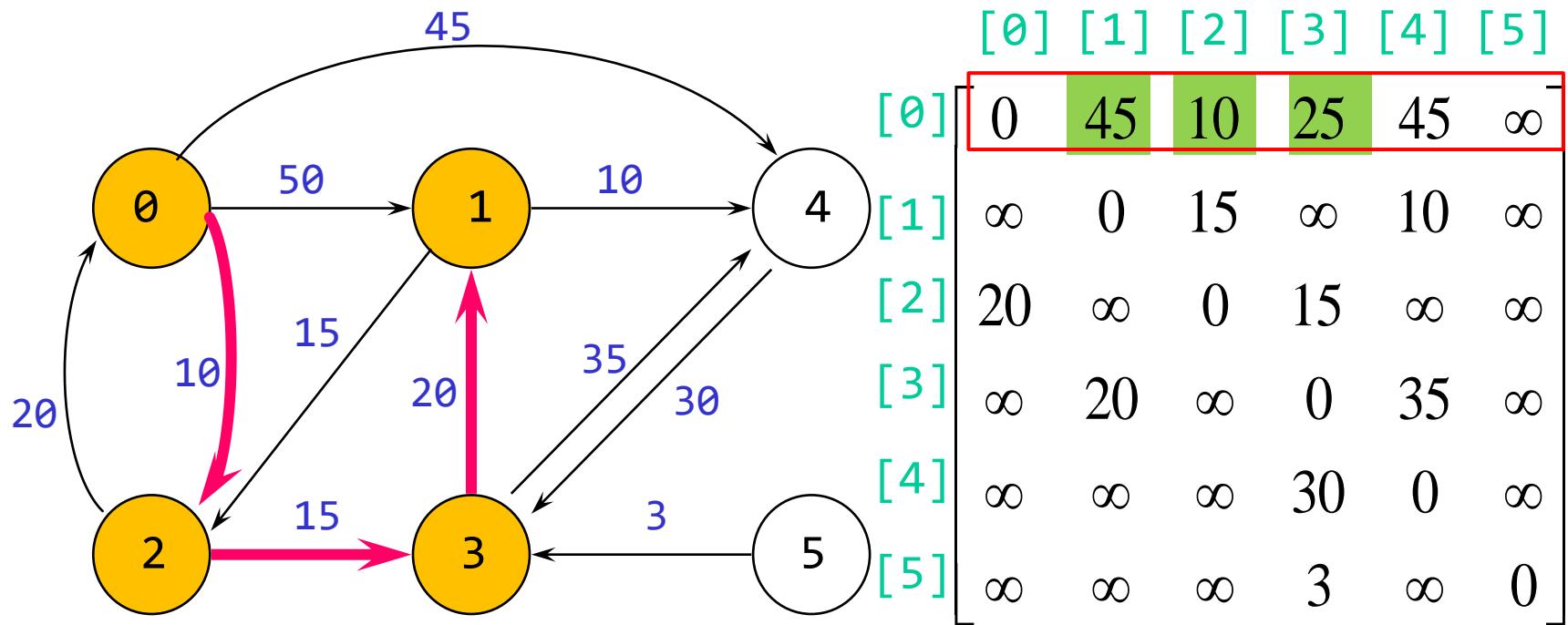
$$S = \{0, 2, 3\}$$

$0 \rightarrow 2$: $(0, 2)$, 10

$0 \rightarrow 3$: $(0, 2, 3)$, 25



Dijkstra's Algorithm Example 1 - 3rd Step



$$S = \{0, 2, 3, \underline{1}\}$$

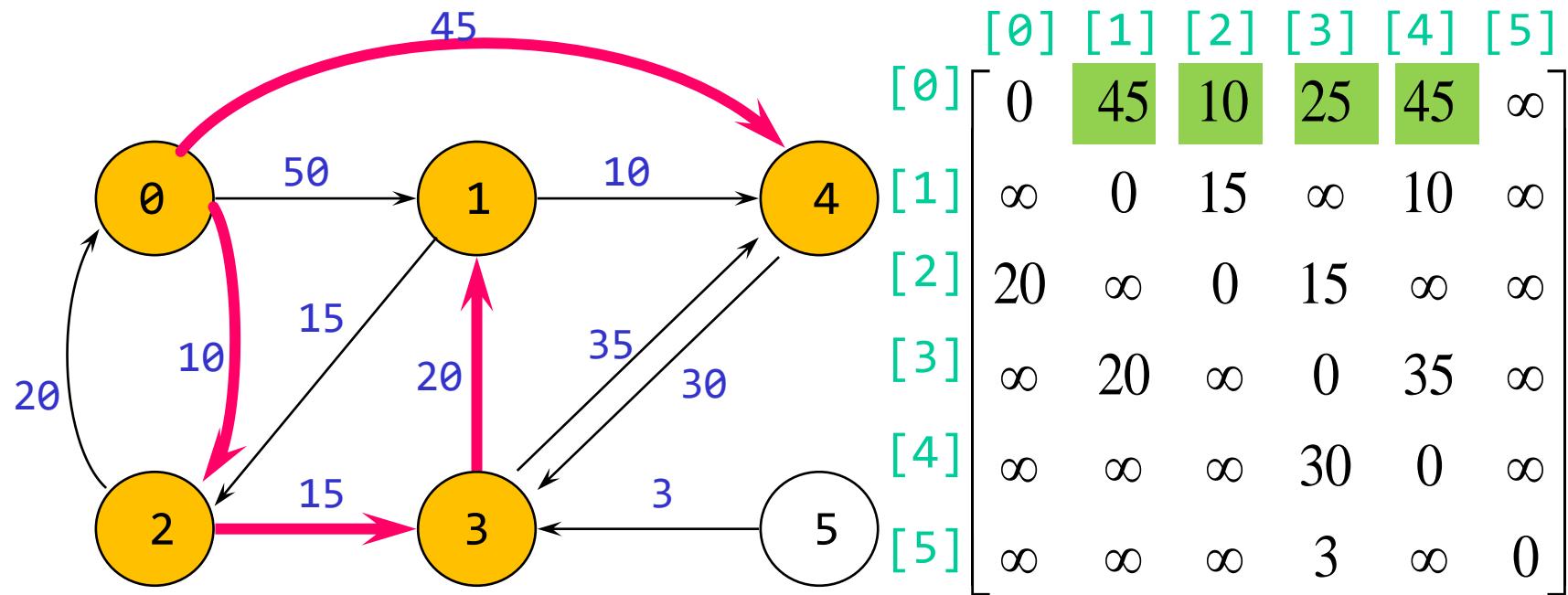
$0 \rightarrow 2: (0, 2), 10$

$0 \rightarrow 3: (0, 2, 3), 25$

$0 \rightarrow 1: (0, 2, 3, 1), 45$



Dijkstra's Algorithm Example 1 - 4th Step



$$S = \{0, 2, 3, 1, \underline{4}\}$$

$$0 \rightarrow 2: (0, 2), 10$$

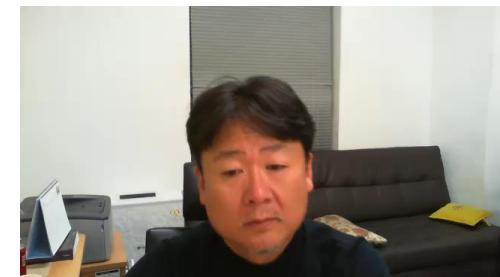
$$0 \rightarrow 3: (0, 2, 3), 25$$

$$0 \rightarrow 1: (0, 2, 3, 1), 45$$

$$0 \rightarrow 4: (0, 4),$$



Sorting



Definitions

Given a list of records $(R_0, R_1, \dots, R_{n-1})$



- R_i has a key value, K_i
- There is an ordering relation ($<$) on the keys s.t. for any two key values x and y , either $x = y$ or $x < y$ or $y < x$
- This ordering relation is transitive:

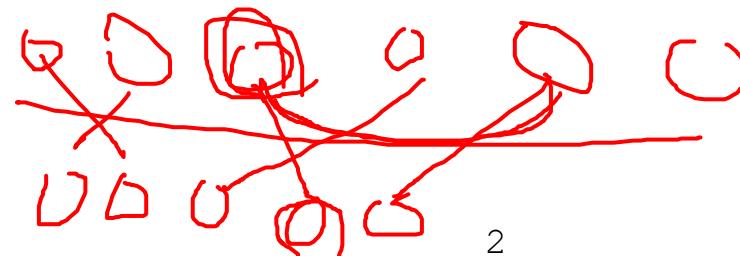
if $x < y$ and $y < z$, then $x < z$

Sorting problem is

Finding a permutation $\sigma(i)$, s.t. $K_{\sigma(i-1)} < K_{\sigma(i)}$, $0 < i \leq n-1$

A sorting algorithm is stable,

if $i < j$ and $K_i = K_j$ in the input list, R_i precedes R_j in the sorted list



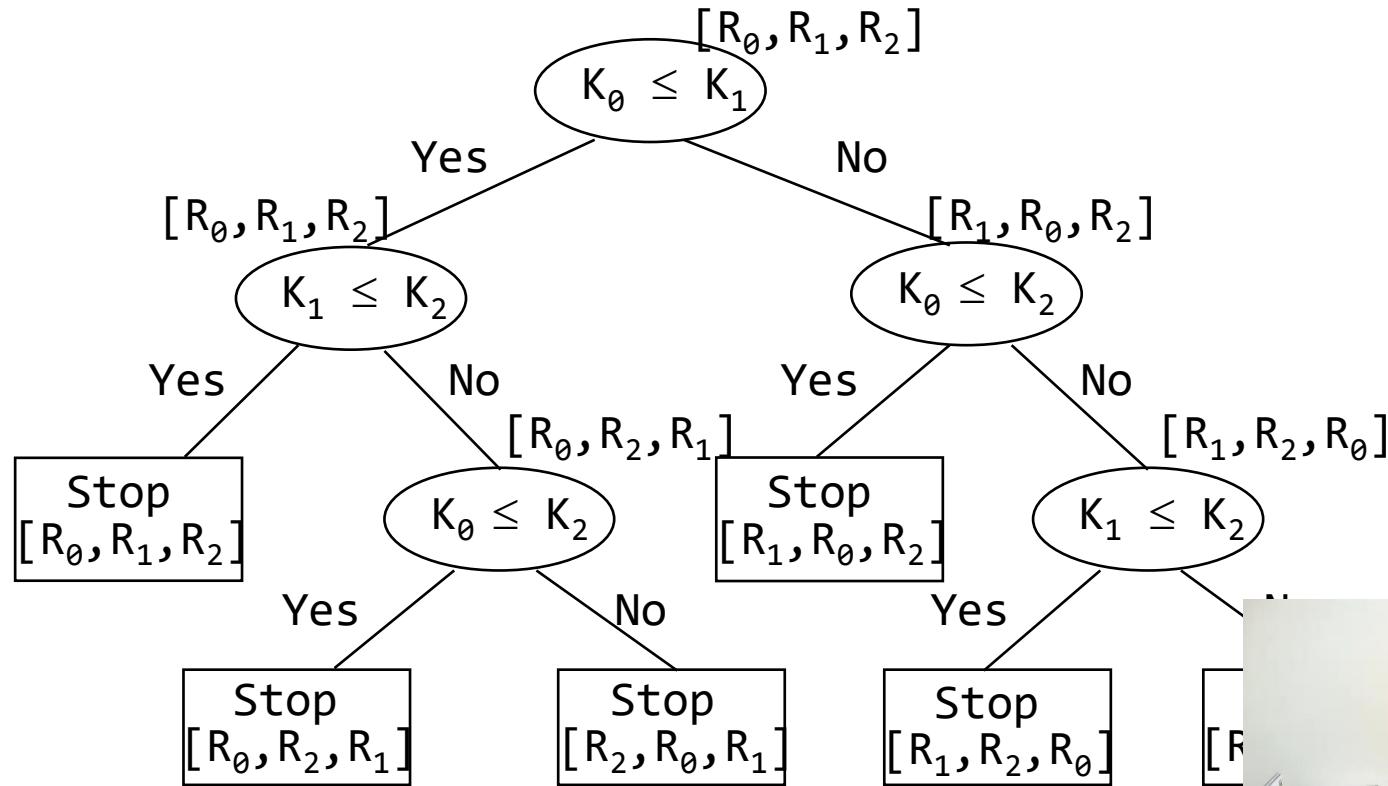
Decision Tree

Describes the sorting process

Each vertex of a tree represents a key comparison

The branches indicate the result

A path through a decision tree represents a sequence of computation that an algorithm could produce



Properties of Decision Tree

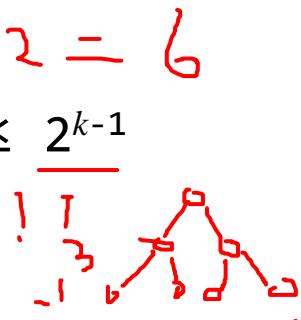
Theorem) Any decision tree that sorts n distinct elements has a height of at least $\log_2(n!) + 1$

→ decision tree of n elements have $n!$ leaves $3 \times 2 = 6$

→ number of leaves of a binary tree of height k $\leq 2^{k-1}$

→ height of the decision tree $\geq \log_2(n!) + 1$

$$\log_2(n!) + 1$$



Corollary) Any algorithm that sorts by comparisons only must have a worst case computing time of $\Omega(n \cdot \log_2 n)$

$$n! = \underline{n \cdot (n-1) \cdot (n-2) \cdots 3 \cdot 2 \cdot 1} \geq \underline{(n/2)^{n/2}}$$

$$\log_2(n!) \geq \underline{(n/2)} \cdot \underline{\log_2(n/2)} = \Omega(n \cdot \log_2 n)$$

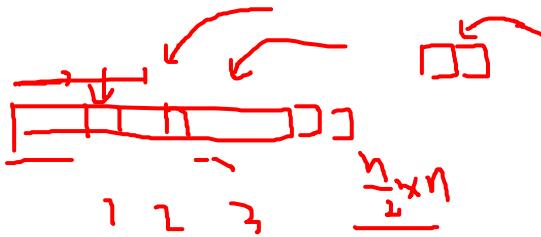
Lower bound of sorting algorithm



Insertion Sort

For each j between 1 and $n-1$, insert $\text{list}[j]$ into already sorted subfile $\text{list}[0], \dots, \text{list}[j-1]$

- Move all keys $> \text{list}[j]$ to the right



- Time complexity

- best case: $O(n)$
- worst case: $O(n^2)$

- It does little work if the list is nearly sorted

Left out of order (L0O)

- R_i is L0O iff $R_i < \max_{0 \leq j < i} \{R_j\}$



- The insertion step is executed only for those records that are L0O

- If number of L0Os = k ,

- Computing time: $O((\underline{n}) \cdot n)$
- Worst case time : $O(\underline{n}^2)$



Insertion Sort Example

- $n = 5$
- input sequence: (26, 5, 37, 1, 19)
- LOOs = R_1, R_2, R_4, R_5

| [0] | [1] | [2] | [3] | [4] | [5] |
|-----|-----|-----|-----|-----|-----|
| | 26 | 5 | 37 | 1 | 19 |

j=2

| | | | | | |
|---|---|----|----|---|----|
| 5 | 5 | 26 | 37 | 1 | 19 |
|---|---|----|----|---|----|

j=3

| | | | | | |
|----|---|----|----|---|----|
| 37 | 5 | 26 | 37 | 1 | 19 |
|----|---|----|----|---|----|

j=4

| | | | | | |
|---|---|---|----|----|----|
| 1 | 1 | 5 | 26 | 37 | 19 |
|---|---|---|----|----|----|

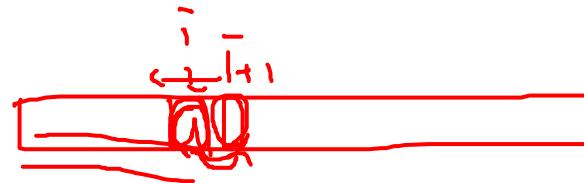
j=5

| | | | | | |
|----|---|---|----|----|----|
| 19 | 1 | 5 | 19 | 26 | 37 |
|----|---|---|----|----|----|



Codes for Insertion Sort

```
void insert(element e, element a[], int i)
{
    a[0] = e;
    while(e.key < a[i].key) {
        a[i+1] = a[i];
        i--;
    }
    a[i+1] = e;
}
```



```
void insertionSort(element a[], int n)
{
    int j;
    for(j = 2; j <= n; j++) {
        element temp = a[j];
        insert(temp, a, j-1);
    }
}
```



Bubble Sort

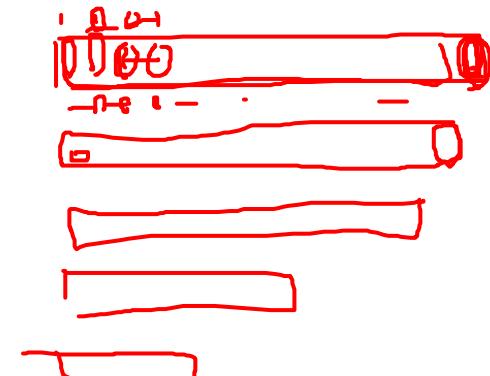
The simplest way sort an array of objects

The basic idea

- To compare two **neighboring** objects
- To swap them if they are in the wrong order.

Codes for bubble sort

```
void bubbleSort (element a[], int n)
{
    element tmp; int i, j;
    for (i=0; i<n-1; i++) {
        for (j=0; j<n-1-i; j++)
            if (a[j+1] < a[j]) {
                tmp = a[j];
                a[j] = a[j+1]; a[j+1] = tmp;
            }
    }
}
```



Example of Bubble Sort

| [0] | [1] | [2] | [3] | [4] | [5] |
|-----|-----|-----|-----|-----|-----|
| 23 | 78 | 45 | 8 | 32 | 36 |

Original list

← unsorted →

| | | | | | | |
|--------------|----|----|---|----|----|----|
| After pass 1 | 23 | 45 | 8 | 32 | 36 | 78 |
|--------------|----|----|---|----|----|----|

← unsorted →

sorted

| | | | | | | |
|--------------|----|---|----|----|----|----|
| After pass 2 | 23 | 8 | 32 | 36 | 45 | 78 |
|--------------|----|---|----|----|----|----|

← unsorted →

sorted

| | | | | | | |
|--------------|---|----|----|----|----|----|
| After pass 3 | 8 | 23 | 32 | 36 | 45 | 78 |
|--------------|---|----|----|----|----|----|

← unsorted →

sorted

| | | | | | | |
|--------------|---|----|----|----|----|----|
| After pass 4 | 8 | 23 | 32 | 36 | 56 | 78 |
|--------------|---|----|----|----|----|----|

← unsorted →

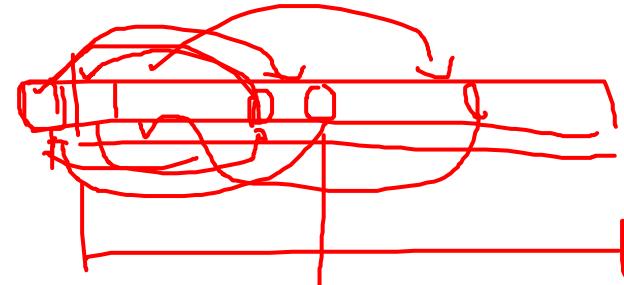
sorted



Selection Sort

Algorithm

```
for (i=0;i<n;i++) {  
    Examine a[i] to a[n-1] and suppose that the smallest  
    integer is at a[min];  
    Interchange a[i] and a[min];  
}
```



Programming in C

```
void sort(int a[], int n){  
    int i,j,min,temp;  
    for (i=0;i<n-1;i++) {  
        min=i;  
        for (j=i+1;j<n;j++) {  
            if (a[j]<a[min]) min=j;  
        }  
        temp=a[i]; a[i]=a[min]; a[min]=temp;  
    }  
}
```



Example of Selection Sort

| | [0] | [1] | [2] | [3] | [4] | [5] |
|-----|-----|-----|-----|-----|-----|-----|
| | 26 | 5 | 37 | 1 | 19 | 24 |
| j=1 | 1 | 5 | 37 | 26 | 19 | 24 |
| j=2 | 1 | 5 | 37 | 26 | 19 | 24 |
| j=3 | 1 | 5 | 19 | 26 | 37 | 24 |
| j=4 | 1 | 5 | 19 | 24 | 37 | 26 |
| j=5 | 1 | 5 | 19 | 24 | 26 | 37 |



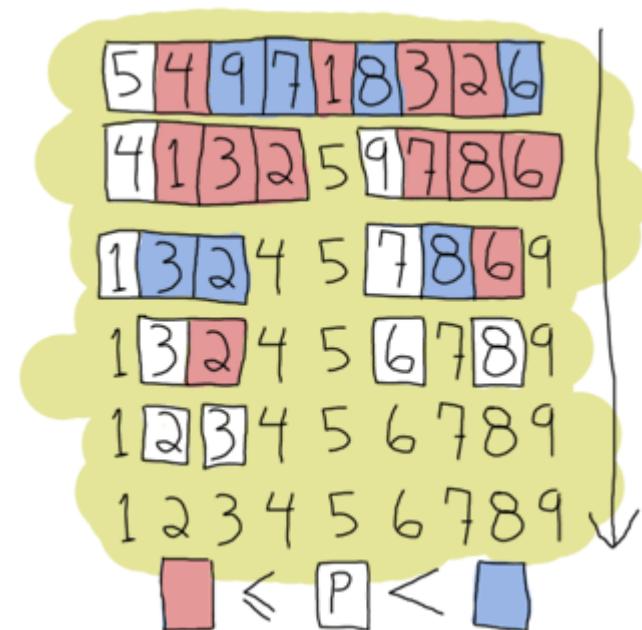
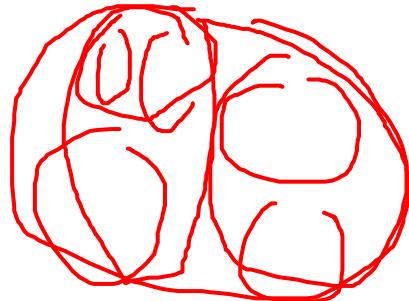
Quick Sort (1)

Divide and conquer

- Two phase
- Split and control

Use recursion : stack is needed

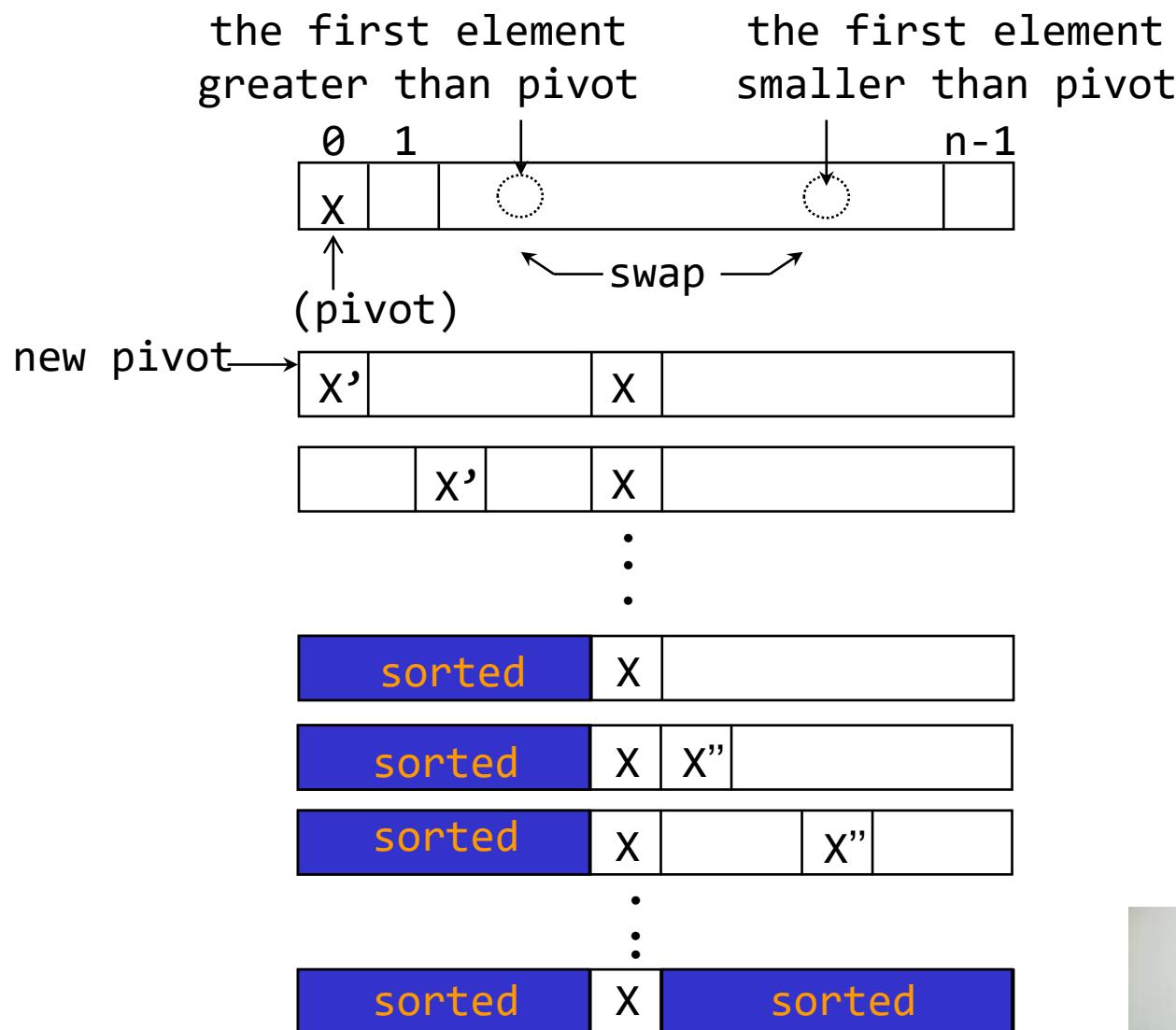
Best average time : $O(n \cdot \log_2 n)$



<http://franzejr.wordpress.com/2011/08/28/a-quick-analysis-about-quicksort/>



Quick Sort (2)



Quick Sort Example

p. 342

| [1] | [2] | [3] | [4] | [5] | [6] | [7] | [8] | [9] | [10] | left | right |
|-----|-----|-----|-----|-----|-----|-----|-----|-----|------|------|-------|
| 26 | 5 | 37 | 1 | 61 | 11 | 59 | 15 | 48 | 19 | 1 | 10 |
| 11 | 5 | 19 | 1 | 15 | 26 | 59 | 61 | 48 | 37 | 1 | 5 |
| 1 | 5 | 11 | 19 | 15 | 26 | 59 | 61 | 48 | 37 | 1 | 2 |
| 1 | 5 | 11 | 19 | 15 | 26 | 59 | 61 | 48 | 37 | 4 | 5 |
| 1 | 5 | 11 | 15 | 19 | 26 | 59 | 61 | 48 | 37 | 7 | 10 |
| 1 | 5 | 11 | 15 | 19 | 26 | 48 | 37 | 59 | 61 | 7 | 8 |
| 1 | 5 | 11 | 15 | 19 | 26 | 37 | 48 | 59 | 61 | 10 | 10 |
| 1 | 5 | 11 | 15 | 19 | 26 | 37 | 48 | 59 | 61 | | |



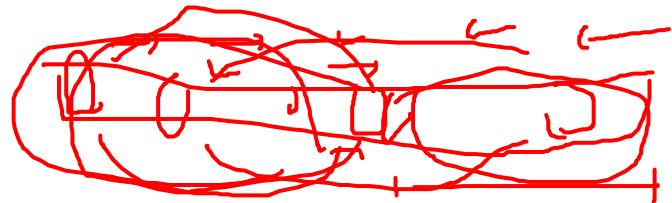
Quick Sort Example – Worst Case

| [1] | [2] | [3] | [4] | [5] | [6] | [7] | [8] | [9] | [10] | left | right |
|-----|-----|-----|-----|-----|-----|-----|-----|-----|------|------|-------|
| 61 | 59 | 48 | 37 | 26 | 19 | 15 | 11 | 5 | 1 | 1 | 10 |
| 1 | 59 | 48 | 37 | 26 | 19 | 15 | 11 | 5 | 61 | 1 | 9 |
| 1 | 59 | 48 | 37 | 26 | 19 | 15 | 11 | 5 | 61 | 2 | 9 |
| 1 | 5 | 48 | 37 | 26 | 19 | 15 | 11 | 59 | 61 | 2 | 8 |
| 1 | 5 | 48 | 37 | 26 | 19 | 15 | 11 | 59 | 61 | 2 | 7 |
| 1 | 5 | 11 | 37 | 26 | 19 | 15 | 48 | 59 | 61 | 2 | 6 |
| 1 | 5 | 11 | 37 | 26 | 19 | 15 | 48 | 59 | 61 | 4 | 7 |
| . | . | . | | | | | | | | | |



Codes for Quick Sort

```
void quicksort (element a[], int left, int right)
{
    int pivot, i, j;    element temp;
    if (left < right) {
        i = left; j = right + 1;
        pivot = a[left].key;
        do {
            do i++; while (a[i].key < pivot); /*&& i<right*/
            do j--; while (a[j].key > pivot); /*&& j>left */
            if (i < j) SWAP(a[i], a[j], temp);
        } while (i < j);
        SWAP (a[left], a[j], temp);
        quicksort(a, left, j - 1);
        quicksort(a, j + 1, right);
    }
}
```



Merge Sort

If $n < 2$ then the array is already sorted. Stop now.

Otherwise, $n > 1$, and perform the following three steps in sequence:

- Sort the left half of the array
- Sort the right half of the array
- Merge the now-sorted left and right halves

Time complexity

- $T(n) = \theta$, if $n < 2$.
- $T(n) = T(n/2) + T(n/2) + \Theta(n)$, if $n > 1$. (similar to quick sort)



Example of Merge Sort

p. 347

| | [0] | [1] | [2] | [3] | [4] | [5] | [6] | [7] | [8] | [9] | length |
|-------|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|--------|
| a | 61 | 59 | 48 | 37 | 26 | 19 | 15 | 11 | 5 | 1 | 1 |
| extra | 59 | 61 | 37 | 48 | 19 | 26 | 11 | 15 | 1 | 5 | 2 |
| a | 37 | 48 | 59 | 61 | 11 | 15 | 19 | 26 | 1 | 5 | 4 |
| extra | 11 | 15 | 19 | 26 | 37 | 48 | 59 | 61 | 1 | 5 | 8 |
| a | 1 | 5 | 11 | 15 | 19 | 26 | 37 | 48 | 59 | 61 | |



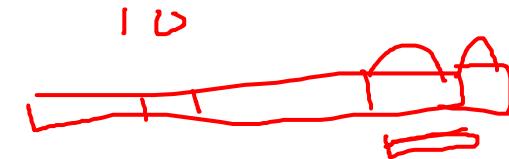
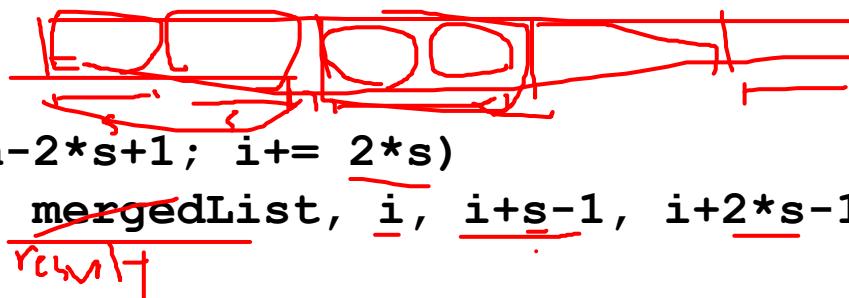
Codes for Merge Sort (1)

```
void merge(element initList[], element mergedList[], int i,  
          int m, int n)  
{  
    int j,k,t;           j = m+1;           k = i;  
    while( i <= m && j <= n ) {  
        if (initList[i].key <= initList[j].key)  
            mergeList[k++] = initList[i++];  
        else  
            mergeList[k++] = initList[j++];  
    }  
    if (i > m) /* mergedList[k:n] = initList[j:n] */  
        for(t = j; t <= n; t++) mergeList[t] =  
initList[t];  
    else /* mergedList[k:n] = initList[i:m] */  
        for(t = i; t <= m; t++)  
            mergeList[k+t-i] = initList[t];  
}
```



Codes for Merge Sort (2)

```
void mergePass(element initList[], element resultList[],
    int n, int s)
{
    for (i = 1; i <= n-2*s+1; i+= 2*s)
        merge(initList, mergedList, i, i+s-1, i+2*s-1);
        return
    if (i+s-1 < n)
        merge(initList, mergedList, i, i+s-1, n);
    else
        for (j=i; j <= n; j++)
            mergedList[j] = initList[j];
}
```



Codes for Merge Sort (3)

```
void mergeSort(element a[], int n)
{
    int s = 1;
    element extra[MAX_SIZE];

    while (s < n) {
        mergePass(a, extra, n, s);
        s *= 2;
        mergePass(extra, a, n, s);
        s *= 2;
    }
}
```



Heap Sort

Utilize the max heap structure

- Average case : $O(n \cdot \log_2 n)$
- Worst case : $O(n \cdot \log_2 n)$

Adjust the binary tree to establish the heap

→ $O(d)$, where d : depth of tree

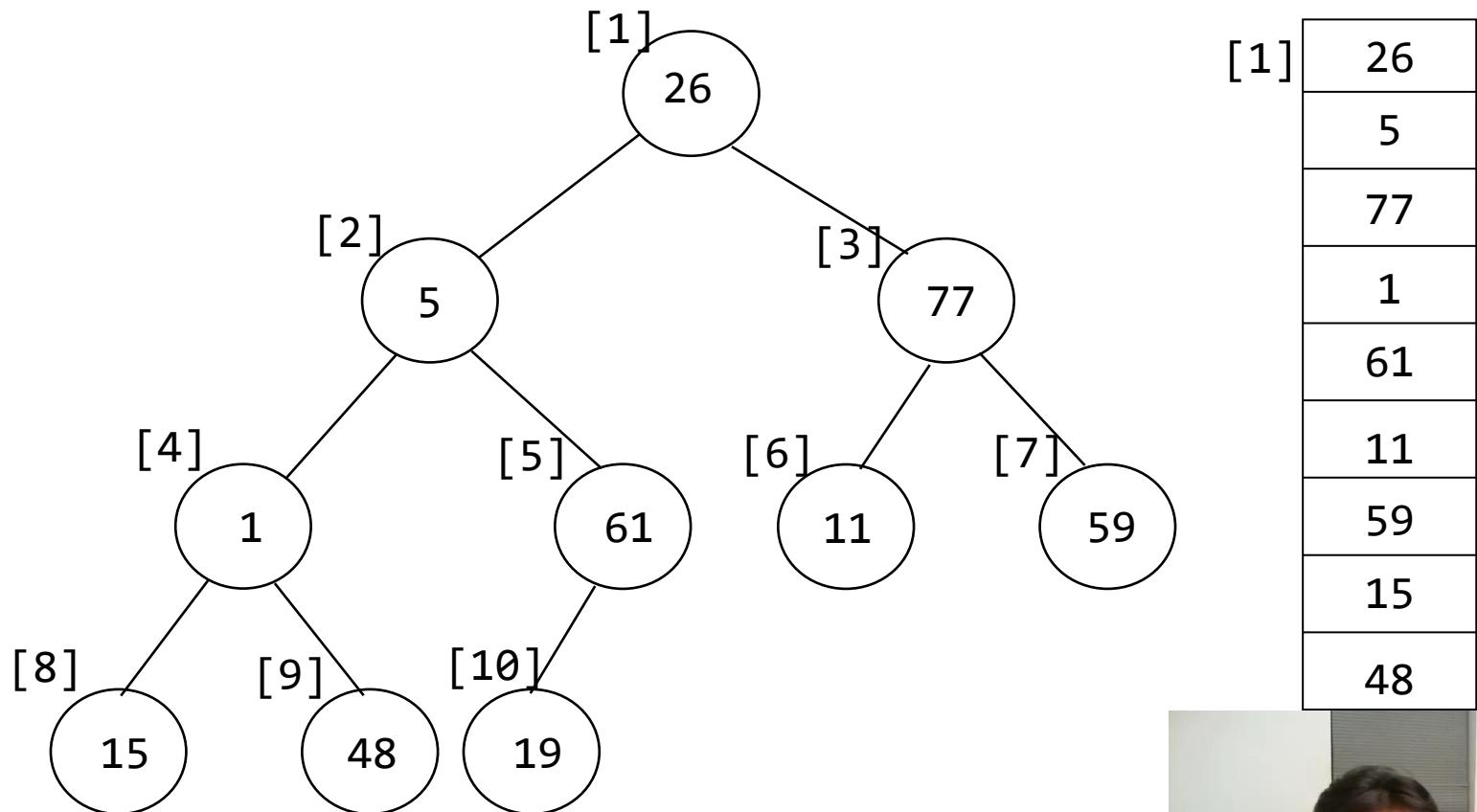


Example of Heap Sort (1)

Input list : (26, 5, 77, 1, 61, 11, 59, 15, 48, 19)

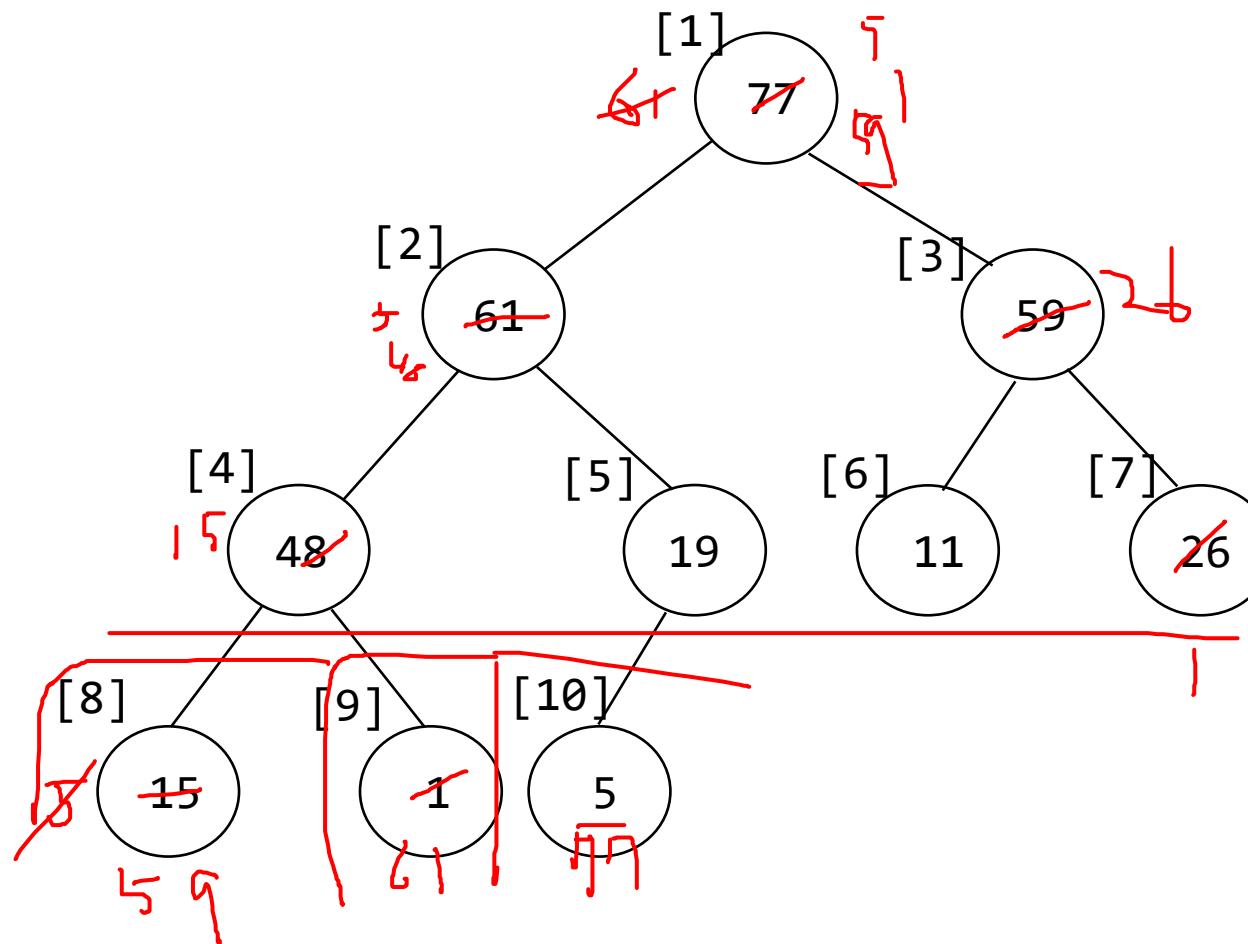
Output list: (1, 5, 11, 15, 19, 26, 48, 59, 61, 77)

Initial state



Example of Heap Sort (2)

Step 1: Build a max heap



| | |
|------|----|
| [1] | 77 |
| [2] | 61 |
| [3] | 59 |
| [4] | 48 |
| [5] | 19 |
| [6] | 11 |
| [7] | 26 |
| [8] | 15 |
| [9] | 1 |
| [10] | 5 |



Example of Heap Sort (5)

Last step

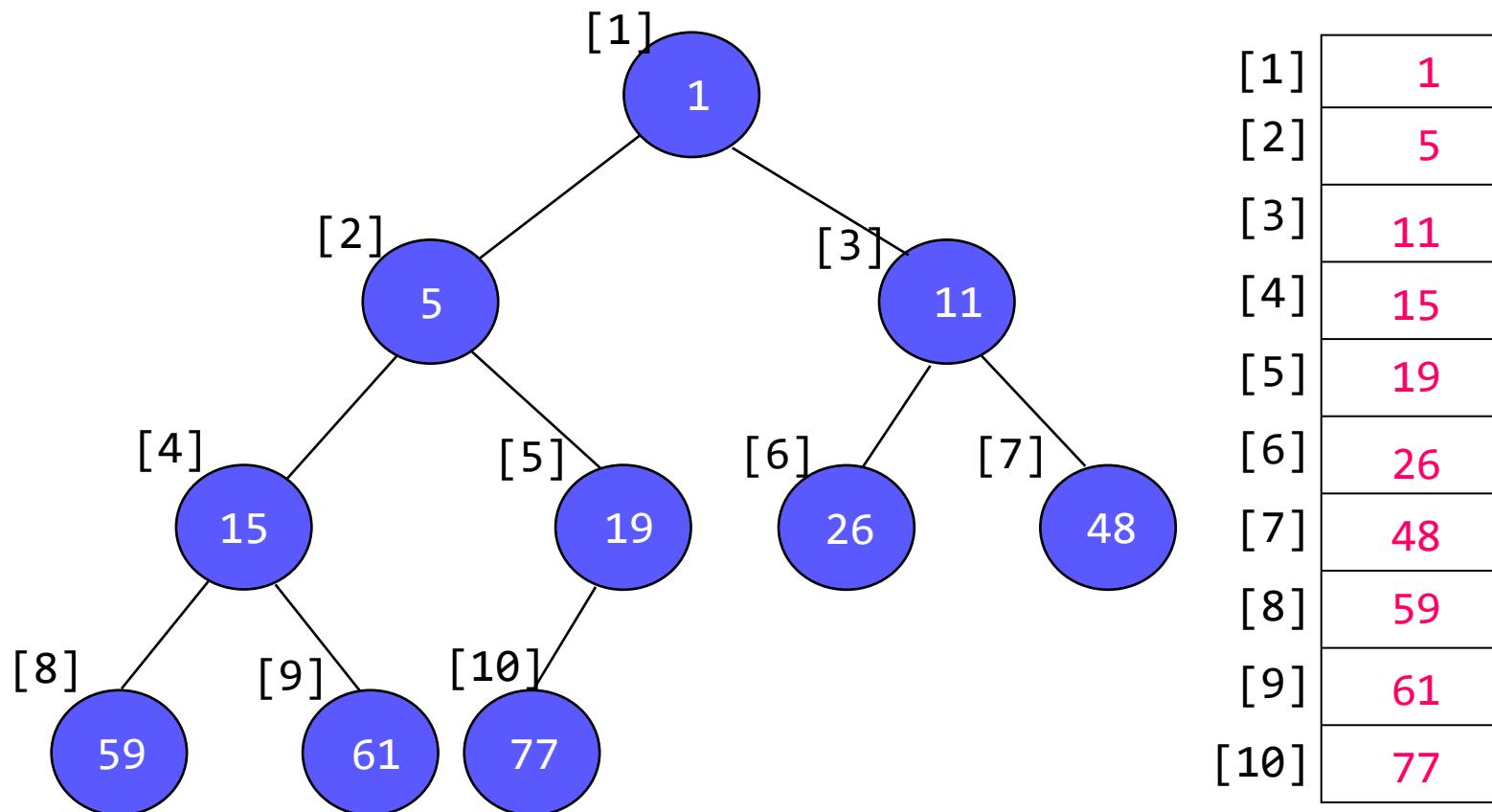


Figure 7.8 in p.355~



Codes for Heap Sort (1)

```
void adjust(element list[], int root, int n)
{  int child, rootkey;
   element temp;
   temp = list[root];
   rootkey = list[root].key;
   child = 2 * root; /* left child */
   while(child <= n) {
      if (child < n && list[child].key < list[child+1].key)
         child++;
      if (rootkey > list[child].key)
         break;
      else {
         list[child/2] = list[child];
         child *= 2;
      } /* else */
   } /* while */
   list[child/2] = temp;
}
```

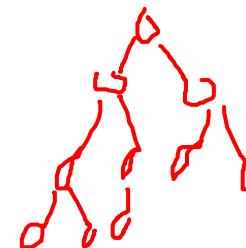


Time Complexity: $\lfloor \log_2 n \rfloor$



Codes for Heap Sort (2)

```
void heapsort(element list[], int n)
{ /* perform a heapsort on the array */
    int i, j;
    element temp;
    for (i = n/2; i > 0; i--)
        adjust(list, i, n); /* initial heap construction */
    for (i = n - 1; i > 0; i--) /* heap adjust */
    {
        SWAP(list[1], list[i+1], temp);
        adjust(list, 1, i);
    }
}
```

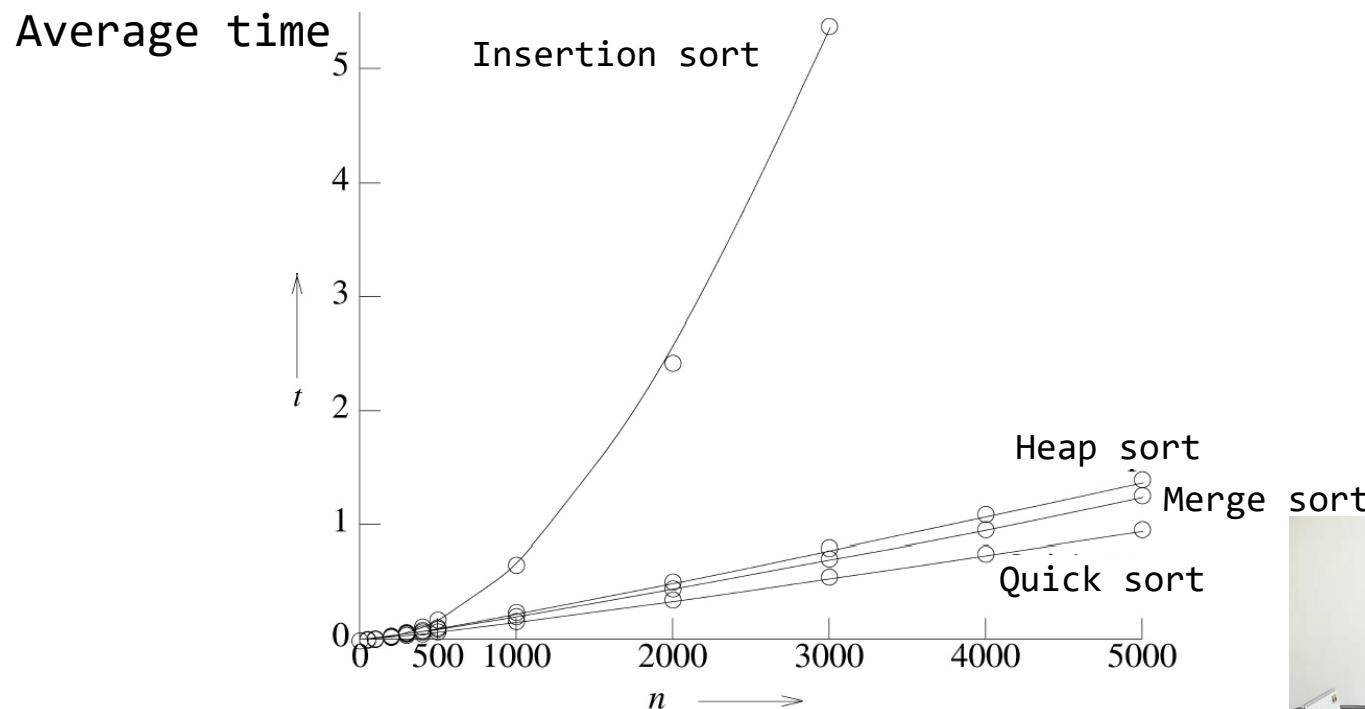


Time Complexity: $\lfloor \log_2 n \rfloor + \lfloor \log_2(n-1) \rfloor + \dots + \lfloor \log_2 2 \rfloor$
 $= O(n \cdot \log_2 n)$



Summary of Internal Sorting

| Method | Worst | Average |
|----------------|--------------------|--------------------|
| Insertion sort | n^2 | n^2 |
| Heap sort | $n \cdot \log_2 n$ | $n \cdot \log_2 n$ |
| Merge sort | $n \cdot \log_2 n$ | $n \cdot \log_2 n$ |
| Quick sort | n^2 | $n \cdot \log_2 n$ |



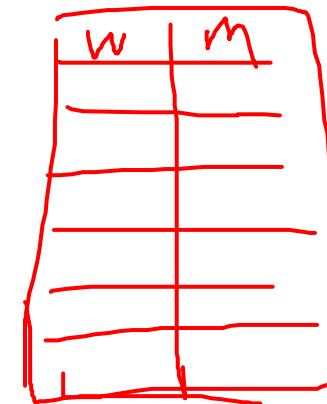
Hashing



Symbol Table

Symbol table

- A set of name-attribute pairs
- Example
 - Thesaurus
 - Compiler symbol table
 - Dictionary
- Operations
 - Determine if a particular name is in the table
 - Retrieve the attributes of that name
 - Modify the attributes of the name
 - Insert a new name and its attributes



Hashing

A technique to insert, search, and delete symbols efficiently



Static Hashing

Static hashing

- Identifiers are stored in a fixed size table called hash table

Hashing table ht

- b buckets and s slots
- $ht[0], \dots, ht[b-1]$

Hash function $h(x)$

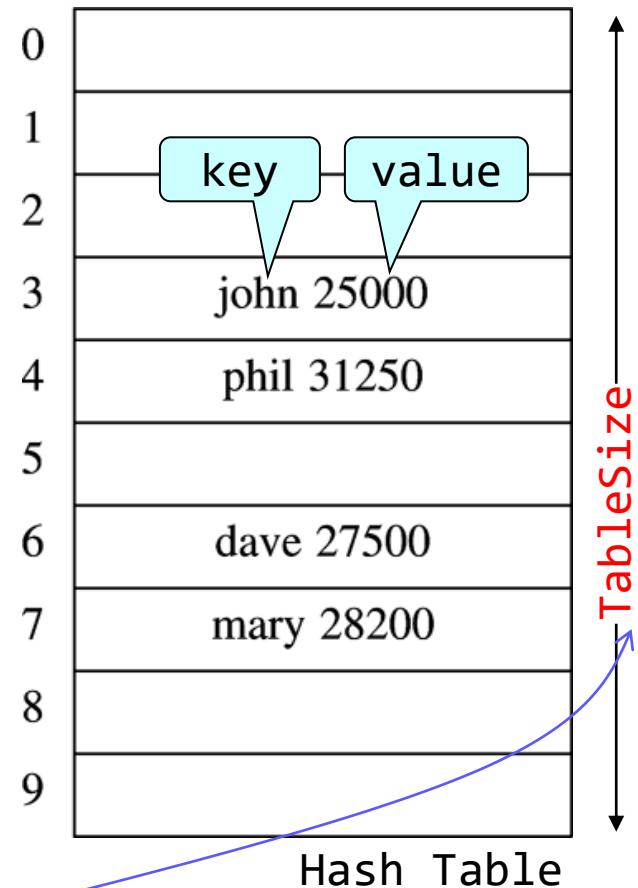
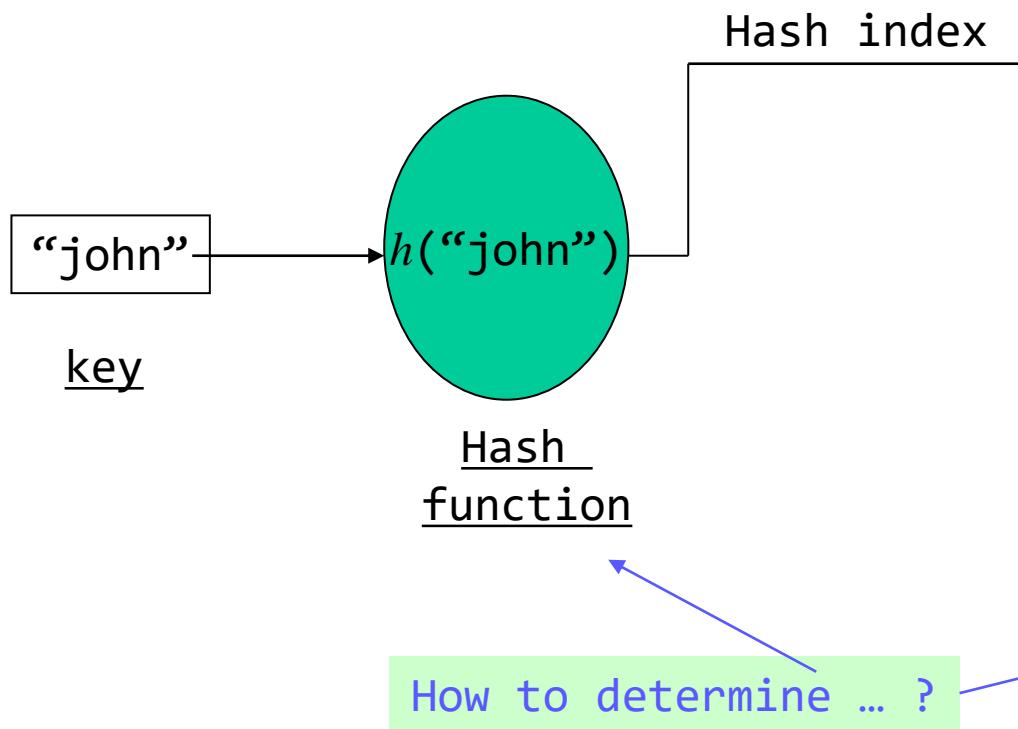
identifier x \rightarrow address in the hash table

| | | |
|-----|---------------|-------------|
| [0] | <i>bucket</i> | |
| [1] | <i>slot</i> | <i>slot</i> |
| [2] | | |
| [3] | | |
| [4] | | |
| [5] | | |
| [6] | | |

Worst-case time for Search, Insert, and Delete is $O(b \times s)$.
Expected time is $O(1)$.



Hash Table



Terminology

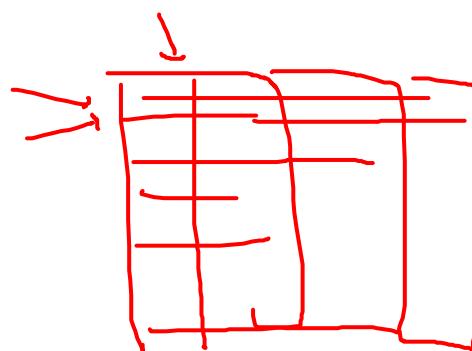
Identifier density is the ratio n/T , where n is the number of identifiers in the table, T is number of possible identifiers

Loading factor $\alpha = n / (s \cdot b)$

Two identifiers i_1 and i_2 are **synonyms** with respect to h if $h(i_1) = h(i_2)$

An **overflow** occurs when we hash a new identifier i into a full bucket

A **collision** occurs when we hash two non-identical identifiers into same bucket



Example of Hashing

Hash table

$b = 26$, $s = 2$, $n = 10$

Loading factor = $10 / 52$

Id

“acos”, “define”, “float”, “exp”, “char”, “atan”, “ceil”,
“floor”, “clock”, “ctime”

Hash function $h(x)$: the first letter of x -’a’

$h(\text{acos})=0$ $h(\text{define})=3$ $h(\text{float})=5$ $h(\text{exp})=4$

$h(\text{char})=2$ $h(\text{atan})=0$ $h(\text{ceil})=2$ $h(\text{floor})=5$

$h(\text{clock})=2$ $h(\text{ctime})=2$



Hashing Functions

Requirements for Hashing function h

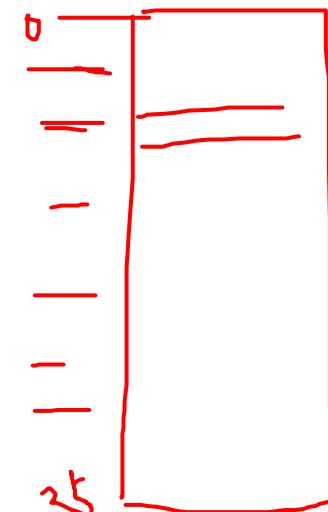
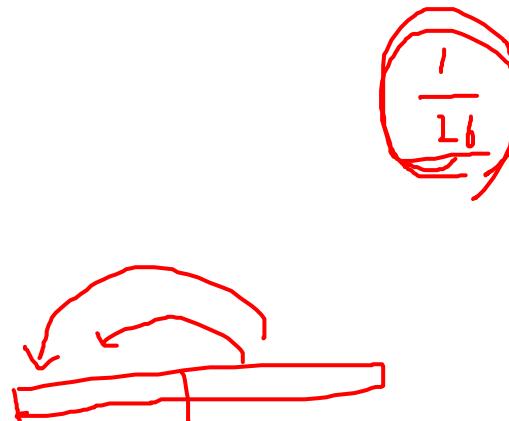
- Easy to compute
- Minimize the number of collisions
- Unbiased

Uniform hashing function

- Probability of $h(x) = i$ is $1/b$ for all i

Examples

- Mid-square
- Division
- Folding
- Digit analysis



Mid-Square

Hash function, h , computed by squaring the identifier
Using appropriate number of bits from the middle of the square to obtain the bucket address

- Take r bits if the table size is 2^r
- Middle bits of a square usually depend on all the characters
- It is expected that different keys will yield different addresses with high probability, even if some of the characters are the same

Table size=16 $r=4$

Key=4562 $(4562 * 4562) \% 65536 = 36932 = 1001000001000100$
 Hash position = 1

Key=3981 $(3981 * 3981) \% 65536 = 54185 = 1101001110101001$
 Hash position = 14

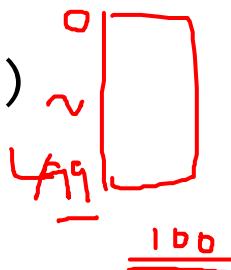


Division

Using modulus (%) operator

- $h_D(x) = x \% M$

- Bucket : $0 \sim (M-1)$



$$M=8=2^3 \quad x=13=01\boxed{1}01_{(2)} \quad h(x)=5$$
$$x=17=10001_{(2)} \quad h(x)=1$$

Choice of M

- $M = 2^p$, $h_D(x)$ is just low order p bits
- If $M = 2^p - 1$ and x is interpreted in radix 2^p , permuting the characters of x does not change hash value
- A prime not too close to an exact power of 2 is often a good choice of M

13

$$M=\underline{7}=2^3-1 \quad \text{String "abcd"}$$

$$x = 97 \cdot 8^3 + 98 \cdot 8^2 + 99 \cdot 8 + 100 = 56828 \quad h(x)=2$$

String "badc"

$$x = 98 \cdot 8^3 + 97 \cdot 8^2 + 100 \cdot 8 + 99 = 57232 \quad h(x)=2$$



Division

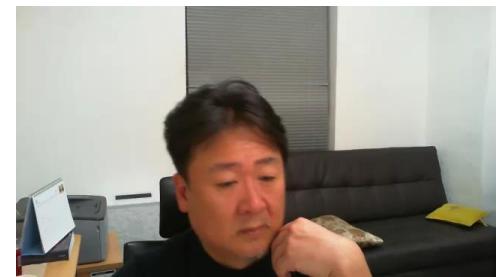
- If divisor is an even number, odd integers hash into odd home buckets and even integers into even home buckets.
- If divisor is an odd number, odd (even) integers may hash into any home.

M=14, $20\%14 = 6$, $30\%14 = 2$, $8\%14 = 8$

$15\%14 = 1$, $3\%14 = 3$, $23\%14 = 9$

M=15, $20\%15 = 5$, $30\%15 = 0$, $8\%15 = 8$

$15\%15 = 0$, $3\%15 = 3$, $23\%15 = 8$



Overflow Handling

Open addressing

- ① Linear probing
- ② Random probing
- ③ Quadratic probing
- ④ Rehashing

Chaining

Maintain a linked list of synonyms for each bucket



Linear Probing

Idea:

Find the closest unfilled bucket
when overflow occur

Hash function

$h(x)$: the first character of x -'a'

$b=26$, $s=1$

Id: "acos", "atan", "char",
"define", "exp", "ceil", "cos",
"float", "atol", "floor", "ctime"

$h(\text{acos})=0$ $h(\text{atan})=0$ $h(\text{char})=2$

$h(\text{define})=3$ $h(\text{exp})=4$ $h(\text{ceil})=2$

$h(\text{cos})=2$ $h(\text{float})=5$ $h(\text{atol})=0$

$h(\text{floor})=5$ $h(\text{ctime})=2$

Search atol?

Search ascii?

| bucket | x | buckets searched |
|--------|--------|------------------|
| 0 | acos | 1 |
| 1 | atan | 2 |
| 2 | char | 1 |
| 3 | define | 1 |
| 4 | exp | 1 |
| 5 | ceil | 4 |
| 6 | cos | 5 |
| 7 | float | 3 |
| 8 | atol | 9 |
| 9 | floor | 5 |
| 10 | ctime | 9 |
| 11 | | |
| ... | | |
| 25 | | |

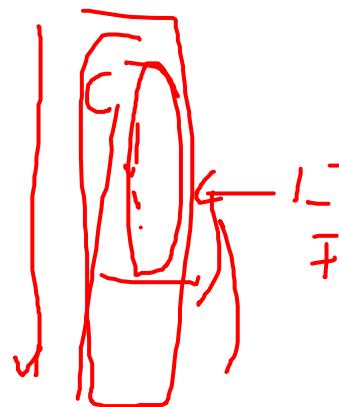


Linear Probing: Issues

Probe sequences can get longer with time

Primary clustering

- Keys tend to cluster in one part of table
- Keys that hash into cluster will be added to the end of the cluster (making it even bigger)
- Side effect: Other keys could also get affected if mapping to a crowded neighborhood



Random Probing

Uses a random number generator to find the next available slot

*i*th slot in the probe sequence is: $(h(X) + r_i) \% b$,
where r_i is the *i*th value in a random permutation of the
numbers 1 to $b - 1$

All insertions and searches use the same sequence of
random numbers



Example of Random Probing

Hash function

$h(x)$: the first character of x -'a'

random number sequence :

2 4 6 1 7 9 13

b=26, s=1

Id: "acos", "atan", "char", "define",
"exp", "ceil", "cos", "float",
"atol", "floor", "ctime"

$h(\text{acos})=0 \quad h(\text{atan})=0 \quad h(\text{char})=2$

$h(\text{define})=3 \quad h(\text{exp})=4 \quad h(\text{ceil})=2$

$h(\text{cos})=2 \quad h(\text{float})=5 \quad h(\text{atol})=0$

$h(\text{floor})=5 \quad h(\text{ctime})=2$

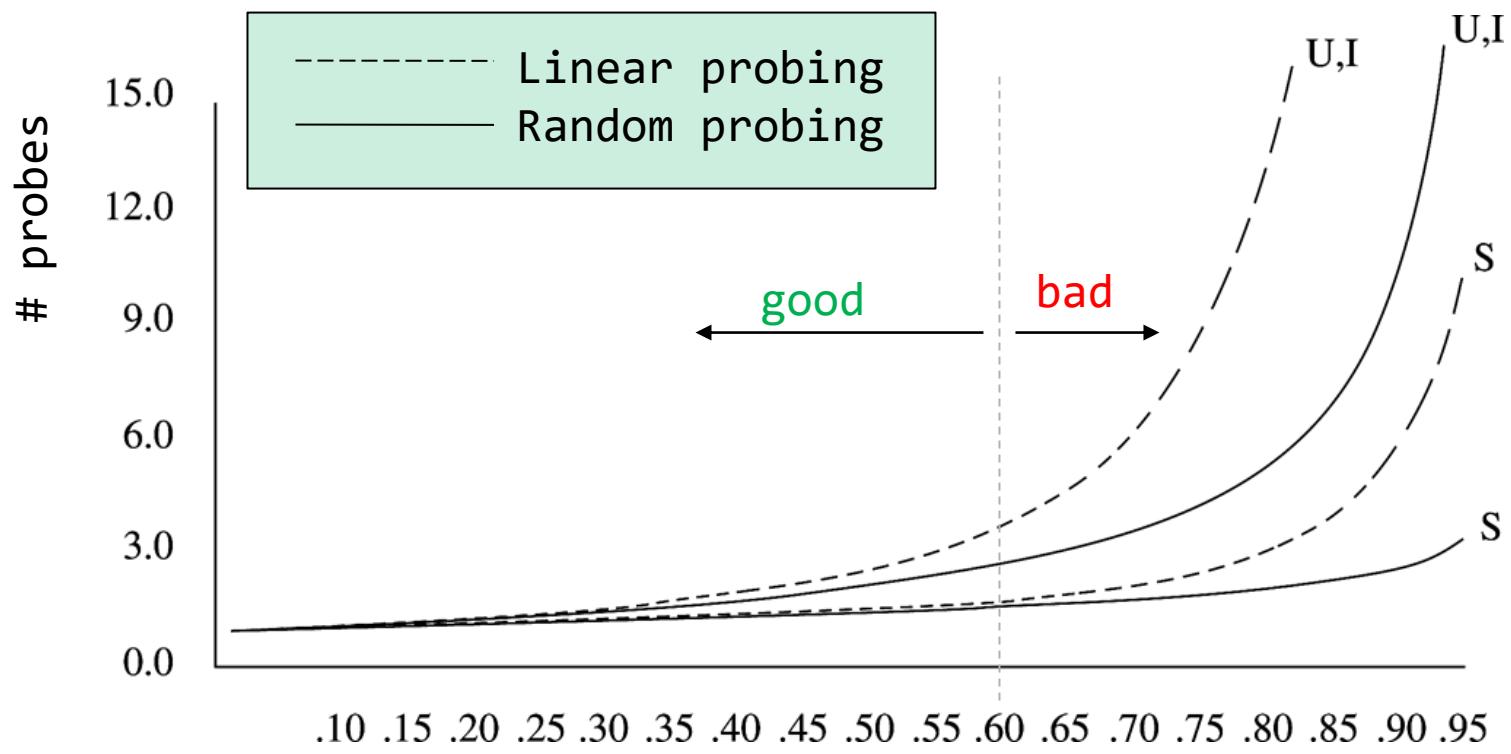
Search atol?

Search ascii?

| bucket | x | buckets searched |
|--------|--------|------------------|
| 0 | acos | 1 |
| 1 | atol | 5 |
| 2 | atan | 2 |
| 3 | define | 1 |
| 4 | char | 2 |
| 5 | float | 1 |
| 6 | exp | 2 |
| 7 | floor | 2 |
| 8 | ceil | 4 |
| 9 | cos | 6 |
| 10 | | |
| 11 | ctime | 7 |
| ... | | |
| 25 | | |



Linear vs. Random Probing



U - unsuccessful search

S - successful search

I - insert

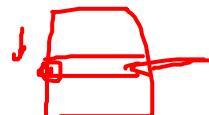
Load factor λ



Deletion in Open Addressing

Two considerations

- Deleting a record must not hinder later searches
- We do not want to make positions in the hash table unusable because of deletion



→ placing a special mark in place of the deleted record, called a **tombstone**

If a tombstone is encountered when searching, the search procedure continues

When a tombstone is encountered during insertion, that slot can be used to store the new record



Chaining

Maintain a linked list of synonyms for each bucket →
s is flexible

Hash function

$h(x)$: the first character of x - 'a'

Id: "acos", "define", "float",
"exp", "char", "atan", "ceil",
"floor", "clock", "ctime"

$h(\text{acos}) = h(\text{atan}) = 0$, $h(\text{define}) = 3$,
 $h(\text{exp}) = 4$,
 $h(\text{char}) = h(\text{ceil}) = h(\text{clock}) = h(\text{ctime})$
 $= 2$, $h(\text{float}) = h(\text{floor}) = 5$

